А.Ахо, Дж.Хопкрофт, Дж.Ульман

ПОСТРОЕНИЕ И АНАЛИЗ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ

М.: Мир, 1979, 536 стр.

В монографии с единых позиций излагаются результаты теоретических и прикладных исследований по построению быстрых алгоритмов и доказательству их отсутствия. Рассмотрены задачи перебора, упорядочения массивов данных, умножения чисел, умножения матриц, обсуждаются алгоритмы на графах. Многие результаты ранее были рассеяны в труднодоступных источниках и в монографическом виде публикуются впервые.

Книга рассчитана на специалистов по современному программированию, разработчиков вычислительных систем и алгоритмов, она может быть использована как учебное пособие студентами и аспирантами, специализирующимися в области вычислительной математики.

Содержание

ПРЕДИСЛОВИЕ К РУССКОМУ ПЕРЕВОДУ 5
ПРЕДИСЛОВИЕ 7
1. МОДЕЛИ ВЫЧИСЛЕНИЙ 11
1.1. Алгоритмы и их сложности 11
1.2. Машины с произвольным доступом к памяти 15
1.3. Вычислительная сложность РАМ-программ 22
1.4. Модель с хранимой программой 26
1.5. Модификация РАМ 32
1.6. Простейшая модель вычислений: машина Тьюринга 39
1.7. Связь машин Тьюринга и РАМ 45
1.8. Язык высокого уровня—Упрощенный Алгол 47
Упражнения 54
Замечания по литературе 56
2. РАЗРАБОТКА ЭФФЕКТИВНЫХ АЛГОРИТМОВ 57
2.1. Структуры данных: списки, очереди и стеки 58
2.2. Представления множеств 63
2.3. Графы 64
2.4. Деревья 67
2.5. Рекурсия 70
2.6. Разделяй и властвуй 75
2.7. Балансировка 81
2.8. Динамическое программирование 83
2.9. Эпилог 85
Упражнения 86
Замечания по литературе 92
3. СОРТИРОВКА И ПОРЯДКОВЫЕ СТАТИСТИКИ 93
3.1. Задача сортировки 94
3.2. Цифровая сортировка 95
3.3. Сортировка с помощью сравнений 104
3.4. Сортдеревом—упорядочение с помощью $O(n \log n)$ сравнений 106

3.5. Быстрсорт—упорядочение за среднее время $O(n \log n)$ 111
3.6. Порядковые статистики 117
3.7. Среднее время для порядковых статистик 119
Упражнения 122
Замечания по литературе 127
4. СТРУКТУРЫ ДАННЫХ ДЛЯ ЗАДАЧ, КАСАЮЩИХСЯ РАБОТЫ С МНОЖЕСТВАМИ **128**
4.1. Основные операции над множествами 128
4.2. Метод расстановки 132
4.3. Двоичный поиск 135
4.4. Деревья двоичного поиска 136
4.5. Оптимальные деревья двоичного поиска 141
4.6. Простой алгоритм для нахождения объединения непересекающихся множеств 146
4.7. Древовидные структуры для задачи ОБЪЕДИНИТЬ—НАЙТИ 150
4.8. Приложения и обобщения алгоритма ОБЪЕДИНИТЬ—НАЙТИ 162
4.9. Схемы сбалансированных деревьев 168
4.10. Словари и очереди с приоритетами 171
4.11. Сливаемые деревья 175
4.12. Сцепляемые очереди 178
4.13. Разбиение 181
4.14. Резюме 188
Упражнения 188
Замечания по литературе 195
5. АЛГОРИТМЫ НА ГРАФАХ **197**
5.1. Остовное дерево наименьшей стоимости 197
5.2. Метод поиска в глубину 202
5.3. Двусвязность 206
5.4. Поиск в глубину в ориентированном графе 214
5.5. Сильная связность 216
5.6. Задачи нахождения путей 223
5.7. Алгоритм транзитивного замыкания 227
5.8. Алгоритм нахождения кратчайшего пути 229
5.9. Задачи о путях и умножение матриц 230
5.10. Задачи с одним источником 235
5.11. Доминаторы в ориентированных ациклических графах: комбинирование понятий 238
Упражнения 247
Замечания по литературе 254
6. УМНОЖЕНИЕ МАТРИЦ И СВЯЗАННЫЕ С НИМ ОПЕРАЦИИ **255**
6.1. Основные понятия 255
6.2. Алгоритм Штрассена для умножения матриц 259
6.3. Обращение матриц 262
6.4. НВП-разложение матрицы 263

6.5. Приложения НВП-разложения 272
6.6. Умножение булевых матриц 274
Упражнения 279
Замечания по литературе 283
7. БЫСТРОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ И ЕГО ПРИЛОЖЕНИЯ 284
7.1. Дискретное преобразование Фурье и обратное к нему 284
7.2. Алгоритм быстрого преобразования Фурье 290
7.3. БПФ при использовании битовых операций 298
7.4. Произведение полиномов 303
7.5. Алгоритм Шёнхаге — Штрассена для умножения целых чисел 304
Упражнения 308
Замечания по литературе 310
8. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ ЧИСЛАМИ И ПОЛИНОМАМИ 311
8.1. Аналогии между целыми числами и полиномами 312
8.2. Умножение и деление целых чисел 313
8.3. Умножение и деление полиномов 320
8.4. Модульная арифметика 323
8.5. Модульная арифметика полиномов и вычисление их значений 327
8.6. Применение китайской теоремы об остатках 329
8.7. Китайская теорема об остатках и интерполяция полиномов 333
8.8. Наибольшие общие делителя и алгоритм Евклида 336
8.9. Асимптотически быстрый алгоритм нахождения НОД полиномов 339
8.10. НОД целых чисел 345
8.11. Еще раз о применении китайской теоремы об остатках 347
8.12. Разреженные полиномы 348
Упражнения 350
Замечания по литературе 353
9. АЛГОРИТМЫ ИДЕНТИФИКАЦИИ 354
9.1. Конечные автоматы и регулярные выражения 354
9.2. Распознавание образов, задаваемых регулярными выражениями 363
9.3. Распознавание подцепочек 367
9.4. Двусторонний детерминированный магазинный автомат 373
9.5. Позиционные деревья и идентификаторы позиций 385
Упражнения 398
Замечания по литературе 403
10. NP-ПОЛНЫЕ ЗАДАЧИ 404
10.1. Недетерминированные машины Тьюринга 405
10.2. Классы P и NP 414
10.3. Языки и задачи 417
10.4. NP-полнота задачи выполнимости 420
10.5. Еще несколько NP-полных задач 428
10.6. Задачи с полиномиально ограниченной памятью 440
Упражнения 446

Замечания по литературе 450
11. НЕКОТОРЫЕ ДОКАЗУЕМО ТРУДНО РАЗРЕШИМЫЕ ЗАДАЧИ 451
11.1. Иерархии по сложности 451
11.2. Иерархия по емкостной сложности для детерминированных машин Тьюринга 452
11.3. Задача, требующая экспоненциальных времени и памяти 456
11.4. Неэлементарная задача 466
Упражнения 471
Замечания по литературе 474
12. НИЖНИЕ ОЦЕНКИ ЧИСЛА АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ 475
12.1. Поля 475
12.2. Еще раз о неветвящихся программах 477
12.3. Матричная формулировка задач 479
12.4. Нижняя граница для числа умножений, связанная с рангом по строкам 480
12.5. Нижняя граница для числа умножений, связанная с рангом по столбцам 483
12.6. Граница для числа умножений, связанная с рассмотрением строк и столбцов 488
12.7. Предварительная обработка 490
Упражнения 493
Замечания по литературе 501
СПИСОК ЛИТЕРАТУРЫ 502
ГЛОССАРИИ 514
ИМЕННОЙ УКАЗАТЕЛЬ 516
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 519

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Автомат (automaton) конечный (finite) 165, 361, 365
- - детерминированный (deterministic) 165, 361
- - недетерминированный (nondeterministic) 356
- линейно ограниченный (linear bounded) 449
- магазинный (pushdown) 373
- - двусторонний (two-way) 373, 400
- - - детерминированный (deterministic) 373—375
- - детерминированный (deterministic) 373, 400, 401
- - недетерминированный (nondeterministic) 400
- - односторонний (one-way) 400
Адрес (address) 16
- возврата (return) 73
- значения (value) 73
- символический (symbolic) 33
Адресация (addressing) — косвенная (indirect) 16
Активный (active) 484, 490
Алгол (ALGOL) — Упрощенный (Pidgin) 48
Алгоритм (algorithm) — Дейкстры (Dijkstra's) 236
- Евклида (Euclid's) 336
- - расширенный (extended) 336, 337
- Крускала (Kruskal's) 199
- префиксный (on-line) 129
- свободный (off-line) 129

- с предварительной обработкой данных (preconditioned) 329
- четырех русских (four Russians') 275, 277
- Шёнхаге — Штрассена (Schonhage — Strassen) 304, 306
- Штрассена (Strassen-'s) 259
Алфавит (alphabet) 19, 355
- входной (input) 165, 356, 375
- магазинный (pushdown list) 374, 375
Аннулятор (anihilator) 224
Антисимметричность (antisymmetry) 94
Аргумент (argument) см. Параметр, 51
База данных (data base) 132, 147, 188
Баланс узла (balance of a vertex) 194
Балансировка (balancing) 81
БЛОК (INBLOCK) 185
Блок (block) 51
БПФ (FFT) см. Преобразование Фурье, быстрое 294
Брат (brother) 174
Быстрсорт (quicksort) 111, 113
БЫСТРСОРТ (QUICKSORT) 113
ВЕРШИНА (TOP) 61
Вес дерева (weight of a tree) 142
ВЗИМОЗАМЕНА (INTERCHANGE) 52
ВНЕШ-ИМЯ (EXTERNAL_NAME) 147
ВНУТПОРЯДОК (INORDER) 71
ВНУТР_ИМЯ (INTERNAL-NAME) 147
В-ОЖИДАНИИ (INWAITING) 186
ВПИСАТЬ (ENQUEUE) 62, 194
ВРЕМ (TEMP) 380
ВСТАВИТЬ (INSERT) 59, 128
Вход (input) 34
- вычисления (of computation) 477
ВЫБОР (SELECT) 118, 122
Вызов (call) — по значению (by value) 52
- - наименованию (by name) 52
- - ссылке (by reference) 52
ВЫПИСАТЬ (DEQUEUE) 62, 194
Выполнимость (satisfiability) 419
Выражение (expression)
- регулярное (regular) 355
- - полу расширенное (semiextended) 457
- - расширенное (extended) 456
Высота (height) — дерева (of a tree) 68
- узла (of a vertex) 68
ВЫТОЛКНУТЬ (POP) 61
Выход (output) 34
Выход (в графе) (output vertex) 497
Вычисление (computation) — битовое (bitwise) 35
- двойственное (dual) 496
- линейное (linear) 497
- машины с данным измерителем, правильное (of a machine with a given yardstick, valid) 462
- относительно поля (with respect to a field) 477
Вычисление (значения) полинома (evaluation of a polynomial) 34, 286, 328, 476, 487, 490—491, 499—500
Глубина (depth) — средняя (expected) 111
- узла (of a vertex) 68
Головка (head) 40
Гомоморфизм (homomorphism) 461
- сохраняющий длину (length-preserving) 461, 520
Грамматика (grammar) — бесконтекстная (context-free) 91, 401
- - в нормальной форме Хомского (in Chomski normal form) 91
- контекстная (context-sensitive) 449
Граф (graph) 64
- ациклический (acyclic) 67
- двусвязный (biconnected) 206
- дополнительный (complement) 431

- корневой (rooted) 239
- неориентированный (undirected) 64
- ориентированный (directed) 64
- переходов см. Диаграмма (переходов) 357
- раскрашиваемый (colorable) 421
- связный (connected) 70, 216, 253
- сильно связный (strongly connected) 216
ДАННЫЕ (DATA) 99
Дважды связанный (doubly linked) 61
Двойственное (вычисление) (dual) 496
Двусвязность (biconnectivity) 206
ДЕЛЕНИЕ (DIVIDE) 179, 180
Деление (division) — полиномов (of polynomials) 320
- целых чисел (of integers) 313
Дерево (tree) 67
- 2, 3, 169
- I 386
- S 386
- АВЛ (AVL) 193
- бинарное см. Дерево двоичное 68
- вспомогательное (auxiliary) 390
- двоичное (binary) 68
- - полное (complete) 68
- двоичного поиска (binary search) 136
- доминаторное (dominator) 239
- корневое (rooted) 67
- - неориентированное (undirected) 70
- неориентированное (undirected) 70
- ориентированное (directed) 67
- остовное (spanning) 130
- - глубинное (depth-first) 203
- позиций см. Дерево позиционное 296, 387
- позиционное (position) 296, 387
- - уплотненное (compact) 397
- помеченное (labeled) 103
- решений (decision) 37
- сбалансированное (balanced) 169, 194
- - ограниченное (bounded) 194
- сливаемое см. Сливаемое дерево 170
- сортирующее см. Сортирующее дерево 106
- упорядоченное (ordered) 68
Диаграмма (переходов) (diagram) 357
Диагонализировать (diagonalize) 453
Диагональ главная (main diagonal) 259
ДКА (DFA) 361
- скелетный (skeletal) 367
ДЛИНА (LENGTH) 99
Длина (length)
- внешних путей (external path) 194
- внутренних путей (internal path) 194
- пути (of a path) 64
- регулярного выражения (of a regular expression) 359
- цепочки (of a string) 355
1дма, 1dpda, 401
2дма, 2dpda, 373
- k-головчатый (k-head) 400
ДОБАВСЫНА (ADDSON) 173
Доминатор (dominator) 239
- непосредственный (immediate) 239
Допускаться (be accepted)
- конечным автоматом (by a finite automaton) 166, 357
- магазинным автоматом (by a pushdown store automaton) 376
- машиной Тьюринга (by a Turing machine) 41, 42, 406
- РАМ (by a RAM) 19
Зависимость линейная по модулю (linear dependence modulo) 480
Задача (problem) — легко разрешимая (tractable) 404
- об упаковке (package placement) 447
- - устойчивом бракосочетании (stable marriage) 87
- о коммивояжере (travelling salesman) 447

- - кратчайшем пути (shortest path) 223
- - потоке (flow) 447—448
- - разбиении (partition) 447
- - ранце (knapsack) 446
- - расписании работ (scheduling) 448
- определения глубины (depth determination) 164
- пустоты дополнения (emptiness of complement) 457
- сортировки (sorting) 94
- трудно разрешимая (intractable) 404
ЗАДНИЙ (REAR) 62
Замыкание (closure) 225
- Клини (Kleene) 355
- рефлексивное и транзитивное (reflexive and transitive) 223
- транзитивное (transitive) 120, 223
ЗАТОЛКНУТЬ (PUSH) 61
Значение переменной (в вычислении) (valuation of a variable) 477
Значение операнда (the value of operand) 17
Идемпотентность (idempotence) 224
Идентификатор (identifier) 386
Идентифицировать (позицию) (identify) 386
Идентифицироваться (match) 399
Иерархия (hierarchy) — временная см. Иерархия по времени 471
- емкостная см. Иерархия по емкости 452
- по времени (time) 471
- - емкости (space) 452
- - памяти см. Иерархия по емкости 522
ИЗВЛЕЧЬ_МИН (EXTRACT_MIN) 162, 191
Изоморфизм (isomorphism) — деревьев (of trees) 102
- подграфу (subgraph) 446
ИМПЛАНТАЦИЯ (IMPLANT) 176, 177
ИМЯ (NAME) 58, 153
Имя переменной (в вычислении) (variable name) 477
Индекс (index) 50
Интерполяция (interpolation) 286
Исток (в графе) (source vertice) 497
Источник (source) 235
Итерация см. Замыкание Клини
- позитивная (positive) 355
Китайская теорема об остатках (Chinese remaindering) 329
Класс эквивалентности (equivalence class) 206
Клетка (cell) 40
Клика (clique) 418
КНФ (CNF) 427
Код операции (operation code) 16
Кольцо (ring) 256
- коммутативное (commutative) 256
КОНЕЦ (HEAD) 66
Конец ребра (head of the edge) 64
Конец составного ребра (head of a composite edge) 242
Конец цепочки (suffix of a string) 355
Конкатенация (concatenation)
- множеств (of sets) 225
- списков (of lists) 50
- цепочек (of strings) 355
- языков (of languages) 355
Конфигурация (configuration) 378
- C' выводима из C (C derives C') 379
- поверхностная (surface) 378
- терминальная (terminal) 379
Команда (instruction) — PAM (of a RAM) 16—18
- РАСП (of a RASP) 26
КОМПОНЕНТА (ELEMENT) 58
Компонента (component) — двусвязная (biconnected) 206
- связная (connected) 202
- сильно связная (strongly connected) 216
КОРЕНЬ (ROOT) 153
Корень (root) — графа (of a graph) 67, 239

- дерева (of a tree) 67
- из единицы (of unity) 285
- - - примитивный (principal) 285
- сильно связной компоненты (of a strongly Connected component) 216
КОРРЕКТОР (UPDATE) 380, 382
Критерий весовой (cost criterion)
- логарифмический (logarithmic) 23, 24
- равномерный (uniform) 23
Левое двойственное (множество выражений) (left dual) 496
ЛЕВЫЙСЫН (LEFTSON) 68
Легко разрешимый (tractable) 404
Лента (tape) 39
- входная (input) 15, 41, 165, 374
- выходная (output) 15, 44
Лес (forest) 67
- остовный (spanning) 130
- - глубинный (depth-first) 203
- - построенный поиском в глубину см. Лес остовный глубинный 203
Лист (leaf) 67
Литерал 16
Литерал (literal) 417, 427
Магазин (pushdown store) 61
Маркер (marker) — дна (bottom) 375
- нижний см. Маркер дна 375
Массив (array) 58
Матрица (matrix) — булева (Boolean) 274
- единичная (identity) 256
- невырожденная (nonsingular) 258
- нормированная (unit) 259
- обратная (inverse) 257
- перестановки (permutation) 259
- положительно определенная (positive definite) 282
- смежностей (adjacency matrix) 64
- тёплицева (Toeplitz) 282
- транспонированная (transpose) 259
- треугольная (triangular)
- - верхняя (upper) 258
- - нижняя (lower) 258
Машина (machine) — адресная 31
- равнодоступная адресная см. Машина с произвольным доступом к памяти 26
- - - с хранимой программой см. Машина с произвольным доступом к памяти и хранимой программой 26
- с произвольным доступом к памяти (random access) 26
- - - - - - и хранимой программой (stored program) 26
- Тьюринга (Turing) 39
- - многоленточная (multitape) 39
- - недетерминированная (nondeterministic) 405, 406
Мгновенное описание (instantaneous description) 42, 356, 375
- - допускающее (accepting) 356
- - - для of, a, 2дма, 2dpda, 376
- - of, a, 2дма, 2dpda, 375
- - MT (of a TM) 42, 406
- - НКА (of a NDFA) 356
- - НМТ (of a NDTM) 406
- - начальное (initial) 42, 356, 376, 406
- - - of, a, 2дма, 2dpda, 376
- - - НКА (of a NDFA) 356
- - - НМТ (of a NDTM) 406
Метка пути (path label) 225
Метод расстановки (hashing) 132, 196
Минимизация конечного автомата (minimization of a finite automaton) 187
- безопасное для разбиения (safe for a partition) 183
- пустое (empty) 355
- ребер, разрезающих циклы (feedback edge) 421
- регулярное (regular) 355
- узлов, разрезающих циклы (feedback vertex) 421

- универсальное (universal) см. База данных 132, 147, 188
МНОЖИТЕЛЬ (FACTOR) 264, 267
Моноид (monoid) 224
Мощность (cardinality) 64
Мультиграф (multigraph) 249
МО (ID) — см. Мгновенное описание
Наибольший общий делитель (greatest common divisor) 336, 339
Наименьшее общее кратное (least common multiple) 352
НАИМЕНЬШИЙ (SMALLEST) 175
НАЙТИ (FIND) 128
НАЙТИ-ГЛУБИНУ (FIND_DEPTH) 164
НАЙТИ-ПУТЬ (FIND _ PATH) 249
Начало отсчета (origin) 164
Начало ребра (tail of the edge) 64
- - составного (composite) 242
Начало цепочки (prefix of a string) 355
НВП-разложение (LUP decomposition) 264
НВ-разложение (LU decomposition) 264
Независимость линейная по модулю (linear independence modulo) 480
НЕПУСТОЙ (NONEMPTY) 99
Ниже (отношение на поверхностных конфигурациях) (below) 380
НИЖНИЙ (LOW) 210
НИЖНЯЯСВЯЗЬ (LOWLINK) 217
НКА (NDFA) 356
НМА (NPDA) 400
1НМА, (NPDA) 401
НОВ (NEW) 380
НОД (GCD) 336, 344
НОК (LCM) 352
НОМЕР (NUMBER) 71
О (порядок величины — order of magnitude) 12
Од (порядок величины для неветвящихся программ) 35
ОБ (порядок величины при битовых вычислениях) 35
Одв (порядок величины при применении модели с двоичными векторами) 37
Омт (порядок величины при использовании в качестве модели машины Тьюринга) 44
Ос (порядок величины при использовании модели деревьев решений) 38
Область действия переменной (the scope of a variable) 48
Обозревать (scan) 40
Обработка предварительная (preconditioning) 490, 491
Образ (pattern) 363
ОБРАТНОЕ (RECIPROCAL) 314, 315
ОБРАТНЫЙ (RECIPROCAL) 321
ОБЪЕДИНИТЬ (UNION) 128, 148
Операнд (operand) 16
Оператор (statement) — COMMENT 52
- FOR 49
- GOTO 51
- IF 49
- READ 52
- REPEAT 49
- WHILE 49
- WRITE 52
- определения процедур (procedure-difinition) 51
- помеченный (labeled) 50
- присваивания (assignment) 49
Операция (operation) — активная мультипликативная (active multiplicative) 490
- битовая (bit) 35
- ассоциативная (associative) 224
- дистрибутивная (distributive) 224
- коммутативная (commutative) 224
- с двоичными векторами (bit vector) 37
- элементарная (scalar) 230

Определитель (determinant) 258
ОТЕЦ (FATHER) 153
Отец (father) 67
Отображение памяти (memory map) 16, 26
ОЧЕРЕДЬ (QUEUE) 96
Очередь (queue) 62
- сцепляемая (concatenable) 170
- с приоритетами (priority) 170
Палиндром (palindrome) 42
Пара лежит ниже (pair is below) 380
Параметр (parameter) — фактический (actual) 51
- формальный (formal) 51
Паросочетание (pairing) 87
- устойчивое (stable) 87
ПЕРЕДНИЙ (FRONT) 62
Переменная (variable) см. Адрес символический 33
Переменная (вычисления) (variable) 477
- входная (input) 33
- выходная (output) 33
- глобальная (global) 52
- локальная (local) 52, 53
Переменная формальная (indeterminate) 476
ПЕРЕСЕЧЕНИЕ (INTERSECTION) 186
Перестановка (permutation) — нечетная (odd) 258
- четная (even) 258
ПЕРЕСЫПКА (HEAPIFY) 108
ПНОД (HGCD) 339, 340
Подграф полный (complete subgraph) см. Клика 418
Поддерево (subtree) 67
- левое (left) 68
- правое (right) 68
Подматрица (submatrix) 259
- главная (principal) 259
Подпоследовательность (subsequence) 402
Подцепочка (substring) 355
ПОЗИЦИЯ (POSITION) 59
Позиция (в цепочке) (position) 386
ПОИСК (SEARCH) 135, 138, 173, 203
Поиск (search) — в глубину (depth-first) 202
- двоичный (binary) 135
ПОИСКЕ (SEARCHB) 212
ПОИСКВ (SEARCHC) 220
Покрытие (cover) — множествами (set) 421—422
- точное (exact) 422
- узельное (vertex) 421
Поле (field) 256, 475
Полином (polynomial) — плотный (dense) 348
- разреженный (sparse) 348
Полиномиально (polynomially) — связанные (related) 39
- трансформируемый (transformable) 416
- эквивалентные см. Полиномиально связанные 39
Полнота (completeness) — NP (NP-completeness) 416
- для NP-TIME (for (NP-TIME) 416
- - P-SPACE (for P-SPACE) 440
Полукольцо замкнутое (closed semiring) 223
Полустепень исхода (out-degree) 64
Порядок (order) — внутренний (in-) 68
- лексикографический (lexicographic) 95
- линейный (linear) 94
- обратный (post-) 68
- полный (total) см. Порядок линейный 94
- прямой (pre-) 68
- частичный (partial) 94
Последовательность остатков (remainder sequence) 336
Постоянная (вычисления) (constant) 477
ПОСТРДЕРЕВА (BUILDTREE) 144

Построение сортирующего дерева (construction of a heap) 108
ПОСТРСОРТДЕРЕВА (BUILDHEAP) 109
Потомок (descendant) 67
- подлинный (proper) 67
Правило Горнера (Horner's rule) 34
ПРАВЫЙСЫН (RIGHTSON) 68
ПРИНАДЛЕЖАТЬ (MEMBER) 128
ПРЕД (PRED) 163, 380
Предок (ancestor) 67
- подлинный (proper) 67
Предшественница (predecessor) 380
- непосредственная (immediate) 380
ПРЕДЫДУЩАЯ (PREVIOUS) 61
Преобразование Фурье (Fourier transform) — быстрое (fast) 294
- дискретное (discrete) 285
- обратное (inverse) 286
Префикс (prefix) 355
Префиксный (режим, алгоритм) (on-line) 129
ПРОВЕРКА (TEST) 412
Программа (program)
- для РАМ (for RAM) 16
- на Упрощенном Алголе (Pidgin ALGOL) 48
- неветвящаяся (straight-line) 32
Программирование динамическое (dynamic programming) 83
Продукция (production) 91
Прохождение дерева (traversal of a tree) 68
- - во внутреннем порядке (inorder) 69
- - в обратном порядке (postorder) 69
- - в прямом порядке (preorder) 69
Процедура (procedure) 51
- рекурсивная (recursive) 70
Путь (path) 64
- внешний (external) 194
- внутренний (internal) 194
- простой (simple) 64
Разбиение (partitioning) 181
Разбиение грубейшее (coarsest partition) 181
Разветвление (branching) см. goto-оператор 51
Разделяй и властвуй (divide and conquer) 75
РАЗМЕР (SIZE) 147, 312
Разность циклическая (cyclic difference) 309
Разрез (cutset) 447
РАМ (RAM) 15
РАМ-программа (program) 16
- недетерминированная (nondeterministic) 415
Ранг (rank) — матрицы (of a matrix) 259
- по столбцам (column) 481
- - строкам (row) 481
- узла (of a vertex) 155
РАСП (RASP) 26
РАСП-программа (program) 26
- недетерминированная (nondeterministic) 415
Расстановка (hashing) 132
Расширение поля формальными переменными (extension of a field by indeterminates) 476
РАСЩЕПИТЬ (SPLIT) 129
Ребро (edge) 64
- древесное (tree) 203, 215
- обратное (back) 203, 215
- поперечное (cross) 215
- прямое (forward) 215
- составное (composite) 242
Регистр (register) 15
Редукция транзитивная (transitive reduction) 249
Режим — префиксный (on-line) 129
- свободный (off-line) 129
Рекурсия (recursion) 70
Свертка (convolution) 287
- отрицательно обернутая (negative wrapped) 289

- положительно обернутая (positive wrapped) 289
СВОБОДНАЯ (FREE) 59
Свободный (режим, алгоритм) (off-line) 129
Сводимый (язык) (reducible) 416
Свойство сортирующего дерева (heap property) 108
СВЯЗАТЬ (LINK) 164
Связность (графа) (connectedness) 253
СВЯЗЬ (LINK) 66
Сеть логическая (logic circuit, network) 35, 55, 498
- - комбинационная (combinational) 55
Сжатие путей (path compression) 152
Символ (symbol) — входной (input) 40
- ленточный (tape) 40
- на ленте, см. Символ ленточный 40
- несущественный (don't care) 399
- нетерминальный (nonterminal) 91
- пустой (the blank) 40
- терминальный (terminal) 91
СЛЕД (SUCC) 163
СЛЕДУЮЩАЯ (NEXT) 58
СЛЕДУЮЩИЙ (NEXT) 66, 147
Сливаемое дерево (mergeable heap) 170
СЛИТЬ (MERGE) 195
СЛИЯНИЕ(MERGE) 82
Словарь (dictionary) 129, 170
Слово (word) см. Цепочка — машинное (computer) 14, 25
См. также Число, хранимое в регистре 14, 25, 37
Сложность (complexity) — арифметическая (arithmetic) 476
- асимптотическая (asymptotic) см. Сложность временная, Сложность емкостная 21, 22, 27, 44
- временная (time) 22, 27, 44
- - в среднем (expected) 22
- - - худшем случае см. Сложность временная 22, 27, 44
- - НМТ (NDTM) 407
- в худшем случае (worst case) 22
- емкостная (space) 21, 44
- - логарифмическая (logarithmic) 24, 27
- - НМТ (NDTM) 409
- реализации булевой функции (realization of a Boolean function) 498, 499
- средняя (expected) 22
- усредненная см. Сложность средняя 22
Смежный (adjacent) 64
Смещение (displacement) 164
Содержимое регистра (the contents of a register) 16
СОРТ (SORT) 82
СОРТВЗБАЛТЫВАНИЕМ (BUBBLESORT) 123
Сортдеревом (heapsort) 106, 110
Сортировка (sorting) 93
- внешняя (external) 94
- внутренняя (internal) 99
- вставками (insertion) 126
- вычерпыванием (bucket) 95
- лексикографическая (lexicographic) 96, 98
- слиянием (merge) 82
- с помощью сравнений (by comparisons) 104
- топологическая (topological) 87
- цифровая (radix) 95
Сортирующее дерево (heap) 106
СОСТОЯНИЕ (STATUS) 53
Состояние (state) 40
- допускающее (accepting) 41, 166, 356, 375
- заключительное (final) 41
- - of, a, 2дма, 2dpda, 375
- начальное (initial) 40
- начальное (start) 166, 356, 375

- управляющего устройства (of a finite control) 375
СОЧЕТ (COMB) 89
СПИСОК (LIST) 147
Список (list) 58
- дважды связанный (doubly linked) 61
- свободный (free) 60
- смежностей (adjacency) 66
Способность пропускная (capacity) 448, 497
Сравнение (comparison) 38
- ключевое (key) 121
СТ (DEG) 312
Стабильный (метод сортировки) (stable) 126
Статистики порядковые (order statistics) 93, 117
Стек (stack) 61
Степень (degree)
- полинома 312, 491
- полинома от нескольких переменных (of a multivariate polynomial) 491
- узла 64
Стоимость (cost) 223, 225
- пути (of a path) 223
Структура данных (data structure) 58
СУММАСТРОК (ROWSUM) 278
Сумматор (accumulator) 16
Суффикс (suffix) 355
Схема логическая (logic circuit) см. Сеть логическая 35, 55, 498
- сдвигающая (shifting network) 497
Сцепление см. Конкатенация 50, 225, 355
СЧЕТ (COUNT) 71, 153
Счетчик команд (location counter) 16, 26
Сын (son) 67
- а 386
- левый (left) 68
- правый (right) 68
Тавтология (tautology) 446
Теорема о свертке (convolution theorem) 288
ТЕРМ (TERM) 380
Терминатор (terminator) 379
Тип данных (data type) 48
Точка сочленения (articulation point) 206
Тройка допустимая (admissable triple) 309
Трудно разрешимый (intractable) 404
УДАЛИТЬ (DELETE) 128
УЗЕЛ (VERTEX) 71
Узел (vertex, node) 64
- смежный (adjacent) 64
Умножение (multiplication) — активное (active) 484
- векторов (of vectors) 493
- и/или (and/or) 352, 399
- комплексных чисел (of complex numbers) 478, 479, 489
- матриц (of matrices) 259, 495
- матрицы на вектор (of a matrix by a vector) 486, 494
- полиномов (of polynomials) 480, 487, 492, 493
- целых чисел (of integers) 77—80, 304
Уровень узла (level of a vertex) 68
Устройство управляющее (finite control) 40, 165, 356
Форма (form) — билинейная (bilinear) 495
- конъюнктивная нормальная (conjunctive normal) 427
- нормальная Хомского (Chomsky normal) 91
Формула булева (Boolean expression) 417
- выполнимая (satisfiable) 419
- Лагранжа интерполяционная (Lagrangian interpolation formula) 329
Фрагмент стека (stack frame) 73
Функция (function) 51

- булева (Boolean) 498
- логарифмическая (logarithmic) 23
- конструируемая по времени (time-constructable) 471
- - - емкости (space) 411
- - - памяти см. Функция, конструируемая по емкости 411
- отказов (failure) 368
- переходов (next-move) 41
- переходов (state transition) 356
- расстановки (hashing) 130
- стоимости (cost) 130
- экспоненциальная (exponential) 39
- элементарная (elementary) 466
Ханойские башни (towers of Hanoi) 88
Характеристический (вектор) (characteristic) 63
Цвет (color) 421
ЦЕПОЧКА (STRING) 99
Цепочка (string) 355
- допускаемая (автоматом, машиной) (accepted) 166, 357, 376
- пустая (empty) 165, 355
- с несущественными символами (with don't cares) 399
Цепочка-текст (text-string) 363
Цепь (chain) 397
ЦИКЛ (CYCLE) 457
Цикл (cycle) 64
- гамильтонов (Hamilton) 421
- - ориентированный (directed) 421
- эйлеров (Euler) 249
Циркулянт (circulant) 310
ЧАСТИЧНО НАЙТИ (partial FIND) 158
Черпак (bucket) 95
ЧИС (NUM) 278
Число (number)
- Каталана (Catalan) 91
- хранимое в регистре (stored in a register) 14, 25, 37
- хроматическое (chromatic) 421
ЧН (PF) 158
Шаг (вычисления) (step) 477
Шаг (работы автомата, машины) (move) — by, a, 2дма, 2dpda, 376
- НМТ (by a NDTM) 406
Эквивалентность (equivalence) — автоматов (of automata) 166
- векторов по модулю (of vectors modulo) 480
- регулярных выражений (of regular expressions) 356
- отношение (relation) 206
- состояний (of states) 166
Элем (Item) 58
ЭЛЕМЕНТ (ELEMENT) 153
ЭЛЕМЕНТ (ITEM) 59
Элемент единичный (identity) 224
- k-й наименьший (the kth smallest element) 117
- обратный (inverse) 256
Элементы смежные матрицы (adjacent entries of a matrix) 252
Язык (language) 19, 355
- бесконтекстный (context-free) 401
- допускаемый автоматом (accepted by an automaton) 166, 357
- - by, a, 2дма, 2dpda, 376
- - МТ (by a TM) 42
- - НМТ (by a NDTM) 407
- - программой (by a program) 19
- контекстный (context-sensitive) 449
NP-полный (NP-complete) 416
- полный для недетерминированного полиномиального времени (for nondeterministic polynomial time) см. Язык NP-полный 416
- порождаемый грамматикой (generated by a grammar) 91
- регулярный (regular) 356

О пользовании книгой

Эта книга задумана как начальный курс по разработке и анализу алгоритмов. Основной упор сделан на идеи и простоту понимания, а не на проработку деталей реализации или программистские трюки. Часто вместо длинных утомительных доказательств даются лишь неформальные интуитивные объяснения.

Книга замкнута в себе и не предполагает специальной подготовки ни по математике, ни по языкам программирования. Однако желательна определенная зрелость в навыках обращения с математическими понятиями, а также некоторое знакомство с языками программирования высокого уровня, такими, как Фортран или Алгол. Для полного понимания гл. 6—8 и 12 нужны некоторые познания в линейной алгебре.

Эта книга использовалась как основа для спецкурсов по разработке алгоритмов. За один семестр изучался материал, покрывающий бóльшую часть гл. 1—5, 9 и 10 вместе с беглым обзором тематики остальных глав. Во вводных курсах основное внимание уделялось материалу из гл. 1—5, но разд. 1.6, 1.7, 4.13, 5.11 и теорема 4.5 обычно не изучались. Книгу можно также использовать для более глубоких спецкурсов, делая упор на теорию алгоритмов. Основой для этого могли бы служить гл. 6—12.

В конце каждой главы приведены упражнения, обеспечивающие преподавателя широким выбором домашних заданий. Упражнения классифицированы по трудности. Упражнения без звездочек годятся для вводных курсов; упражнения с одной звездочкой труднее, а с двумя — еще труднее. В замечаниях по литературе в конце каждой главы делается попытка указать печатный источник для каждого алгоритма и результата, содержащихся в тексте и упражнениях.

Благодарности

Материал книги основан на набросках лекций для курсов, читавшихся авторами в Корнеллском и Принстонском университетах и в Стивенсонском технологическом институте. Авторы хотели бы поблагодарить многих людей, которые критически прочитали различные части рукописи и внесли полезные предложения. Особенно мы хотели бы поблагодарить Келлога Бута, Стэна Брауна, Стива Чена, Алена Сайфера, Арча Дейвиса, Майка Фишера, Хейнию Гаевску, Майка Гэри, Юдая Гупту, Майка Харрисона, Матта Хекта, Гарри Ханта, Дейва Джонсона, Марка Каплана, Дона Джонсона, Стива Джонсона, Брайана Кернигана, Дона Кнута, Ричарда Лэднера, Аниту Ла-Саль, Дуга Мак-Илроя, Альберта Мейера, Кристоса Пападимитриу, Билла Плогера, Джона Сэвиджа, Ховарда Зигеля, Кена Стейглица, Лэрри Стокмейера, Тома Жимански и Теодора Ена.

Мы выражаем особую благодарность Джеме Карневейл, Полине Камерон, Ханне Крессе, Эдит Персер и Руфи Сузуки за быструю и качественную перепечатку рукописи.

Авторы также благодарны фирме Bell Laboratories и университетам Корнеллскому, Принстонскому и Калифорнийскому (отделение в Беркли) за предоставленные возможности для подготовки рукописи.

Июнь 1974

Альфред Ахо
Джон Хопкрофт
Джефри Ульман

МОДЕЛИ ВЫЧИСЛЕНИЙ

Если дана задача, как найти для ее решения эффективный алгоритм? А если алгоритм найден, как сравнить его с другими алгоритмами, решающими ту же задачу? Как оценить его качество? Вопросы такого рода интересуют и программистов, и тех, кто занимается теоретическим исследованием вычислений. В этой книге мы изучим различные подходы, с помощью которых пытаются получить ответ на вопросы, подобные перечисленным.

В настоящей главе рассматриваются несколько моделей вычислительной машины — машина с произвольным доступом к памяти, машина с произвольным доступом к памяти и хранимой программой и машина Тьюринга. Эти модели сравниваются по их способности отражать сложность алгоритма, и на их основе строятся более специализированные модели вычислений, а именно: неветвящиеся арифметические программы, битовые вычисления, вычисления с двоичными векторами и деревья решений. В последнем разделе этой главы вводится язык для описания алгоритмов, называемый Упрощенным Алголом.

1.1. АЛГОРИТМЫ И ИХ СЛОЖНОСТИ

Для оценки алгоритмов существует много критериев. Чаще всего нас будет интересовать порядок роста необходимых для решения задачи времени и емкости памяти при увеличении входных данных. Нам хотелось бы связать с каждой конкретной задачей некоторое число, называемое ее *размером*, которое выражало бы меру количества входных данных. Например, размером задачи умножения матриц может быть наибольший размер матриц-сомножителей. Размером задачи о графах может быть число ребер данного графа.

Время, затрачиваемое алгоритмом, как функция размера задачи, называется *временно́й сложностью* этого алгоритма. Поведение этой сложности в пределе при увеличении размера задачи называется *асимптотической временно́й сложностью*. Аналогично можно опре-

делить *емкостную сложность* и *асимптотическую емкостную сложность*.

Именно асимптотическая сложность алгоритма определяет в итоге размер задач, которые можно решить этим алгоритмом. Если алгоритм обрабатывает входы размера n за время cn^2, где c — некоторая постоянная, то говорят, что временна́я сложность этого алгоритма есть $O(n^2)$ (читается "порядка n^2"). Точнее, говорят, что (неотрицательная) функция $g(n)$ есть $O(f(n))$, если существует такая постоянная c, что $g(n) \leqslant cf(n)$ для всех, кроме конечного (возможно, пустого) множества, неотрицательных значений n.

Можно было бы подумать, что колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения цифровых вычислительных машин, уменьшит значение эффективных алгоритмов. Однако происходит в точности противоположное. Так как вычислительные машины работают все быстрее и мы можем решать все бóльшие задачи, именно сложность алгоритма определяет то увеличение размера задачи, которое можно достичь с увеличением скорости машины.

Допустим, у нас есть пять алгоритмов A_1—A_5 со следующими временными сложностями:

Алгоритм	Временная сложность
A_1	n
A_2	$n \log n$[1])
A_3	n^2
A_4	n^3
A_5	2^n

Здесь временная сложность — это число единиц времени, требуемого для обработки входа размера n. Пусть единицей времени будет одна миллисекунда. Тогда алгоритм A_1 может обработать за одну секунду вход размера 1000, в то время как A_5 — вход размера не более 9. На рис. 1.1 приведены размеры задач, которые можно решить за одну секунду, одну минуту и один час каждым из этих пяти алгоритмов.

Предположим, что следующее поколение вычислительных машин будет в 10 раз быстрее нынешнего. На рис. 1.2 показано, как возрастут размеры задач, которые мы сможем решить благодаря этому увеличению скорости. Заметим, что для алгоритма A_5 десятикратное увеличение скорости увеличивает размер задачи, которую можно решить, только на три, тогда как для алгоритма A_3 размер задачи более чем утраивается.

Вместо эффекта увеличения скорости рассмотрим теперь эффект применения более действенного алгоритма. Вернемся к рис. 1.1.

[1]) Если не оговорено противное, все логарифмы в этой книге берутся по основанию 2.

Алгоритм	Временна́я сложность	Максимальный размер задачи		
		1 с	1 мин	1 ч
A_1	n	1000	6×10^4	$3{,}6\times10^6$
A_2	$n \log n$	140	4893	$2{,}0\times10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Рис. 1.1. Границы размеров задач, определяемые скоростью роста сложности.

Если в качестве основы для сравнения взять 1 мин, то, заменяя алгоритм A_4 алгоритмом A_3, можно решить задачу, в 6 раз большую, а заменяя A_4 на A_2, можно решить задачу, большую в 125 раз. Эти результаты производят гораздо большее впечатление, чем двукратное улучшение, достигаемое за счет десятикратного увеличения скорости. Если в качестве основы для сравнения взять 1 ч, то различие оказывается еще значительнее. Отсюда мы заключаем, что асимптотическая сложность алгоритма служит важной мерой качественности алгоритма, причем такой мерой, которая обещает стать еще важнее при последующем увеличении скорости вычислений.

Несмотря на то что основное внимание здесь уделяется порядку роста величин, надо понимать, что большой порядок роста сложности алгоритма может иметь меньшую мультипликативную посто-

Алгоритм	Временна́я сложность	Максимальный размер задачи	
		до ускорения	после ускорения
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	Примерно $10s_2$ для больших s_2
A_3	n^2	s_3	$3{,}16s_3$
A_4	n^3	s_4	$2{,}15s_4$
A_5	2^n	s_5	$s_5+3{,}3$

Рис. 1.2. Эффект десятикратного ускорения.

янную, чем малый порядок роста сложности другого алгоритма. В таком случае алгоритм с быстро растущей сложностью может оказаться предпочтительнее для задач с малым размером — возможно, даже для всех задач, которые нас интересуют. Например, предположим, что временные сложности алгоритмов A_1, A_2, A_3, A_4 и A_5 в действительности равны соответственно $1000n$, $100n \log n$, $10n^2$, n^3 и 2^n. Тогда A_5 будет наилучшим для задач размера $2 \leqslant n \leqslant 9$, A_3 — для задач размера $10 \leqslant n \leqslant 58$, A_2 — при $59 \leqslant n \leqslant 1024$, а A_1 — при $n > 1024$.

Прежде чем продолжать обсуждение алгоритмов и их сложностей, мы должны точно определить модель вычислительного устройства, используемого для реализации алгоритмов, а также условиться, что понимать под элементарным шагом вычисления. К сожалению, нет такой вычислительной модели, которая была бы хороша во всех ситуациях. Одна из основных трудностей связана с размером машинных слов. Например, если допустить, что машинное слово может быть целым числом произвольной длины, то всю задачу можно закодировать одним числом в виде одного машинного слова. Если же допустить, что машинное слово имеет фиксированную длину, то возникают трудности даже просто запоминания произвольно больших чисел, не говоря уже о других проблемах, которые часто удается обойти, если решаются задачи скромного размера. Для каждой задачи надо выбирать подходящую модель, которая точно отразит действительное время вычисления на реальной вычислительной машине.

В следующих разделах мы обсудим несколько основных моделей вычислительных устройств, наиболее важными из которых являются машина с произвольным доступом к памяти, машина с произвольным доступом к памяти и хранимой программой и машина Тьюринга. Эти три модели эквивалентны с точки зрения принципиальной вычислимости, но не с точки зрения скорости вычислений.

Быть может, наиболее важным мотивом, побудившим рассмотрение формальных моделей вычисления, было желание раскрыть подлинную вычислительную сложность различных задач. Нам хотелось бы получить нижние оценки на время вычислений. Чтобы показать, что не существует алгоритма, выполняющего данное задание менее чем за определенное время, необходимо точное и подчас высоко специализированное определение того, что есть алгоритм. Одним из примеров такого определения служат машины Тьюринга в разд. 1.6.

Для описания алгоритмов желательно иметь обозначения, более естественные и легче понимаемые, чем программа для машины с произвольным доступом к памяти, машины с произвольным доступом к памяти и хранимой программой или машины Тьюринга. Поэтому мы введем также язык высокого уровня, называемый Упрощенным Алголом. Именно этот язык будет использоваться во всей книге для

описания алгоритмов. Однако, чтобы понимать вычислительную сложность алгоритма, написанного на Упрощенном Алголе, мы должны соотнести Упрощенный Алгол с более формальными моделями. Это будет сделано в последнем разделе настоящей главы.

1.2. МАШИНЫ С ПРОИЗВОЛЬНЫМ ДОСТУПОМ К ПАМЯТИ

Машина с произвольным доступом к памяти (или, иначе, равнодоступная адресная машина — сокращенно РАМ) моделирует вычислительную машину с одним сумматором, в которой команды программы не могут изменять сами себя.

РАМ состоит из входной ленты, с которой она может только считывать, выходной ленты, на которую она может только записывать, и памяти (рис. 1.3). Входная лента представляет собой последовательность клеток, каждая из которых может содержать целое число (возможно, отрицательное). Всякий раз, когда символ считывается с входной ленты, ее читающая головка сдвигается на одну клетку вправо. Выход представляет собой ленту, на которую машина может только записывать; она разбита на клетки, которые вначале все пусты. При выполнении команды записи в той клетке выходной ленты, которую в текущий момент обозревает ее головка, печатается целое число и головка затем сдвигается на одну клетку вправо. Как только выходной символ записан, его уже нельзя изменить.

Память состоит из последовательности регистров r_0, r_1, . . ., r_i, . . ., каждый из которых способен хранить произвольное целое

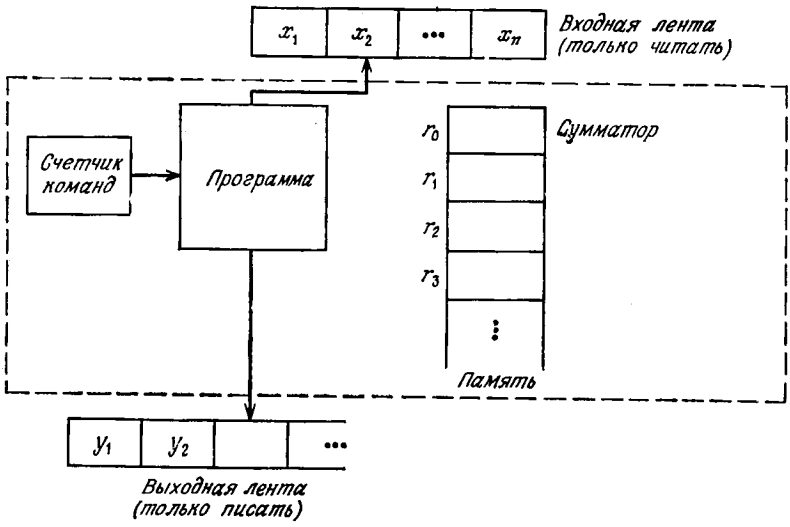


Рис. 1.3. Машина с произвольным доступом к памяти.

число. На число регистров, которые можно использовать, мы не устанавливаем верхней границы. Такая идеализация допустима в случаях, когда

1) размер задачи достаточно мал, чтобы она поместилась в основную память вычислительной машины,
2) целые числа, участвующие в вычислении, достаточно малы, чтобы их можно было помещать в одну ячейку.

Программа для РАМ (или РАМ-программа) не записывается в память. Поэтому мы предполагаем, что программа не изменяет сама себя. Программа является, в сущности, последовательностью (возможно) помеченных команд. Точный тип команд, допустимых в программе, не слишком важен, пока они напоминают те, которые обычно встречаются в реальных вычислительных машинах. Мы предполагаем, что имеются арифметические команды, команды ввода-вывода, косвенная адресация (например, для индексации массивов) и команды разветвления. Все вычисления производятся в первом регистре r_0, называемом *сумматором*, который, как и всякий другой регистр памяти, может хранить произвольное целое число. Пример набора команд для РАМ показан на рис. 1.4. Каждая команда состоит из двух частей — *кода операции* и *адреса*.

В принципе можно было бы добавить к нашему набору любые другие команды, встречающиеся в реальных вычислительных машинах, такие, как логические или литерные операции, и при этом порядок сложности задач не изменится. Читателю разрешается думать, что набор команд дополнен так, как это его устраивает.

Операнд может быть одного из следующих типов:

1) $=i$ означает само целое число i и называется литералом,
2) i — содержимое регистра i (i должно быть неотрицательным),
3) $*i$ означает косвенную адресацию, т. е. значением операнда служит содержимое регистра j, где j — целое число, находящееся в регистре i; если $j<0$, машина останавливается.

Эти команды хорошо знакомы всякому, кто программировал на языке ассемблера. Можно определить значение программы P с помощью двух объектов: отображения c из множества неотрицательных целых чисел в множество целых чисел и «счетчика команд», который определяет очередную выполняемую команду. Функция c есть *отображение памяти*, а именно $c(i)$ — целое число, содержащееся в регистре i (*содержимое* регистра i).

Вначале $c(i)=0$ для всех $i\geqslant 0$, счетчик команд установлен на первую команду в P, а выходная лента пуста. После выполнения k-й команды из P счетчик команд автоматически переходит на $k+1$ (т. е. на очередную команду), если k-я команда не была командой вида JUMP, HALT, JGTZ или JZERO.

Код операции	Адрес
1. LOAD	операнд
2. STORE	операнд
3. ADD	операнд
4. SUB	операнд
5. MULT	операнд
6. DIV	операнд
7. READ	операнд
8. WRITE	операнд
9. JUMP	метка
10. JGTZ	метка
11. JZERO	метка
12. HALT	

Рис. 1.4. Таблица команд RAM.

Чтобы описать действие команды, зададим *значение* $v(a)$ *операнда* a:

$$v(=i)=i,$$
$$v(i)=c(i),$$
$$v(*i)=c(c(i)).$$

Таблица на рис. 1.5 определяет действие каждой команды из таблицы на рис. 1.4. Команды, действию которых не дано определения (такие, как STORE $=i$), можно считать эквивалентными команде HALT. Аналогично деление на нуль останавливает машину.

При выполнении любой из первых восьми команд счетчик команд увеличивается на единицу. Поэтому команды в данной программе выполняются последовательно, до тех пор пока не встретится команда JUMP или HALT либо JGTZ при содержимом сумматора, большем нуля, либо JZERO при содержимом сумматора, равном нулю.

Вообще говоря, RAM-программа определяет отображение из множества входных лент в множество выходных лент. Так как на некоторых входных лентах программа может не останавливаться, это отображение является частичным (т. е. для некоторых входов оно может быть не определено). Это отображение можно интерпретировать разными способами. Две важные интерпретации — интерпретация в виде функции и интерпретация в виде языка.

Предположим, что программа P всегда считывает с входной ленты n целых чисел и записывает на выходную ленту не более одного

Команда	Действие
1. LOAD a	$c(0) \leftarrow v(a)$
2. STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
3. ADD a	$c(0) \leftarrow c(0) + v(a)$
4. SUB a	$c(0) \leftarrow c(0) - v(a)$
5. MULT a	$c(0) \leftarrow c(0) \times v(a)$
6. DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$ [1])
7. READ i	$c(i) \leftarrow$ очередной входной символ.
READ $*i$	$c(c(i)) \leftarrow$ очередной входной символ. В обоих случаях головка входной ленты сдвигается на одну клетку вправо.
8. WRITE a	$v(a)$ печатается в той клетке выходной ленты, которую в данный момент обозревает ее головка. Затем эта головка сдвигается на одну клетку вправо.
9. JUMP b	Счетчик команд устанавливается на команду с меткой b.
10. JGTZ b	Если $c(0) > 0$, то счетчик команд устанавливается на команду с меткой b, в противном случае на следующую команду.
11. JZERO b	Если $c(0) = 0$, то счетчик команд устанавливается на команду с меткой b, в противном случае на следующую команду.
12. HALT	Работа прекращается.

[1]) Повсюду в этой книге $\lceil x \rceil$ *(потолок x)* обозначает наименьшее целое, большее или равное x, $\lfloor x \rfloor$ *(дно, или целая часть x)* обозначает наибольшее целое, меньшее или равное x.

Рис. 1.5. Действие команд РАМ. Операнд a есть $=i$, i или $*i$.

целого числа. Пусть $x_1, x_2, \ldots, x_n$ — целые числа, находящиеся в n первых клетках входной ленты, и пусть программа P записывает y в первую клетку выходной ленты, а затем через некоторое время останавливается. Тогда говорят, что P вычисляет функцию $f(x_1, \ldots, x_n) = y$. Легко показать, что РАМ, как и всякая другая разумная

модель вычислительной машины, может вычислять в точности *частично рекурсивные функции*. Иными словами, для произвольной частично рекурсивной функции f можно написать РАМ-программу, вычисляющую f, и для произвольной РАМ-программы можно указать эквивалентную ей частично рекурсивную функцию. (По поводу рекурсивных функций см. Дэвис [1958] или Роджерс [1967].)

Другой способ интерпретировать программу для РАМ — это посмотреть на нее с точки зрения допускаемого ею языка. *Алфавит* — это конечное множество символов, *язык* — множество цепочек (слов) алфавита. Символы алфавита можно представить целыми числами 1, 2, . . ., k при некотором k. Данная РАМ допускает (воспринимает) язык в следующем смысле. Пусть на входной ленте находится цепочка $s=a_1a_2 \ldots a_n$, причем символ a_1 расположен в первой клетке, a_2— во второй и т. д., а в $(n+1)$-й клетке расположен 0 — символ, который будет использоваться в качестве концевого маркера, т. е. маркера конца входной цепочки.

Входная цепочка s *допускается* РАМ-программой P, если P прочитывает все ее символы и концевой маркер, пишет 1 в первой клетке выходной ленты и останавливается.

Языком, допускаемым программой P, называется множество всех цепочек (слов), допускаемых ею. Для входных цепочек, не принадлежащих языку, допускаемому программой P, она может напечатать на выходной ленте символ, отличный от 1, и остановиться, а может даже и не остановиться вообще. Легко показать, что язык

```
begin
  read r1;
  if r1 ⩽ 0 then write 0
  else
    begin
      r2 ← r1;
      r3 ← r1 — 1;
      while r3 > 0 do
        begin
          r2 ← r2 * r1;
          r3 ← r3 — 1
        end;
      write r2
    end
end
```

Рис. 1.6. Программа для n^n на Упрощенном Алголе.

РАМ-программа			Соответствующие операторы на Упрощенном Алголе
	READ	1	**read** $r1$
	LOAD	1	**if** $r1 \leqslant 0$ **then write** 0
	JGTZ	полож	
	WRITE	$=0$	
	JUMP	конецесли	
полож:	LOAD	1	$r2 \leftarrow r1$
	STORE	2	
	LOAD	1	$r3 \leftarrow r1 - 1$
	SUB	$=1$	
	STORE	3	
пока:	LOAD	3	**while** $r3 > 0$ **do**
	JGTZ	продолж	
	JUMP	конецпока	
продолж:	LOAD	2	$r2 \leftarrow r2 * r1$
	MULT	1	
	STORE	2	
	LOAD	3	$r3 \leftarrow r3 - 1$
	SUB	$=1$	
	STORE	3	
	JUMP	пока	
конецпока:	WRITE	2	**write** $r2$
конецесли:	HALT		

Рис. 1.7. РАМ-программа для n^n.

допускается некоторой РАМ тогда и только тогда, когда он *рекурсивно перечислим*. Язык допускается РАМ, останавливающейся на всех входах, тогда и только тогда, когда он *рекурсивен* (о рекурсивных и рекурсивно перечислимых языках см. Хопкрофт, Ульман [1969]).

Рассмотрим две программы для РАМ. Первая определяет функцию, вторая допускает язык.

Пример 1.1. Пусть

$$f(n) = \begin{cases} n^n & \text{для всех целых } n \geqslant 1, \\ 0 & \text{для } n = 0. \end{cases}$$

```
begin
  d ← 0;
  read x;
  while x ≠ 0 do
    begin
      if x ≠ 1 then d ← d − 1 else d ← d + 1;
      read x
    end;
  if d = 0 then write 1
end
```

Рис. 1.8. Распознавание цепочек с одинаковыми числами вхождений 1 и 2.

РАМ-программа			Соответствующие операторы на Упрощенном Алголе
	LOAD	=0	$d \leftarrow 0$
	STORE	2	
	READ	1	**read** x
пока:	LOAD	1	**while** $x \neq 0$ **do**
	JZERO	конецпока	
	LOAD	1	**if** $x \neq 1$
	SUB	=1	
	JZERO	один	
	LOAD	2	**then** $d \leftarrow d - 1$
	SUB	=1	
	STORE	2	
	JUMP	конецесли	
один:	LOAD	2	**else** $d \leftarrow d + 1$
	ADD	=1	
	STORE	2	
конецесли:	READ	1	**read** x
	JUMP	пока	
конецпока:	LOAD	2	**if** $d = 0$ **then write** 1
	JZERO	выход	
	HALT		
выход:	WRITE	=1	
	HALT		

Рис. 1.9. РАМ-программа, соответствующая алгоритму на рис. 1.8.

Программа на Упрощенном Алголе, вычисляющая $f(n)$ путем $(n-1)$-кратного умножения на n, приведена на рис. 1.6 [1]). Соответствующая программа для РАМ дана на рис. 1.7. Переменные $r1$, $r2$ и $r3$ хранятся в регистрах 1, 2 и 3 соответственно. Мы специально не сделали очевидных улучшений программы, чтобы яснее было видно соответствие между рис. 1.6 и 1.7. □

Пример 1.2. Рассмотрим РАМ-программу, которая допускает язык во входном алфавите {1, 2}, состоящий из всех цепочек с одинаковым числом вхождений 1 и 2. Эта программа считывает каждый входной символ в регистр 1, а в регистре 2 оставляет разность d между количеством символов 1 и 2, поступивших до текущего момента. Встретив концевой маркер 0, программа сравнивает d с нулем и в случае совпадения печатает 1 и останавливается. Мы считаем 0, 1 и 2 единственно возможными входными символами.

Основные детали алгоритма приведены в программе на рис. 1.8. Эквивалентная программа для РАМ дана на рис. 1.9; x хранится в регистре 1, а d — в регистре 2. □

1.3. ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ РАМ-ПРОГРАММ

Двумя важными мерами сложности алгоритма являются временна́я и емкостная сложности, рассматриваемые как функции размера входа. Если при данном размере в качестве меры сложности берется наибольшая из сложностей (по всем входам этого размера), то она называется *сложностью в худшем случае*. Если в качестве меры сложности берется "средняя" сложность по всем входам данного размера, то она называется *средней* (или *усредненной*) сложностью. Обычно среднюю сложность найти труднее, чем сложность в худшем случае. Нужно еще принять некоторое предположение о распределении входов, а реалистичные допущения часто бывает трудно сформулировать математически. Мы будем уделять основное внимание худшему случаю, поскольку его легче исследовать и он имеет универсальную приложимость. Однако следует помнить, что алгоритм с наименьшей сложностью в худшем случае не обязательно имеет лучшую сложность в среднем.

Временна́я сложность в худшем случае (или просто *временна́я сложность*) РАМ-программы — это функция $f(n)$, равная наибольшей (по всем входам размера n) из сумм времен, затраченных на каждую сработавшую команду. *Временна́я сложность в среднем* — это среднее, взятое по всем входам размера n, тех же самых сумм. Такие же понятия определяются для емкости памяти, только вместо "времен, затраченных на каждую сработавшую команду" надо подставить "емкостей всех регистров, к которым было обращение".

[1]) Описание Упрощенного Алгола см. в разд. 1.8.

Чтобы точно определить временную и емкостную сложности, надо указать время, необходимое для выполнения каждой РАМ-команды, и объем памяти, используемый каждым регистром. Мы рассмотрим два таких весовых критерия для наших программ. При *равномерном весовом критерии* каждая РАМ-команда затрачивает одну единицу времени и каждый регистр использует одну единицу памяти. Если не оговорено противное, то сложность РАМ-программы будет измеряться в соответствии с равномерным весовым критерием.

Второе определение, иногда более реалистичное, принимает во внимание ограниченность размера реальной ячейки памяти и называется *логарифмическим весовым критерием*. Пусть $l(i)$ — *логарифмическая функция* на целых числах, заданная равенствами

$$l(i)=\begin{cases}\lfloor \log|i| \rfloor+1, & i\neq 0,\\ 1, & i=0.\end{cases}$$

Таблица на рис. 1.10 дает логарифмические веса $t(a)$ для всех трех возможных видов операнда a. На рис. 1.11 приведены веса РАМ-команд.

Здесь учитывается, что для представления целого числа n в регистре требуется $\lfloor \log n \rfloor+1$ битов. Регистры же, напомним, могут содержать произвольно большие целые числа.

Логарифмический весовой критерий основан на грубом допущении, что цена выполнения команды (ее вес) пропорциональна длине ее операндов. Например, рассмотрим вес команды ADD$*i$. Сначала мы должны определить трудность декодирования операнда, представленного адресом. Просмотр целого числа i занимает время $l(i)$. Затем, чтобы прочитать содержимое $c(i)$ регистра i и определить его местоположение, понадобится время $l(c(i))$. Наконец, считывание содержимого $c(i)$ требует время $l(c(c(i)))$. Так как команда ADD $*i$ прибавляет целое число $c(c(i))$ к целому числу $c(0)$ в сумма-

Операнд a	Вес $t(a)$
$=i$	$l(i)$
i	$l(i)+l(c(i))$
$*i$	$l(i)+l(c(i))+l(c(c(i)))$

Рис. 1.10. Логарифмические веса операндов.

Команда	Вес
1. LOAD a	$t(a)$
2. STORE i	$l(c(0))+l(i)$
STORE $*i$	$l(c(0))+l(i)+l(c(i))$
3. ADD a	$l(c(0))+t(a)$
4. SUB a	$l(c(0))+t(a)$
5. MULT a	$l(c(0))+t(a)$
6. DIV a	$l(c(0))+t(a)$
7. READ i	$l(\text{вход})+l(i)$
READ $*i$	$l(\text{вход})+l(i)+l(c(i))$
8. WRITE a	$t(a)$
9. JUMP b	1
10. JGTZ b	$l(c(0))$
11. JZERO b	$l(c(0))$
12. HALT	1

Рис. 1.11. Логарифмические веса команд РАМ, где $t(a)$ — вес операнда a, а b обозначает метку.

торе, то ясно, что разумным весом, который следует придать команде ADD $*i$, является $l(c(0))+l(i)+l(c(i))+l(c(c(i)))$.

Определим *логарифмическую емкостную сложность* РАМ-программы как сумму по всем работавшим регистрам, включая сумматор, величин $l(x_i)$, где x_i — наибольшее по абсолютной величине целое число, содержавшееся в i-м регистре за все время вычислений.

Само собой разумеется, данная программа может иметь радикально различные временные сложности в зависимости от того, какой используется весовой критерий — равномерный или логарифмический. Если разумно предположить, что каждое число, встречающееся в задаче, можно хранить в виде одного машинного слова, то годится равномерная весовая функция. В иной ситуации для реалистического анализа сложности более подходящим может оказаться логарифмический вес.

Оценим временную и емкостную сложности РАМ-программы из примера 1.1, вычисляющей n^n. При подсчете временной сложности будет доминировать цикл с командой MULT. Когда эта команда выполняется i-й раз, сумматор содержит n^i, а регистр 2 содержит n. Всего команда MULT выполняется $n-1$ раз. При равномерном весовом критерии каждая команда MULT требует одну единицу вре-

мени, и поэтому на выполнение всех команд MULT тратится время $O(n)$. При логарифмическом весовом критерии i-я команда MULT занимает время $l(n^i)+l(n)\approx(i+1)\log n$, так что время выполнения всех команд MULT равно

$$\sum_{i=1}^{n-1}(i+1)\log n,$$

что составляет $O(n^2 \log n)$.

Емкостная сложность определяется числами, которые хранились в регистрах от 0 до 3. При равномерном весовом критерии емкостная сложность составляет $O(1)$, а при логарифмическом — $O(n \log n)$, поскольку наибольшее целое число среди содержавшихся в этих регистрах есть n^n, а $l(n^n)\approx n \log n$. Таким образом, сложности для программы из примера 1.1 таковы:

	Равномерный вес	Логарифмический вес
Временна́я сложность	$O(n)$	$O(n^2 \log n)$
Емкостная сложность	$O(1)$	$O(n \log n)$

Для этой программы равномерный вес реалистичен только в ситуации, когда столь большое целое, как n^n, записывается в виде одного машинного слова. Если n^n превышает то, что можно представить одним машинным словом, то даже логарифмическая временная сложность до некоторой степени нереалистична, поскольку она предполагает, что два целых числа i и j перемножаются за время $O(l(i)+l(j))$, а возможность этого неизвестна.

Для программы из примера 1.2 в предположении, что n — длина входного слова, временные и емкостные сложности таковы:

	Равномерный вес	Логарифмический вес
Временна́я сложность	$O(n)$	$O(n^2 \log n)$
Емкостная сложность	$O(1)$	$O(n \log n)$

Для этой программы в ситуации, когда n больше того, что можно запомнить в одном машинном слове, логарифмический вес оказывается вполне реалистичным.

1.4. МОДЕЛЬ С ХРАНИМОЙ ПРОГРАММОЙ

Поскольку РАМ-программа не хранится в памяти РАМ, она не может изменять себя. Сейчас мы рассмотрим другую модель вычислительной машины, называемую *машиной с произвольным доступом к памяти и хранимой программой* (или, иначе, *равнодоступной адресной машиной с хранимой программой* — сокращенно РАСП), которая отличается от РАМ лишь тем, что ее программа находится в памяти и может изменять себя.

Набор команд для РАСП совпадает с соответствующим набором для РАМ во всем, кроме косвенной адресации, которая исключена, ибо она не нужна. Мы увидим, что РАСП может моделировать косвенную адресацию путем изменения команд в процессе выполнения программы.

Общая структура РАСП также подобна структуре РАМ, но только предполагается, что РАСП-программа находится в регистрах памяти. Каждая РАСП-команда занимает два последовательных регистра памяти. Первый регистр содержит код операции, второй — адрес. Если адрес имеет вид $=i$, то первый регистр будет содержать (в закодированном виде) указание на то, что операнд является литералом, а второй регистр будет содержать i. Для кодирования команд берутся целые числа. На рис. 1.12 представлено одно возможное кодирование. Например, команда LOAD=32 должна храниться в виде числа 2 в одном регистре и 32 в следующем регистре.

Так же как для РАМ, состояние РАСП можно представить с помощью

1) отображения памяти c, где $c(i)$, $i \geqslant 0$,— содержимое i-го регистра,
2) счетчика команд, указывающего первый из двух последовательных регистров памяти, из которых надлежит взять текущую команду.

Вначале счетчик команд устанавливается на некоторый выделенный регистр. Обычно исходное содержимое регистров памяти не состоит из одних нулей, так как в память уже введена программа. Однако мы требуем, чтобы вначале все регистры, кроме конечного числа, содержали 0, и чтобы сумматор также содержал 0. После выполнения каждой команды счетчик команд всегда увеличивается на 2, кроме случаев JUMP i, JGTZ i (при положительном сумматоре) и JZERO i (при нулевом сумматоре), когда он устанавливается на i. Действие каждой команды в точности то же, что и у соответствующей команды РАМ.

Временную сложность РАСП-программы можно определить, по существу, тем же способом, что и для РАМ-программы. Можно использовать либо равномерный весовой критерий, либо логарифмический. В последнем случае, однако, надо приписать вес не только

Команда	Код	Команда	Код
LOAD i	1	DIV i	10
LOAD $=i$	2	DIV $=i$	11
STORE i	3	READ i	12
ADD i	4	WRITE i	13
ADD $=i$	5	WRITE $=i$	14
SUB i	6	JUMP i	15
SUB $=i$	7	JGTZ i	16
MULT i	8	JZERO i	17
MULT $=i$	9	HALT	18

Рис. 1.12. Коды для команд РАСП.

вычислению операнда, но и выбору самой команды. Вес выбора команды равен $l(\text{СК})$, где СК означает содержимое счетчика команд. Например, вес выполнения команды ADD$=i$, хранимой в регистрах j и $j+1$, равен $l(j)+l(c(0))+l(i)$ [1]). Вес команды ADD i, хранимой в регистрах j и $j+1$, равен $l(j)+l(c(0))+l(i)+l(c(i))$.

Интересно узнать, какова разница в сложности между РАМ-программой и соответствующей РАСП-программой. Ответ не будет неожиданным. Независимо от того, взят ли равномерный вес или логарифмический, любое отображение типа вход — выход, выполнимое за время $T(n)$ одной моделью, может выполнить за время $kT(n)$ другая модель, где k — некоторая постоянная. Аналогично объемы памяти, используемые этими моделями, при любой из двух рассматриваемых весовых мер отличаются друг от друга только на постоянный множитель.

Сформулируем эти соотношения в виде двух теорем. Обе теоремы доказываются построением алгоритмов, моделирующих одну машину другой.

Теорема 1.1. *Как при равномерном, так и при логарифмическом весе команд для любой* РАМ*-программы с временной сложностью* $T(n)$ *существует такая постоянная* k*, что найдется эквивалентная* РАСП*-программа, временная сложность которой не превосходит* $kT(n)$.

[1]) Можно было бы также добавить и вес считывания регистра $j+1$, но он не может сильно отличаться от $l(j)$. Во всей этой главе нас не интересует величина мультипликативных постоянных, а только порядок роста функций. Поэтому $l(j)+l(j+1)$ «приблизительно» равняется $l(j)$, т. е. с точностью до множителя, не превышающего 3.

Доказательство. Покажем, как моделировать РАМ-программу некоторой РАСП-программой. Регистр 1 в РАСП будет служить для временного запоминания содержимого сумматора РАМ. Отправляясь от P, мы будем строить РАСП-программу P_s, которая будет занимать следующие $r-1$ регистров РАСП. Постоянная r определяется РАМ-программой P. Содержимое регистра РАМ с номером i, $i \geqslant 1$, будет храниться в регистре РАСП с номером $r+i$, так что все адреса в РАСП-программе будут на r больше соответствующих адресов в РАМ-программе.

Каждая РАМ-команда в P, не содержащая косвенной адресации, прямо кодируется в такую же РАСП-команду (с надлежащим увеличением адресов). Каждая РАМ-команда в P, содержащая косвенную адресацию, переводится в последовательность из шести РАСП-команд, которые моделируют косвенную адресацию путем изменения команд.

Проиллюстрируем моделирование косвенной адресации на примере. Для моделирования РАМ-команды SUB $*i$, где i — положительное целое, построим последовательность РАСП-команд, которые

1) временно запоминают содержимое сумматора в регистре 1,
2) вызывают содержимое регистра $r+i$ в сумматор (РАСП-регистр с номером $r+i$ соответствует РАМ-регистру с номером i),
3) прибавляют r к сумматору,
4) запоминают число, вычисленное на шаге 3 в адресном поле команды SUB,
5) восстанавливают сумматор из временного регистра 1,
6) используют команду SUB, созданную на шаге 4, для выполнения вычитания.

Например, применяя кодирование команд РАСП, приведенное на рис. 1.12, и предполагая, что последовательность РАСП-команд начинается в регистре 100, можно смоделировать SUB $*i$ последовательностью, показанной на рис. 1.13. Сдвиг r можно будет определить, когда станет известно количество РАСП-команд в программе P_s.

Мы видим, что для моделирования каждой РАМ-команды требуется самое большее шесть РАСП-команд, так что при равномерном весовом критерии временная сложность получаемой РАСП-программы не превосходит $6T(n)$. (Заметим, что эта мера не зависит от того, каким способом определен размер входа.)

При логарифмическом весовом критерии каждая команда I из РАМ-программы P моделируется последовательностью S, состоящей в P_s либо из одной, либо из шести РАСП-команд. Можно показать, что существует постоянная k, зависящая от P, такая, что суммарный вес команд в S не более чем в k раз превосходит вес команды I.

Регистр	Содержимое	Значение
100	3	STORE 1
101	1	
102	1	LOAD $r+i$
103	$r+i$	
104	5	ADD $=r$
105	r	
106	3	STORE 111
107	111	
108	1	LOAD 1
109	1	
110	6	SUB b, где b — содержимое i-го
111	—	регистра РАМ

Рис. 1.13. Моделирование SUB $*i$ на РАСП.

Например, команда SUB $*i$ для РАМ имеет вес

$$M = l(c(0)) + l(i) + l(c(i)) + l(c(c(i))).$$

Последовательность S, моделирующая эту команду РАМ, показана на рис. 1.14. Здесь $c(0)$, $c(i)$ и $c(c(i))$ относятся к содержимому регистров РАМ. Так как P_s занимает в РАСП регистры от 2 до r, то $j \leqslant r-11$. Кроме того, $l(x+y) \leqslant l(x)+l(y)$, так что вес S, разуме-

Регистр РАСП	Команда	Вес
j	STORE 1	$l(j)+l(1)+l(c(0))$
$j+2$	LOAD $r+i$	$l(j+2)+l(r+i)+l(c(i))$
$j+4$	ADD $=r$	$l(j+4)+l(c(i))+l(r)$
$j+6$	STORE $j+11$	$l(j+6)+l(j+11)+l(c(i)+r)$
$j+8$	LOAD 1	$l(j+8)+l(1)+l(c(0))$
$j+10$	SUB —	$l(j+10)+l(c(i)+r)+$ $+l(c(0))+l(c(c(i)))$

Рис. 1.14. Веса команд РАСП.

ется, не превосходит

$$2l(1)+4M+11l(r)<(6+11l(r))M.$$

Поэтому можно заключить, что постоянная $k=6+11\,l(r)$ такова, что если P имеет временную сложность $T(n)$, то временная сложность для P_s не превосходит $kT(n)$. □

Теорема 1.2. *Как при равномерном, так и при логарифмическом весе команд для любой* РАСП-*программы с временной сложностью* $T(n)$ *существует такая постоянная* k, *что найдется эквивалентная* РАМ-*программа, временная сложность которой не превосходит* $kT(n)$.

Доказательство. РАМ-программа, которую мы построим для моделирования данной РАСП, будет использовать косвенную адресацию для декодирования и моделирования РАСП-команд, хранящихся в памяти РАМ. Некоторые регистры РАМ будут иметь специальное назначение:

регистр 1 — для косвенной адресации,
регистр 2 — для счетчика команд РАСП,
регистр 3 — для хранения содержимого сумматора РАСП.

РАСП-регистр с номером i будет храниться в РАМ-регистре с номером $i+3$ при $i\geqslant 1$.

Искомая РАМ начинает работу с РАСП-программы конечной длины, расположенной в ее памяти с регистра 4 и далее. Регистр 2 (счетчик команд) содержит число 4; регистры 1 и 3 — число 0. РАМ-программа состоит из цикла моделирования, начинающегося со считывания РАСП-команды (с помощью РАМ-команды LOAD *2), декодирования ее и разветвления на один из 18 наборов команд, каждый из которых предназначен для обработки одного типа РАСП-команды. На неправильном коде операции РАМ, как и РАСП, остановится.

Операции декодирования и разветвления строятся естественным образом; моделью может служить пример 1.2 (хотя символ, декодируемый там, был считан со входа, а здесь он считывается из памяти).

В качестве примера приведем те команды РАМ, которые моделируют РАСП-команду с кодом 6, т. е. SUB i. Эта программа, изображенная на рис. 1.15, вызывается, когда $c(c(2))=6$, т. е. когда счетчик команд указывает на регистр, содержащий число 6 — код команды SUB.

Дальнейшие детали построения нужной РАМ-программы мы опускаем. В качестве упражнения предлагаем доказать, что при равномерном и логарифмическом весовых критериях временная сложность РАМ-программы самое большее в постоянное число раз превосходит временную сложность исходной РАСП-программы. □

LOAD ADD STORE	2 =1 2	Увеличение счетчика команд на 1, так что он начинает указывать на регистр, содержащий операнд i команды SUB i.
LOAD ADD STORE	*2 =3 1	Вызов i в сумматор, прибавление числа 3 и запоминание результата в регистре 1.
LOAD SUB STORE	3 *1 3	Извлечение содержимого сумматора РАСП из регистра 3. Вычитание содержимого регистра $i+3$ и помещение результата обратно в регистр 3.
LOAD ADD STORE	2 =1 2	Увеличение счетчика команд снова на 1, так что теперь он указывает на следующую команду РАСП
JUMP	a	Возвращение к началу цикла моделирования (обозначенному здесь через "a").

Рис. 1.15. Моделирование SUB i на РАМ.

Из теорем 1.1 и 1.2 следует, что в отношении временной сложности (а также и емкостной — это остается в качестве упражнения) модели РАМ и РАСП эквивалентны с точностью до постоянного множителя, т. е. порядки величин их сложностей одинаковы для одного и того же алгоритма. Обычно мы будем выбирать из этих двух моделей модель РАМ, поскольку она проще [1]).

[1]) Значительную часть недостатков РАМ и РАСП, указываемых авторами, можно устранить, если рассмотреть следующую модель, также основанную на принципе адресной организации памяти. *Адресная машина* состоит из бесконечного числа регистров, занумерованных двоичными числами. Первые три регистра служат для специальных целей: *вход*, *выход* и *сумматор*. (Мы рассматриваем лишь модель с хранимой программой.) В регистры можно записывать слова в алфавите {0, 1}. Для определенности выберем систему команд LOAD $=i$, LOAD i, LOAD $*i$, STORE i, STORE $*i$, ADD i, SUB i, SHIFT i (сдвиг содержимого сумматора на число разрядов, равное содержимому регистра i, знак этого содержимого определяет направление сдвига), AND i (поразрядное булево умножение), OR i, EXCLUSIVE OR i, HALT. *Машиной* M будем называть пару (P, l), где $P=p_1, \ldots, p_k$ — программа (т. е. список конкретных команд p_i), а l — функция, ограничивающая длину содержимого регистров: при работе над входом длины n в регистры можно записывать слова длины ровно $l(n)$. Работа машины M над словом w определяется, как обычно: программа P записывается в память машины, начиная с четвертого регистра; при k-м срабатывании команды LOAD *вход* в сумматор записывается k-я компонента входа; результатом работы (если машина остановилась) считается слово, получаемое последовательным приписыванием всех слов, засылавшихся в выходной регистр; если при выполнении какой-то

1.5. МОДИФИКАЦИЯ РАМ

РАМ и РАСП — более сложные модели вычислений, чем это часто бывает необходимо. Поэтому мы введем ряд других моделей, которые наследуют одни черты РАМ и игнорируют другие. Оправданием для них будет то, что суммарный вес игнорируемых команд не превосходит некоторой фиксированной доли веса любого эффективного алгоритма для задач, к которым данная модель применяется.

1. Неветвящиеся программы

Первая наша модель — неветвящаяся программа. Для многих задач разумно ограничиться рассмотрением класса РАМ-программ, в которых команды разветвления используются только для того, чтобы повторить последовательность команд, причем число повторений пропорционально размеру входа n. В этом случае можно "развернуть" программу для каждого размера n, копируя повторяющиеся команды надлежащее число раз. В результате получается последовательность неветвящихся программ (т. е. программ без циклов), вообще говоря, возрастающей длины — по одной программе для каждого значения n.

Пример 1.3. Рассмотрим умножение двух целочисленных матриц размера $n \times n$. Разумно ожидать, что в РАМ-программе число выполнений цикла не будет зависеть от фактических значений элементов матрицы. Поэтому можно в качестве полезного упрощения считать, что допускаются только такие циклы, у которых проверка конца зависит лишь от n, т. е. от размера задачи. Например, обыч-

---

команды получается слово, не помещающееся в регистр, то переполняющая часть этого слова бесследно исчезает. Пусть $t^*_M(w)$ и $s^*_M(w)$ — соответственно число шагов и память при работе M над w, а $t_M(n)$ и $s_M(n)$ — время и память в худшем случае, т. е.

$$t_M(n) = \max_{|w| \leqslant n} t^*_M(w), \quad s_M(n) = \max_{|w| \leqslant n} s^*_M(w).$$

Очевидно, что $\log s_M(n) \leqslant l(n)$ (здесь и ниже через $\log n$ обозначается длина двоичного представления числа n). Разумно ввести ограничение на функцию l:

$$l(n) \leqslant \max\{\max_{1 \leqslant i \leqslant k} |p_i|, \log t_M(n)\}.$$

Первый член, а именно $\max|p_i|$, стоит для того, чтобы программа могла помещаться в память машины естественным образом — одна команда в один регистр. Поскольку обязательно $l(n) \geqslant \log s_M(n)$, то для адресной машины всегда $s_M(n) \leqslant t_M(n)$. Если наложить на $l(n)$ еще и некоторые требования конструируемости (см. гл. 10), например считать, что $l(n)$ можно вычислить на машине с $l(n)$ ячейками без переполнений за время, не большее $2^{l(n)}$, то почти весь материал настоящей книги можно будет основать на понятии адресной машины. К числу важных преимуществ адресной машины по сравнению с РАМ, РАСП и машиной Тьюринга относится возможность в ее терминах точно и достаточно адекватно ставить вопрос о нижних оценках сложности и для задач с заведомо небольшой, например квадратичной, верхней оценкой.— *Прим. перев.*

ный алгоритм умножения матриц содержит циклы, которые следует выполнить точно n раз, при этом от команд разветвления требуется только сравнение параметра цикла с n. □

Развертывание циклов в программе позволяет обходиться без команд разветвления. Оправданием служит то, что во многих задачах не более чем постоянная доля сложности работы РАМ-программы приходится на команды, управляющие циклом. Подобным же образом часто можно допускать, что входные операторы образуют лишь постоянную долю сложности работы программы, и мы устраняем их, допуская, что перед началом выполнения программы в памяти находится конечное множество входов, требуемых при данном n. Действие косвенной адресации можно определить для фиксированного n, если предполагать, что регистры, используемые для нее, содержат величины, зависящие только от n и не зависящие от значений входных переменных. Поэтому мы будем считать, что наши неветвящиеся программы не имеют косвенных адресаций.

Кроме того, поскольку каждая неветвящаяся программа может обращаться только к конечному числу регистров памяти, удобно присвоить этим регистрам имена. Потому при ссылке на регистры мы будем употреблять *символические адреса* (символы или цепочки букв), а не целые числа.

Устранив потребность в командах READ, JUMP, JGTZ и JZERO, мы остаемся с командами LOAD, STORE, WRITE, HALT и арифметическими операциями из системы команд РАМ. Нам не нужна команда HALT, ибо на остановку указывает конец программы. Можно обойтись и без WRITE, назначив в качестве *выходных переменных* определенные символические адреса; выходом программы будет значение, принимаемое этими переменными к окончанию работы программы.

Наконец, можно "встроить" LOAD и STORE в арифметические операции, заменяя последовательности вида

LOAD a
ADD b
STORE c

на $c \leftarrow a+b$. Весь набор команд неветвящейся программы выглядит так:

$$x \leftarrow y+z$$
$$x \leftarrow y-z$$
$$z \leftarrow y*z$$
$$z \leftarrow y/z$$
$$x \leftarrow i$$

где x, y и z — символические адреса (или *переменные*), а i — постоянная. Легко видеть, что любую последовательность команд

LOAD, STORE и арифметических операций в сумматоре можно заменить последовательностью, составленной из пяти выписанных выше команд.

С неветвящейся программой связаны два выделенных набора переменных — *входы* и *выходы*. Функцией, вычисляемой данной неветвящейся программой, называется множество значений выходных переменных (в определенном порядке), выраженных через значения ее входных переменных.

Пример 1.4. Рассмотрим вычисление полинома

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Входными переменными служат коэффициенты a_0, a_1, ..., a_n и неопределенная переменная x. Выходной переменной будет p. По *правилу Горнера* $p(x)$ вычисляется так:

1) $a_1 x + a_0$ для $n=1$,
2) $(a_2 x + a_1)\, x + a_0$ для $n=2$,
3) $((a_3 x + a_2)\, x + a_1)\, x + a_0$ для $n=3$.

На рис. 1.16 приведены неветвящиеся программы, соответствующие этим выражениям. Правило Горнера для произвольного n теперь должно быть понятно. Для каждого n у нас есть неветвящаяся программа из $2n$ шагов, вычисляющая полином n-й степени. В гл. 12 мы покажем, что для вычисления произвольного полинома n-й степени по его коэффициентам требуется n умножений и n сложений. Таким образом, если в качестве модели брать неветвящиеся программы, правило Горнера оптимально. □

Если брать в качестве модели вычислений неветвящуюся программу, то временная сложность последовательности программ равна числу шагов n-й программы как функции от n. Например, правило Горнера порождает последовательность с временной слож-

$n=1$	$n=2$	$n=3$
$t \leftarrow a_1 * x$	$t \leftarrow a_2 * x$	$t \leftarrow a_3 * x$
$p \leftarrow t + a_0$	$t \leftarrow t + a_1$	$t \leftarrow t + a_2$
	$t \leftarrow t * x$	$t \leftarrow t * x$
	$p \leftarrow t + a_0$	$t \leftarrow t + a_1$
		$t \leftarrow t * x$
		$p \leftarrow t + a_0$

Рис. 1.16. Неветвящиеся программы, соответствующие правилу Горнера.

ностью $2n$. Заметим, что измерение временной сложности есть не что иное, как подсчет числа арифметических операций. Емкостная сложность последовательности программ равна числу переменных, участвовавших в программе, снова как функций от n. Программы примера 1.4 имеют емкостную сложность $n+4$.

Определение. В случае когда в качестве модели вычислений берутся неветвящиеся программы, говорят, что данная задача имеет временную или емкостную сложность $O_A(f(n))$, если найдется последовательность программ для ее решения с временной или емкостной сложностью не более $cf(n)$ для некоторой постоянной c. ($O_A(f(n))$ читается так: "порядка $f(n)$ шагов неветвящейся программы". Индекс А снизу обозначает "арифметический" — это основная характеристика неветвящихся программ.) Таким образом, вычисление полинома имеет временную, а также и емкостную сложность $O_A(n)$.

2. Битовые вычисления

Очевидно, что модель неветвящихся программ основана на равномерной весовой функции. Как мы уже отмечали, этот вес годится в предположении, что все вычисляемые величины имеют "разумный" размер. Существует простая модификация модели неветвящихся программ, которая соответствует логарифмической весовой функции. Эта модель, называемая *битовым вычислением*, по существу, является той же неветвящейся программой, но только в ней

1) все переменные принимают значения 0 или 1, т. е. это биты,
2) используются логические операции вместо арифметических [1]) (**and** обозначается через $\wedge$, **or** — через $\vee$, **exclusive or** — через $\oplus$, **not** — через $\neg$).

Для битовой модели арифметические операции над целыми числами i и j занимают по меньшей мере $l(i)+l(j)$ шагов, что соответствует логарифмическому весу операндов. В самом деле, умножение и деление с помощью наилучших известных алгоритмов для умножения или деления i на j требуют более чем $l(i)+l(j)$ шагов.

При битовых вычислениях порядок величин обозначается через $O_Б$. Битовая модель полезна, когда речь идет об основных операциях, таких, как арифметические, которые исходны в других моделях. Например, для модели неветвящихся программ умножение двух n-разрядных двоичных целых чисел можно осуществить за $O_A(1)$ шагов, тогда как для битовой модели наилучший известный результат — это $O_Б(n \log n \log \log n)$ шагов.

Другое применение битовой модели — логические сети (схемы). Неветвящиеся программы с двоичными входами и операциями вза-

[1]) Таким образом, набор команд для наших РАМ должен состоять из этих операций.

имно однозначно соответствуют комбинационным логическим сетям, вычисляющим набор булевых функций. Число шагов такой программы — это число логических элементов в сети.

Пример 1.5. На рис. 1.17,*а* приведена программа для сложения двух двуразрядных чисел $[a_1a_0]$ и $[b_1\ b_0]$. Выходные переменные — это такие числа c_2, c_1 и c_0, что $[a_1\ a_0]+[b_1b_0]=[c_2\,c_1\,c_0]$. Неветвящаяся программа на рис. 1.17, *а* вычисляет

$$c_0 = a_0 \oplus b_0,$$
$$c_1 = ((a_0 \wedge b_0) \oplus a_1) \oplus b_1,$$
$$c_2 = ((a_0 \wedge b_0) \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1).$$

На рис. 1.17,*б* изображена соответствующая логическая сеть. В качестве упражнения предлагаем показать, что сложение двух n-разрядных чисел можно выполнить за $O_{\text{Б}}(n)$ шагов. □

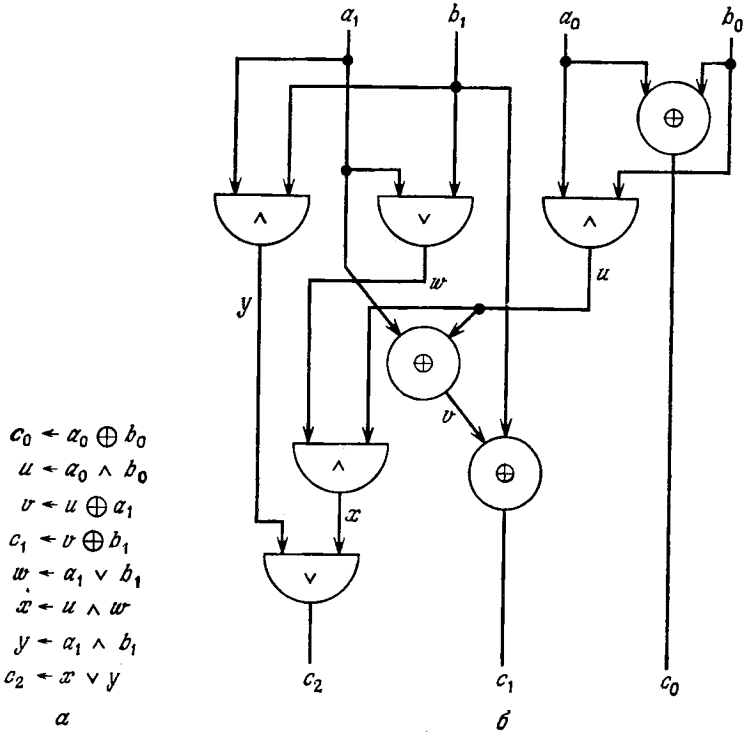


Рис. 1.17. *а* — битовая программа для сложения; *б* — эквивалентная логическая сеть.

3. Операции с двоичными векторами

Можно было бы не ограничивать значения переменных символами 0 и 1, а разрешить переменным принимать в качестве значения любой вектор из 0 и 1. Фактически двоичные векторы фиксированной длины очевидным образом соответствуют целым числам, так что здесь не допускается ничего такого, что не допускалось бы в РАМ, т. е., когда это удобно, просто разрешаются регистры неограниченного размера.

Однако, как мы увидим, в тех немногих алгоритмах, где применяется модель с двоичными векторами, длина векторов будет значительно больше числа битов, требуемых для представления размера задачи. Величина большинства целых чисел, фигурирующих в таком алгоритме, будет того же порядка, что и размер задачи. Например, решая задачу выбора пути в графе со 100 узлами можно было бы для представления наличия или отсутствия пути из данного узла v в каждый из узлов использовать двоичные векторы длины 100, а именно i-ю позицию в векторе для узла v занимает 1 тогда и только тогда, когда существует путь из v в v_i. В этой же задаче можно также использовать целые числа для счета и индексации, например, и они, вероятно, были бы размера числа 100. Таким образом, для целых чисел требовалось бы 7 битов, тогда как для векторов — 100 битов.

Хотя это сравнение и бросает некоторую тень на вычисления с двоичными векторами, большинство вычислительных машин выполняют логические операции на двоичных векторах, составляющих полное машинное слово, за одну команду. Поэтому двоичные векторы длины 100 можно было бы обрабатывать за три или четыре шага (вместо одного для чисел). Тем не менее на результаты о временной и емкостной сложностей алгоритмов при применении модели с двоичными векторами, мы должны смотреть *cum grano salis* [1]), ибо размер задачи, при котором модель становится нереалистичной, в этом случае много меньше, чем в случае моделей РАМ и неветвящихся программ. Порядок величин при применении модели с двоичными векторами мы будем обозначать через $O_{\text{дв}}$.

4. Деревья решений

Мы рассмотрели три модификации РАМ, игнорирующие команды разветвления и учитывающие только те шаги программы, которые включают арифметический счет. В некоторых задачах удобно в качестве основной меры сложности брать число выполняемых команд разветвления В случае сортировки, например, выходы совпадают

[1]) Буквально (с *лат.*) — с крупинкой соли; в переносном смысле — с иронией, язвительно.— *Прим. ред.*

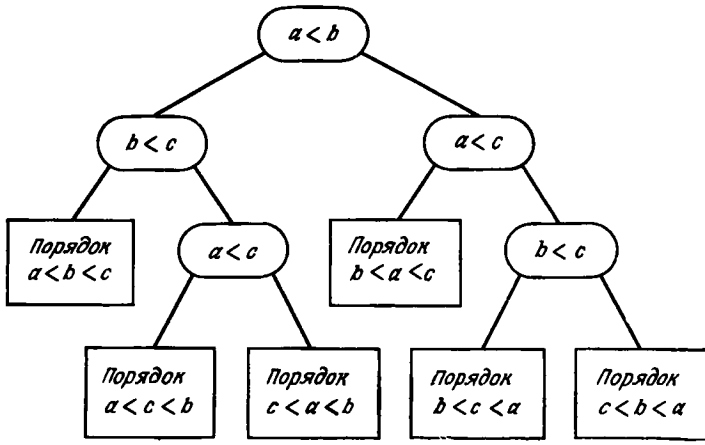


Рис. 1.18. Дерево решений.

со входами с точностью до порядка. Поэтому разумно рассматривать модель, в которой все шаги дают разветвления, возникающие в результате сравнения двух величин.

Обычно программу, состоящую из разветвлений, представляют в виде двоичного дерева [1]), называемого *деревом решений*. Каждый внутренний узел представляет один из шагов решения. Тест, представленный корнем, выполняется первым, и затем "управление" передается одному из его сыновей в зависимости от исхода теста. В общем случае управление переходит от узла к одному из его сыновей (причем выбор в каждом случае зависит от исхода теста в этом узле), до тех пор пока не будет достигнут лист. Нужный выход находится на достигнутом листе.

Пример 1.6. На рис. 1.18 изображено дерево решений для программы, сортирующей три числа a, b и c. Тесты указаны заключенными в овал сравнениями в узлах; управление переходит влево, если ответ на тест — "да", и вправо, если — "нет". □

Временная сложность дерева решений равна высоте этого дерева как функции размера задачи. Обычно мы хотим измерить наибольшее число сравнений, которые приходится делать, чтобы найти нужный путь от корня к листу. Порядок величин при использовании модели деревьев решений (сравнений) мы будем обозначать через O_C. Заметим, что общее число узлов в дереве может значительно превосходить его высоту. Например, дерево решений для сортировки n чисел должно содержать по крайней мере $n!$ листьев, хотя его высота может быть $n \log n$.

[1]) По поводу определений, касающихся деревьев, см. разд. 2.4.

1.6. ПРОСТЕЙШАЯ МОДЕЛЬ ВЫЧИСЛЕНИЙ: МАШИНА ТЬЮРИНГА

Для доказательства того, что для вычисления данной функции требуется какое-то минимальное время, нужна некоторая модель, столь же общая, как те модели, которые у нас были, но более простая. Система команд должна быть ограничена, насколько возможно, хотя эта модель должна быть в состоянии не только вычислить все то, что может вычислить РАМ, но и сделать это "почти" так же быстро. Под словом "почти" мы будем подразумевать "полиномиальную связанность".

Определение. Говорят, что неотрицательные функции $f_1(n)$ и $f_2(n)$ *полиномиально связаны* (*эквивалентны*), если найдутся такие полиномы $p_1(x)$ и $p_2(x)$, что для всех n справедливы неравенства $f_1(n) \leqslant p_1(f_2(n))$ и $f_2(n) \leqslant p_2(f_1(n))$.

Пример 1.7. Две функции $f_1(n)=2n^2$ и $f_2(n)=n^5$ полиномиально связаны: можно взять $p_1(x)=2x$, ибо $2n^2 \leqslant 2n^5$, и $p_2(x)=x^3$, ибо $n^5 \leqslant (2n^2)^3$. Но n^2 и 2^n не являются полиномиально связанными, так как нет такого полинома $p(x)$, что $p(n^2) \geqslant 2^n$ для всех n. □

В настоящее время единственный класс функций, для которых мы можем применить такие общие вычислительные модели, как машины Тьюринга, чтобы получить нижние оценки вычислительной сложности, составляют "быстро растущие" функции. Например, в гл. 11 будет показано, что некоторые задачи требуют экспоненциальные время и память. (Функция $f(n)$ называется *экспоненциальной*, если существуют такие постоянные $c_1>0$, $k_1>1$, $c_2>0$ и $k_2>1$, что $c_1 k_1^n \leqslant f(n) \leqslant c_2 k_2^n$ для всех, кроме конечного числа, значений n.) Относительно полиномиальной связанности все экспоненциальные функции, по существу, одинаковы; любая функция, полиномиально связанная с экспоненциальной, сама экспоненциальна.

Таким образом, это побуждает нас использовать простую модель, для которой временная сложность задач полиномиально связана с их сложностью на РАМ. В частности, модель, которую мы будем применять, а именно многоленточная машина Тьюринга, может потребовать $(f(n))^4$ шагов [1]), чтобы сделать то, что РАМ при логарифмической весовой функции делает за $f(n)$ шагов, но не больше. Итак, временные сложности на РАМ и машинах Тьюринга, как мы увидим, полиномиально связаны.

Определение. *Многоленточная машина Тьюринга* (МТ) изображена на рис. 1.19. Она состоит из нескольких, скажем k, *лент*,

[1]) На самом деле верна более низкая оценка $O([f(n)\log f(n) \log\log f(n)]^2)$, но мы интересуемся оценками с точностью до полиномов и нас вполне устроит результат с четвертой степенью (см. разд. 7.5).

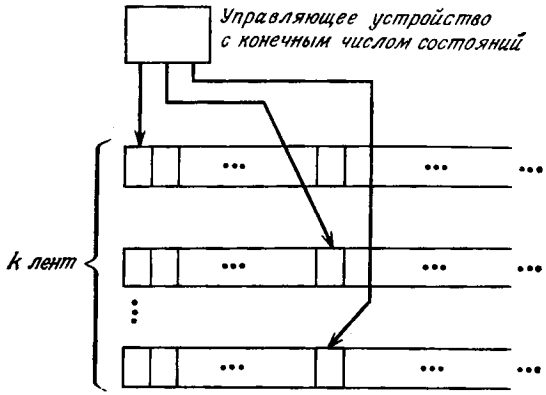


Рис. 1.19. Многоленточная машина Тьюринга.

бесконечно простирающихся вправо. Каждая лента разбита на *клетки*, каждая из которых содержит один из конечного числа *символов на ленте*. Одна клетка на каждой ленте обозревается *головкой* этой ленты; головка может считывать с ленты и записывать на нее. Работа машины Тьюринга определяется простой программой, называемой *управляющим устройством*. Оно всегда находится в одном из конечного числа *состояний*, которое можно рассматривать как номер текущей команды в программе.

Один шаг вычисления на машине Тьюринга состоит в следующем. В соответствии с текущим состоянием управляющего устройства и символами на лентах, *обозреваемыми* (т. е. находящимися под) каждой из головок, машина Тьюринга может выполнить некоторые или все из следующих операций:

1) изменить состояние управляющего устройства,
2) напечатать новые символы на лентах вместо старых в каких-нибудь или во всех клетках под головками,
3) сдвинуть какие-нибудь или все головки независимо друг от друга на одну клетку влево (L) или вправо (R) либо оставить на месте (S).

Формально k-ленточная машина Тьюринга задается семеркой

$$(Q,\ T,\ I,\ \delta,\ \mathrm{b},\ q_0,\ q_f),$$

где

1) Q — множество *состояний*,
2) T — множество *символов на лентах*,
3) I — множество *входных символов*, $I \subseteq T$,
4) b — *пустой символ*, $\mathrm{b} \in T-I$,
5) q_0 — *начальное состояние*,

6) q_f — *заключительное* (или *допускающее*) *состояние*,

7) δ — *функция переходов*, она отображает некоторое подмножество множества $Q \times T^k$ в $Q \times (T \times \{L, R, S\})^k$, т. е. по произвольному набору из состояния и k символов на лентах она выдает новое состояние и k пар, каждая из которых состоит из нового символа на ленте и направления сдвига головки.

Пусть $\delta(q_1, a_1, a_2, \ldots, a_k) = (q', (a_1', d_1), (a_2', d_2), \ldots, (a_k', d_k))$ и машина Тьюринга находится в состоянии q, а ее головка на i-й ленте обозревает символ a_i, $1 \leqslant i \leqslant k$. Тогда за один шаг эта машина Тьюринга переходит в состояние q', заменяет a_i на a_i' и сдвигает головку на i-й ленте в направлении (или в соответствии с) d_i, $1 \leqslant i \leqslant k$.

Машину Тьюринга можно приспособить для распознавания языка. Символы на лентах такой машины включают алфавит рассматриваемого языка (его буквы играют роль *входных символов*), пустой символ, обозначаемый b, и, возможно, другие символы. Вначале на первой ленте записано слово из входных символов по одному на клетку, начиная с самой левой. Все клетки справа от клеток, содержащих входное слово, пусты. Все остальные ленты целиком пусты. Слово из входных символов *допускается (воспринимается)* тогда и только тогда, когда машина Тьюринга, начав

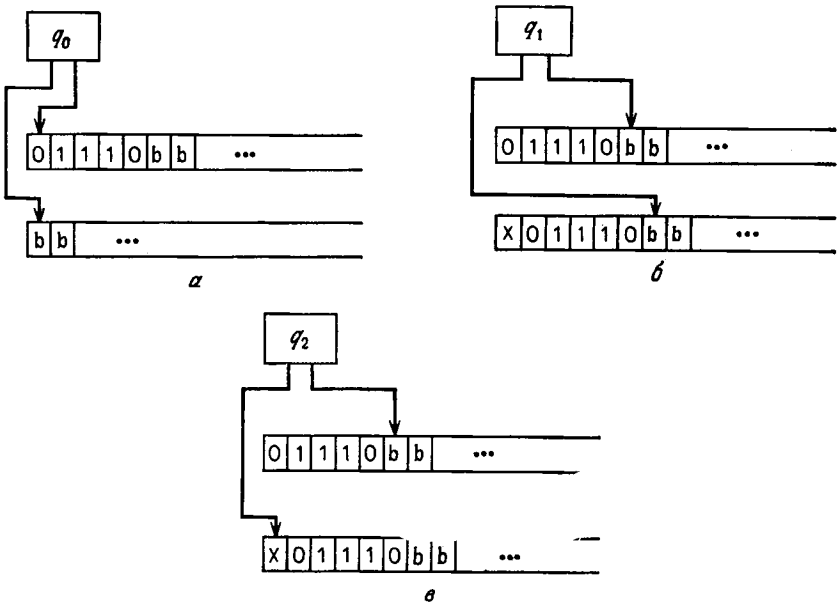


Рис. 1.20. Работа машины Тьюринга над цепочкой 01110.

работу в выделенном начальном состоянии (головки при этом находились на левых концах своих лент), сделает последовательность шагов, которые в конце концов приведут ее в допускающее состояние. *Языком, допускаемым* данной машиной Тьюринга, называется множество всех слов из входных символов, допускаемых в описанном только что смысле.

Пример 1.8. Двухленточная машина Тьюринга на рис. 1.20 распознает палиндромы [1]) в алфавите {0, 1} следующим образом.

1) Первая клетка на ленте 2 отмечается специальным знаком $\times$, и вход копируется с ленты 1, где он записан вначале (рис. 1.20, *а*), на ленту 2 (рис. 1.20, *б*).
2) Затем головка на ленте 2 сдвигается к $\times$ (рис. 1.20, *в*).
3) Повторяется такая процедура: головка на ленте 2 сдвигается вправо на одну клетку, а на ленте 1 — влево на одну клетку и соответствующие символы сравниваются. Если все символы совпадают, то вход является палиндромом и машина Тьюринга доходит до допускающего состояния q_5. В противном случае в некоторый момент очередной шаг машины Тьюринга будет не определен, а она остановится, не допустив входного слова.

Функция переходов соответствующей машины Тьюринга приведена на рис. 1.21. □

Работу машины Тьюринга формально можно описать с помощью "мгновенных описаний". *Мгновенным описанием* (МО) k-ленточной машины Тьюринга M называется набор $(\alpha_1, \alpha_2, \ldots, \alpha_k)$, где α_i для каждого i представляет собой слово вида xqy, причем xy — слово на i-й ленте машины M (пустые символы, стоящие справа от его правого конца, опускаются), а q — текущее состояние машины. Головка на i-й ленте обозревает символ, стоящий справа от q.

Если мгновенное описание D_1 переходит в мгновенное описание D_2 за один шаг машины Тьюринга M, то пишут $D_1 \vdash_M D_2$ (знак $\vdash$ читается "переходит в"). Если $D_1 \vdash_M D_2 \vdash_M \ldots \vdash_M D_n$ для некоторого $n \geqslant 2$, то пишут $D_1 \vdash_M^+ D_n$. Если либо $D = D'$, либо $D \vdash_M^+ D'$, то пишут $D \vdash_M^* D'$.

Данная k-ленточная машина Тьюринга $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ *допускает* слово $a_1 a_2 \ldots a_n$, где a_i — элементы из I, если $(q_0 a_1 a_2 \ldots a_n, q_0, q_0, \ldots, q_0) \vdash_M^* (\alpha_1, \alpha_2, \ldots, \alpha_k)$ для некоторых α_i, содержащих q_f.

Пример 1.9. На рис. 1.22 приведена последовательность мгновенных описаний машины Тьюринга, изображенной на рис. 1.21, ко-

[1]) *Палиндромом* называется слово, совпадающее с самим собой при чтении с конца, например 0100010.

Текущее состояние	Символ на ленте 1	Символ на ленте 2	Новый символ, сдвиг головки: лента 1	Новый символ, сдвиг головки: лента 2	Новое состояние	Примечания
q_0	0 1 b	b b b	0, S 1, S b, S	×, R ×, R b, S	q_1 q_1 q_5	Если вход непуст, то на ленте 2 печатается × и головка сдвигается вправо; машина переходит в состояние q_1. В противном случае машина переходит в состояние q_5.
q_1	0 1 b	b b b	0, R 1, R b, S	0, R 1, R b, L	q_1 q_1 q_2	Машина остается в состоянии q_1 и переписывает ленту 1 на ленту 2, пока на ленте 1 не встретится b. Затем она переходит в состояние q_2.
q_2	b b b	0 1 ×	b, S b, S b, L	0, L 1, L ×, R	q_2 q_2 q_3	Головка на ленте 1 остается на месте, а на ленте 2 движется влево до × Затем машина переходит в состояние q_3.
q_3	0 1	0 1	0, S 1, S	0, R 1, R	q_4 q_4	Состояния q_3 и q_4 чередуются. В q_3 сравниваются символы на обеих лентах, головка на ленте 2 сдвигается вправо и машина переходит в q_4. Из q_4 она переходит в q_5 и допускает входное слово, если головка достигла b на ленте 2. Если же не достигла, то головка на ленте 1 сдвигается влево и машина возвращается в состояние q_3. Чередование q_3 и q_4 предотвращает уход головки на ленте 1 за ее левый конец.
q_4	0 0 1 1 0 1	0 1 0 1 b b	0, L 0, L 1, L 1, L 0, S 1, S	0, S 1, S 0, S 1, S b, S b, S	q_3 q_3 q_3 q_3 q_5 q_5	
q_5						Входное слово допускается

Рис. 1.21. Функция переходов машины Тьюринга распознающей палиндромы.

$$
\begin{aligned}
(q_0010,\ q_0) &\vdash (q_1010,\ \times q_1)\\
&\vdash (0q_110,\ \times 0q_1)\\
&\vdash (01q_10,\ \times 01q_1)\\
&\vdash (010q_1,\ \times 010q_1)\\
&\vdash (010q_2,\ \times 01q_20)\\
&\vdash (010q_2,\ \times 0q_210)\\
&\vdash (010q_2,\ \times q_2010)\\
&\vdash (010q_2,\ q_2\times 010)\\
&\vdash (01q_30,\ \times q_3010)\\
&\vdash (01q_40,\ \times 0q_410)\\
&\vdash (0q_310,\ \times 0q_310)\\
&\vdash (0q_410,\ \times 01q_40)\\
&\vdash (q_3010,\ \times 01q_30)\\
&\vdash (q_4010,\ \times 010q_4)\\
&\vdash (q_5010,\ \times 010q_5)
\end{aligned}
$$

Рис. 1.22. Последовательность МО машины Тьюринга.

торой подано на вход слово 010. Так как q_5— заключительное состояние, эта машина Тьюринга допускает 010. □

В дополнение к естественной интерпретации машины Тьюринга как устройства, допускающего какой-то язык, ее можно рассматривать как устройство, которое вычисляет некоторую функцию f. Аргументы этой функции кодируются на входной ленте в виде слова x со специальным маркером #, отделяющим их друг от друга. Если машина Тьюринга останавливается, имея на ленте, выделенной в качестве *выходной*, целое число y (значение функции), то полагают $f(x)=y$. Таким образом, процесс вычисления мало отличается от процесса допускания языка.

Временная сложность $T(n)$ машины Тьюринга M равна наибольшему числу шагов, сделанных ею при обработке входа длины n (для всех входов длины n). Если на каком-нибудь входе длины n машина Тьюринга не останавливается, то для этого n значение $T(n)$ не определено.

Емкостная сложность $S(n)$ машины Тьюринга равна наибольшему расстоянию от левого конца ленты, которое должна пройти головка при обработке входа длины n. Если головка на какой-то ленте неопределенно долго движется вправо, то функция $S(n)$ не определена. Порядок величин при применении в качестве модели машины Тьюринга мы будем обозначать через $O_{\text{МГ}}$.

Пример 1.10. Временная сложность машины Тьюринга, изображенной на рис. 1.20, равна $T(n)=4n+3$, а ее емкостная сложность равна $S(n)=n+2$. Это можно проверить, исследовав случай, когда вход на самом деле является палиндромом. □

1.7. СВЯЗЬ МАШИН ТЬЮРИНГА И РАМ

Основное применение машины Тьюринга (МТ) находят в установлении нижних оценок на время и память, необходимые для решения данной задачи. В большинстве случаев мы можем установить нижние оценки только с точностью до полиномиальной связанности. Для более точных оценок потребуется рассматривать более специфические детали конкретных моделей. К счастью, вычисления на РАМ и РАСП полиномиально связаны с вычислениями на МТ.

Рассмотрим связь между РАМ и МТ. Очевидно, что РАМ может моделировать работу k-ленточной МТ, помещая по одной клетке МТ в регистр. В частности, i-ю клетку j-й ленты можно хранить в регистре $ki+j+c$, где c — постоянная, назначение которой — дать РАМ некоторое "оперативное пространство". В него включаются k регистров для запоминания положений k головок МТ. РАМ может считывать клетки МТ, используя косвенную адресацию с помощью регистров, содержащих положения головок на лентах.

Предположим, что данная МТ имеет временную сложность $T(n)\geqslant n$. Тогда РАМ может прочитать ее вход, запомнить его в регистрах, представляющих первую ленту, и смоделировать эту МТ за время $O(T(n))$ при равномерном весовом критерии и $O(T(n)\log T(n))$ при логарифмическом. В любом случае время на РАМ ограничено сверху полиномом от времени МТ, ибо любая функция типа $O(T(n)\log n)$ есть, разумеется, $O(T^2(n))$.

Обратное утверждение верно только при логарифмическом весовом критерии для РАМ. При равномерном весе n-шаговая программа для РАМ без входа может вычислять числа порядка 2^{2^n}, что требует порядка 2^n клеток МТ только для запоминания и считывания. Поэтому при равномерном весе, очевидно, нет полиномиальной связи между РАМ и МТ (упр. 1.19).

Хотя при анализе алгоритмов мы предпочитаем равномерный весовой критерий в силу его простоты, мы должны отвергнуть его, если пытаемся установить нижние границы на временную сложность. РАМ с равномерным весом вполне разумна, когда рост чисел по порядку не превосходит размера задачи. Но, как мы уже говорили, при использовании РАМ в качестве модели размер чисел "заметается под ковер", и вряд ли можно получить полезные нижние оценки. Для логарифмического веса, однако, верна следующая теорема.

Теорема 1.3. *Пусть L — язык, допускаемый некоторой* РАМ *за время $T(n)$ при логарифмическом весе. Если в* РАМ-*программе*

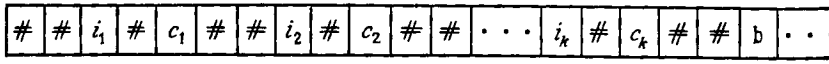


Рис. 1.23. Представление РАМ на МТ.

нет умножений и делений, то на многоленточных машинах Тьюринга L имеет временную сложность не более $O(T^2(n))$.

Доказательство. Представим все не содержащие нуля регистры рассматриваемой РАМ, как показано на рис. 1.23. На ленте помещена последовательность пар чисел (i_j, c_j), записанных в двоичной форме без нулей в начале слова и разделенных маркерами. Для каждого j число c_j есть содержимое регистра i_j РАМ. Содержимое сумматора РАМ хранится в двоичном коде на второй ленте, а третья играет роль рабочей памяти. Еще две ленты служат для записи входа и выхода РАМ. Каждый шаг программы РАМ представлен конечным числом состояний МТ. Мы не будем описывать моделирование всех команд РАМ, а рассмотрим только команды ADD *20 и STORE 30, которые разъясняют общую идею. Для ADD *20 можно устроить МТ так, чтобы она работала следующим образом.

1. На ленте 1 разыскивается место, соответствующее регистру 20 в РАМ, т. е. последовательность ##10100#. Если оно находится, на ленту 3 помещается следующее за ним число, которое будет содержимым регистра 20. Если такое место не нашлось, машина останавливается — содержимое регистра 20 равно 0, и поэтому косвенную адресацию произвести нельзя.
2. На ленте 1 разыскивается место, где хранится регистр РАМ, номер которого записан на ленте 3. Если оно находится, то содержимое этого регистра записывается на ленту 3. Если нет, туда помещается 0.
3. Число, записанное на ленту 3 на шаге 2, прибавляется к содержимому сумматора, которое хранится на ленте 2.

Для моделирования STORE 30 можно так построить МТ, чтобы она работала следующим образом.

1. Разыскивается место расположения регистра 30 в РАМ, т. е. ##11110#.
2. Если оно находится, то на ленту 3 записывается все, что расположено справа от ##11110#, кроме числа, стоящего непосредственно справа (т. е. старого содержимого регистра 30). Затем содержимое сумматора (на ленте 2) записывается непосредственно справа от ##11110#, а за ним помещается слово, скопированное на ленту 3.

3. Если на ленте 1 не нашлось места, соответствующего регистру 30, то, дойдя до самого левого пустого символа, машина печатает 11110#, затем содержимое сумматора и, наконец, ##.

Подумав немного, нетрудно понять, что можно указать МТ, которая правильно смоделирует РАМ. Мы должны показать, что вычисление на РАМ с логарифмическим весом k потребует не более $O(k^2)$ шагов на машине Тьюринга. Сначала заметим, что регистр может появиться на ленте 1, только если его текущее значение когда-то раньше было помещено в этот регистр. Вес записи c_j в регистр i_j равен $l(c_j)+l(i_j)$, что с точностью до постоянной равно длине нашего представления $\#\#i_j\#c_j\#\#$. Отсюда мы заключаем, что длина непустой части ленты 1 есть $O(k)$.

Моделирование любой команды, отличной от STORE, имеет порядок длины ленты 1, т. е. $O(k)$, ибо основное время уходит на поиск на ленте. Аналогично время моделирования STORE не больше суммы времен поиска на ленте 1 и копирования ее — все вместе $O(k)$. Следовательно, одну команду РАМ (кроме умножения и деления) можно промоделировать не более чем за $O(k)$ шагов МТ. Так как одна команда РАМ занимает при логарифмическом весе по крайней мере одну единицу времени, общее время, затрачиваемое соответствующей МТ, есть $O(k^2)$, что и требовалось доказать. □

Если в РАМ-программе участвуют команды умножения и деления, то можно написать подпрограммы для МТ, выполняющие эти операции с помощью сложений и вычитаний. Читателю предоставляем доказать, что логарифмические веса этих подпрограмм не больше квадрата логарифмических весов соответствующих команд. Таким образом, нетрудно доказать следующую теорему.

Теорема 1.4. РАМ *и* РАСП *с логарифмическим весом и многоленточные машины Тьюринга полиномиально связаны.*

Доказательство. Примените теоремы 1.1—1.3 и проанализируйте подпрограммы для умножения и деления. □

Аналогичный результат справедлив и для емкостной сложности хотя он представляется менее интересным.

1.8. ЯЗЫК ВЫСОКОГО УРОВНЯ — УПРОЩЕННЫЙ АЛГОЛ

Наши основные меры сложности определяются в терминах операций для РАМ, РАСП и машин Тьюринга, но мы, вообще говоря, не хотим описывать алгоритмы в терминах столь простых машин, да в этом и нет необходимости. Для того чтобы нагляднее описывать алгоритмы, мы введем некоторый язык высокого уровня, называемый Упрощенным Алголом.

Программу на Упрощенном Алголе можно перевести непосредственно в программу для РАМ или РАСП. Фактически это была бы

в точности роль транслятора с Упрощенного Алгола. Нас, однако, не будут интересовать детали перевода Упрощенного Алгола в конкретные программы для РАМ или РАСП. Для наших целей нужно рассматривать лишь время и память, необходимые для выполнения команд, соответствующих операторам на Упрощенном Алголе.

Упрощенный Алгол отличается от всех принятых языков программирования тем, что он разрешает использовать любой тип математических предписаний, если только их значения понятны, а перевод в команды РАМ или РАСП очевиден. Этот язык не имеет также фиксированного набора типов данных. Переменные могут представлять целые числа, слова и массивы. Дополнительные типы данных — множества, графы, списки, очереди и т. п. — можно вводить по мере необходимости. Формальные описания типов данных по возможности избегаются. Тип данных какой-то переменной и ее область действия [1]) должны быть ясны либо по ее названию, либо по контексту.

В Упрощенном Алголе применяются традиционные конструкции математики и языков программирования, такие, как выражения, условия, операторы и процедуры. Ниже приведены неформальные описания некоторых из этих конструкций. Никаких попыток дать точное определение не делается, ибо это выходит далеко за рамки тематики данной книги. Заметим, что легко написать программы, смысл которых зависит от деталей, не рассмотренных здесь, но лучше избегать этого, что мы и проделали (мы надеемся) в нашей книге.

Программа на Упрощенном Алголе — это оператор одного из следующих типов:

1) переменная ← выражение
2) **if** условие **then** оператор **else** оператор [2])
3а) **while** условие **do** оператор
3б) **repeat** оператор **until** условие
4) **for** переменная ← исходное значение **step** размер шага [3]) **until** заключительное значение **do** оператор
5) метка: оператор
6) **goto** метка
7) **begin**
   оператор;
   оператор;
   .
   .
   .

[1]) *Область действия* переменной — это окружение, в котором она осмыслена. Например, областью действия индекса суммирования является выражение, стоящее под знаком суммы, и вне его он не имеет значения.

[2]) Часть «**else** оператор» не обязательна. Но такой вариант приводит к обычной двусмысленности типа «непривязанное **else**». Мы выбираем традиционный путь и предполагаем, что **else** спаривается с ближайшим еще не спаренным **then**.

[3]) Часть «**step** размер шага» не обязательна, если размер шага равен 1.

.
.
.
оператор;
оператор
end

8а) **procedure** имя (список параметров): оператор
8б) **return** выражение
8в) имя процедуры (аргументы)
9а) **read** переменная
9б) **write** выражение
10) **comment** комментарий
11) любой другой произвольный оператор

Дадим обзор каждого из этих операторов.

1. Оператор присваивания

переменная ← выражение

означает, что надо вычислить выражение справа от стрелки и его значение присвоить переменной, стоящей слева от стрелки. Временная сложность оператора присваивания определяется временем, затрачиваемым на вычисление значения выражения и присваивание этого значения переменной. Если значение выражения не принадлежит к данным основного типа, таким как целые числа, то в некоторых случаях можно уменьшить время с помощью указателей. Например, присваивание $A \leftarrow B$, где A и B — матрицы размера $n \times n$, обычно потребовало бы $O(n^2)$ шагов. Но если B больше нигде не встречается, то путем простого переименования массива можно сделать это время фиксированным, не зависящим от n.

2. В **if**-операторе

if условие **then** оператор **else** оператор

условием, следующим за **if**, может быть любое выражение, принимающее значение **true** или **false**. Если это условие имеет значение **true**, то надо выполнять оператор, стоящий за **then**. В противном случае надо выполнять оператор, стоящий за **else** (если **else** есть). Вес **if**-оператора равен сумме весов, требуемых для вычисления значения и проверки его, и веса оператора, стоящего сразу за **then**, или оператора, стоящего за **else**, в зависимости от того, какой из них выполнялся на самом деле.

3. Назначение **while**-оператора

while условие **do** оператор

и **repeat**-оператора

repeat оператор **until** условие

состоит в организации цикла. В **while**-операторе вычисляется значение условия, идущего после **while**. Если оно истинно (принимает значение **true**), то выполняется оператор, стоящий после **do**. Этот процесс повторяется до тех пор, пока условие не станет ложным. Если вначале это условие было истинным, то выполнение оператора должно в конце концов привести это условие к значению **false**, чтобы закончилось выполнение **while**-оператора. Для вычисления веса **while**-оператора суммируются веса всех проверок условия и всех выполненных операторов.

repeat-оператор трактуется аналогично, но только теперь оператор, стоящий за **repeat**, выполняется перед проверкой условия.

4. В **for**-операторе

for переменная ← исходное значение **step** размер шага
until заключительное значение **do** оператор

"исходное значение", "размер шага" и "заключительное значение" являются выражениями. В случае когда размер шага положителен, "переменная" (называемая *индексом*) полагается равной значению выражения для исходного значения. Если оно больше заключительного значения, то выполнение оператора заканчивается. В противном случае выполняется оператор, стоящий после **do**, значение переменной увеличивается на величину шага и сравнивается с заключительным значением. Процесс повторяется до тех пор, пока значение переменной не превзойдет заключительное значение. Случай, когда размер шага отрицателен, трактуется аналогично с той лишь разницей, что окончание происходит, когда значение переменной становится меньше заключительного значения. Вес **for**-оператора должен быть очевиден в свете предшествующего анализа **while**-оператора.

В приведенном описании совершенно игнорируется такая деталь, как момент вычисления выражений для исходного значения, размера шага и заключительного значения. Может случиться, что выполнение оператора, стоящего после **do**, изменяет значение выражения для размера шага. В таком случае вычисление значения выражения для размера шага каждый раз, когда переменная возрастает, может привести к результату, отличному от того, который получился бы, если бы размер шага вычислялся раз и навсегда. Точно так же вычисление размера шага может воздействовать на заключительное значение, а изменение размера шага может повлиять на тест на окончание. Мы обходим эти трудности, отказываясь от программ, в которых подобные явления сделали бы их смысл неясным.

5. Любой оператор можно переделать в *помеченный оператор*, написав перед ним метку, за которой стоит двоеточие. Главное назначение метки — создание цели для **goto**-оператора. Меткам не приписывается никакого веса.

6. **goto**-оператор

goto метка

указывает, что дальше выполняется оператор с данной меткой. Этот оператор не может стоять внутри блока типа 7, если сам **goto**-оператор не находится в том же блоке. Вес **goto**-оператора равен 1. **goto**-операторы следует применять изредка, ибо, вообще говоря, они затрудняют понимание программы. Основное применение **goto**-оператора — это выход из **while**-операторов.

7. Последовательность операторов, отделенных друг от друга точками с запятыми и заключенных между выделенными словами **begin** и **end**, образует оператор, который называется *блоком*. Так как блок является оператором, его можно применять всюду, где предусмотрено применение оператора. Обычно программа будет блоком. Вес блока равен сумме весов операторов, составляющих блок.

8. *Процедуры*. В Упрощенном Алголе процедуры можно определять и впоследствии вызывать. Процедуры определяются *оператором определения процедур*

procedure имя (список параметров): оператор

Список параметров — это последовательность фиктивных переменных, называемых *формальными параметрами*. Например, следующий оператор определяет процедуру-функцию MIN:

procedure MIN (x, y):
if $x > y$ **then return** y **else return** x

Аргументы x и y являются формальными параметрами.

Процедуры используются одним из двух способов. Один способ — в качестве *функции*. После того как процедура-функция описана, к ней можно обратиться в некотором выражении, вызывая ее имя с нужными аргументами. В этом случае последним оператором, выполняемым в данной процедуре, должен быть **return**-оператор (8б). Этот оператор приводит к вычислению значения выражения, следующего за выделенным словом **return**, и окончанию выполнения процедуры. Значением функции будет значение этого выражения. Например,

$$A \leftarrow \mathrm{MIN}(2+3,\ 7)$$

приводит к тому, что A получает значение 5. Выражения 2+3 и 7 называются *фактическими параметрами* этого обращения к данной процедуре.

Второй способ применения процедуры состоит в вызове ее с помощью оператора вызова процедуры (8в). Этот оператор есть, по существу, имя процедуры, за которым идет список фактических параметров. Оператор вызова процедуры может изменить и (обыч-

но изменяет) данные (значения переменных, массивов и т. д.) вызываемой программы. В определении вызываемой таким способом процедуры **return**-оператор не нужен. Завершение выполнения последнего оператора процедуры завершает и выполнение оператора ее вызова. Например, следующий оператор определяет процедуру ВЗАИМОЗАМЕНА:

```
procedure ВЗАИМОЗАМЕНА (x, y);
begin
    t ← x;
    x ← y;
    y ← t
end
```

Для обращения к этой процедуре можно было бы написать оператор вызова процедуры, такой, как

ВЗАИМОЗАМЕНА ($A[i]$, $A[j]$)

Обмен информацией между процедурами можно осуществлять двумя способами. Во-первых, с помощью глобальных переменных. Мы предполагаем, что глобальные переменные неявно описаны в некоторой универсальной области. В этой области есть подобласть, в которой определены процедуры.

Во-вторых, связь между процедурами можно осуществлять с помощью параметров. В Алголе 60 используется вызов по значению и вызов по наименованию. При *вызове по значению* формальные параметры процедуры трактуются как локальные переменные, которым в качестве начальных значений придаются значения фактических параметров. При *вызове по наименованию* формальные параметры служат указателями места в программе, куда подставляются фактические параметры вместо каждого вхождения соответствующих формальных параметров. Для простоты мы отходим от Алгола 60 и используем вызов по ссылке. При *вызове по ссылке* параметры обрабатываются с помощью указателей на фактические параметры. Если фактический параметр является выражением (возможно, постоянной), то соответствующий формальный параметр трактуется как локальная переменная, которой в качестве начального значения присвоено значение этого выражения. Поэтому вес реализации вычисления функции или выполнения вызова процедуры на РАМ и РАСП равен сумме весов выполнения операторов в определении соответствующей процедуры. Вес выполнения процедуры, вызывающей другие процедуры (возможно, себя), обсуждается в гл. 2.

9. Смысл **read**-оператора и **write**-оператора очевиден. Вес **read**-оператора равен 1. Вес **write**-оператора на 1 больше веса вычисления значения выражения, стоящего за выделенным словом **write**.

10. **comment**-оператор позволяет вставлять замечания, облегчающие понимание программы, и имеет вес 0.

11 В дополнение к операторам общепринятых языков программирования мы включили под именем "произвольные" любые операторы, благодаря которым алгоритм можно понять легче, чем эквивалентную последовательность стандартных операторов языка программирования. Такие операторы используются в ситуациях, когда детали реализации либо несущественны, либо очевидны или когда желательно дать описание на языке еще более высокого уровня. Приведем примеры обычно используемых "произвольных" операторов:

а) пусть a будет наименьшим элементом множества S
б) пометить элемент a как "старый" [1])
в) **without loss of generality (wlg)** считаем, что... **otherwise** ... **in** оператор

Например,

wlg считаем $a \leqslant b$ **otherwise** переставить c и d **in** оператор

означает, что если $a \leqslant b$, то стоящий дальше оператор следует выполнять так, как он записан, а если $a > b$, то выполнять этот оператор, предварительно поменяв в нем c и d местами.

Реализация таких операторов в терминах общепринятых языков программирования или команд РАМ не вызывает затруднений, но она очень утомительна. Приписывание весов операторам такого типа зависит от контекста, в котором оказывается данный оператор. Дальнейшие примеры операторов такого рода не раз встретятся нам в этой книге в программах на Упрощенном Алголе.

Поскольку переменные обычно не будут описываться, условимся об областях их действия. В одной программе или процедуре мы не будем употреблять одинаковые имена для двух разных переменных. Поэтому в качестве области действия переменной обычно можно брать всю процедуру или программу, в которую она входит [2]).

Одно важное исключение возникает в случае, когда несколько процедур действуют на общей базе данных. В этом случае предполагается, что переменные общей базы данных глобальны, а переменные, используемые процедурой для временного запоминания в ходе работы с базой данных, локальны для этой процедуры. Всякий раз, когда может возникнуть недоразумение по поводу области действия переменной, будет даваться явное описание.

---

[1]) Под этим мы подразумеваем, что имеется массив СОСТОЯНИЕ, такой, что СОСТОЯНИЕ[a] есть 1, если a — «старый» элемент, и 0, если a — «новый».

[2]) Это утверждение имеет несколько несущественных исключений. Например, в процедуру могут входить два невложенных **for**-оператора, оба с индексом i. Строго говоря, область действия индекса **for**-оператора — это сам **for**-оператор, и потому эти индексы i являются разными переменными.

УПРАЖНЕНИЯ

1.1. Докажите, что $g(n)$ есть $O(f(n))$, если (а) $f(n) \geqslant \varepsilon$ для некоторого $\varepsilon > 0$ и для почти всех (т. е. для всех, кроме конечного числа) n и (б) существуют такие постоянные $c_1 > 0$ и $c_2 > 0$, что $g(n) \leqslant c_1 f(n) + c_2$ для почти всех $n \geqslant 0$.

1.2. Будем писать $f(n) \lesssim g(n)$, если существует такая положительная постоянная c, что $f(n) \leqslant cg(n)$ для всех n. Покажите, что $f_1 \lesssim g_1$ и $f_2 \lesssim g_2$ влечет $f_1 + f_2 \lesssim g_1 + g_2$. Какие еще свойства сохраняет отношение $\lesssim$?

1.3. Напишите программы для РАМ, РАСП и на Упрощенном Алголе для следующих задач:

(а) Вычислить $n!$ по входу n.

(б) Прочитать n положительных чисел, за которыми следует концевой маркер 0, а затем напечатать их в порядке неубывания.

(в) Допустить все входы вида $1^{n}2^{n}0$.

1.4. Проанализируйте временную и емкостную сложности ваших программ из упр. 1.3 при (а) равномерном и (б) логарифмическом весе. Введите меру "размера" входа.

***1.5.** Напишите РАМ-программу для вычисления n^n с временной сложностью $O(\log n)$ при равномерном весе. Докажите, что ваша программа правильна.

***1.6.** Покажите, что для любой РАМ-программы с временной сложностью $T(n)$ при логарифмическом весе существует эквивалентная РАМ-программа с временной сложностью $O(T^2(n))$, в которой нет команд MULT и DIV. *Указание:* Смоделируйте MULT и DIV подпрограммами, в которых регистры с четными номерами используются для промежуточной памяти. В случае MULT покажите, что если i надо умножить на j, то можно сосчитать каждое из $l(j)$ частичных произведений и сложить их за $O(l(j))$ шагов, а каждый шаг занимает время $O(l(i))$.

***1.7.** Что случится с вычислительной силой РАМ или РАСП, если из системы команд убрать MULT вместе с DIV? Как это отразится на весе вычисления?

****1.8.** Покажите, что любой язык, допускаемый РАМ, может допустить РАМ без косвенной адресации. *Указание:* Покажите, что всю ленту машины Тьюринга можно целиком закодировать одним целым числом. Таким образом, машину Тьюринга можно смоделировать в конечном числе регистров РАМ.

1.9. Покажите, что при (а) равномерном и (б) логарифмическом весах РАМ и РАСП эквивалентны в смысле равенства емкостных сложностей с точностью до постоянного множителя.

1.10. Найдите неветвящуюся программу, которая вычисляет определитель (3×3)-матрицы по ее 9 элементам в качестве входов.

1.11. Напишите последовательность битовых операций для вычисления произведения двух двуразрядных двоичных целых чисел.

1.12. Покажите, что множество функций, вычислимых любой n-строчной неветвящейся программой с двоичными входами и булевыми операциями, можно реализовать в виде комбинационной логической сети из n булевых элементов.

1.13. Покажите, что любую булеву функцию можно вычислить неветвящейся программой.

***1.14.** Пусть граф с n узлами представлен множеством двоичных векторов $\mathbf{v}_i$, где j-я компонента вектора $\mathbf{v}_i$ равна 1 тогда и только тогда, когда узлы i и j соединены ребром. Найдите алгоритм сложности $O_{\text{дв}}(n)$, строящий вектор $\mathbf{p}_1$, у которого на j-м месте стоит 1 тогда и только тогда, когда есть путь из узла 1 в узел j. Можно применить поразрядные битовые логические операции на двоичных векторах, арифметические операции (на переменных типа "целые"), команды, которые преобразуют отдельные компоненты данных векторов в 0 или 1, и команду, которая присваивает значение j целочисленной переменной a, если самая левая единица в векторе $\mathbf{v}$ находится в разряде j, и полагает $a=0$, если $\mathbf{v}$ состоит из одних нулей.

***1.15.** Постройте машину Тьюринга, которая по двум данным двоичным целым числам на лентах 1 и 2 печатает их сумму на ленте 3. Можете считать, что левые концы лент отмечены специальным символом #.

1.16. Приведите последовательность конфигураций (мгновенных описаний) МТ с рис. 1.21, если входом является (а) 0010, (б) 01110.

***1.17.** Постройте МТ, которая
(а) печатает 0^{n^2} на ленте 2, начиная работу с 0^n на ленте 1,
(б) допускает входы вида $0^n10^{n^2}$.

1.18. Укажите множество состояний и функцию переходов МТ, моделирующей РАМ-команду LOAD 3, как в доказательстве теоремы 1.3.

1.19. Постройте РАМ-программу, вычисляющую 2^{2^n} по данному n за $O(n)$ шагов. Чему равны (а) равномерный и (б) логарифмический веса вашей программы?

***1.20.** Определим $g(m, n)$ равенствами $g(0, n)=n$ и $g(m, n)=2^{g(m-1,\, n)}$ при $m>0$. Напишите РАМ-программу, вычисляющую $g(n, n)$ по n. Как соотносятся равномерный и логарифмический веса вашей программы?

1.21. Выполните процедуру ВЗАИМОЗАМЕНА из разд. 1.8 с фактическими параметрами i и $A[i]$, используя сначала вызов по наименованию, а затем вызов по ссылке. Будут ли результаты одинаковы?

Проблема для исследования

1.22. Можно ли улучшить верхнюю оценку $O(T^2(n))$ времени моделирования РАМ машиной Тьюринга, как в теореме 1.3?

Замечания по литературе

РАМ и РАСП получили формальную трактовку у Шепердсона, Стерджиса [1963], Элгота, Робинсона [1964] и Хартманиса [1971]. В изложении большей части результатов о РАМ и РАСП в этой главе мы следуем работе Кука, Рекхау [1973].

Идея машины Тьюринга принадлежит Тьюрингу [1936]. Более подробное изложение этого понятия можно найти у Минского [1967] и Хопкрофта, Ульмана [1969]; это же касается и ответа к упр. 1.8. Изучение временной сложности машин Тьюринга было начато Хартманисом, Стирнзом [1965], а емкостной сложности — Хартманисом, Льюисом, Стирнзом [1965] и Льюисом, Стирнзом, Хартманисом [1965] [1]). Начиная с работы Блюма [1967], делались попытки трактовать вычислительную сложность с гораздо более абстрактной точки зрения. Обзоры можно найти у Хартманиса, Хопкрофта [1971] и Бородина [1973а].

Рабин [1972] предложил одно интересное обобщение понятия дерева решений.

---

[1]) В СССР первые работы, в которых изучались временная и емкостная сложности (для моделей, отличных от рассматриваемых в гл. 1), были выполнены в 1956 г. Г. С. Цейтиным (см. С. А. Яновская. Математическая логика и основания математики. Математика в СССР за 40 лет, т. 1, М., Физматгиз, 1959, 44—45) и Трахтенбротом [1956].— *Прим. перев.*

2 РАЗРАБОТКА ЭФФЕКТИВНЫХ АЛГОРИТМОВ

Эта глава преследует двойную цель. Во-первых, мы вводим некоторые из основных структур данных, которые полезны при разработке эффективных алгоритмов для обширных классов задач. Во-вторых, описываем некоторую технику "программирования", такую, как рекурсия и динамическое программирование, которая применяется во многих эффективных алгоритмах.

В гл. 1 мы рассмотрели основные модели вычислений. Хотя нашей простейшей моделью является РАМ, мы не хотим, как правило, описывать алгоритмы в терминах подобного устройства. Поэтому мы ввели Упрощенный Алгол (разд. 1.8). Но даже этот язык оказывается слишком примитивным, если не ввести в рассмотрение структуры данных, более сложные, чем массивы. В этой главе мы познакомим читателя с такими элементарными структурами данных, как списки и стеки; они часто используются в эффективных алгоритмах. Мы покажем, что с помощью этих структур можно представлять множества, графы и деревья.

Наше изложение по необходимости будет кратким, и читателю, не знакомому с обработкой списков, следует обратиться к одному из более полных источников, указанных в конце данной главы, или уделить особое внимание упражнениям.

Мы включили также раздел о рекурсии. Один из важных аспектов рекурсии — облегчить понимание алгоритмов. Хотя примеры, приводимые в этой главе, слишком просты, чтобы полностью подтвердить это заявление, но в последующих главах рекурсия очень поможет в учете организации информации при точном изложении сравнительно сложных алгоритмов. Рекурсия сама по себе не приводит к более эффективным алгоритмам. Но в сочетании с другой техникой, такой, как балансировка, прием "разделяй и властвуй", алгебраические упрощения, она, как мы увидим, часто дает алгоритмы, одновременно и эффективные, и элегантные.

2.1. СТРУКТУРЫ ДАННЫХ: СПИСКИ, ОЧЕРЕДИ И СТЕКИ

Мы считаем, что читатель знаком с элементарными понятиями математики (такими, как множества и отношения) и основными типами данных — целыми числами, цепочками (словами) и массивами. В этом разделе мы дадим краткий обзор основных операций над списками.

С математической точки зрения *список* — это конечная последовательность элементов, взятых из некоторого подходящего множества. Описание алгоритма часто будет включать в себя некоторый список, к которому добавляются и из которого удаляются элементы. В частности мы можем захотеть добавить или удалить элемент где-то в середине списка. По этой причине нам надо разработать структуры данных, позволяющие реализовать списки, в которых разрешается удалять и добавлять новые элементы так, как нам захочется.

Рассмотрим список

Элем 1, Элем 2, Элем 3, Элем 4. (2.1)

Простейшей его реализацией будет структура последовательно связанных компонент, изображенная на рис. 2.1. Каждая компонента в этой структуре состоит из двух ячеек памяти. Первая ячейка содержит сам элемент [1]), вторая — указатель следующего элемента. Это можно реализовать в виде двух массивов, которые на рис. 2.2 названы ИМЯ и СЛЕДУЮЩАЯ [2]). Если КОМПОНЕНТА — индекс в рассматриваемом массиве, то ИМЯ[КОМПОНЕНТА] — сам элемент, хранящийся там, а СЛЕДУЮЩАЯ[КОМПОНЕНТА]— индекс следующего элемента в списке (если такой элемент существует). Если КОМПОНЕНТА — индекс последнего элемента в этом списке, то СЛЕДУЮЩАЯ[КОМПОНЕНТА]=0.

На рис. 2.2 СЛЕДУЮЩАЯ[0] означает постоянный указатель на первую компоненту в списке. Заметим, что порядок элементов в

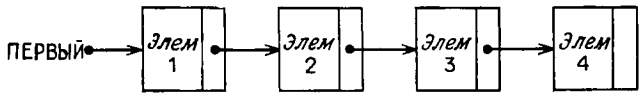


Рис. 2.1. Список со связями.

[1]) Если этот элемент сам является сложной структурой, то первая ячейка может содержать указатель ее местоположения.

[2]) Другая (и эквивалентная) точка зрения такова: имеется «клетка» для каждой компоненты. Каждая «клетка» обладает адресом, который представляет собой номер первого (возможно, единственного) регистра памяти в группе регистров, зарезервированных для этой компоненты. Внутри каждой клетки находится одно или более «полей». У нас полями служат ИМЯ и СЛЕДУЮЩАЯ, а ИМЯ[КОМПОНЕНТА] и СЛЕДУЮЩАЯ[КОМПОНЕНТА] играют роль имен этих полей в клетке с адресом КОМПОНЕНТА.

	ИМЯ	СЛЕДУЮЩАЯ
0	—	1
1	*Элем* 1	3
2	*Элем* 4	0
3	*Элем* 2	4
4	*Элем* 3	2

Рис. 2.2. Представление списка из четырех элементов.

массиве ИМЯ не совпадает с их порядком в списке. Тем не менее рис. 2.2 дает верное представление списка, изображенного на рис. 2.1, так как массив СЛЕДУЮЩАЯ располагает элементы в том же порядке, в каком они расположены в списке (2.1).

Опишем процедуру, вставляющую новую компоненту в список. В ней предполагается, что СВОБОДНАЯ — номер незанятой ячейки в массивах ИМЯ и СЛЕДУЮЩАЯ, а ПОЗИЦИЯ — индекс той компоненты в списке, после которой надлежит вставить ЭЛЕМЕНТ:

```
procedure ВСТАВИТЬ(ЭЛЕМЕНТ, СВОБОДНАЯ, ПОЗИЦИЯ):
begin
 ИМЯ[СВОБОДНАЯ] ← ЭЛЕМЕНТ;
 СЛЕДУЮЩАЯ[СВОБОДНАЯ] ← СЛЕДУЮЩАЯ[ПОЗИЦИЯ];
 СЛЕДУЮЩАЯ [ПОЗИЦИЯ] ← СВОБОДНАЯ
end
```

Любой разумный перевод в команды РАМ приведет к тому, что время выполнения процедуры ВСТАВИТЬ не будет зависеть от размера списка.

Пример 2.1. Допустим, что мы хотим вставить в список (2.1) элемент Новэлем после Элем 2 и получить список

Элем 1, Элем 2, Новэлем, Элем 3, Элем 4.

Если пятая ячейка в каждом массиве на рис. 2.2 пуста, можно вставить Новэлем после Элем 2 (позиция 3), вызвав ВСТАВИТЬ(Новэлем, 5, 3). В результате выполнения трех операторов в процедуре ВСТАВИТЬ получим: ИМЯ[5] = Новэлем,

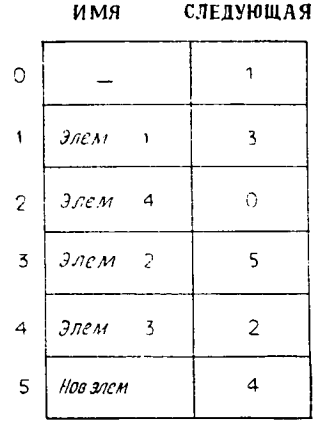


Рис. 2.3. Список со вставленным элементом Новэлем.

СЛЕДУЮЩАЯ[5]=4 и СЛЕДУЮЩАЯ[3]=5; тем самым создадутся массивы, показанные на рис. 2.3. □

Для того чтобы удалить компоненту, следующую за компонентой в ячейке *I*, можно положить СЛЕДУЮЩАЯ[*I*]=СЛЕДУЮЩАЯ [СЛЕДУЮЩАЯ[*I*]]. При желании индекс удаленной компоненты можно поместить в список незанятых ячеек памяти.

Часто в один и тот же массив вкладываются несколько списков Обычно один из этих списков состоит из незанятых ячеек; мы будем называть его *свободным списком*. Для добавления элемента к списку *A* можно так изменить процедуру ВСТАВИТЬ, чтобы незанятая ячейка получалась путем удаления первой ячейки в свободном списке. При удалении элемента из списка *A* соответствующая ячейка возвращается в свободный список для будущего употребления.

Этот метод организации памяти — не единственный прием, которым обычно пользуются, но он приведен здесь для того, чтобы показать, что операции добавления и удаления элементов списка можно выполнить за ограниченное число шагов, если определено местоположение элемента, который мы хотим добавить или удалить.

Еще две основные операции над списками — конкатенация (сцепление) двух списков, в результате которой образуется единственный список, и обратная к ней операция расцепления списка, стоящего после некоторого элемента, результатом которой будут два списка. Конкатенацию можно выполнить за ограниченное (постоянной величиной) число шагов, включив в представление списка еще один указатель. Этот указатель дает индекс последней компоненты списка и тем самым позволяет обойтись без просмотра всего списка для определения его последнего элемента. Расцепление мож-

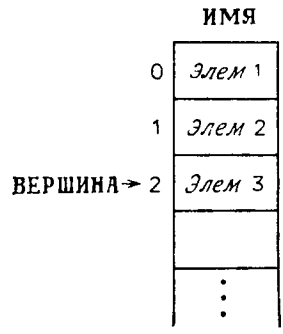


Рис. 2.4. Реализация стека.

но сделать за ограниченное (постоянной величиной) время, если известен индекс компоненты, непосредственно предшествующей месту расцепления.

Списки можно сделать проходимыми в обоих направлениях, если добавить еще один массив, называемый ПРЕДЫДУЩАЯ. Значение ПРЕДЫДУЩАЯ[*l*] равно ячейке, в которой помещается тот элемент списка, который стоит непосредственно перед элементом из ячейки *l*. Список такого рода называется *дважды связанным*. Из дважды связанного списка можно удалить элемент или вставить в него элемент, не зная ячейку, где находится предыдущий элемент.

Со списком часто работают очень ограниченными приемами. Например, элементы добавляются или удаляются только на конце списка. Иными словами, элементы вставляются и удаляются по принципу "последний вошел — первый вышел". В этом случае список называют *стеком* или *магазином*.

Часто стек реализуется в виде одного массива. Например, список

Элем 1, Элем 2, Элем 3

можно было бы хранить в массиве ИМЯ, как показано на рис. 2.4. Переменная ВЕРШИНА является указателем последнего элемента, добавленного к стеку. Чтобы добавить (ЗАТОЛКНУТЬ) новый элемент в стек, значение ВЕРШИНА увеличивают на 1, а затем помещают новый элемент в ИМЯ[ВЕРШИНА]. (Поскольку массив ИМЯ имеет конечную длину *l*, перед попыткой вставить новый элемент следует проверить, что ВЕРШИНА $< l-1$.) Чтобы удалить (ВЫТОЛКНУТЬ) элемент из вершины стека, надо просто уменьшить значение ВЕРШИНА на 1. Заметим, что не обязательно физически стирать элемент, удаляемый из стека. Чтобы узнать, пуст ли стек, достаточно проверить, не имеет ли ВЕРШИНА значение, меньшее нуля. Понятно, что время выполнения операций ЗАТОЛКНУТЬ и

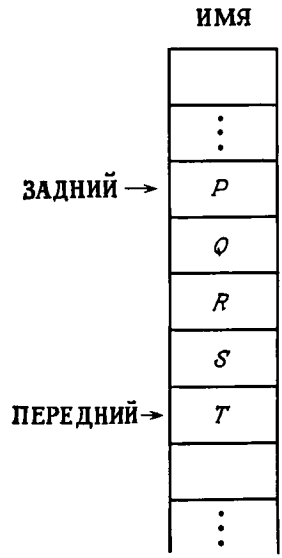


Рис. 2.5. Реализация очереди в виде одного массива.

ВЫТОЛКНУТЬ и проверка пустоты не зависят от числа элементов в стеке.

Другой специальный вид списка — *очередь*, т. е. список, в который элементы всегда добавляются с одного (переднего) конца, а удаляются с другого. Как и стек, очередь можно реализовать одним массивом, как показано на рис. 2.5, где приведена очередь, содержащая список из элементов P, Q, R, S, T. Два указателя обозначают ячейки текущего переднего и заднего концов очереди. Чтобы добавить (ВПИСАТЬ) новый элемент к очереди, как и в случае стека, полагают ПЕРЕДНИЙ = ПЕРЕДНИЙ +1 и помещают новый элемент в ИМЯ[ПЕРЕДНИЙ]. Чтобы удалить (ВЫПИСАТЬ) элемент из очереди, заменяют ЗАДНИЙ на ЗАДНИЙ +1. Заметьте, что эта техника с точки зрения доступа к элементам основана на принципе "первый вошел — первый вышел".

Поскольку массив ИМЯ имеет конечную длину, скажем l, указатели ПЕРЕДНИЙ и ЗАДНИЙ рано или поздно доберутся до его конца. Если длина списка, представленного этой очередью, никогда не превосходит l, то можно трактовать ИМЯ[0] как элемент, следующий за элементом ИМЯ[l—1].

Элементы, расположенные в виде списка, сами могут быть сложными структурами. Работая, например, со списком массивов, мы на самом деле не добавляем и не удаляем массивы, ибо каждое добавление или удаление потребовало бы времени, пропорционального раз-

меру массива. Вместо этого мы добавляем или удаляем указатели массивов. Таким образом, сложную структуру можно добавить или удалить за фиксированное время, не зависящее от ее размера.

2.2. ПРЕДСТАВЛЕНИЯ МНОЖЕСТВ

Обычно списки применяются для представления множеств. При этом объем памяти, необходимый для представления множества, пропорционален числу его элементов. Время, требуемое для выполнения операции над множествами, зависит от природы операции. Например, пусть A и B — два множества. Операция $A \cap B$ требует времени, по крайней мере пропорционального сумме размеров этих множеств, поскольку как список, представляющий A, так и список, представляющий B, надо просмотреть хотя бы один раз [1]).

Подобным же образом операция $A \cup B$ требует времени, пропорционального сумме размеров множеств, поскольку надо выделить элементы, входящие в оба множества, и вычеркнуть один экземпляр каждого такого элемента. Если же A и B не пересекаются, можно найти $A \cup B$ за время, не зависящее от размера A и B. Для этого достаточно сделать конкатенацию списков, представляющих A и B. Задача объединения двух непересекающихся множеств усложняется, если необходимо быстро определять, входит ли данный элемент в данное множество. Этот вопрос подробно обсуждается в разд. 4.6 и 4.7.

Другой способ представления множества, отличный от представления его в виде списка,— представление в виде двоичного (битового) вектора. Пусть U — универсальное множество (т. е. все рассматриваемые множества являются его подмножествами), состоящее из n элементов. Линейно упорядочим его. Подмножество $S \subseteq U$ представляется в виде вектора $\mathbf{v}_S$ из n битов, такого, что i-й разряд в $\mathbf{v}_S$ равен 1 тогда и только тогда, когда i-й элемент множества U принадлежит S. Будем называть $\mathbf{v}_S$ *характеристическим вектором* для S.

Представление в виде двоичного вектора удобнее тем, что можно определять принадлежность i-го элемента множества U данному множеству за время, не зависящее от размера данного множества. Более того, основные операции над множествами, такие, как объединение и пересечение, можно осуществить как операции $\vee$ и $\wedge$ над двоичными векторами.

Если мы не хотим считать операции над двоичными векторами первичными (т. е. выполняемыми за единицу времени), то можно с таким же успехом вместо характеристического вектора определить массив A, для которого $A[i]=1$ тогда и только тогда, когда i-й эле-

[1]) Если оба списка упорядочены, то для нахождения их пересечения существует линейный алгоритм.

мент множества U принадлежит S. При таком представлении все еще легко выяснять, принадлежит ли данный элемент данному множеству. Недостаток этого представления заключается в том, что объединение и пересечение занимают время, пропорциональное $\|U\|$ [1]), а не размерам рассматриваемых множеств. Подобно этому память, требуемая для хранения множества S, пропорциональна $\|U\|$, а не $\|S\|$.

2.3. ГРАФЫ

Сейчас мы введем математическое понятие графа и те структуры данных, которые обычно применяются для его представления.

Определение. *Граф* $G=(V, E)$ состоит из конечного непустого множества *узлов* V и множества *ребер* E. Если ребра представлены в виде упорядоченных пар (v, w) узлов, то граф называется *ориентированным*; v называется *началом*, а w — *концом* ребра (v, w). Если ребра — неупорядоченные пары (множества) различных вершин, также обозначаемые (v, w), то граф называют *неориентированным* [2]).

Если в ориентированном графе $G=(V, E)$ пара (v, w) принадлежит множеству ребер E, то узел w называется *смежным* с узлом v. Говорят, что ребро (v, w) *идет из* v *в* w. Число узлов, смежных с узлом v, называется *полустепенью его исхода*.

Если (v, w) — ребро неориентированного графа $G=(V, E)$, то мы считаем, что $(v, w)=(w, v)$, так что (w, v) — то же самое ребро. Узел w называется *смежным* с узлом v, если (v, w) (а значит, и (w, v)) принадлежит E. *Степень* узла — это число узлов, смежных с ним.

Путем в ориентированном или неориентированном графе называют последовательность ребер вида $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$. Говорят, что этот путь *идет из* v_1 *в* v_n и *имеет длину* $n-1$. Часто такой путь представляют последовательностью $v_1, v_2, \ldots, v_n$ узлов, лежащих на нем. В вырожденном случае один узел обозначает путь длины 0, идущий из этого узла в него же. Путь называется *простым*, если все ребра и все узлы на нем, кроме, быть может, первого и последнего, различны. *Цикл* — это простой путь длины не менее 1, который начинается и кончается в одном и том же узле. Заметим, что в неориентированном графе длина цикла должна быть не менее 3.

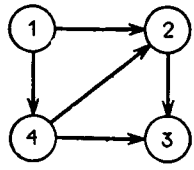
Известно несколько представлений графа $G=(V, E)$. Один из них — *матрица смежностей*, т. е. матрица A размера $\|V\| \times \|V\|$, состоящая из 0 и 1, в которой $A[i, j]=1$ тогда и только тогда, когда есть ребро из узла i в узел j. Представление в виде матрицы смеж-

[1]) $\|X\|$ обозначает здесь число элементов (размер, или *мощность*) множества X.

[2]) Заметим, что в ориентированном графе может быть ребро (a, a), а в неориентированном — нет.

ностей удобно для тех алгоритмов на графах, которым часто нужно знать, есть ли в графе данное ребро, ибо время, необходимое для определения наличия ребра, фиксировано и не зависит от $||V||$ и $||E||$. Основной недостаток применения матрицы смежностей заключается в том, что она занимает память объема $||V||^2$ даже тогда, когда граф содержит только $O(||V||)$ ребер. Уже начальное заполнение матрицы смежностей посредством "естественной" процедуры требует времени $O(||V||^2)$, что сводит на нет алгоритмы сложности $O(||V||)$ при работе с графами, содержащими лишь $O(||V||)$ ребер. Хотя разработаны методы для преодоления этой трудности (см. упр. 2.12), почти неизбежно возникают другие проблемы, которые приводят к тому, что алгоритмы сложности $O(||V||)$, основанные на работе с матрицей смежностей, встречаются редко.

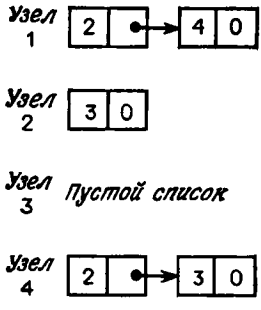
Интересной альтернативой является представление строк и (или) столбцов матрицы смежностей в виде двоичных векторов. Такое



а

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

б



в

		КОНЕЦ	СЛЕДУЮЩИЙ
Узлы	1		5
	2		7
	3		0
	4		8
Ребра	5	2	6
	6	4	0
	7	3	0
	8	2	9
	9	3	0

г

Рис. 2.6. Ориентированный граф и его представления: *а* — ориентированный граф; *б* — матрица смежностей; *в* — списки смежностей; *г* — табличное представление списков смежностей.

представление может способствовать значительной эффективности алгоритмов на графах.

Еще одно возможное представление графа — с помощью списков. *Списком смежностей* для узла v называется список всех узлов w, смежных с v. Граф можно представить с помощью $||V||$ списков смежностей, по одному для каждого узла.

Пример 2.2. На рис. 2.6,*а* изображен ориентированный граф, содержащий четыре узла, а на рис. 2.6,*б* — его матрица смежностей. На рис. 2.6,*в* показаны четыре списка смежностей, по одному для каждого узла. Например, из узла 1 в узлы 2 и 4 идут ребра, так что список смежностей для 1 содержит компоненты 2 и 4, связанные в смысле рис. 2.1.

Табличное представление списков смежностей приведено на рис. 2.6,*г*. Каждая из первых четырех ячеек в массиве СЛЕДУЮЩИЙ содержит указатель на первый узел списка смежностей, а именно СЛЕДУЮЩИЙ[i] указывает на первый узел списка смежностей для узла i. Заметим, что СЛЕДУЮЩИЙ[3]=0, поскольку в списке смежностей для узла 3 нет узлов. Остальные составляющие массива СЛЕДУЮЩИЙ представляют ребра графа. Массив КОНЕЦ содержит узлы из списков смежностей. Таким образом, список смежностей узла 1 начинается в ячейке 5, ибо СЛЕДУЮЩИЙ[1]=5, КОНЕЦ[5]=2; это показывает, что есть ребро (1, 2). Равенства СЛЕДУЮЩИЙ[5]=6 и КОНЕЦ[6]=4 означают, что есть ребро (1, 4), а СЛЕДУЮЩИЙ[6]=0 — что больше нет ребер, начинающихся в 1. □

Заметим, что представление графа в виде списков смежностей требует памяти порядка $||V||+||E||$. Представлением с помощью списков смежностей часто пользуются, когда $||E|| \ll ||V||^2$.

Если граф неориентирован, то каждое ребро (v, w) представляется дважды: один раз в списке смежностей для v и один раз в списке смежностей для w. В этом случае можно добавить новый массив, называемый СВЯЗЬ, чтобы коррелировать оба экземпляра неориентированного ребра. Таким образом, если i — ячейка, соответствующая узлу w в списке смежностей для v, то СВЯЗЬ[i] — ячейка, соответствующая узлу v в списке смежностей для w.

Если мы хотим с удобством удалять ребра из неориентированного графа, то списки смежностей можно связать дважды (как описано в разд. 2.1). Это обычно бывает нужно потому, что даже если удалять всегда ребро (v, w), стоящее первым в списке смежностей узла v, все равно может оказаться, что ребро, идущее в обратном направлении, стоит в середине списка смежностей узла w. Чтобы быстро удалить ребро (v, w) из списка смежностей для w, надо уметь быстро находить ячейку, содержащую предыдущее ребро в этом списке смежностей.

2.4. ДЕРЕВЬЯ

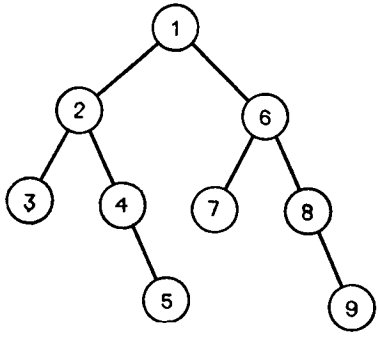
Теперь введем очень важный вид ориентированных графов — деревья — и рассмотрим структуры данных, пригодные для их представления.

Определение. Ориентированный граф без циклов называется *ориентированным ациклическим графом*. *(Ориентированное) дерево* (иногда его называют *корневым деревом*) — это ориентированный ациклический граф, удовлетворяющий следующим условиям:

1) имеется в точности один узел, называемый *корнем*, в который не входит ни одно ребро,
2) в каждый узел, кроме корня, входит ровно одно ребро,
3) из корня к каждому узлу идет путь (который, как легко показать, единствен).

Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*. Леса и деревья — столь часто встречающиеся частные случаи ориентированных ациклических графов, что для описания их свойств стóит ввести специальную терминологию.

Определение. Пусть $F=(V, E)$ — граф, являющийся лесом. Если (v, w) принадлежит E, то v называется *отцом* узла w, а w — *сыном* узла v. Если есть путь из v в w, то v называется *предком* узла w, а w — *потомком* узла v. Более того, если $v \neq w$, то v называется *подлинным предком* узла w, а w — *подлинным потомком* узла v. Узел без подлинных потомков называется *листом*. Узел v и его потомки вместе образуют *поддерево* леса F, и узел v называется *корнем* этого поддерева.



а

	ЛЕВЫЙСЫН	ПРАВЫЙСЫН
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	0

б

Рис. 2.7. Двоичное дерево и его представление.

Глубина узла v в дереве — это длина пути из корня в v. *Высота узла* v в дереве — это длина самого длинного пути из v в какой-нибудь лист. *Высотой дерева* называется высота его корня. *Уровень* узла v в дереве равен разности высоты дерева и глубины узла v. Например, на рис. 2.7,*а* узел 3 имеет глубину 2, высоту 0 и уровень 1.

Упорядоченным деревом называется дерево, в котором множество сыновей каждого узла упорядочено. При изображении упорядоченного дерева мы будем считать, что множество сыновей каждого узла упорядочено слева направо. *Двоичным (бинарным) деревом* называется такое упорядоченное дерево, что

1) каждый сын произвольного узла идентифицируется либо как *левый сын*, либо как *правый сын*,
2) каждый узел имеет не более одного левого сына и не более одного правого сына.

Поддерево T_l, корнем которого является левый сын узла v (если такое существует), называется *левым поддеревом* узла v. Аналогично поддерево T_r, корнем которого является правый сын узла v (если такое существует), называется *правым поддеревом* узла v. Все узлы в T_l расположены *левее* всех узлов в T_r.

Двоичное дерево обычно представляют в виде двух массивов ЛЕВЫЙСЫН и ПРАВЫЙСЫН. Пусть узлы двоичного дерева занумерованы целыми числами от 1 до n. В этом случае ЛЕВЫЙСЫН[i]=j тогда и только тогда, когда узел с номером j является левым сыном узла с номером i. Если у узла i нет левого сына, то ЛЕВЫЙСЫН[i]=0. ПРАВЫЙСЫН[i] определяется аналогично.

Пример 2.3. Двоичное дерево и его представление изображены на рис. 2.7. □

Определение. Двоичное дерево называется *полным*, если для некоторого целого числа k каждый узел глубины меньшей k имеет как левого, так и правого сына и каждый узел глубины k является листом. Полное двоичное дерево высоты k имеет ровно $2^{k+1}-1$ узлов.

Полное двоичное дерево высоты k часто представляют одним массивом. В позиции 1 этого массива находится корень. Левый сын узла в позиции i расположен в позиции $2i$, а его правый сын — в позиции $2i+1$. Отец узла, находящегося в позиции $i>1$, расположен в позиции $\lfloor i/2 \rfloor$.

Многие алгоритмы, использующие деревья, часто *проходят* дерево (посещают каждый его узел) в некотором порядке. Известно несколько систематических способов сделать это. Мы рассмотрим три широко распространенных способа: прохождение дерева в прямом порядке, обратном и внутреннем.

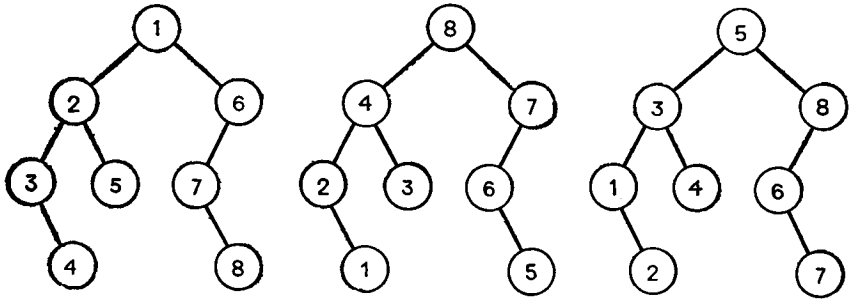


Рис. 2.8. Прохождение дерева: *а* — в прямом порядке; *б* — обратном; *в* — внутреннем.

Определение. Пусть T — дерево с корнем r и сыновьями $v_1, \ldots, v_k$, $k \geqslant 0$. При $k=0$ это дерево состоит из единственного узла r.

Прохождение дерева T *в прямом порядке* определяется следующей рекурсией:

1) посетить корень r,
2) посетить в прямом порядке поддеревья с корнями $v_1, \ldots, v_k$ в указанной последовательности.

Прохождение дерева T *в обратном порядке* определяется следующей рекурсией:

1) посетить в обратном порядке поддеревья с корнями $v_1, \ldots, v_k$ в указанной последовательности,
2) посетить корень r.

Прохождение двоичного дерева *во внутреннем порядке* определяется следующей рекурсией:

1) посетить во внутреннем порядке левое поддерево корня (если оно существует),
2) посетить корень,
3) посетить во внутреннем порядке правое поддерево корня (если оно существует).

Пример 2.4. На рис. 2.8 изображено двоичное дерево, узлы которого пронумерованы в соответствии с прохождением его в прямом порядке (рис. 2.8,*а*), обратном (рис. 2.8,*б*) и внутреннем (рис. 2.8,*в*). □

Если при некотором прохождении дерева его узлам были присвоены какие-то номера, то на узлы удобно ссылаться по этим номерам. Так, v будет обозначать узел, которому был присвоен номер v.

Если все узлы занумерованы в порядке посещения, то рассмотренные нумерации обладают рядом интересных свойств.

При нумерации в прямом порядке все узлы поддерева с корнем r имеют номера, не меньшие r. Точнее, если D_r — множество потомков узла r, то v будет номером некоторого узла из D_r тогда и только тогда, когда $r \leqslant v < r + \|D_r\|$. Поставив в соответствие каждому узлу v его номер в прямом порядке и количество его потомков, легко определить, является ли некоторый узел w потомком для v. После того как номера присвоены (в соответствии с прямым порядком) и вычислено количество потомков каждого узла, на вопрос, является ли w потомком для v, можно ответить за фиксированное время, не зависящее от размера дерева. Номера, соответствующие обратному порядку, обладают аналогичным свойством.

Номера узлов двоичного дерева, соответствующие внутреннему порядку, обладают тем свойством, что номера узлов в левом поддереве для v меньше v, а в правом поддереве больше v. Таким образом, чтобы найти узел с номером w, надо сравнить w с корнем r. Если $w = r$, то искомый узел найден. Если $w < r$, то надо повторить этот процесс для левого поддерева; если $w > r$, то повторить процесс для правого поддерева. В конце концов узел с номером w будет найден. Такие свойства прохождений нам понадобятся в следующих главах.

Дадим теперь еще одно, последнее определение, касающееся деревьев.

Определение. *Неориентированным деревом* называется неориентированный ациклический связный (любые два узла соединены путем) граф. *Корневое* неориентированное дерево — это неориентированное дерево, в котором один узел выделен в качестве корня.

Ориентированное дерево можно превратить в корневое неориентированное, просто сделав все его ребра неориентированными. Мы будем употреблять одну и ту же терминологию и пользоваться одними и теми же обозначениями для корневых неориентированных и для ориентированных деревьев. Основное математическое различие здесь состоит в том, что все пути в ориентированном дереве идут от предков к потомкам, тогда как в корневом неориентированном дереве пути могут идти в обоих направлениях.

2.5. РЕКУРСИЯ

Процедуру, которая прямо или косвенно обращается к себе, называют *рекурсивной*. Применение рекурсии часто позволяет давать более прозрачные и сжатые описания алгоритмов, чем это было бы возможно без рекурсии. В настоящем разделе мы приведем пример рекурсивного алгоритма и кратко опишем, как можно реализовать рекурсию на РАМ.

Рассмотрим определение прохождения двоичного дерева во внутреннем порядке, данное в разд. 2.4. При создании алгоритма, который присваивает узлам номера в соответствии с внутренним порядком, хорошо было бы отразить в нем определение прохождения во внутреннем порядке. Один из таких алгоритмов приведен ниже. Заметим, что он рекурсивно обращается к себе для нумерации поддерева.

Алгоритм 2.1. Нумерация узлов двоичного дерева в соответствии с внутренним порядком

Вход. Двоичное дерево, представленное массивами ЛЕВЫЙСЫН и ПРАВЫЙСЫН.

Выход. Массив, называемый НОМЕР, такой, что НОМЕР[*i*] — номер узла *i* во внутреннем порядке.

Метод. Кроме массивов ЛЕВЫЙСЫН, ПРАВЫЙСЫН и НОМЕР, алгоритм использует глобальную переменную СЧЕТ, значение которой — номер очередного узла в соответствии с внутренним порядком. Начальным значением переменной СЧЕТ является 1. Параметр УЗЕЛ вначале равен корню. Процедура, изображенная на рис. 2.9, применяется рекурсивно.

Сам алгоритм таков:

```
begin
    СЧЕТ ← 1;
    ВНУТПОРЯДОК(КОРЕНЬ)
end                                    □
```

Рекурсия дает несколько преимуществ и прежде всего простоту программ. Если бы приведенный выше алгоритм не был записан рекурсивно, надо было бы строить явный механизм для прохождения дерева. Двигаться вниз по дереву нетрудно, но чтобы обеспечить возможность вернуться к предку, надо запомнить всех предков в стеке, а операторы работы со стеком усложнили бы алгоритм.

```
    procedure ВНУТПОРЯДОК(УЗЕЛ):
    begin
1.    if ЛЕВЫЙСЫН[УЗЕЛ] ≠ 0 then
          ВНУТПОРЯДОК(ЛЕВЫЙСЫН[УЗЕЛ]);
2.    НОМЕР[УЗЕЛ] ← СЧЕТ;
3.    СЧЕТ ← СЧЕТ + 1;
4.    if ПРАВЫЙСЫН[УЗЕЛ] ≠ 0 then
          ВНУТПОРЯДОК(ПРАВЫЙСЫН[УЗЕЛ])
    end
```

Рис. 2.9. Рекурсивная процедура для внутреннего порядка.

Вариант того же алгоритма, не содержащий рекурсии, мог бы быть таким:

Алгоритм 2.2. Вариант алгоритма 2.1 без рекурсии

Вход. Тот же, что у алгоритма 2.1.

Выход. Тот же, что у алгоритма 2.1.

Метод. При прохождении дерева в стеке запоминаются все узлы, которые еще не были занумерованы и которые лежат на пути из корня в узел, рассматриваемый в данный момент. При переходе из узла v к его левому сыну узел v запоминается в стеке. После нахождения левого поддерева для v узел v нумеруется и выталкивается из стека. Затем нумеруется правое поддерево для v.

При переходе из v к его правому сыну узел v не помещается в стек, поскольку после нумерации правого поддерева мы не хотим возвращаться в v; более того, мы хотим вернуться к тому предку

```
          begin
            СЧЕТ ← 1;
            УЗЕЛ ← КОРЕНЬ;
            СТЕК ← пустой;
левый:      while ЛЕВЫЙСЫН[УЗЕЛ] ≠ 0 do
              begin
                затолкнуть УЗЕЛ в СТЕК;
                УЗЕЛ ← ЛЕВЫЙСЫН[УЗЕЛ]
              end;
центр:      НОМЕР[УЗЕЛ] ← СЧЕТ;
            СЧЕТ ← СЧЕТ + 1;
            if ПРАВЫЙСЫН[УЗЕЛ[ ≠ 0 then
              begin
                    УЗЕЛ ← ПРАВЫЙСЫН[УЗЕЛ];
                    goto левый
              end;
            if СТЕК не пуст then
              begin
                    УЗЕЛ ← элемент в вершине СТЕКа;
                    вытолкнуть из СТЕКа;
                    goto центр
              end
          end
```

Рис. 2.10. Алгоритм без рекурсии для внутреннего порядка.

узла v, который еще не занумерован (т. е. к ближайшему предку w узла v, такому, что v лежит в левом поддереве для w). Этот алгоритм приведен на рис. 2.10. □

Корректность варианта с рекурсией легко доказать индукцией по числу вершин в двоичном дереве. Корректность варианта без рекурсии также можно доказать, но здесь предположение индукции не столь прозрачно и возникают дополнительные трудности в связи с работой со стеком и правильным прохождением двоичного дерева. С другой стороны, платой за рекурсию может оказаться увеличение постоянных множителей у временной и емкостной сложностей.

Естественно выяснить теперь, как перевести алгоритмы с рекурсией в команды РАМ. В свете теоремы 1.2 достаточно рассмотреть построение РАСП-программы, так как РАСП можно моделировать с помощью РАМ с замедлением не более чем в постоянное число раз. Обсудим довольно прямолинейный способ реализации рекурсии. Он пригоден для всех программ, о которых идет речь в этой книге, но не охватывает всех случаев, которые могут встретиться.

В основе реализации процедуры с рекурсией лежит стек, где хранятся данные, участвующие во всех вызовах процедуры, при которых она еще не завершила свою работу. Иными словами, в стеке находятся все неглобальные данные. Стек разбит на *фрагменты стека*, представляющие собой блоки последовательных ячеек (регистров). Каждый вызов процедуры использует фрагмент стека, длина которого зависит от вызываемой процедуры.

Пусть сейчас выполняется процедура A, а стек выглядит, как на рис. 2.11. Если A вызывает процедуру B, происходит следующее:

1. В вершину стека помещается фрагмент стека нужного размера. В него входят в порядке, известном процедуре B,
   (а) указатели фактических параметров этого вызова процедуры B [1]),
   (б) пустое место для локальных переменных, участвующих в процедуре B,
   (в) адрес РАСП-команды в подпрограмме A, которую следует выполнить после того, как данный вызов B кончит работу *(адрес возврата)* [2]). Если B — функция, вырабатывающая некоторое значение, то во фрагмент стека для B также помещается указатель ячейки во фрагменте стека для A, в которую надлежит поместить это значение *(адрес значения)*.

[1]) Если фактическим параметром является выражение, то его значение вычисляется во фрагменте стека процедуры A, а указатель помещается во фрагмент для B. Если фактическим параметром является структура (например, массив), то достаточно будет указателя на ее первое слово.

[2]) Мы берем здесь РАСП в качестве модели именно из-за этих скачков к адресам возврата, доставляющих нам неудобства (хотя, конечно, преодолимые) при использовании РАМ.

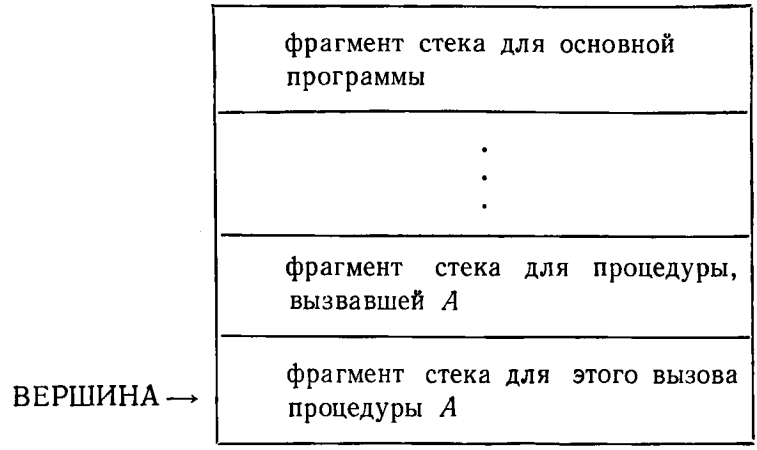


Рис. 2.11. Стек для вызовов рекурсивной процедуры.

2. Управление переходит к первой команде процедуры *B*. Адрес значения любого параметра или локального идентификатора, принадлежащего *B*, разыскивается с помощью индексации во фрагменте стека для *B*.

3. Когда процедура *B* кончает работу, управление передается процедуре *A* с помощью такой последовательности шагов:

(а) адрес возврата извлекается из вершины стека,

(б) если *B* — функция, то значение, обозначенное выражением из **return**-оператора, запоминается в ячейке, предписанной адресом значения в стеке,

(в) фрагмент стека процедуры *B* выталкивается из стека (это ставит в вершину стека фрагмент процедуры *A*),

(г) выполнение процедуры *A* возобновляется в ячейке, указанной в адресе возврата.

Пример 2.5. Рассмотрим процедуру ВНУТПОРЯДОК из алгоритма 2.1. Когда, например, она вызывает себя с фактическим параметром ЛЕВЫЙСЫН[УЗЕЛ], она запоминает в стеке адрес нового значения параметра УЗЕЛ вместе с адресом возврата, указывающим, что по окончании работы этого вызова выполнение программы продолжается со строки 2. Таким образом, переменная УЗЕЛ эффективно заменяется на ЛЕВЫЙСЫН[УЗЕЛ], где бы ни входил УЗЕЛ в это определение процедуры.

Описанную выше реализацию моделирует в некотором смысле соответствующий вариант без рекурсии (алгоритм 2.2). Однако там мы сделали так, что окончание выполнения вызова ВНУТПОРЯДОК с фактическим параметром ПРАВЫЙСЫН[УЗЕЛ] завершает выполнение и самой вызывающей процедуры. Поэтому нам не обяза-

тельно теперь хранить адрес возврата или УЗЕЛ в стеке, если фактическим параметром является ПРАВЫЙСЫН[УЗЕЛ]. □

Время, требуемое для вызова процедуры, пропорционально времени, требуемому для вычисления значений фактических параметров и запоминания указателей их значений в стеке. Время возвращения, разумеется, не превосходит этого времени. При подсчете времени, затрачиваемого несколькими рекурсивными процедурами, обычно легче всего оценить вес вызова процедуры, осуществляющей этот вызов. Тогда можно оценить сверху как функцию от размера входа разность времени, затрачиваемого на вызов каждой процедуры, и времени, затрачиваемого теми процедурами, которые она вызывает. Суммируя эти оценки по всем вызовам процедур, получаем верхнюю границу общего затраченного времени.

Для подсчета времени работы алгоритма с рекурсией применяются рекуррентные уравнения. С i-й процедурой связывается функция $T_i(n)$, обозначающая время выполнения i-й процедуры как функцию некоторого параметра n рассматриваемого входа. Обычно рекуррентное уравнение для $T_i(n)$ можно записать в терминах времен выполнения процедур, вызываемых процедурой i. Затем полученная система рекуррентных уравнений решается. Часто в алгоритме фигурирует только одна процедура, и $T(n)$ зависит лишь от значений $T(m)$ для конечного числа аргументов m, меньших n. В следующем разделе мы изучим решения некоторых часто встречающихся систем рекуррентных уравнений.

Напомним, что здесь, как и в других местах, весь анализ сложности (веса) основан на равномерной весовой функции. Если брать логарифмическую весовую функцию, то на анализе временной сложности может сказаться длина стека, используемого для реализации процедур с рекурсией.

2.6. РАЗДЕЛЯЙ И ВЛАСТВУЙ

Для решения той или иной задачи ее часто разбивают на части, находят их решения и затем из них получают решение всей задачи. Этот прием, особенно если его применять рекурсивно, часто приводит к эффективному решению задачи, подзадачи которой представляют собой ее меньшие версии. Проиллюстрируем эту технику на двух примерах, сопровождаемых анализом получающихся рекуррентных уравнений.

Рассмотрим задачу о нахождении наибольшего и наименьшего элементов множества S, содержащего n элементов. Для простоты будем считать, что n есть степень числа 2. Очевидный путь поиска наибольшего и наименьшего элементов состоит в том, чтобы искать их по отдельности. Например, следующая процедура находит наибольший элемент множества S, произведя $n-1$ сравнений его эле-

ментов:

```
begin
    MAX ← произвольный элемент из S;
    for все другие элементы x из S do
        if x > MAX then MAX ← x
end
```

Аналогично можно найти наименьший из остальных n—1 элементов, произведя n—2 сравнений. Итак, видим, что для нахождения наибольшего и наименьшего элементов при $n \geqslant 2$ потребуется $2n$—3 сравнений.

Применяя прием "разделяй и властвуй", мы разбили бы множество S на два подмножества S_1 и S_2 по $n/2$ элементов в каждом. Тогда описанный выше алгоритм нашел бы наибольший и наименьший элементы в каждой из двух половин с помощью рекурсии. Наибольший и наименьший элементы множества S можно было бы определить, произведя еще два сравнения — наибольших элементов в S_1 и S_2 и наименьших элементов в них. Сформулируем этот алгоритм более точно.

Алгоритм 2.3. Нахождение наибольшего и наименьшего элементов множества

Вход. Множество S из n элементов, где n — степень числа 2, $n \geqslant 2$.

Выход. Наибольший и наименьший элементы множества S.

Метод. К множеству S применяется рекурсивная процедура MAXMIN. Она имеет один аргумент [1]), представляющий собой множество S, такое, что $\|S\| = 2^k$ при некотором $k \geqslant 1$, и вырабатывает пару (a, b), где a — наибольший и b — наименьший элементы в S. Процедура MAXMIN приведена на рис. 2.12. □

Заметим, что сравнения элементов множества S происходят только на шаге 3, где сравниваются два элемента множества S, из которых оно состоит, и на шаге 7, где сравниваются max1 с max2 и min1 с min2. Пусть $T(n)$ — число сравнений элементов множества S, которые надо произвести в процедуре MAXMIN, чтобы найти наибольший и наименьший элементы n-элементного множества. Ясно, что $T(2)=1$. Если $n>2$, то $T(n)$ — общее число сравнений, произведенных в двух вызовах процедуры MAXMIN (строки 5 и 6), работающей на множествах размера $n/2$, и еще два сравнения в

[1]) Так как здесь подсчитываются только сравнения, способ прохождения аргументов несуществен. Однако, если множество S представлено массивом, можно организовать эффективный вызов MAXMIN, поставив указатели на первый и последний элементы подмножества S, состоящего из последовательных компонент этого массива.

```
   procedure MAXMIN(S):
1. if ‖S‖=2 then
      begin
2.       пусть S={a, b};
3.       return (MAX(a, b), MIN(a, b))
      end
   else
      begin
4.       разбить S на два равных подмножества S1 и S2;
5.       (max1, min1)←MAXMIN(S1);
6.       (max2, min2)←MAXMIN(S2);
7.       return (MAX(max1, max2), MIN(min1, min2))
      end
```

Рис. 2.12. Процедура для нахождения MAX и MIN.

строке 7. Таким образом,

$$T(n)=\begin{cases}1 & \text{при } n=2,\\ 2T(n/2)=2 & \text{при } n>2.\end{cases} \tag{2.2}$$

Решением рекуррентных уравнений (2.2) служит функция $T(n)={}^3/_2n-2$. Легко проверить, что эта функция удовлетворяет (2.2) при $n=2$, и если она удовлетворяет (2.2) при $n=m$, где $m\geqslant 2$, то

$$T(2m)=2\left(\frac{3m}{2}-2\right)+2=\frac{3}{2}(2m)-2,$$

т. е. она удовлетворяет (2.2) при $n=2m$. Таким образом, индукцией по n доказано, что $T(n)={}^3/_2n-2$ удовлетворяет (2.2), если n есть степень числа 2.

Можно показать, что для одновременного нахождения наибольшего и наименьшего элементов n-элементного множества надо сделать не менее ${}^3/_2n-2$ сравнений его элементов. Следовательно, алгоритм 2.3 оптимален в смысле числа сравнений между элементами из S, когда n есть степень числа 2.

В предыдущем примере прием "разделяй и властвуй" позволил уменьшить количество сравнений лишь в фиксированное число раз. В следующем примере мы с помощью этого приема уменьшим даже порядок роста сложности алгоритма.

Рассмотрим умножение двух n-разрядных двоичных чисел. Традиционный метод требует $O(n^2)$ битовых операций. В методе, изло-

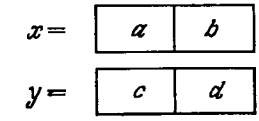


Рис. 2.13. Разбиение цепочек, представленных в виде двоичных чисел.

женном ниже, достаточно уже порядка $n^{\log 3}$, т. е. примерно $n^{1,59}$ битовых операций [1]).

Пусть x и y — два n-разрядных двоичных числа. Снова будем считать для простоты, что n есть степень числа 2. Разобьем x и y на две равные части, как показано на рис. 2.13. Если рассматривать каждую из этих частей как $(n/2)$-разрядное число, то можно представить произведение чисел x и y в виде

$$\begin{aligned} xy &= (a2^{n/2}+b)(c2^{n/2}+d) = \\ &= ac2^n + (ad+bc)\,2^{n/2} + bd. \end{aligned} \tag{2.3}$$

Равенство (2.3) дает способ вычисления произведения x и y с помощью четырех умножений $(n/2)$-разрядных чисел и нескольких сложений и сдвигов (умножений на степень числа 2). Произведение z чисел x и y также можно вычислить по следующей программе:

```
begin
    u ← (a + b) * (c + d);
    v ← a * c;
    w ← b * d;                                   (2.4)
    z ← v * 2^n + (u − v − w) * 2^(n/2) + w
end
```

На время забудем, что из-за переноса $a+b$ и $c+d$ могут иметь $n/2+1$ разрядов, и предположим, что они состоят лишь из $n/2$ разрядов. Наша схема для умножения двух n-разрядных чисел требует только трех умножений $(n/2)$-разрядных чисел и нескольких сложений и сдвигов. Для вычисления произведений u, v и w можно применять эту программу рекурсивно. Сложения и сдвиги занимают $O_Б(n)$ времени. Следовательно, временная сложность умножения двух n-разрядных чисел ограничена сверху функцией

$$T(n) = \begin{cases} k & \text{при } n = 1, \\ 3T(n/2) + kn & \text{при } n > 1, \end{cases} \tag{2.5}$$

где k — постоянная, отражающая сложение и сдвиги в выражениях, входящих в (2.4). Решение рекуррентных уравнений (2.5) ограничено сверху функцией

$$3kn^{\log 3} \approx 3kn^{1,59}.$$

[1]) Напомним, что, если не оговорено противное, все логарифмы в этой книге берутся по основанию 2.

На самом деле можно показать, что в (2.5)

$$T(n) = 3kn^{\log 3} - 2kn.$$

Доказательство проведем индукцией по n, где n — степень числа 2. Базис, т. е. случай $n=1$, тривиален. Если функция $T(n) = 3kn^{\log 3} - 2kn$ удовлетворяет (2.5) при $n=m$, то

$$\begin{aligned} T(2m) &= 3T(m) + 2km = \\ &= 3[3km^{\log 3} - 2km] + 2km = \\ &= 3k(2m)^{\log 3} - 2k(2m), \end{aligned}$$

так что она удовлетворяет (2.5) и при $n=2m$. Отсюда следует, что $T(n) \leqslant 3kn^{\log 3}$. Заметим, что попытка использовать в индукции $3kn^{\log 3}$ вместо $3kn^{\log 3} - 2kn$ не проходит.

Для завершения описания алгоритма умножения мы должны учесть, что числа $a+b$ и $c+d$, вообще говоря, имеют $n/2+1$ разрядов, и поэтому произведение $(a+b)(c+d)$ нельзя вычислить непосредственным рекурсивным применением нашего алгоритма к задаче размера $n/2$. Вместо этого надо записать $a+b$ в виде $a_1 2^{n/2} + b_1$, где $a_1 = 0$ или 1. Аналогично запишем $c+d$ в виде $c_1 2^{n/2} + d_1$. Тогда произведение $(a+b)(c+d)$ можно представить в виде

$$a_1 c_1 2^n + (a_1 d_1 + b_1 c_1) 2^{n/2} + b_1 d_1. \tag{2.6}$$

Слагаемое $b_1 d_1$ вычисляется с помощью рекурсивного применения нашего алгоритма умножения к задаче размера $n/2$. Остальные умножения в (2.6) можно сделать за время $O_Б(n)$, поскольку они содержат в качестве одного из аргументов либо единственный бит a_1 или c_1, либо степень числа 2.

Пример 2.6. Этот асимптотически быстрый алгоритм умножения целых чисел можно применять не только к двоичным, но и к десятичным числам. Проиллюстрируем это на примере:

$$\begin{array}{lll} x = 3141 & a = 31 & c = 59 \\ y = 5927 & b = 41 & d = 27 \\ & a+b = 72 & c+d = 86 \end{array}$$

$u = (a+b)(c+d) = 72 \times 86 = 6192$

$v = ac = 31 \times 59 = 1829$

$w = bd = 41 \times 27 = 1107$

$xy = 18290000^{1}) + (6192 - 1829 - 1107) \times 100 + 1107 = 18616707.$ □

Заметим, что алгоритм, основанный на (2.4), заменял одно умножение тремя сложениями и вычитанием (ср. с (2.3)). Почему такая замена приводит к асимптотической эффективности, можно интуи-

[1]) Число v следует сдвинуть на четыре десятичных разряда, а u—v—w — на два.

тивно объяснить тем, что умножение выполнить труднее, чем сложение, и для достаточно больших n любое фиксированное число n-разрядных сложений требует меньше времени, чем n-разрядное умножение, независимо от того, какой из (известных) алгоритмов применяется. Вначале кажется, что уменьшение числа $(n/2)$-разрядных произведений с четырех до трех может уменьшить общее время в лучшем случае на 25%. Однако соотношения (2.4) применяются рекурсивно для вычисления $(n/2)$-разрядных, $(n/4)$-разрядных и т. д. произведений. Эти 25-процентные уменьшения объединяются и дают в результате асимптотическое улучшение временной сложности.

Временная сложность процедуры определяется числом и размером подзадач и в меньшей степени работой, необходимой для разбиения данной задачи на подзадачи. Так как рекуррентные уравнения вида (2.2) и (2.5) часто возникают при анализе рекурсивных алгоритмов типа „разделяй и властвуй", рассмотрим решение таких уравнений в общем виде.

Теорема 2.1. *Пусть a, b и c — неотрицательные постоянные. Решение рекуррентных уравнений*

$$T(n) = \begin{cases} b & \text{при } n = 1, \\ aT(n/c) + bn & \text{при } n > 1, \end{cases}$$

где n — степень числа c, имеет вид,

$$T(n) = \begin{cases} O(n), & \text{если } a < c, \\ O(n \log n), & \text{если } a = c, \\ O(n^{\log_c a}), & \text{если } a > c. \end{cases}$$

Доказательство. Если n — степень числа c, то

$$T(n) = bn \sum_{i=0}^{\log_c n} r^i, \text{ где } r = a/c.$$

Если $a<c$, то $\sum_{i=0}^{\infty} r^i$ сходится и, следовательно, $T(n)=O(n)$.

Если $a=c$, то каждым членом этого ряда будет 1, а всего в нем $O(\log n)$ членов. Поэтому $T(n)=O(n \log n)$. Наконец, если $a>c$, то

$$bn \sum_{i=0}^{\log_c n} r^i = bn \frac{r^{1+\log_c n} - 1}{r - 1},$$

что составляет $O(a^{\log_c n})$, или $O(n^{\log_c a})$. □

Из теоремы 2.1 вытекает, что разбиение задачи размера n (за линейное время) на две подзадачи размера $n/2$ дает алгоритм сложности $O(n \log n)$. Если бы подзадач было 3, 4 или 8, то получился бы алгоритм сложности порядка $n^{\log 3}$, n^2 или n^3 соответственно.

С другой стороны, разбиение задачи на 4 подзадачи размера $n/4$ дает алгоритм сложности $O(n \log n)$, а на 9 и 16 — порядка $n^{\log 3}$ и n^2 соответственно. Поэтому асимптотически более быстрый алгоритм умножения целых чисел можно было бы получить, если бы удалось так разбить исходные целые числа на 4 части, чтобы суметь выразить исходное умножение через 8 или менее меньших умножений. Другой тип рекуррентных соотношений возникает в случае, когда работа по разбиению задачи не пропорциональна ее размеру. Некоторые типы рекуррентных соотношений вынесены в упражнения.

Если n не является степенью числа c, то обычно можно вложить задачу размера n в задачу размера n', где n' — наименьшая степень числа c, большая или равная n. Поэтому порядки роста, указанные в теореме 2.1, сохраняются для любого n. На практике часто можно разработать рекурсивные алгоритмы, разбивающие задачи произвольного размера на c равных частей, где c велико, насколько возможно. Эти алгоритмы, как правило, эффективнее (на постоянный множитель) тех, которые получаются путем представления размера входа в виде ближайшей сверху степени числа c.

2.7. БАЛАНСИРОВКА

В обоих наших примерах на технику "разделяй и властвуй" задача разбивалась на подзадачи равных размеров. Это не было случайностью. Поддержание равновесия — основной руководящий принцип при разработке хорошего алгоритма. Для иллюстрации этого принципа мы приведем пример из сортировки и сопоставим эффекты от разбиения задачи на подзадачи неравных размеров и на подзадачи равных размеров. Из этого примера не следует заключать, что "разделяй и властвуй" — единственная техника, в которой полезна балансировка. В гл. 4 приводится несколько примеров, в которых эффективные алгоритмы получаются в результате балансировки размеров поддеревьев или весов двух операций.

Рассмотрим задачу расположения целых чисел в порядке неубывания. По-видимому, простейший способ сделать это — найти наименьший элемент, исследуя всю последовательность и затем меняя местами наименьший элемент с первым. Процесс повторяется на остальных n—1 элементах, и это повторение приводит к тому, что второй наименьший элемент оказывается на втором месте. Повторение процесса на остальных n—2, n—3, . . . , 2 элементах сортирует всю последовательность.

Этот алгоритм приводит к рекуррентным уравнениям

$$T(n) = \begin{cases} 0 & \text{при } n = 1, \\ T(n-1) + n - 1 & \text{при } n > 1 \end{cases} \qquad (2.7)$$

для числа сравнений, произведенных между сортируемыми элементами. Решением для (2.7) служит $T(n)=n(n-1)/2$, что составляет $O(n^2)$.

Хотя этот алгоритм можно считать рекурсивным применением приема "разделяй и властвуй" с разбиением задачи на неравные части, он не эффективен для больших n. Для разработки асимптотически эффективного алгоритма сортировки надо позаботиться о сбалансированности. Вместо того чтобы разбивать задачу размера n на две подзадачи, одна из которых имеет размер 1, а другая — размер $n-1$, надо разбить ее на две подзадачи с размерами примерно $n/2$. Это выполняется методом, известным как *сортировка слиянием*.

Рассмотрим последовательность целых чисел $x_1, x_2, \ldots, x_n$. Снова предположим для простоты, что n есть степень числа 2. Один из способов упорядочить эту последовательность — разбить ее на две подпоследовательности $x_1, x_2, \ldots, x_{n/2}$ и $x_{(n/2)+1}, \ldots, x_n$, упорядочить каждую из них и затем слить их. Под "слиянием" мы понимаем объединение двух уже упорядоченных последовательностей в одну упорядоченную последовательность.

Алгоритм 2.4. Сортировка слиянием

Вход. Последовательность чисел $x_1, x_2, \ldots, x_n$, где n — степень числа 2.

Выход. Последовательность $y_1, y_2, \ldots, y_n$, являющаяся перестановкой входа и удовлетворяющая неравенствам $y_1 \leqslant y_2 \leqslant \ldots \leqslant y_n$.

Метод. Применим процедуру СЛИЯНИЕ(S, T), входом которой служат две упорядоченные последовательности S и T, а выходом — последовательность элементов из S и T, расположенных в порядке неубывания. Поскольку S и T сами упорядочены, СЛИЯНИЕ требует сравнений не больше, чем сумма длин S и T без единицы. Работа этой процедуры состоит в выборе большего из наибольших элементов, остающихся в S и T, и в последующем удалении выбранного элемента. В случае совпадения можно отдавать предпочтение последовательности S.

```
procedure СОРТ(i, j):
if i = j then retu   x_i
else
  begin
      m ← (i + j − 1)/2;
      return СЛИЯНИЕ(СОРТ(i, m), СОРТ(m + 1, j))
  end
```

Рис. 2.14. Сортировка слиянием.

Кроме того, применяется процедура COPT(i, j) (рис. 2.14), сортирующая подпоследовательность x_i, x_{i+1}, . . . , x_j в предположении, что она имеет длину 2^k для некоторого $k \geqslant 0$.

Для сортировки данной последовательности x_1, x_2, . . . , x_n вызывается процедура COPT(1, n). □

Подсчет числа сравнений в алгоритме 2.4 приводит к рекуррентным уравнениям

$$T(n) = \begin{cases} 0 & \text{при } n = 1, \\ 2T(n/2) + n - 1 & \text{при } n > 1, \end{cases}$$

решением которых по теореме 2.1 является $T(n) = O(n \log n)$. Для больших n сбалансированность размеров подзадач дала значительную выгоду. Аналогичный анализ показывает, что общее время работы процедуры COPT, затрачиваемое не только на сравнения, также есть $O(n \log n)$.

2.8. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Рекурсивная техника полезна, если задачу можно разбить на подзадачи за разумное время, а суммарный размер подзадач будет небольшим. Из теоремы 2.1 вытекает, что если сумма размеров подзадач равна an для некоторой постоянной $a > 1$, то рекурсивный алгоритм, вероятно, имеет полиномиальную временную сложность. Но если очевидное разбиение задачи размера n сводит ее к n задачам размера $n-1$, то рекурсивный алгоритм, вероятно, имеет экспоненциальную сложность. В этом случае часто можно получить более эффективные алгоритмы с помощью табличной техники, называемой *динамическим программированием*.

Динамическое программирование, в сущности, вычисляет решение для всех подзадач. Вычисление идет от малых подзадач к бóльшим, и ответы запоминаются в таблице. Преимущество этого метода состоит в том, что раз уж подзадача решена, ее ответ где-то хранится и никогда не вычисляется заново. Эту технику легко понять на простом примере.

Рассмотрим вычисление произведения n матриц

$$M = M_1 \times M_2 \times \ldots \times M_n,$$

где M_i — матрица с r_{i-1} строками и r_i столбцами. Порядок, в котором эти матрицы перемножаются, может существенно сказаться на общем числе операций, требуемых для вычисления M, независимо от алгоритма, применяемого для умножения матриц.

Пример 2.7. Предположим, что умножение $(p \times q)$-матрицы на $(q \times r)$-матрицу требует pqr операций, как это и бывает в "обычном"

алгоритме, и рассмотрим произведение

$$M = \underset{[10\times 20]}{M_1} \times \underset{[20\times 50]}{M_2} \times \underset{[50\times 1]}{M_3} \times \underset{[1\times 100]}{M_4}, \tag{2.8}$$

где размеры каждой матрицы M_i указаны в скобках. Если вычислять M в порядке

$$M_1\times(M_2\times(M_3\times M_4)),$$

то потребуется 125 000 операций, тогда как вычисление M в порядке

$$(M_1\times(M_2\times M_3))\times M_4$$

занимает лишь 2 200 операций. □

Процесс перебора всех порядков, в которых можно вычислить рассматриваемое произведение n матриц, с целью минимизировать число операций имеет экспоненциальную сложность (упр. 2.31), что при больших n практически неприемлемо. Однако динамическое программирование приводит к алгоритму сложности $O(n^3)$. Пусть m_{ij} — минимальная сложность вычисления $M_i\times M_{i+1}\times\ldots\times M_j$, $1\leqslant i\leqslant j\leqslant n$. Очевидно, что

$$m_{ij} = \begin{cases} 0, & \text{если } i=j, \\ \underset{i\leqslant k<j}{\text{MIN}}\,(m_{ik}+m_{k+1,j}+r_{i-1}r_kr_j), & \text{если } j>i. \end{cases} \tag{2.9}$$

Здесь m_{ik} — минимальная сложность вычисления $M'=M_i\times\times M_{i+1}\times\ldots\times M_k$, а $m_{k+1,j}$ — минимальная сложность вычисления

$$M''=M_{k+1}\times M_{k+2}\times\ldots\times M_j.$$

Третье слагаемое равно сложности умножения M' на M''. Заметим, что M' — матрица размера $r_{i-1}\times r_k$, а M'' — матрица размера $r_k\times$

```
    begin
1.    for i ← 1 until n do m_ii ← 0;
2.    for l ← 1 until n — 1 do
3.      for i ← 1 until n — 1 do
          begin
4.          j ← i + l;
5.          m_ij ← MIN_{i≤k<j} (m_ik + m_{k+1,j} + r_{i-1} * r_k * r_j)
          end;
6.    write m_1n
    end
```

Рис. 2.15. Алгоритм динамического программирования для определения порядка умножения матриц.

$m_{11}=0$	$m_{22}=0$	$m_{33}=0$	$m_{44}=0$
$m_{12}=10000$	$m_{23}=1000$	$m_{34}=5000$	
$m_{13}=1200$	$m_{24}=3000$		
$m_{14}=2200$			

Рис. 2.16. Сложности вычисления произведений $M_i \times M_{i+1} \times \ldots \times M_j$.

$\times r_j$. В (2.9) утверждается, что m_{ij} $(j>i)$ — наименьшая из сумм этих трех членов по всем возможным значениям k, лежащим между i и j—1.

При динамическом программировании m_{ij} вычисляются в порядке возрастания разностей нижних индексов. Начинают с вычисления m_{ii} для всех i, затем $m_{i,\,i+1}$ для всех i, потом $m_{i,\,i+2}$ и т. д. При этом m_{ik} и $m_{k+1,\,j}$ в (2.9) будут уже вычислены, когда мы приступим к вычислению m_{ij}. Это следует из того, что при $i \leqslant k < j$ разность j—i должна быть больше k—i, а также и j—$(k+1)$. Приведем теперь алгоритм.

Алгоритм 2.5. Алгоритм динамического программирования для вычисления порядка, минимизирующего сложность умножения цепочки из n матриц $M_1 \times M_2 \times \ldots \times M_n$

Вход. $r_0, r_1, \ldots, r_n$, где r_{i-1} и r_i — числа строк и столбцов матрицы M_i.

Выход. Минимальная сложность умножения матриц M_i в предположении, что для умножения $(p \times q)$-матрицы на $(q \times r)$-матрицу требуется pqr операций.

Метод. Алгоритм, приведенный на рис. 2.15. □

Пример 2.8. Если применить этот алгоритм к цепочке из четырех матриц (2.8), где $r_0, \ldots, r_4$ равны соответственно 10, 20, 50, 1, 100, то в результате вычислений получатся значения m_{ij}, приведенные на рис. 2.16. Таким образом, минимальное число операций, требуемых для вычисления этого произведения, равно 2 200. Порядок, в котором можно произвести эти умножения, легко определить, приписав каждой клетке таблицы то значение k, на котором достигается минимум в (2.9). □

2.9. ЭПИЛОГ

В этой главе мы изложили ряд основных методов, которыми пользуются при разработке эффективных алгоритмов. Было показано, как структуры высокого уровня — списки, очереди и стеки — избавляют разработчика алгоритмов от скучной работы (например,

с указателями) и позволяют сосредоточиться на общей структуре самого алгоритма. Кроме того, мы увидели, как мощная техника рекурсии и динамического программирования часто приводит к элегантным и естественным алгоритмам. Мы познакомились также с некоторыми общими приемами, такими, как "разделяй и властвуй" и балансировка.

Эти методы, разумеется, не являются единственно доступными, но они принадлежат к числу важнейших. Далее в этой книге мы изложим и другие примеры. Они будут относиться к различным задачам — от выбора подходящего представления до нахождения разумного порядка выполнения операций. Возможно, важнейшим принципом хорошего разработчика алгоритмов должно быть чувство неудовлетворенности. Разработчику следует изучать задачу с разных точек зрения, пока он не убедится, что получил алгоритм, наиболее подходящий для его целей.

УПРАЖНЕНИЯ

2.1. Выберите реализацию для дважды связанного списка. Напишите алгоритм на Упрощенном Алголе для вставки и удаления элемента. Убедитесь, что ваши программы работают, когда надо удалить первый и (или) последний элементы и когда список пуст.

2.2. Напишите алгоритм для обращения порядка элементов в списке. Докажите, что он работает правильно.

2.3. Напишите алгоритмы, реализующие операции ЗАТОЛКНУТЬ, ВЫТОЛКНУТЬ, ВПИСАТЬ и ВЫПИСАТЬ, упомянутые в разд. 2.1. Не забывайте проверять, достиг ли указатель конца массива, зарезервированного для стека или очереди.

2.4. Напишите условия для проверки пустоты очереди. Допустим, что массив ИМЯ из разд. 2.1 имеет размер k. Сколько элементов можно хранить в соответствующей очереди? Сделайте рисунки, иллюстрирующие эту очередь и типичные положения указателей ПЕРЕДНИЙ и ЗАДНИЙ в случае, когда очередь (а) пуста, (б) содержит один элемент и (в) полна.

2.5. Напишите алгоритм для удаления первого ребра (v, w) из списка смежностей для v в неориентированном графе. Считайте, что списки смежностей дважды связаны и что СВЯЗЬ помещает v в список смежностей узла w, как описано в разд. 2.3

2.6. Напишите алгоритм для построения списков смежностей для неориентированного графа. Каждое ребро (v, w) надо представить дважды: в списках смежностей для v и для w. Оба экземпляра каждого ребра должны связываться между собой так, что если один из них вычеркивается, другой также можно легко вычеркнуть.

*2.7. *Топологическая сортировка.* Пусть $G=(V, E)$ — ориентированный ациклический граф. Напишите алгоритм, который бы так приписывал целые числа узлам в G, что если из узла с номером i в узел с номером j идет ориентированное ребро, то $i<j$. *Указание*: Ациклический граф должен иметь узел, в который не входит ни одно ребро. Почему? Одно из решений этой задачи таково: найти узел, в который не входят ребра; приписать ему наименьший номер и удалить его из графа вместе со всеми ребрами, выходящими из него. Повторить этот процесс для графа, полученного в результате такого удаления, приписывая следующий наименьший номер, и т. д. Для того чтобы сделать этот алгоритм эффективным, т. е. чтобы его сложность была $O(\|E\|+\|V\|)$, нужно, чтобы в поисках узла, в который не входит ни одно ребро, не приходилось просматривать каждый новый граф.

*2.8. Пусть $G=(V, E)$ — ориентированный ациклический граф с двумя выделенными узлами — начальным и целевым. Напишите алгоритм для нахождения такого множества путей из начального узла в целевой, чтобы

1) ни один узел, кроме начального и целевого, не лежал на двух путях,

2) к нему нельзя было добавить ни одного нового пути, не нарушив условия 1.

Заметим, что этим условиям удовлетворяет много множеств. Не обязательно находить множество с наибольшим числом путей, достаточно какого-нибудь множества, удовлетворяющего приведенным выше условиям. Ваш алгоритм должен иметь временную сложность $O(\|E\|+\|V\|)$.

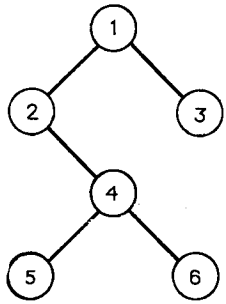
*2.9. *Задача об устойчивом бракосочетании.* Пусть B — множество из n юношей, а G — множество из n девушек. Каждый юноша оценивает девушек числами от 1 до n, и каждая девушка оценивает юношей числами от 1 до n. *Паросочетанием* называется взаимно однозначное соответствие между юношами и девушками. Паросочетание *устойчиво*, если для любых двух юношей b_1 и b_2 и соответствующих им в этом паросочетании девушек g_1 и g_2 выполняются следующие два условия:

1) либо b_1 оценивает g_1 выше, чем g_2, либо g_2 оценивает b_2 выше, чем b_1;

2) либо b_2 оценивает g_2 выше, чем g_1, либо g_1 оценивает b_1 выше, чем b_2.

Докажите, что устойчивое паросочетание всегда существует, и напишите алгоритм для нахождения одного из таких паросочетаний.

2.10. Рассмотрим двоичное дерево, узлам которого приписаны имена. Напишите алгоритм, печатающий эти имена в (а) прямом порядке, (б) обратном и (в) внутреннем.



	ЛЕВЫЙСЫН	ПРАВЫЙСЫН
1	2	3
2	0	4
3	0	0
4	5	6
5	0	0
6	0	0

Рис. 2.17. Двоичное дерево.

*2.11. Напишите алгоритм для вычисления значений арифметических выражений, содержащих операции + и × и представленных в (а) префиксной польской записи, (б) инфиксной записи и (в) постфиксной польской записи.

*2.12. Разработайте технику, которая позволяла бы полагать равными нулю те элементы матрицы, которые выбираются первый раз: таким способом устраняется работа по исходной записи матрицы смежностей, занимающая время $O(\|V\|^2)$. *Указание*: Устанавливайте в каждом элементе в момент первого присваивания ему значения указатель на обратный указатель во вспомогательном стеке. Всякий раз, когда выбирается какой-то элемент, проверяйте, чтобы он имел не случайное значение; для этого надо убедиться, что его указатель указывает в активную часть стека, а соответствующий обратный указатель — на выбираемый элемент.

2.13. Рассмотрите работу алгоритмов 2.1 и 2.2 на двоичном дереве рис. 2.17.

*2.14. Докажите, что алгоритм 2.2 корректен.

*2.15. *Ханойские башни*. Имеются три стержня A, B и C и n дисков разных размеров. Вначале все диски нанизаны на стержень A в порядке убывания размеров, так что наибольший диск находится внизу. Надо так переместить диски со стержня A на стержень B, чтобы никакой больший диск ни разу не оказался над меньшим; за один шаг можно перемещать только один диск. Стержень C можно использовать для временного хранения дисков. Напишите рекурсивный алгоритм решения этой задачи. Каково время работы вашего алгоритма в терминах числа перемещений дисков?

**2.16. Решите упр. 2.15 с помощью алгоритма без рекурсии. Какой из алгоритмов легче понять и корректность какого из них легче доказать?

*2.17. Докажите, что для решения упр. 2.15 необходимо и достаточно 2^n—1 перемещений.

2.18. Напишите алгоритм порождения всех перестановок целых чисел от 1 до n. *Указание*: Множество перестановок целых чисел от 1 до n можно получить из множества перестановок целых чисел от 1 до $n-1$, вставляя n во все возможные позиции в каждой перестановке.

2.19. Напишите алгоритм нахождения высоты двоичного дерева, представленного на рис. 2.7, *б*.

2.20. Напишите алгоритм подсчета числа потомков каждого узла дерева.

****2.21.** Рассмотрим двухпозиционный переключатель с двумя входами и двумя выходами, показанный на рис. 2.18. В одной позиции входы 1 и 2 соединяются соответственно с выходами 1 и 2, в другой — с выходами 2 и 1. Используя эти переключатели, разработайте сеть с n входами и n выходами, на которой можно получить любую из $n!$ возможных перестановок входов. В вашей сети должно быть не более $O(n \log n)$ переключателей. *Указание*: Примените прием "разделяй и властвуй".

2.22. Напишите РАСП-программу, которая выполняла бы работу следующей программы, вычисляющей $\binom{n}{m}$:

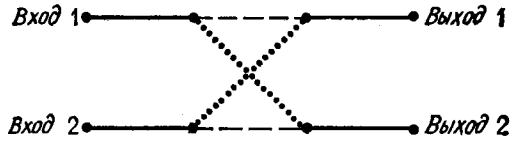
procedure СОЧЕТ (n, m):
if $m=0$ *или* $n=m$ **then return** 1
else return (СОЧЕТ ($n-1$, m)+СОЧЕТ ($n-1$, $m-1$))

В стеке можно хранить текущие значения n и m, а также адреса возврата и значения, когда происходят вызовы.

***2.23.** В ряде ситуаций задачу размера n выгодно разбивать на $\sqrt{n}$ подзадач размера $\sqrt{n}$. В результате получается рекуррентное уравнение вида

$$T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2,$$

где r — целое число, $r \geqslant 1$. Покажите, что решение этого рекуррентного уравнения есть $O(n(\log n)^r \log\log n)$.



— — — *Соединения в 1-й позиции*

····· *Соединения во 2-й позиции*

Рис. 2.18. Двухпозиционный переключатель.

2.24. Вычислите значения сумм:

$$\text{(а)}\ \sum_{i=1}^{n} i, \qquad \text{(б)}\ \sum_{i=1}^{n} a^i, \qquad \text{(в)}\ \sum_{i=1}^{n} i a^i,$$

$$\text{(г)}\ \sum_{i=1}^{k} 2^{k-i} i^2, \qquad \text{(д)}\ \sum_{i=0}^{n} \binom{n}{i}, \qquad \text{(е)}\ \sum_{i=1}^{n} i \binom{n}{i}.$$

***2.25.** Решите следующие рекуррентные уравнения, считая, что $T(1)=1$:

(а) $T(n)=aT(n-1)+bn$, (б) $T(n)=T(n/2)+bn \log n$,
(в) $T(n)=aT(n-1)+bn^c$, (г) $T(n)=aT(n/2)+bn^c$.

***2.26.** Модифицируйте алгоритм 2.3 для нахождения наибольшего и наименьшего элементов множества, разрешив рекурсии идти до уровня $\|S\|=1$. Какова асимптотическая скорость роста числа сравнений?

****2.27.** Покажите, что для одновременного нахождения наибольшего и наименьшего элементов n-элементного множества необходимо и достаточно $\lceil 3/2 n - 2 \rceil$ сравнений.

***2.28.** Модифицируйте алгоритм для умножения целых чисел с помощью разбиения каждого целого числа на (а) три и (б) четыре части. Какова сложность каждого из ваших алгоритмов?

***2.29.** Пусть A — массив размера n, состоящий из положительных и отрицательных целых чисел, причем $A[1]<A[2]<\ldots<A[n]$. Напишите алгоритм нахождения числа i, для которого $A[i]=i$ (если такое i существует). Каков порядок времени работы вашего алгоритма? Докажите, что $O_C(\log n)$ — наилучший возможный порядок.

****2.30.** Для n, не являющегося степенью числа 2, также можно получить из алгоритма 2.4 корректный алгоритм сортировки слиянием, если заменить оператор $m \leftarrow (i+j-1)/2$ на рис. 2.14 оператором $m \leftarrow \lfloor (i+j)/2 \rfloor$. Пусть $T(n)$ — число сравнений, требуемых для сортировки этим методом n элементов.

(а) Покажите, что

$$T(1)=0,$$
$$T(n)=T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+n-1.$$

(б) Покажите, что решением этого рекуррентного уравнения служит функция

$$T(n)=n\lceil \log n \rceil - 2^{\lceil \log n \rceil}+1.$$

*2.31. Покажите, что решением рекуррентного уравнения

$$X(1)=1.$$
$$X(n)=\sum_{i=1}^{n-1} X(i)\,X(n-i) \quad \text{при } n>1$$

служит

$$X(n+1)=\frac{1}{n+1}\binom{2n}{n}.$$

$X(n)$ — это число способов правильной расстановки скобок в цепочке из n символов. Числа $X(n)$ называются числами Каталана. Покажите, что $X(n)\geqslant 2^{n-2}$.

2.32. Модифицируйте алгоритм 2.5 так, чтобы он выдавал порядок, в котором следует умножать матрицы, чтобы минимизировать число умножений их элементов.

2.33. Напишите эффективный алгоритм для определения порядка, в котором следует вычислять произведение матриц $M_1\times M_2\times \ldots\times M_n$, чтобы минимизировать число умножений элементов в случае, когда каждая матрица M_i имеет один из размеров 1×1, $1\times d$, $d\times 1$ или $d\times d$ при некотором фиксированном d.

Определение. *Бесконтекстной грамматикой* G в нормальной форме Хомского называется четверка (N, Σ, P, S), где (1) N — конечное множество *нетерминальных символов*, (2) Σ — конечное множество *терминальных символов*, (3) P — конечное множество пар, называемых *продукциями*, вида $A\to BC$ или $A\to a$ (A, B, C — символы из N, а a — из Σ) и (4) S — выделенный символ из N. Мы будем писать $\alpha A\gamma\Rightarrow\alpha\beta\gamma$, если α, β, γ — цепочки, состоящие из терминальных и нетерминальных символов, и $A\to\beta$ принадлежит P. *Языком* $L(G)$, *порождаемым грамматикой* G, называется множество $\{w\,|\,S\Rightarrow^* w\}$ цепочек терминальных символов, где $\Rightarrow^*$ обозначает рефлексивное и транзитивное замыкание отношения $\Rightarrow$.

*2.34. Напишите алгоритм сложности $O(n^3)$, который определял бы принадлежность данной цепочки $w=a_1a_2\ldots a_n$ языку $L(G)$, где $G=(N, \Sigma, P, S)$ — бесконтекстная грамматика в нормальной форме Хомского. *Указание*: Пусть $m_{ij}=\{A\,|\,A\in N \text{ и } A\Rightarrow^* a_ia_{i+1}\ldots a_j\}$. Слово w принадлежит языку $L(G)$ тогда и только тогда, когда $S\in m_{1n}$. Для вычисления m_{ij} примените динамическое программирование.

*2.35. Пусть x и y — цепочки символов и некоторого алфавита. Рассмотрите операции удаления символа из x, вставки символа в x и замены символа в x другим символом. Опишите алгоритм нахождения минимального числа таких операций, необходимых для преобразования x в y.

Замечания по литературе

Дополнительную информацию о структурах данных и их реализации можно найти у Кнута [1968] и Стоуна [1972]. Книга Пратта [1975] содержит описание реализации рекурсии в языках, подобных Алголу. Берж [1958] и Харари [1969] излагают теорию графов. Кнут [1968] дает информацию о деревьях и их прохождении. Дальнейшие сведения об алгоритмах прохождения деревьев можно найти у Беркхарда [1973].

Оптимальность алгоритма 2.3 (для нахождения максимума и минимума) показал Поль [1972]. Алгоритм сложности $O(n^{1,59})$ для умножения целых чисел (разд. 2.6) приведен в работе Карацубы, Офмана [1962]. Виноград [1973] рассматривает подобные приемы ускорения с более общей точки зрения.

Понятие динамического программирования популяризировал Беллман [1957], а алгоритм 2.5 — известное приложение этого понятия опубликованное Годбоулом [1973] и Мураокой, Кукком [1973]. Приложением динамического программирования к распознаванию принадлежности бесконтекстному языку (упр. 2.34) независимо друг от друга занимались Касами [1965], Янгер [1967] и Кок. В работе Вагнера, Фишера [1974] содержится решение упр. 2.35.

Дальнейшую информацию о решении рекуррентных уравнений см. Лю [1968] или Слоун [1973].

3 СОРТИРОВКА И ПОРЯДКОВЫЕ СТАТИСТИКИ

В этой главе изучаются две взаимосвязанные задачи — сортировка последовательности элементов и выбор k-го наименьшего элемента последовательности. Под сортировкой последовательности мы подразумеваем переупорядочение ее элементов в такую последовательность, где все элементы расположены в порядке невозрастания или неубывания. Можно найти k-й наименьший элемент последовательности, расположив ее элементы в порядке неубывания и выбрав из полученной последовательности k-й элемент. Однако мы увидим, что выбрать k-й наименьший элемент можно быстрее.

Сортировка — практически важная и теоретически интересная задача. Поскольку сортировка больших объемов данных составляет значительную часть коммерческой обработки данных, эффективные алгоритмы для сортировки важны и с экономической точки зрения. Что же касается разработки алгоритмов, то даже здесь процесс сортировки последовательности элементов очень важен — он является существенной частью многих алгоритмов.

В данной главе рассматриваются два класса сортирующих алгоритмов. Первый основан на использовании структуры сортируемых элементов. Например, если сортируемые элементы представлены числами от 0 до m—1, то можно упорядочить последовательность из n элементов за время $O(n+m)$, а если словами в некотором алфавите, то за время, пропорциональное сумме длин этих слов.

Второй класс алгоритмов не предполагает у сортируемых элементов никакой структуры. Основная операция — сравнение двух элементов. В алгоритмах этого типа, как мы увидим, для сортировки последовательности, состоящей из n элементов, требуется по крайней мере $n \log n$ сравнений. Мы дадим два сортирующих алгоритма сложности $O_C(n \log n)$, а именно Сортдеревом со сложностью $O_C(n \log n)$ в худшем случае и Быстрсорт со средней сложностью $O_C(n \log n)$.

3.1. ЗАДАЧА СОРТИРОВКИ

Определение. *Частичным порядком* на множестве S называется такое двухместное отношение R, что для любых a, b и c из S

1) aRa (R рефлексивно),
2) из aRb и bRc следует aRc (R транзитивно),
3) из aRb и bRa следует $a=b$ (R антисимметрично).

Примеры частичных порядков — отношение $\leqslant$ на множестве целых чисел и отношение $\subseteq$ (включение множеств).

Линейным, или *полным*, *порядком* на множестве S называется такой частичный порядок R на S, что для любых двух элементов a, b выполняется либо aRb, либо bRa.

Отношение $\leqslant$ на множестве целых чисел является линейным порядком [1]), а включение множеств $\subseteq$ нет.

Задачу сортировки можно сформулировать так: дана последовательность из n элементов $a_1, a_2, \ldots, a_n$, выбранных из множества, на котором задан линейный порядок (обычно мы будем обозначать его $\leqslant$). Требуется найти перестановку π этих n элементов, которая отобразит данную последовательность в неубывающую последовательность $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$, т. е. $a_{\pi(i)} \leqslant a_{\pi(i+1)}$ при $1 \leqslant i < n$. Как правило, мы будем вырабатывать саму упорядоченную последовательность, а не упорядочивающую перестановку π.

Методы сортировки классифицируются на *внутренние* (когда данные размещаются в памяти с произвольным доступом) и *внешние* (когда данные размещаются преимущественно вне памяти с произвольным доступом). Внешняя сортировка составляет часть таких приложений, как обработка расчетов, при которой в работу вовлекается гораздо больше элементов, чем можно сразу запомнить в оперативной памяти. Поэтому методы внешней сортировки, применимые к данным, находящимся на внешних устройствах памяти (таких, как диски или магнитные ленты), имеют огромное коммерческое значение.

Внутренняя сортировка важна как для разработки алгоритмов, так и для коммерческих приложений. В тех случаях, когда сортировка возникает как часть другого алгоритма, число сортируемых элементов обычно мало, и они помещаются в оперативную память. Тем не менее мы будем предполагать, что элементов, подлежащих сортировке, довольно много. Если же надо упорядочить только горстку элементов, то гораздо выгоднее простая стратегия вроде "сортировки взбалтыванием" со сложностью $O(n^2)$ (упр. 3.5).

Известно много алгоритмов сортировки. Мы не пытаемся охватить даже все те из них, которые считаются важными; скорее мы

[1]) Если $\leqslant$ — линейный порядок, то пишут $a<b$, когда $a \leqslant b$ и $a \neq b$ (как и следовало ожидать). Кроме того, $b>a$ означает то же, что и $a<b$, а $b \geqslant a$ — то же, что и $a \leqslant b$.

ограничимся методами, оказавшимися полезными в разработке алгоритмов. Сначала будет рассмотрен случай, когда элементы, подлежащие сортировке, представлены целыми числами или (что почти эквивалентно) словами в конечном алфавите. В этой ситуации мы увидим, что сортировку можно выполнить за линейное время. Затем изучим задачу сортировки, когда специальные свойства чисел или слов не используются и приходится строить разветвления в программе только на основе результатов сравнений сортируемых элементов. При этих условиях необходимо $O(n \log n)$ сравнений; это же количество сравнений достаточно для упорядочения последовательности из n элементов.

3.2. ЦИФРОВАЯ СОРТИРОВКА

Начнем с последовательности целых чисел $a_1, a_2, \ldots, a_n$, заключенных между 0 и $m-1$. Если m не слишком велико, ее можно упорядочить следующим образом.

1. Организуем m пустых очередей по одной для каждого целого числа от 0 до $m-1$. Каждую такую очередь назовем *черпаком*.
2. Просмотрим последовательность $a_1, a_2, \ldots, a_n$ слева направо, помещая элемент a_i в очередь с номером a_i.
3. Сцепим эти очереди (содержимое $(i+1)$-й очереди приписывается к концу i-й очереди) и получим в результате упорядоченную последовательность.

Так как любой элемент можно вставить в i-ю очередь за постоянное время, то n элементов можно вставить в очереди за время $O(n)$. Конкатенация (сцепление) m очередей требует времени $O(m)$. Если m есть $O(n)$, то этот алгоритм сортирует n целых чисел за время $O(n)$. Назовем его *сортировкой вычерпыванием*.

Процедуру сортировки вычерпыванием можно обобщить так, чтобы она упорядочивала последовательность кортежей (т. е. списков) целых чисел в лексикографическом порядке. Пусть $\leqslant$ — линейный порядок на множестве S. *Лексикографическим порядком* называется такое продолжение отношения $\leqslant$ на кортежи элементов из S, при котором $(s_1, s_2, \ldots, s_p) \leqslant (t_1, t_2, \ldots, t_q)$ означает, что выполнено одно из условий:

1) существует такое целое число j, что $s_j < t_j$ и для всех $i<j$ справедливо $s_i = t_i$,
2) $p \leqslant q$ и $s_i = t_i$ при $1 \leqslant i \leqslant p$.

Например, в словаре слова расположены в лексикографическом порядке, если рассматривать их как кортежи из букв (на буквах задан естественный алфавитный порядок).

Сначала обобщим сортировку вычерпыванием на последовательности, состоящие из k-членных кортежей, компонентами которых

являются целые числа от 0 до m—1. Сортировка осуществляется с помощью k прохождений по данной последовательности, на каждом из которых применяется сортировка вычерпыванием. Во время первого прохождения кортежи упорядочиваются по их k-м компонентам. Во время второго прохождения полученная последовательность упорядочивается по (k—1)-м компонентам. При третьем последовательность, полученная после второго прохождения, упорядочивается по (k—2)-м компонентам и т. д. При k-м (последнем) прохождении последовательность, полученная после (k—1)-го прохождения, упорядочивается по первым компонентам [1]).

Теперь элементы последовательности расположены в лексикографическом порядке.

Дадим точное описание этого алгоритма.

Алгоритм 3.1. Лексикографическая сортировка

Вход. Последовательность $A_1, A_2, \ldots, A_n$, где A_i есть k-членный кортеж $(a_{i1}, a_{i2}, \ldots, a_{ik})$, в котором a_{ij} — целое число между 0 и m—1. (Удобной структурой данных для этой последовательности кортежей является массив размера $n \times k$.)

Выход. Последовательность $B_1, B_2, \ldots, B_n$, представляющая собой такую перестановку $A_1, A_2, \ldots, A_n$, что $B_i \leqslant B_{i+1}$ при $1 \leqslant i < n$.

Метод. Чтобы поместить k-членный кортеж A_i в некоторый черпак, в действительности надо сдвинуть лишь указатель на A_i. Поэтому кортеж A_i можно добавить в черпак за фиксированное время, а не только за время, ограниченное числом k. Для хранения "текущей" последовательности элементов применяется очередь, называемая ОЧЕРЕДЬ. Используется также массив Q, состоящий из m черпаков, в котором черпак $Q[i]$ предназначен для хранения тех k-членных кортежей, у которых число i стоит в компоненте, рассматриваемой в данный момент. Алгоритм приведен на рис. 3.1. □

Теорема 3.1. *Алгоритм* 3.1 *лексикографически упорядочивает последовательность, состоящую из* n k*-членных кортежей, каждая компонента которых является целым числом между* 0 *и* m—1, *за время* $O((m+n)k)$.

Доказательство. Доказательство корректности алгоритма 3.1 проводится индукцией по числу выполнений внешнего цикла. Предположение индукции таково: после r выполнений этого цикла кортежи в ОЧЕРЕДИ будут расположены в лексикографическом порядке по их r последним компонентам. Требуемый результат

[1]) Во многих практических ситуациях достаточно применить сортировку вычерпыванием только по нескольким первым компонентам. Если число элементов, попавших в каждый черпак, мало, можно упорядочить слова в каждом черпаке с помощью какого-нибудь прямолинейного алгоритма сортировки, такого, как сортировка взбалтыванием.

```
begin
  поместить A_1, A_2, ..., A_n в ОЧЕРЕДЬ;
  for j ← k step −1 until 1 do
    begin
      for l ← 0 until m−1 do сделать Q[l] пустым;
      while список ОЧЕРЕДЬ не пуст do
        begin
          пусть A_l — первый элемент в списке ОЧЕРЕДЬ;
          переместить A_l из списка ОЧЕРЕДЬ в черпак Q[a_lj]
        end;
      for l ← 0 until m−1 do
          присоединить содержимое Q[l] к концу списка ОЧЕРЕДЬ
    end
end
```

Рис. 3.1. Алгоритм лексикографической сортировки.

легко получить, если заметить, что $(r+1)$-е выполнение внешнего цикла упорядочивает множество кортежей по их $(r+1)$-й с конца компоненте, а в ситуации, когда два кортежа оказались в одном и том же черпаке, первый из них предшествует второму в лексикографическом порядке, определяемом r последними компонентами.

Одно выполнение внешнего цикла алгоритма 3.1 занимает время $O(m+n)$. Цикл повторяется k раз, и это дает временную сложность $O((m+n)k)$. □

Алгоритм 3.1 находит разнообразные приложения. Долгое время он применялся в машинах, сортирующих перфокарты. С его помощью можно также упорядочить множество из $O(n)$ целых чисел, лежащих между 0 и n^k-1, за время $O(kn)$, так как любое такое число можно считать k-членным кортежем цифр от 0 до $n-1$ (т. е. представленным в системе счисления с основанием n).

Нашим последним обобщением сортировки вычерпыванием будет распространение ее на кортежи неодинаковой длины, которые мы будем называть цепочками. Если самая длинная цепочка имеет длину k, то можно каждую цепочку дополнить специальным символом до длины k и затем применить алгоритм 3.1. Однако, если длинных цепочек мало, такой подход по двум причинам приведет к неоправданной неэффективности. Во-первых, при каждом прохождении просматривается каждая цепочка; во-вторых, каждый черпак $Q[i]$ анализируется даже в том случае, когда почти все черпаки пу-

сты. Мы дадим алгоритм, сортирующий n-элементную последовательность цепочек различной длины, компонентами (т. е. буквами) которых служат числа между 0 и m—1; время работы этого алгоритма на такой последовательности равно $O(m+l^*)$, где l_i — длина i-й цепочки, а $l^*=\sum_{i=1}^{n} l_i$. Этот алгоритм полезен в ситуации, когда числа m и l^* имеют одинаковый порядок $O(n)$.

Суть этого алгоритма в том, что сначала он располагает цепочки в порядке убывания длин. Пусть $l_{\max}$ — длина самой длинной цепочки. Тогда, как и в алгоритме 3.1, сортировка вычерпыванием применяется $l_{\max}$ раз. Но первое такое применение (по самой правой компоненте) сортирует лишь цепочки длины $l_{\max}$. Второе применение сортирует (в соответствии с ($l_{\max}$—1)-й компонентой) те цепочки, длина которых не менее $l_{\max}$—1, и т. д.

Например, пусть нужно упорядочить последовательность из трех цепочек: *bab*, *abc* и *a*. (Мы условились считать компонентами кортежей целые числа, но для удобства записи мы часто будем использовать буквы. Это не вызывает трудностей, потому что всегда, когда мы пожелаем, можно заменить *a*, *b* и *c* на 0, 1 и 2.) В этом примере $l_{\max}=3$, так что при первом прохождении последовательности сортируются только первые две цепочки по третьей компоненте. При этом *bab* помещается в *b*-черпак, а *abc* — в *c*-черпак; *a*-черпак остается пустым. При втором прохождении эти же две цепочки сортируются по второй компоненте. Теперь *a*-черпак и *b*-черпак становятся занятыми, а *c*-черпак пустым. При третьем (последнем) прохождении сортируются все три цепочки по первой компоненте. На этот раз *a*- и *b*-черпаки оказываются занятыми, а *c*-черпак пустым.

Заметим, что в общем случае при данном прохождении могут оказаться пустыми много черпаков. Поэтому полезен предварительный шаг, определяющий, какие черпаки будут при данном прохождении непустыми. Список непустых черпаков для каждого прохождения формируется в порядке возрастания номеров черпаков. Это позволяет сцепить непустые черпаки за время, пропорциональное их числу.

Алгоритм 3.2. Лексикографическая сортировка цепочек разной длины

Вход. Последовательность цепочек (кортежей) $A_1, A_2, \ldots, A_n$, компоненты которых представлены целыми числами от 0 до m—1. Пусть l_i — длина цепочки $A_i=(a_{i1}, a_{i2}, \ldots, a_{il_i})$ и $l_{\max}$ — наибольшее из чисел l_i.

Выход. Перестановка $B_1, B_2, \ldots, B_n$ цепочек A_i, такая, что $B_1 \leqslant B_2 \leqslant \ldots \leqslant B_n$.

Метод.

1. Начнем с формирования списков (по одному для каждого l, $1 \leqslant l \leqslant l_{max}$) тех символов, которые появляются в l-й компоненте одной или более цепочек. Для этого сначала построим пары (l, a_{il}) для каждой компоненты a_{il} каждой цепочки A_i, $1 \leqslant i \leqslant n$, $1 \leqslant l \leqslant l_i$. Такая пара указывает, что в l-й компоненте некоторой цепочки стоит число a_{il}. Множество этих пар лексикографически упорядочивается с помощью очевидного обобщения алгоритма 3.1 [1]). Затем, просматривая упорядоченный так список слева направо, формируем упорядоченные списки НЕПУСТОЙ[l] для $1 \leqslant l \leqslant l_{max}$, такие, что НЕПУСТОЙ[$l$] содержит точно те символы, которые появляются в l-й компоненте некоторой цепочки. Иными словами, НЕПУСТОЙ[l] содержит в упорядоченном виде все целые числа j, для которых $a_{il}=j$ при некотором i.

2. Определяем длину каждой цепочки. Затем формируем списки ДЛИНА[l] для $1 \leqslant l \leqslant l_{max}$, такие, что ДЛИНА[$l$] содержит все цепочки длины l. (Хотя речь все время идет о перемещении цепочек, фактически мы передвигаем только указатели на них. Поэтому каждую цепочку можно добавить в список ДЛИНА[l] за фиксированное число шагов.)

3. Теперь сортируются цепочки по компонентам, как в алгоритме 3.1, начиная с компонент с номером l_{max}. Однако после i-го прохождения цепочек ОЧЕРЕДЬ содержит только те из них, длина которых не менее $l_{max}-i+1$, и они уже будут расположены в соответствии с компонентами, имеющими номера от $l_{max}-i+1$ до l_{max}. Списки типа НЕПУСТОЙ, сформированные на шаге 1, помогают определить при каждом применении сортировки вычерпыванием занятые черпаки. Эта информация используется для того, чтобы ускорить сцепление черпаков. Соответствующая часть алгоритма на Упрощенном Алголе приведена на рис. 3.2. □

Пример 3.1. Упорядочим последовательность цепочек *a*, *bab* и *abc* с помощью алгоритма 3.2. Одно из возможных представлений этих цепочек — структура данных, изображенная на рис. 3.3. ЦЕПОЧКА — это такой массив, что ЦЕПОЧКА[i] — указатель на представление i-й цепочки, у которой длина и компоненты хранятся в массиве ДАННЫЕ. Клетка в массиве ДАННЫЕ, куда указывает указатель из элемента ЦЕПОЧКА[i], дает число j символов в i-й цепочке. Следующие j клеток массива ДАННЫЕ содержат эти символы.

Списки цепочек, используемые алгоритмом 3.2, в действительности являются списками указателей того же типа, что и в массиве ЦЕПОЧКА. В остальной части этого примера мы будем для удоб-

[1]) В алгоритме 3.1 было сделано допущение, что все компоненты выбираются из одного и того же алфавита. Здесь же вторая компонента принимает значения от 0 до $m-1$, а первая — от 1 до l_{max}.

```
    begin
1.    сделать список ОЧЕРЕДЬ пустым;
2.    for j ← 0 until m — 1 do сделать Q[j] пустым;
3.    for l ← l_max step —1 until 1 do
        begin
4.        присоединить содержимое списка ДЛИНА[l] к началу
            списка ОЧЕРЕДЬ[1]);
5.        while список ОЧЕРЕДЬ не пуст do
            begin
6.            пусть A_i — первая цепочка в списке ОЧЕРЕДЬ;
7.            переместить A_i из списка ОЧЕРЕДЬ в черпак
                Q[a_il]
            end;
8.        for j ∈ НЕПУСТОЙ[l] do
            begin
9.            присоединить содержимое списка Q[j] к концу
                списка ОЧЕРЕДЬ;
10.           сделать список Q[j] пустым
            end
        end
    end
```

[1]) С формальной точки зрения можно присоединять лишь к концу очереди, но конкатенация к началу не должна вызывать никаких трудностей. Более эффективным приемом здесь был бы такой: выбрать цепочки A_i из списка ДЛИНА[l] сначала в строке 6, а затем из списка ОЧЕРЕДЬ и совсем не устраивать конкатенации списков ДЛИНА[l] и ОЧЕРЕДЬ.

Рис. 3.2. Лексикографическая сортировка цепочек разной длины.

ства писать в списках сами цепочки, а не указатели на них. Однако надо помнить, что в очередях хранятся эти указатели, а не сами цепочки.

В части 1 алгоритма 3.2 формируется пара $(1, a)$ из первой цепочки, пары $(1, b)$, $(2, a)$, $(3, b)$ из второй и $(1, a)$, $(2, b)$, $(3, c)$ из третьей. Упорядоченный список этих пар выглядит так:

$$(1, a)\,(1, a)\,(1, b)\,(2, a)\,(2, b)\,(3, b)\,(3, c)$$

Просматривая его слева направо, заключаем, что

$$\text{НЕПУСТОЙ}[1] = a, b;$$
$$\text{НЕПУСТОЙ}[2] = a, b;$$
$$\text{НЕПУСТОЙ}[3] = b, c$$

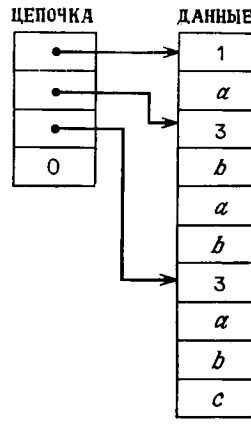


Рис. 3.3. Структура данных для цепочек.

В части 2 алгоритма 3.2 вычисляем $l_1=1$, $l_2=3$ и $l_3=3$. Следовательно, ДЛИНА[1]=*a*, список ДЛИНА[2] пуст и ДЛИНА[3]=*bab*, *abc*. Поэтому часть 3 начинаем с того, что полагаем ОЧЕРЕДЬ= *bab*, *abc*, и затем располагаем эти цепочки по их третьей компоненте. Равенство НЕПУСТОЙ[3]=*b*, *c* гарантирует, что при построении упорядоченного списка в соответствии со строками 8—10 рис. 3.2 не обязательно присоединить *Q*[*a*] к концу списка ОЧЕРЕДЬ. Таким образом, после первого прохождения цикла в строках 3—10 рис. 3.2 ОЧЕРЕДЬ=*bab*, *abc*.

При втором прохождении ОЧЕРЕДЬ не меняется, так как список ДЛИНА[2] пуст, а упорядочение по второй компоненте не изменяет порядок. При третьем прохождении мы, согласно строке 4, получаем ОЧЕРЕДЬ=*a*, *bab*, *abc*. Упорядочение по первой компоненте дает ОЧЕРЕДЬ=*a*, *abc*, *bab*; получили правильный порядок. Заметим, что при третьем прохождении список *Q*[*c*] становится пустым, а поскольку *c* не входит в список НЕПУСТОЙ[1], не нужно присоединять *Q*[*c*] к концу списка ОЧЕРЕДЬ. □

Теорема 3.2. *Алгоритм* 3.2 *упорядочивает свой вход за время* $O(l^*+m)$, *где* $l^*=\sum_{i=1}^{n} l_i$.

Доказательство. Простая индукция по числу прохождений внешнего цикла в программе на рис. 3.2 показывает, что после *i* прохождений список ОЧЕРЕДЬ содержит цепочки длины, не меньшей $l_{max}-i+1$, и они расположены в соответствии с их компонентами с номерами от $l_{max}-i+1$ до l_{max}. Поэтому рассматриваемый алгоритм лексикографически упорядочивает свой вход.

Что касается затрачиваемого времени, то в части 1 расходуется время $O(l^*)$ на образование списка пар и $O(m+l^*)$ на его упорядочение. Аналогично часть 2 занимает не более $O(l^*)$ времени.

Теперь исследуем часть 3 и программу на рис. 3.2. Пусть n_i — число цепочек, имеющих i-е компоненты. Пусть m_i — число различных символов, появляющихся в i-х компонентах цепочек (т. е. m_i — длина списка НЕПУСТОЙ[i]).

Рассмотрим фиксированное значение l в строке 3 рис. 3.2. Цикл в строках 5—7 занимает $O(n_l)$ времени, а в строках 8—10 — $O(m_l)$ времени. Шаг 4 выполняется за постоянное время, так что одно прохождение цикла в строках 3—10 занимает $O(m_l+n_l)$ времени. Таким образом, весь цикл занимает

$$O\left(\sum_{l=1}^{l_{\max}} (m_l + n_l)\right)$$

времени. Так как

$$\sum_{l=1}^{l_{\max}} m_l \leqslant l^* \quad \text{и} \quad \sum_{l=1}^{l_{\max}} n_l \leqslant l^*,$$

то строки 3—10 выполняются за время $O(l^*)$. Теперь, учитывая, что на строку 1 тратится постоянное время, а на строку 2 время $O(m)$, получаем нужный результат. □

Приведем пример, когда упорядочение возникает при разработке алгоритма.

Пример 3.2. Два дерева называются *изоморфными*, если можно отобразить одно из них в другое, изменив порядок сыновей его узлов. Рассмотрим задачу распознавания изоморфизма двух данных деревьев T_1 и T_2. Следующий алгоритм делает это за время, пропорциональное числу узлов. Этот алгоритм приписывает целые числа узлам двух данных деревьев, начиная с узлов уровня 0 и двигаясь вверх до корней, так что деревья изоморфны тогда и только тогда, когда их корням приписано одно и то же число. Алгоритм работает так.

1. Приписать число 0 всем листьям деревьев T_1 и T_2.
2. (Индукция.) Предположим, что целые числа уже приписаны всем узлам, находящимся в деревьях T_1 и T_2 на уровне i—1. Пусть L — список узлов уровня i—1 в дереве T_1, расположенных в порядке неубывания приписанных им чисел, а L_2 — соответствующий список для T_2[1]).
3. Приписать нелистьям уровня i в дереве T_1 кортеж целых чисел следующим образом: просмотреть список L_1 слева направо и для каждого узла v из L_1 взять число, ему приписанное, в

[1]) Надо убедиться, что, проходя дерево в прямом порядке, можно приписать номера уровней за $O(n)$ шагов.

качестве очередной компоненты кортежа, который ставится в соответствии отцу узла v. После выполнения этого шага каждому нелисту w уровня i в дереве T_1 будет поставлен в соответствие кортеж $(i_1, i_2, \ldots, i_k)$, где $i_1, i_2, \ldots, i_k$ — целые числа, приписанные сыновьям узла w и расположенные в порядке неубывания. Обозначим через S_1 последовательность кортежей, построенных для узлов уровня i в дереве T_1.

4. Повторить шаг 3 для T_2; пусть S_2 — последовательность кортежей, построенных для узлов уровня i в дереве T_2.
5. Упорядочить S_1 и S_2 с помощью алгоритма 3.2. Пусть соответственно S_1' и S_2' — упорядоченные последовательности кортежей.
6. Если S_1' и S_2' не совпадают, то остановиться; в этом случае исходные деревья не изоморфны. В противном случае приписать число 1 тем узлам уровня i в дереве T_1, которые представлены первым кортежем в S_1', число 2 — вторым отличающимся кортежем [1]), и т. д. Так как эти целые числа приписаны узлам уровня i в дереве T_1, построить список L_1 из узлов, которым таким способом приписаны числа. Добавить к началу списка L_1 все листья дерева T_1, расположенные на уровне i. Пусть L_2 — соответствующий список узлов дерева T_2. Эти два списка теперь можно использовать для приписывания кортежей узлам уровня $i+1$ с помощью процедуры на шаге 3.
7. Если корням деревьев T_1 и T_2 приписано одно и то же число, то T_1 и T_2 изоморфны. □

На рис. 3.4 иллюстрируется приписывание чисел и кортежей узлам двух изоморфных деревьев.

Теорема 3.3. *Изоморфизм двух деревьев с n узлами можно распознать за время $O(n)$.*

Доказательство. Теорема следует из формализации алгоритма, изложенного в примере 3.2. Доказательство корректности алгоритма мы опускаем. Время работы можно оценить, если заметить, что приписывание целых чисел узлам уровня i, отличным от листьев, занимает время, пропорциональное числу узлов уровня i—1. Суммирование по всем уровням дает время $O(n)$. Работа по приписыванию чисел листьям также пропорциональна n, и, таким образом, весь алгоритм занимает время $O(n)$. □

Помеченным называется дерево, в котором узлам приписаны метки. Допустим, что метками узлов служат целые числа между 1

[1]) Имеется в виду не кортеж, стоящий на 2-м месте в списке S_1' (который может совпасть с кортежем, стоящим на 1-м месте), а тот кортеж, который окажется на 2-м месте после вычеркивания из S_1' всех повторений.— *Прим. ред.*

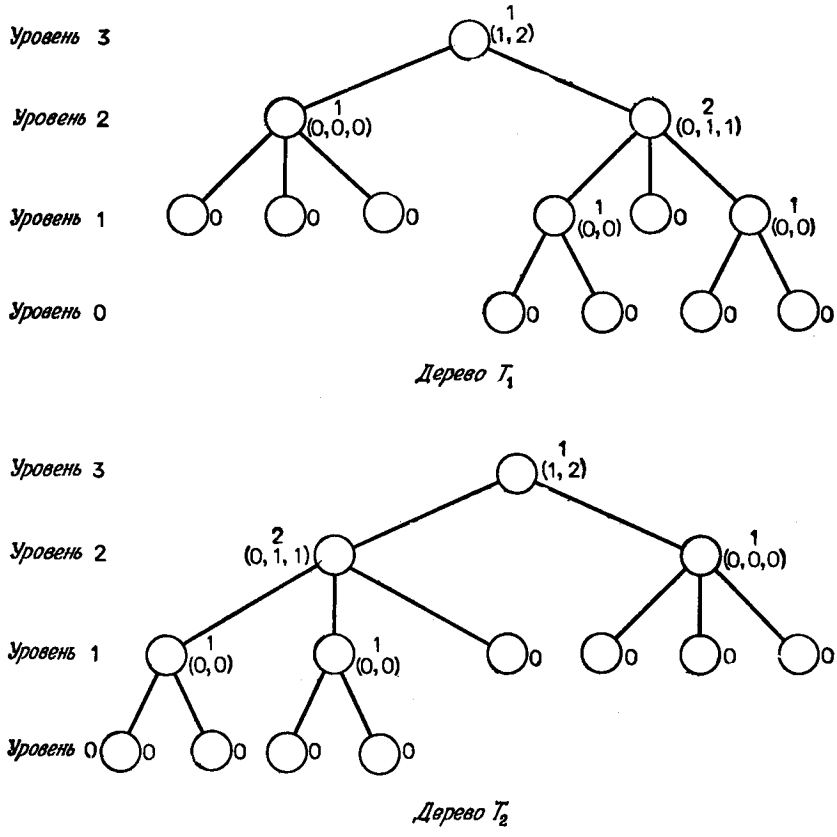


Рис. 3.4. Числа, приписанные алгоритмом распознавания изоморфизма деревьев.

и n. Тогда изоморфизм двух помеченных деревьев можно распознать за линейное время, если включить метку каждого узла в качестве первой компоненты кортежа, приписываемого этому узлу изложенным выше алгоритмом. Таким образом, справедливо

Следствие. *Распознавание изоморфизма двух помеченных деревьев с n узлами, метками которых служат целые числа между* 1 *и* n, *занимает время* $O(n)$.

3.3. СОРТИРОВКА С ПОМОЩЬЮ СРАВНЕНИЙ

Здесь мы изучим задачу упорядочения последовательности из n элементов, взятых из линейно упорядоченного множества S, о структуре которых ничего не известно. Информацию об этой последовательности можно получить только с помощью операции сравнения двух элементов. Сначала мы покажем, что любой алгоритм, упо-

рядочивающий с помощью сравнений, должен делать по крайней мере $O(n \log n)$ сравнений на некоторой последовательности длины n. Пусть надо упорядочить последовательность, состоящую из n различных элементов $a_1, a_2, \ldots, a_n$. Алгоритм, упорядочивающий с помощью сравнений, можно представить в виде дерева решений так, как описано в разд. 1.5. На рис. 1.18 изображено дерево решений, упорядочивающее последовательность a, b, c. Далее мы предполагаем, что если элемент a сравнивается с элементом b в некотором узле v дерева решений, то надо перейти к левому сыну узла v при $a<b$ и к правому — при $a \geqslant b$.

Как правило, алгоритмы сортировки, в которых для разветвления используются сравнения, ограничиваются сравнением за один раз двух входных элементов. В самом деле, алгоритм, который работает на произвольном линейно упорядоченном множестве, не может никак преобразовать входные данные, поскольку при самой общей постановке задачи операции над данными "не имеют смысла". Так или иначе, мы докажем сильный результат о высоте любого дерева решений, упорядочивающего последовательность из n элементов.

Лемма 3.1. *Двоичное дерево высоты h содержит не более 2^h листьев.*

Доказательство. Элементарная индукция по h. Нужно лишь заметить, что двоичное дерево высоты h составлено из корня и самое большее двух поддеревьев, каждое высоты не более $h-1$. □

Теорема 3.4. *Высота любого дерева решений, упорядочивающего последовательность из n различных элементов, не меньше* $\log n!$.

Доказательство. Так как результатом упорядочения последовательности из n элементов может быть любая из $n!$ перестановок входа, то в дереве решений должно быть по крайней мере $n!$ листьев. По лемме 3.1 высота такого дерева должна быть не меньше $\log n!$. □

Следствие. *В любом алгоритме, упорядочивающем с помощью сравнений, на упорядочение последовательности из n элементов тратится не меньше $cn \log n$ сравнений при некотором $c>0$ и достаточно большом n.*

Доказательство. Заметим, что при $n>1$

$$n! \geqslant n(n-1)(n-2)\ldots\left(\left\lceil \frac{n}{2} \right\rceil\right) \geqslant \left(\frac{n}{2}\right)^{n/2},$$

так что $\log n! \geqslant (n/2)\log(n/2) \geqslant (n/4)\log n$ при $n \geqslant 4$. □

По формуле Стирлинга точнее приближает $n!$ функция $(n/e)^n$, так что $n(\log n - \log e) = n \log n - 1{,}44n$ служит хорошим приближением нижней границы числа сравнений, необходимых для упорядочения последовательности из n элементов.

3.4. СОРТДЕРЕВОМ — УПОРЯДОЧЕНИЕ С ПОМОЩЬЮ $O(n \log n)$ СРАВНЕНИЙ

Так как любой сортирующий алгоритм, упорядочивающий с помощью сравнений, затрачивает по необходимости $n \log n$ сравнений для упорядочения хотя бы одной последовательности длины n, естественно спросить, существуют ли сортирующие алгоритмы, затрачивающие не более $O(n \log n)$ сравнений для упорядочения любой последовательности длины n. Один такой алгоритм мы уже видели — это сортировка слиянием из разд. 2.7. Другой алгоритм — Сортдеревом. Помимо того, что это полезный алгоритм упорядочения, в нем используется интересная структура данных, которая находит и другие приложения.

Сортдеревом лучше всего понять в терминах двоичного дерева вроде изображенного на рис. 3.5, у которого каждый лист имеет глубину d или d—1. Узлы дерева помечаются элементами последовательности, которую хотят упорядочить. Затем Сортдеревом меняет размещение этих элементов на дереве до тех пор, пока элемент, соответствующий произвольному узлу, станет не меньше элементов, соответствующих его сыновьям. Такое помеченное дерево мы будем называть *сортирующим*.

Пример 3.3. На рис. 3.5 изображено сортирующее дерево. Заметим, что последовательность элементов, лежащих на пути из любого листа в корень, линейно упорядочена и наибольший элемент в поддереве всегда соответствует его корню. □

На следующем шаге алгоритма Сортдеревом из сортирующего дерева удаляется наибольший элемент — он соответствует корню дерева. Метка некоторого листа переносится в корень, а сам лист удаляется. Затем полученное дерево переделывается в сортирующее, и процесс повторяется. Последовательность элементов, удаленных из сортирующего дерева, упорядочена по невозрастанию.

Удобной структурой данных для сортирующего дерева служит такой массив A, что $A[1]$ — элемент в корне, а $A[2i]$ и $A[2i+1]$ —

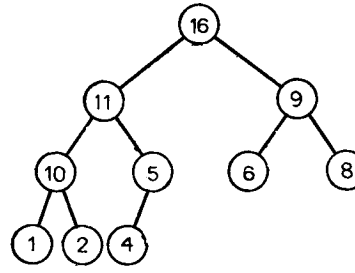


Рис. 3.5. Сортирующее дерево.

элементы в левом и правом сыновьях (если они существуют) того узла, в котором хранится $A[i]$.

Например, сортирующее дерево на рис. 3.5 можно представить массивом

16	11	9	10	5	6	8	1	2	4

Заметим, что узлы наименьшей глубины стоят в этом массиве первыми, а узлы равной глубины стоят в порядке слева направо.

Не каждое сортирующее дерево можно представить таким способом. На языке представления деревьев можно сказать, что для образования такого массива требуется, чтобы листья самого низкого уровня стояли как можно левее (как, например, на рис. 3.5).

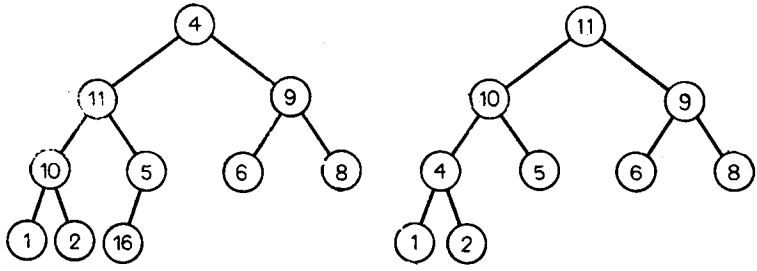
Если сортирующее дерево представляется описанным массивом, то некоторые операции из алгоритма Сортдеревом легко выполнить. Например, согласно алгоритму, нужно удалить элемент из корня, где-то запомнить его, переделать оставшееся дерево в сортирующее и удалить непомеченный лист. Можно удалить наибольший элемент из сортирующего дерева и запомнить его, поменяв местами $A[1]$ и $A[n]$, и затем не считать более ячейку n нашего массива частью сортирующего дерева. Ячейка n рассматривается как лист, удаленный из этого дерева. Для того чтобы переделать дерево, хранящееся в ячейках $1,2, \ldots, n-1$, в сортирующее, надо взять новый элемент $A[1]$ и провести его вдоль подходящего пути в дереве. Затем можно повторить процесс, меняя местами $A[1]$ и $A[n-1]$ и считая, что дерево занимает ячейки $1, 2, \ldots, n-2$ и т. д.

Пример 3.4. Рассмотрим на примере сортирующего дерева рис. 3.5, что происходит, когда мы поменяем местами первый и последний элементы массива, представляющего это дерево. Новый массив

4	11	9	10	5	6	8	1	2	16

соответствует помеченному дереву на рис. 3.6,*а*. Элемент 16 исключается из дальнейшего рассмотрения. Чтобы превратить полученное дерево в сортирующее, надо поменять местами элемент 4 с бóльшим из его сыновей, т. е. с элементом 11.

В своем новом положении элемент 4 обладает сыновьями 10 и 5. Так как они больше 4, то 4 переставляется с 10, бóльшим сыном. После этого сыновьями элемента 4 в новом положении становятся 1 и 2. Поскольку 4 превосходит их обоих, дальнейшие перестановки не нужны.



а *б*

Рис. 3.6. *а* — результат перестановки элементов 4 и 16 в сортирующем дереве рис. 3.5; *б* — результат перестройки сортирующего дерева и удаления элемента 16.

Полученное в результате сортирующее дерево показано на рис. 3.6,*б*. Заметим, что хотя элемент 16 был удален из сортирующего дерева, он все же присутствует в конце массива A. □

Теперь перейдем к формальному описанию алгоритма Сортдеревом.

Пусть $a_1, a_2, \ldots, a_n$— последовательность сортируемых элементов. Предположим, что вначале они помещаются в массив A именно в этом порядке, т. е. $A[i]=a_i$, $1 \leqslant i \leqslant n$. Первый шаг состоит в построении сортирующего дерева, т. е. элементы в A перераспределяются так, чтобы удовлетворялось *свойство сортирующего дерева*: $A[i] \geqslant A[2i]$ при $1 \leqslant i \leqslant n/2$ и $A[i] \geqslant A[2i+1]$ при $1 \leqslant i < n/2$. Это делают, строя, начиная с листьев, все большие и большие сортирующие деревья. Всякое поддерево, состоящее из листа, уже является сортирующим. Чтобы сделать поддерево высоты h сортирующим, надо переставить элемент в корне с наибольшим из элементов, соответствующих его сыновьям, если, конечно, он меньше какого-то из них. Такое действие может испортить сортирующее дерево высоты h—1, и тогда его надо снова перестроить в сортирующее. Приведем точное описание этого алгоритма.

Алгоритм 3.3. Построение сортирующего дерева

Вход. Массив элементов $A[i]$, $1 \leqslant i \leqslant n$.

Выход. Элементы массива A, организованные в виде сортирующего дерева, т. е. $A[i] \leqslant A[\lfloor n/2 \rfloor]$ для $1 < i \leqslant n$.

Метод. В основе алгоритма лежит рекурсивная процедура ПЕРЕСЫПКА. Ее параметры i и j задают область ячеек массива A, обладающую свойством сортирующего дерева; корень строящегося дерева помещается в i.

procedure ПЕРЕСЫПКА (i, j):
1. **if** i — не лист и какой-то его сын содержит элемент, превосходящий элемент в i **then**
   begin
2. пусть k — тот сын узла i, в котором хранится наибольший элемент;
3. переставить $A[i]$ и $A[k]$;
4. ПЕРЕСЫПКА (k, j)
   end

Параметр j используется, чтобы определить, является ли i листом и имеет он одного или двух сыновей. Если $i>j/2$, то i — лист, и процедуре ПЕРЕСЫПКА (i, j) ничего не нужно делать, поскольку $A[i]$ — уже сортирующее дерево.

Алгоритм, превращающий весь массив A в сортирующее дерево, выглядит просто:

procedure ПОСТРСОРТДЕРЕВА:
for $i \leftarrow n$[1]) **step** -1 **until** 1 **do** ПЕРЕСЫПКА (i, n) □

Покажем, что алгоритм 3.3 преобразует A в сортирующее дерево за линейное время.

Лемма 3.2. *Если узлы* $i+1, i+2, \ldots, n$ *являются корнями сортирующих деревьев, то после вызова процедуры* ПЕРЕСЫПКА (i, n) *все узлы* $i, i+1, \ldots, n$ *будут корнями сортирующих деревьев.*

Доказательство. Доказательство проводится возвратной индукцией по i.

Базис, т. е. случай $i=n$, тривиален, так как узел n должен быть листом и условие, проверяемое в строке 1, гарантирует, что ПЕРЕСЫПКА (n, n) ничего не делает.

Для шага индукции заметим, что если i — лист или у него нет сына с бóльшим элементом, то доказывать нечего (как и при обосновании базиса). Но если у узла i есть один сын (т. е. если $2i=n$) и $A[i]<A[2i]$, то строка 3 процедуры ПЕРЕСЫПКА переставляет $A[i]$ и $A[2i]$. В строке 4 вызывается ПЕРЕСЫПКА ($2i$, n); поэтому из предположения индукции вытекает, что дерево с корнем $2i$ будет переделано в сортирующее. Что касается узлов $i+1, i+2, \ldots, 2i-1$, то они и не переставали быть корнями сортирующих деревьев. Так как после этой новой перестановки в массиве A выполняется неравенство $A[i]>A[2i]$, то дерево с корнем i также оказывается сортирующим.

Аналогично, если узел i имеет двух сыновей (т. е. если $2i+1\leqslant n$) и наибольший из элементов в $A[2i]$ и в $A[2i+1]$ больше элемента

[1]) На практике мы бы начали с $\lfloor n/2 \rfloor$.

в $A[i]$, то, рассуждая, как и выше, можно показать, что после вызова процедуры ПЕРЕСЫПКА(i, n) все узлы i, $i+1$, . . ., n будут корнями сортирующих деревьев. □

Теорема 3.5. *Алгоритм* 3.3 *преобразует* A *в сортирующее дерево за линейное время.*

Доказательство. Применяя лемму 3.2, можно с помощью простой возвратной индукции по i показать, что узел i становится корнем какого-нибудь сортирующего дерева для всех i, $1 \leqslant i \leqslant n$.

Пусть $T(h)$ — время выполнения процедуры ПЕРЕСЫПКА на узле высоты h. Тогда $T(h) \leqslant T(h-1)+c$ для некоторой постоянной c. Отсюда вытекает, что $T(h)$ есть $O(h)$.

Алгоритм 3.3 вызывает процедуру ПЕРЕСЫПКА, если не считать рекурсивных вызовов, один раз для каждого узла. Поэтому время, затрачиваемое на ПОСТРСОРТДЕРЕВА, имеет тот же порядок, что и сумма высот всех узлов. Но узлов высоты i не больше, чем $\lceil n/2^{i+1} \rceil$. Следовательно, общее время, затрачиваемое процедурой ПОСТРСОРТДЕРЕВА, имеет порядок $\sum_{i=1}^{n} in/2^i$, т. е. $O(n)$. □

Теперь можно завершить детальное описание алгоритма СОРТДЕРЕВОМ. Коль скоро элементы массива A преобразованы в сортирующее дерево, некоторые элементы удаляются из корня по одному за раз. Это делается перестановкой $A[1]$ и $A[n]$ и последующим преобразованием $A[1]$, $A[2]$, . . ., $A[n-1]$ в сортирующее дерево. Затем переставляются $A[1]$ и $A[n-1]$, а $A[1]$, $A[2]$, ..., $A[n-2]$ преобразуется в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда $A[1]$, $A[2]$, . . ., $A[n]$ — упорядоченная последовательность.

Алгоритм 3.4. Сортдеревом

Вход. Массив элементов $A[i]$, $1 \leqslant i \leqslant n$.

Выход. Элементы массива A, расположенные в порядке неубывания.

Метод. Применяется процедура ПОСТРСОРТДЕРЕВА, т. е. алгоритм 3.3. Наш алгоритм выглядит так:

```
begin
    ПОСТРСОРТДЕРЕВА;
    for i ← n step −1 until 2 do
        begin
            переставить A[1] и A[i];
            ПЕРЕСЫПКА(1, i − 1)
        end
end
```

□

Теорема 3.6. *Алгоритм* 3.4 *упорядочивает последовательность из n элементов за время* $O(n \log n)$.

Доказательство. Корректность алгоритма доказывается индукцией по числу выполнений основного цикла, которое обозначим через m. Предположение индукции состоит в том, что после m итераций список $A[n-m+1], \ldots, A[n]$ содержит m наибольших элементов, расположенных в порядке неубывания, а список $A[1], \ldots, A[n-m]$ образует сортирующее дерево. Детали доказательства оставляем в качестве упражнения. Время выполнения процедуры ПЕРЕСЫПКА$(1, i)$ есть $O(\log i)$. Следовательно, алгоритм 3.4 имеет сложность $O(n \log n)$. □

Следствие. *Алгоритм Сортдеревом имеет временную сложность* $O_C(n \log n)$.

3.5. БЫСТРСОРТ — УПОРЯДОЧЕНИЕ ЗА СРЕДНЕЕ ВРЕМЯ $O(n \log n)$

До сих пор мы рассматривали поведение сортирующих алгоритмов только в худшем случае. Для многих приложений более реалистичной мерой временной сложности является усредненное время работы. В случае сортировки с помощью деревьев решений мы видим, что никакой сортирующий алгоритм не может иметь среднюю временную сложность, существенно меньшую $n \log n$. Однако известны алгоритмы сортировки, которые работают в худшем случае cn^2 времени, где c — некоторая постоянная, но среднее время работы которых относит их к лучшим алгоритмам сортировки. Примером такого алгоритма служит алгоритм Быстрсорт, рассматриваемый в этом разделе.

Чтобы можно было рассуждать о среднем времени работы алгоритма, мы должны договориться о вероятностном распределении входов. Для сортировки естественно допустить, что мы и сделаем, что любая перестановка упорядочиваемой последовательности равновероятна в качестве входа. При таком допущении уже можно оценить снизу среднее число сравнений, необходимых для упорядочения последовательности из n элементов.

Общий метод состоит в том, чтобы поставить в соответствие каждому листу v дерева решений вероятность быть достигнутым при данном входе. Зная распределение вероятностей на входах, можно найти вероятности, поставленные в соответствие листьям. Таким образом, можно определить среднее число сравнений, производимых данным алгоритмом сортировки, если вычислить сумму $\sum_i p_i d_i$, взятую по всем листьям дерева решений данного алгоритма, в которой p_i — вероятность достижения i-го листа, а d_i — его глубина. Это число называется *средней глубиной* дерева решений. Итак, мы пришли к следующему обобщению теоремы 3.4.

Теорема 3.7. *В предположении, что все перестановки n-элементной последовательности появляются на входе с равными вероятностями, любое дерево решений, упорядочивающее последовательность из n элементов, имеет среднюю глубину не менее* $\log n!$.

Доказательство. Обозначим через $D(T)$ сумму глубин листьев двоичного дерева T. Пусть $D(T)$ — ее наименьшее значение, взятое по всем двоичным деревьям T с m листьями. Покажем индукцией по m, что $D(m) \geqslant m \log m$.

Базис, т. е. случай $m=1$, тривиален. Допустим, что предположение индукции верно для всех значений m, меньших k. Рассмотрим дерево решений T с k листьями. Оно состоит из корня с левым поддеревом T_i с i листьями и правым поддеревом T_{k-i} с $k-i$ листьями при некотором i, $1 \leqslant i < k$. Ясно, что

$$D(T) = i + D(T_i) + (k-i) + D(T_{k-i}).$$

Поэтому наименьшее значение $D(T)$ дается равенством

$$D(k) = \underset{1 \leqslant i \leqslant k}{\text{MIN}} [k + D(i) + D(k-i)]. \tag{3.1}$$

Учитывая предположение индукции, получаем отсюда

$$D(k) \geqslant k + \underset{1 \leqslant i \leqslant k}{\text{MIN}} [i \log i + (k-i) \log (k-i)]. \tag{3.2}$$

Легко показать, что этот минимум достигается при $i=k/2$. Следовательно,

$$D(k) \geqslant k + k \log \frac{k}{2} = k \log k.$$

Таким образом, $D(m) \geqslant m \log m$ для всех $m \geqslant 1$.

Теперь мы утверждаем, что дерево решений T, упорядочивающее n случайных элементов, имеет не меньше $n!$ листьев. Более того, в точности $n!$ листьев появляются с вероятностью $1/n!$ каждый, а остальные — с вероятностью 0. Не изменяя средней глубины дерева T можно удалить из него все узлы, которые являются предками только листьев вероятности 0. Тогда останется дерево T' с $n!$ листьями, каждый из которых достигается с вероятностью $1/n!$. Так как $D(T') \geqslant n! \log n!$, то средняя глубина дерева T' (а значит, и T) не меньше $(1/n!)\, n! \log n! = \log n!$. □

Следствие. *Любая сортировка с помощью сравнений выполняет в среднем не менее* $cn \log n$ *сравнений для некоторой постоянной* $c > 0$.

Заслуживает упоминания эффективный алгоритм, называемый Быстрсорт, поскольку среднее время его работы, хотя и ограничено снизу функцией $cn \log n$ для некоторой постоянной c (как и у всякой сортировки с помощью сравнений), но составляет лишь часть

```
procedure БЫСТРСОРТ(S):
1. if S содержит не больше одного элемента then return S
   else
     begin
2.     выбрать произвольный элемент a из S;
3.     пусть S1, S2 и S3 — последовательности элементов из S,
         соответственно меньших, равных и больших a;
4.     return (БЫСТРСОРТ(S1), затем S2, затем БЫСТРСОРТ (S3))
     end
```

Рис. 3.7. Программа Быстрсорт.

времени работы других известных алгоритмов при их реализации на большинстве существующих машин. В худшем случае Быстрсорт имеет квадратичное время работы, но для многих приложений это не существенно.

Алгоритм 3.5. Быстрсорт

Вход. Последовательность S из n элементов $a_1, a_2, \ldots, a_n$.

Выход. Элементы последовательности S, расположенные по порядку.

Метод. Рекурсивная процедура БЫСТРСОРТ определяется на рис. 3.7. Алгоритм состоит в вызове БЫСТРСОРТ(S). □

Теорема 3.8. *Алгоритм* 3.5. *упорядочивает последовательность из n элементов за среднее время* $O(n \log n)$.

Доказательство. Корректность алгоритма 3.5 доказывается прямой индукцией по длине последовательности S. Чтобы проще было анализировать время работы, допустим, что все элементы в S различны. Это допущение максимизирует размеры последовательностей S_1 и S_3, которые строятся в строке 3, и тем самым максимизирует среднее время, затрачиваемое в рекурсивных вызовах в строке 4. Пусть $T(n)$ — среднее время, затрачиваемое алгоритмом БЫСТРСОРТ на упорядочение последовательности из n элементов. Ясно, что $T(0)=T(1)=b$ для некоторой постоянной b.

Допустим, что элемент a, выбираемый в строке 2, является i-м наименьшим элементом среди n элементов последовательности S. Тогда на два рекурсивных вызова БЫСТРСОРТ в строке 4 тратится среднее время $T(i-1)$ и $T(n-i)$ соответственно. Так как i равновероятно принимает любое значение между 1 и n, а итоговое построение последовательности БЫСТРСОРТ(S) очевидно занимает

время cn для некоторой постоянной c, то

$$T(n) \leqslant cn + \frac{1}{n}\sum_{i=1}^{n}[T(i-1)+T(n-i)] \text{ для } n \geqslant 2. \qquad (3.3)$$

Алгебраические преобразования в (3.3) приводят к неравенству

$$T(n) \leqslant cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \qquad (3.4)$$

Покажем, что при $n \geqslant 2$ справедливо неравенство $T(n) \leqslant kn \ln n$, где $k = 2c+2b$ и $b = T(0) = T(1)$. Для базиса ($n=2$) неравенство $T(2) \leqslant 2c+2b$ непосредственно вытекает из (3.4). Для проведения шага индукции запишем (3.4) в виде

$$T(n) \leqslant cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki \ln i. \qquad (3.5)$$

Так как функция $i \ln i$ вогнута, легко показать, что

$$\sum_{i=2}^{n-1} i \ln i \leqslant \int_{2}^{n} x \ln x \, dx \leqslant \frac{n^2 \ln n}{2} - \frac{n^2}{4}. \qquad (3.6)$$

Подставляя (3.6) в (3.5), получаем

$$T(n) \leqslant cn + \frac{4b}{n} + kn \ln n - \frac{kn}{2}. \qquad (3.7)$$

Поскольку $n \geqslant 2$ и $k = 2c+2b$, то $cn+4\,b/n \leqslant kn/2$. Таким образом, неравенство $T(n) \leqslant kn \ln n$ следует из (3.7). □

Рассмотрим две детали, важные для практической реализации алгоритма. Первая — способ "произвольного" выбора элемента a в строке 2 процедуры БЫСТРСОРТ. При реализации этого оператора может возникнуть искушение встать на простой путь, а именно всегда выбирать, скажем, первый элемент последовательности S. Подобный выбор мог бы оказаться причиной значительно худшей работы алгоритма БЫСТРСОРТ, чем это вытекает из (3.3). Последовательность, к которой применяется подпрограмма сортировки, часто бывает уже "как-то" рассортирована, так что первый элемент мал с вероятностью выше средней. Читатель может проверить, что в крайнем случае, когда БЫСТРСОРТ начинает работу на уже упорядоченной последовательности без повторений, а в качестве элемента a всегда выбирается первый элемент из S, в последовательности S всегда будет на один элемент меньше, чем в той, из которой она строится. В этом случае БЫСТРСОРТ требует квадратичного числа шагов.

Лучшей техникой для выбора разбивающего элемента в строке 2 было бы использование генератора случайных чисел для порождения целого числа i, $1 \leqslant i \leqslant |S|$ [1]), и выбора затем i-го элемента из S в качестве a. Более простой подход — произвести выборку элементов из S, а затем взять ее медиану в качестве разбивающего элемента. Например, в качестве разбивающего элемента можно было бы взять медиану выборки, состоящей из первого, среднего и последнего элементов последовательности S.

Вторая деталь — как эффективно разбить S на три последовательности S_1, S_2 и S_3? Можно (и желательно) иметь в массиве A все n исходных элементов. Так как процедура БЫСТРСОРТ вызывает себя рекурсивно, ее аргумент S всегда будет находиться в последовательных компонентах массива, скажем $A[f]$, $A[f+1]$, ..., $A[l]$ для некоторых f и l, $1 \leqslant f \leqslant l \leqslant n$. Выбрав "произвольный" элемент a, можно устроить разбиение последовательности S на этом же месте. Иными словами, можно расположить S_1 в компонентах $A[f]$, $A[f+1]$, ..., $A[k]$, а $S_2 \cup S_3$ — в $A[k+1]$, $A[k+2]$, ..., $A[l]$ при некотором k, $f \leqslant k \leqslant l$. Затем можно, если нужно, расщепить $S_2 \cup S_3$, но обычно эффективнее просто рекурсивно вызвать БЫСТРСОРТ на S_1 и $S_2 \cup S_3$, если оба этих множества не пусты.

По-видимому, самый легкий способ разбить S на том же месте — это использовать два указателя на массив, назовем их i и j. Вначале

```
   begin
1.   i ← f;
2.   j ← l;
3.   while i ⩽ j do
       begin
4.       while A[j] ⩾ a и j ⩾ f do j ← j − 1;
5.       while A[j] < a и i ⩽ l do i ← i + 1;
6.       if i < j then
           begin
7.           переставить A[i] и A[j];
8.           i ← i + 1;
9.           j ← j − 1
           end
       end
   end
```

Рис. 3.8. Разбиение S на S_1 и $S_2 \cup S_3$ на месте их расположения.

[1]) Через $|S|$ обозначена длина последовательности S.

$i=f$, и все время в $A[f], \ldots, A[i-1]$ будут находиться элементы из S_1. Аналогично вначале $j=l$, а в $A[j+1], \ldots, A[l]$ все время будут находиться элементы из $S_2 \cup S_3$. Это разбиение производит подпрограмма на рис. 3.8.

Затем можно вызвать БЫСТРСОРТ для массива $A[f], \ldots, A[i-1]$, т. е. S_1, и для массива $A[j+1], \ldots, A[l]$, т. е. $S_2 \cup S_3$. Но если $i=f$ (и тогда $S_1=\varnothing$), то надо сначала удалить из $S_2 \cup S_3$ хотя бы один элемент, равный a. Удобно удалять тот элемент, по которому производилось разбиение. Следует также заметить, что если это представление в виде массива применяется для последовательностей, то можно подать аргументы для БЫСТРСОРТ, просто поставив указатели на первую и последнюю ячейку используемого куска массива.

Пример 3.5. Разобьем массив A

1	2	3	4	5	6	7	8	9
6	9	3	1	2	7	1	8	3

по элементу $a=3$. **while**-оператор (строка 4) уменьшает j с 9 до 7, поскольку числа $A[9]=3$ и $A[8]=8$ оба не меньше a, но $A[7]=1<a$. Строка 5 не увеличивает i с его начального значения 1, поскольку $A[1]=6 \geqslant a$. Поэтому мы переставляем $A[1]$ и $A[7]$, полагаем $i=2$, $j=6$ и получаем массив на рис. 3.9, *а*. Результаты, получаемые после следующих двух срабатываний цикла в строках 3—9, показаны на рис. 3.9, *б* и *в*. В этот момент $i>j$, и выполнение **while**-оператора, стоящего в строке 3, заканчивается. □

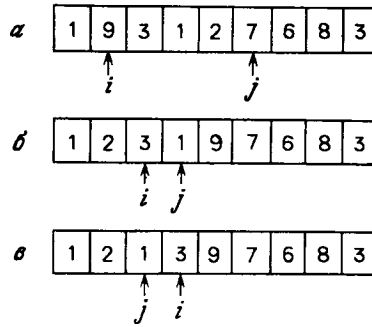


Рис. 3.9. Разбиение массива.

3.6. ПОРЯДКОВЫЕ СТАТИСТИКИ

С упорядочением тесно связана задача нахождения k-го наименьшего элемента в n-элементной последовательности [1]). Одно из очевидных решений состоит в следующем: упорядочить эту последовательность в порядке неубывания ее элементов и взять k-й элемент. Как мы уже видели, это потребует $n \log n$ сравнений. Аккуратно применяя стратегию "разделяй и властвуй", можно найти k-й наименьший элемент за $O(n)$ шагов. Важный частный случай — $k = \lceil n/2 \rceil$; в этом случае середина последовательности отыскивается за линейное время.

Алгоритм 3.6. Нахождение k-го наименьшего элемента

Вход. Последовательность S из n элементов, принадлежащих линейно упорядоченному множеству, и целое число k, $1 \leqslant k \leqslant n$.

Выход. k-й наименьший элемент последовательности S.

Метод. Применяется рекурсивная процедура ВЫБОР, приведенная на рис. 3.10. □

Проанализируем алгоритм 3.6 на интуитивном уровне, чтобы увидеть, почему он работает, как надо. Основная идея — данная последовательность разбивается по некоторому элементу m на такие три подпоследовательности S_1, S_2, S_3, что S_1 содержит все элементы, меньшие m, S_2 — все элементы, равные m, и S_3 — все элементы, большие m. Сосчитав число элементов в S_1 и S_2, можно узнать, в каком из множеств S_1, S_2 и S_3 лежит k-й наименьший элемент. Таким приемом можно свести данную задачу к меньшей.

Чтобы получить линейный алгоритм, надо уметь за линейное время находить разбивающий элемент так, чтобы длина каждой из подпоследовательностей S_1 и S_2 была бы не больше фиксированной доли длины последовательности S. Вся хитрость в способе выбора разбивающего элемента m. Последовательность S разбивается на подпоследовательности по пять элементов в каждой. Каждая подпоследовательность упорядочивается, и из медиан всех этих подпоследовательностей составляется последовательность M. Теперь M содержит только $\lfloor n/5 \rfloor$ элементов, и можно найти ее медиану в пять раз быстрее, чем у последовательности из n элементов.

Далее, не меньше четверти всех элементов последовательности S не превосходят m, и не меньше четверти ее элементов больше или равны m. Это проиллюстрировано на рис. 3.11. Возникает вопрос: причем здесь "магическое число" 5? Ответ заключается в том, что процедура ВЫБОР рекурсивно вызывается два раза, каждый раз

[1]) Строго говоря, *k-м наименьшим* элементом последовательности $a_1, a_2, \ldots, a_n$ называется такой элемент b этой последовательности, что $a_i < b$ не более чем для $k-1$ значений i и $a_i \leqslant b$ не менее чем для k значений i. Например, 4 — второй и третий наименьший элемент последовательности 7, 4, 2, 4.

```
    procedure ВЫБОР(k, S):
 1. if |S| < 50 then
      begin
 2.     упорядочить S;
 3.     return k-й наименьший элемент в S
      end
    else
      begin
 4.     разбить S на ⌊|S|/5⌋ последовательностей по 5 элементов в каждой;
 5.     при этом останется не более четырех неиспользованных элементов;
 6.     упорядочить каждую пятиэлементную последовательность;
 7.     пусть M — последовательность медиан этих пятиэлементных множеств;
 8.     m ← ВЫБОР(⌈|M|/2⌉, M);
 9.     пусть S1, S2 и S3 — последовательности элементов из S,
          соответственно меньших, равных и больших m;
10.     if |S1| ⩾ k then return ВЫБОР(k, S1)
        else
11.       if (|S1| + |S2| ⩾ k) then return m
12.       else return ВЫБОР(k − |S1| − |S2|, S3)
      end
```

Рис. 3.10. Алгоритм выбора k-го наименьшего элемента.

на последовательности, длина которой равна некоторой части длины последовательности S. Чтобы алгоритм работал линейное время, сумма длин этих двух последовательностей должна быть меньше $|S|$. Числа, отличные от 5, также годятся, но для некоторых из них процесс упорядочения подпоследовательностей будет дольше. В качестве упражнения предлагаем выяснить, какие же числа можно использовать вместо 5.

Теорема 3.9. *Алгоритм* 3.6 *находит k-й наименьший элемент в n-элементной последовательности S за время $O(n)$.*

Доказательство. Корректность алгоритма доказывается непосредственно индукцией по $|S|$, и эту часть доказательства мы оставляем в качестве упражнения. Пусть $T(n)$ — время, затра-

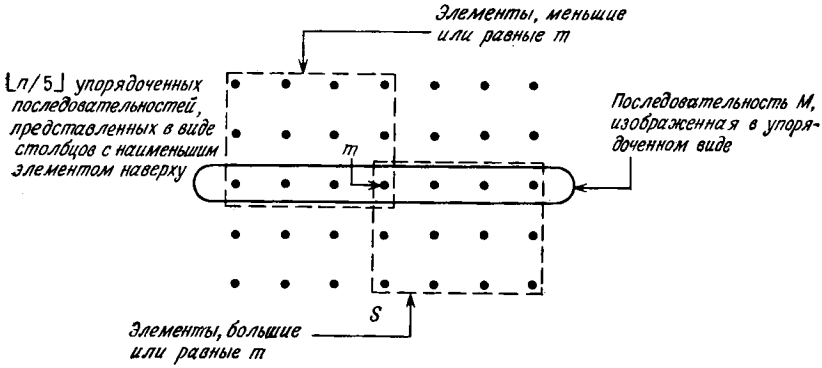


Рис. 3.11. Разбиение S по алгоритму 3.6.

чиваемое на выбор k-го наименьшего элемента из последовательности длины n. Длина последовательности M, составленной из медиан подпоследовательностей, не больше $n/5$, и потому рекурсивный вызов процедуры ВЫБОР($\lceil |M|/2 \rceil$, M) занимает время, не большее $T(n/5)$.

Каждая из последовательностей S_1 и S_3 имеет длину не более $3n/4$. Чтобы увидеть это, заметим, что не менее $\lfloor n/10 \rfloor$ элементов последовательности M больше или равны m и для каждого из них найдутся в S два различных элемента, которые так же соотносятся с m. Следовательно, $|S_1| \leqslant n - 3\lfloor n/10 \rfloor$, т. е. при $n \geqslant 50$ длина последовательности S_1 меньше $3n/4$. Аналогичное рассуждение применимо и к S_3. Поэтому рекурсивный вызов в строке 10 или 12 занимает времени не более $T(3n/4)$. Все остальные операторы тратят не более $O(n)$ времени. Таким образом, для некоторой постоянной c

$$T(n) \leqslant \begin{cases} cn & \text{для } n \leqslant 49, \\ T(n/5) + T(3n/4) + cn & \text{для } n \geqslant 50. \end{cases} \tag{3.8}$$

Из (3.8) по индукции можно доказать, что $T(n) \leqslant 20\,cn$. □

3.7. СРЕДНЕЕ ВРЕМЯ ДЛЯ ПОРЯДКОВЫХ СТАТИСТИК

В этом разделе мы изучим среднее время, затрачиваемое на выбор k-го наименьшего элемента в последовательности из n элементов. Мы увидим, что для нахождения k-го наименьшего элемента требуется провести по меньшей мере $n-1$ сравнений как в худшем случае, так и в среднем. Поэтому выбирающий алгоритм, описанный в предыдущем разделе, с точностью до постоянного множителя оптимален в классе деревьев решений. Здесь мы приведем другой выбирающий алгоритм, который имеет квадратичную сложность в худшем случае,

но среднее время работы которого составляет лишь долю времени работы алгоритма 3.6.

Пусть $S=\{a_1, a_2, \ldots, a_n\}$ — множество из n различных элементов, а T — дерево решений какого-нибудь алгоритма для нахождения k-го наименьшего элемента в S. Каждый путь p в T определяет такое отношение R_p на S, что $a_i R_p a_j$, если два различных элемента a_i и a_j сравниваются в некотором узле, лежащем на p, и в результате этого сравнения либо $a_i<a_j$, либо $a_i \leqslant a_j$ [1]). Пусть R_p^+ — транзитивное замыкание отношения R_p [2]). Образно говоря, если $a_i R_p^+ a_j$, то последовательность сравнений, представленная путем p, выясняет, что $a_i<a_j$, поскольку никакой элемент не сравнивается сам с собой.

Лемма 3.3. *Если на пути p выясняется, что a_m является k-м наименьшим элементом в S, то для любого $i \neq m$, $1 \leqslant i \leqslant n$, либо $a_i R_p^+ a_m$, либо $a_m R_p^+ a_i$.*

Доказательство. Допустим, что некоторый элемент a_u не связан с a_m отношением R_p^+. Покажем, что, поместив a_u либо перед, либо после a_m в линейном порядке, заданном на S, мы получим противоречие с предположением, что путь p правильно установил, что a_m является k-м наименьшим элементом в S. Пусть $S_1=\{a_j | a_j R_p^+ a_u\}$, $S_2=\{a_j | a_u R_p^+ a_j\}$ и в S_3 входят остальные элементы из S. По предположению a_u и a_m принадлежат S_3.

Если a_j— произвольный элемент в S_1 (соответственно в S_2) и $a_l R_p^+ a_j$ (соответственно $a_j R_p^+ a_l$), то в силу транзитивности a_l также принадлежит S_1 (соответственно S_2). Следовательно, можно построить такой линейный порядок R, совместимый с R_p^+, что все элементы множества S_1 предшествуют всем элементам из S_3, которые в свою очередь предшествуют всем элементам из S_2.

По предположению элемент a_u не связан отношением R_p^+ ни с одним элементом из S_3. Допустим, что a_u предшествует a_m при линейном порядке R, т. е. $a_u R a_m$. Тогда можно построить новый линейный порядок R', совпадающий с R во всем, кроме того, что a_u следует непосредственно за a_m. Отношение R' также совместимо с R_p. Для каждого порядка R и R' можно найти различные элементы, удовлетворяющие соответственно R или R'. Но a_m не может быть k-м наименьшим элементом в обоих случаях, так как в R' элементу a_m предшествует на один элемент меньше, чем в R. Поэтому заключаем, что если какой-то элемент из S не связан с a_m

---

[1]) Напомним, что мы предполагаем, что каждое сравнение a с b дает результат $a<b$ или $b \leqslant a$. В случае $a_i<a_j$ сравнивался элемент a_i с a_j, в случае $a_i \leqslant a_j$ сравнивался элемент a_j с a_i.

[2]) *Транзитивным замыканием* отношения R называется такое отношение R^+, что cR^+d тогда и только тогда, когда существует последовательность истинных утверждений вида $e_1 R e_2$, $e_2 R e_3$, ..., $e_{m-1} R e_m$, где $m \geqslant 2$, $c=e_1$ и $d=e_m$.

отношением R_p^+, то дерево T не может правильно выбрать k-й наименьший элемент из S. Случай $a_m R a_u$ исследуется аналогично. □

Теорема 3.10. *Если T — дерево решений, выбирающее k-й наименьший элемент в множестве S, $||S||=n$, то глубина любого листа дерева T не меньше n—1.*

Доказательство. Рассмотрим путь p в T из корня к листу. По лемме 3.3 либо $a_i R_p^+ a_m$, либо $a_m R_p^+ a_i$ для каждого $i \neq m$, где a_m — элемент, выбранный в качестве k-го наименьшего. Для элемента a_i, $i \neq m$, определим *ключевое* сравнение как первое сравнение на p, содержащее a_i и такое, что выполнено одно из условий:

1) a_i сравнивается с a_m,
2) a_i сравнивается с a_j, $a_i R_p a_j$ и $a_j R_p^+ a_m$,
3) a_i сравнивается с a_j, $a_j R_p a_i$ и $a_m R_p^+ a_j$.

Интуитивно ключевое сравнение для a_i — это первое сравнение, из которого можно определить, предшествует элемент a_i элементу a_m или следует за ним.

Понятно, что всякий элемент a_i, кроме a_m, обладает ключевым сравнением; иначе не выполнилось бы ни $a_i R_p^+ a_m$, ни $a_m R_p^+ a_i$. Далее, легко видеть, что никакое сравнение не может быть ключевым для обоих сравниваемых элементов. Так как в ключевые сравнения должны вовлекаться n—1 элементов, то длина пути p должна быть не меньше n—1. □

Следствие. *Нахождение k-го наименьшего элемента в S требует не меньше n—1 сравнений как в среднем, так и в худшем случае.*

На самом деле для всех k, кроме 1 и n, можно доказать более сильный результат, чем теорема 3.10. (См. упр. 3.21—3.23.)

Если приходится искать алгоритм с хорошим средним временем, который вычисляет k-й наименьший элемент в S, то подходит стратегия типа той, что применялась в алгоритме Быстрсорт.

Алгоритм 3.7. Нахождение k-го наименьшего элемента

Вход. Последовательность S, состоящая из n элементов произвольного множества с линейным порядком $\leqslant$, и целое число k, $1 \leqslant k \leqslant n$.

Выход. k-й наименьший элемент в S.

Метод. Применяется рекурсивная процедура ВЫБОР, приведенная на рис. 3.12. □

Теорема 3.11. *Среднее время работы алгоритма 3.7 линейно.*

```
   procedure ВЫБОР(k, S):
1. if |S| = 1 then return единственный элемент множества S
   else
     begin
2.     случайно выбрать элемент a из S;
3.     пусть S1, S2 и S3 — последовательности элементов из S,
         меньших, равных и больших a соответственно;
4.     if |S1| ≥ k then return ВЫБОР(k, S1)
       else
5.       if |S1| + |S2| ≥ k then return a
6.       else return ВЫБОР(k — |S1| — |S2|, S3)
     end
```

Рис. 3.12. Алгоритм выбора.

Доказательство. Пусть $T(n)$ — среднее время работы процедуры ВЫБОР на последовательности из n элементов. Предположим для простоты, что все элементы множества S различны. (При наличии повторяющихся элементов результат не изменится.)

Пусть элемент a, выбранный в строке 2, является i-м наименьшим элементом в S. Число i может принимать значения 1, 2, . . ., n равновероятно. Если $i>k$, то ВЫБОР вызывается для последовательности, состоящей из $i-1$ элементов, а если $i<k$, то для последовательности, состоящей из $n-i$ элементов. Поэтому среднее время работы рекурсии в строке 4 или 6 равно

$$\frac{1}{n}\left[\sum_{i=1}^{k-1} T(n-i)+\sum_{i=k+1}^{n} T(i-1)\right]=\frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i)+\sum_{i=k}^{n-1} T(i)\right].$$

Остальная часть процедуры ВЫБОР занимает время cn для некоторой постоянной c, так что при $n\geqslant 2$

$$T(n)\leqslant cn+\operatorname*{MAX}_{k}\left\{\frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i)+\sum_{i=k}^{n-1} T(i)\right]\right\}. \qquad (3.9)$$

В качестве упражнения на индукцию предлагаем доказать, что если $T(1)\leqslant c$, то $T(n)\leqslant 4cn$ для всех $n\geqslant 2$. □

УПРАЖНЕНИЯ

3.1. Примените алгоритм 3.1 для упорядочения последовательности цепочек *abc*, *acb*, *bca*, *bbc*, *acc*, *bac*, *baa*.

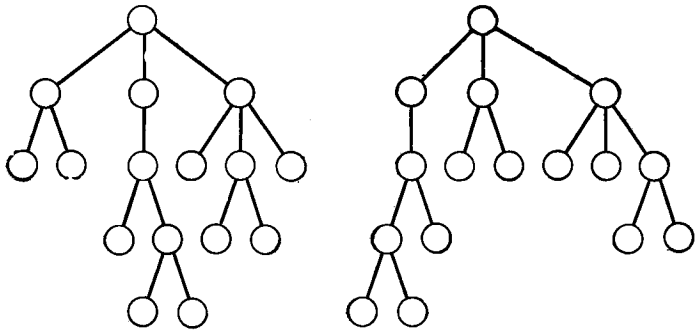


Рис. 3.13. Два дерева.

3.2. Примените алгоритм 3.2 для упорядочения последовательности цепочек *a*, *bc*, *aab*, *baca*, *cbc*, *cc*.

3.3. Проверьте, изоморфны ли два дерева на рис. 3.13 в смысле примера 3.2.

3.4. Упорядочите список 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9 7, применяя (а) Сортдеревом, (б) Быстрсорт, (в) Сортслиянием (алгоритм 2.4). Сколько сравнений производится в каждом случае?

3.5. Рассмотрите следующий алгоритм, упорядочивающий последовательность элементов $a_1, a_2, \ldots, a_n$, хранящихся в массиве A, т. е. $A[i]=a_i$ для $1 \leqslant i \leqslant n$:

procedure СОРТВЗБАЛТЫВАНИЕМ(A):
for $j=n-1$ **step** -1 **until** 1 **do**
  for $i=1$ **step** 1 **until** j **do**
    if $A[i+1] < A[i]$ **then** переставить $A[i]$ и $A[i+1]$

(а) Докажите, что СОРТВЗБАЛТЫВАНИЕМ располагает элементы в A в порядке неубывания.
(б) Определите время работы СОРТВЗБАЛТЫВАНИЕМ в худшем случае и в среднем.

3.6. Закончите доказательство того, что сортирующее дерево можно построить за линейное время (теорема 3.5).

3.7. Закончите доказательство того, что Сортдеревом требует время $O(n \log n)$ (теорема 3.6).

3.8. Покажите, что время работы Быстрсорт в худшем случае есть $O(n^2)$.

3.9. Каково время работы в худшем случае у алгоритма 3.7, находящего k-й наименьший элемент?

*3.10. Докажите, что среднее время упорядочения последовательности из n элементов ограничено снизу величиной $cn \log n$ для некоторой постоянной c, закончив доказательство теоремы 3.7 и решив уравнение (3.2).

3.11. Закончите доказательство того, что алгоритм 3.6 находит k-й наименьший элемент за время $O(n)$ (теорема 3.9), решив уравнение (3.8).

*3.12. Докажите корректность подпрограммы разбиения, приведенной на рис. 3.8, и проанализируйте время ее работы.

3.13. Закончите доказательство того, что алгоритм 3.7 для нахождения k-го наименьшего элемента работает в среднем $O(n)$ времени (теорема 3.11), решив уравнение (3.9).

*3.14. Покажите, что среднее число сравнений, требуемых алгоритмом 3.7, не превосходят $4n$. Можете ли вы улучшить эту оценку, если известно значение k, при котором будет применяться алгоритм?

*3.15. Пусть S — последовательность элементов, содержащая m_i экземпляров i-го элемента для $1 \leqslant i \leqslant k$. Пусть $n = \sum_{i=1}^{k} m_i$. Докажите, что

$$O\left(n + \log \frac{n!}{m_1!\, m_2! \ldots m_k!}\right)$$

сравнений необходимо и достаточно, чтобы упорядочить S.

3.16. Пусть $S_1, S_2, \ldots, S_k$ — множества чисел, лежащих между 1 и n, и сумма мощностей всех S_i равна n. Опишите алгоритм сложности $O(n)$, упорядочивающий все S_i.

3.17. Пусть даны последовательность $a_1, a_2, \ldots, a_n$ и перестановка $\pi(1), \pi(2), \ldots, \pi(n)$. Напишите алгоритм на Упрощенном Алголе, который располагал бы эту последовательность в порядке $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$, работая на том же месте, где она записана вначале. Каково время работы вашего алгоритма в худшем случае и в среднем?

*3.18. В процессе построения сортирующего дерева размера $2^k - 1$ мы строим два сортирующих дерева размера $2^{k-1} - 1$, затем соединяем их, добавляя корень и проталкивая элемент, стоящий в корне, вниз на его надлежащее место. Столь же легко можно было бы построить сортирующее дерево, добавляя за один шаг один элемент в качестве листа и проталкивая этот новый элемент вверх по дереву. Напишите алгоритм, который строил бы сортирующее дерево с помощью добавления листа (одного на каждом шаге), и сравните порядок сложности вашего алгоритма и алгоритма 3.3.

3.19. Рассмотрим прямоугольный массив. Расположим элементы в каждой строке в порядке возрастания. Затем расположим в порядке возрастания элементы каждого столбца. Докажите, что каждая строка останется упорядоченной.

***3.20.** Пусть $a_1, a_2, \ldots, a_n$ — последовательность элементов, а p и q — положительные целые числа. Рассмотрим подпоследовательности, образованные выбором каждого p-го элемента. Упорядочим их. Повторим этот процесс для q. Докажите, что подпоследовательности шага p останутся упорядоченными.

****3.21.** Рассмотрим нахождение с помощью сравнений наибольшего и второго по величине элемента в n-элементном множестве. Докажите, что для этого необходимо и достаточно $n+\lceil \log n \rceil - 2$ сравнений.

****3.22.** Покажите, что среднее число сравнений, требуемых для нахождения k-го наименьшего элемента в n-элементной последовательности, не меньше $(1+0{,}75\,\alpha(1-\alpha))\,n$, где $\alpha=k/n$, а k и n достаточно велики.

****3.23.** Покажите, что в худшем случае требуется $n+\mathrm{MIN}(k, n-k+1)-2$ сравнений для нахождения k-го наименьшего элемента в n-элементном множестве.

****3.24.** Пусть S — множество, состоящее из n целых чисел. Допустим, что вы можете только складывать его элементы и сравнивать суммы. Сколько требуется сравнений для нахождения при этих условиях наибольшего элемента в S?

***3.25.** Алгоритм 3.6 разбивает свой аргумент на подпоследовательности размера 5. Будет ли этот алгоритм работать для других размеров, например 3, 7 или 9? Выберите тот размер, который минимизирует общее число сравнений. На рис. 3.14 указано наименьшее известное число сравнений, достаточное для упорядочения множеств различных размеров. Известно, что для $n\leqslant 12$ эти числа оптимальны.

***3.26.** Алгоритм 3.5 разбивает данную последовательность на подпоследовательности длины 5, а затем находит медиану медиан. Вместо того, чтобы искать медиану медиан, не будет ли эффективнее найти какой-нибудь другой элемент, например $\lfloor k/5 \rfloor$-й?

Размер n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Число сравнений	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

Рис. 3.14. Число сравнений, достаточное для упорядочения последовательности из n элементов (наименьшие известные значения).

*3.27. Рассмотрим следующий метод сортировки. Начнем с последовательности, состоящей из одного элемента. В эту последовательность по одному будем вставлять другие элементы с помощью двоичного поиска. Придумайте структуру данных, позволяющую быстро производить двоичный поиск и вставлять элементы. Можете ли вы реализовать эту сортировку за время $O(n \log n)$?

**3.28. Вместо того, чтобы для разбиения множества размера n выбирать произвольный элемент, как мы делали в Быстрсорт или алгоритме 3.7, можно было бы сделать небольшую выборку, скажем, размера s, найти ее медиану и использовать ее, чтобы разбить все множество. Укажите, как надо выбрать s как функцию от n, чтобы минимизировать среднее число сравнений, необходимых для сортировки.

**3.29. Обобщите идею упр. 3.28 так, чтобы минимизировать число сравнений, необходимых для нахождения порядковой статистики. *Указание*: Извлеките из выборки два элемента, между которыми с большой вероятностью лежит искомый элемент.

3.30. Метод сортировки *стабилен*, если равные элементы остаются в упорядоченной последовательности в том же относительном порядке, в каком они были в исходной последовательности. Какой из перечисленных ниже алгоритмов стабилен?

(а) СОРТВЗБАЛТЫВАНИЕМ (упр. 3.5)
(б) Сортслиянием (алгоритм 2.4)
(в) Сортдеревом (алгоритм 3.4)
(г) Быстрсорт (алгоритм 3.5)

Проблема для исследования

3.31. Некоторые проблемы, касающиеся числа сравнений, необходимых в определенных ситуациях, все еще не решены. Например, пусть требуется найти k наименьших элементов из n-элементного множества. Случай $k=3$ рассмотрен Праттом, Яо [1973]. Для $n \geqslant 13$ неизвестно, оптимальны ли для упорядочения n элементов числа, приведенные в таблице на рис. 3.14. Для малых n оптимальным в смысле числа сравнений является алгоритм сортировки из работы Форда, Джонсона [1959] [1]).

[1]) Отметим, что окончательное решение задачи о порядке сложности сортировки не получено, а именно не известны ответы на некоторые естественные вопросы. Например, рассмотрим сортировку n натуральных чисел, каждое из которых имеет длину двоичного представления k, на адресной машине с длиной регистров l (k и l — функции от n). Если сортируемые числа подаются на вход поразрядно, то время считывания входных данных (т. е. тривиальная нижняя оценка сложности) есть kn; алгоритмы такой сложности изложены в гл. 3. Пусть теперь сортируемые числа считываются частями длины l, т. е. целыми регистрами. Тогда время считывания входных данных есть kn/l. Если отношение k/l ограничено постоянной, то тривиальная нижняя оценка сложности сортировки имеет вид cn, где c — некоторая постоянная. Вопрос о достижимости (по порядку) таких нижних оценок открыт.— *Прим. перев.*

Замечания по литературе

Книга Кнута [1973а] представляет собой энциклопедическое руководство по методам сортировки. Сортдеревом берет начало от Уильямса [1964]; некоторые улучшения сделаны Флойдом [1964]. Быстрсорт ввел Хоор [1962]. Улучшения Быстрсорт по схеме упр. 3.28 предложены Синглтоном [1969], а также Фрейзером, Мак-Келларом [1970]. Алгоритм 3.6 для нахождения порядковой статистики, линейный в худшем случае, изложен в работе Блюма, Флойда, Пратта, Ривеста, Тарьяна [1972]. Хейдиан, Соубел [1969] и Пратт, Яо [1973] исследуют число сравнений, необходимое для нахождения некоторых порядковых статистик.

Результат упр. 3.21 получен Кислицыным [1964]. Упр. 3.22, 3.23 и 3.29 взяты из работы Флойда, Ривеста [1973], где также приведена более сильная нижняя оценка, чем та, что указана в упр. 3.22. Упр. 3.19, 3.20 и некоторые обобщения обсуждаются в работах Гейла, Карпа [1970] и Лю [1972]. Интересное приложение сортировки к нахождению выпуклой оболочки множества точек на плоскости дано Грэхемом [1972]. Стабильная сортировка рассмотрена Хорватом [1974].

4 СТРУКТУРЫ ДАННЫХ ДЛЯ ЗАДАЧ, КАСАЮЩИХСЯ РАБОТЫ С МНОЖЕСТВАМИ

Хороший подход к разработке эффективного алгоритма для данной задачи — изучить ее сущность. Часто задачу можно сформулировать на языке основных математических понятий, таких, как множество, и тогда алгоритм для нее можно изложить в терминах основных операций над основными объектами. Преимущество такой точки зрения в том, что можно проанализировать несколько различных структур данных и выбрать из них ту, которая лучше всего подходит для задачи в целом. Таким образом, разработка хорошей структуры данных идет рука об руку с разработкой хорошего алгоритма.

В настоящей главе изучаются семь основных операций над множествами, типичных для многих задач информационного поиска. Мы приведем разнообразные структуры данных для представления множеств и рассмотрим пригодность каждой из них в ситуациях, когда выполняется та или иная последовательность операций различных типов.

4.1. ОСНОВНЫЕ ОПЕРАЦИИ НАД МНОЖЕСТВАМИ

Будем рассматривать следующие основные операции над множествами.

1. ПРИНАДЛЕЖАТЬ (a, S). Устанавливает принадлежность a множеству S; если $a \in S$, печатается "да"; в противном случае — "нет."
2. ВСТАВИТЬ (a, S). Заменяет S на $S \cup \{a\}$.
3. УДАЛИТЬ (a, S). Заменяет S на $S - \{a\}$.
4. ОБЪЕДИНИТЬ (S_1, S_2, S_3). Строит множество $S_3 = S_1 \cup S_2$. Мы будем предполагать, что в тех случаях, когда применяется эта операция, множества S_1 и S_2 не пересекаются; это делается для того, чтобы избежать необходимости удалять повторяющиеся экземпляры одного и того же элемента в $S_1 \cup S_2$.
5. НАЙТИ (a). Печатает имя того множества, которому в данный

момент принадлежит a. Если a принадлежит более чем одному множеству, то операция не определена.

6. РАСЦЕПИТЬ(a, S). Здесь предполагается, что множество S наделено линейным порядком $\leqslant$. Операция разбивает S на два множества $S_1=\{b|b\leqslant a$ и $b\in S\}$ и $S_2=\{b|b>a$ и $b\in S\}$.
7. MIN(S). Выдает наименьший (относительно $\leqslant$) элемент множества S.

Многие задачи, встречающиеся на практике, можно свести к одной или нескольким подзадачам, таким, что каждую подзадачу можно абстрактно сформулировать как последовательность основных команд, которые следует выполнить на некоторой базе данных (универсальном множестве элементов). В данной главе мы будем изучать последовательности σ команд, каждая из которых является операцией одного из перечисленных выше семи типов.

Например, выполнение последовательностей, состоящих из операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ и УДАЛИТЬ, составляет существенную часть многих задач поиска. Структура данных, которую можно использовать для выполнения последовательности этих операций, будет называться *словарем*. В настоящей главе рассматривается несколько структур данных (такие, как таблицы расстановки, деревья двоичного поиска и 2-3-деревья), которые могут быть словарями.

Здесь возникает много интересных вопросов. Нас будет интересовать главным образом временна́я сложность выполнения операций в σ, т. е. время как функция длины последовательности σ и размера базы данных. Мы исследуем временную сложность в худшем случае и в среднем, а также сложность выполнения σ в префиксном и свободном режимах.

Определение. Выполнение последовательности операций σ в *префиксном режиме* требует, чтобы операции в σ выполнялись в порядке слева направо, причем i-я операция в σ должна выполняться без просмотра какой бы то ни было последующей операции. При *свободном режиме* разрешается просматривать всю последовательность σ в любой момент времени.

Очевидно, что всякий префиксный алгоритм (т. е. работающий в префиксном режиме) может работать как свободный (т. е. в свободном режиме), но обратное верно не всегда. Мы увидим ситуации, когда свободный алгоритм работает быстрее любого из известных префиксных. Однако во многих приложениях приходится рассматривать только префиксные алгоритмы.

Пусть дана последовательность операций, которую надлежит выполнить. Основной вопрос: какую структуру данных использовать для представления универсальной базы данных? Часто задача будет требовать осторожного взвешивания всех за и против каждого

конкретного выбора. В типичной ситуации последовательность будет состоять из нескольких операций, подлежащих многократному выполнению, часто в неизвестном порядке. Может оказаться несколько структур данных, каждая из которых позволяет одну операцию выполнять легко, а другие — с большим трудом. Во многих случаях лучшим решением будет компромисс. Часто мы будем использовать структуру, которая не делает максимально легким выполнение ни одной из операций, но позволяет выполнить всю работу лучше, чем при любом очевидном подходе.

Приведем пример, показывающий, как сформулировать задачу о построении остовного дерева для графа в терминах последовательности операций над множествами.

Определение. Пусть $G=(V, E)$ — неориентированный граф. *Остовным деревом* S (для) графа G называется неориентированное дерево вида (V, T), $T \subseteq E$. *Остовным лесом* (для) графа G называется множество неориентированных деревьев вида $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$, в котором V_i образуют разбиение множества [1]) V, а каждое множество T_i является (возможно, пустым) подмножеством в E.

Функцией стоимости для графа $G=(V, E)$ называется отображение c из E в множество вещественных чисел. Стоимостью подграфа $G'=(V', E')$ графа G считается $c(G')=\sum_{e \in E'} c(e)$.

Пример 4.1. Рассмотрим алгоритм, приведенный на рис. 4.1, для нахождения для данного графа $G=(V, E)$ остовного дерева $S=(V, T)$ наименьшей стоимости. Этот алгоритм построения остовного дерева наименьшей стоимости подробно обсуждается в разд. 5.1, а его применение проиллюстрировано в примере 5.1.

В алгоритме на рис. 4.1 участвуют три множества E, T и VS. Множество E содержит ребра данного графа G, а T используется для сбора ребер окончательного остовного дерева. Работа алгоритма состоит в преобразовании остовного леса для G в единое остовное дерево. Множество VS содержит множества узлов, входящих в деревья этого остовного леса. Вначале VS содержит каждый узел графа G в виде одноэлементного множества.

Можно считать этот алгоритм последовательностью операций с тремя множествами E, T и VS. В строке 1 формируется начальное значение для T, в строках 2 и 3 — для VS (элементы множества VS сами являются множествами). В строке 3 добавляются исходные одноэлементные множества к VS. Строка 4, управляющая основным циклом алгоритма, предполагает наличие счетчика множеств, составляющих VS. В строке 7 выясняется, соединяет ли ребро (v, w) два остовных дерева в остовном лесу. Если да, то в строке 8 эти два

[1]) То есть $V_1 \cup V_2 \cup \ldots \cup V_k = V$ и $V_i \cap V_j = \varnothing$ при $i \neq j$.

```
   begin
1.   T ← ∅;
2.   VS ← ∅;
3.   for каждый узел v ∈ V do добавить одноэлементное множе-
       ство {v} к VS;
4.   while ‖VS‖ > 1 do
       begin
5.       выбрать из E ребро (v, w) наименьшей стоимости;
6.       удалить (v, w) из E;
7.       if v и w принадлежат разным множествам W1 и W2
           из VS then
           begin
8.           заменить W1 и W2 в VS на W1 ∪ W2;
9.           добавить (v, w) к T
           end
       end
   end
```

Рис. 4.1. Алгоритм для нахождения остовного дерева наименьшей стоимости.

дерева сливаются, а в строке 9 ребро (v, w) добавляется к окончательному остовному дереву.

Строка 7 предполагает, что мы можем найти имя того множества в VS, которое содержит конкретный узел. (Вид имен, действительно используемых для множеств в VS, не важен, так что можно употреблять произвольные имена.) По существу, строка 7 предполагает, что мы можем эффективно выполнить основную операцию НАЙТИ. Аналогично строка 8 предполагает, что над непересекающимися множествами узлов мы можем выполнить операцию ОБЪЕДИНИТЬ.

Отыскать структуру данных, на которой быстро выполняется операция ОБЪЕДИНИТЬ или НАЙТИ, довольно легко. Но здесь требуется, чтобы на одной структуре данных можно было легко реализовать обе операции ОБЪЕДИНИТЬ и НАЙТИ. Более того, поскольку выполнение операции ОБЪЕДИНИТЬ в строке 8 зависит от результата выполнения операций НАЙТИ в строке 7, требуемая последовательность операций ОБЪЕДИНИТЬ и НАЙТИ должна выполняться в префиксном режиме. Две такие структуры изучаются в разд. 4.6 и 4.7.

Рассмотрим последовательность операций, выполняемых на множестве ребер E. В строке 5 применяется основная операция MIN, а в строке 6 — операция УДАЛИТЬ. Хорошая структура данных для этих двух операций нам уже встречалась — сортирую-

щее дерево из разд. 3.4. (Хотя там с помощью сортирующего дерева разыскивался наибольший элемент, должно быть ясно, что с его помощью можно также найти и наименьший элемент.)

Наконец, множество T ребер остовного дерева нуждается только в операции ВСТАВИТЬ в строке 9 для добавления к T нового ребра. Здесь достаточно простого списка. □

4.2. МЕТОД РАССТАНОВКИ

Кроме вопроса о том, какие операции появляются в данной последовательности σ, возникает еще один важный вопрос, связанный с выбором подходящей структуры данных для выполнения σ. Это вопрос о размере базы данных (универсального множества), на которой работают операции из σ. Например, в гл. 3 мы видели, что с помощью сортировки вычерпыванием можно упорядочить последовательность из n элементов за линейное время, если элементами рассматриваемого множества являются целые числа, заключенные между подходящими границами. Однако если эти элементы берутся из произвольного линейно упорядоченного множества, то наилучшим временем, которого мы смогли добиться, было время $O(n \log n)$.

В данном разделе мы исследуем задачу о том, как поддержать определенную структуру в изменяющемся множестве S. Новые элементы будут добавляться к S, старые — удаляться из S, и время от времени надо будет отвечать на вопрос: "Принадлежит ли в данный момент элемент x множеству S?" Эта задача естественно моделируется словарем; нам нужна структура данных, которая позволит удобно выполнять последовательности, состоящие из операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ и УДАЛИТЬ. Мы будем предполагать, что элементы, которые могут появиться в S, берутся из очень большого универсального множества, так что представлять S в виде двоичного вектора неразумно с практической точки зрения.

Пример 4.2. Транслятор или ассемблер следит за "таблицей символов" всех идентификаторов, которых он встретил в транслируемой программе. Для большинства языков программирования множество всех возможных идентификаторов очень велико. Например, в Фортране около $1{,}62 \times 10^9$ возможных идентификаторов. Поэтому нереально представить таблицу символов массивом с одним входом для каждого возможного идентификатора независимо от того, появляется он в программе на самом деле или нет.

Операции, производимые транслятором с таблицей символов, относятся к двум типам. Во-первых, надо добавлять в таблицу новые идентификаторы, когда они появляются. Эта работа включает в себя образование той ячейки в таблице, где запоминается конкретный идентификатор и где можно запомнить данные о нем (например, вещественный он или целый). Во-вторых, время от времени трансля-

тор может затребовать информацию о каком-нибудь идентификаторе (например, имеет ли этот идентификатор тип "целое").

Таким образом, структура данных, способная обеспечить выполнение операций ВСТАВИТЬ и ПРИНАДЛЕЖАТЬ, вероятно, достаточна для реализации таблицы символов. И действительно, структура данных, обсуждаемая в этом разделе, часто используется для реализации таблицы символов. □

Мы рассмотрим способ *расстановки*, обеспечивающий выполнение не только операций ВСТАВИТЬ и ПРИНАДЛЕЖАТЬ, необходимых для построения таблицы символов, но и операции УДАЛИТЬ. Известно много вариантов этого способа, но мы обсудим здесь только основную идею.

Схема расстановки показана на рис. 4.2. *Функция расстановки* h отображает элементы универсального множества (например, в случае таблицы символов все возможные идентификаторы) в множество целых чисел от 0 до $m-1$. Мы будем предполагать, что для всех элементов a значение $h(a)$ можно вычислить за постоянное время. Компонентами массива A размера m служат указатели на списки элементов множества S. Список, на который указывает $A[i]$, состоит из всех тех элементов $a \in S$, для которых $h(a)=i$.

Чтобы выполнить операцию ВСТАВИТЬ(a, S), надо вычислить $h(a)$ и затем просмотреть список, на который указывает $A[h(a)]$. Если элемента a в этом списке нет, его надо добавить к концу списка. Чтобы выполнить операцию УДАЛИТЬ(a, S), надо также просмотреть список $A[h(a)]$ и удалить элемент a, если он там есть. Аналогично просматривается список $A[h(a)]$ и в случае операции ПРИНАДЛЕЖАТЬ(a, S).

Вычислительную сложность этой схемы расстановки легко проанализировать. С точки зрения работы в худшем случае она не очень хороша. Например, допустим, что последовательность σ состоит из n различных операций ВСТАВИТЬ. Может случиться, что на всех элементах, которые надлежит вставить, h принимает одинаковые значения, так что все элементы оказываются в одном и том же спис-

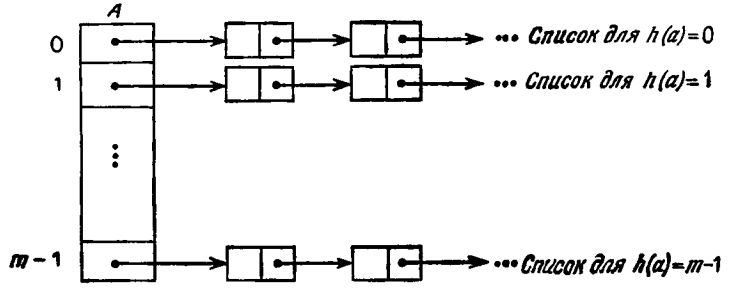


Рис. 4.2. Схема расстановки.

ке. В этой ситуации для выполнения i-й операции из σ требуется время, пропорциональное i. Таким образом, чтобы добавить все n элементов к множеству S, расстановка может потребовать времени порядка n^2.

Однако в смысле среднего времени этот процесс выглядит много лучше. Если значение $h(a)$ с равной вероятностью может быть любым числом между 0 и $m-1$ и вставляется $n \leqslant m$ элементов, то при вставке i-го элемента средняя длина того списка, в который он помещается, равна $(i-1)/m$, т. е. всегда меньше 1. Поэтому среднее время, необходимое для вставки n элементов, есть $O(n)$. Если $O(n)$ операций УДАЛИТЬ и ПРИНАДЛЕЖАТЬ выполняются вместе с операциями ВСТАВИТЬ, то общее среднее время по-прежнему составляет $O(n)$.

Следует иметь в виду, что проведенный анализ предполагает, что размер m таблицы расстановки не меньше максимального размера n множества S. Однако n, как правило, заранее не известно. В таком случае разумнее всего подготовиться к построению последовательности таблиц расстановки $T_0, T_1, T_2, \ldots$.

Сначала выбирается подходящее значение m для размера исходной таблицы T_0. Затем, как только число элементов, вставленных в T_0, превзойдет m, строится новая таблица T_1 размера $2m$ и с помощью новой функции расстановки [1]) все элементы, находящиеся в данный момент в T_0, перемещаются в таблицу T_1. Теперь старая таблица T_0 больше не нужна. Вставка новых элементов в T_1 продолжается до тех пор, пока число элементов не превзойдет $2m$. В этот момент создается новая таблица T_2 размера $4m$ и меняется функция расстановки, чтобы переместить старые элементы из T_1 в T_2. Вообще всякий раз, когда в таблице T_{k-1} оказывается $2^{k-1}m$ элементов, строится таблица T_k размера $2^k m$. Процесс продолжается до тех пор, пока не будут исчерпаны все элементы.

Подсчитаем среднее время, требуемое для вставки в таблицу расстановки 2^k элементов, если применять изложенную только что схему и считать $m=1$. Легко видеть, что этот процесс описывается рекуррентным уравнением

$$T(1) = 1,$$
$$T(2^k) = T(2^{k-1}) + 2^k,$$

решением которого, очевидно, служит $T(2^k) = 2^{k+1} - 1$.

Таким образом, с помощью расстановки последовательность из n операций ВСТАВИТЬ, ПРИНАДЛЕЖАТЬ и УДАЛИТЬ можно выполнить за среднее время $O(n)$.

Здесь важен выбор функции расстановки h. Если элементы, добавляемые в S, представлены целыми числами, равномерно распределенными между 0 и некоторым числом $r \gg n$, то в качестве $h(a)$

[1]) Эта функция принимает значения от 0 до $2m-1$.

можно взять $a \bmod m$, где m — размер таблицы расстановки в данный момент. Другие примеры функций расстановки приведены в работах, упоминаемых в конце главы.

4.3. ДВОИЧНЫЙ ПОИСК

В данном разделе сравниваются три различных решения одной простой задачи поиска. Дано множество S из n элементов, взятых из большого универсального множества. Требуется выполнить последовательность σ, состоящую только из операций ПРИНАДЛЕЖАТЬ.

Наиболее прямым путем решения было бы запомнить элементы множества S в некотором списке. Каждая операция ПРИНАДЛЕЖАТЬ(a, S) выполняется последовательными просмотрами этого списка до тех пор, пока не найдется данный элемент a или не будут просмотрены все элементы списка. Выполнение всех операций из σ заняло бы тогда время порядка $n \times |\sigma|$ как в худшем случае, так и в среднем. Основное преимущество этой схемы в том, что предварительная работа здесь занимает очень мало времени.

Другой путь — разместить элементы множества S в таблице расстановки размера $||S||$. Операция ПРИНАДЛЕЖАТЬ(a, S) выполняется поиском в списке $h(a)$. Если можно найти хорошую функцию h, то выполнение σ займет время $O(|\sigma|)$ в среднем и $O(n|\sigma|)$ в худшем случае. Основная трудность здесь связана с нахождением функции расстановки, равномерно распределяющей элементы из S в таблице расстановки.

Если на S задан линейный порядок $\leqslant$, то третьим решением будет двоичный поиск. Элементы из S хранятся в массиве A. Этот массив упорядочивается так, чтобы было $A[1] < A[2] < \ldots < A[n]$. Теперь, чтобы установить, принадлежит ли элемент a множеству S, надо сравнить его с элементом b, который хранится в ячейке $\lfloor (1+n)/2 \rfloor$. Если $a=b$, то остановиться и ответить "да". В противном случае повторить эту процедуру на первой половине массива, если $a<b$, и на второй, если $a>b$. Повторно разбивая область поиска пополам, можно не более чем за $\lceil \log (n+1) \rceil$ сравнений либо найти a, либо установить, что его нет в S.

Рекурсивная процедура ПОИСК(a, f, l), приведенная на рис. 4.3, ищет элемент a в ячейках $f, f+1, f+2, \ldots, l$ массива A. Для установления принадлежности a множеству S вызывается ПОИСК$(a, 1, n)$.

Чтобы понять, почему эта процедура работает, представим массив A в виде двоичного дерева. Корень находится в ячейке $\lfloor (1+n)/2 \rfloor$, а его левый и правый сыновья — в ячейках $\lfloor (1+n)/4 \rfloor$ и $\lfloor 3(1+n)/4 \rfloor$ соответственно, и т. д. Эта интерпретация двоичного поиска станет яснее в следующем разделе.

```
procedure ПОИСК(a, f, l):
if f > l then return "нет"
else
  if a = A[⌊(f+l)/2⌋] then return "да"
  else
    if a < A[⌊(f+l)/2⌋] then
       return ПОИСК(a, f, ⌊(f+l)/2⌋ − 1)
    else return ПОИСК(a, ⌊(f+l)/2⌋ + 1, l)
```

Рис. 4.3. Процедура двоичного поиска.

Легко показать, что на розыск элемента в A ПОИСК тратит не более $\lceil \log(n+1) \rceil$ сравнений, так как никакой путь в рассматриваемом дереве не длиннее $\lceil \log(n+1) \rceil$. Если все элементы как цели для поиска равновероятны, то можно также показать (упр. 4.4), что ПОИСК дает оптимальное среднее число сравнений (а именно, $|\sigma| \log n$), необходимое для выполнения операций ПРИНАДЛЕЖАТЬ в последовательности σ [1]).

4.4. ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА

Рассмотрим следующую задачу. В множество S вставляются и из него удаляются элементы. Время от времени нам может понадобиться узнать, принадлежит ли данный элемент множеству S или, например, каков в данный момент наименьший элемент в S. Мы считаем, что элементы добавляются в S из большого универсального множества, линейно упорядоченного отношением $\leqslant$. Эту задачу можно сформулировать в общем виде как выполнение последовательности, состоящей из операций ВСТАВИТЬ, УДАЛИТЬ, ПРИНАДЛЕЖАТЬ и MIN.

Мы уже видели, что для выполнения последовательности операций ВСТАВИТЬ, УДАЛИТЬ и ПРИНАДЛЕЖАТЬ хорошей структурой данных может служить таблица расстановки. Но нельзя найти наименьший элемент, не просмотрев всю таблицу. Структурой данных, пригодной для всех четырех операций, является дерево двоичного поиска.

Определение. *Деревом двоичного поиска* для множества S называется помеченное двоичное дерево, каждый узел v которого помечен элементом $l(v) \in S$ так, что

[1]) Разумеется, поскольку в расстановке участвуют не только сравнения, то, возможно, она окажется лучше двоичного поиска; во многих случаях это действительно так.

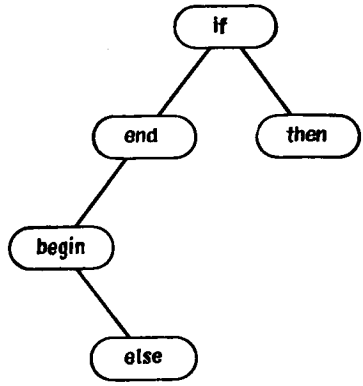


Рис. 4.4. Дерево двоичного поиска.

1) $l(u) < l(v)$ для каждого узла u из левого поддерева узла v,
2) $l(u) > l(v)$ для каждого узла u из правого поддерева узла v,
3) для каждого элемента $a \in S$ существует в точности один узел v, для которого $l(v) = a$.

Заметим, что из условий 1 и 2 вытекает, что метки этого дерева расположены в соответствии с внутренним порядком. Кроме того, условие 3 следует из условий 1 и 2.

Пример 4.3. На рис. 4.4 изображено возможное двоичное дерево для выделенных слов Алгола **begin, else, end, if, then.** Здесь линейным порядком является лексикографический порядок. □

Чтобы выяснить, принадлежит ли элемент a множеству S, представленному деревом двоичного поиска, надо сравнить a с меткой корня. Если метка корня равна a, то очевидно, что $a \in S$. Если a меньше метки корня, то надо перейти к левому поддереву корня (если оно есть). Если a больше метки корня, то надо перейти к правому поддереву корня. Если a присутствует в дереве, его местоположение будет в конце концов обнаружено. В противном случае процесс окончится, когда надо будет найти несуществующее левое или правое поддерево.

Алгоритм 4.1. Просмотр дерева двоичного поиска

Вход. Дерево T двоичного поиска для множества S и элемент a.
Выход. "Да", если $a \in S$, и "нет" в противном случае.
Метод. Если дерево T пусто[1]), выдать "нет." В противном слу-

[1]) Хотя наше определение понятия дерева требует, чтобы дерево содержало хотя бы один узел, а именно корень, во многих алгоритмах мы будем трактовать пустое дерево (дерево без узлов) как двоичное.

```
   procedure ПОИСК(a, v):
1. if a = l(v) then return "да"
   else
2.    if a < l(v) then
3.       if v имеет левого сына w then return ПОИСК(a, w)
4.       else return "нет"
      else
5.       if v имеет правого сына w then return ПОИСК(a, w)
6.       else return "нет"
```

Рис. 4.5. Просмотр дерева двоичного поиска.

чае пусть r — корень дерева T. Тогда алгоритм состоит из единственного вызова ПОИСК(a, r) рекурсивной процедуры ПОИСК, приведенной на рис. 4.5. □

Очевидно, что алгоритм 4.1 достаточен для выполнения операции ПРИНАДЛЕЖАТЬ(a, S). Более того, его легко модифицировать так, чтобы он выполнял операцию ВСТАВИТЬ(a, S). Если дерево пусто, строится корень с меткой a. Если дерево непусто, а элемент, который надлежит вставить, не обнаружен в дереве, то процедуре ПОИСК не удается найти сына в строке 3 или 5. Вместо того чтобы выдать "нет" в строке 4 или 6 соответственно, для этого элемента строится новый узел там, где должен был быть отсутствующий сын.

Деревья двоичного поиска также удобны для выполнения операций MIN и УДАЛИТЬ. Для нахождения наименьшего элемента в дереве двоичного поиска T просматривается путь $v_0, v_1, \ldots, v_p$, где v_0 — корень дерева T, v_i — левый сын узла v_{i-1}, $1 \leqslant i \leqslant p$, и у узла v_p нет левого сына. Метка в узле v_p является наименьшим элементом в T. В некоторых задачах может оказаться удобным иметь указатель на v_p, чтобы обеспечить доступ к наименьшему элементу за постоянное время.

Реализовать операцию УДАЛИТЬ(a, S) немного труднее. Допустим, что элемент a, подлежащий удалению, расположен в узле v. Возможны три случая:

1) v — лист; в этом случае v удаляется из дерева;
2) у v в точности один сын; в этом случае делаем отца узла v отцом его сына, тем самым удаляя v из дерева (если v — корень, то его сына делаем новым корнем);
3) у v два сына; в этом случае находим в левом поддереве узла v наибольший элемент b, рекурсивно удаляем из этого поддерева узел, содержащий b, и метим узел v элементом b. (Заметим, что среди элементов, меньших a, b будет наибольшим элементом всего дерева.)

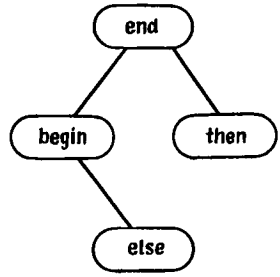


Рис. 4.6. Дерево двоичного поиска после выполнения операции УДАЛИТЬ.

В качестве упражнения предлагаем написать программу на Упрощенном Алголе для операции УДАЛИТЬ. Заметим, что одно выполнение любой из операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ, УДАЛИТЬ и MIN можно осуществить за время $O(n)$.

Пример 4.4. Предположим, что надо удалить слово **if** из дерева двоичного поиска, изображенного на рис. 4.4. Слово **if** расположено в корне, у которого два сына. Наибольшее слово, меньшее **if** (лексикографически), расположенное в левом поддереве корня,— это **end**. Удаляем из дерева узел с меткой **end** и заменяем **if** на **end** в корне. Затем, поскольку **end** имеет одного сына **(begin)**, делаем **begin** сыном корня. Полученное дерево показано на рис. 4.6. □

Подсчитаем временну́ю сложность последовательности из n операций ВСТАВИТЬ, когда рассматриваемое множество представлено деревом двоичного поиска. Время, требуемое, чтобы в дерево двоичного поиска вставить элемент a, ограничено по порядку числом сравнений, производимых между a и элементами, уже находящимися в дереве. Поэтому время можно измерять числом производимых сравнений.

В худшем случае добавление к дереву n элементов может потребовать квадратичного времени. Например, допустим, что последовательность элементов, которые надлежит добавить, оказалась упорядоченной (в порядке возрастания). Тогда дерево поиска будет просто цепью правых сыновей. Но если вставляется n случайных элементов, то, как показывает следующая теорема, вставка потребует время $O(n \log n)$.

Теорема 4.4. *Среднее число сравнений, необходимых для вставки n случайных элементов в дерево двоичного поиска, пустое вначале, равно $O(n \log n)$ для $n \geqslant 1$.*

Доказательство. Пусть $T(n)$ — число сравнений, производимых между элементами последовательности $a_1, a_2, \ldots, a_n$ при построении дерева двоичного поиска, $T(0)=0$. Пусть $b_1, b_2, \ldots, b_n$ — та же последовательность в порядке возрастания.

Если $a_1, a_2, \ldots, a_n$ — случайная последовательность элементов, то a_1 с равной вероятностью совпадает с b_j для любого j, $1 \leqslant j \leqslant n$. Элемент a_1 становится корнем дерева двоичного поиска, и в окончательном дереве $j-1$ элементов $b_1, b_2, \ldots, b_{j-1}$ будут находиться в левом поддереве корня и $n-j$ элементов $b_{j+1}, b_{j+2}, \ldots, b_n$ — в правом поддереве.

Подсчитаем среднее число сравнений, необходимых для вставки элементов $b_1, b_2, \ldots, b_{j-1}$ в дерево. Каждый из этих элементов когда-нибудь сравнивается с корнем, и это дает $j-1$ сравнений с корнем. Затем по индукции получаем, что еще потребуется $T(j-1)$ сравнений, чтобы вставить $b_1, b_2, \ldots, b_{j-1}$ в левое поддерево. Итак, необходимо $j-1+T(j-1)$ сравнений, чтобы вставить $b_1, b_2, \ldots, b_{j-1}$ в дерево двоичного поиска. Аналогично $n-j+T(n-j)$ сравнений потребуется, чтобы вставить в дерево элементы $b_{j+1}, b_{j+2}, \ldots, b_n$.

Поскольку j с равной вероятностью принимает любое значение от 1 до n, то

$$T(n)=\frac{1}{n}\sum_{j=1}^{n}(n-1+T(j-1)+T(n-j)), \tag{4.1}$$

или, с учетом простых алгебраических преобразований,

$$T(n)=n-1+\frac{2}{n}\sum_{j=0}^{n-1}T(j). \tag{4.2}$$

Способом, описанным в разд. 3.5, можно показать, что

$$T(n) \leqslant kn \log n,$$

где $k=\ln 4=1{,}39$. Таким образом, в среднем на вставку n элементов в дерево двоичного поиска тратится $O(n \log n)$ сравнений. □

Итак, методами этого раздела можно выполнить случайную последовательность из n операций ВСТАВИТЬ, УДАЛИТЬ, ПРИНАДЛЕЖАТЬ и MIN за среднее время $O(n \log n)$. Выполнение в худшем случае занимает квадратичное время. Однако даже это худшее время можно улучшить до $O(n \log n)$ с помощью одной из схем сбалансированных деревьев (2-3-деревьев, АВЛ-деревьев или деревьев с ограниченной балансировкой), которые обсуждаются в разд. 4.9 и упр. 4.30—4.37.

4.5. ОПТИМАЛЬНЫЕ ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА

В разд. 4.3 по данному множеству $S=\{a_1, a_2, \ldots, a_n\}$, являющемуся подмножеством некоторого большого универсального множества U, строилась структура данных, позволяющая эффективно выполнить последовательность σ, составленную только из операций ПРИНАДЛЕЖАТЬ. Рассмотрим снова эту задачу, но теперь предположим, что кроме множества S задается для каждого $a \in U$ вероятность появления в σ операции ПРИНАДЛЕЖАТЬ (a, S). Теперь нам хотелось бы построить дерево двоичного поиска для S так, чтобы последовательность σ операций ПРИНАДЛЕЖАТЬ можно было выполнить в префиксном режиме с наименьшим средним числом сравнений.

Пусть $a_1, a_2, \ldots, a_n$ — элементы множества S в порядке возрастания и p_i — вероятность появления в σ операции ПРИНАДЛЕЖАТЬ (a_i, S). Пусть q_0 — вероятность появления в σ операции ПРИНАДЛЕЖАТЬ (a, S) для некоторого $a < a_1$, а q_i — вероятность появления в σ операции ПРИНАДЛЕЖАТЬ (a, S) для некоторого a, $a_i < a < a_{i+1}$. Наконец, пусть q_n — вероятность появления в σ операции ПРИНАДЛЕЖАТЬ (a, S) для некоторого $a > a_n$. Для определения стоимости дерева двоичного поиска удобно добавить к нему $n+1$ фиктивных листьев, чтобы отразить элементы из $U-S$. Мы будем обозначать эти листья числами $0, 1, \ldots, n$.

На рис. 4.7 показано дерево двоичного поиска, изображенное на рис. 4.4, с добавленными фиктивными листьями. Например, лист с меткой 3 представляет те элементы a, для которых **end** $< a <$ **if**.

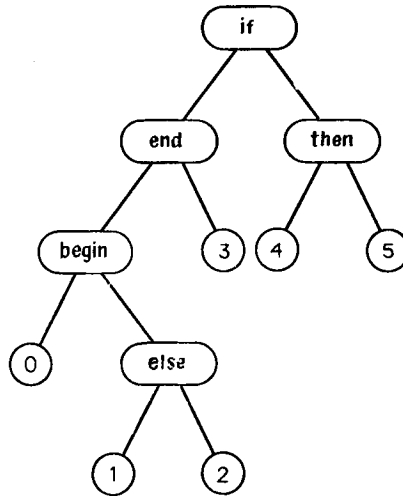


Рис. 4.7. Дерево двоичного поиска с добавленными листьями.

Нам надо определить стоимость дерева двоичного поиска. Если элемент a равен метке $l(v)$ некоторого узла v, то число узлов, посещенных во время выполнения операции ПРИНАДЛЕЖАТЬ (a, S), на единицу больше глубины узла v. Если $a \notin S$ и $a_i < a < a_{i+1}$, то число узлов, посещенных при выполнении операции ПРИНАДЛЕЖАТЬ (a, S), равно глубине фиктивного листа i. Поэтому *стоимость* дерева двоичного поиска можно определить как

$$\sum_{i=1}^{n} p_i\,(\text{ГЛУБИНА}\,(a_i)+1)+\sum_{i=0}^{n} q_i\ \text{ГЛУБИНА}\,(i).$$

Теперь, если у нас есть дерево двоичного поиска T, обладающее наименьшей стоимостью, мы можем выполнить последовательность операций ПРИНАДЛЕЖАТЬ с наименьшим средним числом посещений узлов, просто применив для выполнения каждой операции ПРИНАДЛЕЖАТЬ алгоритм 4.1 на T.

Если даны числа p_i и q_i, то как найти дерево наименьшей стоимости? Подход "разделяй и властвуй" предлагает определить элемент a_i, который надо поставить в корень. Это разбило бы данную задачу на две подзадачи: построение левого и правого поддеревьев. Однако, похоже, нет легкого пути определения корня без решения всей задачи. Поэтому придется решать $2n$ подзадач: по две для каждого возможного корня. Это естественно приводит к решению с помощью динамического программирования.

Для $0 \leqslant i < j \leqslant n$ обозначим через T_{ij} дерево наименьшей стоимости для подмножества $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$. Пусть c_{ij} — стоимость дерева T_{ij}, а r_{ij} — его корень. *Вес* w_{ij} дерева T_{ij} определяется как $q_i+(p_{i+1}+q_{i+1})+\ldots+(p_j+q_j)$.

Дерево T_{ij} состоит из корня a_k, левого поддерева $T_{i,k-1}$ (т. е. дерева наименьшей стоимости для $\{a_{i+1}, a_{i+2}, \ldots, a_{k-1}\}$) и правого поддерева T_{kj} (т. е. дерева наименьшей стоимости для $\{a_{k+1}, a_{k+2}, \ldots, a_j\}$); см. рис. 4.8. Если $i=k-1$, то нет левого поддерева, а при $k=j$ нет правого поддерева. Для удобства мы будем трактовать T_{ii} как пустое дерево. Вес w_{ii} дерева T_{ii} равен q_i, а его стоимость c_{ii} равна 0.

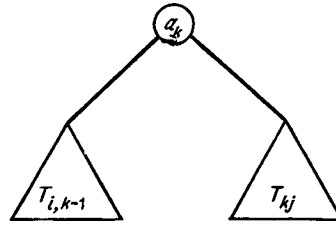


Рис. 4.8. Поддерево T_{ij}.

В T_{ij}, $i<j$, глубина каждого узла в левом и правом поддеревьях на единицу больше глубины того же узла в $T_{i,k-1}$ или T_{kj}. Поэтому стоимость c_{ij} дерева T_{ij} можно выразить так:

$$c_{ij}=w_{i,k-1}+p_k+w_{kj}+c_{i,k-1}+c_{kj}=w_{ij}+c_{i,k-1}+c_{kj}.$$

Следует брать здесь то значение k, которое минимизирует сумму $c_{i,k-1}+c_{kj}$. Поэтому для нахождения оптимального дерева T_{ij} вычисляют для каждого k, $i<k\leqslant j$, стоимость дерева с корнем a_k, левым поддеревом $T_{i,k-1}$ и правым поддеревом T_{kj}, а затем выбирают дерево наименьшей стоимости. Приведем соответствующий алгоритм.

Алгоритм 4.2. Построение оптимального дерева двоичного поиска

Вход. Множество S вида $\{a_1, a_2, \ldots, a_n\}$, где $a_1<a_2<\ldots<a_n$. Известны также вероятности $q_0, q_1, \ldots, q_n$ и $p_1, p_2, \ldots, p_n$, где q_i, $1\leqslant i<n$,— вероятность выполнения операции ПРИНАДЛЕЖАТЬ(a, S) для $a_i<a<a_{i+1}$, q_0— вероятность выполнения операции ПРИНАДЛЕЖАТЬ(a, S) для $a<a_1$, q_n— вероятность выполнения операции ПРИНАДЛЕЖАТЬ(a, S) для $a>a_n$, а p_i, $1\leqslant i\leqslant n$,— вероятность выполнения операции ПРИНАДЛЕЖАТЬ(a_i, S).

Выход. Дерево двоичного поиска для S, обладающее наименьшей стоимостью.

```
    begin
1.    for i ← 0 until n do
        begin
2.        w_ii ← q_i;
3.        c_ii ← 0
        end;
4.    for l ← 1 until n do
5.      for i ← 0 until n − l do
          begin
6.          j ← i + l;
7.          w_ij ← w_{i,j−1} + p_j + q_j;
8.          пусть m — значение k, i < k ⩽ j, для которого сумма
              c_{i,k−1} + c_kj минимальна;
9.          c_ij ← w_ij + c_{i,m−1} + c_mj;
10.         r_ij ← a_m
          end
    end
```

Рис. 4.9. Алгоритм для нахождения корней оптимальных поддеревьев.

```
procedure ПОСТРДЕРЕВА(i, j):
begin
  образовать корень v_ij дерева T_ij;
  пометить v_ij меткой r_ij;
  пусть m — индекс числа r_ij (т. е. r_ij = a_m);
  if i < m — 1 then сделать ПОСТРДЕРЕВА(i, m — 1) левым
    поддеревом узла v_ij;
  if m < j then сделать ПОСТРДЕРЕВА(m, j) правым подде-
    ревом узла v_ij
end
```

Рис. 4.10. Процедура для построения оптимального дерева двоичного поиска.

Метод.

1. Для $0 \leqslant i < j \leqslant n$ вычисляются r_{ij} и c_{ij} в порядке возрастания разности $j-i$ с помощью алгоритма динамического программирования, приведенного на рис. 4.9.

2. После вычисления всех r_{ij} вызывается процедура ПОСТРДЕРЕВА(0, n) для рекурсивного построения оптимального дерева для T_{0n}. Процедура ПОСТРДЕРЕВА приведена на рис. 4.10. □

$l=j-i$ ↓ \ i →	0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$	$w_{11}=3$ $c_{11}=0$	$w_{22}=1$ $c_{22}=0$	$w_{33}=1$ $c_{33}=0$	$w_{44}=1$ $c_{44}=0$
1	$w_{01}=9$ $c_{01}=9$ $r_{01}=a_1$	$w_{12}=6$ $c_{12}=6$ $r_{12}=a_2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=a_3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=a_4$	
2	$w_{02}=12$ $c_{02}=18$ $r_{02}=a_1$	$w_{13}=8$ $c_{13}=11$ $r_{13}=a_2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=a_4$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=a_1$	$w_{14}=10$ $c_{14}=18$ $r_{14}=a_2$			
4	$w_{04}=16$ $c_{04}=33$ $r_{04}=a_2$				

Рис. 4.11. Значения w_{ij}, c_{ij} и r_{ij}.

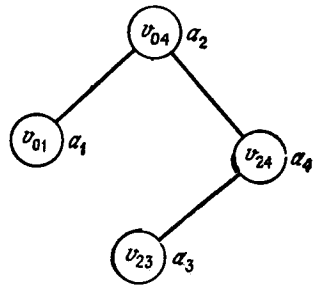


Рис. 4.12. Дерево наименьшей стоимости.

Пример 4.5. Рассмотрим четыре элемента $a_1<a_2<a_3<a_4$ с $q_0=1/8$, $q_1=3/16$, $q_2=q_3=q_4=1/16$ и $p_1=1/4$, $p_2=1/8$, $p_3=p_4=1/16$. На рис. 4.11 даны значения w_{ij}, r_{ij} и c_{ij}, вычисленные алгоритмом, приведенным на рис. 4.9. Для удобства записи значения w_{ij} и c_{ij} в этой таблице были умножены на 16.

Например, чтобы вычислить r_{14}, надо сравнить значения $c_{11}+c_{24}$, $c_{12}+c_{34}$ и $c_{13}+c_{44}$, равные (после умножения на 16) соответственно 8, 9 и 11. Таким образом, в строке 8 рис. 4.9 $k=2$ дает минимум, так что $r_{14}=a_2$.

Вычислив таблицу рис. 4.11, строим дерево T_{04}, вызывая ПОСТРДЕРЕВА(0, 4). На рис. 4.12 изображено полученное в результате дерево двоичного поиска. Его стоимость равна 33/16. □

Теорема 4.2. *Алгоритм* 4.2 *строит оптимальное дерево двоичного поиска за время* $O(n^3)$.

Доказательство. Вычислив таблицу значений r_{ij}, мы строим по ней оптимальное дерево за время $O(n)$, вызывая процедуру ПОСТРДЕРЕВА. Эта процедура вызывается только n раз, и каждый вызов занимает постоянное время.

Наиболее дорого стоит алгоритм динамического программирования (рис. 4.9). Строка 8 находит значение k, минимизирующее $c_{i,k-1}+c_{kj}$, за время $O(j—i)$. Остальные шаги цикла в строках 5—10 занимают постоянное время. Внешний цикл в строке 4 выполняется n раз, внутренний — не более n раз для каждой итерации внешнего цикла. Таким образом, суммарная сложность составляет $O(n^3)$.

Что касается корректности алгоритма, то простой индукцией по $l=j—i$ можно показать, что r_{ij} и c_{ij} правильно вычисляются в строках 9 и 10.

Чтобы показать, что оптимальное дерево правильно строится процедурой ПОСТРДЕРЕВА, заметим, что если v_{ij}— корень поддерева для $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$, то его левый сын будет корнем оптимального дерева для $\{a_{i+1}, a_{i+2}, \ldots, a_{m-1}\}$, где $r_{ij}=a_m$, а правый будет корнем оптимального дерева для $\{a_{m+1}, a_{m+2}, \ldots, a_j\}$.

Теперь должно быть ясно, как доказать по индукции, что процедура ПОСТРДЕРЕВА (i, j) правильно строит оптимальное дерево для $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$. □

В алгоритме 4.2 можно ограничить поиск m в строке 8 рис. 4.9 областью между положениями корней деревьев $T_{i, j-1}$ и $T_{i+1, j}$, при этом гарантируется нахождение минимума. Тогда алгоритм 4.2 сможет находить оптимальное дерево за время $O(n^2)$.

4.6. ПРОСТОЙ АЛГОРИТМ ДЛЯ НАХОЖДЕНИЯ ОБЪЕДИНЕНИЯ НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ

Рассмотрим обработку узлов в алгоритме для нахождения остовного дерева (пример 4.1). Возникающая здесь задача преобразования множеств обладает следующими тремя особенностями.

1. Всякий раз сливаются только непересекающиеся множества.
2. Элементы множеств можно считать целыми числами от 1 до n.
3. Операциями над множествами являются ОБЪЕДИНИТЬ и НАЙТИ

В этом и следующем разделах мы изучим структуры данных для задач такого типа. Пусть даны n элементов, которые мы будем считать целыми числами 1, 2, . . ., n. Предположим, что вначале каждый элемент образует одноэлементное множество. Пусть надо выполнить последовательность операций ОБЪЕДИНИТЬ и НАЙТИ. Напомним, что операция ОБЪЕДИНИТЬ имеет вид ОБЪЕДИНИТЬ(A, B, C), указывающий, что два непересекающихся множества с именами A и B надо заменить их объединением и назвать это объединение C. В приложениях часто неважно, что выбирается в качестве имени множества, так что мы будем предполагать, что множества можно именовать целыми числами от 1 до n. Кроме того, мы будем предполагать, что никакие два множества ни в один момент не названы одинаково.

Эту задачу позволяют решить несколько интересных структур данных. Здесь мы познакомимся со структурой данных, благодаря которой можно выполнить за время $O(n \log n)$ последовательность, содержащую до n—1 операций ОБЪЕДИНИТЬ и до $O(n \log n)$ операций НАЙТИ. В следующем разделе опишем структуру данных, позволяющую обрабатывать последовательность из $O(n)$ операций ОБЪЕДИНИТЬ и НАЙТИ в худшем случае за время, почти линейное по n. Эти структуры данных могут обрабатывать последовательности операций ВСТАВИТЬ, УДАЛИТЬ и ПРИНАДЛЕЖАТЬ с той же вычислительной сложностью.

Заметим, что в алгоритмах поиска, изложенных в разд. 4.2—4.5, предполагалось, что элементы выбираются из универсального

множества, много большего, чем множество выполняемых операций. В этом разделе универсальное множество будет приблизительно того же размера, что и длина последовательности выполняемых операций.

По-видимому, простейшей структурой данных для задачи типа ОБЪЕДИНИТЬ — НАЙТИ служит массив, представляющий набор множеств в данный момент. Пусть R — массив размера n, а $R[i]$ — имя множества содержащего элемент i. Так как вид имен множеств не существен, можно вначале взять $R[i]=i$, $1\leqslant i\leqslant n$, и выразить тем самым факт, что перед началом работы набором множеств является $\{\{1\}, \{2\}, \ldots, \{n\}\}$ и множество $\{i\}$ имеет имя i.

Операция НАЙТИ(i) выполняется путем печати значения $R[i]$ в данный момент. Поэтому сложность выполнения операции НАЙТИ постоянна, а это лучшее, на что можно надеяться.

Чтобы выполнить операцию ОБЪЕДИНИТЬ(A, B, C), надо последовательно просмотреть массив R и заменить каждую его компоненту, равную A или B, на C. Поэтому сложность выполнения такой операции есть $O(n)$. Последовательность из n операций ОБЪЕДИНИТЬ могла бы потребовать $O(n^2)$ времени, что нежелательно.

Этот безыскусный алгоритм можно улучшить несколькими способами. Для одного улучшения можно воспользоваться преимуществом связанных списков. Для другого — понять, что всегда эффективнее влить меньшее множество в большее. Чтобы сделать это, надо отличать "внутренние имена", используемые для идентификации множеств в массиве R, от "внешних имен", упоминаемых в операциях ОБЪЕДИНИТЬ. И те, и другие предполагаются числами от 1 до n, но не обязательно одинаковыми.

Рассмотрим следующую структуру данных для этой задачи. Как и ранее, возьмем такой массив R, что $R[i]$ содержит "внутреннее" имя множества, которому принадлежит элемент i. Но теперь для каждого множества A мы построим связанный список СПИСОК[A], содержащий его элементы. Для реализации этого связанного списка применяются два массива СПИСОК и СЛЕДУЮЩИЙ. СПИСОК[A] представляет собой целое число j, указывающее, что j — первый элемент в множестве с внутренним именем A. СЛЕДУЮЩИЙ[j] дает следующий элемент в A, СЛЕДУЮЩИЙ[СЛЕДУЮЩИЙ[j]] — следующий за ним элемент, и т. д.

Кроме того, возьмем еще массив, называемый РАЗМЕР, такой, что РАЗМЕР[A] — число элементов в множестве A. Множества будут переименовываться по внутренним именам, а два массива ВНУТР_ИМЯ и ВНЕШ_ИМЯ будут устанавливать соответствие между внутренними и внешними именами. Иными словами, ВНЕШ_ИМЯ[A] — это настоящее имя (диктуемое операциями ОБЪЕДИНИТЬ) множества с внутренним именем A. ВНУТР_ИМЯ[j] — это внутреннее имя множества с внешним именем j. Внутренние имена — это имена, используемые в массиве R.

	R	СЛЕДУЮЩИЙ
1	2	3
2	3	4
3	2	5
4	3	8
5	2	7
6	1	0
7	2	0
8	3	0

	СПИСОК	РАЗМЕР	ВНЕШ_ИМЯ
1	6	1	3
2	1	4	1
3	2	3	2

Множества (с внешними именами)

1 = {1, 3, 5, 7}

2 = {2, 4, 8}

3 = {6}

	ВНУТР_ИМЯ
1	2
2	3
3	1

Рис. 4.13. Структуры данных для алгоритма ОБЪЕДИНИТЬ — НАЙТИ.

```
   procedure ОБЪЕДИНИТЬ(I, J, K):
   begin
1.   A ← ВНУТР_ИМЯ[I];
2.   B ← ВНУТР_ИМЯ[J];
3.   wlg положить РАЗМЕР[A] ⩽ РАЗМЕР[B]
4.     otherwise поменять ролями A и B in
       begin
5.       ЭЛЕМЕНТ ← СПИСОК[A];
6.       while ЭЛЕМЕНТ ≠ 0 do
           begin
7.           R[ЭЛЕМЕНТ] ← B;
8.           ПОСЛЕДНИЙ ← ЭЛЕМЕНТ;
9.           ЭЛЕМЕНТ ← СЛЕДУЮЩИЙ[ЭЛЕМЕНТ]
           end;
10.      СЛЕДУЮЩИЙ[ПОСЛЕДНИЙ] ← СПИСОК[B];
11.      СПИСОК[B] ← СПИСОК[A];
12.      РАЗМЕР[B] ← РАЗМЕР[A] + РАЗМЕР[B];
13.      ВНУТР_ИМЯ [K] ← B;
14.      ВНЕШ_ИМЯ[B] ← K
       end
   end
```

Рис. 4.14. Реализация операции ОБЪЕДИНИТЬ.

Пример 4.6. Пусть $n=8$ и у нас есть набор из трех множеств {1, 3, 5, 7}, {2, 4, 8} и {6} с внешними именами 1, 2 и 3 соответственно. Структуры данных для этих трех множеств показаны на рис. 4.13, где 2,3 и 1 — внутренние имена для 1,2 и 3 соответственно. □

Операция НАЙТИ(i) выполняется, как и раньше, обращением к $R[i]$ для установления внутреннего имени множества, содержащего элемент i в данный момент. Затем ВНЕШ_ИМЯ[$R[i]$] дает настоящее имя множества, которому принадлежит i.

Операцию объединения вида ОБЪЕДИНИТЬ(I, J, K) выполняем следующим образом. (Номера строк относятся к рис. 4.14.)

1. Определяем внутренние имена для множеств I и J (строки 1, 2).
2. Сравниваем относительные размеры множеств I и J, справляясь в массиве РАЗМЕР (строки 3, 4).
3. Проходим список элементов меньшего множества и изменяем соответствующие компоненты в массиве R на внутреннее имя большего множества (строки 5—9).
4. Вливаем меньшее множество в большее, добавляя список элементов меньшего множества к началу списка для большего множества (строки 10—12).
5. Присваиваем полученному множеству внешнее имя K (строки 13, 14).

Вливая меньшее множество в большее, мы делаем время выполнения операции ОБЪЕДИНИТЬ пропорциональным мощности меньшего множества. Все детали приведены в процедуре на рис.4.14.

Пример 4.7. После выполнения операции ОБЪЕДИНИТЬ (1, 2, 4) структура данных рис. 4.13 превратится в структуру данных, изображенную на рис. 4.15. □

Теорема 4.3. *С помощью алгоритма рис.* 4.14 *можно выполнить* $n-1$ *(максимально возможное число) операций* ОБЪЕДИНИТЬ *за* $O(n \log n)$ *шагов.*

Доказательство. За каждое выполнение операции ОБЪЕДИНИТЬ будем налагать на перемещаемые элементы одинаковые штрафы, в сумме равные сложности этого выполнения. Так как сложность пропорциональна числу перемещаемых элементов, то величина штрафа, налагаемого на один элемент, будет всегда одна и та же. Основное здесь — это заметить, что всякий раз, когда элемент перемещается из списка, он оказывается в списке, по крайней мере в два раза длиннее прежнего. Поэтому никакой элемент нельзя переместить более чем $\log n$ раз и, значит, суммарный штраф, налагаемый на один элемент, составляет $O(\log n)$.

	R	СЛЕДУЮЩИЙ
1	2	3
2	2	4
3	2	5
4	2	8
5	2	7
6	1	0
7	2	0
8	2	1

	СПИСОК	РАЗМЕР	ВНЕШ_ИМЯ
1	6	1	3
2	2	7	4
3	–	–	–
4	–	–	–

Множества (с внешними именами)

3 = {6}

4 = {1, 2, 3, 4, 5, 7, 8}

	ВНУТР_ИМЯ
1	–
2	–
3	1
4	2

Рис. 4.15. Структура данных после выполнения операции ОБЪЕДИНИТЬ.

Общая сложность получается суммированием штрафов, наложенных на отдельные элементы. Таким образом, общая сложность есть $O(n \log n)$. □

Из теоремы 4.3 вытекает, что если выполняется m операций НАЙТИ и до n—1 операций ОБЪЕДИНИТЬ, то тратится время $O(\text{MAX}(m, n \log n))$. Если m имеет порядок $n \log n$ или больше, то этот алгоритм действительно оптимален с точностью до постоянного множителя. Однако во многих ситуациях m есть $O(n)$, а в таком случае, как мы увидим в следующем разделе, можно добиться лучшего времени, нежели $O(\text{MAX}(m, n \log n))$.

4.7. ДРЕВОВИДНЫЕ СТРУКТУРЫ ДЛЯ ЗАДАЧИ ОБЪЕДИНИТЬ — НАЙТИ

В предыдущем разделе мы познакомились со структурой данных для задачи ОБЪЕДИНИТЬ — НАЙТИ, позволяющей выполнить n—1 операций ОБЪЕДИНИТЬ и $O(n \log n)$ операций НАЙТИ за время $O(n \log n)$. В данном разделе будет описана структура данных, которая состоит из деревьев, образующих лес, и предназначена для представления набора множеств. Эта структура данных позволит выполнить $O(n)$ операций ОБЪЕДИНИТЬ и НАЙТИ за почти линейное время.

Допустим, что каждое множество A представлено корневым неориентированным деревом T_A, узлам которого поставлены в соответствие элементы из A. Корню этого дерева приписано имя самого множества. Операцию ОБЪЕДИНИТЬ(A, B, C) можно выполнить, преобразуя корень дерева T_A в сына корня дерева T_B и заменяя имя в корне дерева T_B на C. Операцию НАЙТИ(i) можно выполнить, определяя положение узла, представляющего элемент i в некотором дереве T из леса, и проходя путь из этого узла в корень дерева T, где помещено имя множества, содержащего i.

В такой схеме сложность слияния двух деревьев равна постоянной. Однако сложность выполнения операции НАЙТИ(i) имеет порядок длины пути из узла i в корень. Эта длина может быть n—1. Поэтому сложность выполнения n—1 операций ОБЪЕДИНИТЬ, за которыми идут n операций НАЙТИ, может достигать $O(n^2)$. Например, вычислим сложность выполнения последовательности операций

ОБЪЕДИНИТЬ(1, 2, 2)
ОБЪЕДИНИТЬ(2, 3, 3)
⋮
ОБЪЕДИНИТЬ($n-1$, n, n)
НАЙТИ(1)
НАЙТИ(2)
⋮
НАЙТИ(n)

Выполнение n—1 операций ОБЪЕДИНИТЬ приводит к дереву, изображенному на рис. 4.16. Сложность выполнения n операций НАЙТИ пропорциональна

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

Однако ее можно уменьшить, если проводить балансировку деревьев. Для этого можно в каждом дереве считать число узлов и, сливая два множества, всегда присоединять меньшее дерево к корню большего. Этот способ аналогичен вливанию меньшего множества в большее, которое мы применяли в предыдущем разделе.

Лемма 4.1. *Если при выполнении каждой операции* ОБЪЕДИНИТЬ *корень дерева с меньшим числом узлов (при равенстве узлов берется любое дерево) преобразуется в сына корня дерева с большим числом узлов, то высота дерева в лесу может достичь значения h, только если оно имеет не менее 2^h узлов.*

Доказательство. Доказательство проведем индукцией по h. Для $h=0$ утверждение верно, поскольку каждое дерево имеет

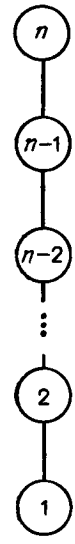


Рис. 4.16. Дерево после выполнения операций ОБЪЕДИНИТЬ.

по крайней мере один узел. Допустим, что утверждение верно для всех значений параметра, меньших $h \geqslant 1$. Пусть T — дерево высоты h с наименьшим числом узлов. Тогда оно должно было получиться в результате слияния деревьев T_1 и T_2, где T_1 — дерево высоты $h-1$ и у него узлов не больше, чем у T_2. По предположению индукции T_1 имеет не менее 2^{h-1} узлов, и, значит, T_2 имеет не менее 2^{h-1} узлов, откуда следует, что T имеет не менее 2^h узлов. □

Оценим время выполнения (в худшем случае) последовательности из n операций ОБЪЕДИНИТЬ и НАЙТИ, если пользоваться структурой данных в виде леса, модифицированной так, что в операции ОБЪЕДИНИТЬ корень меньшего дерева становится сыном корня большего дерева. Никакое дерево не может иметь высоту, большую $\log n$. Следовательно, сложность выполнения $O(n)$ операций ОБЪЕДИНИТЬ и НАЙТИ не превосходит $O(n \log n)$. Эта граница точна в том смысле, что существуют последовательности из n операций, занимающие время, пропорциональное $n \log n$.

Изложим еще одну модификацию этого алгоритма, называемую *сжатием путей*. Поскольку в общей сложности преобладает сложность операций НАЙТИ, попробуем уменьшить ее. Всякий раз, когда выполняется операция НАЙТИ(i), мы проходим путь от узла i до корня r. Пусть $i, v_1, v_2, \ldots, v_n, r$ — узлы на этом пути. Тогда каждый из узлов $i, v_1, v_2, \ldots, v_{n-1}$ делается сыном корня. На рис. 4.17, *б* отражено влияние операции НАЙТИ(i) на дерево, приведенное на рис. 4.17, *а*.

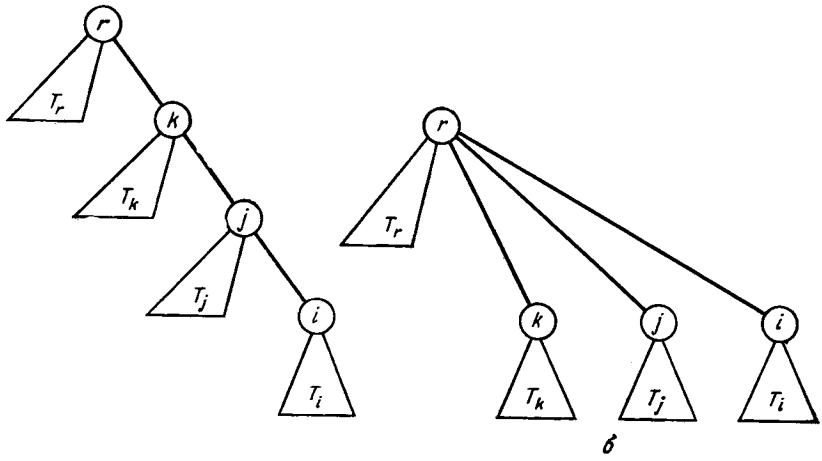


Рис. 4.17. Результат сжатия путей.

Вся процедура слияния деревьев для задачи ОБЪЕДИНИТЬ — НАЙТИ, включая сжатие путей, выражена в следующем алгоритме.

Алгоритм 4.3. Алгоритм быстрого объединения непересекающихся множеств

Вход. Последовательность σ операций ОБЪЕДИНИТЬ и НАЙТИ, выполняемых на наборе множеств, элементы которых представлены целыми числами от 1 до n. Предполагается, что имена множеств также являются целыми числами от 1 до n и вначале множество с именем i состоит из единственного элемента — самого числа i.

Выход. Последовательность ответов на операции НАЙТИ из σ. Ответ на каждую операцию НАЙТИ надо получить перед просмотром очередной операции в σ.

Метод. Описание алгоритма состоит из трех частей: организация начальной структуры, выполнение операции НАЙТИ и выполнение операции ОБЪЕДИНИТЬ.

1. *Организация начальной структуры.* Для каждого элемента i, $1 \leqslant i \leqslant n$, образуем узел v_i. Полагаем СЧЕТ $[v_i]=1$, ИМЯ $[v_i]=i$ и ОТЕЦ $[v_i]=0$. Вначале каждый узел v_i сам является деревом. Чтобы можно было определять корень множества i, образуем такой массив КОРЕНЬ, что КОРЕНЬ $[i]$ указывает на v_i. Чтобы определить узел, содержащий элемент i, образуем массив ЭЛЕМЕНТ; вначале ЭЛЕМЕНТ $[i]=v_i$.

2. *Выполнение операции* НАЙТИ (i). Программа приведена на рис. 4.18. Начиная с узла ЭЛЕМЕНТ $[i]$, идем по пути, ведущему к корню дерева, и выписываем встретившиеся узлы. По достижении

```
begin
  сделать СПИСОК пустым;
  v ← ЭЛЕМЕНТ[i];
  while ОТЕЦ[v] ≠ 0 do
    begin
      добавить v в список;
      v ← ОТЕЦ[v]
    end;
  comment v теперь корень;
  print ИМЯ[v];
  for для каждого w, входящего в СПИСОК do ОТЕЦ[w] ← v
end
```

Рис. 4.18. Выполнение операции НАЙТИ (i).

корня выдается имя множества, а каждый узел из пройденного пути делается сыном корня.

3. *Выполнение операции* ОБЪЕДИНИТЬ (i, j, k). С помощью массива КОРЕНЬ находим корни деревьев, представляющих множества i и j. Затем делаем корень меньшего дерева сыном корня большего дерева. См. рис. 4.19. □

Покажем, что сжатие путей значительно ускоряет работу алгоритма. Чтобы подсчитать это улучшение, введем функции F и G.

```
begin
  wlg положить СЧЕТ[КОРЕНЬ[i]] ⩽ СЧЕТ[КОРЕНЬ[j]]
    otherwise переставить i и j in
    begin
      БОЛЬШОЙ ← КОРЕНЬ[j];
      МАЛЫЙ ← КОРЕНЬ[i];
      ОТЕЦ[МАЛЫЙ] ← БОЛЬШОЙ;
      СЧЕТ[БОЛЬШОЙ] ← СЧЕТ[БОЛЬШОЙ] + СЧЕТ[МА-
        ЛЫЙ];
      ИМЯ[БОЛЬШОЙ] ← k;
      КОРЕНЬ[k] ← БОЛЬШОЙ
    end
end
```

Рис. 4.19. Выполнение операции ОБЪЕДИНИТЬ (i, j, k).

n	$F(n)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Рис. 4.20. Несколько значений функции F.

Пусть

$$F(0) = 1,$$
$$F(i) = 2^{F(i-1)} \quad \text{для} \quad i > 0.$$

Функция F растет очень быстро, как видно из таблицы на рис. 4.20. Функция $G(n)$ определяется как наименьшее число k, для которого $F(k) \geqslant n$. Функция G растет очень медленно. Действительно, $G(n) \leqslant 5$ для всех "практических" значений n, а именно для всех $n \leqslant 2^{65536}$.

Теперь покажем, что алгоритм 4.3 выполнит последовательность σ из cn операций ОБЪЕДИНИТЬ и НАЙТИ за время, не большее $c'nG(n)$, где c и c' — постоянные, причем c' зависит от c. Для простоты предположим, что выполнение операции ОБЪЕДИНИТЬ занимает "единицу времени", а операции НАЙТИ(i) — время, пропорциональное числу узлов, лежащих на пути из узла с меткой i в корень дерева, содержащего этот узел [1]).

Определение. Удобно определить *ранг* узла относительно последовательности σ операций ОБЪЕДИНИТЬ и НАЙТИ следующим образом:

1. Удалить из σ операции НАЙТИ.
2. Выполнить получившуюся последовательность σ' операций ОБЪЕДИНИТЬ.
3. Ранг узла v — это его высота в получившемся лесу.

Докажем некоторые важные свойства ранга узла.

[1]) Таким образом, «единица времени» здесь занимает некоторое постоянное число шагов на РАМ. Так как мы пренебрегаем постоянными множителями, то результаты о порядке величины можно также выражать через «единицы времени».

Лемма 4.2. *Существует не более $n/2^r$ узлов ранга r.*

Доказательство. По лемме 4.1 каждый узел ранга r имеет по крайней мере 2^r потомков в лесу, построенном при выполнении σ'. Так как множества потомков любых двух различных узлов одинаковой высоты в лесу не пересекаются, а общее число непересекающихся множеств, содержащих не менее 2^r узлов, не превосходит $n/2^r$, то узлов ранга r не может быть больше $n/2^r$. □

Следствие. *Ранг любого узла не превосходит* $\log n$.

Лемма 4.3. *Если в какой-то момент выполнения последовательности σ узел w оказался подлинным потомком узла v, то его ранг меньше ранга узла v.*

Доказательство. Если в какой-то момент выполнения последовательности σ узел w стал потомком узла v, то w будет потомком узла v и в лесу, построенном при выполнении σ'. Поэтому высота узла w должна быть меньше высоты узла v, так что ранг узла w меньше ранга узла v. □

Разобьем ранги на *группы*. Отнесем ранг r к группе $G(r)$. Например, ранги 0 и 1 находятся в группе 0, ранг 2 — в группе 1, ранги 3 и 4 — в группе 2, ранги от 5 до 16 — в группе 3. Для $n \geqslant 1$ наибольший возможный ранг, т. е. $\lfloor \log n \rfloor$, попадает в группу $G(\lfloor \log n \rfloor) \leqslant G(n) - 1$.

Исследуем сложность выполнения последовательности σ из cn операций ОБЪЕДИНИТЬ и НАЙТИ. Так как каждую операцию ОБЪЕДИНИТЬ можно выполнить за единицу времени, то все операции ОБЪЕДИНИТЬ из σ можно выполнить за время $O(n)$. Для того чтобы оценить сложность выполнения всех операций НАЙТИ, применим один важный "бухгалтерский" трюк. За каждое выполнение операции НАЙТИ наложим штраф как на само выполнение, так и на некоторые перемещаемые узлы. Искомая общая сложность будет равна сумме всех штрафов, налагаемых на выполнение операций НАЙТИ и на перемещаемые узлы.

Штрафы за выполнение операции НАЙТИ(i) налагаются следующим образом. Пусть v — узел на пути из узла, представляющего i, в корень дерева, содержащего i.

1. Если v — корень или ОТЕЦ$[v]$ имеет ранг, который принадлежит не той же группе, что и ранг узла v, то выполнение штрафуется на одну единицу времени.
2. Если ранги узла v и его отца принадлежат одной группе, то на узел v налагается штраф в одну единицу времени.

По лемме 4.3 ранг узлов по мере прохождения вверх по пути монотонно возрастает. А так как различных групп рангов не больше $G(n)$, то по правилу 1 никакое выполнение операции НАЙТИ не

штрафуется более чем на $G(n)$ единиц времени. Если применяется правило 2, то узел v будет перемещен и сделан сыном узла с бóльшим рангом, чем его предыдущий отец. Если узел v принадлежит группе $g>0$, то он может перемещаться и штрафоваться не более $F(g) - F(g-1)$ раз, прежде чем приобретет отца из группы с более высоким рангом. Если ранг узла принадлежит группе 0, то он может перемещаться не более одного раза, прежде чем приобретет отца из более высокой группы. С этого момента перемещение узла v будет штрафоваться по правилу 1 сложностью выполнения операции НАЙТИ.

Чтобы получить верхнюю границу штрафов, налагаемых на узлы, умножим наибольший штраф, который можно наложить на узел с рангом из данной группы, на число узлов в этой группе и просуммируем по всем группам рангов. Пусть $N(g)$ — число узлов, ранг которых принадлежит группе $g>0$. Тогда по лемме 4.2

$$N(g) \leqslant \sum_{r=F(g-1)+1}^{F(g)} n/2^r \leqslant (n/2^{F(g-1)+1})\left[1+\frac{1}{2}+\frac{1}{4}+\ldots\right] \leqslant$$
$$\leqslant n/2^{F(g-1)} \leqslant n/F(g).$$

Наибольший штраф на узел с рангом из группы $g>0$ не превосходит $F(g) - F(g-1)$. Поэтому наибольший штраф для узлов, ранги которых принадлежат группе g, ограничен числом n. Это утверждение, очевидно, верно и для $g=0$. Поскольку групп рангов не больше $G(n)$, то суммарный штраф, налагаемый на все узлы, не превосходит $nG(n)$. Следовательно, общее время, требуемое для выполнения cn операций НАЙТИ, состоит из времени, не большего $cnG(n)$, на которое оштрафовано выполнение этих операций, и времени, не большего $nG(n)$, на которое оштрафованы узлы. Таким образом, мы доказали следующую теорему.

Теорема 4.4. *Пусть c — произвольная постоянная. Найдется другая постоянная c', зависящая от c и такая, что алгоритм* 4.3 *выполнит последовательность σ из cn операций* ОБЪЕДИНИТЬ *и* НАЙТИ *на n элементах не более чем за $c'nG(n)$ единиц времени.*

Доказательство. Рассуждать, как выше. □

В качестве упражнения предлагаем показать, что если в последовательности σ наряду с операциями ОБЪЕДИНИТЬ и НАЙТИ допускаются основные операции ВСТАВИТЬ и УДАЛИТЬ, то ее все равно можно выполнить за время $O(nG(n))$.

Неизвестно, точна ли оценка времени работы алгоритма 4.3, даваемая теоремой 4.4[1]). Однако в оставшейся части раздела мы докажем, поскольку это теоретически интересно, что время работы ал-

[1]) По поводу сложности задачи, рассматриваемой в данном разделе, см. работу Тарьяна [1977].— *Прим. перев.*

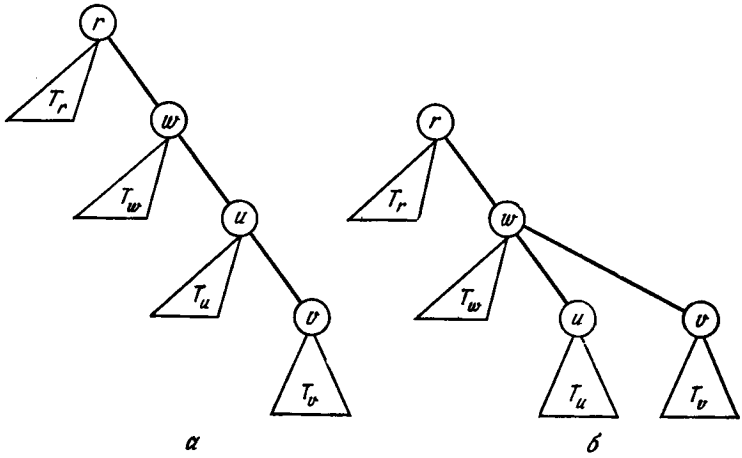


Рис. 4.21. Результат операции ЧН.

горитма 4.3 нелинейно по n. Для этого построим конкретную последовательность операций ОБЪЕДИНИТЬ и НАЙТИ, на которой алгоритм 4.3 работает нелинейное время.

Удобно ввести новую операцию над деревьями, которую мы будем называть ЧАСТИЧНО НАЙТИ — сокращенно ЧН. Пусть T — дерево, в котором узлы v, v_1, v_2, . . ., v_m, w образуют путь из узла v к его предку w (w не обязательно корень). Операция ЧН(v, w) делает каждый из узлов v, v_1, v_2, . . ., v_{m-1} сыном узла w. Мы будем говорить, что эта операция ЧН *имеет длину* $m+1$ (если $v=w$, то длина равна 0). На рис. 4.21,*б* отражено влияние операции ЧН(v, w) на дерево, приведенное на рис. 4.21,*а*.

Пусть дана последовательность σ операций ОБЪЕДИНИТЬ и НАЙТИ. Когда выполняется данная операция НАЙТИ из σ, мы находим положение узла v в некотором дереве T и идем по пути из v в корень w этого дерева. Теперь предположим, что выполняются только операции ОБЪЕДИНИТЬ из σ, а операции НАЙТИ не выполняются. В результате получается некоторый лес F. Действие данной операции НАЙТИ из σ все еще можно уловить, если найти в F положение узлов v и w, которые должна была использовать эта операция НАЙТИ, и затем выполнить ЧН(v, w). Заметим, что узел w может больше не быть корнем в F.

Для вывода нижней оценки времени работы алгоритма 4.3 исследуем его поведение на последовательности операций НАЙТИ, за которыми следуют операции ЧН. Эту последовательность можно заменить последовательностью операций ОБЪЕДИНИТЬ и НАЙТИ с тем же временем работы. Из описанных ниже специальных де-

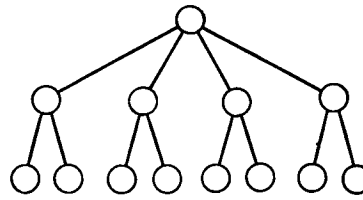


Рис. 4.22. Дерево $T(2)$.

ревьев мы построим последовательности операций ОБЪЕДИНИТЬ и ЧН, на которых время работы алгоритма 4.3 более чем линейно.

Определение. Для $k \geqslant 0$ пусть $T(k)$ — дерево, в котором

1) глубина каждого листа равна k
2) каждый узел высоты h имеет 2^h сыновей, $h \geqslant 1$.

Таким образом, корень дерева $T(k)$ имеет 2^k сыновей, каждый из которых является корнем экземпляра дерева $T(k-1)$. На рис. 4.22 изображено дерево $T(2)$.

Лемма 4.4. *Для каждого $k \geqslant 0$ можно построить дерево $T'(k)$, которое порождается некоторой последовательностью операций* ОБЪЕДИНИТЬ *и содержит в качестве подграфа дерево $T(k)$, причем по меньшей мере четверть узлов дерева $T'(k)$ являются листьями в $T(k)$.*

Доказательство. Доказательство проводится индукцией по k. Для $k=0$ лемма тривиальна, ибо $T(0)$ состоит из одного узла. Чтобы построить $T'(k)$ для $k>0$, построим сначала 2^k+1 экземпляров $T'(k-1)$. Образуем дерево $T'(k)$, выбрав один экземпляр $T'(k-1)$ и затем влив в него один за другим каждый из оставшихся экземпляров. Корень полученного дерева имеет (среди других) 2^k сыновей, каждый из которых является корнем дерева $T'(k-1)$.

Пусть $N'(k)$ — общее число узлов в $T'(k)$ и $L(k)$ — число листьев в $T(k)$. Тогда

$$N'(0) = 1,$$
$$N'(k) = (2^k + 1)\, N'(k-1) \quad \text{для} \quad k \geqslant 1$$

и

$$L(0) = 1,$$
$$L(k) = 2^k L(k-1) \quad \text{для} \quad k \geqslant 1,$$

откуда

$$\frac{L(k)}{N'(k)} = \frac{\prod_{i=1}^{k} 2^i}{\prod_{i=1}^{k} (2^i + 1)} = \frac{2}{3} \prod_{i=2}^{k} \frac{1}{1 + 2^{-i}} \quad \text{для} \quad k \geqslant 1. \qquad (4.3)$$

Заметим, что для $i \geqslant 2$ справедливо неравенство $\ln(1+2^{-i}) < 2^{-i}$, поэтому

$$\ln\left(\prod_{i=2}^{k} \frac{1}{1+2^{-i}}\right) \geqslant -\sum_{i=2}^{k} 2^{-i} \geqslant -\frac{1}{2}. \tag{4.4}$$

Комбинируя (4.3) с (4.4), получаем

$$\frac{L(k)}{N'(k)} \geqslant \frac{2}{3} e^{-1/2} \geqslant \frac{1}{4}.$$

Лемма доказана. □

Мы будем строить последовательность операций ОБЪЕДИНИТЬ и ЧН, которая сначала даст дерево $T'(k)$, а затем выполнит ЧН на листьях подграфа $T(k)$. Сначала покажем, что для каждого $l>0$ найдется такое число k, что можно выполнить ЧН длины l последовательно на каждом листе дерева $T(k)$.

Определение. Пусть $D(c, l, h)$ — такое наименьшее значение k, что если заменить каждое поддерево в $T(k)$ с корнем высоты h любым деревом, имеющим l листьев и высоту не меньше 1, то на каждом листе полученного дерева можно будет выполнить ЧН длины c.

Лемма 4.5. *Значение $D(c, l, h)$ определено (т. е. конечно) для всех c, l и h, больших нуля.*

Доказательство. В доказательстве применяется двойная индукция. Мы хотим доказать наше утверждение индукцией по c. Но для доказательства этого утверждения для c, исходя из истинности его для $c-1$, надо провести индукцию по l.

Базис, т. е. случай $c=1$, доказывается легко. $D(1, l, h)=h$ для всех l и h, поскольку ЧН длины 1 не перемещает никаких узлов.

Теперь, чтобы провести индукцию по c, предположим, что для всех l и h определено значение $D(c-1, l, h)$. Мы должны показать, что $D(c, l, h)$ определено для всех l и h. Это делается индукцией по l.

Для базиса индукции по l докажем, что

$$D(c, 1, h) \leqslant D(c-1, 2^{h+1}, h+1).$$

Заметим, что по определению числа $D(c, l, h)$ при $l=1$ поддеревья в $T(k)$ с корнями высоты h заменяются деревьями с единственным листом. Пусть H — множество узлов высоты h в дереве $T(k)$. Очевидно, что в этом преобразованном дереве каждый лист является подлинным потомком единственного элемента множества H. Поэтому, если бы мы смогли выполнить операции ЧН длины $c-1$ на всех элементах множества H, то, конечно, смогли бы выполнить ЧН длины c на всех листьях.

Пусть $k=D(c-1,\ 2^{h+1},\ h+1)$. По предположению индукции для c число k существует. Все узлы высоты $h+1$ в $T(k)$ имеют по 2^{h+1} сыновей каждый, причем все сыновья принадлежат H. Если удалить из $T(k)$ всех подлинных потомков узлов, входящих в H, то тем самым каждое поддерево с корнями на высоте $h+1$ заменится деревом высоты 1 с 2^{h+1} листьями. По определению D число $k=D(c-1,\ 2^{h+1},\ h+1)$ достаточно велико, так что операции ЧН длины $c-1$ можно выполнить на всех листьях, т. е. элементах множества H.

Теперь, чтобы завершить индукцию по c, осталось сделать шаг индукции по l. В частности, покажем, что для $l>1$

$$D(c,\ l,\ h)\leqslant D(c-1,\ 2^{D(c,\ l-1,\ h)(1+D(c,\ l-1,\ h))/2},\ D(c,\ l-1,\ h)). \quad (4.5)$$

Чтобы доказать неравенство (4.5), положим $k=D(c,\ l-1,\ h)$ и обозначим через k' его правую часть. Нам надо найти способ заменить каждый узел высоты h в $T(k')$ деревом с l листьями, а затем на каждом листе выполнить операцию ЧН длины c. Начнем с выполнения операций ЧН на $l-1$ листьях каждого подставляемого дерева. По предположению индукции по l это сделать можно.

Выполнив ЧН на $l-1$ листьях, находим, что l-й лист каждого подставляемого дерева теперь имеет отца, отличного от отца l-го листа любого другого подставляемого дерева. Множество таких отцов обозначим через F. Если мы смогли выполнить операции ЧН длины $c-1$ на этих отцах, то можем выполнить ЧН длины c на листьях. Пусть S — поддерево с корнем высоты k в $T(k')$. Легко проверить, что S содержит $2^{k(k+1)/2}$ листьев в $T(k')$. Поэтому после выполнения операций ЧН число узлов в S, которые также принадлежат F, не превосходит $2^{k(k+1)/2}$. Множество, оставшееся от S, можно, следовательно, рассматривать как произвольное дерево с $2^{k(k+1)/2}$ листьями, которые принадлежат F. По предположению индукции для c и l неравенство (4.5) справедливо. □

Теорема 4.5. *Временная сложность алгоритма* 4.3 *больше* cn *для любой постоянной* c.

Доказательство. Пусть c — такая постоянная, что алгоритм 4.3 выполнит любую последовательность из $n-1$ операций ОБЪЕДИНИТЬ и n операций НАЙТИ не более чем за cn единиц времени. Выберем $d>4c$ и вычислим $k=D(d,\ 1,\ 1)$. Построим $T'(k)$ с помощью последовательности операций ОБЪЕДИНИТЬ. Так как можно выполнить ЧН длины d на каждом листе вложенного дерева $T(k)$, листья которого занимают более четверти узлов дерева $T'(k)$, то эта последовательность операций ОБЪЕДИНИТЬ и ЧН потратит более cn единиц времени; получили противоречие. □

4.8. ПРИЛОЖЕНИЯ И ОБОБЩЕНИЯ АЛГОРИТМА ОБЪЕДИНИТЬ — НАЙТИ

Мы уже видели, как в задаче построения остовного дерева, описанной в примере 4.1, естественно возникает последовательность основных операций ОБЪЕДИНИТЬ и НАЙТИ. В этом разделе мы познакомимся с несколькими другими задачами, которые приводят к последовательностям операций ОБЪЕДИНИТЬ и НАЙТИ. В нашей первой задаче надо осуществить вычисления в свободном режиме, т. е. прочесть всю последовательность операций перед выдачей каких бы то ни было ответов.

Приложение 1. MIN-задача в свободном режиме

Даны два типа операций: ВСТАВИТЬ(i) и ИЗВЛЕЧЬ_MIN. Начнем с множества S, вначале пустого. Каждый раз, когда встречается операция ВСТАВИТЬ(i), целое число i помещается в S. Каждый раз, когда выполняется операция ИЗВЛЕЧЬ_MIN, разыскивается и удаляется наименьший элемент в S.

Пусть σ — такая последовательность операций ВСТАВИТЬ и ИЗВЛЕЧЬ_MIN, что для каждого i, $1 \leqslant i \leqslant n$, операция ВСТАВИТЬ($i$) встречается не более одного раза. По данной последовательности σ надо найти последовательность целых чисел, удаляемых операцией ИЗВЛЕЧЬ_MIN. Задача решается в свободном режиме, поскольку предполагается, что вся последовательность σ известна до того, как надо вычислить даже первый элемент выходной последовательности.

MIN-задачу в свободном режиме можно решить следующим методом. Пусть k — число операций ИЗВЛЕЧЬ_MIN в σ. Можно записать σ в виде $\sigma_1 E \sigma_2 E \sigma_3 E \ldots \sigma_k E \sigma_{k+1}$, где каждая подпоследовательность σ_j, $1 \leqslant j \leqslant k+1$, состоит только из операций ВСТАВИТЬ,

```
for i ← 1 until n do
  begin
    j ← НАЙТИ(i);
    if j ⩽ k then
      begin
        print i "удаляется j-й операцией ИЗВЛЕЧЬ_MIN";
        ОБЪЕДИНИТЬ(j, СЛЕД[j], СЛЕД[j]);
        СЛЕД[ПРЕД[j]] ← СЛЕД[j];
        ПРЕД[СЛЕД[j]] ← ПРЕД[j]
      end
  end
```

Рис. 4.23. Программа для решения MIN-задачи в свободном режиме.

а E обозначает операцию ИЗВЛЕЧЬ_MIN. Промоделируем σ с помощью алгоритма 4.3. Начальная последовательность множеств для работы алгоритма объединения строится так: если в последовательности σ_j встречается операция ВСТАВИТЬ(i), то считаем, что множество с именем j, $1 \leqslant j \leqslant k+1$, содержит элемент i. Для того чтобы для тех значений j, для которых существует множество с именем j, образовать дважды связанный упорядоченный список, пользуемся двумя массивами ПРЕД и СЛЕД. Вначале ПРЕД[j]$=j-1$ для $1 \leqslant j \leqslant k+1$ и СЛЕД[j]$=j+1$ для $0 \leqslant j \leqslant k$. Затем выполняется программа, приведенная на рис. 4.23.

Легко видеть, что время выполнения этой программы ограничено временем работы алгоритма объединения множеств. Следовательно, MIN-задача в свободном режиме имеет временную сложность $O(nG(n))$.

Пример 4.8. Рассмотрим последовательность операций $\sigma = 4\ 3\ E\ 2\ E\ 1\ E$, где j означает ВСТАВИТЬ(j), а E — ИЗВЛЕЧЬ_MIN. Тогда $\sigma_1 = 4\ 3$, $\sigma_2 = 2$, $\sigma_3 = 1$ и σ_4 — пустая последовательность. Начальная структура данных представляет собой последовательность множеств

$$1 = \{3, 4\}, \quad 2 = \{2\}, \quad 3 = \{1\}, \quad 4 = \varnothing.$$

При первом выполнении **for**-цикла выясняется, что НАЙТИ(1)$=3$. Следовательно, ответом на третью операцию в σ ИЗВЛЕЧЬ_MIN будет 1. Последовательность множеств превращается в

$$1 = \{3, 4\}, \quad 2 = \{2\}, \quad 4 = \{1\}.$$

В этот момент СЛЕД[2]$=4$ и ПРЕД[4]$=2$, поскольку множества с именами 3 и 4 были слиты в одно множество с именем 4.

При следующем прохождении с $i=2$ выясняется, что НАЙТИ(2)$=2$. Таким образом, ответом на вторую операцию ИЗВЛЕЧЬ_MIN будет 2. Множество с именем 2 сливается со следующим множеством (с именем 4) и получается последовательность множеств

$$1 = \{3, 4\}, \quad 4 = \{1, 2\}.$$

Два последних прохождения устанавливают, что ответ на первую операцию ИЗВЛЕЧЬ_MIN есть 3 и что 4 никогда не извлекается. □

Рассмотрим другое приложение — задачу определения глубины. В частности, она возникает при "приравнивании" идентификаторов в программе на языке ассемблера. Во многих языках ассемблера есть операторы, описывающие, что два идентификатора представляют одну и ту же ячейку памяти. Как только ассемблер встречает оператор, приравнивающий два идентификатора α и β, он должен найти два множества S_α и S_β идентификаторов, экви-

валентных соответственно α и β, и заменить эти два множества их объединением. Очевидно, задачу можно смоделировать последовательностью операций ОБЪЕДИНИТЬ и НАЙТИ.

Однако если внимательнее проанализировать эту задачу, можно по-другому применить структуры данных из предыдущего раздела. Каждому идентификатору соответствует графа таблицы символов, и, если несколько идентификаторов эквивалентны, удобно хранить данные о них только в одной графе таблицы символов. Это означает, что для каждого множества эквивалентных идентификаторов есть *начало отсчета*, т. е. место в таблице символов, где хранится информация об этом множестве, и каждый элемент этого множества имеет *смещение* от начала. Чтобы найти положение идентификатора в таблице символов, надо прибавить его смещение к началу отсчета множества, которому принадлежит данный идентификатор. Но когда два множества идентификаторов становятся эквивалентными, надо изменить смещение. Эта задача корректировки смещений в абстрактном виде решается в приложении 2.

Приложение 2. Задача определения глубины

Дана последовательность операций двух типов: СВЯЗАТЬ(v, r) и НАЙТИ_ГЛУБИНУ(v). Начнем с n неориентированных корневых деревьев, каждое из которых состоит из единственного узла i, $1 \leqslant i \leqslant n$. Операция СВЯЗАТЬ($v$, r), где r — корень дерева, а v — узел другого дерева, делает корень r сыном узла v. Условия о том, что v и r принадлежат различным деревьям и что r — корень, гарантируют, что получаемый в результате граф также будет лесом. Операция НАЙТИ_ГЛУБИНУ(v) состоит в том, чтобы найти и напечатать глубину узла v в текущий момент.

Если при работе с лесом использовать обычное представление в виде списков смежностей и вычислять глубину узлов очевидным образом, то сложность алгоритма будет $O(n^2)$. Вместо этого мы представим исходный лес другим лесом, который будем называть D-лесом. Он нужен нам только для того, чтобы можно было быстро вычислять глубины. Каждому узлу в D-лесу приписывается целочисленный вес так, чтобы сумма весов вдоль пути в D-лесу от узла v к корню равнялась глубине узла v в исходном лесу. Для каждого дерева в D-лесу хранится счетчик числа узлов в этом дереве.

Вначале D-лес состоит из n деревьев, каждое из которых содержит единственный узел, соответствующий целому числу i, $1 \leqslant i \leqslant n$. Начальный вес каждого узла равен нулю.

Для выполнения операции НАЙТИ_ГЛУБИНУ(v) мы проходим путь из узла v в корень r. Пусть $v_1, v_2, \ldots, v_k$ — узлы на этом пути ($v_1 = v$, $v_k = r$). Тогда

$$\text{ГЛУБИНА}(v) = \sum_{i=1}^{k} \text{ВЕС}[v_i].$$

Кроме того, применяем сжатие путей. Каждый узел v_i, $1 \leqslant i \leqslant k-2$, делается сыном корня r. Чтобы сохранялось сформулированное выше свойство весов, новый ВЕС узла v_i должен равняться $\sum_{j=i}^{k-1} \text{ВЕС}[v_j]$ для $1 \leqslant i < k$. Так как новые веса можно вычислить за время $O(k)$, то операция НАЙТИ_ГЛУБИНУ имеет ту же временну́ю сложность, что и операция НАЙТИ.

Чтобы выполнить операцию СВЯЗАТЬ(v, r), соединим деревья, содержащие узлы v и r, снова вливая меньшее дерево в большее. Пусть T_v и T_r — деревья в D-лесу, содержащие v и r соответственно, а v' и r' — их корни. Деревья в D-лесу не обязательно изоморфны деревьям в исходном лесу, так что, в частности, r может не быть корнем для T_r. Пусть СЧЁТ(T) обозначает число узлов в дереве T. Рассмотрим отдельно два случая.

Случай 1. СЧЕТ(T_r)$\leqslant$СЧЕТ(T_v). Делаем r' сыном узла v'. Мы должны также скорректировать вес старого корня r' дерева T_r так, чтобы глубина каждого узла w в T_r правильно вычислялась при прохождении пути из w в v' в объединенном дереве. Чтобы сделать это, выполняем операцию НАЙТИ_ГЛУБИНУ(v), а затем

$$\text{ВЕС}[r'] \leftarrow \text{ВЕС}[r'] - \text{ВЕС}[v'] + \text{ГЛУБИНА}(v) + 1.$$

Таким образом, глубина каждого узла в T_r эффективно увеличена на глубину узла v плюс 1. Наконец, полагаем счетчик числа узлов объединенного дерева равным сумме счетчиков для T_r и T_v.

Случай 2. СЧЕТ(T_v)$<$СЧЕТ(T_r). Здесь вычисляется ГЛУБИНА(v), преобразуется узел v' в сына узла r' и

$$\begin{aligned} \text{ВЕС}[r'] &\leftarrow \text{ВЕС}[r'] + \text{ГЛУБИНА}(v) + 1; \\ \text{ВЕС}[v'] &\leftarrow \text{ВЕС}[v'] - \text{ВЕС}[r']; \\ \text{СЧЕТ}(T_r) &\leftarrow \text{СЧЕТ}(T_v) + \text{СЧЕТ}(T_r). \end{aligned}$$

Итак, $O(n)$ операций СВЯЗАТЬ и НАЙТИ_ГЛУБИНУ можно выполнить за время $O(nG(n))$.

Приложение 3. Эквивалентность конечных автоматов

Детерминированным конечным автоматом называется машина, распознающая цепочки символов. Она имеет входную ленту, разбитую на клетки, головку на входной ленте (входную головку) и управляющее устройство с конечным числом состояний (рис. 4.24). Конечный автомат M можно представить в виде пятерки (S, I, δ, s_0, F), где

1) S — множество состояний *управляющего устройства*,
2) I — *входной алфавит* (каждая клетка входной ленты содержит символ из I),

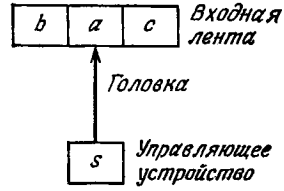


Рис. 4.24. Конечный автомат.

3) δ — отображение из $S \times I$ в S (если $\delta(s, a)=s'$, то всякий раз, когда M находится в состоянии s, а входная головка обозревает символ a, M сдвигает входную головку вправо и переходит в состояние s'),
4) s_0 — выделенное состояние в S, называемое *начальным*,
5) F — подмножество в S, называемое множеством *допускающих* (или *заключительных*) *состояний*.

Пусть I^* — множество цепочек (слов) конечной длины, состоящих из символов алфавита I. В I^* включается и пустая цепочка ε. Продолжим δ до отображения из $S \times I^*$ в S:

1) $\delta(s, \varepsilon)=s$,
2) $\delta(s, xa)=\delta(\delta(s, x), a)$ для всех $x \in I^*$ и $a \in I$.

Входная цепочка x *допускается* автоматом M, если $\delta(s_0, x) \in F$. *Языком* $L(M)$, *допускаемым автоматом* M, называется множество всех цепочек, допускаемых M. Более подробное введение в теорию конечных автоматов изложено в разд. 9.1.

Два состояния s_1 и s_2 считаются *эквивалентными*, если для каждого $x \in I^*$ состояние $\delta(s_1, x)$ будет допускающим тогда и только тогда, когда $\delta(s_2, x)$ — допускающее состояние.

Два конечных автомата M_1 и M_2 считаются *эквивалентными*, если $L(M_1)=L(M_2)$. Мы покажем здесь, что с помощью алгоритма ОБЪЕДИНИТЬ — НАЙТИ можно распознать эквивалентность двух конечных автоматов $M_1=(S_1, I, \delta_1, s_1, F_1)$ и $M_2=(S_2, I, \delta_2, s_2, F_2)$ за $O(nG(n))$ шагов, где $n=\|S_1\|+\|S_2\|$.

Отношение эквивалентности двух состояний обладает важным свойством: если два состояния s и s' эквивалентны, то для всех входных символов a состояния $\delta(s, a)$ и $\delta(s', a)$ также эквивалентны. Кроме того, благодаря наличию пустой цепочки, никакое допускающее состояние не может оказаться эквивалентным недопускающему. Таким образом, если допустить, что начальные состояния s_1 и s_2 автоматов M_1 и M_2 эквивалентны, то можно вывести другие пары эквивалентных состояний. Если в одну из таких пар попадет допускающее состояние вместе с недопускающим, то s_1 и s_2 были неэквивалентными. Если же это не произойдет, то верно обратное

```
begin
  СПИСОК ← (s1, s2);
  НАБОР ← ∅;
  for s ∈ S1 ∪ S2 do добавить {s} в НАБОР;
  comment Мы только что образовали исходные множества для
      каждого состояния из S1 ∪ S2;
  while есть пара (s, s'), входящая в СПИСОК do
    begin
      удалить (s, s') из множества СПИСОК;
      пусть A и A' обозначают НАЙТИ(s) и НАЙТИ(s') со-
        ответственно;
      if A ≠ A' then
        begin
          ОБЪЕДИНИТЬ(A, A', A);
          for a ∈ I do
            добавить (δ(s, a), δ(s', a)) в СПИСОК
        end
    end
end
```

Рис. 4.25. Алгоритм для нахождения множеств эквивалентных состояний в предположении, что s_1 и s_2 эквивалентны.

Чтобы узнать, эквивалентны ли два конечных автомата $M_1 = (S_1, I, \delta_1, s_1, F_1)$ и $M_2 = (S_2, I, \delta_2, s_2, F_2)$, мы поступаем так:

1. С помощью программы на рис. 4.25 находим все множества состояний, которые должны быть эквивалентными, если эквивалентны s_1 и s_2. СПИСОК содержит такие пары состояний (s, s'), что s и s' оказались эквивалентными, а следующие за ними состояния $(\delta(s, a), \delta(s', a))$ еще не рассматривались. Вначале СПИСОК содержит только пару (s_1, s_2). Чтобы найти множества эквивалентных состояний, программа применяет алгоритм объединения непересекающихся множеств. НАБОР представляет некоторое семейство множеств. Вначале каждое состояние из $S_1 \cup S_2$ образует одноэлементное множество. (Без потери общности можно считать, что множества S_1 и S_2 не пересекаются.) Затем всякий раз, когда s и s' оказываются эквивалентными, содержащие их множества A и A', входящие в НАБОР, сливаются, и новое множество получает имя A.

2. После выполнения этой программы множества, входящие в НАБОР, представляют разбиение множества $S_1 \cup S_2$ на блоки

состояний, которые должны быть эквивалентными. M_1 и M_2 эквивалентны тогда и только тогда, когда никакой блок не содержит допускающего состояния вместе с недопускающим.

Время работы этого алгоритма (как функция от числа состояний $n=\|S_1\|+\|S_2\|$) определяется в основном работой алгоритма объединения множеств. Операций ОБЪЕДИНИТЬ может быть не более n—1, поскольку каждая такая операция уменьшает на единицу число множеств, входящих в НАБОР, а вначале было только n множеств. Число операций НАЙТИ пропорционально числу пар, помещенных в СПИСОК. Это число не больше $n\|I\|$, так как всякая пара, кроме начальной (s_1, s_2), попадает в СПИСОК только после операции ОБЪЕДИНИТЬ. Таким образом, при постоянном размере входного алфавита на распознавание эквивалентности конечных автоматов M_1 и M_2 тратится время $O(nG(n))$.

4.9. СХЕМЫ СБАЛАНСИРОВАННЫХ ДЕРЕВЬЕВ

При решении ряда важных классов задач, аналогичных задаче ОБЪЕДИНИТЬ — НАЙТИ, мы, по-видимому, вынуждены вернуться к способам, при которых последовательность из n операций выполняется за время $O(n \log n)$ (в худшем случае). В один такой класс входят задачи, в которых последовательность из операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ и УДАЛИТЬ выполняется в случае, когда допускается гораздо больше элем нтов, чем используется на самом деле. В этом случае нельзя выбрать элемент, обращаясь прямо в массив указателей. Надо применять расстановку или дерево двоичного поиска.

Если n элементов уже вставлены, то в методе расстановки один выбор осуществляется за постоянное время в среднем и $O(n)$ в худшем случае. Дерево двоичного поиска дает среднее время $O(\log n)$ на один выбор, но в худшем случае может тоже дать плохое время выбора, если множество имен изменяется. Если же просто добавлять имена к дереву, не имея никакого механизма для поддержания его сбалансированности, то можно в результате придти к дереву с n узлами, глубина которого близка к n. Поэтому в худшем случае производительность дерева двоичного поиска будет $O(n)$ шагов на операцию. Методом, изложенным в этом разделе, можно уменьшить сложность в худшем случае до $O(\log n)$ шагов на операцию.

Другой класс задач, требующих $O(n \log n)$ времени, образуют задачи выполнения последовательности из n операций, включающих ВСТАВИТЬ, УДАЛИТЬ и MIN, в префиксном режиме. Еще один, третий, класс задач возникает, когда нужно представлять упорядоченные списки и уметь сцеплять и расцеплять их.

В настоящем разделе мы познакомимся с техникой, позволяющей выполнять в префиксном режиме последовательности, содер-

жащие важные подмножества семи основных операций на множествах, введенных в разд. 4.1. Структурой данных, лежащей в основе метода, является *сбалансированное дерево*, под которым мы понимаем дерево с высотой, приблизительно равной логарифму числа его узлов.

Вначале сбалансированное дерево строится легко. Трудно не дать ему в процессе выполнения последовательности операций ВСТАВИТЬ и УДАЛИТЬ превратиться в несбалансированное. Например, если периодически не делать перебалансировку, то при выполнении последовательности операций УДАЛИТЬ, которые удаляют узлы только из левой части дерева, получается дерево, смещенное вправо.

Известно много способов перебалансировки дерева в случае необходимости. Некоторые из них оставляют структуру дерева достаточно гибкой, так что число узлов в дереве высоты h может изменяться от 2^h до 2^{h+1} или 3^h. В них позволяется по меньшей мере удвоить число узлов в поддереве, прежде чем что-то менять выше его корня.

Мы обсудим два метода этого рода, называемые методами 2-3-деревьев и АВЛ-деревьев. Алгоритмы, работающие с 2-3-деревьями, понять легче, и сейчас мы обсудим их. Алгоритмы с АВЛ-деревьями похожи на них, и потому вынесены в упражнения.

Определение. 2-3-*деревом* называется дерево, в котором каждый узел, не являющийся листом, имеет двух или трех сыновей, а длины всех путей из корня в листья одинаковы. Заметим, что дерево, состоящее из единственного узла, является 2-3-деревом. На рис. 4.26 приведены два 2-3-дерева с шестью листьями.

В следующей лемме показана связь числа узлов и числа листьев в 2-3-дереве с его высотой.

Лемма 4.6. *Пусть T будет 2-3-деревом высоты h. Число узлов дерева T заключено между $2^{h+1}-1$ и $(3^{h+1}-1)/2$, а число листьев — между 2^h и 3^h.*

Доказательство. Элементарная индукция по h. □

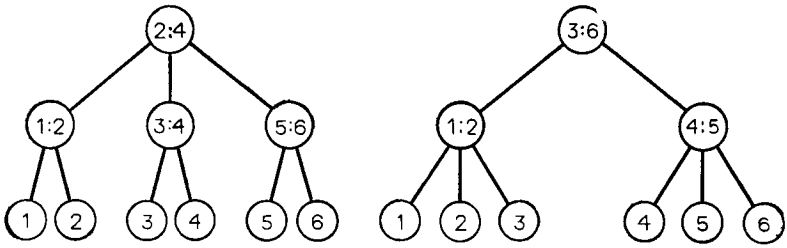


Рис. 4.26. 2-3-деревья.

Линейно упорядоченное множество S можно представить 2-3-деревом, приписав его элементы листьям дерева. Обозначим элемент, приписанный листу l, через $E[l]$. Существуют два основных метода приписывания элементов листьям; какой из них применить, зависит от рассматриваемой задачи.

Если допускается элементов гораздо больше, чем на самом деле используется, и дереву предстоит быть словарем, то, вероятно, лучше приписывать элементы в порядке возрастания слева направо. В каждом узле v, не являющемся листом, нам потребуется еще два данных: $L[v]$ и $M[v]$. $L[v]$ — это наибольший элемент множества S в поддереве, корнем которого служит самый левый сын узла v; $M[v]$ — это наибольший элемент множества S в поддереве, корнем которого служит второй сын узла v (см. рис. 4.26). Значения L и M, приписанные узлам, позволяют искать элемент, начиная с корня, способом, аналогичным двоичному поиску. Время обнаружения произвольного элемента пропорционально высоте дерева. Поэтому операцию ПРИНАДЛЕЖАТЬ можно выполнить на множестве из n элементов за время $O(\log n)$, если представить его в виде 2-3-дерева такого рода.

Во втором методе приписывания элементов листьям на порядок, в котором приписываются эти элементы, не налагается никаких ограничений. Метод особенно полезен для реализации операции ОБЪЕДИНИТЬ. Однако для выполнения операций типа УДАЛИТЬ нужен механизм для определения положения листа, представляющего данный элемент. Если элементами множества являются целые числа из фиксированной области, скажем от 1 до n, то лист, представляющий элемент i, можно находить с помощью i-й ячейки некоторого массива. Если же элементы рассматриваемых множеств принадлежат некоторому большому универсальному множеству, то лист, представляющий элемент i, можно находить с помощью вспомогательного словаря.

Рассмотрим следующие наборы операций.

1) ВСТАВИТЬ, УДАЛИТЬ, ПРИНАДЛЕЖАТЬ.
2) ВСТАВИТЬ, УДАЛИТЬ, MIN.
3) ВСТАВИТЬ, УДАЛИТЬ, ОБЪЕДИНИТЬ, MIN.
4) ВСТАВИТЬ, УДАЛИТЬ, НАЙТИ, СЦЕПИТЬ, РАСЦЕПИТЬ.

Структуру данных, обеспечивающую выполнение операций из множества 1, будем называть *словарем*, из множества 2 — *очередью с приоритетами*, из множества 3 — *сливаемым деревом*, из множества 4 — *сцепляемой очередью*.

Покажем, что 2-3-деревья могут служить для реализации словарей, очередей с приоритетами, сцепляемых очередей и сливаемых деревьев, применение которых обеспечивает выполнение n операций за время $O(n \log n)$.

Развитая здесь техника достаточно мощна, чтобы выполнить последовательности, составленные из любого совместного подмножества семи операций, перечисленных в начале главы. Единственная несовместность состоит в том, что операция ОБЪЕДИНИТЬ приводит к неупорядоченному множеству, а РАСЦЕПИТЬ и СЦЕПИТЬ предполагают наличие порядка.

4.10. СЛОВАРИ И ОЧЕРЕДИ С ПРИОРИТЕТАМИ

В этом разделе мы изучим основные преобразования, требуемые для реализации словарей и очередей с приоритетами. На протяжении всего раздела будем предполагать, что элементы приписаны листьям 2-3-дерева в порядке слева направо и в каждом нелисте v определены функции $L[v]$ и $M[v]$, введенные в предыдущем разделе.

Чтобы в 2-3-дерево вставить новый элемент a, надо найти место для нового листа l, который будет содержать a. Для этого ищут элемент a в дереве. Если дерево содержит более одного элемента, то поиск a окончится в узле f, имеющем двух или трех сыновей, которые являются листьями.

Если из узла f выходит только два листа l_1 и l_2, то l делаем сыном узла f. Если $a<E[l_1]$ [1]), то l делаем самым левым сыном узла f и полагаем $L[f]=a$ и $M[f]=E[l_1]$; если $E[l_1]<a<E[l_2]$, то l делаем средним сыном узла f и полагаем $M[f]=a$; если $E[l_2]<a$, то l делаем третьим сыном узла f. В последнем случае, возможно, надо будет изменить значения L и M на некоторых подлинных предках узла f.

Пример 4.9. Если в 2-3-дерево, изображенное на рис. 4.27,*а*, вставляется элемент 2, то получается 2-3-дерево, изображенное на рис. 4.27,*б*. □

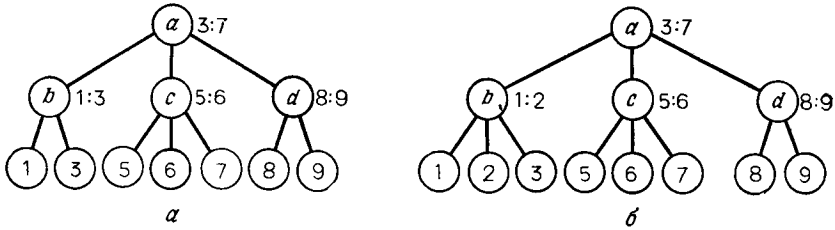


Рис. 4.27. Вставка в 2-3-дерево: *а* — дерево перед вставкой; *б* — дерево после вставки элемента 2.

Теперь предположим, что у f уже есть три листа l_1, l_2 и l_3. Сделаем l надлежащим сыном узла f. Теперь f имеет четырех сыновей. Чтобы сохранить 2-3-свойство, образуем новый узел g. Два левых

[1]) $E[v]$ — элемент, приписанный узлу v.

сына оставим сыновьями узла f, а два правых переделаем в сыновей узла g. Затем сделаем g братом узла f, сделав его сыном отца узла f. Если отец узла f имел двух сыновей, то на этом мы остановимся. Если же трех, то надо рекурсивно повторять эту процедуру до тех пор, пока у всех узлов в дереве останется не более трех сыновей. Если у корня окажется четыре сына, образуем новый корень с двумя новыми сыновьями, каждый из которых будет иметь в качестве двух своих сыновей двух из четырех сыновей старого корня.

Пример 4.10. Если в 2-3-дерево на рис. 4.27,*а*, вставляется элемент 4, то новый лист с меткой 4 надо сделать самым левым сыном узла c. Поскольку у c уже есть три сына, строим новый узел c'. Затем делаем листья 4 и 5 сыновьями узла c, а листья 6 и 7 — сыновьями узла c'. Теперь делаем c' сыном узла a. Но поскольку у a уже есть три сына, строим новый узел a'. Делаем узлы b и c сыновьями старого узла a, а узлы c' и d — сыновьями нового узла a'. Наконец, образуем новый корень r и делаем a и a' его сыновьями. Полученное дерево изображено на рис. 4.28. □

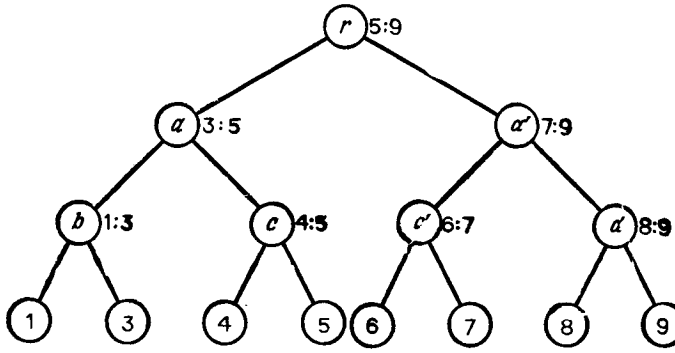


Рис. 4.28. Дерево с рис. 4.27, *а* после вставки элемента 4.

Алгоритм 4.4. Вставка нового элемента в 2-3-дерево

Вход. Непустое 2-3-дерево T с корнем r и новый элемент $a \notin T$.

Выход. Преобразованное 2-3-дерево с новым листом, помеченным a.

Метод. По условию T содержит хотя бы один элемент. Чтобы упростить описание алгоритма, опустим детали корректировки L и M.

1. Если T состоит из единственного узла l с меткой b, образуем новый корень r'. Образуем новый узел v с меткой a. Делаем l и v сыновьями корня r', причем l будет левым сыном, если $b<a$, и правым в противном случае.

2. Если T содержит более одного узла, положим f=ПОИСК(a, r); процедура ПОИСК приведена на рис. 4.29. Образуем новый

```
procedure ПОИСК(a, r):
if любой сын узла r является листом then return r
else
  begin
    пусть s_i будет i-м сыном узла r;
    if a ≤ L[r] then return ПОИСК(a, s_1)
    else
      if у r два сына или a ≤ M[r] then return ПОИСК(a, s_2)
      else return ПОИСК(a, s_3)
  end
```

Рис. 4.29. Процедура ПОИСК.

лист l с меткой a. Если у f два сына с метками b_1 и b_2, то делаем l надлежащим сыном узла f. А именно, l будет левым сыном, если $a<b_1$, средним, если $b_1<a<b_2$, и правым, если $b_2<a$. Если у f три сына, делаем l надлежащим сыном узла f, а затем чтобы включить в T узел f и его четырех сыновей, вызываем ДОБАВСЫНА (f). Процедура ДОБАВСЫНА приведена на рис. 4.30. Чтобы учесть присутствие узла a, корректируем значения L и M вдоль пути из a

```
procedure ДОБАВСЫНА (v):
begin
  образовать новый узел v';
  сделать двух самых правых сыновей узла v левым и правым
    сыновьями узла v';
  if у v нет отца then
    begin
      образовать новый корень r;
      сделать v левым, а v' правым сыном корня r
    end
  else
    begin
      пусть f — отец узла v;
      сделать v' сыном узла f, расположенным непосредственно
        справа от v;
      if теперь у f четыре сына then ДОБАВСЫНА(f)
    end
end
```

Рис. 4.30. Процедура ДОБАВСЫНА.

в корень [1]). Другие очевидные изменения, корректирующие значения L и M, производятся процедурой ДОБАВСЫНА (здесь они опущены — предлагаем их в качестве упражнения). □

Теорема 4.6. *Алгоритм* 4.4 *вставляет новый элемент в* 2-3-*дерево с* n *листьями за время, не превосходящее* $O(\log n)$. *Более того, этот алгоритм сохраняет порядок исходных листьев и структуру* 2-3-*дерева.*

Доказательство. Очевидная индукция по числу вызовов процедуры ПОИСК показывает, что новый лист становится сыном того узла, какого надо. Порядок исходных листьев не затрагивается. Что касается времени работы, то в силу леммы 4.6 высота 2-3-дерева с n листьями не превосходит $\log n$. Поскольку ДОБАВСЫНА(v) рекурсивно вызывает себя только на отце узла v, может произойти не более $\log n$ рекурсивных вызовов Каждый вызов ДОБАВСЫНА занимает постоянное время, так что общее время не превосходит $O(\log n)$. □

Элемент a можно удалить из 2-3-дерева способом, по существу обратным к вставке. Пусть a — метка листа l. Рассмотрим отдельно три случая.

Случай 1. Если l — корень, удаляем его. (В этом случае a был единственным элементом в дереве.)

Случай 2. Если l — сын узла, имеющего трех сыновей, удаляем его.

Случай 3. Если l — сын узла, имеющего двух сыновей s и l, то может быть одно из двух:

а) f — корень; удаляем l и f и делаем корнем второго сына s;

б) f — не корень. Допустим, что f имеет брата [2]) слева от себя. Случай, когда брат находится справа, рассматривается аналогично. Если у g только два сына, делаем узел s самым правым сыном узла g, удаляем l и рекурсивно вызываем процедуру удаления, чтобы удалить f. Если у g три сына, то самого правого сына делаем левым сыном узла f и удаляем l.

Пример 4.11. Из 2-3-дерева на рис. 4.28 удалим элемент 4. Лист с меткой 4 является сыном узла c, у которого два сына. Поэтому делаем лист с меткой 5 самым правым сыном узла b, удаляем лист с меткой 4 и затем рекурсивно удаляем узел c.

Узел c — сын узла a, у которого два сына. Узел a' — правый брат узла a. Поэтому по симметрии делаем b самым левым сыном узла a', удаляем c и затем рекурсивно удаляем a.

---

[1]) Достаточно пройти путь от a до такого узла, что a — не наибольший элемент в его поддереве.

[2]) Два узла с одним отцом называются *братьями*.

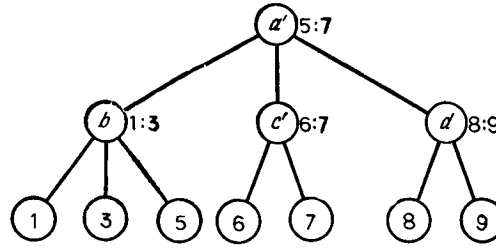


Рис. 4.31. Дерево с рис. 4.28 после удаления элемента 4.

Узел a — сын корня. Применяя случай 3а, делаем a' корнем остающегося дерева (рис. 4.31). □

Формальную детализацию процесса, а также доказательство того, что на 2-3-дереве с n листьями его можно выполнить не более чем за $O(\log n)$ шагов, оставляем в качестве упражнения.

Итак, операции ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ и УДАЛИТЬ на 2-3-дереве с n листьями можно выполнить не более чем за $O(\log n)$ шагов. Следовательно, 2-3-дерево может служить словарем с производительностью $O(n \log n)$, ибо оно может обеспечить выполнение последовательности из n операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ и УДАЛИТЬ не более чем за $O(n \log n)$ шагов.

Исследуем теперь операцию MIN. Наименьший элемент в 2-3-дереве расположен в самом левом листе, который, конечно, можно найти за $O(\log n)$ шагов. Поэтому любую последовательность из n операций ВСТАВИТЬ, УДАЛИТЬ и MIN можно с помощью 2-3-дерева выполнить за время $O(n \log n)$. Тем самым обосновано наше утверждение о том, что 2-3-дерево может служить для реализации очереди с приоритетами с производительностью $O(n \log n)$. Для этой же цели годятся также сортирующее дерево, используемое в алгоритме Сортдеревом, и АВЛ-дерево, обсуждаемое в упр. 4.30—4.33.

4.11. СЛИВАЕМЫЕ ДЕРЕВЬЯ

В данном разделе мы познакомимся со структурой данных, с помощью которой можно выполнить последовательность из операций ВСТАВИТЬ, УДАЛИТЬ, ОБЪЕДИНИТЬ и MIN за время $O(n \log n)$. В этой структуре, которую можно воспринимать как обобщение сортирующего дерева, рассмотренного в разд. 3.4, множество элементов S представляется 2-3-деревом T. Каждый элемент из S появляется в виде метки листа дерева T, но множество листьев не упорядочено, как это было в двух предыдущих разделах. Каждый внутренний узел дерева T пометим значением НАИМЕНЬШИЙ[v], т. е. значением наименьшего элемента, хранящегося в

поддереве с корнем v. Для этого применения 2-3-деревьев $L[v]$ и $M[v]$ не нужны.

Наименьший элемент множества S можно найти, если следующим образом двигаться вниз по дереву T, начиная от его корня. Находясь во внутреннем узле v, переходим к сыну узла v, помеченному наименьшим значением функции НАИМЕНЬШИЙ. Следовательно, если T содержит n листьев, то операция MIN занимает $O(\log n)$ шагов.

Во многих приложениях всякий раз, когда надо удалить из S какой-то элемент, он всегда наименьший. Но если мы хотим удалить из S произвольный элемент, мы должны уметь найти лист, содержащий его. В тех приложениях, где элементы можно представить целыми числами 1, 2, . . ., n, можно пронумеровать сами листья. Если же элементы произвольны, то можно воспользоваться вспомогательным 2-3-словарем, листья которого содержат указатели на листья дерева T. С помощью этого словаря можно достичь произвольного листа за $O(\log n)$ шагов. Словарь надо корректировать

---

procedure ИМПЛАНТАЦИЯ(T_1, T_2):
if ВЫСОТА(T_1) = ВЫСОТА(T_2) **then**
  begin
    образовать новый корень r;
    сделать КОРЕНЬ[T_1] и КОРЕНЬ[T_2] соответственно левым и правым сыновьями узла r;
  end
else
  wlg положим ВЫСОТА(T_1) > ВЫСОТА(T_2) **otherwise** переставить T_1 с T_2 и "левый" с "правым" **in**
  begin
    пусть v—такой узел на самом правом пути в T_1, что ГЛУБИНА(v) = ВЫСОТА(T_1) — ВЫСОТА(T_2);
    пусть f—отец узла v;
    сделать КОРЕНЬ[T_2] сыном узла f, расположенным непосредственно справа от v;
    if у f сейчас четыре сына **then** ДОБАВСЫНА(f) [1])
  end

---

[1]) Если нам нужны значения L и M для нового узла, образованного процедурой ДОБАВСЫНА(f), то сначала надо найти наибольшего потомка узла v, следуя по пути, идущему в самый правый лист.

Рис. 4.32. Процедура ИМПЛАНТАЦИЯ.

всякий раз, когда выполняется операция ВСТАВИТЬ, но это требует не более $O(\log n)$ шагов.

Коль скоро лист l удален из T, надо для каждого его подлинного предка v пересчитать значения функции НАИМЕНЬШИЙ. Новым значением для НАИМЕНЬШИЙ[v] будет наименьшее из значений НАИМЕНЬШИЙ[s] для двух или трех сыновей s узла v. Если всегда пересчитывать снизу вверх, то индукцией по числу пересчетов можно показать, что каждое вычисление дает для функции НАИМЕНЬШИЙ правильный ответ. Так как эта функция меняется только в предках удаленного листа, то операцию УДАЛИТЬ можно выполнить за $O(\log n)$ шагов.

Изучим операцию ОБЪЕДИНИТЬ. Каждое множество представлено отдельным 2-3-деревом. Чтобы слить два множества S_1 и S_2, вызываем процедуру ИМПЛАНТАЦИЯ (T_1, T_2), приведенную на рис. 4.32, где T_1 и T_2 — это 2-3-деревья, представляющие S_1 и S_2 [1]).

Пусть высота h_1 дерева T_1 не меньше высоты h_2 дерева T_2. ИМПЛАНТАЦИЯ находит на самом правом пути в T_1 узел v с высотой h_2 и делает корень дерева T_2 его самым правым братом. Если у его отца f окажется четыре сына, ИМПЛАНТАЦИЯ вызовет процедуру ДОБАВСЫНА (f). Значения функции НАИМЕНЬШИЙ на узлах, потомки которых изменяются в процессе выполнения процедуры ИМПЛАНТАЦИЯ, можно скорректировать тем же способом, что и в операции УДАЛИТЬ.

В качестве упражнения предлагаем показать, что процедура ИМПЛАНТАЦИЯ соединяет T_1 и T_2 в одно 2-3-дерево за время $O(h_1 - h_2)$ (при $h_1 > h_2$). Если учитывать время на корректировку значений L и M, то процедура ИМПЛАНТАЦИЯ может занять $O(\text{MAX}(\log\|S_1\|, \log\|S_2\|))$ времени.

Рассмотрим теперь приложение, в котором естественно возникают операции ОБЪЕДИНИТЬ, MIN и УДАЛИТЬ.

Пример 4.12. В примере 4.1 мы изложили алгоритм для нахождения остовных деревьев наименьшей стоимости. Он формировал из узлов все бóльшие и бóльшие множества, такие, что элементы каждого из них соединялись ребрами, выбранными для остовного дерева наименьшей стоимости. Стратегия нахождения новых ребер для этого остовного дерева состояла в том, чтобы перебирать ребра (сначала наименьшей стоимости) и проверять, соединяют ли они какие-нибудь еще не соединенные узлы.

Рассмотрим другую стратегию. Для каждого множества узлов V_i будем хранить множество E_i всех нерассмотренных ребер, ин-

---

[1]) Здесь различие между «левым» и «правым» неважно; оно сделано ради сцепляемых очередей, которые обсуждаются в следующем разделе.

цидентных каким-то узлам в V_i. Если выбрать не рассмотренное ранее ребро e, инцидентное узлу из относительно малого множества V_i, то другой конец ребра e с большой вероятностью не будет лежать в V_i, и можно будет добавить e к остовному дереву. Если это нерассмотренное ребро e обладает наименьшей стоимостью среди всех ребер, инцидентных узлам из V_i, то можно показать, что включение его в остовное дерево приведет к остовному дереву наименьшей стоимости.

Для реализации этого алгоритма надо сначала образовать для каждого узла множество инцидентных ему ребер. Чтобы среди ребер, инцидентных узлам из V_i, найти нерассмотренное ребро наименьшей стоимости, применим MIN-оператор к множеству E_i нерассмотренных ребер для V_i. Затем удалим из E_i найденное так ребро e. Если окажется, что e имеет в V_i только один конец, а другой лежит в множестве V_i', отличном от V_i, то выполним операцию ОБЪЕДИНИТЬ для V_i и V_i' (например, используя структуру данных алгоритма 4.3), а также для E_i и E_i'.

В качестве структуры данных для представления каждого множества ребер E_i можно взять 2-3-дерево, каждый лист которого помечен ребром и его стоимостью. На множестве ребер нет никакого специального порядка. Каждому нелисту приписана наименьшая стоимость его потомков-листьев, обозначаемая НАИМЕНЬШИЙ[v].

Вначале для каждого узла образуем 2-3-дерево, содержащее все инцидентные ему ребра. Чтобы построить такое дерево, начнем с листьев. Затем добавим узлы высоты 1, так объединяя листья в группы по два или три, чтобы групп из двух листьев было не больше двух. Сделав это, вычислим для каждого узла высоты 1 наименьшую стоимость листа-потомка. Затем соберем узлы высоты 1 в группы по два или три и будем продолжать процесс до тех пор, пока на некотором уровне не образуется один узел, а именно корень. Время, затрачиваемое на такое построение дерева, пропорционально числу листьев. Реализация остальной части алгоритма теперь должна быть очевидна. Общее время работы составляет $O(e \log e)$, где e — число всех ребер. □

4.12. СЦЕПЛЯЕМЫЕ ОЧЕРЕДИ

В разд. 4.10 было показано, как на 2-3-дереве с n листьями выполнить каждую из операций ВСТАВИТЬ, УДАЛИТЬ, MIN и ПРИНАДЛЕЖАТЬ за время $O(\log n)$, если пользоваться значениями L и M. Сейчас покажем, как за время $O(\log n)$ выполнить каждую из операций СЦЕПИТЬ и РАСЦЕПИТЬ. Снова будем предполагать, что элементы расположены на листьях 2-3-дерева в порядке возрастания слева направо и для каждого узла v вычисляются значения $L[v]$ и $M[v]$.

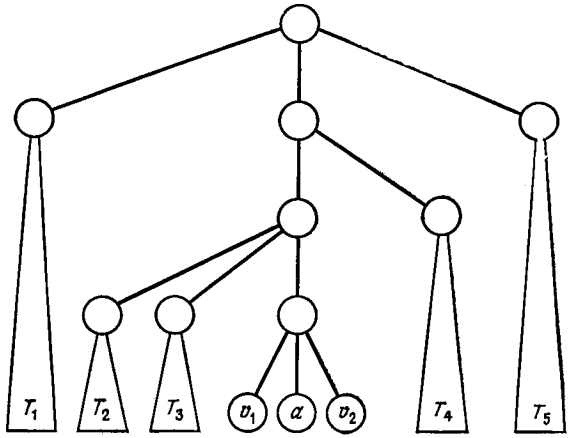


Рис. 4.33. Расцепление 2-3-дерева.

Операции СЦЕПИТЬ(S_1, S_2) на вход подаются такие две последовательности S_1 и S_2, что каждый элемент из S_1 меньше каждого элемента из S_2; на выход она выдает конкатенацию этих последовательностей, т. е. S_1S_2. Если S_1 и S_2 представлены соответственно 2-3-деревьями T_1 и T_2, то мы хотим соединить T_1 и T_2 в одно дерево T, листьями которого являются листья дерева T_1 в их первоначальном порядке и следующие за ними листья дерева T_2 в их первоначальном порядке. Это можно осуществить, вызвав процедуру ИМПЛАНТАЦИЯ(T_1, T_2), приведенную на рис. 4.32.

Наконец, рассмотрим операцию РАСЦЕПИТЬ. Напомним, что операция РАСЦЕПИТЬ(a, S) разбивает S на два множества $S_1 = \{b \mid b \leqslant a$ и $b \in S\}$ и $S_2 = \{b \mid b > a$ и $b \in S\}$. Для ее реализации определим процедуру ДЕЛЕНИЕ (a, T), которая расцепляет 2-3-дерево T на два такие 2-3-дерева T_1 и T_2, что метки всех листьев в T_1 не больше a, а метки всех листьев в T_2 больше a.

Метод можно неформально описать следующим образом. Дано 2-3-дерево T, содержащее элемент a. Идем по пути из корня в лист с меткой a. Этот путь разбивает наше дерево на поддеревья, корнями которых служат не сами узлы, лежащие на нем, а их сыновья. Это иллюстрирует рис. 4.33, где слева от пути находятся деревья T_1, T_2, T_3 и тривиальное дерево, состоящее из одного узла v_1, а справа — деревья T_4, T_5 и v_2.

Деревья слева от рассматриваемого пути и дерево, состоящее из одного узла a, соединяются с помощью только что описанного алгоритма конкатенации деревьев. Аналогично соединяются деревья, расположенные справа от пути. Необходимые детали даны в процедуре ДЕЛЕНИЕ (рис. 4.34).

```
procedure ДЕЛЕНИЕ(a, T):
begin
  на пути из узла КОРЕНЬ[T] к листу с меткой a удалить все
      узлы, кроме этого листа;
  comment В данный момент дерево T оказалось разделенным
   на два леса — левый, состоящий из всех деревьев, листья
   которых лежат слева от a, и из узла с меткой a, и правый,
   состоящий из всех деревьев, листья которых лежат справа от a;
  while в левом лесу более одного дерева do
    begin
      пусть T' и T'' — два самых правых дерева в левом лесу;
      ИМПЛАНТАЦИЯ(T', T'') [1])
    end;
  while в правом лесу более одного дерева do
    begin
      пусть T' и T'' — два самых левых дерева в правом лесу;
      ИМПЛАНТАЦИЯ(T', T'')
    end
end
```

[1]) Результат процедуры ИМПЛАНТАЦИЯ(T', T'') отнести к левому лесу. Аналогично при применении к деревьям из правого леса результат процедуры ИМПЛАНТАЦИЯ относится к правому лесу.

Рис. 4.34. Процедура для расцепления 2-3-дерева.

Теорема 4.7. *Процедура* ДЕЛЕНИЕ *разбивает* 2-3-*дерево* T *по листу* a *так, что все листья слева от* a *и сам лист* a *оказываются в одном* 2-3-*дереве, а все листья справа от* a *— в другом. Эта процедура занимает время* O(ВЫСОТА(T)). *Порядок листьев сохраняется.*

Доказательство. Из свойств процедуры ИМПЛАНТАЦИЯ вытекает, что деревья объединены правильно. Результат о времени работы получается из следующих соображений. Вначале число деревьев любой данной высоты не более 2, только деревьев высоты 0 может быть 3. Когда два дерева соединяются, получается дерево, высота которого не более чем на единицу больше наибольшей из высот двух исходных деревьев. В случае когда получается дерево высоты, на единицу большей высоты любого из исходных деревьев, его корень имеет степень 2. Таким образом, если соединяются три дерева высоты h, то в результате получается дерево

высоты не более $h+1$. Следовательно, на каждой стадии процесса число деревьев одинаковой высоты не превосходит 3.

Время, требуемое для соединения двух деревьев разной высоты, пропорционально разности их высот, а одинаковой высоты — постоянно. Поэтому объединение всех деревьев происходит за время, пропорциональное сумме числа деревьев и наибольшей из разностей высот любых двух деревьев. Таким образом, всего тратится времени порядка высоты исходного дерева. □

Заметим, что с помощью сцепляемой очереди последовательность S_2 можно вставить между парой элементов последовательности S_1 за время $O(\text{MAX}(\log|S_1|, \log|S_2|))$. Если $S_2=b_1, b_2, \ldots, b_n$, $S_1=a_1, a_2, \ldots, a_m$ и S_2 нужно вставить между элементами a_i и a_{i+1}, то можно применить операцию РАСЦЕПИТЬ(a_i, S_1) и разбить S_1 по элементу a_i на две последовательности $S_1'=a_1, \ldots, a_i$ и $S_1''=a_{i+1}, \ldots, a_m$. Затем применить операцию СЦЕПИТЬ(S_1', S_2), результатом которой будет последовательность $S_3=a_1, \ldots, a_i, b_1, \ldots, b_n$, и, наконец, операцию СЦЕПИТЬ(S_3, S_1''), дающую нужную последовательность.

4.13. РАЗБИЕНИЕ

Рассмотрим специальный тип расцепления, называемый *разбиением*. Задача разбиения множества возникает довольно часто, и решение, которое мы продемонстрируем здесь, поучительно само по себе. Пусть даны множество S и разбиение π множества S на непересекающиеся блоки $\{B_1, B_2, \ldots, B_p\}$. Кроме того, дана функция f, отображающая S на S.

Наша задача состоит в том, чтобы найти такое *грубейшее* (с наименьшим числом блоков) разбиение $\pi'=\{E_1, E_2, \ldots, E_q\}$ множества S, что

1) π' — подразбиение разбиения π (т. е. каждое множество E_i является подмножеством некоторого блока B_j),
2) если a и b принадлежат E_i, то $f(a)$ и $f(b)$ принадлежат E_j для некоторого j.

Будем называть π' *грубейшим разбиением множества* S, *совместимым с* π *и* f.

Очевидное решение состоит в повторном утончении блоков исходного разбиения следующим способом. Пусть B_i — какой-нибудь блок. Рассмотрим $f(a)$ для каждого a из B_i. Разобьем B_i так, чтобы два элемента a и b попадали в один блок тогда и только тогда, когда $f(a)$ и $f(b)$ оба принадлежат некоторому блоку B_j. Процесс повторяется до тех пор, пока уже нельзя будет проводить дальнейшие утончения. Этот метод дает алгоритм сложности $O(n^2)$, поскольку каждое утончение занимает время $O(n)$, а всего может

быть $O(n)$ утончений. Пример 4.13 показывает, что действительно может потребоваться квадратичное число шагов.

Пример 4.13. Пусть $S=\{1, 2, \ldots, n\}$ и $B_1=\{1, 2, \ldots, n-1\}$, $B_2=\{n\}$ — исходное разбиение. Пусть f — такая функция на S, что $f(i)=i+1$ для $1\leqslant i<n$ и $f(n)=n$. При первой итерации B_1 разбиваем на $\{1, 2, \ldots, n-2\}$ и $\{n-1\}$. Эта итерация занимает $n-1$ шагов, поскольку надо просмотреть каждый элемент в B_1. При следующей итерации разбиваем $\{1, 2, \ldots, n-2\}$ на $\{1, 2, \ldots, n-3\}$ и $\{n-2\}$. Продолжая в том же духе, выполняем всего $n-2$ итерации, причем i-я итерация занимает $n-i$ шагов. Следовательно, всего требуется

$$\sum_{i=1}^{n-2}(n-i)=\frac{n(n-1)}{2}-1$$

шагов. Окончательным разбиением будет $E_i=\{i\}$ для $1\leqslant i\leqslant n$. □

Недостаток этого метода состоит в том, что утончение блока может потребовать $O(n)$ шагов, даже если из него удаляется только один элемент. Сейчас мы опишем алгоритм разбиения, который для разделения блока на два подблока требует время, пропорциональное размеру меньшего подблока. Этот подход приводит к алгоритму сложности $O(n \log n)$.

Для каждого блока $B\subseteq S$ положим $f^{-1}(B)=\{b|f(b)\in B\}$. Вместо того чтобы разбивать B_i по значениям $f(a)$ для $a\in B_i$, разобьем относительно B_i те блоки B_j, которые содержат хотя бы один элемент из $f^{-1}(B_i)$ и один элемент не из $f^{-1}(B_i)$. Иными словами, каждый такой блок B_j разбивается на множества $\{b|b\in B_j$ и $f(b)\in B_i\}$ и $\{b|b\in B_j$ и $f(b)\notin B_i\}$.

Как только проведено разбиение относительно блока B_i, больше уже не нужно проводить разбиения относительно него, пока он сам не будет расщеплен. Если вначале $f(b)\in B_i$ для каждого элемента $b\in B_j$ и B_i расщеплен на B_i' и B_i'', то можно разбить B_j относительно B_i' или B_i'' и получить при этом один и тот же результат, поскольку $\{b|b\in B_j$ и $f(b)\in B_i'\}$ совпадает с $B_j-\{b|b\in B_j$ и $f(b)\in B_i''\}$.

Так как можно выбирать, по отношению к какому из блоков B_i' или B_i'' проводить разбиение, то мы разбиваем относительно того, для которого это сделать проще. Точнее мы выбираем меньшее из множеств $f^{-1}(B_i')$ и $f^{-1}(B_i'')$. Алгоритм приведен на рис. 4.35.

Алгоритм 4.5. Разбиение

Вход. Множество S из n элементов, разбиение $\pi=\{B[1], \ldots, B[p]\}$ и функция f: $S\to S$.

Выход. Грубейшее разбиение $\pi'=\{B[1], B[2], \ldots, B[q]\}$ множества S, совместимое с π и f.

```
    begin
1.    ОЖИДАНИЕ ← {1, 2, ..., p};
2.    q ← p;
3.    while ОЖИДАНИЕ не пусто do
        begin
4.        выбрать и удалить любое целое число i из множества
            ОЖИДАНИЕ;
5.        ОБРАЩЕНИЕ ← f⁻¹(B[i]);
6.        for j, для которых B[j] ∩ ОБРАЩЕНИЕ ≠ ∅ и
            B[j] ⊈ ОБРАЩЕНИЕ do
            begin
7.            q ← q + 1;
8.            создать новый блок B[q];
9.            B[q] ← B[j] ∩ ОБРАЩЕНИЕ;
10.           B[j] ← B[j] − B[q];
11.           if j ∈ ОЖИДАНИЕ then
                  добавить q в ОЖИДАНИЕ
              else
12.               if ||B[j]|| ≤ ||B[q]|| then
13.                   добавить j в ОЖИДАНИЕ
14.               else добавить q в ОЖИДАНИЕ
            end
        end
    end
```

Рис. 4.35. Алгоритм разбиения.

Метод. К π применяется программа, приведенная на рис. 4.35. В ней опущены некоторые детали, важные для ее реализации. Мы обсудим эти детали при анализе времени работы алгоритма. □

Анализ алгоритма 4.5 начнем с доказательства того, что он разбивает S нужным образом. Пусть π — разбиение множества S и f — отображение множества S на себя. Множество $T \subseteq S$ будем называть *безопасным* для π, если для любого блока $B \in \pi$ либо $B \subseteq f^{-1}(T)$, либо $B \cap f^{-1}(T) = \varnothing$. Например, на рис. 4.35 строки 9 и 10 гарантируют, что множество $B[i]$ безопасно для получающегося разбиения, поскольку если $B \cap f^{-1}(B[i]) \neq \varnothing$ для некоторого блока B, то либо $B \subseteq$ ОБРАЩЕНИЕ и тогда включение $B \subseteq f^{-1}(B[i])$ очевидно, либо в строках 9 и 10 B разбивается на два блока, один

из которых является подмножеством множества $f^{-1}(B[i])$, а другой с ним не пересекается.

В доказательство корректности алгоритма 4.5 входит доказательство того, что окончательное разбиение не оказывается слишком грубым. Иными словами, надо доказать следующее.

Лемма 4.7. *После окончания работы алгоритма* 4.5 *каждый блок B в результирующем разбиении π' безопасен для π'.*

Доказательство. На самом деле мы покажем, что для каждого блока $B[l]$

> после каждого выполнения цикла в строках 4—14 на рис. 4.35, если $l \notin$ ОЖИДАНИЕ и $l \leqslant q$, формируется такой список $q_1, q_2, \ldots, q_k$ (возможно, пустой), что $q_i \in$ ОЖИДАНИЕ, $1 \leqslant i \leqslant k$, и множество $B[l] \cup B[q_1] \cup \ldots \cup B[q_k]$ безопасно для разбиения, построенного в данный момент. (4.6)

Интуитивно это можно изложить так: когда число l удаляется из множества ОЖИДАНИЕ, строки 6—14 делают блок $B[l]$ безопасным для разбиения, которое получается после строки 14. $B[l]$ остается безопасным до тех пор, пока он не будет разбит. Когда $B[l]$ разбивается, индекс одного подблока, назовем его $B[q]$, помещается в ОЖИДАНИЕ. Другой подблок по-прежнему называется $B[l]$. Очевидно, что объединение этих двух подблоков, т. е. $B[l] \cup B[q]$, безопасно для рассматриваемого разбиения, поскольку оно совпадает со старым блоком $B[l]$. Дальнейшее разбиение образует такие блоки $B[q_1], \ldots, B[q_k]$, что $q_1, \ldots, q_k$ входят в ОЖИДАНИЕ, а объединение $R = B[l] \cup B[q_1] \cup \ldots \cup B[q_k]$ безопасно. Когда q_i при некотором i ($1 \leqslant i \leqslant k$) удаляется из множества ОЖИДАНИЕ, строки 6—14 снова делают $B[q_i]$ и $R - B[q_i]$ безопасными. Изложим теперь это формально. Докажем справедливость утверждения (4.6) для всех l индукцией по числу выполнений строк 4—14. Когда алгоритм заканчивает работу, множество ОЖИДАНИЕ пусто, и, значит, из (4.6) будет вытекать, что каждый блок окончательного разбиения π' безопасен для π'.

Для доказательства базиса возьмем 0 выполнений, тогда (4.6) тривиально, поскольку $l \in$ ОЖИДАНИЕ для всех $1 \leqslant l \leqslant q = p$.

Для проведения шага индукции предположим, что после выполнения строки 14 $l \notin$ ОЖИДАНИЕ. Если число l всегда ранее входило в ОЖИДАНИЕ, то оно имеет значение i, которое определялось в строке 4. Легко показать, что цикл в строках 6—14 делает $B[i]$ безопасным для разбиения, получающегося после выполнения строки 14. Это мы уже обосновали после определения понятия "безопасный".

Если число l не входило в ОЖИДАНИЕ во время предыдущего

выполнения цикла до строки 14, то по предположению индукции найдется такой список $q_1, \ldots, q_k$, что (4.6) было справедливо для l на предыдущем этапе. Кроме того, в строке 4 обязательно $i \neq l$.

Случай 1. i нет в $L=\{q_1, q_2, \ldots, q_k\}$. В строках 9, 10 могло произойти разбиение нескольких блоков. Для каждого такого блока $B[q_r]$, $1 \leqslant r \leqslant k$ (т. е. $j=q_r$), добавим в L индекс блока, образованного в строке 8. Благодаря строке 11 список L по-прежнему будет состоять только из индексов, входящих в ОЖИДАНИЕ. Если сам блок $B[l]$ не разбит, то $B[l]$ и множество блоков с индексами из L все еще образуют множество, безопасное для текущего разбиения, так что (4.6) удовлетворяется. Если же блок $B[l]$ разбит, надо также добавить в L индекс q, выбранный в строке 8, когда $j=l$. Следовательно, множество $B[l] \cup \bigcup_{r \in L} B[r]$ будет безопасно для текущего разбиения.

Случай 2. i находится в $L=\{q_1, q_2, \ldots, q_k\}$. Без потери общности будем считать, что $i=q_1$. Рассуждение проводится почти так же, как в случае 1, но q_1 в конце рассматриваемой итерации строк 4—14 может не входить в ОЖИДАНИЕ. Мы знаем, что каждый блок B текущего разбиения будет либо подмножеством множества $f^{-1}(B[q_1])$, либо не будет с ним пересекаться. Пусть $T=B[l] \cup \bigcup_{r \in L} B[r]$, где список L модифицирован, как в случае 1. Если $B \subseteq f^{-1}(B[q_1])$, то, разумеется, $B \cap f^{-1}(T)=\varnothing$. Если $B \cap f^{-1}(B[q_1])=\varnothing$, то аналогично случаю 1 можно доказать, что $B \cap f^{-1}(T)=\varnothing$ или $B \subseteq f^{-1}(T)$.

Наконец, когда алгоритм 4.5 заканчивает работу, множество ОЖИДАНИЕ должно быть пусто. Следовательно, из (4.6) вытекает, что для каждого l блок $B[l]$ безопасен для окончательного разбиения. □

Теорема 4.8. *Алгоритм* 4.5 *правильно вычисляет грубейшее разбиение множества* S, *совместимое с* π *и* f.

Доказательство. Лемма 4.7 показывает, что выход π' алгоритма 4.5 совместим с π и f. Надо доказать, что разбиение π' грубо, насколько возможно. Простая индукция по числу блоков, разбиваемых в строках 9, 10, показывает, что каждое такое разбиение, сделанное алгоритмом, необходимо для совместимости. Оставляем эту индукцию в качестве упражнения. □

Теперь мы должны детально исследовать реализацию алгоритма 4.5, чтобы показать, что время его работы составляет $O(n \log n)$, где $n=\|S\|$. Основным при оценке времени работы будет демонстрация того, что цикл в строках 6—14 можно выполнить за время, пропорциональное ||ОБРАЩЕНИЕ||. Наша первая задача — эффективно найти подходящее множество чисел j в строке 6. Нам понадобится такой массив БЛОК, что БЛОК[l] — индекс блока, содержащего l. Начальное состояние массива БЛОК можно по-

строить за $O(n)$ шагов, после строки 9 скорректировать его за время, не большее того, что тратится на формирование списка $B[q]$ в этой строке. Следовательно, поскольку нас интересует только порядок временной сложности, можно не учитывать время, затрачиваемое на работу с массивом БЛОК.

С помощью массива БЛОК легко построить список JСПИСОК чисел j, необходимых в строке 6, за $O(\|\text{ОБРАЩЕНИЕ}\|)$ шагов. Для каждого элемента a, входящего в ОБРАЩЕНИЕ, индекс блока, содержащего a, добавляем в JСПИСОК, если его там еще нет. Для каждого j хранится счетчик числа элементов, входящих в ОБРАЩЕНИЕ, которые входят также и в $B[j]$. Если этот счетчик достигнет величины $\|B[j]\|$, то $B[j] \subseteq$ ОБРАЩЕНИЕ и j удаляется из списка JСПИСОК.

Используя БЛОК, можно для каждого j, входящего в JСПИСОК, построить также список ПЕРЕСЕЧЕНИЕ[j], в который войдут все целые числа, принадлежащие обоим множествам $B[j]$ и ОБРАЩЕНИЕ. Чтобы быстро удалить элементы списка ПЕРЕСЕЧЕНИЕ[j] из $B[j]$ и добавить их в $B[q]$, надо поддерживать списки $B[l]$, $1 \leqslant l \leqslant q$, в дважды связанном виде, т. е. с указателями как к следующей, так и к предыдущей компонентам.

Строки 9 и 10 требуют $O(\|B[q]\|)$ шагов. Для данного выполнения **for**-цикла суммарное время, затрачиваемое на нахождение подходящих чисел j и на выполнение строк 7—10, составляет $O(\|\text{ОБРАЩЕНИЕ}\|)$. Кроме того, легко видеть, что тест в строках 12—14, если его выполнять должным образом, занимает $O(\|B[q]\|)$ времени, так что общее время есть $O(\|\text{ОБРАЩЕНИЕ}\|)$.

Осталось рассмотреть строку 11. Чтобы быстро сказать, входит ли j в ОЖИДАНИЕ, образуем другой массив В_ОЖИДАНИИ[j]. Начальное состояние В_ОЖИДАНИИ можно построить за $O(n)$ шагов и без труда корректировать его в строках 11—14. Таким образом, мы доказали следующую лемму.

Лемма 4.8. **for**-*цикл в строках* 6—14 *на рис.* 4.35 *можно реализовать за* $O(\|\text{ОБРАЩЕНИЕ}\|)$ *шагов.*

Доказательство. В силу изложенного выше. □

Теорема 4.9. *Алгоритм* 4.5 *можно реализовать за время* $O(n \log n)$.

Доказательство. Рассмотрим условия, при которых блок, содержащий целое число s и не представленный в множестве ОЖИДАНИЕ, может попасть туда. В строке 1 это может случиться лишь один раз. В строке 11 это произойти не может, даже если $s \in B[q]$, поскольку тогда s было бы в $B[j]$ еще раньше, а индекс j уже входил в ОЖИДАНИЕ. Если это случилось в строках 13 и 14, то число s находится в блоке, размер которого не больше половины размера того блока, куда оно входило в тот предыдущий момент, когда индекс множества, содержащего s, попал в ОЖИДА-

НИЕ. Отсюда можно заключить, что индекс множества, содержащего s, попадает в ОЖИДАНИЕ не больше $1+\log n$ раз. Следовательно, s не может быть в блоке i, который выбирался в строке 4 более чем $1+\log n$ раз.

Допустим, что на каждый элемент s из $B[i]$ каждый раз налагается штраф, пропорциональный $||f^{-1}(s)||$, так что сумма всех штрафов равна сложности выполнения цикла в строках 6—14. Тогда существует такая постоянная c, что за одно выполнение цикла элемент s штрафуется не более чем на $c||f^{-1}(s)||$. Как мы уже показали, элемент s не может попасть в выбираемый список $B[i]$ более чем $O(\log n)$ раз, так что суммарный штраф, налагаемый на него, составляет $O(||f^{-1}(s)||\log n)$. Так как сумма $\sum_{s \in S} ||f^{-1}(s)||$ должна равняться n, то общая сложность всех выполнений **for**-цикла есть $O(n \log n)$. Легко видеть, что сложность остальной части алгоритма 4.5 есть $O(n)$; теорема доказана. □

Алгоритм 4.5 имеет несколько приложений. Одно из важных приложений — минимизация числа состояний конечного автомата. Дан конечный автомат $M=(S, I, \delta, s_0, F)$ и требуется найти эквивалентный ему автомат M' с наименьшим числом состояний. Для каждого состояния s и входного символа a обозначим через $\delta(s, a)$ очередное состояние автомата. Вначале все состояния автомата M можно разбить на множество F допускающих состояний и множество $S-F$ недопускающих состояний. Задача минимизации числа состояний автомата M эквивалентна нахождению грубейшего разбиения π' множества S, совместимого с начальным разбиением $\{F, S-F\}$ и такого, что если состояния s и t находятся в одном блоке разбиения π', то состояния $\delta(s, a)$ и $\delta(t, a)$ также находятся в одном блоке автомата M' для каждого входного символа a.

Единственное отличие этой задачи от задачи, решаемой алгоритмом 4.5, состоит в том, что δ — отображение из $S \times I$ в S, а не из S в S. Однако можно трактовать δ как множество $\{\delta_{a_1}, \delta_{a_2}, \ldots, \delta_{a_m}\}$ функций на S, где δ_a — сужение функции δ на a.

Алгоритм 4.5 легко модифицировать так, что он справится с этой более общей задачей, помещая в множество ОЖИДАНИЕ пары (i, δ_a). Каждая пара (i, δ_a) состоит из индекса i некоторого блока разбиения и функции δ_a, по которой проводится разбиение. Вначале ОЖИДАНИЕ$=\{(i, \delta_a) | i=1$ или 2 и $a \in I\}$, поскольку начальное разбиение $\{F, S-F\}$ состоит из двух блоков. Всякий раз, когда блок $B[j]$ разбивается на $B[j]$ и $B[q]$, каждая допустимая функция δ_a спаривается с j и q. Остальные детали оставляем в качестве упражнения.

4.14. РЕЗЮМЕ

На рис. 4.36 приведены различные структуры данных, рассмотренные в этой главе, типы операций, которые можно совершать на их основе, и принимаемые нами соглашения о размере и природе того универсального множества, откуда брались элементы.

	Структура данных	Тип универсума	Допустимые операции	Время выполнения n операций над множествами размера n	
				среднее	в худшем случае
1	Таблица расстановки	Любое множество, на котором можно вычислять функцию расстановки	ПРИНАДЛЕЖАТЬ ВСТАВИТЬ УДАЛИТЬ	$O(n)$	$O(n^2)$
2	Дерево двоичного поиска	Любое упорядоченное множество	ПРИНАДЛЕЖАТЬ ВСТАВИТЬ УДАЛИТЬ MIN	$O(n \log n)$	$O(n^2)$
3	Древовидная структура из алгоритма 4.3	Множество целых чисел от 1 до n	ПРИНАДЛЕЖАТЬ ВСТАВИТЬ УДАЛИТЬ ОБЪЕДИНИТЬ НАЙТИ	не более $O(n\,G\,(n))$	не более $O(n\,G\,(n))$
4	2-3-деревья с неупорядоченным множеством листьев	Любое упорядоченное множество	ПРИНАДЛЕЖАТЬ ВСТАВИТЬ УДАЛИТЬ ОБЪЕДИНИТЬ НАЙТИ MIN	$O(n \log n)$	$O(n \log n)$
5	2-3-деревья с упорядоченным множеством листьев	Любое упорядоченное множество	ПРИНАДЛЕЖАТЬ ВСТАВИТЬ УДАЛИТЬ НАЙТИ РАСЦЕПИТЬ СЦЕПИТЬ MIN	$O(n \log n)$	$O(n \log n)$

Рис. 4.36. Сводка свойств структур данных.

УПРАЖНЕНИЯ

4.1. Приведите подмножество семи основных операций из разд. 4.1, достаточное для упорядочения любой последовательности из n элементов. Что можно сказать о сложности выполнения потока из n операций, выбранных из вашего подмножества?

4.2. Пусть элементами являются цепочки из букв и используется следующая функция расстановки для таблицы размера $m=5$: сложить "значения" букв, где A имеет значение 1, B — значение 2 и т. д.; разделить результат на 5 и взять остаток. Выпишите содержимое таблицы расстановки и списков, при условии что вставляются цепочки DASHER, DANCER, PRANCER, VIXEN, COMET, CUPID, DONNER, BLITZEN.

4.3. Вставьте восемь цепочек из упр. 4.2 в дерево двоичного поиска. Какую последовательность узлов надо посетить, чтобы проверить, принадлежит ли RUDOLPH этому множеству?

4.4. Покажите, что процедура ПОИСК (рис. 4.3) дает наименьшее возможное среднее время поиска, если все элементы универсального множества разыскиваются с равной вероятностью.

4.5. Найдите оптимальное дерево двоичного поиска для a, b, ... ,h, если эти элементы имеют вероятности соответственно 0,1; 0,2; 0,05; 0,1; 0,3; 0,05; 0,15; 0,05, а вероятность всех остальных равна 0.

****4.6.** Покажите, что в алгоритме 4.2 можно ограничить поиск числа m в строке 8 на рис. 4.9 областью позиций от $r_{i,j-1}$ до $r_{i+1,j}$ и все равно с гарантией находить максимум.

***4.7.** Используйте упр. 4.6 для такого изменения алгоритма 4.2, чтобы он работал время $O(n^2)$.

4.8. Закончите доказательство теоремы 4.2, показав, что алгоритм построения дерева на рис. 4.10 работает правильно.

4.9. Закончите доказательство теоремы 4.3.

***4.10.** Постройте интерфейс для перевода множества n внешних имен, лежащих между 1 и r, в множество внутренних имен, состоящее из целых чисел от 1 до n. Этот интерфейс должен переводить в обе стороны в предположении, что $r \gg n$.

(а) Разработайте интерфейс с хорошим поведением в среднем.
(б) Разработайте интерфейс с хорошим поведением в худшем случае.

*4.11. Найдите эффективную структуру данных для представления подмножества S целых чисел, лежащих между 1 и n. Мы хотим выполнять на множестве следующие операции:

1) выбирать и удалять из него какой-то один его элемент,
2) добавлять в него целое число i.

Предполагается, что механизм не должен добавлять в S целое число i, если оно уже есть там. Структура данных должна быть такой, чтобы время на выборку и удаление элемента и время на добавление элемента не зависели от $||S||$.

4.12. Найдите дерево, которое получается, когда алгоритм 4.3 выполняет последовательность операций ОБЪЕДИНИТЬ и НАЙТИ, порождаемую следующей программой. Допустим, что множество i вначале есть $\{i\}$ для $1 \leqslant i \leqslant 16$.

```
begin
    for i ← 1 step 2 until 15 do ОБЪЕДИНИТЬ (i, i+1, i);
    for i ← 1 step 4 until 13 do ОБЪЕДИНИТЬ (i, i+2, i);
    ОБЪЕДИНИТЬ (1, 5, 1);
    ОБЪЕДИНИТЬ (9, 13, 9);
    ОБЪЕДИНИТЬ (1, 9, 1);
    for i ← 1 step 4 until 13 do НАЙТИ (i)
end
```

4.13. Пусть σ — последовательность операций ОБЪЕДИНИТЬ и НАЙТИ, причем все операции ОБЪЕДИНИТЬ входят в σ перед операциями НАЙТИ. Докажите, что алгоритм 4.3 выполняет σ за время, пропорциональное $|\sigma|$.

4.14. *S_0-деревом* назовем дерево, состоящее из единственного узла. *S_i-дерево* для $i>0$ получается операцией, которая делает корень одного S_{i-1}-дерева сыном корня другого S_{i-1}-дерева. Докажите, что

(а) S_n-дерево имеет $\binom{n}{h}$ узлов высоты h,
(б) S_n-дерево можно получить из S_m-дерева, $m \leqslant n$, заменив каждый узел S_m-дерева на S_{n-m}-дерево так, что сыновья этого узла становятся сыновьями корня подставляемого S_{n-m}-дерева,
(в) S_n-дерево содержит узел с n сыновьями; они служат корнями S_0-, S_1-, . . ., S_{n-1}-деревьев.

4.15. Рассмотрим алгоритм объединения непересекающихся множеств, который делает корень дерева с меньшим числом узлов (в случае одинакового количества узлов выбор дерева произволен)

сыном корня большего дерева, но не использует сжатие путей. Докажите, что верхнюю границу $O(n \log n)$ нельзя улучшить. Иными словами покажите, что для некоторой постоянной c и произвольно больших значений n существуют последовательности операций ОБЪЕДИНИТЬ и НАЙТИ, требующие $cn \log n$ шагов.

**4.16. Пусть $T(n)$ — временная сложность (в худшем случае) выполнения n операций ОБЪЕДИНИТЬ и НАЙТИ при условии, что применяется древовидная структура из разд. 4.7 и сжатие путей для операций НАЙТИ, но операция ОБЪЕДИНИТЬ(A, B, C) выполняется так: корень дерева A становится сыном корня дерева B независимо от того, какое множество больше. Покажите, что $T(n) \geqslant k_1 n \log n$ для некоторой постоянной $k_1 > 0$.

**4.17. Покажите, что $T(n) \leqslant k_2 n \log n$ для некоторой постоянной k_2, где $T(n)$ обозначает то же, что и в упр. 4.16.

**4.18. Покажите, как можно выполнить последовательность из n операций ОБЪЕДИНИТЬ, НАЙТИ, ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ, УДАЛИТЬ на множестве целых чисел $1, 2, \ldots, n$ за время $O(nG(n))$. При этом считайте, что УДАЛИТЬ(i, S) делает i членом нового множества $\{i\}$, которому надо присвоить (произвольное) имя. Это множество можно впоследствии слить с другим. Кроме того, предположите, что никакой элемент не принадлежит более чем одному множеству.

4.19. Обобщите MIN-задачу для свободного режима так, чтобы можно было выполнять операцию MIN вида MIN(i), которая находит все целые числа, меньшие i, вставленные до нее и еще не найденные предыдущей операцией MIN.

4.20. Разработайте полную структуру данных для MIN-задачи в свободном режиме, включающую представление деревьев массивами, и напишите программу, в которой употребляются массивы, а не команды высокого уровня, используемые алгоритмом объединения непересекающихся множеств.

4.21. Обобщите MIN-задачу для свободного режима следующим образом. Пусть T — дерево с n узлами. Каждому узлу ставится в соответствие целое число от 1 до n. Некоторым узлам приписаны операции ИЗВЛЕЧЬ_MIN. Пройдите дерево в обратном порядке. Встретив операцию ИЗВЛЕЧЬ_MIN в узле v, найдите в дереве с корнем v наименьшее целое число (исключая число в самом узле v) среди тех чисел, которые не были удалены ранее, и удалите его. Укажите алгоритм для этого процесса, работающий в свободном режиме и имеющий сложность $O(nG(n))$.

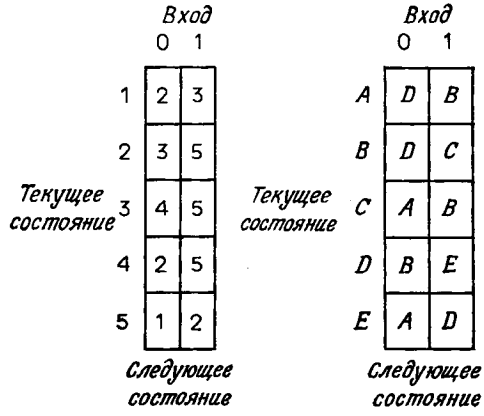
**4.22. Разработайте алгоритм сложности $O(n \log \log n)$ для задачи ОБЪЕДИНИТЬ — НАЙТИ, пользуясь структурой данных, со-

стоящей из дерева, каждый лист которого отстоит от корня на расстояние 2. Ограничьте степень корня числом между 1 и $n/\log n$, а степень каждого его сына — числом между 1 и $\log n$. Как модифицировать этот алгоритм, чтобы он делал не более $O(n \log\log\log n)$ шагов? Какую наилучшую границу на время можете вы получить, обобщая этот метод?

4.23. Покажите, как можно хранить в таблице символов смещения от начала отсчета, упомянутые в приложении 2 разд. 4.8. *Указание*: Примените технику, использованную для определения глубины.

4.24. Напишите программу для операций ОБЪЕДИНИТЬ и НАЙТИ, осуществляющую вычисления с весом, как в приложении 2 разд. 4.8.

4.25. С помощью алгоритма на рис. 4.25 выясните эквивалентность конечных автоматов



Начальными состояниями являются 1 и A соответственно, а множествами заключительных состояний — множества $\{5\}$ и $\{C, E\}$ соответственно.

4.26. Напишите полные программы для выполнения операций на 2-3-деревьях:

(а) УДАЛИТЬ,
(б) ОБЪЕДИНИТЬ,
(в) ПРИНАДЛЕЖАТЬ (в предположении, что множество листьев упорядочено и каждый узел помечен номером самого высокого листа среди всех листьев, с которыми он соединен),
(г) РАСЦЕПИТЬ (в предположении, что задан лист, по которому производится расцепление, и множество листьев упорядочено).

4.27. Напишите полную программу для вставки нового узла в 2-3-дерево, предполагая, что множество листьев упорядочено.

4.28. Напишите программы для основных операций ПРИНАДЛЕЖАТЬ, ВСТАВИТЬ, УДАЛИТЬ, MIN, ОБЪЕДИНИТЬ и НАЙТИ, используя 2-3-деревья с метками НАИМЕНЬШИЙ из разд. 4.11. Универсальным множеством считайте $\{1, 2, \ldots, n\}$.

4.29. Рассмотрите реализацию сливаемого дерева в виде 2-3-дерева (ВСТАВИТЬ, УДАЛИТЬ, ОБЪЕДИНИТЬ, MIN). Считайте при этом, что универсум, из которого берутся элементы, велик. Опишите, как реализовать НАЙТИ за $O(\log n)$ шагов на одну операцию, где n — общее число элементов во всех соединяемых деревьях.

Определение. АВЛ-*деревом* [1]) называют такое дерево двоичного поиска, что для каждого узла v высоты его левого и правого поддеревьев отличаются не более чем на единицу. Если какого-то поддерева нет, его "высота" считается равной —1.

Пример 4.14. Дерево на рис. 4.37 не является АВЛ-деревом, потому что высота левого поддерева узла * равна 2, а правого — 0. Во всех остальных узлах АВЛ-условие выполняется. □

4.30. Покажите, что АВЛ-дерево высоты h содержит не более $2^{h+1}-1$ и не менее

$$\frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^h+\frac{5-2\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^h-1$$

узлов.

***4.31.** Пусть T — дерево двоичного поиска с n узлами, обладающее АВЛ-свойством. Напишите алгоритм сложности $O(\log n)$

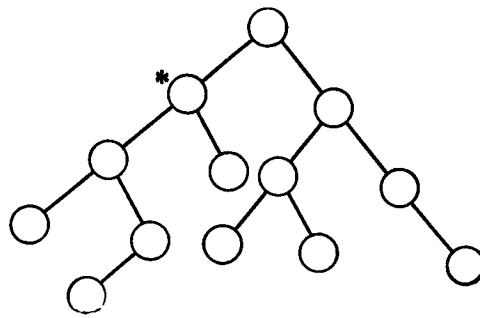


Рис. 4.37. Не АВЛ-дерево.

[1]) В честь его авторов Г. М. Адельсона-Вельского и Е. М. Ландиса; см. их работу [1962].

для операций ВСТАВИТЬ и УДАЛИТЬ, которые сохраняют АВЛ-свойство. Можете считать, что высоту каждого узла можно найти в нем самом и что информация о высотах корректируется автоматически.

*4.32. Напишите алгоритмы для расцепления АВЛ-дерева и сцепления двух АВЛ-деревьев. Эти алгоритмы должны расходовать время, пропорциональное высотам деревьев.

*4.33. Используйте АВЛ-дерево в качестве базиса алгоритма, выполняющего операции MIN, ОБЪЕДИНИТЬ и УДАЛИТЬ над множествами целых чисел от 1 до n и затрачивающего $O(\log n)$ шагов на операцию.

Определение. *Балансом* узла v двоичного дерева называется число $(1+L)/(2+L+R)$, где L и R — числа узлов в левом и правом поддеревьях узла v. Дерево двоичного поиска называется *α-сбалансированным*, если баланс каждого узла заключен между α и $1-\alpha$.

Пример 4.15. Баланс узла, отмеченного знаком * на рис. 4.37, равен $^5/_7$. Ни один другой узел не имеет баланса, столь сильно уклоняющегося от $^1/_2$, так что дерево на рис. 4.37 $^2/_7$-сбалансировано. □

*4.34. Укажите верхнюю и нижнюю границы числа узлов в α-сбалансированном дереве высоты h.

**4.35. Повторите упр. 4.31—4.33 для деревьев с балансом α, где $\alpha \leqslant {}^1/_3$.

*4.36. Сможете ли вы сделать упр. 4.31—4.33, если $\alpha > {}^1/_3$?

4.37. Разработайте понятие сбалансированного дерева с данными на листьях, в котором балансировка достигается за счет того, что разность высот поддеревьев поддерживается в пределах фиксированной постоянной.

*4.38. Напишите алгоритм сложности $O(nG(n))$, который по данному дереву с n узлами и списку из n пар узлов находил бы для каждой пары (v, w) ближайшего общего предка узлов v и w.

*4.39. Длина *внешних путей* двоичного дерева определяется как сумма глубин всех его листьев, длина *внутренних путей* — как сумма глубин всех его узлов. Каково соотношение между длинами внешних и внутренних путей, если у каждого узла либо два сына, либо ни одного?

*4.40. Разработайте структуру данных для реализации очередей, которая позволяла бы выполнять в префиксном режиме сле-

дующие операции:

(а) ВПИСАТЬ (i, A), т. е. добавить целое число i к очереди A. Допускаются различные вхождения одного и того же числа.

(б) ВЫПИСАТЬ (A), т. е. извлечь из A тот элемент, который находится там дольше всех.

(в) СЛИТЬ (A, B, C), т. е. объединить очереди A и B в новую очередь C. Предполагается, что элементы находятся в объединенной очереди так долго, как они находились в соответствующей очереди A или B. Заметьте, что одно и то же число может несколько раз входить в A и B и каждое вхождение следует рассматривать как отдельный элемент.

Сколько времени нужно для выполнения последовательности из n операций?

*4.41. Разработайте структуру данных для реализации операций упр. 4.40 в свободном режиме. Сколько времени нужно для выполнения n таких операций в свободном режиме?

*4.42. Пусть начальное разбиение $\pi=\{B_1, \ldots, B_q\}$ в алгоритме 4.5 обладает тем свойством, что для каждого $1\leqslant i\leqslant q$ справедливо включение $f^{-1}(B_i)\subseteq B_j$ при некотором j. Покажите, что в строке 6 на рис. 4.35 выбирается не более одного j. Уменьшает ли наше допущение время работы алгоритма 4.5?

Проблема для исследования

4.43. Со скоростью выполнения последовательности из n операций, выбранных из тех семи, что приведены в разд. 4.1 (или других основных операций того же типа), связано много нерешенных вопросов. Если элементы берутся из произвольного множества и единственный путь получения информации о них — это попарное сравнение, то любое множество основных операций, с помощью которых можно сортировать, потребует время $O(n \log n)$. Но если универсум представляет собой множество целых чисел $\{1,2, \ldots, n\}$ (или даже $\{1, 2, \ldots, n^k\}$ для фиксированного k), то трудно привести соображения за то, что для сортировки потребуется более $O(n)$ времени, если только основных операций достаточно для нее. Таким образом, есть возможность улучшить среднее время или время в худшем случае, приведенные на рис. 4.36. Напротив, можно ли для некоторого подмножества (или даже всего множества) основных операций на целых числах от 1 до n получить нижние оценки, лучшие чем $O(n)$?

Замечания по литературе

Информацию о разнообразии методов расстановки можно почерпнуть в книгах Морриса [1968], Ахо, Ульмана [1973] и Кнута [1973а] [1]). Последняя содержит

[1]) Кнут [1973а, § 6.4] дает краткую историю возникновения и развития этих методов, а также интересное объяснение употребляемого в литературе на английском языке термина «hashing». По свидетельству Кнута, это слово «в разных частях света уже стало общепринятым жаргоном». В нашу литературу также

также информацию о деревьях двоичного поиска. Алгоритм 4.2 для построения статического дерева двоичного поиска заимствован у Гилберта, Мура [1959]. Алгоритм сложности $O(n^2)$ для той же задачи можно найти у Кнута [1971], где содержится также и решение упр. 4.6. Ху, Таккер [1971] показали, что время $O(n^2)$ и память $O(n)$ достаточны, чтобы построить оптимальное дерево двоичного поиска, в котором данные появляются только на листьях. Кнут [1973а] дал реализацию этого алгоритма за время $O(n \log n)$.

Рейнгольд [1972] приводит оптимальные алгоритмы для многих основных преобразований множеств, таких, как объединение и пересечение. Алгоритм 4.3 для задачи ОБЪЕДИНИТЬ — НАЙТИ был, по-видимому, впервые применен Мак-Илроем и Моррисом. Кнут [1969] приписывает сжатие путей Триттеру. Теорема 4.4 взята у Хопкрофта, Ульмана [1974], а теорема 4.5 — у Тарьяна [1974]. Приложение к приравниванию идентификаторов и вычислению смещения, обсуждавшиеся в разд. 4.8, заимствованы у Галлера, Фишера [1964]. Приложение 3 об эквивалентности конечных автоматов взято из статьи Хопкрофта, Карпа [1971]. Упр. 4.16 и 4.17, касающиеся сложности алгоритма 4.3 без сжатия путей, взяты у Фишера [1972] и Патерсона [1973] соответственно.

АВЛ-деревья даны по Адельсону-Вельскому, Ландису [1962], а ответы на упр. 4.31 и 4.32 можно найти у Крейна [1972]. Понятие α-сбалансированного дерева изложено по Нивергельту, Рейнгольду [1973]. Читатель может обратиться к этой работе по поводу упр. 4.34—4.37. Дальнейшие применения 2-3-деревьев можно найти в работе Ульмана [1974].

Упр. 4.22 представляет собой раннее решение задачи ОБЪЕДИНИТЬ — НАЙТИ; оно приведено Стирнзом, Розенкранцем [1969]. Упр. 4.38 содержится у Ахо, Хопкрофта, Ульмана [1974].

Различие между префиксным и свободным режимами работы алгоритмов ввели Хартманис, Льюис, Стирнз [1965]. Рабин [1963] изучил важный частный случай префиксных вычислений, называемый *вычислением в реальное время*.

Алгоритм разбиения и приложение к минимизации числа состояний конечного автомата взяты из статьи Хопкрофта [1971].

проник уже этот жаргон: хеширование, хеш-таблицы, хеш-функции. В данной книге мы вслед за переводчиками двухтомного труда Ахо, Ульмана [1972, 1973] называем эти методы *методами расстановки*.— *Прим. ред.*

5 АЛГОРИТМЫ НА ГРАФАХ

Многие задачи теоретического и прикладного характера можно сформулировать в терминах неориентированных или ориентированных графов. В этой главе мы обсудим некоторые из основных задач на графах, решения которых полиномиально зависят (в смысле времени выполнения) от числа узлов и, следовательно, ребер графа. В основном мы будем заниматься задачами, в которых речь идет о связности графа. Сюда входят алгоритмы для нахождения остовных деревьев, двусвязных компонент, сильно связных компонент и путей между узлами. В гл. 10 мы изучим более сложные задачи о графах.

5.1. ОСТОВНОЕ ДЕРЕВО НАИМЕНЬШЕЙ СТОИМОСТИ

Пусть $G=(V, E)$ — связный неориентированный граф, для которого задана функция стоимости, отображающая ребра в вещественные числа. Напомним, что остовным деревом для данного графа называется неориентированное дерево, содержащее все узлы графа. Стоимость остовного дерева определяется как сумма стоимостей его ребер. Наша цель — найти для G остовное дерево наименьшей стоимости. Мы увидим, что остовное дерево наименьшей стоимости для графа с e ребрами можно найти за время $O(e \log e)$ в общем случае и $O(e)$, если e достаточно велико по сравнению с числом узлов (см. упр. 5.3). Многие алгоритмы нахождения остовного дерева основаны на следующих двух леммах.

Лемма 5.1. *Пусть $G=(V, E)$ — связный неориентированный граф и $S=(V, T)$ — остовное дерево для него. Тогда*

(а) *для любых двух узлов v_1 и v_2 из V путь между v_1 и v_2 в S единствен и*

(б) *если к S добавить ребро из $E-T$, то возникнет ровно один цикл.*

Доказательство. Утверждение (а) тривиально, ибо если бы было более одного пути, то в S был бы цикл.

Утверждение (б) тоже тривиально, ибо между концами добавляемого ребра уже был один путь. □

Лемма 5.2. *Пусть $G=(V, E)$ — связный неориентированный граф и c — функция стоимости, заданная на его ребрах. Пусть $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$ — произвольный остовный лес для G, $k>1$, и $T=\bigcup_{i=1}^{k} T_i$. Допустим, что $e=(v, w)$ — ребро наименьшей стоимости в $E-T$, причем $v \in V_1$ и $w \notin V_1$. Тогда найдется остовное дерево для G, содержащее $T \cup \{e\}$, стоимость которого не больше стоимости любого остовного дерева для G, содержащего T.*

Доказательство. Допустим противное и обозначим через $S'=(V, T')$ остовное дерево для G, содержащее T и не содержащее e, стоимость которого меньше стоимости любого остовного дерева для G, содержащего $T \cup \{e\}$.

По лемме 5.1(б) добавление e к S' образует цикл (см. рис. 5.1). Этот цикл должен содержать такое ребро $e'=(v', w')$, отличное от e, что $v' \in V_1$ и $w' \notin V_1$. По условию $c(e) \leqslant c(e')$.

Рассмотрим граф S, образованный добавлением e к S' и удалением e' из S'. В S нет циклов, поскольку единственный цикл был разорван удалением ребра e'. Кроме того, все узлы в V все еще соединены, так как есть путь между v' и w' в S. Следовательно, S — остовное дерево для G. Так как $c(e) \leqslant c(e')$, то стоимость дерева S не больше стоимости дерева S'. Но S содержит и T, и e, что противоречит минимальности дерева S'. □

Опишем алгоритм нахождения остовного дерева наименьшей стоимости для неориентированного графа $G=(V, E)$. Этот алгоритм по существу тот же, что и в примере 4.1. Он работает с набором VS непересекающихся множеств узлов. Каждое множество W из VS представляет связное множество узлов, образующее остовное дерево в остовном лесу, представленном набором VS. Ребра выби-

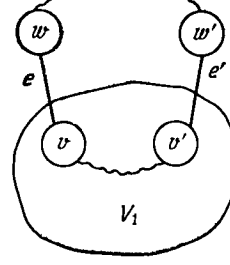


Рис. 5.1. Цикл в графе G.

```
    begin
1.  T ← ∅;
2.  VS ← ∅;
3.  построить очередь с приоритетами Q, содержащую все
        ребра из E;
4.  for v ∈ V do добавить {v} к VS;
5.  while ‖VS‖ > 1 do
      begin
6.      выбрать в Q ребро (v, w) наименьшей стоимости;
7.      удалить (v, w) из Q;
8.      if v и w принадлежат различным множествам W1 и
          W2 из VS then
          begin
9.          заменить W1 и W2 на W1 ∪ W2 в VS;
10.         добавить (v, w) к T
          end
      end
    end
```

Рис. 5.2. Алгоритм построения остовного дерева наименьшей стоимости.

раются из E в порядке возрастания стоимости. Ребра (v, w) рассматриваются по очереди. Если v и w принадлежат одному и тому же множеству из VS, то ребро (v, w) исключается из рассмотрения. Если v и w принадлежат разным множествам W_1 и W_2 (это означает, что W_1 и W_2 еще не соединены), то сливаем их в одно множество и добавляем (v, w) к множеству T ребер, входящих в окончательное остовное дерево. Здесь можно воспользоваться алгоритмом объединения непересекающихся множеств, описанным в разд. 4.7. Применив лемму 5.2 и проведя несложную индукцию по числу выбранных ребер, заключаем, что это ребро содержится по крайней мере в одном остовном дереве наименьшей стоимости для G.

Алгоритм 5.1. Остовное дерево наименьшей стоимости (алгоритм Крускала)

Вход. Неориентированный граф $G=(V, E)$ с функцией стоимости c, заданной на его ребрах.

Выход. Остовное дерево $S=(V, T)$ наименьшей стоимости для G.

Метод. Программа приведена на рис. 5.2. □

Пример 5.1. Рассмотрим неориентированный граф на рис. 5.3. Перечислим его ребра в порядке возрастания их стоимостей:

Ребро	Стоимость	Ребро	Стоимость
(v_1, v_7)	1	(v_4, v_5)	17
(v_3, v_4)	3	(v_1, v_2)	20
(v_2, v_7)	4	(v_1, v_6)	23
(v_3, v_7)	9	(v_5, v_7)	25
(v_2, v_3)	15	(v_5, v_6)	28
(v_4, v_7)	16	(v_6, v_7)	36

Разумеется, на самом деле на шаге 3 алгоритма 5.1 ребра не сортируются, а хранятся в виде сортирующего дерева, 2-3-дерева или какой-нибудь другой приемлемой структуры данных, пока они не потребуются. Сортирующее дерево из разд. 3.4 фактически дает идеальный способ реализации очереди с приоритетами. Повторное обращение в строке 6 с целью найти ребро наименьшей стоимости является основной операцией с Сортдеревом. Кроме того, число ребер, выбираемых в строке 6 для построения остовного дерева, часто меньше $\|E\|$. В таких ситуациях мы экономим время, поскольку никогда не упорядочиваем E полностью.

Вначале каждое ребро находится в одноэлементном множестве, входящем в VS и состоящем из самого этого ребра. Наименьшую стоимость имеет ребро (v_1, v_7), так что оно добавляется к дереву и множества $\{v_1\}$ и $\{v_7\}$ в VS сливаются.

Затем рассматривается ребро (v_3, v_4). Так как v_3 и v_4 принадлежат разным множествам из VS, добавляем (v_3, v_4) к дереву и сливаем $\{v_3\}$ и $\{v_4\}$. Далее добавляем $\{v_2, v_7\}$ и сливаем $\{v_2\}$ с $\{v_1, v_7\}$. Добавляем также четвертое ребро $\{v_3, v_7\}$ и сливаем $\{v_1, v_2, v_7\}$ с $\{v_3, v_4\}$.

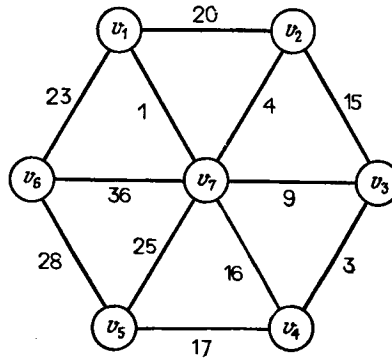


Рис. 5.3. Неориентированный граф с указанными стоимостями его ребер.

Ребро	Действие	Множества в VS (связные подграфы)
(v_1, v_7)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_4)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_2, v_7)	Добавить	$\{v_1, v_2, v_7\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_7)	Добавить	$\{v_1, v_2, v_3, v_4, v_7\}, \{v_5\}, \{v_6\}$
(v_2, v_3)	Отвергнуть	
(v_4, v_7)	Отвергнуть	
(v_4, v_5)	Добавить	$\{v_1, v_2, v_3, v_4, v_5, v_7\}, \{v_6\}$
(v_1, v_2)	Отвергнуть	
(v_1, v_6)	Добавить	$\{v_1, \ldots, v_7\}$

Рис. 5.4. Последовательность шагов построения остовного дерева.

Затем рассматривается ребро (v_2, v_3). Оба узла v_2 и v_3 принадлежат одному и тому же множеству $\{v_1, v_2, v_3, v_4, v_7\}$. Следовательно, из v_2 в v_3 идет путь из ребер, уже вошедших в остовное дерево, и потому (v_2, v_3) не добавляется. Вся последовательность шагов приведена на рис. 5.4. Остовное дерево, получающееся в результате, изображено на рис. 5.5. □

Теорема 5.1. *Алгоритм* 5.1 *находит остовное дерево наименьшей стоимости для связного графа* G. *Если в цикле, описанном строками* 5—10, *рассматривается* d *ребер, то затрачивается время* $O(d \log e)$, *где* $e = \|E\|$. *Следовательно, алгоритм* 5.1 *занимает не более* $O(e \log e)$ *времени.*

Доказательство. Корректность алгоритма вытекает непосредственно из леммы 5.2 и того факта, что строки 8 и 9 сохраняют узлы в том же самом множестве тогда и только тогда, когда

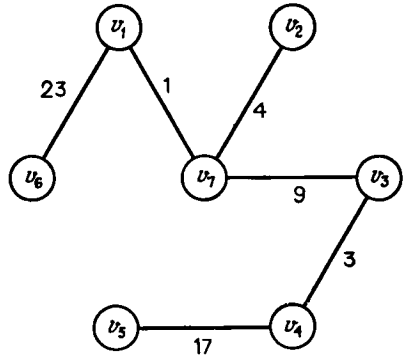


Рис. 5.5. Остовное дерево наименьшей стоимости.

они принадлежат одному и тому же дереву остовного леса, представленного набором VS.

Для оценки времени допустим, что требуется d итераций цикла в строках 5—10. Нахождение ребра наименьшей стоимости в Q (строка 6) занимает $O(\log e)$ шагов, если Q реализуется очередью с приоритетами. Общее время нахождения всех множеств W_1 и W_2, содержащих v и w (строка 8), и замены их на их объединение (строка 9) не превосходит $O(eG(e))$ [1]), если применяется быстрый алгоритм объединения непересекающихся множеств. Остальная часть цикла, очевидно, занимает постоянное время, не зависящее от размера G. Засылка начальных данных в Q занимает время $O(e)$, а в VS — время $O(n)$, где n — число узлов в V. □

5.2. МЕТОД ПОИСКА В ГЛУБИНУ

Рассмотрим прохождение узлов неориентированного графа в следующем порядке. Выбираем и "посещаем" некоторый начальный узел v. Затем выбираем произвольное ребро (v, w), инцидентное v, и посещаем w. Вообще пусть x — последний посещенный узел. Для продолжения процесса выбираем какое-нибудь не рассмотренное еще ребро (x, y), инцидентное x. Если узел y уже посещался, ищем другое новое ребро, инцидентное x. Если y раньше не посещался, идем в y и заново начинаем поиск от узла y. Пройдя все пути, начинающиеся в узле y, возвращаемся в x, т. е. в тот узел, из которого впервые был достигнут узел y. Затем продолжаем выбор нерассмотренных ребер, инцидентных узлу x, до тех пор, пока не будет исчерпан список этих ребер. Этот метод обхода узлов неориентированного графа называется *поиском в глубину*, поскольку процесс поиска идет в направлении вперед (вглубь) до тех пор, пока это возможно.

Поиск в глубину можно также осуществлять и на ориентированном графе. Если граф ориентирован, то, находясь в узле x, мы выбираем ребра (x, y), только выходящие из x. Исследовав все ребра, выходящие из y, мы возвращаемся в x даже тогда, когда в y входят другие ребра, еще не рассмотренные.

Если поиск в глубину осуществляется на связном неориентированном графе, то, как легко показать, посещается каждый узел и исследуется каждое ребро. Если граф несвязен, то отыскивается его связная компонента. Закончив работу с ней, выбирают в качестве нового начального узла узел, который еще не посещался, и начинают новый поиск.

Поиск в глубину на неориентированном графе $G=(V, E)$ разбивает ребра, составляющие E, на два множества T и B. Ребро (v, w) помещается в множество T, если узел w не посещался до того момента, когда мы, рассматривая ребро (v, w), оказались в узле v.

[1]) Здесь G — функция, введенная в разд. 4.7. — *Прим. ред.*

Ребро	Действие	Множества в VS (связные подграфы)
(v_1, v_7)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_4)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_2, v_7)	Добавить	$\{v_1, v_2, v_7\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_7)	Добавить	$\{v_1, v_2, v_3, v_4, v_7\}, \{v_5\}, \{v_6\}$
(v_2, v_3)	Отвергнуть	
(v_4, v_7)	Отвергнуть	
(v_4, v_5)	Добавить	$\{v_1, v_2, v_3, v_4, v_5, v_7\}, \{v_6\}$
(v_1, v_2)	Отвергнуть	
(v_1, v_6)	Добавить	$\{v_1, \ldots, v_7\}$

Рис. 5.4. Последовательность шагов построения остовного дерева.

Затем рассматривается ребро (v_2, v_3). Оба узла v_2 и v_3 принадлежат одному и тому же множеству $\{v_1, v_2, v_3, v_4, v_7\}$. Следовательно, из v_2 в v_3 идет путь из ребер, уже вошедших в остовное дерево, и потому (v_2, v_3) не добавляется. Вся последовательность шагов приведена на рис. 5.4. Остовное дерево, получающееся в результате, изображено на рис. 5.5. □

Теорема 5.1. *Алгоритм* 5.1 *находит остовное дерево наименьшей стоимости для связного графа* G. *Если в цикле, описанном строками* 5—10, *рассматривается* d *ребер, то затрачивается время* $O(d \log e)$, *где* $e = \|E\|$. *Следовательно, алгоритм* 5.1 *занимает не более* $O(e \log e)$ *времени.*

Доказательство. Корректность алгоритма вытекает непосредственно из леммы 5.2 и того факта, что строки 8 и 9 сохраняют узлы в том же самом множестве тогда и только тогда, когда

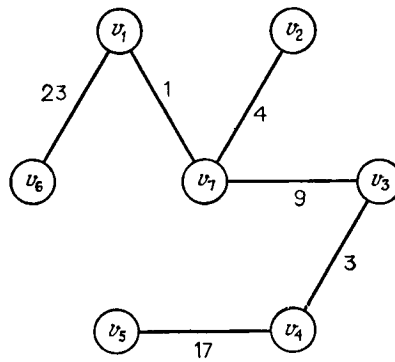


Рис. 5.5. Остовное дерево наименьшей стоимости.

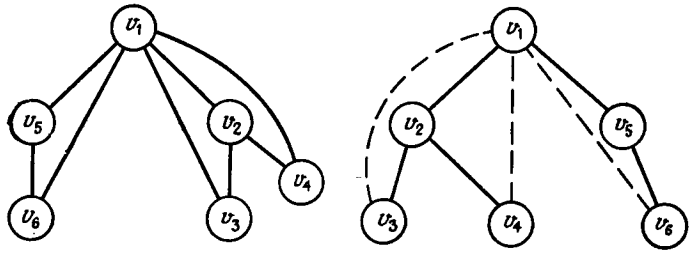
димся в v, а узел w помечен как "старый", ибо w может оказаться отцом узла v. □

Пример 5.2. Условимся изображать ребра, входящие в T, сплошными линиями, а ребра из B — штриховыми. Кроме того, корень дерева (начальный узел, выбираемый в строке 8) будем рисовать наверху, а сыновей каждого узла будем располагать слева направо в том порядке, в котором их ребра добавлялись в строке 4 процедуры ПОИСК. На рис. 5.7,*б* показано одно возможнсе разбиение графа, изображенного на рис. 5.7,*а*, на множества древесных и обратных ребер, построенное в процессе поиска в глубину.

Вначале все узлы помечены как "новые". Допустим, что в строке 8 выбирается v_1. Выполняя ПОИСК(v_1), можно в строке 2 взять $w=v_2$. Так как узел v_2 помечен как "новый", добавляем (v_1, v_2) к T и вызываем ПОИСК(v_2). Теперь можно было бы выбрать узел v_1 из $L[v_2]$, но он уже помечен как "старый". Тогда выберем $w=v_3$. Так как v_3 — "новый" узел, добавляем (v_2, v_3) к T и вызываем ПОИСК(v_3). Каждый узел, смежный с v_3, является теперь "старым", так что снова вызываем ПОИСК(v_2).

Выполняя процедуру ПОИСК(v_2), находим ребро (v_2, v_4), добавляем его к T и вызываем ПОИСК(v_4). Заметим, что на рис. 5.7,*б* узел v_4 расположен справа от найденного ранее сына v_3 узла v_2. Никакие "новые" узлы не смежны с v_4, так что опять вызываем ПОИСК(v_2). На этот раз мы не обнаруживаем "новых" узлов, смежных с v_2, и потому вызываем ПОИСК(v_1). Далее ПОИСК(v_1) находит v_5, а ПОИСК(v_5) находит v_6. Все узлы оказываются на дереве, и они помечены как "старые". Таким образом, алгоритм закончит работу. Если бы граф не был связным, то цикл в строках 8, 9 надо было повторить для каждой компоненты. □

Теорема 5.2. *Алгоритм* 5.2 *на графе с* n *узлами и* e *ребрами требует* $O(\text{MAX}(n, e))$ *шагов.*



а *б*

Рис. 5.7. Граф и его остовное дерево, построенное в процессе поиска в глубину.

Доказательство. Строка 7 и поиск "нового" узла в строке 8 требуют $O(n)$ шагов, если список узлов составлен и один раз просмотрен. Время, затрачиваемое на ПОИСК(v), если не считать рекурсивных вызовов процедуры самой себя, пропорционально числу узлов, смежных с v. ПОИСК(v) вызывается только один раз для данного v, поскольку после вызова узел v помечается как "старый". Таким образом, всего на ПОИСК тратится время $O(\text{МАХ}(n, e))$; теорема доказана. □

Мощность метода поиска в глубину частично отражена в приведенной ниже лемме, в которой утверждается, что каждое ребро неориентированного графа G либо принадлежит глубинному остовному лесу, либо соединяет предка с потомком в некотором дереве, входящем в глубинный остовный лес. Таким образом, все ребра графа G, будь то древесные или обратные, соединяют два узла, один из которых является предком другого в остовном лесу.

Лемма 5.3. *Если* (v, w) *— обратное ребро, то либо* v *— предок узла* w, *либо* w *— предок узла* v *в остовном лесу*,.

Доказательство. Без потери общности будем считать, что v посещается раньше w, т. е. ПОИСК(v) вызывается раньше, чем ПОИСК(w). Поэтому в тот момент, когда достигнут узел v, узел w все еще помечен как "новый". Все "новые" узлы, посещенные во время вызова процедуры ПОИСК(v), станут потомками узла v в остовном лесу. Но ПОИСК(v) не может закончиться, пока узел w не достигнут, ибо w находится в списке $L[v]$. □

Поиск в глубину задает на узлах остовного леса естественный порядок, а именно: узлы можно пометить в том порядке, в каком они посещались, если положить вначале СЧЕТ=1 между строками 6 и 7 алгоритма 5.2 и вставить в начало процедуры ПОИСК

ПГНОМЕР[v] ← СЧЕТ;
СЧЕТ ← СЧЕТ + 1;

Тогда узлы леса получат метки от 1 до их числа в лесу.

Очевидно, что для графа с n узлами эти метки можно приписать за время $O(n)$. Такой порядок соответствует прямому порядку прохождения каждого дерева в результирующем остовном лесу. В дальнейшем мы будем считать, что все глубинные остовные леса помечены таким образом. Часто мы будем обращаться с этими метками узлов так, как будто это имена узлов. Поэтому высказывание типа $v<w$, где v и w — узлы, имеет смысл.

Пример 5.3. Поиск в глубину задает на графе, изображенном на рис. 5.7,*а*, следующий порядок: v_1, v_2, v_3, v_4, v_5, v_6. В этом можно убедиться, проследив порядок вызовов процедуры ПОИСК

на различных узлах, или пройдя дерево на рис. 5.7,*б* в прямом порядке. □

Особо отметим, что если v — подлинный предок узла w, то $v<w$. Аналогично если v находится слева от w в дереве, то $v<w$.

5.3. ДВУСВЯЗНОСТЬ

Рассмотрим приложение метода поиска в глубину к выделению в неориентированном графе двусвязных компонент. Пусть $G=(V, E)$ — связный неориентированный граф. Узел a называют *точкой сочленения* графа G, если существуют такие узлы v и w, что v, w и a различны и всякий путь между v и w содержит узел a. Иначе говоря, a — точка сочленения графа G, если удаление узла a расщепляет G не менее чем на две части. Граф G называется *двусвязным*, если для любой тройки различных узлов v, w, a существует путь между v и w, не содержащий a. Таким образом, неориентированный связный граф двусвязен тогда и только тогда, когда в нем нет точек сочленения.

На множестве ребер графа G можно задать естественное отношение, полагая, что для ребер e_1 и e_2 выполняется это отношение, если $e_1=e_2$ или они лежат на некотором цикле. Легко показать, что это отношение является отношением эквивалентности [1]), разбивающим множество ребер графа G на такие классы эквивалентности $E_1, E_2, \ldots, E_k$, что два различных ребра принадлежат одному и тому же классу тогда и только тогда, когда они лежат на общем цикле. Для $1\leqslant i\leqslant k$ обозначим через V_i множество узлов, лежащих на ребрах из E_i. Каждый граф $G_i=(V_i, E_i)$ называется *двусвязной компонентой* графа G.

Пример 5.4. Рассмотрим неориентированный граф на рис. 5.8,*а*. Узел v_4 является точкой сочленения, так как каждый путь между v_1 и v_7 проходит через v_4. Классы эквивалентности, состоящие из ребер, лежащих на общих циклах, таковы:

$$\{(v_1, v_2), (v_1, (v_3), (v_2, v_3)\},$$
$$\{(v_2, v_4), (v_2, v_5), (v_4, v_5)\},$$
$$\{(v_4, v_6)\},$$
$$\{(v_6, v_7), (v_6, v_8), (v_6, v_9), (v_7, v_8), (v_8, v_9)\}.$$

Эти классы порождают двусвязные компоненты, изображенные на рис. 5.8,*б*. Не очевидно здесь лишь то, что ребро (v_4, v_6), будучи

[1]) R называется *отношением эквивалентности* на множестве S, если R *рефлексивно* (aRa для всех $a\in S$), *симметрично* (из aRb следует bRa для всех $a, b\in S$) и *транзитивно* (из aRb и bRc следует aRc). Легко показать, что отношение эквивалентности на S разбивает S на непересекающиеся классы эквивалентности. (Подмножество $[a]=\{b|bRa\}$ называется *классом эквивалентности*.)

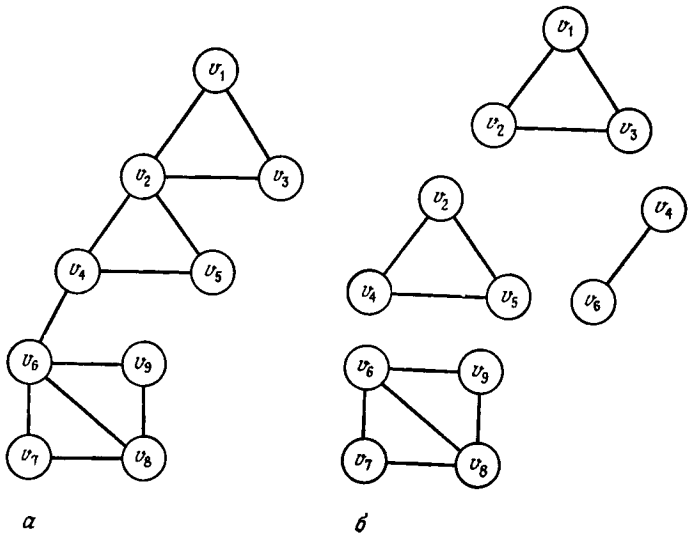


Рис. 5.8. *а* — неориентированный граф, *б* — его двусвязные компоненты.

само классом эквивалентности (оно не входит ни в один цикл), порождает "двусвязную компоненту", состоящую из v_4 и v_6. □

Следующая лемма дает полезную информацию о двусвязности.

Лемма 5.4. *Пусть* $G_i=(V_i, E_i)$ *для* $1\leqslant i\leqslant k$ — *двусвязные компоненты связного неориентированного графа* $G=(V, E)$. *Тогда*

1) *граф* G_i *двусвязен для каждого* i, $1\leqslant i\leqslant k$;
2) *для всех* $i\neq j$ *пересечение* $V_i \cap V_j$ *содержит не более одного узла*;
3) *а — точка сочленения графа* G *тогда и только тогда, когда* $a\in V_i \cap V_j$ *для некоторых* $i\neq j$.

Доказательство. 1) Допустим, что найдутся такие три различных узла v, w и a в V_i, что все пути в G_i между v и w проходят через a. Тогда, разумеется, $(v, w)\notin E_i$. Следовательно, в E_i есть различные ребра (v, v') и (w, w'), а в G_i — цикл, содержащий их. По определению двусвязной компоненты все ребра и узлы на этом цикле принадлежат E_i и V_i соответственно. Поэтому в G_i есть два пути между v и w и только один из них содержит a; получили противоречие.

2) Допустим, что пересечению $V_i \cap V_j$ принадлежат два различных узла v и w. Тогда существуют цикл C_1 в G_i, содержащий v и w, и цикл C_2 в G_j, также содержащий v и w. Поскольку E_i и E_j не пересекаются, то множества ребер, входящих в C_1 и C_2, тоже не пере-

секаются. Тем не менее из ребер этих циклов можно построить новый цикл, содержащий узлы v и w. Отсюда следует, что хотя бы одно ребро в E_i совпадает с каким-то ребром в E_j. Таким образом, вопреки предположению E_i и E_j не являются классами эквивалентности.

3) Пусть a — точка сочленения графа G. Тогда существуют такие два узла v и w, что v, w и a различны и каждый путь между v и w содержит a. Так как граф G связен, то хотя бы один такой путь найдется. Пусть (x, a) и (y, a) — два ребра, лежащие на некотором пути между v и w и инцидентные a. Если существует цикл, содержащий эти два ребра, то некоторый путь между v и w не содержит a. Следовательно, (x, a) и (y, a) входят в разные двусвязные компоненты, а узел a принадлежит пересечению множеств их узлов.

Обратно, если $a \in V_i \cap V_j$, то найдутся ребра (x, a) и (y, a) соответственно в E_i и E_j. Так как они не лежат оба на одном и том же цикле, то всякий путь из x в y содержит a. Следовательно, a — точка сочленения. □

Поиск в глубину особенно полезен для нахождения двусвязных компонент неориентированного графа. Отчасти это связано с тем, что, согласно лемме 5.3, в нем нет "поперечных ребер". Иными словами, если узел v — не предок и не потомок узла w в остовном лесу, то v и w не могут соединяться никаким ребром.

Если a — точка сочленения, то удаление ребра a и всех ребер, инцидентных ему, расщепляет граф G по крайней мере на две части.

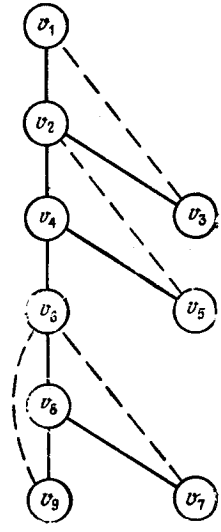


Рис. 5.9. Остовное дерево, построенное поиском в глубину.

Одна из них состоит из сына s узла a и всех его потомков в глубинном остовном дереве. Следовательно, в этом остовном дереве узел a должен иметь сына s, потомки которого не соединены обратными ребрами с подлинными предками узла a. Обратно, из-за отсутствия поперечных ребер узел a, отличный от корня, является точкой сочленения, если никакой потомок некоторого его сына не соединен обратным ребром ни с каким подлинным предком узла a. Корень глубинного остовного дерева является точкой сочленения тогда и только тогда, когда он имеет не менее двух сыновей.

Пример 5.5. Остовное дерево, построенное поиском в глубину, для графа, изображенного на рис. 5.8,*а*, показано на рис. 5.9. Точками сочленения являются v_2, v_4 и v_6. Узел v_2 имеет сына v_4, и ни из какого потомка узла v_4 не выходит обратного ребра к подлинному предку узла v_2. Аналогично узел v_4 имеет сына v_6, а v_6 — сына v_8 с подобными свойствами. □

Высказанные выше идеи выражены в следующей лемме.

Лемма 5.5. *Пусть $G=(V, E)$ — связный неориентированный граф, а $S=(V, T)$ — глубинное остовное дерево для него. Узел a является точкой сочленения графа G тогда и только тогда, когда выполнено одно из условий*:

1) *a — корень и a имеет более одного сына*;
2) *a — не корень и для некоторого его сына s нет обратных ребер между потомками узла s (в том числе самим s) и подлинными предками узла a.*

Доказательство. Легко показать, что корень является точкой сочленения тогда и только тогда, когда у него больше одного сына. Эту часть оставляем в качестве упражнения.

Предположим, что условие 2 выполнено. Пусть f — отец узла a. По лемме 5.3 каждое обратное ребро идет из некоторого узла в его же предка. Следовательно, любое обратное ребро, выходящее из потомка v узла s, идет к предку узла v. По условию леммы обратное ребро не может идти к подлинному предку узла a. Поэтому оно идет либо к a, либо к потомку узла s. Таким образом, всякий путь из s в f содержит a, откуда вытекает, что a — точка сочленения.

Для доказательства обратного утверждения допустим, что a — точка сочленения, отличная от корня. Пусть x и y — различные узлы, отличные от a и такие, что всякий путь в G между x и y содержит a. По крайней мере один из узлов x и y, скажем x, является подлинным потомком узла a в S; в противном случае в G был бы путь между x и y, проходящий по ребрам из T и не содержащий a. Пусть s — такой сын узла a, что x — потомок узла s (возможно, $x=s$). Тогда либо между потомком v узла s и подлинным предком w узла a нет обратного ребра и, значит, условие 2 выполняется, либо

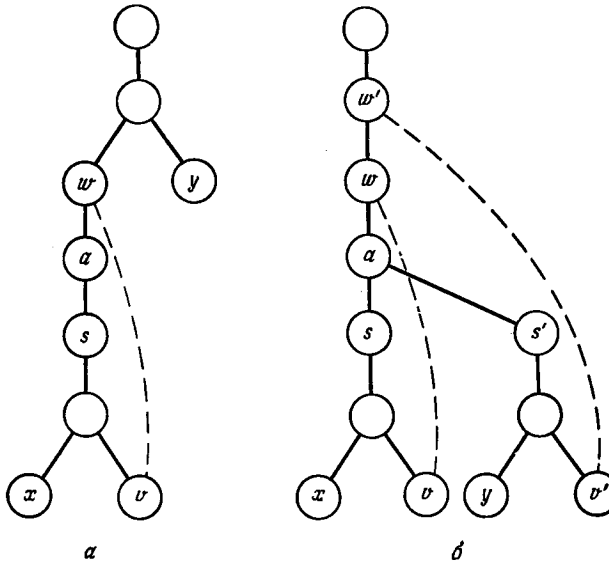


Рис. 5.10. Пути для контрпримеров.

такое ребро (v, w) есть. В этой ситуации надо рассмотреть отдельно два случая.

Случай 1. Допустим, что y — не потомок узла a. Тогда из x в v и далее в w и y идет путь, не содержащий a; получили противоречие (см. рис. 5.10,*а*).

Случай 2. Допустим, что y — потомок узла a. Ясно, что y — не потомок узла s, ибо иначе был бы путь из x в y, не содержащий a. Пусть s' — такой сын узла a, что y — потомок узла s'. Тогда либо между потомком v' узла s' и подлинным предком w' узла a нет обратного ребра и, значит, условие 2 выполняется, либо такое ребро (v', w') есть. В последнем случае из x в v и далее в w', v' и y идет путь, не содержащий a; получили противоречие (см. рис. 5.10,*б*). Итак, условие 2 выполнено. □

Пусть T и B — множества соответственно древесных и обратных ребер, построенных поиском в глубину в связном неориентированном графе $G=(V, E)$. Мы будем считать, что узлам в V приписаны их номера в последовательности, образованной в процессе поиска в глубину. Для каждого v из V определим

$$\text{НИЖНИЙ}[v] = \text{MIN}(\{v\} \cup \{w \mid \text{существует такое обратное ребро } (x, w) \in B, \text{ что } x \text{ — потомок узла } v, \text{ а } w \text{ — предок узла } v \text{ в глубинном остовном лесу } (V, T)\}). \quad (5.1)$$

Нумерация в прямом порядке обладает тем свойством, что если x — потомок узла v, а (x, w) — обратное ребро, причем $w<v$, то w — подлинный предок узла v. Следовательно, по лемме 5.5, если v — не корень, то v является точкой сочленения тогда и только тогда, когда имеет сына s, для которого НИЖНИЙ[s]$\geqslant v$.

В процедуру ПОИСК можно включить и вычисление значения функции НИЖНИЙ для каждого узла, если переписать (5.1) так, чтобы выразить НИЖНИЙ[v] через узлы, смежные с v, используя обратные ребра и значения НИЖНИЙ на сыновьях узла v. Точнее, НИЖНИЙ[v] можно вычислить, определив наименьшее значение тех узлов w, которые обладают хотя бы одним из свойств

1) $w=v$,
2) $w=$НИЖНИЙ[s] и s — сын узла v,
3) (v, w) — обратное ребро в B.

Наименьшее значение w можно будет определить, как только будет исчерпан список $L[v]$ узлов, смежных с v. Таким образом, (5.1) эквивалентно равенству

$$\text{НИЖНИЙ}[v] = \text{MIN}(\{v\} \cup \{\text{НИЖНИЙ}[s] \mid s \text{ сын узла } v\} \cup \cup \{w \mid (v, w) \in B\}). \quad (5.2)$$

В новый вариант процедуры ПОИСК, приведенный на рис. 5.11, мы включили как переименование узлов по первому посещению, так и вычисление функции НИЖНИЙ. В строке 4 величине НИЖНИЙ[v] придается начальное значение, равное ее максимально возможному значению. Если v имеет сына w в глубинном остовном лесу, то в строке 11 корректируется значение НИЖНИЙ[v], если оно оказывается больше, чем НИЖНИЙ[w]. Если узел v соединен обратным ребром с узлом w, то в строке 13 полагаем НИЖНИЙ[v]$=$ $=$ПГНОМЕР[w] при условии, что номер узла w в поиске в глубину меньше текущего значения НИЖНИЙ[v]. Текст в строке 12 проверяет, не является ли ребро (v, w) обратным из-за того, что w — отец узла v в глубинном остовном дереве. Таким образом, рис. 5.11 реализует формулу (5 2).

Вычислив значение НИЖНИЙ[v] для каждого узла v, легко найти точки сочленения. Сначала изложим полный алгоритм, а затем докажем его корректность и покажем, что он требует время $O(e)$.

Алгоритм 5.3. Нахождение двусвязных компонент

Вход. Связный неориентированный граф $G=(V, E)$.

Выход. Список ребер каждой двусвязной компоненты графа G.

Метод.

1. Вначале полагаем $T=\varnothing$ и СЧЕТ$=1$. Помечаем каждый узел в V как "новый", выбираем произвольный узел v_0 в V и вызы-

```
    procedure ПОИСКБ(v):
    begin
1.    пометить узел v как "старый";
2.    ПГНОМЕР[v] ← СЧЕТ;
3.    СЧЕТ ← СЧЕТ + 1;
4.    НИЖНИЙ[v] ← ПГНОМЕР[v];
5.    for w ∈ L[v] do
6.      if узел w помечен как "новый" then
          begin
7.          добавить (v, w) к T;
8.          ОТЕЦ[w] ← v;
9.          ПОИСКБ(w);
10.         if НИЖНИЙ[w] ≥ ПГНОМЕР[v] then
                обнаружена двусвязная компонента;
11.         НИЖНИЙ[v] ← MIN(НИЖНИЙ[v], НИЖНИЙ[w])
          end
12.     else if w — не ОТЕЦ[v] then
13.       НИЖНИЙ[v] ← MIN(НИЖНИЙ[v], ПГНОМЕР[w])
    end
```

Рис. 5.11. Поиск в глубину с вычислением функции НИЖНИЙ.

ваем ПОИСКБ(v_0) (рис. 5.11), чтобы построить глубинное остовное дерево $S=(V, T)$ и вычислить НИЖНИЙ[v] для каждого v из V.

2. Когда в строке 5 процедуры ПОИСКБ выбирается узел w, помещаем ребро (v, w) в СТЕК (т. е. магазинную память для ребер), если оно еще не там [1]). Когда в строке 10 обнаруживается пара (v, w), для которой w — сын узла v и НИЖНИЙ[w]$\geqslant v$, выталкиваем из СТЕКа все ребра вплоть до (v, w). Эти ребра образуют двусвязную компоненту графа G. □

Пример 5.6. Остовное дерево с рис. 5.9, построенное поиском в глубину, воспроизведено на рис. 5.12, причем узлы переименованы в соответствии с ПГНОМЕР и указаны значения функции НИЖНИЙ. Например, ПОИСКБ(6) устанавливает, что НИЖНИЙ[6]=4, так как есть обратное ребро (6, 4). Тогда ПОИСКБ(5),

[1]) Заметим, что если (v, w) — ребро, то $v \in L[w]$ и $w \in L[v]$. Поэтому (v, w) встречается дважды: один раз, когда посещается узел v, и второй раз, когда посещается узел w. Узнать, попало ли уже ребро (v, w) в СТЕК, можно, установив, что $v<w$ и w — «старый» узел или $v>w$ и w=ОТЕЦ[v].

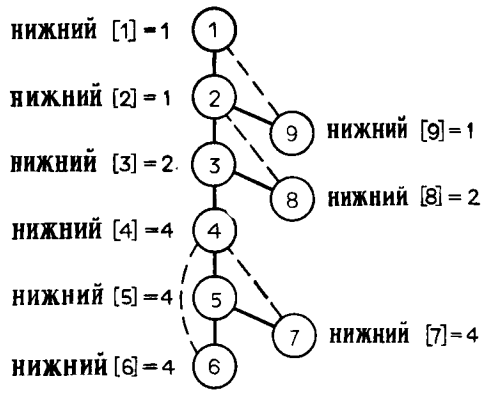


Рис. 5.12. Остовное дерево с рис. 5.9 со значениями функции НИЖНИЙ.

который вызвал ПОИСКБ(6), полагает НИЖНИЙ[5]=4, поскольку 4 меньше начального значения НИЖНИЙ[5], равного 5.

Закончив ПОИСКБ(5), обнаруживаем (строка 10), что НИЖНИЙ[5]=4. Следовательно, 4 — точка сочленения. В этот момент магазин содержит такие ребра (от дна к вершине):

(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 4), (5, 7), (7, 4).

Поэтому выталкиваем все ребра вплоть до (4, 5). Таким образом, мы подаем на выход ребра (7, 4), (5, 7), (6, 4), (5, 6) и (4, 5), которые являются ребрами первой найденной двусвязной компоненты.

Закончив ПОИСКБ(2), обнаруживаем, что НИЖНИЙ[2]=1, и опустошаем магазин, хотя 1 — не точка сочленения. Это гарантирует, что двусвязная компонента, содержащая корень, тоже будет найдена. □

Теорема 5.3. *Алгоритм* 5.3 *правильно находит двусвязные компоненты графа* G *с* e *ребрами и тратит на это время* $O(e)$.

Доказательство. Для доказательства того, что шаг 1 требует время $O(e)$, надо просто обобщить аналогичное свойство процедуры ПОИСК (теорема 5.2). Шаг 2 просматривает каждое ребро один раз, помещает его в магазин и впоследствии выталкивает его оттуда. Поэтому сложность шага 2 равна $O(e)$.

Что касается корректности алгоритма, то лемма 5.5 гарантирует, что точки сочленения будут найдены правильно. Даже если корень не является точкой сочленения, алгоритм обращается с ним как с таковой, чтобы выделить двусвязную компоненту, содержащую корень.

Мы должны доказать, что если НИЖНИЙ[w]$\geqslant v$, то по окончании процедуры ПОИСКБ(w) ребра, расположенные в СТЕКе над (v, w), будут в точности ребрами из двусвязной компоненты,

содержащей (v, w). Это делается индукцией по числу b двусвязных компонент в G. Базис, т. е. случай $b=1$, тривиален, поскольку тогда v — корень дерева, (v, w) — единственное древесное ребро, выходящее из v, и по окончании процедуры ПОИСКБ (w) все ребра графа G находятся в СТЕКе.

Теперь допустим, что предположение индукции верно для всех графов с b двусвязными компонентами, и пусть G — граф с $b+1$ двусвязными компонентами. Пусть ПОИСКБ (w) — это первый вызов процедуры ПОИСКБ, по окончании которого НИЖНИЙ$[w] \geqslant v$, где (v, w) — древесное ребро. Так как никакие ребра не были удалены из СТЕКа, то множество ребер в СТЕКе, расположенных выше (v, w), состоит из всех ребер, инцидентных потомкам узла w. Легко показать, что эти ребра в точности совпадают с ребрами той двусвязной компоненты, куда входит (v, w). Удалив их из СТЕКа, алгоритм ведет себя так, как он вел бы себя на графе G', получаемом из G, если удалить двусвязную компоненту, содержащую ребро (v, w). Осталось только сделать шаг индукции, поскольку G' состоит из b двусвязных компонент. □

5.4. ПОИСК В ГЛУБИНУ В ОРИЕНТИРОВАННОМ ГРАФЕ

Алгоритм 5.2 может также находить ориентированный остовный лес для ориентированного графа $G=(V, E)$, если определить список $L[v]$ узлов, "смежных" с узлом v, как $\{w \mid (v, w)$ — ребро из $E\}$, т. е. $L[v]$ — это список узлов, которые являются концами ребер с началом v.

Пример 5.7. На рис. 5.13,*а* изображен ориентированный граф, а на рис. 5.13,*б* — глубинный остовный лес для него. Как и раньше, мы рисуем древесные ребра сплошными линиями, а прочие ребра — штриховыми.

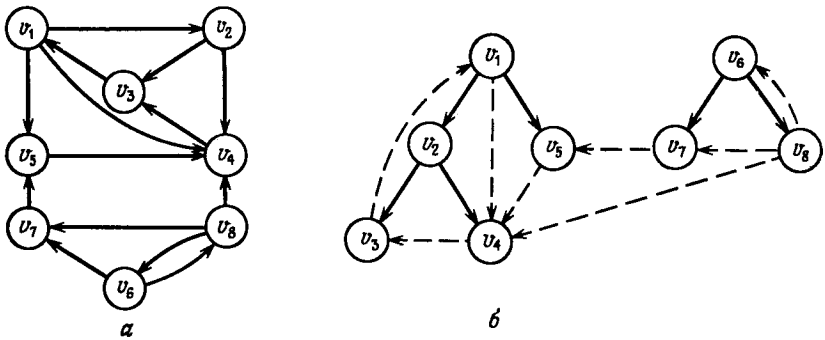


Рис. 5.13. Поиск в глубину в ориентированном графе: *а* — ориентированный граф; *б* — остовный лес.

Чтобы построить остовный лес, начнем с узла v_1. ПОИСК(v_1) вызывает ПОИСК(v_2), а тот вызывает ПОИСК(v_3). Последний вызов ничего не добавляет к дереву, ибо единственное ребро с началом v_3 идет в узел v_1, уже помеченный как "старый". Поэтому возвращаемся к ПОИСК(v_2), который добавляет v_4 в качестве второго сына узла v_2. ПОИСК(v_4) заканчивает работу, ничего не добавляя к лесу, поскольку узел v_3 уже "старый". Затем заканчивается ПОИСК(v_2), ибо все ребра, выходящие из v_2, к данному моменту просмотрены. Поэтому возвращаемся обратно в v_1 и вызываем ПОИСК(v_5). Последний вызов ничего не добавляет к дереву; аналогично ничего не может добавить и ПОИСК(v_1).

Теперь возьмем v_6 в качестве корня нового глубинного остовного дерева. Его построение аналогично предыдущему, и мы оставляем его читателю. Заметим, что узлы посещались в порядке $v_1, v_2, \ldots, v_8$. Следовательно, в процессе поиска в глубину узлу v_i был приписан номер i, $1 \leqslant i \leqslant 8$. □

При поиске в глубину на ориентированном графе в дополнение к древесным ребрам возникает еще три типа ребер. Это обратные ребра, такие, как (v_3, v_1) на рис. 5.13,*б*, поперечные справа налево, такие, как (v_4, v_3) на том же рисунке, и прямые ребра, такие, как (v_1, v_4). Однако никакое ребро не идет из узла с меньшим номером, присвоенным в процессе поиска в глубину, в ребро с бóльшим номером, если только последнее не является потомком первого. Это не случайно.

Объяснение аналогично объяснению того, почему в неориентированном случае нет поперечных ребер. Пусть (v, w) — ребро и узел v был посещен до w (т. е. $v < w$). Каждый узел, которому приписывается номер в период между началом и концом процедуры ПОИСК(v), становится потомком узла v. Но узел w должен получить свой номер, когда рассматривается ребро (v, w), если только он уже не получил номер раньше. Если w получает номер в то время, когда рассматривается ребро (v, w), то (v, w) становится древесным ребром, а в противном случае прямым ребром. Таким образом, не может быть такого поперечного ребра (v, w), что $v < w$.

Поиск в глубину в графе G разбивает множество его ребер на четыре класса.

1) *Древесные ребра*, идущие к новым узлам в процессе поиска.
2) *Прямые ребра*, идущие от предков к подлинным потомкам, но не являющиеся древесными ребрами.
3) *Обратные ребра*, идущие от потомков к предкам (возможно, из узла в себя).
4) *Поперечные ребра*, соединяющие узлы, которые не являются ни предками, ни потомками друг друга.

Ключевое свойство поперечных ребер устанавливается в следующей лемме.

Лемма 5.6. *Если* (v, w) *— поперечное ребро, то* $v > w$, *т. е. поперечные ребра идут справа налево.*

Доказательство. Доказательство аналогично доказательству леммы 5.3 и остается в качестве упражнения. □

5.5. СИЛЬНАЯ СВЯЗНОСТЬ

В качестве примера эффективного алгоритма, который стал возможным благодаря поиску в глубину на ориентированном графе, рассмотрим задачу распознавания сильной связности ориентированного графа, т. е. существования на нем пути из каждого узла в любой другой.

Определение. Пусть $G=(V, E)$ — ориентированный граф. Разобьем множество его вершин V на такие классы эквивалентности V_i, $1 \leqslant i \leqslant r$, что узлы v и w эквивалентны тогда и только тогда, когда на графе есть пути из v в w и обратно. Пусть E_i, $1 \leqslant i \leqslant r$, — множество ребер, соединяющих пары узлов в V_i. Графы $G_i=(V_i, E_i)$ называются *сильно связными компонентами* графа G. Хотя каждый узел графа G принадлежит некоторому множеству V_i, у графа G могут быть ребра, не принадлежащие ни одному множеству E_i. Граф называется *сильно связным*, если он имеет только одну сильно связную компоненту.

Применим поиск в глубину для нахождения сильно связных компонент графа. Сначала покажем, что узлы каждой сильно связной компоненты образуют связный подграф в глубинном остовном лесу. Этот связный подграф представляет собой дерево, и его корень называется *корнем сильно связной компоненты*. Однако не каждое дерево в глубинном остовном лесу непременно представляет какую-то сильно связную компоненту.

Лемма 5.7. *Пусть* $G_i=(V_i, E_i)$ *— сильно связная компонента ориентированного графа* G, *а* $S=(V, T)$ *— глубинный остовный лес для* G. *Тогда узлы графа* G_i *вместе с ребрами, входящими в пересечение* $E_i \cap T$, *образуют дерево.*

Доказательство. Пусть v и w — узлы из V_i. (Мы предполагаем, что именами узлов являются номера, присвоенные им в процессе поиска в глубину.) Без потери общности будем считать, что $v < w$. Так как узлы v и w оба принадлежат одной и той же сильно связной компоненте, то существует путь P в G_i, идущий из v в w. Пусть x — узел на P с наименьшим номером (возможно, это сам узел v). Путь P, дойдя до какого-нибудь потомка узла x, уже не сможет выйти за пределы поддерева потомков узла x, ибо за пределы этого поддерева выходят лишь поперечные ребра и обратные ребра, идущие в узлы с номерами, меньшими x. (Так

как потомки узла x занумерованы последовательно, начиная с x, то поперечное или обратное ребро, выходящее из поддерева потомков узла x, должно идти в узел с номером, меньшим x.) Следовательно, w — потомок узла x. Поскольку узлы занумерованы в прямом порядке, то все узлы, номера которых заключены между x и w, также являются потомками узла x. Так как $x \leqslant v < w$, то v — потомок узла x.

Мы только что показали, что любые два узла в G_i имеют общего предка в G_i. Пусть r — общий предок узлов графа G_i с наименьшим номером. Если v — узел графа G_i, то любой узел на пути из r в v, проходящем в остовном дереве, также является узлом графа G_i. □

Сильно связные компоненты ориентированного графа G можно найти, построив корни компонент в том порядке, в каком они встретились в последний раз в процессе поиска в глубину на G. Пусть $r_1, r_2, \ldots, r_k$ — эти корни в том порядке, в каком заканчивался их поиск в глубину (т. е. поиск узла r_i заканчивался перед поиском r_{i+1}). Тогда для каждого $i < j$ либо r_i лежит слева от r_j, либо r_i — потомок узла r_j в глубинном остовном лесу.

Пусть G_i — сильно связная компонента с корнем r_i, $1 \leqslant i \leqslant k$. Тогда G_1 состоит из всех потомков узла r_1, поскольку никакой узел r_j, $j > 1$, не может быть потомком узла r_1. Аналогично можно доказать следующую лемму.

Лемма 5.8. *Для любого i, $1 \leqslant i \leqslant k$, граф G_i состоит из узлов, являющихся потомками узла r_i и не принадлежащих ни одному из графов $G_1, G_2, \ldots, G_{i-1}$.*

Доказательство. Корень r_j для $j > 1$ не может быть потомком узла r_i, поскольку вызов процедуры ПОИСК(r_j) завершается после работы процедуры ПОИСК(r_i). □

Для нахождения корней введем функцию, называемую НИЖНЯЯСВЯЗЬ:

НИЖНЯЯСВЯЗЬ[v] = MIN ({v} ∪ {w | есть поперечное или обратное ребро из некоторого потомка узла v в узел w, а корень той сильно связной компоненты, которая содержит w, является предком узла v}). (5.3)

На рис. 5.14 изображено поперечное ребро из потомка узла v в узел w, а корень r сильно связной компоненты, содержащей w, является предком узла v.

Мы скоро увидим, как в процессе поиска в глубину вычислять функцию НИЖНЯЯСВЯЗЬ. Но прежде охарактеризуем корни сильно связных компонент в терминах значений этой функции.

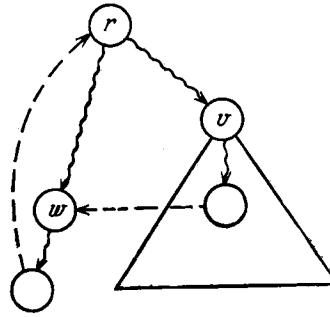


Рис. 5.14. Поперечное ребро, удовлетворяющее условию из определения функции НИЖНЯЯСВЯЗЬ.

Лемма 5.9. *Пусть G — ориентированный граф. Для того чтобы узел v был корнем сильно связной компоненты графа G, необходимо и достаточно, чтобы* НИЖНЯЯСВЯЗЬ$[v]=v$.

Доказательство. *Необходимость.* Пусть v — корень сильно связной компоненты графа G. По определению НИЖНЯЯСВЯЗЬ$[v]\leqslant v$. Допустим, что НИЖНЯЯСВЯЗЬ$[v]<v$. Тогда найдутся такие узлы w и r, что

1) в w входит поперечное или обратное ребро, выходящее из потомка узла v,
2) r — корень сильно связной компоненты, содержащей w,
3) r — предок узла v,
4) $w<v$.

По условию 2 r — предок узла w, так что $r\leqslant w$. Следовательно, по условию 4 $r<v$, откуда в силу условия 3 вытекает, что r — подлинный предок узла v. Но r и v должны принадлежать одной и той же сильно связной компоненте, поскольку в G идут пути из r в v и из v в r через w. Поэтому v не корень сильно связной компоненты; получили противоречие. Таким образом, НИЖНЯЯСВЯЗЬ$[v]=v$.

Достаточность. Пусть НИЖНЯЯСВЯЗЬ$[v]=v$. Если v — не корень сильно связной компоненты, содержащей v, то корнем будет некоторый подлинный предок r узла v. Следовательно, есть путь P из v в r. Рассмотрим первое ребро в P, идущее из потомка узла v в узел w, не являющийся потомком узла v. Это ребро либо обратное и идет к предку узла v, либо поперечное и идет в узел, номер которого меньше v. В любом случае $w<v$.

Осталось показать, что r и w принадлежат одной и той же сильно связной компоненте графа G. Из r в v должен идти путь, поскольку r — предок узла v. Путь P идет из v через w в r. Следовательно,

r и w принадлежат одной и той же сильно связной компоненте. Таким образом, НИЖНЯЯСВЯЗЬ[v]$\leqslant w<v$; получили противоречие. □

При поиске в глубину легко вычислить значения функции НИЖНЯЯСВЯЗЬ. Сами сильно связные компоненты также легко найти. Для этого узлы графа G помещаются в стек в том порядке, в каком они посещаются во время поиска. Всякий раз, когда обнаруживается корень, все узлы соответствующей сильно связной компоненты находятся в верхней части стека и выталкиваются наружу. Эта стратегия "работает" в силу леммы 5.8 и свойств нумерации, порождаемой поиском в глубину.

Попав в узел v в первый раз, полагаем НИЖНЯЯСВЯЗЬ[v]=v. Когда встречается обратное или поперечное ребро (v, w) и w принадлежит той же сильно связной компоненте, что и некоторый предок узла v, то полагаем значение функции НИЖНЯЯСВЯЗЬ в узле v равным меньшему из двух чисел: ее текущего значения и w. Когда встречается древесное ребро (v, w), рекурсивно просматривается поддерево с корнем w и вычисляется НИЖНЯЯСВЯЗЬ[w]. По возвращении к v значение функции НИЖНЯЯСВЯЗЬ в узле v полагается равным меньшему из чисел: ее текущего значения и НИЖНЯЯСВЯЗЬ[w].

Чтобы проверить, принадлежит ли w той же компоненте, что и некоторый предок узла v, достаточно посмотреть, находится ли еще узел w в стеке для узлов. Эта проверка осуществляется с помощью массива, указывающего наличие узла в стеке. Заметим, что по лемме 5.8 если w еще в стеке, то корень сильно связной компоненты, содержащей w, является предком узла v.

Изложим модификацию процедуры ПОИСК, которая вычисляет функцию НИЖНЯЯСВЯЗЬ. Она использует магазинный список СТЕК для узлов. Эта процедура приведена на рис. 5.15.

Функция НИЖНЯЯСВЯЗЬ вычисляется в строках 4, 9 и 11. В строке 4 в качестве начального значения для НИЖНЯЯСВЯЗЬ[v] берется номер узла v, присвоенный ему поиском в глубину. В строке 9 полагаем НИЖНЯЯСВЯЗЬ[v]=НИЖНЯЯСВЯЗЬ[w], если для некоторого сына w узел НИЖНЯЯСВЯЗЬ[w] оказался меньше текущего значения НИЖНЯЯСВЯЗЬ[v]. В строке 10 узнаем, является (v, w) обратным или поперечным ребром, и проверяем, не найдена ли уже сильно связная компонента, содержащая w. Если нет, то корень сильно связной компоненты, содержащей w, будет предком узла v. По необходимости полагаем в строке 11 значение НИЖНЯЯСВЯЗЬ[v] равным номеру узла w, присвоенному ему поиском в глубину, если оно раньше не получило меньшее значение.

Теперь сформулируем весь алгоритм нахождения сильно связных компонент ориентированного графа.

```
    procedure ПОИСКВ(v):
    begin
 1.   пометить узел v как "старый";
 2.   ПГНОМЕР[v] ← СЧЕТ;
 3.   СЧЕТ ← СЧЕТ + 1;
 4.   НИЖНЯЯСВЯЗЬ[v] ← ПГНОМЕР[v];
 5.   затолкнуть v в СТЕК;
 6.   for w ∈ L[v] do
 7.       if узел w помечен как "новый" then
            begin
 8.            ПОИСКВ(w);
 9.            НИЖНЯЯСВЯЗЬ[v] ← MIN(НИЖНЯЯСВЯЗЬ[v],
                                    НИЖНЯЯСВЯЗЬ[w])
            end
       else
10.       if ПГНОМЕР[w] < ПГНОМЕР[v] и w ∈ СТЕК then
11.         НИЖНЯЯСВЯЗЬ[v] ← MIN(ПГНОМЕР[w],
                                 НИЖНЯЯСВЯЗЬ[v]);
12.  if НИЖНЯЯСВЯЗЬ[v] = ПГНОМЕР[v] then
       begin
         repeat
           begin
13.          вытолкнуть x из вершины СТЕКа;
14.          print x
           end
15.      until x = v;
16.      print "конец сильно связной компоненты"
       end
    end
```

Рис. 5.15. Процедура для вычисления функции НИЖНЯЯСВЯЗЬ.

Алгоритм 5.4. Нахождение сильно связных компонент ориентированного графа

Вход. Ориентированный граф $G=(V, E)$.
Выход. Список сильно связных компонент графа G.

Метод.
begin
СЧЕТ ← 1;
for $v \in V$ **do** пометить узел v как "новый";
сделать СТЕК пустым;
while существует узел v, помеченный как "новый" **do**
ПОИСКВ (v)
end □

Пример 5.8. Рассмотрим глубинный остовный лес на рис. 5.13. Этот лес воспроизведен на рис. 5.16, где указаны также значения функции НИЖНЯЯСВЯЗЬ. Вызов ПОИСКВ (3) заканчивает работу первым среди всех вызовов процедуры ПОИСКВ. Когда мы исследуем ребро (3,1), полагаем НИЖНЯЯСВЯЗЬ[3]=MIN(1, 3)=1. По возвращении к ПОИСКВ (2) полагаем НИЖНЯЯСВЯЗЬ[2]= =MIN (2, НИЖНЯЯСВЯЗЬ[3])=1. Затем вызываем ПОИСКВ (4) для исследования ребра (4, 3). Так как 3<4 и 3 еще находится в СТЕКе, полагаем НИЖНЯЯСВЯЗЬ[4]=MIN (3, 4)=3.

Затем возвращаемся к вызову ПОИСКВ (2) и полагаем НИЖНЯЯСВЯЗЬ[2] равным меньшему из чисел: НИЖНЯЯСВЯЗЬ[4] и текущего значения НИЖНЯЯСВЯЗЬ[2] (равного 1). Поскольку последнее значение меньше, то НИЖНЯЯСВЯЗЬ[2] не меняется. Возвращаемся к вызову ПОИСКВ (1), полагая НИЖНЯЯСВЯЗЬ[1]=MIN (1, НИЖНЯЯСВЯЗЬ[2])=1. Исследовав далее ребро (1, 4), ничего не делаем, ибо (1, 4) — прямое ребро, и условие строки 10 процедуры ПОИСКВ не выполняется, так как 4>1.

Теперь вызываем ПОИСКВ (5), и поперечное ребро (5, 4) вынуждает нас положить НИЖНЯЯСВЯЗЬ[5]=4, ибо 4<5 и 4 ∈ СТЕК. Когда снова возвращаемся к вызову ПОИСКВ (1), полагаем значение НИЖНЯЯСВЯЗЬ[1] равным меньшему из чисел : 1 (предыдущее значение) и НИЖНЯЯСВЯЗЬ[5], т. е. НИЖНЯЯСВЯЗЬ[1]=1.

Затем, поскольку все ребра, выходящие из 1, уже рассмотрены и НИЖНЯЯСВЯЗЬ[1]=1, мы обнаруживаем, что 1 — корень сильно связной компоненты. Эта компонента состоит из 1 и всех узлов, находящихся в стеке выше 1. Так как узел 1 посещался первым, то узлы 2, 3, 4 и 5 находятся выше 1 и именно в этом по-

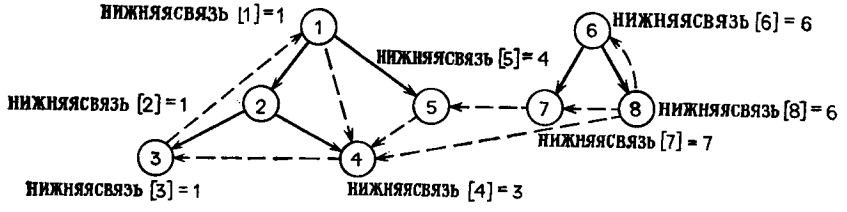


Рис. 5.16. Остовный лес с вычисленной функцией НИЖНЯЯСВЯЗЬ.

рядке. Поэтому стек опустошается и список узлов 1, 2, 3, 4, 5 печатается в качестве сильно связной компоненты графа G.

Другие сильно связные компоненты — это {7} и {6, 8}. Оставляем читателю закончить вычисление функции НИЖНЯЯСВЯЗЬ и сильно связных компонент, отправляясь от узла 6. Заметим, что последние встречи с корнями сильно связных компонент происходили в порядке 1, 7, 6. □

Теорема 5.4. *Алгоритм* 5.4 *правильно находит сильно связные компоненты графа* G *за время* $O(\text{MAX}(n, e))$, *где* n *— число узлов, а* e *— число ребер ориентированного графа* G.

Доказательство. Легко проверить, что на один вызов процедуры ПОИСКВ (v), не считая рекурсивных вызовов ПОИСКВ внутри этого вызова, тратится, кроме постоянного количества времени, время, пропорциональное числу ребер, выходящих из v. Поэтому все вызовы ПОИСКВ вместе занимают время, пропорциональное сумме количества узлов и количества ребер, поскольку ПОИСКВ вызывается только один раз для каждого узла. Участки алгоритма 5.4, отличные от процедуры ПОИСКВ, очевидно, можно реализовать за время $O(n)$. Тем самым оценка времени установлена.

Чтобы доказать корректность алгоритма, достаточно показать индукцией по числу тех вызовов процедуры ПОИСКВ, которые завершили работу, что по окончании ПОИСКВ (v) значение НИЖНЯЯСВЯЗЬ[v] вычислено правильно. После строк 12—16 процедуры ПОИСКВ узел v становится корнем сильно связной компоненты тогда и только тогда, когда НИЖНЯЯСВЯЗЬ[v]=v. Далее, в качестве результата печатаются в точности потомки узла v, не входящие в компоненты, корни которых были найдены раньше v (по лемме 5.8). А именно, узлы, находящиеся в стеке выше узла v, являются его потомками, а их корни не были найдены ранее v, поскольку эти узлы все еще находятся в стеке.

Чтобы доказать корректность вычисления функции НИЖНЯЯСВЯЗЬ, заметим, что на рис. 5.15 есть два места, где значение НИЖНЯЯСВЯЗЬ[v] могло быть меньше v; это строки 9 и 11 процедуры ПОИСКВ. В первом случае w — сын узла v и НИЖНЯЯСВЯЗЬ[w]$<v$. Тогда некоторый узел x=НИЖНЯЯСВЯЗЬ[w] можно достичь из потомка y узла w через поперечное или обратное ребро. Кроме того, корень r сильно связной компоненты, содержащей x, является предком узла w. Поскольку $x<v$, то $r<v$, и, значит, r — подлинный предок узла v. Таким образом, значение НИЖНЯЯСВЯЗЬ[v] должно быть не больше, чем НИЖНЯЯСВЯЗЬ[w].

Во втором случае — в строке 11 — из v идет поперечное или обратное ребро в узел $w<v$, сильно связная компонента C которого еще не найдена. Вызов процедуры ПОИСКВ на корне r из C еще не закончился, так что r должен быть предком узла v. (Так как $r\leqslant w<v$, то либо r лежит слева от v, либо r — предок узла v. Но

если бы r был слева от v, то вызов ПОИСКВ(r) закончился бы.) Снова получаем, что значение НИЖНЯЯСВЯЗЬ[v] должно быть не больше w.

Осталось доказать, что ПОИСКВ вычисляет значение НИЖНЯЯСВЯЗЬ[v] столь малым, сколь оно должно быть. Допустим, что это не так, т. е. найдется предок x узла v, из которого в некоторый узел y идет поперечное или обратное ребро, а корень r сильно связной компоненты, содержащей y, является предком узла v. Надо показать, что НИЖНЯЯСВЯЗЬ[v] оказывается не больше y.

Случай 1. $x=v$. В силу предположения индукции и леммы 5.9 можно считать, что все сильно связные компоненты, найденные до сих пор, корректны. Тогда узел y все еще должен быть в СТЕКе, поскольку ПОИСКВ(r) еще не закончился. Следовательно, в строке 11 значение НИЖНЯЯСВЯЗЬ[v] полагается равным y или меньшему числу.

Случай 2. $x\neq v$. Пусть z — сын узла v, потомком которого является x. Тогда в силу предположения индукции по окончании вызова ПОИСКВ(z) значение НИЖНЯЯСВЯЗЬ[z] должно быть равно y или меньшему числу. В строке 9 значение НИЖНЯЯСВЯЗЬ[v] становится именно таким, если раньше оно уже не было меньше. □

5.6. ЗАДАЧИ НАХОЖДЕНИЯ ПУТЕЙ

В этом разделе мы изучим две часто встречающиеся задачи, касающиеся путей, соединяющих узлы. В дальнейшем G будет ориентированным графом. *(Рефлексивным* и) *транзитивным замыканием* графа G называется граф G^*, который содержит то же множество узлов, что и G, но в котором из v в w идет ребро тогда и только тогда, когда в G существует путь (длины 0 или больше) из v в w.

С задачей нахождения транзитивного замыкания графа тесно связана *задача о кратчайшем пути.* Поставим в соответствие каждому ребру e графа G неотрицательную *стоимость* $c(e)$. *Стоимость пути* определяется как сумма стоимостей ребер, образующих его. Задача о кратчайшем пути состоит в нахождении для каждой пары узлов (v, w) пути наименьшей стоимости среди всех путей из v в w.

Оказывается, что идеи, на которых основаны лучшие из известных алгоритмов нахождения транзитивного замыкания и кратчайшего пути, являются (простыми) частными случаями соображений, исходя из которых решается задача нахождения (бесконечного) множества всех путей между каждой парой узлов. Чтобы обсудить эту задачу во всей ее общности, введем специальную алгебраическую структуру.

Определение. *Замкнутым полукольцом* называется система

$(S, +, \cdot, 0, 1)$, где S — множество элементов, а $+$ и $\cdot$ — бинарные операции на S, обладающие следующими пятью свойствами:

1) $(S, +, 0)$ — *моноид*, т. е. множество S *замкнуто* относительно $+$ ($a+b \in S$ для всех a и b из S), операция $+$ *ассоциативна* ($a+(b+c)=(a+b)+c$ для всех a, b, c из S) и 0 служит *единичным элементом* ($a+0=0+a=a$ для всех a из S). Тройка $(S, \cdot, 1)$ также является моноидом. Кроме того, мы предполагаем, что 0 служит *аннулятором*, т. е. $a \cdot 0=0 \cdot a=0$.
2) Операция $+$ *коммутативна* ($a+b=b+a$) и *идемпотентна* ($a+a=a$).
3) Операция $\cdot$ *дистрибутивна* относительно $+$ ($a \cdot (b+c)=a \cdot b+a \cdot c$ и $(b+c) \cdot a=b \cdot a+c \cdot a$):
4) Если $a_1, a_2, \ldots, a_i, \ldots$ — счетная последовательность элементов из S, то сумма $a_1+a_2+\ldots+a_i+\ldots$ существует и единственна. Более того, ассоциативность, коммутативность и идемпотентность выполняются не только для конечных, но и для бесконечных сумм.
5) Операция $\cdot$ дистрибутивна не только относительно конечных сумм, но и относительно счетных (это не следует из свойства 3). Таким образом, из свойств 4 и 5 вытекает, что

$$\left(\sum_i a_i\right) \cdot \left(\sum_j b_j\right) = \sum_{i,j} (a_i \cdot b_j) = \sum_i \left(\sum_j (a_i \cdot b_j)\right).$$

Пример 5.9. Приведем три системы замкнутых полуколец.

1. Пусть $S_1=(\{0, 1\}, +, \cdot, 0, 1)$, а сложение и умножение заданы таблицами

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Свойства 1—3 проверить легко. Что касается свойств 4 и 5, то заметим, что счетная сумма равна 0 тогда и только тогда, когда все слагаемые равны 0.

2. Пусть $S_2=(R, \mathrm{MIN}, +, +\infty, 0)$, где R — множество неотрицательных вещественных чисел, включая $+\infty$. Легко проверить, что $+\infty$ служит единичным элементом для MIN, а 0 — для $+$.

3. Пусть Σ — конечный алфавит (т. е. множество символов) и $S_3=(F_\Sigma, \cup, \cdot, \varnothing, \{\varepsilon\})$, где F_Σ — семейство множеств, состоящих из конечных цепочек символов из Σ, включая пустую цепочку ε (т. е. цепочку длины 0). Здесь $\cup$ обозначает операцию объединения

множеств, а $\cdot$ — их конкатенацию [1]). Единичным элементом для $\cup$ служит $\varnothing$, а для $\cdot$ — $\{\varepsilon\}$. Свойства 1—3 читатель может проверить сам. Что касается свойств 4 и 5, то достаточно заметить, что счетные объединения определяются такой эквивалентностью: $x \in (A_1 \cup A_2 \cup \ldots)$ равносильно $x \in A_i$ для некоторого i. □

Центральной операцией в нашем анализе замкнутых полуколец будет унарная операция *, называемая *замыканием*. Если $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо и $a \in S$, то положим $a^* = \sum_{i=0}^{\infty} a^i$, где $a^0 = 1$ и $a^i = a \cdot a^{i-1}$. Иными словами, значение a^* равно бесконечной сумме $1 + a + a \cdot a + a \cdot a \cdot a + \ldots$. Заметим, что свойство 4 определения замкнутого полукольца гарантирует, что $a^* \in S$. Из свойств 4 и 5 вытекает равенство $a^* = 1 + a \cdot a^*$. Отметим, что $0^* = 1^* = 1$.

Пример 5.10. Рассмотрим полукольца S_1, S_2 и S_3 из примера 5.9. В S_1 выполняется $a^* = 1$ для $a = 0$ или 1. В S_2 равенство $a^* = 0$ выполнено для всех $a \in R$. В S_3 выполняется $A^* = \{\varepsilon\} \cup \{x_1 x_2 \ldots x_k \mid k \geqslant 1$ и $x_i \in A$ для $1 \leqslant i \leqslant k\}$ для всех $A \in F_\Sigma$. Например, $\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$, т. е. $\{a, b\}^*$ состоит из всех цепочек символов a и b, включая пустую цепочку. Вообще $F_\Sigma = \mathcal{P}(\Sigma^*)$, где $\mathcal{P}(X)$ обозначает множество-степень для X. □

Теперь возьмем ориентированный граф $G = (V, E)$, в котором каждое ребро помечено элементом некоторого замкнутого полукольца $(S, +, \cdot, 0, 1)$ [2]). Определим *метку пути* как произведение $(\cdot)$ меток ребер, составляющих этот путь, причем метки берутся в порядке прохождения ребер. В частности, метка пути нулевой длины равна 1 (единичному элементу для операции $\cdot$ рассматриваемого полукольца). Для каждой пары узлов (v, w) определим $c(v, w)$ как сумму меток всех путей между v и w. Назовем $c(v, w)$ *стоимостью* прохождения из v в w. Условимся считать, что сумма по пустому множеству путей равна 0 (единичному элементу для операции $+$ нашего полукольца). Заметим, что если G имеет циклы, то между v и w может быть бесконечно много путей, но аксиомы замкнутого полукольца гарантируют, что $c(v, w)$ определяется корректно.

Пример 5.11. Рассмотрим ориентированный граф на рис. 5.17, в котором каждое ребро помечено элементом полукольца S_1, описанного в примере 5.9. Метка пути v, w, x равна $1 \cdot 1 = 1$. Простой

[1]) *Конкатенацией* $A \cdot B$ множеств A и B называется множество $\{x \mid x = yz$ $y \in A$ и $z \in B\}$.

[2]) Читателю следует не упустить из виду аналогию между рассматриваемой ситуацией и недетерминированным конечным автоматом (см. Хопкрофт, Ульман [1969] или Ахо, Ульман [1972]), который мы еще обсудим в разд. 9.1. Там узлы представляют собой состояния, а метки ребер — символы из некоторого конечного алфавита.

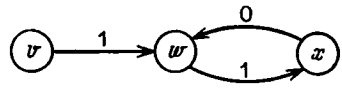


Рис. 5.17. Помеченный ориентированный граф.

цикл из w в w имеет метку $1 \cdot 0 = 0$. Вообще каждый путь положительной длины, идущий из w в w, имеет метку 0. Однако стоимость пути нулевой длины из w в w равна 1. Следовательно, $c(w, w) = 1$ □

Изложим алгоритм вычисления $c(v, w)$ для всех пар узлов v и w. Основными шагами этого алгоритма, занимающими единицу времени, будут операции $+$, $\cdot$ и $*$ произвольного замкнутого полукольца. Конечно, для таких структур, как множество всех множеств цепочек над некоторым алфавитом, не понятно, можно ли в принципе реализовать эти операции в отведенное им "единичное время". Но для тех полуколец, которыми мы будем пользоваться, эти операции легко выполнимы.

Алгоритм 5.5. Вычисление стоимости прохождения между узлами

Вход. Ориентированный граф $G = (V, E)$, где $V = \{v_1, v_2, \ldots, v_n\}$ и функция разметки l: $(V \times V) \to S$, где $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо. Полагаем $l(v_i, v_j) = 0$, если $(v_i, v_j) \notin E$.

Выход. Для всех i и j, заключенных между 1 и n, элемент $c(v_i, v_j)$ из S, равный сумме по всем путям из v_i в v_j меток этих путей.

Метод. Вычисляем C_{ij}^k для всех $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$ и $0 \leqslant k \leqslant n$. Величины C_{ij}^k задуманы как суммы меток путей, идущих из v_i в v_j, все узлы которых, кроме, быть может, концов, принадлежат множеству $\{v_1, v_2, \ldots, v_k\}$. Например, путь v_9, v_3, v_8 рассматривается при вычислении C_{98}^4 и C_{98}^3, но не C_{98}^2. Алгоритм выглядит так:

```
        begin
1.        for i ← 1 until n do C⁰_ii ← 1 + l(v_i, v_i);
2.        for 1 ⩽ i, j ⩽ n и i ≠ j do C⁰_ij ← l(v_i, v_j);
3.        for k ← 1 until n do
4.            for 1 ⩽ i, j ⩽ n do
5.                C^k_ij ← C^{k-1}_ij + C^{k-1}_ik · (C^{k-1}_kk)* · C^{k-1}_kj;
6.        for 1 ⩽ i, j ⩽ n do c(v_i, v_j) ← C^n_ij
        end
```
□

Теорема 5.5. *Алгоритм* 5.5 *использует* $O(n^3)$ *операций* $+$, $\cdot$ *и* $*$ *из полукольца и вычисляет* $c(v_i, v_j)$ *для* $1 \leqslant i, j \leqslant n$.

Доказательство. Легко проверить, что строка 5 выполняется n^3 раз, причем каждый раз используются четыре операции, и **for**-циклы в строках 1, 2 и 6 итерируются не более n^2 раз каждый. Поэтому достаточно $O(n^3)$ операций.

Чтобы доказать корректность, надо доказать индукцией по k, что C_{ij}^k — сумма меток всех путей из v_i в v_j, не содержащих промежуточных (т. е. отличных от концов) узлов с индексами, большими k. Базис, т. е. случай $k=0$, тривиален в силу строк 1 и 2, поскольку любой такой путь имеет длину 0 или состоит из единственного ребра. Шаг индукции получается на основании строки 5, ибо путь из v_i в v_j, не содержащий промежуточных узлов с индексами, большими k, либо

1) не содержит промежуточных узлов с индексами, большими k—1 (слагаемое C_{ij}^{k-1}), либо
2) идет из v_i в v_k, затем несколько раз (возможно, ни разу) из v_k в v_k и, наконец, из v_k в v_j, причем везде на этих участках нет промежуточных узлов с индексами, большими k—1 (слагаемое $C_{ik}^{k-1} \cdot (C_{kk}^{k-1})^* \cdot C_{kj}^{k-1}$).

Аксиомы замкнутого кольца гарантируют, что в строке 5 корректно вычисляется сумма меток всех этих путей. □

5.7. АЛГОРИТМ ТРАНЗИТИВНОГО ЗАМЫКАНИЯ

Изучим алгоритм 5.5 для двух интересных случаев. Первый случай — замкнутое полукольцо S_1, описанное в примере 5.9. В S_1 легко выполнить сложение и умножение, а также и операцию *, поскольку $0^*=1^*=1$. Действительно, так как 1 служит единичным элементом относительно ·, строку 5 алгоритма 5.5 можно заменить на [1])

$$C_{ij}^k \leftarrow C_{ij}^{k-1} + C_{ik}^{k-1} \cdot C_{kj}^{k-1}. \tag{5.4}$$

Чтобы вычислить рефлексивно-транзитивное замыкание графа, зададим функцию разметки

$$l(v, w) = \begin{cases} 1, & \text{если } (v, w) \text{ — ребро графа,} \\ 0 & \text{в противном случае.} \end{cases}$$

Стоимость $c(v, w)$ равна 1 тогда и только тогда, когда из v в w идет путь длины 0 или больше. Это легко проверить с помощью аксиом замкнутого полукольца $\{0, 1\}$.

Пример 5.12. Рассмотрим граф на рис. 5.18, временно игнорируя числа на ребрах. Функция разметки $l(v_i, v_j)$ задается таблицей

[1]) Естественно интерпретировать это так: «достаточно исследовать только пути без циклов». Тем не менее заметим, что с оператором (5.4) вместо строки 5 алгоритм 5.5 будет вычислять суммы по множествам путей, включающим не только все пути без циклов, но и некоторые другие.

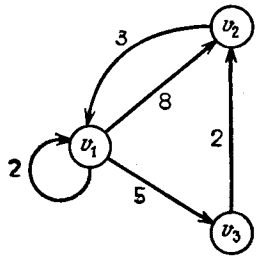


Рис. 5.18. Граф.

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	0	0
v_3	0	1	0

Строка 1 алгоритма 5.5 дает $C^0_{11}=C^0_{22}=C^0_{33}=1$. Строка 2 дает $C^0_{ij}=l(v_i, v_j)$ для $i\neq j$, так что получаем для C^0_{ij} таблицу

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	0
v_3	0	1	1

Теперь полагаем $k=1$ и выполняем цикл в строках 4 и 5, где вместо строки 5 стоит (5.4). Например, $C^1_{23}=C^0_{23}+C^0_{21}\cdot C^0_{13}=0+1\cdot 1=1$. Таблицы значений C^1_{ij}, C^2_{ij} и C^3_{ij} приведены на рис. 5.19. □

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	0	1	1

C^1_{ij}

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	1	1	1

$C^2_{ij}=C^3_{ij}=c(v_i, v_j)$

Рис. 5.19. Значения C^k_{ij}.

5.8. АЛГОРИТМ НАХОЖДЕНИЯ КРАТЧАЙШЕГО ПУТИ

Для вычисления кратчайшего пути воспользуемся вторым замкнутым полукольцом из примера 5.9, а именно множеством неотрицательных вещественных чисел вместе с $+\infty$. Напомним, что аддитивной операцией является MIN, а мультипликативной — сложение вещественных чисел. Иными словами, рассмотрим структуру $(R, \text{MIN}, +, +\infty, 0)$. В примере 5.10 отмечалось, что $a^* = 0$ для всех $a \in R$, так что снова можно обойтись без операции * в строке 5 алгоритма 5.5, заменив эту строку на

$$C_{ij}^k \leftarrow \text{MIN}\,(C_{ij}^{k-1},\ C_{ik}^{k-1} + C_{kj}^{k-1}). \tag{5.5}$$

Неформально можно сказать, что оператор (5.5) означает, что кратчайшим путем из v_i в v_j, не проходящим через узлы, индексы которых больше k, будет более короткий из следующих двух путей:

1) кратчайшего пути, не проходящего через узлы, индексы которых больше k—1, и
2) кратчайшего пути из v_i в v_k и затем в v_j, не проходящего через другие узлы, индексы которых больше k—1.

Чтобы превратить алгоритм 5.5 в алгоритм нахождения кратчайшего пути, зададим $l(v_i, v_j)$ как стоимость ребра (v_i, v_j), если

	v_1	v_2	v_3
v_1	2	8	5
v_2	3	∞	∞
v_3	∞	2	∞

$l(v_i, v_j)$

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	∞
v_3	∞	2	0

C_{ij}^0

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	8
v_3	∞	2	0

C_{ij}^1

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	8
v_3	5	2	0

C_{ij}^2

	v_1	v_2	v_3
v_1	0	7	5
v_2	3	0	8
v_3	5	2	0

$C_{ij}^3 = c(v_i, v_j)$

Рис. 5.20. Вычисление кратчайшего пути.

оно есть, и $+\infty$ в противном случае. Затем заменим строку 5 оператором (5.5), и по теореме 5.5 значение $c(v_i, v_j)$, выданное алгоритмом 5.5, будет наименьшей стоимостью (т. е. суммой стоимостей ребер) пути из всех путей между v_i и v_j.

Пример 5.13. Рассмотрим опять граф на рис. 5.18. Пусть каждое ребро помечено, как указано на рисунке. Тогда функции l, C^0_{ij}, C^1_{ij}, C^2_{ij} и C^3_{ij} примут значения, как в таблицах на рис. 5.20. Например, $C^3_{12}=\text{MIN}(C^2_{12},\ C^2_{12}+C^2_{32})=\text{MIN}(8,\ 5+2)=7$. □

5.9. ЗАДАЧИ О ПУТЯХ И УМНОЖЕНИЕ МАТРИЦ

Пусть $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо. Можно определить $(n\times n)$-матрицы элементов из S с обычными суммой и произведением. Иными словами, пусть матрицы A, B и C состоят из элементов a_{ij}, b_{ij} и c_{ij} соответственно для $1\leqslant i, j\leqslant n$. Тогда $C=A+B$ означает, что $c_{ij}=a_{ij}+b_{ij}$ для всех i и j, а $C=A\cdot B$ означает, что $c_{ij}=\sum_{k=1}^{n} a_{ik}\cdot b_{kj}$ для всех i и j. Легко доказать следующую лемму.

Лемма 5.10. *Пусть* $(S, +, \cdot, 0, 1)$ *— замкнутое полукольцо и* M_n *— множество* $(n\times n)$*-матриц над* S. *Пусть* $+_n$ *и* $\cdot_n$ *обозначают соответственно сложение и умножение матриц,* 0_n *— матрица, все элементы которой равны* 0, *а* I_n *— матрица с* 1 *на главной диагонали и* 0 *вне ее. Тогда* $(M_n, +_n, \cdot_n, 0_n, I_n)$ *— замкнутое полукольцо.*

Доказательство. Оставляем в качестве упражнения. □

Пусть $G=(V, E)$ — ориентированный граф, где $V=\{v_1, v_2, \ldots, v_n\}$. Пусть функция разметки l: $(V\times V)\to S$ определена, как в алгоритме 5.5, а A_G обозначает $(n\times n)$-матрицу с $a_{ij}=l(v_i, v_j)$. Следующая лемма связывает вычисление, выполняемое алгоритмом 5.5, с вычислением замыкания A^*_G матрицы A_G в полукольце M_n $(n\times n)$-матриц над S.

Лемма 5.11. *Пусть* G *и* A_G *таковы, как сказано выше. Тогда* (i, j)*-й элемент матрицы* A^*_G *равен* $c(v_i, v_j)$.

Доказательство. $A^*_G=\sum_{i=0}^{\infty} A^i_G$, где $A^0_G=I_n$ и $A^i_G=A_G\cdot A^{i-1}_G$ для $i\geqslant 1$. Индукцией по i легко показать, что (i, j)-й элемент матрицы A^k_G равен сумме стоимостей путей длины k, идущих из v_i в v_j. Отсюда непосредственно следует, что (i, j)-й элемент матрицы A^*_G равен сумме стоимостей всех путей из v_i в v_j. □

Из леммы 5.11 вытекает, что алгоритм 5.5 можно рассматривать как алгоритм вычисления замыкания матрицы. В теореме 5.5 утверждалось, что алгоритм 5.5 требует $O(n^3)$ *элементарных* операций

(т. е. операций $+$, $\cdot$ и $*$ над элементами полукольца). Обычный алгоритм умножения матриц также требует $O(n^3)$ элементарных операций. Мы покажем, что, когда элементы берутся из замкнутого полукольца, вычисление произведения двух матриц эквивалентно (по сложности) вычислению замыкания матрицы. Точнее, если дан какой-то алгоритм вычисления произведения матриц, то можно построить алгоритм вычисления замыкания, и обратно, причем порядок времени вычисления будет тем же. Значение этого результата лучше прояснится в гл. 6, где будет показано, что, когда элементы берутся из кольца, для умножения матриц хватает менее чем $O(n^3)$ операций. Рассматриваемая конструкция состоит из двух частей.

Теорема 5.6. *Если замыкание произвольной $(n\times n)$-матрицы над замкнутым полукольцом можно вычислить за такое время $T(n)$, что $T(3n)\leqslant 27T(n)$* [1]*, то существует алгоритм умножения двух $(n\times n)$-матриц A и B с временем работы $M(n)$, удовлетворяющим неравенству $M(n)\leqslant cT(n)$ для некоторой постоянной c.*

Доказательство. Пусть надо вычислить произведение $A\cdot B$ двух $(n\times n)$-матриц A и B. Сначала образуем $(3n\times 3n)$-матрицу

$$X=\begin{pmatrix}0 & A & 0\\ 0 & 0 & B\\ 0 & 0 & 0\end{pmatrix}$$

и найдем ее замыкание. Заметим, что

$$X^2=\begin{pmatrix}0 & 0 & A\cdot B\\ 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix} \quad \text{и} \quad X^3=X^4=\ldots=\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix}.$$

Следовательно,

$$X^*=I_{3n}+X+X^2=\begin{pmatrix}I_n & A & A\cdot B\\ 0 & I_n & B\\ 0 & 0 & I_n\end{pmatrix}.$$

Так как $A\cdot B$ можно прочитать в правом верхнем углу, заключаем отсюда, что $M(n)\leqslant T(3n)\leqslant 27T(n)$. □

Это доказательство теоремы 5.6 имеет следующую интерпретацию в виде графа. Рассмотрим граф G с $3n$ узлами $\{1, 2, \ldots, 3n\}$. Разобьем их на три множества $V_1=\{1, 2, \ldots, n\}$, $V_2=\{n+1, n+2, \ldots, 2n\}$ и $V_3=\{2n+1, 2n+2, \ldots, 3n\}$. Будем считать, что все ребра графа G идут либо из V_1 в V_2, либо из V_2 в V_3. Такой граф

[1]) Это допущение правдоподобно, поскольку $T(n)$ есть $O(n^3)$ в худшем случае. На самом деле в теореме подойдет любая постоянная (а не только 27).

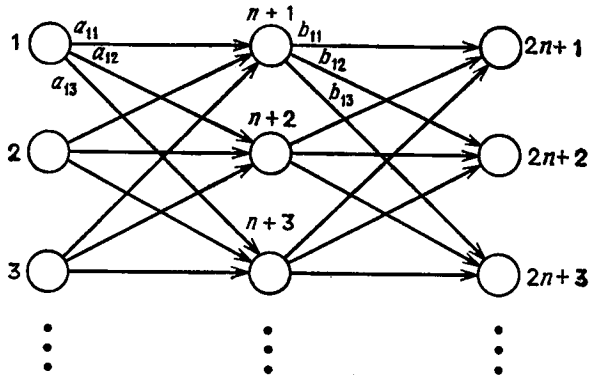


Рис. 5.21. Интерпретация теоремы 5.6 в виде графа.

изображен на рис. 5.21. Пусть A и B — такие $(n\times n)$-матрицы, что (i, j)-й элемент матрицы A служит меткой ребра, идущего из i в $n+j$, а (i, j)-й элемент матрицы B служит меткой ребра, идущего из $n+i$ в $2n+j$. Тогда (i, j)-й элемент произведения $A\cdot B$ оказывается суммой меток путей, идущих из i в $2n+j$, поскольку каждый такой путь образован ребром из i в некоторый узел k, $n<k\leqslant 2n$, и следующим за ним ребром из k в $2n+j$. Поэтому сумма меток всех путей из i в $2n+j$ совпадает с (i, j)-м элементом матрицы $A\cdot B$.

Теперь докажем обращение теоремы 5.6. По данному алгоритму умножения матриц можно найти алгоритм замыкания, время работы которого с точностью до постоянного множителя совпадает с временем работы данного алгоритма.

Теорема 5.7. *Если произведение любых двух $(n\times n)$-матриц над замкнутым полукольцом можно вычислить за такое время $M(n)$, что $M(2n)\geqslant 4M(n)$, то существует алгоритм, вычисляющий замыкание матрицы за время $T(n)$, удовлетворяющее неравенству $T(n)\leqslant cM(n)$ для некоторой постоянной c.*

Доказательство. Пусть X — $(n\times n)$-матрица. Сначала рассмотрим случай, когда n — степень числа 2, скажем 2^k. Разобьем X на четыре матрицы размера $2^{k-1}\times 2^{k-1}$:

$$X=\begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

В силу леммы 5.11 можно считать, что матрица X представляет граф $G=(V, E)$, в котором множество узлов разбито на два множества V_1 и V_2 размера 2^{k-1}. Матрица A представляет ребра между узлами множества V_1, матрица D — ребра между узлами из V_2, матрица B — ребра, идущие из узлов множества V_1 в узлы множества V_2, а мат-

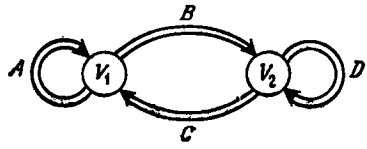


Рис. 5.22. Граф G.

рица C — ребра, идущие из узлов множества V_2 в узлы множества V_1. Это отражено на рис. 5.22.

Путь из v_1 в v_2, концы которого принадлежат V_1, либо

1) никогда не выходит за пределы V_1, либо
2) для некоторого $k \geqslant 1$ найдутся такие узлы w_i, x_i, y_i и z_i, где $1 \leqslant i \leqslant k$, что все w_i и z_i лежат в V_1, все x_i и y_i — в V_2, а рассматриваемый путь идет из v_1 в w_1 внутри V_1, затем по некоторому ребру уходит в x_1, потом вдоль некоторого пути внутри V_2 идет в узел y_1, после чего по некоторому ребру уходит в z_1, затем по пути внутри V_1 идет в w_2 и т. д. и, наконец, приходит в z_k, откуда вдоль пути внутри V_1 идет в v_2. Этот тип пути показан на рис. 5.23.

Сумма меток, приписанных путям, удовлетворяющим условию 1 или 2, очевидно, равна $(A+B \cdot D^* \cdot C)^*$. А именно, $B \cdot D^* \cdot C$ представляет пути, идущие из w_i в x_i, затем в y_i и далее в z_i на рис. 5.23.

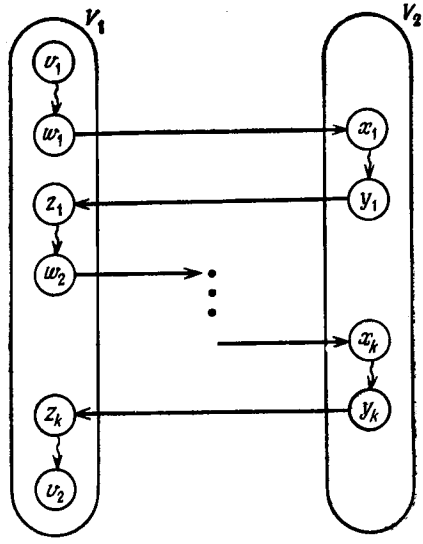


Рис. 5.23. Пути, удовлетворяющие условию 2.

Следовательно, $A+B\cdot D^*\cdot C$ представляет те пути, соединяющие узлы в V_1, которые либо состоят из одного ребра, либо прыгают прямо в V_2, остаются там на некоторое время и затем прыгают обратно в V_1. Все пути, соединяющие узлы из множества V_1, можно представить в виде последовательности путей, представленных матрицей $A+B\cdot D^*\cdot C$. Таким образом, $(A+B\cdot D^*\cdot C)^*$ представляет все пути, соединяющие узлы из V_1. Следовательно, верхний левый угол матрицы X^* занят матрицей $(A+B\cdot D^*\cdot C)^*$.

Обозначим $(A+B\cdot D^*\cdot C)^*$ через E. С помощью аналогичных рассуждений можно представить матрицу X^* в виде

$$X^*=\begin{pmatrix} E & E\cdot B\cdot D^* \\ D^*\cdot C\cdot E & D^*+D^*\cdot C\cdot E\cdot B\cdot D^* \end{pmatrix}=\begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Матрицы E, F, G и H можно вычислить, проделав такую последовательность шагов:

$$\begin{aligned} T_1&=D^*,\\ T_2&=B\cdot T_1,\\ E&=(A+T_2\cdot C)^*,\\ F&=E\cdot T_2,\\ T_3&=T_1\cdot C,\\ G&=T_3\cdot E,\\ H&=T_1+G\cdot T_2. \end{aligned}$$

Эти шаги требуют выполнения двух замыканий, шести умножений и двух сложений $(2^{k-1}\times 2^{k-1})$-матриц. Поэтому можно записать:

$$\begin{aligned} &T(1)=1,\\ &T(2^k)\leqslant 2T(2^{k-1})+6M(2^{k-1})+2\cdot 2^{2k-2},\quad k>1. \end{aligned} \tag{5.6}$$

Три слагаемых в правой части неравенства (5.6) — это сложности соответственно замыканий, умножений и сложений. Мы утверждаем, что найдутся такие постоянная c и функция T, что $T(2^k)\leqslant cM(2^k)$ и T удовлетворяет (5.6) для всех k. Базис, т. е. случай $k=0$, тривиален, поскольку постоянную c можно взять сколь угодно большой. Для проведения шага индукции предположим, что $T(2^{k-1})\leqslant cM(2^{k-1})$ для некоторой постоянной c. Из условия теоремы, т. е. из неравенства $M(2n)\geqslant 4M(n)$, заключаем, что $M(2^{k-1})\geqslant 2^{2k-2}$. (Заметим, что $M(1)=1$.) Следовательно, из (5.6) вытекает, что

$$T(2^k)\leqslant(2c+8)M(2^{k-1}).$$

Снова в силу условия теоремы $M(2^{k-1})\leqslant {}^1/_4 M(2^k)$, так что

$$T(2^k)\leqslant\left(\frac{1}{2}c+2\right)M(2^k). \tag{5.7}$$

Если взять $c \geqslant 4$, то (5.7) влечет за собой $T(2^k) \leqslant cM(2^k)$, а это и требовалось доказать.

Если n не является степенью числа 2, то можно вложить X в бóльшую матрицу, размер которой есть степень числа 2:

$$\begin{pmatrix} X & 0 \\ 0 & I \end{pmatrix},$$

где I — единичная матрица минимально возможного размера. Это вложение увеличит размер матрицы не более чем вдвое, и, значит, постоянная увеличится не более чем в 8 раз. Таким образом, найдется такая постоянная c', что $T(n) \leqslant c'M(n)$ для всех n независимо от того, являются ли они степенями числа 2 или нет. □

Следствие 1. *Время, необходимое для вычисления замыкания булевой матрицы, имеет тот же порядок, что и время вычисления произведения двух булевых матриц того же размера.*

В гл. 6 мы познакомимся с асимптотически более быстрыми алгоритмами вычисления произведения булевых матриц и тем самым покажем, что транзитивное замыкание графа можно вычислить за время, меньшее $O(n^3)$.

Следствие 2. *Время, необходимое для вычисления замыкания матрицы над множеством неотрицательных целых чисел с операциями* MIN *и* $+$ *в качестве сложения и умножения элементов, имеет тот же порядок, что и время вычисления произведения двух матриц такого же типа.*

В настоящее время все известные методы решения задачи о нахождении кратчайших путей для всех пар узлов тратят не меньше cn^3 времени для некоторой постоянной c.

5.10. ЗАДАЧИ С ОДНИМ ИСТОЧНИКОМ

Во многих приложениях достаточно находить кратчайшие пути только из одного узла *(источника)*. На самом деле нам, быть может, надо найти кратчайший путь между двумя конкретными узлами, однако неизвестен алгоритм, который решал бы эту задачу эффективнее (в худшем случае), чем лучший из известных алгоритмов для одного источника.

Для задач с одним источником мы не знаем объединяющего понятия, аналогичного замкнутым полукольцам и алгоритму 5.5. Если мы хотим только узнать, к каким узлам идут пути из источника, то задача тривиальна и ее можно решить несколькими алгоритмами, работающими $O(e)$ времени на e-реберном графе. Так как алгоритм, если не "просматривает" все ребра, не может быть

```
    begin
1.    S ← {v0};
2.    D[v0] ← 0;
3.    for v ∈ V − {v0} do D[v] ← l(v0, v);
4.    while S ≠ V do
        begin
5.        выбрать узел w ∈ V − S, для которого D[w] принимает
              наименьшее значение;
6.        добавить w к S;
7.        for v ∈ V − S do
8.          D[v] ← MIN(D[v], D[w] + l(w, v))
        end
    end
```

Рис. 5.24. Алгоритм Дейкстры.

корректным, нетрудно убедиться, что время $O(e)$ наилучшее, на что можно надеяться.

В задаче нахождения кратчайших путей из одного источника ситуация иная. Хотя нет причин предполагать, что для ее решения понадобится более $O(e)$ времени, ни один такой алгоритм не известен. Мы обсудим алгоритм сложности $O(n^2)$, работа которого основана на построении множества S узлов, кратчайшие расстояния до которых от источника известны. На каждом шаге к S добавляется тот из оставшихся узлов, скажем v, расстояние до которого от источника меньше всех других оставшихся узлов. Если стоимости всех ребер неотрицательны, то можно быть уверенным, что путь из источника в v проходит только через узлы из S. Поэтому достаточно найти для каждого узла v кратчайшее расстояние до него от источника вдоль пути, проходящего только через узлы из S. Изложим алгоритм формально.

Алгоритм 5.6. Кратчайший путь из одного источника (алгоритм Дейкстры)

Вход. Ориентированный граф $G=(V, E)$, источник $v_0 \in V$ и функция l, отображающая множество $E \times E$ в множество неотрицательных вещественных чисел. Полагаем $l(v_i, v_j)=+\infty$, если ребро (v_i, v_j), $v_i \neq v_j$, не принадлежит графу, и $l(v, v)=0$.

Выход. Для каждого $v \in V$ наименьшая сумма меток на ребрах из P, взятая по всем путям P, идущим из v_0 в v.

Метод. Строим такое множество $S \subseteq V$, что кратчайший путь из источника в каждый узел $v \in S$ целиком лежит в S. Массив $D[v]$ содержит стоимость текущего кратчайшего пути из v_0 в v,

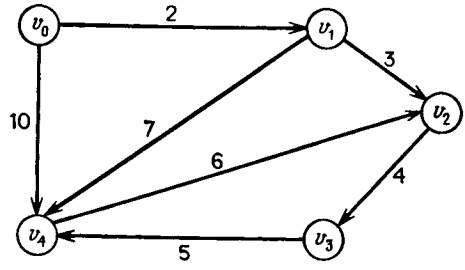


Рис. 5.25. Граф с помеченными ребрами.

который проходит только через узлы из S. Алгоритм приведен на рис. 5.24. □

Пример 5.14. Рассмотрим граф, изображенный на рис. 5.25. Вначале $S=\{v_0\}$, $D[v_0]=0$, а $D[v_i]=2, +\infty, +\infty, 10$ для $i=1, 2, 3, 4$ соответственно. При первой итерации цикла в строках 4—8 берем $w=v_1$, поскольку $D[v_1]=2$ — наименьшее значение для D. Затем вычисляем $D[v_2]=\text{MIN}(+\infty, 2+3)=5$ и $D[v_4]=\text{MIN}(10, 2+7)=9$. Последовательность значений для D и другие вычисления алгоритма 5.6 приведены на рис. 5.26. □

Теорема 5.8. *Алгоритм* 5.6 *вычисляет стоимость кратчайшего пути среди всех путей, идущих из* v_0 *в любой узел, и занимает* $O(n^2)$ *времени.*

Доказательство. **for**-цикл в строках 7, 8 требует $O(n)$ шагов, столько же шагов требует и выбор w в строке 5. В оценке сложности **while**-цикла в строках 4—8 эти процессы преобладают над остальными. **while**-цикл выполняется $O(n)$ раз, так что общая его сложность равна $O(n^2)$. Строки 1—3, очевидно, занимают $O(n)$ времени.

Итерация	S	w	$D[w]$	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$
Начальное значение	$\{v_0\}$	—	—	2	$+\infty$	$+\infty$	10
1	$\{v_0, v_1\}$	v_1	2	2	5	$+\infty$	9
2	$\{v_0, v_1, v_2\}$	v_2	5	2	5	9	9
3	$\{v_0, v_1, v_2, v_3\}$	v_3	9	2	5	9	9
4	Все узлы	v_4	9	2	5	9	9

Рис. 5.26. Вычисления алгоритма 5.6 на графе с рис. 5.25.

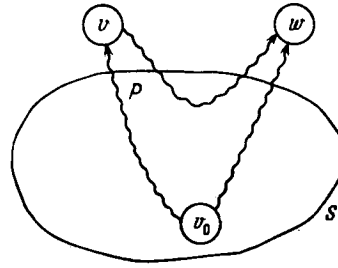


Рис. 5.27. Пути, идущие в узел v.

Чтобы показать корректность, надо доказать индукцией по размеру множества S, что для каждого ребра $v \in S$ число $D[v]$ равно длине кратчайшего пути из v_0 в v. Более того, для всех $v \in V-S$ число $D[v]$ равно длине кратчайшего пути из v_0 в v, лежащего целиком (если не считать сам узел v) в S.

Базис. Пусть $||S||=1$. Кратчайший путь из v_0 в себя имеет длину 0, а путь из v_0 в v, лежащий целиком (исключая v) в S, состоит из единственного ребра (v_0, v). Таким образом, строки 2 и 3 корректно формируют начальные значения массива D.

Шаг индукции. Пусть w — узел, выбранный в строке 5. Если число $D[w]$ не равно длине кратчайшего пути из v_0 в w, то должен быть более короткий путь P. Этот путь P должен содержать некоторый узел, отличный от w и не принадлежащий S. Пусть v — первый такой узел на P. Но тогда расстояние от v_0 до v меньше $D[w]$, а кратчайший путь в v целиком (исключая сам узел v) лежит в S (см. рис. 5.27). Следовательно, по предположению индукции $D[v]<D[w]$ в момент выбора w; пришли к противоречию. Отсюда заключаем, что такого пути P нет и $D[w]$ — длина кратчайшего пути из v_0 в w.

Второе утверждение (о том, что $D[w]$ остается корректным) очевидно ввиду строки 8. □

5.11. ДОМИНАТОРЫ В ОРИЕНТИРОВАННЫХ АЦИКЛИЧЕСКИХ ГРАФАХ: КОМБИНИРОВАНИЕ ПОНЯТИЙ

В этой главе мы уже познакомились с несколькими приемами, применяемыми для разработки эффективных алгоритмов, например с поиском в глубину и разумным упорядочением вычислений. В гл. 4 изучили много структур данных, полезных для представления множеств, над которыми выполняются различные операции. Эту главу мы закончим примером, иллюстрирующим, как можно строить эффективные алгоритмы, комбинируя структуры данных из гл. 4 с техникой для графов, развитой в данной главе. Мы будем

находить доминаторы в ориентированном ациклическом графе, используя, в частности, алгоритм нахождения MIN в свободном режиме, алгоритм объединения непересекающихся множеств и 2-3-деревья вместе с поиском в глубину.

Ориентированный граф $G=(V, E)$ называется *корневым* для (относительно) узла r, если существуют пути из r в каждый его узел. В остальной части раздела мы будем предполагать, что $G=(V, E)$ — корневой ориентированный ациклический граф с корнем r.

Узел v называется *доминатором* узла w, если любой путь из корня в w проходит через v. Каждый узел доминирует над самим собой, а корень доминирует над каждым узлом. Множество доминаторов узла w можно линейно упорядочить, расположив их в порядке вхождения в какой-нибудь кратчайший путь из корня в w. Доминатор узла w, ближайший к нему и отличный от него, называется его *непосредственным доминатором*. Так как множество доминаторов каждого узла линейно упорядочено, то отношение доминирования можно представить деревом с корнем r, называемым *доминаторным деревом* для G. Вычисление доминаторов полезно в задаче оптимизации кодов (Ахо, Ульман [1973], Хект [1973]).

Мы построим алгоритм сложности $O(e \log e)$, вычисляющий доминаторное дерево для корневого ориентированного ациклического графа с e ребрами. Основная наша цель — показать, как комбинировать технику этой главы и предыдущей. Алгоритм основан на трех леммах.

Пусть $G=(V, E)$ — граф и $G'=(V', E')$ — его подграф. Если $(a, b) \in E'$, будем писать $a \Rightarrow_{G'} b$. Через $\overset{+}{\Rightarrow}_{G'}$ и $\overset{*}{\Rightarrow}_{G'}$ будем обозначать соответственно транзитивное и рефлексивно-транзитивное замыкания отношения $\Rightarrow_{G'}$. Если $(a, b) \in E$, будем писать $a \Rightarrow b$.

Лемма 5.12. *Пусть $G=(V, E)$ — ориентированный ациклический граф с корнем r, а $S=(V, T)$ — остовное дерево для него с корнем r, построенное поиском в глубину. Пусть a, b, c и d — такие узлы из V, что $a \overset{*}{\Rightarrow}_S b \overset{+}{\Rightarrow}_S c \overset{*}{\Rightarrow}_S d$. Далее, пусть (a, c) и (b, d) — прямые ребра. Тогда замена прямого ребра (b, d) прямым ребром (a, d) не меняет доминаторов никакого узла в G* (см. рис. 5.28).

Доказательство. Обозначим через G' граф, полученный при замене ребра (b, d) в G ребром (a, d). Допустим, что v — доминатор узла w в G, но не в G'. Тогда в G' найдется путь P из r в w, не содержащий v. Очевидно, что этот путь должен идти по ребру (a, d), поскольку это единственное ребро в G', которого нет в G. Поэтому можно написать P: $r \Rightarrow^* a \Rightarrow d \Rightarrow^* w$. Отсюда следует, что узел v должен лежать на пути $a \overset{+}{\Rightarrow}_S d$ и что он отличен от a и d. Если $a<v \leqslant b$ при нумерации, порождаемой поиском в глубину, то замена ребра $a \Rightarrow d$ в P на путь $a \Rightarrow c \overset{*}{\Rightarrow}_S d$ дает путь в графе G,

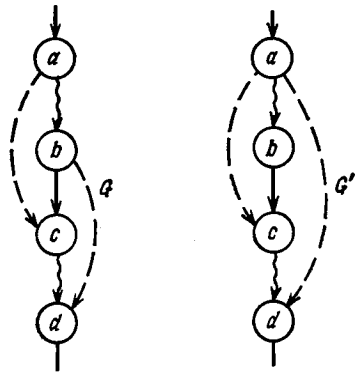


Рис. 5.28. Преобразование из леммы 5.12.

идущий из r в w и не содержащий v. Если $b<v<d$, то замена ребра $a \Rightarrow d$ в P на путь $a \Rightarrow_S^* b \Rightarrow d$ дает путь в графе G, идущий из r в w и не содержащий v. Так как $a<v \leqslant b$ или $b<v<d$, то найдется путь в G, идущий из r в w и не содержащий v; получили противоречие.

Допустим, что v — доминатор узла w в G', но не в G. Тогда в G найдется путь P из r в w, не содержащий v. Так как этот путь не проходит целиком в G', то он должен содержать ребро $b \Rightarrow d$. Следовательно, узел v лежит на пути $b \Rightarrow^* d$ и $b<v<d$. Таким образом, в G' есть путь $r \Rightarrow_S^* a \Rightarrow d \Rightarrow_S^* v$, не содержащий узел v; получили противоречие. □

Лемма 5.13. *Пусть G и S те же, что и в лемме* 5.12, $a \Rightarrow_S^* b$ *и (a, b) — прямое ребро графа G. Если в G нет прямого ребра (c,d), для которого $c<a$ и $a<d \leqslant b$, и поперечного ребра (e, d), для которого $a < d \leqslant b$, то удаление древесного ребра, входящего в b, не меняет доминаторов никакого узла в G* (см. рис. 5.29).

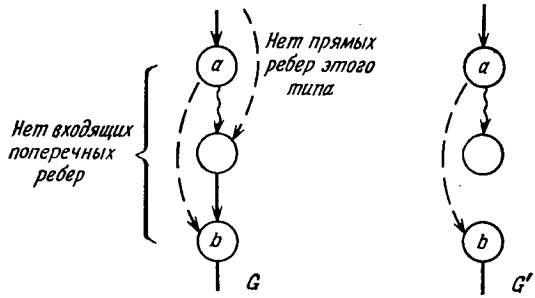


Рис. 5.29. Преобразование из леммы 5.13.

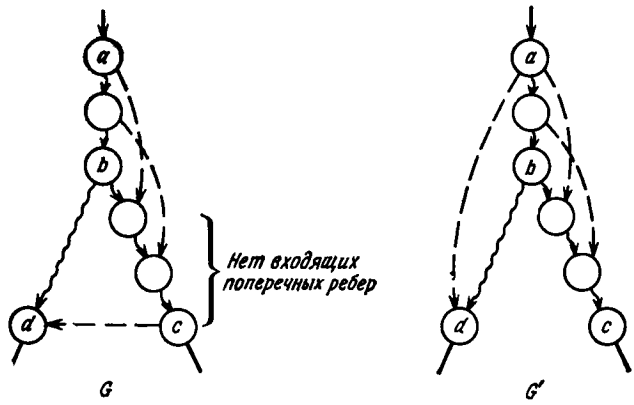


Рис. 5.30. Преобразование из леммы 5.14.

Доказательство. Доказательство аналогично доказательству леммы 5.12. □

Лемма 5.14. *Пусть G и S те же, что и в лемме* 5.12, *(c, d) — поперечное ребро графа G и b — общий предок для c и d с наибольшим номером. Пусть $a = \mathrm{MIN}(\{b\} \cup \{v \mid (v, w)$ — прямое ребро и $b < w \leqslant c\})$. Если ни в один узел пути $b \Rightarrow_S^* c$, отличный от b, не входит поперечное ребро, то замена поперечного ребра (c, d) прямым ребром (a, d) не меняет доминаторов никакого узла в G* (см. рис. 5.30).

Доказательство. Обозначим через G' граф, полученный при замене ребра (c, d) в G ребром (a, d). Допустим, что v — доминатор узла w в G, но не в G'. Тогда в G' найдется путь P из r в w, не содержащий v. Понятно, что этот путь должен идти по ребру (a, d). Следовательно, узел v должен лежать на пути $a \Rightarrow_S^* b$, проходящем по остовному дереву, ибо в противном случае замена ребра $a \Rightarrow d$ в P на $a \Rightarrow_S^* d$ или на $a \Rightarrow_S^* c \Rightarrow d$ дала бы путь в G, идущий из r в w и не содержащий v. Но тогда замена ребра $a \Rightarrow d$ в P на $a \Rightarrow u \Rightarrow_S^* c \Rightarrow d$, где u лежит на пути $b \Rightarrow_S^* c$ и $u > b$, дала бы путь в G, идущий из r в w и не содержащий v; получили противоречие.

Допустим, что v — доминатор узла w в G', но не в G. Тогда в G найдется путь P из r в w, не содержащий v. Понятно, что P содержит ребро (c, d). Запишем P в виде $r \Rightarrow_S^* c \Rightarrow d \Rightarrow_S^* w$. Путь $r \Rightarrow_S^* c$ должен содержать узел, лежащий на пути $a \Rightarrow_S^* b$, поскольку нет поперечных ребер, входящих в узлы на пути $b \Rightarrow_S^* c$ (исключая узел b). Пусть x — такой узел с наибольшим номером, а P_1 — участок пути P от r до x, за которым следует $x \Rightarrow_S^* d$, а за ним участок пути P от d до w. Пусть P_2 — путь $r \Rightarrow_S^* a \Rightarrow d$, за которым

следует участок пути P от d до w. Если узел v лежит на P_1, то он должен лежать и на $x \Rightarrow_S^* d$, причем $v > x$. Если он лежит на P_2, то должен лежать на $r \Rightarrow_S^* a$. Так как $a \leqslant x$, то один из этих путей в G' не содержит v; получили противоречие. □

Легко показать, что повторное применение преобразований из лемм 5.12—5.14 до тех пор, пока возможно, переводит граф G в дерево. Так как эти преобразования не меняют отношения доминирования, то окончательное дерево должно быть доминаторным для G. Это и будет нашим алгоритмом вычисления доминаторов графа G. Вся хитрость — в построении структуры данных, позволяющей эффективно находить подходящие ребра, к которым надо применять преобразования из лемм 5.12—5.14.

Говоря неформально, можно сказать, что мы строим доминаторное дерево для данного ориентированного ациклического графа $G = (V, E)$ следующим образом. Сначала поиском в глубину строим в G, отправляясь от корня, соответствующее остовное дерево $S = (V, T)$. Номера узлов графа G меняем на их номера, порожденные поиском в глубину. Затем применяем к G преобразования из лемм 5.12—5.14. Они выполняются двумя взаимосвязанными подпрограммами, одна из которых обрабатывает прямые ребра, а другая — поперечные.

1. Прямые ребра

Предположим на время, что в G нет поперечных ребер. Если в узел v входит более одного прямого ребра, то по лемме 5.12 все прямые ребра, входящие в v, за исключением одного с наименьшим началом, можно удалить, не изменив доминаторов ни одного узла.

Составным ребром назовем упорядоченную пару (t, H), где t — узел, называемый *началом* составного ребра, а H — множество узлов, называемое *концом* составного ребра. Составное ребро $(t, \{h_1, h_2, \ldots, h_k\})$ представляет множество ребер $(t, h_1), (t, h_2), \ldots, (t, h_k)$.

Многократно применяем лемму 5.12, чтобы изменить начала прямых ребер. В какой-то момент начало каждого ребра из множества ребер с общим началом t изменится на t'. Чтобы сделать это эффективно, представим одним составным ребром некоторые множества прямых ребер с общим началом. Вначале каждое прямое ребро (t, h) представляется составным ребром $(t, \{h\})$.

Каждому узлу v графа G поставим в соответствие множество $F[v]$. Это множество содержит такие пары $(t, \{h_1, h_2, \ldots, h_m\})$, что t — предок узла v, каждый узел h_i — потомок узла v и (t, h_i) — прямое ребро в G. Вначале $F[v] = \{(t, \{v\})\}$, где t — начало (с наименьшим номером) прямого ребра с концом v.

Затем проходим остовное дерево в порядке, обратном к прямому. Возвращаясь по ребру (v, w) остовного дерева, обнаруживаем,

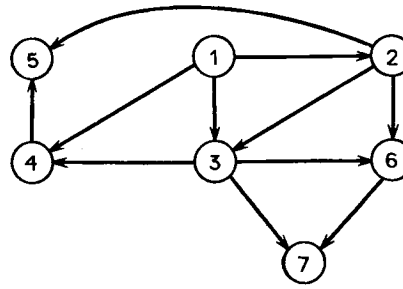


Рис. 5.31. Корневой ориентированный ациклический граф.

что множество $F[w]$ содержит составное ребро для каждого подлинного предка t узла w, который в данный момент является началом прямого ребра для некоторого потомка узла w. Далее делаем следующее.

1) Пусть $(t, \{h_1, h_2, \ldots, h_m\})$ — составное ребро в $F[w]$, имеющее начало с наибольшим номером. Если $t=v$, удаляем его из $F[w]$. (Это составное ребро представляет множество прямых ребер, начала которых перемещены в узел v, но больше не будут перемещаться под действием преобразования из леммы 5.12.) Удалим из G ребро остовного дерева с концом h_i, $1 \leqslant i \leqslant m$.
2) Если есть в G прямое ребро (t, v) [1]), то для каждого составного ребра $(u, \{h_1, \ldots, h_m\})$ из $F[w]$, для которого $u \geqslant t$,
   (а) удаляем $(u, \{h_1, \ldots, h_m\})$ из $F[w]$,
   (б) объединяем $\{h_1, \ldots, h_m\}$ с концом того составного ребра из $F[v]$, которое представляет среди прочих и ребро (t, v).
3) Заменяем $F[v]$ на $F[v] \cup F[w]$.

(Шаг 1 соответствует применению леммы 5.13, а шаг 2 — леммы 5.12.)

Пример 5.15. Рассмотрим корневой ориентированный ациклический граф G, изображенный на рис. 5.31. Поиск в глубину на графе G может породить граф, изображенный на рис. 5.32,*а*, где указаны также F-множества, поставленные в соответствие каждому узлу. На рис. 5.32,*б* — *г* приведены результаты преобразований прямых ребер. Окончательное доминаторное дерево изображено на рис. 5.32,*г*. □

Оставляем читателю доказать, что по достижении корня действительно получается доминаторное дерево (в предположении, что

[1]) Напомним, что мы предполагаем, что все прямые ребра с концом v, кроме имеющих начало с наименьшим номером, удалены из G.

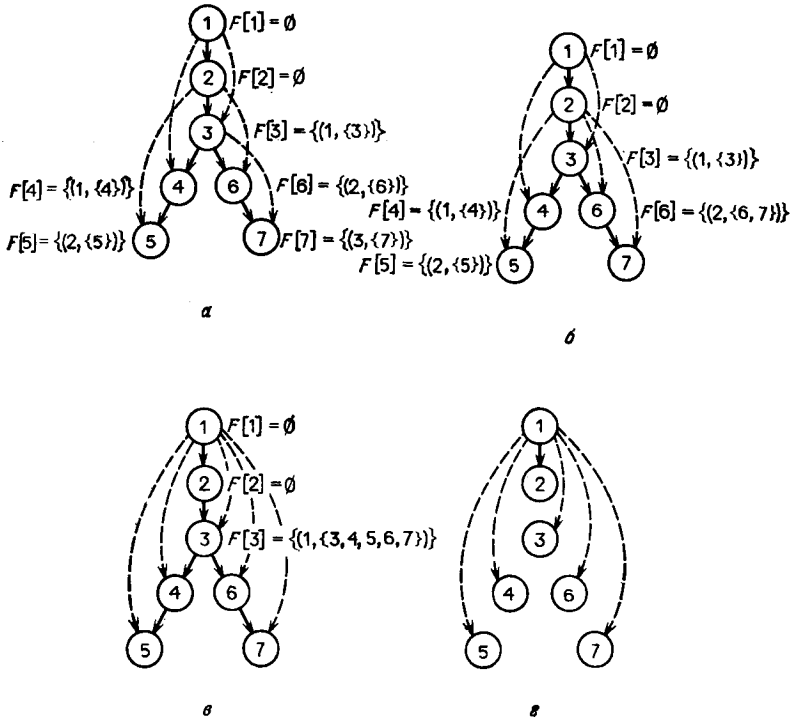


Рис. 5.32. Результаты преобразований прямых ребер: *а* — вначале; *б* — по возвращении вдоль ребра (6, 7) остовного дерева; *в* — по возвращении вдоль ребра (3, 4) остовного дерева; *г* — по возвращении вдоль ребра (1, 2) остовного дерева.

нет поперечных ребер). Этот алгоритм можно эффективно реализовать с помощью алгоритма объединения непересекающихся множеств и обрабатывать им множества концов у составных ребер. Множества $F[v]$ составных ребер можно представить 2-3-деревьями. Это связано с тем, что мы должны уметь эффективно удалять составные ребра, выделять в множестве составных ребер ребро с наибольшим началом и строить объединение множеств составных ребер. При такой структуре данных для обработки графа с e ребрами требуется время $O(e \log e)$.

2. *Поперечные ребра*

В общем случае нельзя предполагать, что поперечных ребер нет. Но можно с помощью метода, излагаемого ниже, заменить поперечные ребра прямыми. Однако эту замену не следует делать до работы на прямых ребрах, описанной в п. 1, ибо структура данных, ко-

торая строится в п. 1, помогает эффективно применить лемму 5.14. С другой стороны, не надо полностью выполнять п. 1 до устранения поперечных ребер, поскольку каждое устраненное поперечное ребро становится прямым. На самом деле мы должны добавить шаги обработки поперечных ребер к тому прохождению в порядке, обратном к прямому, которое было описано применительно к прямым ребрам. Заметим, что в п. 1 требуется (из-за применения леммы 5.13), чтобы в определенные моменты времени в определенные узлы не входили поперечные ребра. Поскольку прохождение ведется в порядке, обратном к прямому, шаги, описанные ниже, преобразуют поперечное ребро в прямое перед тем моментом, когда его наличие делало лемму 5.13 неприменимой.

Пусть S — глубинное остовное дерево для G. Вначале для каждого поперечного ребра (v, w) вычисляем общего предка узлов v и w с наибольшим номером. Каждому узлу v припишем множество $L[v]$ упорядоченных пар (u, w), где (u, w), $u>w$, представляет запрос о предке узлов u и w с наибольшим номером. Вначале $L[v]=\{(v, w) |$ есть поперечное ребро (v, w), $v>w\}$. Во время прохождения дерева S в соответствии с процедурой в п. 1 делаем следующее.

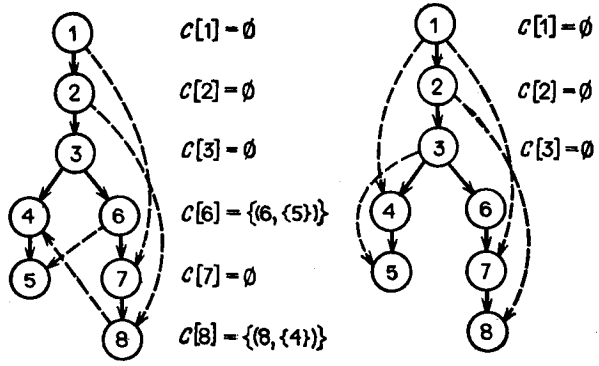
1) При прохождении древесного ребра (v, w), $v<w$, удаляем из $L[v]$ каждую пару (x, y), в которой $y \geqslant w$. Узел v — общий предок с наибольшим номером узлов x и y.
2) По возвращении к v по ребру (v, w) остовного дерева заменяем $L[v]$ на $L[v] \cup L[w]$.

Вычисление предков с наибольшим номером можно осуществить не более чем за $O(eG(e))$ шагов [1]), где e — число ребер графа G; для этого можно воспользоваться обобщением MIN-алгоритма, работающим в свободном режиме, о котором упоминается в упр. 4.21.

Обработка поперечных ребер состоит в том, что они преобразуются в прямые по лемме 5.14. Процесс преобразования поперечных ребер в прямые надо выполнять во время обработки прямых ребер. Каждому узлу v ставим в соответствие некоторое множество $C[v]$ составных ребер. Вначале $C[v]=\{(v, \{h_1, \ldots, h_m\}) | (v, h_i)$ — поперечное ребро при $1 \leqslant i \leqslant m\}$. По возвращении в узел v вдоль древесного ребра (v, w) совершаем помимо шагов, связанных с обработкой прямых ребер, следующие шаги.

1) Заменяем $C[v]$ на $C[v] \cup C[w]$.
2) Удаляем каждое поперечное ребро (x, y), для которого v — предок с наибольшим номером узлов x и y, из составного ребра, где оно представлено в данный момент. Если t — начало этого составного ребра, заменяем поперечное ребро (x, y)

[1]) См. примечание на стр. 202. — *Прим. ред.*



а *б*

Рис. 5.33. Удаление поперечных ребер: *а* — вначале; *б* — после рассмотрения ребра (3, 6).

прямым ребром (t, y). Если в y уже входит прямое ребро, оставляем только прямое ребро с тем началом, у которого номер меньше.

3) Пусть (u, v) — прямое ребро (если оно есть) с концом v. Если такого ребра нет, то пусть (u, v) — древесное ребро, входящее в v. После просмотра всех потомков узла v удаляем из $C[v]$ все составные ребра, начала которых лежат выше u. Объединяем множества концов удаленных составных ребер и образуем новое составное ребро с началом u. Добавляем новое составное ребро к $C[v]$.

Пример 5.16. Рассмотрим глубинное остовное дерево на рис. 5.33,*а*. Значения $C[v]$ приведены для обрабатываемых узлов. Так как из узла 2 в узел 8 идет прямое ребро, заменяем составное ребро (8, {4}) в $C[8]$ на (2, {4}). Затем присоединяем $C[8]$ к $C[7]$. Так как (1, 7) — прямое ребро, преобразуем составное ребро (2, {4}) в (1, {4}). По возвращении в узел 6 множество $C[6]$ станет равным {(1, {4}), (6, {5})}.

По возвращении из узла 6 в узел 3 множество $C[3]$ становится равным {(3, {5}), (1, {4})}. Узел 3 является предком с наибольшим номером узлов 6 и 5 и узлов 8 и 4. Поэтому удаляем из $C[3]$ составные ребра (3, {5}) и (1, {4}) и добавляем к G прямые ребра (3, 5) и (1, 4). Результат показан на рис. 5.33,*б*. Последующая часть поиска не приводит ни к каким изменениям. □

Составные поперечные ребра можно представить 2-3-деревьями. Предлагаем в качестве упражнения формальное описание доминаторного алгоритма. Если вы сможете найти подходящие структуры, вы освоили технику гл. 4 и 5.

УПРАЖНЕНИЯ

5.1. Найдите остовное дерево наименьшей стоимости для графа на рис. 5.34, если все нарисованные ребра неориентированны.

5.2. Пусть $S=(V, T)$ — остовное дерево наименьшей стоимости, построенное алгоритмом 5.1. Пусть $c_1 \leqslant c_2 \leqslant \ldots \leqslant c_n$ — стоимости ребер в T. Пусть S' — произвольное остовное дерево со стоимостями ребер $d_1 \leqslant d_2 \leqslant \ldots \leqslant d_n$. Покажите, что $c_i \leqslant d_i$ для $1 \leqslant i \leqslant n$.

****5.3.** Чтобы за время $O(e)$ построить остовное дерево наименьшей стоимости для графа с n узлами и e ребрами в предположении, что $e \geqslant f(n)$, где f — некоторая функция, которую вы должны найти, можно воспользоваться следующей стратегией. В различные моменты времени узлы будут группироваться в множества, связанные древесными ребрами, которые к тому времени уже найдены. Все ребра, инцидентные одному или двум узлам из множества, будут храниться в приоритетной очереди для этого множества. Вначале каждый узел находится в множестве, состоящем из него самого.

Итерируемый шаг состоит в том, чтобы найти наименьшее множество S и ребро с наименьшей стоимостью среди всех ребер, выходящих из S; пусть это ребро ведет в множество T. Затем добавляем его к остовному дереву и сливаем множества S и T. Но если все множества имеют размер не меньше $g(n)$, где g — другая функция, которую вам надо найти, то вместо этого образуем новый граф, где каждое множество представлено одним узлом. Узлы нового графа, соответствующие множествам S_1 и S_2, полагаются смежными, если некоторый узел в S_1 был смежен некоторому узлу в S_2 в исходном графе. Стоимостью ребра, соединяющего S_1 и S_2 в новом графе, считается наименьшая из стоимостей ребер, соединяющих узел из S_1 с узлом из S_2 в исходном графе. Далее этот алгоритм рекурсивно применяется к новому графу.

Ваша задача — выбрать $g(n)$ так, чтобы минимизировать $f(n)$.

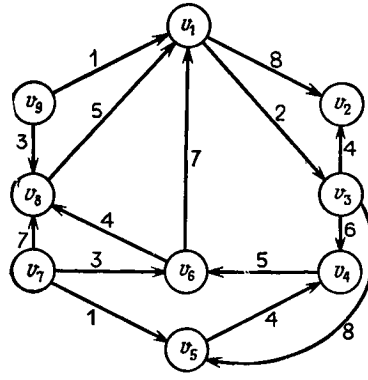


Рис. 5.34.

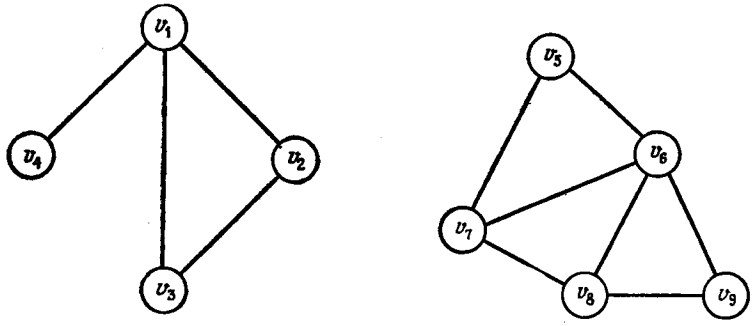


Рис. 5.35.

5.4. Постройте с помощью поиска в глубину остовный лес для неориентированного графа на рис. 5.35. Выбирайте в качестве исходного узла для построения очередного дерева любой узел. Найдите двусвязные компоненты этого графа.

5.5. Постройте с помощью поиска в глубину остовный лес для ориентированного графа на рис. 5.34. Затем найдите сильно связные компоненты.

5.6. Найдите двусвязные компоненты графа на рис. 5.36.

5.7. С помощью поиска в глубину разработайте эффективные алгоритмы, которые

(а) разбивают неориентированный граф на его связные компоненты,

(б) находят в неориентированном связном графе путь, проходящий через каждое ребро точно один раз в каждом направлении,

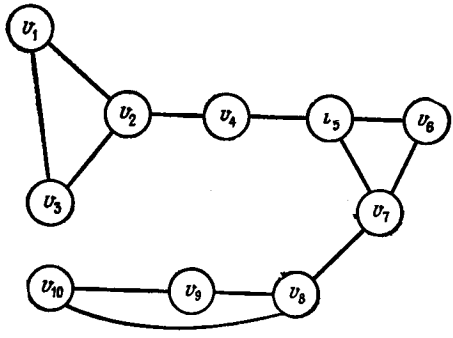


Рис. 5.36.

(в) проверяют, ацикличен ли ориентированный граф,
(г) находят такой порядок узлов ациклического ориентированного графа, при котором $v<w$, если из v в w ведет путь ненулевой длины,
(д) выясняют, можно ли так ориентировать ребра связного неориентированного графа, чтобы получить сильно связный ориентированный граф. *Указание*: Покажите, что это можно сделать тогда и только тогда, когда удаление из G любого ребра оставляет граф связным.

5.8. Пусть $G=(V, E)$ — *мультиграф* (т. е. ориентированный граф, у которого между двумя данными узлами может быть более одного ребра). Напишите алгоритм сложности $O(\|E\|)$, который удаляет каждый узел v степени 2 (заменяя ребра (u, v) и (v, w) ребром (u, w)) и уничтожает кратные ребра (заменяя их одним ребром). Заметим, что замена кратных ребер одним ребром может привести к образованию узла степени 2, который затем следует удалить. Аналогично удаление узла степени 2 может привести к образованию кратных ребер, которые затем надо уничтожить.

5.9. *Эйлеровым циклом* в неориентированном графе называется путь, который начинается и кончается в одном и том же узле и проходит по каждому ребру точно один раз. Связный неориентированный граф G имеет эйлеров цикл тогда и только тогда, когда степень каждого узла четна. Постройте алгоритм сложности $O(e)$ для нахождения эйлерова цикла в графе с e ребрами при условии, что такой цикл существует.

***5.10.** Пусть $G=(V, E)$ — двусвязный неориентированный граф и (v, w) — его ребро. Пусть $G'=(\{v, w\}, \{(v, w)\})$. Укажите способ выполнения в префиксном режиме последовательности операций НАЙТИ_ПУТЬ(s, t), где s — узел графа G', а (s, t) — ребро графа G, но не G'. НАЙТИ_ПУТЬ(s, t) выполняется так: находим простой путь в G, начинающийся с ребра (s, t) и кончающийся в некотором узле графа G', отличном от s, и добавляем узлы и ребра этого пути к G'. Время выполнения любой последовательности должно быть $O(\|E\|)$.

5.11. Для ориентированного графа G на рис. 5.37 найдите
(а) транзитивное замыкание,
(б) длину кратчайшего пути между каждой парой узлов (стоимости ребер приведены на рис. 5.37).

***5.12.** Найдите замкнутое полукольцо из четырех элементов.

***5.13.** *Транзитивная редукция* ориентированного графа $G=(V, E)$ определяется как произвольный граф $G'=(V, E')$ с минимально возможным числом узлов, транзитивное замыкание которого совпадает с транзитивным замыканием графа G. Покажите, что если граф G ацикличен, то его транзитивная редукция единственна.

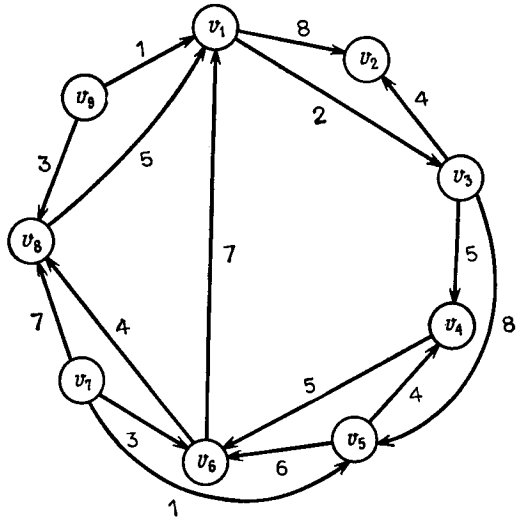


Рис. 5.37.

**5.14. Покажите, что время $R(n)$ вычисления транзитивной редукции ациклического графа с n узлами имеет тот же порядок, что и время $T(n)$ вычисления транзитивного замыкания, если принять (разумное) допущение, что $8R(n) \geqslant R(2n) \geqslant 4R(n)$ и $8T(n) \geqslant T(2n) \geqslant 4T(n)$.

*5.15. Покажите, что время вычисления транзитивной редукции ациклического графа имеет тот же порядок, что и время вычисления транзитивной редукции произвольного графа, если принять те же допущения, что и в упр. 5.14.

*5.16. Докажите то же, что и в упр. 5.15, для транзитивного замыкания.

*5.17. Пусть A будет $(n \times n)$-матрицей над замкнутым полукольцом $\{0, 1\}$. Не используя интерпретации на графах, докажите, что

(а) $A^* = I_n + A + A^2 + \ldots + A^n$,

(б) $\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^* + \begin{pmatrix} A^* & A^*BC^* \\ 0 & C^* \end{pmatrix}$.

Указание: Покажите, что

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^l = \begin{pmatrix} A^l & A^{l-1}B + A^{l-2}BC + \ldots + BC^{l-1} \\ 0 & C^l \end{pmatrix}.$$

*5.18. Покажите, что алгоритм нахождения кратчайшего пути из разд. 5.8 работает, когда некоторые ребра имеют отрицательную стоимость, но никакой цикл не имеет отрицательной стоимости. Что случится, если у какого-нибудь цикла будет отрицательная стоимость?

**5.19. Покажите, что положительные и отрицательные вещественные числа с $+\infty$ и $-\infty$ не образуют замкнутого полукольца. Как в этом свете объяснить упр. 5.18? *Указание:* Какие свойства замкнутого полукольца фактически используются в алгоритме 5.5?

5.20. Примените алгоритм 5.6 для нахождения кратчайшего расстояния от узла v_6 в каждый узел v графа G на рис. 5.37.

*5.21. Покажите, что задачу о нахождении кратчайшего пути из одного источника в случае неотрицательных стоимостей ребер для графа с e ребрами и n узлами можно решить за время $O(e \log n)$. *Указание:* Используйте подходящую структуру данных, чтобы строки 5 и 8 алгоритма 5.6 можно было эффективно выполнить при $e \ll n^2$.

**5.22. Покажите, что задачу о нахождении кратчайшего пути из одного источника в случае неотрицательных стоимостей ребер для графа с e ребрами и n узлами можно решить за время $O(ke + kn^{1+1/k})$ для любой фиксированной постоянной k.

```
begin
  S ← {v0};
  D[v0] ← 0;
  for v ∈ V − {v0} do D[v] ← l(v0, v);
  while S ≠ V do
    begin
      выбрать такой узел w ∈ V − S, что D[w] принимает наименьшее значение;
      S ← S ∪ {w};
      for v ∈ V, для которого D[v] > D[w] + l(w, v) do
        begin
          D[v] = D[w] + l(w, v);
          S ← S − {v}
        end
    end
end
```

Рис. 5.38. Алгоритм нахождения кратчайшего пути из одного источника.

	1	2	3	4	5
1	0	7	8	9	10
2	0	0	8	9	10
3	0	—2	0	9	10
4	0	—4	—3	0	10
5	0	—7	—6	—5	0

Рис. 5.39. Стоимости ребер для графа с 5 узлами.

*5.23. Докажите, что алгоритм нахождения кратчайшего пути, приведенный на рис. 5.38, строит кратчайший путь из v_0 в каждый узел v произвольного графа, у которого могут быть ребра с отрицательными стоимостями, но циклов с отрицательными стоимостями нет.

*5.24. Каков порядок времени работы алгоритма, приведенного на рис. 5.38? *Указание:* Рассмотрите работу алгоритма на графе с 5 узлами, стоимости ребер которого указаны на рис. 5.39.

5.25. Постройте алгоритм, который распознавал бы, есть ли в данном ориентированном графе, ребрам которого приписаны положительные и отрицательные стоимости, цикл отрицательной стоимости.

5.26. Измените правило выбора узла w в алгоритме на рис. 5.38 так, чтобы гарантировать оценку времени $O(n^3)$ в случае, когда ребрам приписаны произвольные стоимости.

5.27. Напишите алгоритм, который по данной $(n \times n)$-матрице M положительных целых чисел строил бы последовательность смежных элементов, начинающуюся с $M[n, 1]$ и оканчивающуюся $M[1, n]$, так, чтобы минимизировать сумму абсолютных значений разностей между соседними элементами. Два элемента $M[i, j]$ и $M[k, l]$ считаются *смежными*, если (а) $i=k\pm1$ и $j=l$ или (б) $i=k$ и $j=l\pm1$. Например, для рис. 5.40 решением будет последовательность 7, 5, 8, 7, 9, 6, 12.

5.28. Алгоритм на рис. 5.24 для каждого $v \in V$ вычисляет наименьшую стоимость пути среди всех путей из v_0 в v. Модифицируйте этот алгоритм так, чтобы он строил некоторый путь наименьшей стоимости для каждого узла v из V.

$$\begin{bmatrix} 1 & 9 & 6 & 12 \\ 8 & 7 & 3 & 5 \\ 5 & 9 & 11 & 4 \\ 7 & 3 & 2 & 6 \end{bmatrix}$$

Рис. 5.40. Матрица положительных целых чисел.

**5.29. Напишите программу, реализующую доминаторный алгоритм из разд. 5.11.

Проблемы для исследования

5.30. Для многих проблем, связанных с графами, поиск в глубину мог бы принести пользу. Одна из них касается *k-связности*. Неориентированный граф называется *k-связным*, если для каждой пары узлов v и w существуют такие k путей между ними, что ни один узел (кроме v и w) не входит более чем в один путь. Таким образом, двусвязность означает 2-связность. Хопкрофт, Тарьян [1973б] построили алгоритм, отыскивающий 3-связные компоненты за линейное время. Естественно выдвинуть гипотезу, что для каждого k есть алгоритм, отыскивающий k-связные компоненты за линейное (по числу узлов и ребер) время. Не сможете ли вы найти такой алгоритм?

5.31. Другой интересной проблемой, для решения которой, быть может, нужен поиск в глубину (а, быть может, и нет), является построение алгоритма, отыскивающего за линейное (по числу ребер) время остовное дерево наименьшей стоимости.

5.32. Заслуживает рассмотрения и задача нахождения кратчайшего пути из одного источника, когда $e \ll n^2$. Существует ли алгоритм, отыскивающий за время $O(e)$ хотя бы кратчайшее расстояние между двумя конкретными точками? Читателю было бы полезно ознакомиться с упр. 5.21 и 5.22, основанными на диссертации Джонсона [1973], а также с работой Спиры, Пана [1973], в которой показано, что в общем случае для таких алгоритмов требуется $n^2/4$ сравнений, если алгоритмы используют только сравнения между суммами стоимостей ребер. Вагнер [1974] применил технику сортировки вычерпыванием, чтобы получить алгоритм сложности $O(e)$ в случае, когда в качестве стоимостей ребер фигурируют малые целые числа.

5.33. Было показано (Керр [1972]), что если допускаются только операции MIN и +, то проблема нахождения кратчайших путей между всеми парами узлов требует kn^3 шагов, где k — некоторая положительная постоянная. Рабин усилил этот результат до $n^3/6$. Тем не менее может оказаться, что если допустить другие операции, то можно будет получить алгоритм, работающий менее $O(n^3)$ времени. Например, как мы увидим в следующей главе, можно построить транзитивное замыкание (или, что эквивалентно, перемножить две булевы матрицы) быстрее, чем за $O(n^3)$ шагов, если допустить операции, отличные от булевых.

Замечания по литературе

Сведения по теории графов можно почерпнуть из книг Бержа [1958] и Харари [1969]. Алгоритм 5.1, строящий остовные деревья наименьшей стоимости, взят из работы Крускала [1956]. Прим [1957] указывает другой подход к этой задаче. Алгоритм, предложенный в упр. 5.3, указал нам Спира.

Алгоритм, касающийся двусвязности и использующий поиск в глубину, принадлежит Хопкрофту. Алгоритм для сильно связных компонент взят у Тарьяна [1972]. В литературе отражены многочисленные применения поиска в глубину для построения оптимальных или наилучших из известных алгоритмов. Хопкрофт, Тарьян [1973а] дают линейную проверку планарности. В другой работе Хопкрофт, Тарьян [1973б] описывают линейный алгоритм нахождения 3-связных компонент. Тарьян [1973а] на основе той же идеи разработал наилучший из известных алгоритмов нахождения доминаторов. Кроме того, Тарьян [1973б] предлагает тест для «редуцируемости потоковых графов».

Алгоритм 5.5, т. е. общий алгоритм нахождения путей, по существу, принадлежит Клини [1956], который использовал его в связи с «регулярными выражениями» (разд. 9.1). Данная здесь форма этого алгоритма заимствована у Мак-Нотона, Ямады [1960]. Алгоритм сложности $O(n^3)$, строящий транзитивное замыкание, взят из работы Уоршола [1962]. Аналогичный алгоритм, отыскивающий кратчайшие пути для всех пар узлов, заимствован у Флойда [1962] (см. также Ху [1968]). Алгоритм для случая одного источника изложен по работе Дейкстры [1959]. Спира, Пан [1973] показывают, что алгоритм Дейкстры, по существу, оптимален, если в качестве модели брать деревья решений.

Данциг, Блаттнер, Рао [1967] заметили, что наличие ребер с отрицательными стоимостями не влияет на решение задачи нахождения кратчайших путей между всеми парами точек, если нет цикла с отрицательной стоимостью. Джонсон [1973] обсуждает задачи с одним источником при наличии ребер с отрицательными стоимостями; он же приводит решения упр. 5.21 и 5.22. Спира [1973] дает алгоритм нахождения кратчайшего пути за среднее время $O(n^2 \log^2 n)$.

Связь между задачей нахождения путей и умножением матриц описана в работах Мунро [1971] и Фурмана [1970] (теорема 5.7) и Фишера, Мейера [1971] (теорема 5.6). Связь с транзитивной редукцией (упр. 5.13—5.15) заимствована у Ахо, Гэри, Ульмана [1972]. Ивен [1973] обсуждает k-связность графов.

6 УМНОЖЕНИЕ МАТРИЦ И СВЯЗАННЫЕ С НИМ ОПЕРАЦИИ

В этой главе мы исследуем асимптотическую вычислительную сложность умножения матриц, элементы которых берутся из произвольного кольца. Мы увидим, что "обычный" алгоритм умножения матриц, имеющий сложность $O(n^3)$, можно асимптотически улучшить; чтобы умножить две $(n \times n)$-матрицы, достаточно времени $O(n^{2,81})$. Более того, мы увидим, что и другие операции, такие, как НВП-разложение, обращение матрицы и вычисление определителя, сводятся к умножению матриц, и потому их можно выполнить столь же быстро, как и умножение матриц. Затем покажем, что умножение матриц сводится к обращению матрицы, и потому любое улучшение асимптотического времени решения одной из этих задач привело бы к аналогичному улучшению для другой. Закончим главу двумя алгоритмами умножения булевых матриц, асимптотическая временная сложность которых меньше $O(n^3)$.

По существу к алгоритмам этой главы не следует относиться как к практическим, если исходить из технических возможностей современных вычислительных машин. Одна из причин состоит в том, что из-за скрытых постоянных множителей они работают быстрее обычных алгоритмов сложности $O(n^3)$ только для достаточно больших значений n. Кроме того, для этого семейства алгоритмов недостаточно хорошо понято поведение ошибок округления. Тем не менее идеи данной главы, на наш взгляд, заслуживают рассмотрения, поскольку они свидетельствуют, что очевидные алгоритмы не всегда наилучшие, а также потому, что могут служить основой разработки еще более эффективных и действительно практических алгоритмов для этого важного класса задач.

6.1. ОСНОВНЫЕ ПОНЯТИЯ

В настоящем разделе приводятся основные алгебраические понятия, полезные при изучении задач умножения матриц. Читатель, знакомый с кольцами и линейной алгеброй, может сразу перейти к разд. 6.2.

Определение. *Кольцом* называется алгебраическая структура $(S, +, \cdot, 0, 1)$, где S — множество элементов, а $+$ и $\cdot$ — бинарные операции на нем. Для каждых a, b и c из S выполняются следующие соотношения:

1) $(a+b)+c=a+(b+c)$ и $(a\cdot b)\cdot c=a\cdot(b\cdot c)$ (операции $+$ и $\cdot$ ассоциативны);
2) $a+b=b+a$ (операция $+$ коммутативна);
3) $(a+b)\cdot c=a\cdot c+b\cdot c$ и $a\cdot(b+c)=a\cdot b+a\cdot c$ (операция $\cdot$ дистрибутивна относительно $+$);
4) $a+0=0+a=a$ (0 служит единичным элементом для $+$);
5) $a\cdot 1=1\cdot a=a$ (1 служит единичным элементом для $\cdot$);
6) для каждого $a\in S$ существует *обратный* элемент $-a$, т. е. $a+(-a)=(-a)+a=0$.

Заметим, что последнее свойство — существование обратного элемента относительно сложения — в замкнутом полукольце может не выполняться (разд. 5.6). А свойство 4 замкнутого полукольца — существование и единственность бесконечных сумм — не обязательно выполняется в кольце. Кольцо, в котором операция $\cdot$ коммутативна, называется *коммутативным*.

Если в коммутативном кольце для каждого элемента a, отличного от 0, существует элемент a^{-1}, обратный относительно умножения, т. е. $a\cdot a^{-1}=a^{-1}\cdot a=1$, то кольцо называется *полем*.

Пример 6.1. Вещественные числа образуют кольцо, если взять в качестве $+$ и $\cdot$ обычные сложение и умножение. Вещественные числа, однако, не образуют замкнутого полукольца.

Система $(\{0, 1\}, +, \cdot, 0, 1)$, где $+$ — сложение по модулю 2 и $\cdot$ — обычное умножение, образует кольцо, но не замкнутое полукольцо, поскольку значение $1+1+\ldots$ не определено. Но если переопределить $+$ так, что $a+b=0$, если $a=b=0$, и $a+b=1$ в прочих случаях, то получится замкнутое полукольцо S_2 из упр. 5.1. S_2 не является кольцом, поскольку 1 не имеет обратного. □

Введем важный класс колец матриц.

Определение. Пусть $R=(S, +, \cdot, 0, 1)$ — кольцо и M_n — множество $(n\times n)$-матриц, элементы которых принадлежат R. Пусть O_n обозначает $(n\times n)$-матрицу, все элементы которой равны 0, и I_n — *единичную* $(n\times n)$-матрицу, у которой на главной диагонали стоят 1, а на остальных местах 0. Для A и B из M_n обозначим через $A+_n B$ такую $(n\times n)$-матрицу C, что $C[i, j]=A[i, j]+B[i, j]$ [1]), а через $A\cdot_n B$ — такую $(n\times n)$-матрицу D, что $D[i, j]=\sum_{k=1}^{n} A[i, k]\cdot B[k, j]$.

[1]) $M[i, j]$ — элемент, стоящий в матрице M на пересечении i-й строки и j-го столбца.

Лемма 6.1. $(M_n, +_n, \cdot_n, 0_n, I_n)$ — *кольцо.*

Доказательство. Элементарное упражнение. □

Заметим, что определенная выше операция умножения матриц $\cdot_n$ не коммутативна при $n>1$, даже если умножение в исходном кольце R коммутативно. Мы будем писать $+$ и $\cdot$ вместо $+_n$ и $\cdot_n$, если это не будет вызывать путаницы со сложением и умножением в исходном кольце R. Кроме того, мы будем часто опускать знак умножения, если он восстанавливается очевидным образом.

Пусть R — кольцо и M_n— кольцо $(n\times n)$-матриц с элементами из R. Допустим, что n четно. Тогда $(n\times n)$-матрицу M_n можно разбить на четыре $((n/2)\times(n/2))$-матрицы. Пусть $R_{2,n/2}$ — кольцо (2×2)-матриц с элементами из $M_{n/2}$. Непосредственно проверяется, что умножение и сложение $(n\times n)$-матриц в M_n эквивалентно умножению и сложению соответствующих (2×2)-матриц в $R_{2,n/2}$, элементами которых служат $((n/2)\times(n/2))$-матрицы.

Лемма 6.2. *Пусть f — такое отображение из M_n в $R_{2,n/2}$, что $f(A)$ — матрица*

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

из $R_{2,n/2}$, где A_{11}— левый верхний квадрант матрицы A, A_{12}— ее правый верхний квадрант, A_{21}— левый нижний, A_{22}— правый нижний. Тогда

$$f(A+B)=f(A)+f(B),$$
$$f(AB)=f(A)\cdot f(B).$$

Доказательство. Доказательство состоит в подстановке определяющих равенств для $+$ и $\cdot$ в $M_{n/2}$ в определяющие равенства для $+$ и $\cdot$ в $R_{2,n/2}$, а это элементарное упражнение. □

Лемма 6.2 важна тем, что указывает возможность построения алгоритма умножения $(n\times n)$-матриц из алгоритмов умножения (2×2)-матриц и $((n/2)\times(n/2))$-матриц. Мы воспользуемся ею в следующем разделе и построим асимптотически быстрый алгоритм умножения матриц. Здесь же подчеркнем тот факт, что алгоритм умножения (2×2)-матриц будет не произвольным, а специально ориентированным на работу с $R_{2,n/2}$. Так как кольцо $R_{2,n/2}$ не коммутативно, даже если таково кольцо R, то этот алгоритм умножения (2×2)-матриц не может предполагать коммутативности умножения $((n/2)\times(n/2))$-матриц. Но он может, разумеется, использовать любое из свойств кольца.

Определение. Пусть A будет $(n\times n)$-матрицей с элементами из некоторого поля. Матрицей A^{-1}, *обратной* к A, называется такая $(n\times n)$-матрица, что $AA^{-1}=I_n$.

Легко показать, что если матрица A^{-1} существует, то она единственна и $AA^{-1}=A^{-1}A=I_n$. Кроме того, $(AB)^{-1}=B^{-1}A^{-1}$.

Определение. Пусть A будет $(n\times n)$-матрицей. *Определителем* $\det(A)$ матрицы A называется сумма, взятая по всем перестановкам $p=(i_1, i_2, \ldots, i_n)$ целых чисел от 1 до n, произведений

$$(-1)^{k_p}\prod_{j=1}^{n} A[j, i_j],$$

где $k_p=0$, если перестановка p *четная* (т. е. ее можно получить из $(1, 2, \ldots, n)$ четным числом транспозиций), и $k_p=1$, если перестановка p *нечетная* (получается нечетным числом транспозиций).

Легко показать, что каждая перестановка либо четна, либо нечетна (но не то и другое одновременно).

Пример 6.2. Пусть

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Шесть перестановок чисел 1, 2, 3 таковы: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). Очевидно, перестановка (1, 2, 3) четна, ибо получается из себя с помощью 0 транспозиций. Перестановка (2, 3, 1) четна, поскольку, переставляя 2 и 3 в (1, 2, 3), получаем (1, 3, 2), затем переставляем 1 и 2 и получаем (2, 3, 1). Аналогично (3, 1, 2) — четная перестановка, а остальные три — нечетные. Таким образом, $\det(A)=a_{11}a_{22}a_{33}-a_{11}a_{23}a_{32}-a_{12}a_{21}a_{33}+a_{12}a_{23}a_{31}+a_{13}a_{21}a_{32}-a_{13}a_{22}a_{31}$. □

Пусть $(n\times n)$-матрица A образована элементами из некоторого поля. Можно показать, что A^{-1} существует тогда и только тогда, когда $\det(A)\neq 0$, и что $\det(AB)=\det(A)\det(B)$. В случае $\det(A)\neq 0$ матрицу A называют *невырожденной*.

Определение. $(m\times n)$-матрица A называется *верхней треугольной*, если $A[i, j]=0$ при $1\leqslant j<i\leqslant m$, и *нижней треугольной*, если $A[i, j]=0$ при $1\leqslant i<j\leqslant n$.

Умножение двух верхних треугольных матриц дает верхнюю треугольную матрицу. Аналогичное утверждение верно и для нижних треугольных матриц.

Лемма 6.3. *Если A — квадратная верхняя или нижняя треугольная матрица, то*

(а) $\det(A)$ *равняется произведению элементов, стоящих на главной диагонали* (*т. е.* $\prod_i A[i, i]$),

(б) *A невырожденна тогда и только тогда, когда все элементы на главной диагонали отличны от нуля.*

Доказательство. (а) Всякая перестановка $(i_1, i_2, \ldots, i_n)$, кроме $(1, 2, \ldots, n)$, содержит такую компоненту i_j, что $i_j<j$, и такую компоненту i_k, что $i_k>k$. Поэтому в сумме, определяющей $\det(A)$, все слагаемые, кроме того, которое соответствует перестановке $(1, 2, \ldots, n)$, равны 0. (б) непосредственно следует из (а). □

Определение. *Нормированной* называется матрица, у которой на главной диагонали стоят 1 (элементы вне главной диагонали произвольны).

Заметим, что определитель нормированной верхней треугольной или нижней треугольной матрицы равен 1.

Определение. *Матрицей перестановки* называется такая матрица из 0 и 1, что в каждой строке и каждом столбце стоит ровно одна единица.

Определение. *Подматрицей* матрицы A называется матрица, получаемая вычеркиванием некоторых строк и столбцов матрицы A. *Главной подматрицей* $(n\times n)$-матрицы A называется квадратная подматрица матрицы A, состоящая из первых k строк первых k столбцов матрицы A, $1\leqslant k\leqslant n$.

Рангом $\mathbf{rank}(A)$ матрицы A называется размер ее наибольшей квадратной невырожденной подматрицы. Например, ранг $(n\times n)$-матрицы перестановки равен n.

Заметим, что если $A=BC$, то $\mathbf{rank}(A)\leqslant \mathrm{MIN}(\mathbf{rank}(B), \mathbf{rank}(C))$. Кроме того, если A имеет m строк и ранг m, то любые ее k строк образуют матрицу ранга k.

Определение. Матрицей A^T, *транспонированной* к A, называют матрицу, получаемую перестановкой $A[i, j]$ и $A[j, i]$ для всех i и j.

6.2. АЛГОРИТМ ШТРАССЕНА ДЛЯ УМНОЖЕНИЯ МАТРИЦ

Пусть A и B — две $(n\times n)$-матрицы, причем n — степень числа 2. По лемме 6.2 можно разбить каждую матрицу A и B на четыре $((n/2)\times(n/2))$-матрицы и через них выразить произведение матриц A и B:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}=\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

где

$$\begin{aligned} C_{11} &= A_{11}B_{11}+A_{12}B_{21},\\ C_{12} &= A_{11}B_{12}+A_{12}B_{22},\\ C_{21} &= A_{21}B_{11}+A_{22}B_{21},\\ C_{22} &= A_{21}B_{12}+A_{22}B_{22}. \end{aligned} \tag{6.1}$$

Если рассматривать A и B как (2×2)-матрицы, элементами каждой из которых служат $((n/2)\times(n/2))$-матрицы, то их произведение можно выразить через суммы и произведения $((n/2)\times(n/2))$-матриц (формулы (6.1)). Допустим, что C_{ij} вычисляются с помощью m умножений и a сложений (или вычитаний) $((n/2)\times(n/2))$-матриц. Рекурсивно применяя этот алгоритм, можно вычислить произведение двух $(n\times n)$-матриц за время $T(n)$, удовлетворяющее неравенству

$$T(n)\leqslant mT\left(\frac{n}{2}\right)+\frac{an^2}{4},\quad n>2, \tag{6.2}$$

если n — степень числа 2. Первое слагаемое в правой части неравенства (6.2) — это сложность умножения m пар $((n/2)\times(n/2))$-матриц, а второе — сложность выполнения a сложений при условии, что каждое сложение или вычитание двух $((n/2)\times(n/2))$-матриц требует $n^2/4$ времени. Рассуждения, аналогичные проведенным в теореме 2.1 для $c=2$, показывают, что при $m>4$ решение неравенства (6.2) ограничено сверху величиной $kn^{\log m}$, где k — некоторая постоянная. Вид этого решения не зависит от a, т. е. от числа сложений. Таким образом, если $m<8$, то мы получаем метод, асимптотически лучший обычного метода сложности $O(n^3)$.

Штрассен изобрел искусный метод умножения двух (2×2)-матриц с элементами из произвольного кольца, в котором достаточно семи умножений. Рекурсивно применяя свой метод, он смог умножить две $(n\times n)$-матрицы за время $O(n^{\log 7})$, что по порядку примерно равно $n^{2{,}81}$.

Лемма 6.4. *Произведение двух* (2×2)*-матриц с элементами из произвольного кольца можно вычислить, выполнив* 7 *умножений и* 18 *сложений (вычитаний).*

Доказательство. Чтобы вычислить произведение матриц

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

сначала вычислим произведения

$$\begin{aligned}
m_1 &= (a_{12}-a_{22})(b_{21}+b_{22}),\\
m_2 &= (a_{11}+a_{22})(b_{11}+b_{22}),\\
m_3 &= (a_{11}-a_{21})(b_{11}+b_{12}),\\
m_4 &= (a_{11}+a_{12})b_{22},\\
m_5 &= a_{11}(b_{12}-b_{22}),\\
m_6 &= a_{22}(b_{21}-b_{11}),\\
m_7 &= (a_{21}+a_{22})b_{11}.
\end{aligned}$$

Затем вычисляем c_{ij} по формулам

$$c_{11} = m_1 + m_2 - m_4 + m_6,$$
$$c_{12} = m_4 + m_5,$$
$$c_{21} = m_6 + m_7,$$
$$c_{22} = m_2 - m_3 + m_5 - m_7.$$

Число операций подсчитывается тривиально. Доказательство того, что в результате получаются требуемые величины c_{ij}, представляет собой простое алгебраическое упражнение на использование аксиом кольца. □

В упр. 6.5 приведен способ вычисления произведения двух (2×2) матриц за 7 умножений и 15 сложений.

Теорема 6.1. *Две $(n\times n)$-матрицы с элементами из произвольного кольца можно перемножить за $O(n^{\log 7})$ арифметических операций.*

Доказательство. Сначала рассмотрим случай $n=2^k$. Пусть $T(n)$ — число арифметических операций, необходимых для умножения двух $(n\times n)$-матриц. По лемме 6.4

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \text{для} \quad n \geqslant 2.$$

Следовательно, в силу простой модификации теоремы 2.1 $T(n)$ составляет $O(7^{\log n})$, или $O(n^{\log 7})$.

Если n не является степенью числа 2, то каждую матрицу вложим в матрицу, порядок которой равен наименьшей степени числа 2, большей n. Это увеличит порядок матриц не более чем вдвое и, значит, постоянную увеличит не более чем в 7 раз. Таким образом, $T(n)$ есть $O(n^{\log 7})$ для всех $n\geqslant 1$. □

В теореме 6.1 нас интересовал только порядок роста функции $T(n)$. Но для того, чтобы выяснить, для каких значений n алгоритм Штрассена работает быстрее обычного алгоритма, надо найти соответствующую мультипликативную постоянную. Однако если n не является степенью числа 2, то простое вложение каждой исходной матрицы в матрицу, порядок которой равен степени числа 2, ближайшей к n сверху, даст слишком большую постоянную. Вместо этого можно вложить каждую матрицу в матрицу порядка $2^p r$ для некоторого небольшого числа r и p раз применить лемму 6.4, умножая $(r\times r)$-матрицы обычным $O(r^3)$-методом. Или, действуя иначе, можно было бы написать более общий рекурсивный алгоритм, который при четном n расщеплял бы каждую матрицу на четыре подматрицы, как и раньше, а при нечетном n сначала увеличивал бы порядок матриц на 1.

Следует подчеркнуть, что, вычислив мультипликативную постоянную, мы найдем границу только на число арифметических операций. Чтобы сравнить метод Штрассена с обычным методом умножения матриц, надо также учесть дополнительную сложность процедур, связанных с поиском элементов матриц.

6.3. ОБРАЩЕНИЕ МАТРИЦ

В данном разделе мы покажем, что задачу обращения матриц из определенного класса можно свести к задаче умножения матриц. Этот класс включает все обратимые треугольные матрицы, но не все обратимые матрицы вообще. В дальнейшем мы обобщим полученный результат на произвольные обратимые матрицы. В данном разделе и в разд. 6.4 и 6.5 рассматриваются матрицы с элементами из некоторого поля.

Лемма 6.5. *Пусть матрица A разбита так:*

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Предположим, что существует A_{11}^{-1}. Обозначим $A_{22}-A_{21}A_{11}^{-1}A_{12}$ через Δ и допустим, что Δ^{-1} существует. Тогда

$$A^{-1}=\begin{bmatrix} A_{11}^{-1}+A_{11}^{-1}A_{12}\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \qquad (6.3)$$

Доказательство. С помощью очевидных алгебраических преобразований можно показать, что

$$A=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}=\begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix}\begin{bmatrix} A_{11} & 0 \\ 0 & \Delta \end{bmatrix}\begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix},$$

откуда

$$A^{-1}=\begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix}\begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & \Delta^{-1} \end{bmatrix}\begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix}=$$

$$=\begin{bmatrix} A_{11}^{-1}+A_{11}^{-1}A_{12}\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \quad \square$$

Лемму 6.5 нельзя применять ко всем невырожденным матрицам. Например, $(n\times n)$-матрица перестановки A, у которой $A[i, j]=1$, если $j=n-i+1$, и $A[i, j]=0$ в противном случае, невырожденна, но $\det(A_{11})=0$ для любой главной подматрицы A_{11}. Тем не менее лемма 6.5 применима ко всем невырожденным нижним и верхним треугольным матрицам.

Лемма 6.6. *Если A — невырожденная верхняя (нижняя) треугольная матрица, то матрицы A_{11} и Δ из леммы* 6.5 *обратимы и являются верхними (нижними) треугольными.*

Доказательство. Допустим, что A — верхняя треугольная матрица. Для случая нижней треугольной матрицы доказательство аналогично. Очевидно, что A_{11}— невырожденная верхняя треугольная матрица и, значит, A_{11}^{-1} существует. Далее заметим, что $A_{21}=0$. Поэтому $\Delta=A_{22}-A_{21}A_{11}^{-1}A_{12}=A_{22}$— невырожденная верхняя треугольная матрица. □

Теорема 6.2. *Пусть $M(n)$ — время, требуемое для умножения двух $(n\times n)$-матриц. Если $8M(n)\geqslant M(2n)\geqslant 4M(n)$ для всех n, то найдется такая постоянная c, что обращение любой невырожденной верхней (нижней) треугольной $(n\times n)$-матрицы A можно вычислить за время $cM(n)$.*

Доказательство. Докажем теорему для случая, когда n — степень числа 2. Очевидно, что в противном случае можно вложить A в матрицу вида

$$\begin{bmatrix} A & 0 \\ 0 & I_m \end{bmatrix},$$

где $m+n(\leqslant 2n)$ является степенью числа 2. Тем самым, увеличив постоянную c не более чем в 8 раз, мы установим наш результат для произвольного n [1]).

Если n — степень числа 2, можно разбить A на четыре $((n/2)\times\times(n/2))$-подматрицы и рекурсивно применить формулу (6.3). Заметим, что $A_{21}=0$, так что $\Delta=A_{22}$. Следовательно, для обращения треугольных матриц A_{11} и Δ требуется $2T(n/2)$ времени, для двух нетривиальных умножений еще $2M(n/2)$ времени и, для того чтобы поставить минус в правом верхнем углу, еще $n^2/4$ операций. Из условия теоремы и неравенства $M(1)\geqslant 1$ выводим, что $n^2/4\leqslant M(n/2)$. Таким образом,

$$T(1)=1,$$
$$T(n)\leqslant 2T\left(\frac{n}{2}\right)+3M\left(\frac{n}{2}\right) \quad \text{для} \quad n\geqslant 2. \tag{6.4}$$

Легко доказать, что из (6.4) следует $T(n)\leqslant 3M(n)/2$. □

6.4. НВП-РАЗЛОЖЕНИЕ МАТРИЦЫ

Один из эффективных методов решения системы линейных уравнений состоит в применении так называемого НВП-разложения.

[1]) И опять более тщательный анализ дает для произвольного n постоянную c, несильно отличающуюся от наилучшей известной постоянной для случая, когда n — степень числа 2.

Определение. НВ-*разложением* [1]) $(m\times n)$-матрицы A, $m\leqslant n$, называется представление ее в виде $A=LU$, где L — нормированная нижняя треугольная $(m\times m)$-матрица, а U — верхняя треугольная $(m\times n)$-матрица.

Уравнение $A\mathbf{x}=\mathbf{b}$, где A — это $(n\times n)$-матрица, x — n-мерный вектор-столбец неизвестных, а $\mathbf{b}$ — n-мерный вектор-столбец, можно решить, сначала НВ-разложив A в произведение LU в предположении, что это возможно. Затем представим $A\mathbf{x}=\mathbf{b}$ в виде $LU\mathbf{x}=\mathbf{b}$. Чтобы получить решение $\mathbf{x}$, решим сначала $L\mathbf{y}=\mathbf{b}$ относительно $\mathbf{y}$, а затем $U\mathbf{x}=\mathbf{y}$ относительно $\mathbf{x}$.

Трудность применения этого метода заключается в том, что у матрицы A может не быть НВ-разложения, даже если она невырожденна. Например, матрица

$$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$

невырожденна, но у нее нет НВ-разложения. Однако если матрица A невырожденна, то найдется такая матрица перестановки P, что AP^{-1} имеет НВ-разложение. Изложим алгоритм, который по любой невырожденной матрице A находит такие матрицы L, U и P, что $A=LUP$. Матрицы L, U и P образуют НВП-*разложение* [2]) матрицы A.

Алгоритм 6.1. НВП-разложение

Вход. Невырожденная $(n\times n)$-матрица M, n — степень числа 2.

Выход. Матрицы L, U и P, для которых $M=LUP$, причем L — нормированная нижняя треугольная матрица, U — верхняя треугольная и P — матрица перестановки.

Метод. Вызываем процедуру МНОЖИТЕЛЬ(M, n, n), где МНОЖИТЕЛЬ — рекурсивная процедура, показанная на рис. 6.4. При описании этой процедуры используются диаграммы на рис. 6.1—6.3, где затемненная область матрицы представляет ту ее часть, о которой известно, что она состоит из одних нулей.

Каждый рекурсивный вызов процедуры МНОЖИТЕЛЬ происходит на $(m\times p)$-подматрице A $(n\times n)$-матрицы M. При каждом вызове m есть степень числа 2 и $m\leqslant p\leqslant n$. Выходом этой процедуры являются три матрицы L, U и P, показанные на рис. 6.1. □

Пример 6.3. Найдем НВП-разложение матрицы

$$M=\begin{bmatrix}0 & 0 & 0 & 1\\ 0 & 0 & 2 & 0\\ 0 & 3 & 0 & 0\\ 4 & 0 & 0 & 0\end{bmatrix}.$$

[1]) По первым буквам слов «нижняя» и «верхняя».— *Прим. перев.*

[2]) По первым буквам слов «нижняя», «верхняя» и «перестановка».— *Прим. перев.*

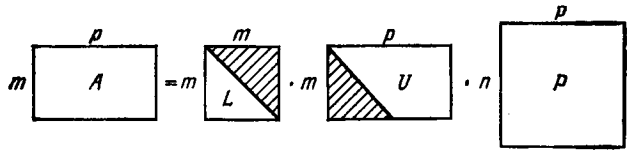


Рис. 6.1. Выход процедуры МНОЖИТЕЛЬ.

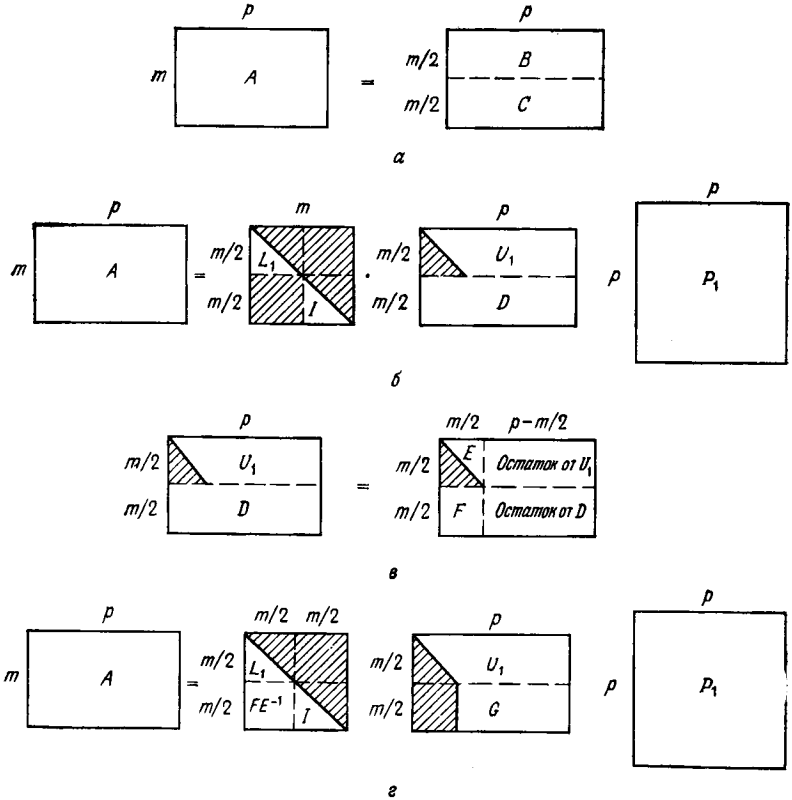


Рис. 6.2. Шаги процедуры МНОЖИТЕЛЬ: *а* — начальное разбиение матрицы A; *б* — разложение матрицы A после первого вызова процедуры; *в* — разбиение матриц U_1 и D; *г* — обнуление нижнего левого угла матрицы D. (Отметим, что I можно рассматривать как нормированную нижнюю (или верхнюю) треугольную матрицу.)

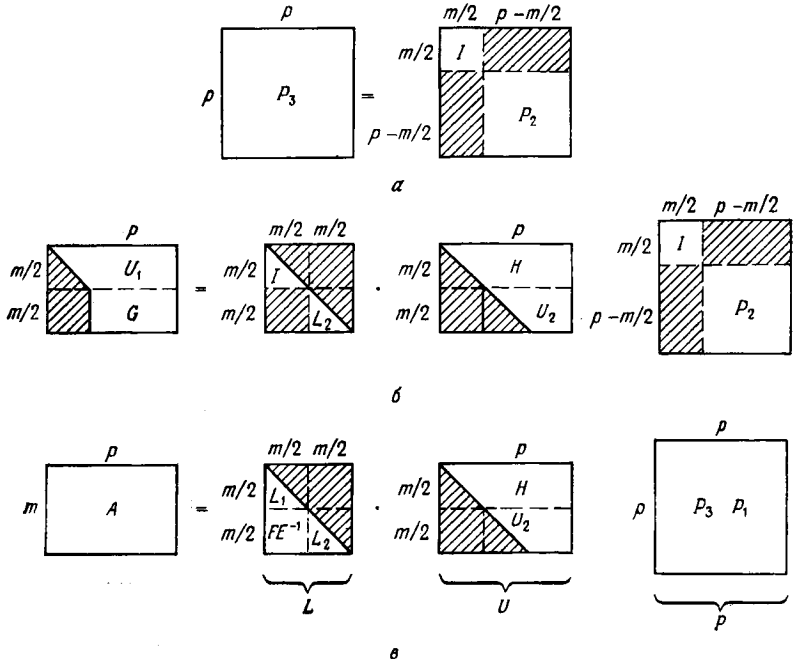


Рис. 6.3. Завершение процедуры МНОЖИТЕЛЬ: *а* — построение матрицы P_3; *б* — разложение матриц U_1 и G; *в* — разложение матрицы A.

Начнем с вызова процедуры МНОЖИТЕЛЬ (M, 4, 4), которая сразу же вызывает

$$\text{МНОЖИТЕЛЬ}\left(\left[\begin{matrix}0&0&0&1\\0&0&2&0\end{matrix}\right], 2, 4\right).$$

Взяв в качестве матрицы A первый аргумент этого вызова, вызываем МНОЖИТЕЛЬ ([0 0 0 1], 1, 4). В результате получаем

$$L_1=[1], \quad U_1=[1\ \ 0\ \ 0\ \ 0], \quad P_1=\left[\begin{matrix}0&0&0&1\\0&1&0&0\\0&0&1&0\\1&0&0&0\end{matrix}\right];$$

последняя матрица переставляет столбцы 1 и 4.

procedure МНОЖИТЕЛЬ(A, m, p):
if $m=1$ **then**
begin
1. пусть $L=[1]$ (т. е. L — нормированная (1×1)-матрица;
2. найти, если можно, столбец c матрицы A с ненулевым элементом, и пусть P будет $(p\times p)$-матрицей, переставляющей столбцы 1 и c;
comment Заметьте, что $P=P^{-1}$;
3. пусть $U=AP$;
4. **return** (L, U, P)
end
else
begin
5. разбить A на $((m/2)\times p)$-матрицы B и C, как показано на рис. 6.2,*а*;
6. вызвать МНОЖИТЕЛЬ(B,$m/2$, p), чтобы получить L_1, U_1, P_1;
7. вычислить $D=CP_1^{-1}$;
comment В данный момент A можно записать как произведение трех матриц, показанных на рис. 6.2,*б*;
8. пусть E и F — первые $m/2$ столбцов соответственно матриц U_1 и D (рис. 6.2,*в*);
9. вычислить $G=D-FE^{-1}U_1$;
comment Заметьте, что первые $m/2$ столбцов матрицы G состоят из одних нулей. Матрицу A можно записать в виде произведения матриц, показанных на рис. 6.2,*г*;
10. пусть G' — самые правые $p-m/2$ столбцов матрицы G;
11. вызвать МНОЖИТЕЛЬ(G', $m/2$, $p-m/2$) и получить L_2, U_2 и P_2;
12. пусть P_3 будет $(p\times p)$-матрицей перестановки, у которой в левом верхнем углу стоит $I_{m/2}$, а в правом нижнем P_2 (рис. 6.3,*а*);
13. вычислить $H=U_1P_3^{-1}$;
comment В это время матрицу, составленную из U_1 и U, можно записать так, как показано на рис. 6.3,*б*. Если в рис. 6.2,*г* подставить правую часть равенства на рис. 6.3,*б*, то получится представление

матрицы A в виде произведения пяти матриц. Первые две из них — нормированные нижние треугольные, третья — верхняя треугольная, а последние две — матрицы перестановок. Умножим первые две и последние две, чтобы получить искомое разложение матрицы A;

14. пусть L — это $(m \times m)$-матрица, состоящая из L_1, $O_{m/2}$, FE^{-1} и L_2 (рис. 6.3,*в*);
15. пусть U — это $(m \times p)$-матрица, у которой в верхней части стоит H, а в нижней $O_{m/2}$ и U_2 (рис. 6.3,*в*);
16. пусть P — произведение P_3P_1;
17. **return** (L, U, P)

end

Рис. 6.4. Процедура МНОЖИТЕЛЬ.

В строке 7 вычисляем [1])

$$D = CP_1^{-1} = [0\ 0\ 2\ 0]\begin{bmatrix}0&0&0&1\\0&1&0&0\\0&0&1&0\\1&0&0&0\end{bmatrix} = [0\ 0\ 2\ 0].$$

В строке 8 имеем $E=[1]$ и $F=[0]$, так что после выполнения строки 9 $G=D=[0\ 0\ 2\ 0]$. Строка 10 дает $G' = [0\ 2\ 0]$; следовательно, после строки 11 будет

$$L_2 = [1], \quad U_2 = [2\ 0\ 0], \qquad P_2 = \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}.$$

В строке 12

$$P_3 = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix},$$

а в строке 13

$$H = U_1P_3^{-1} = [1\ 0\ 0\ 0]\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix} = [1\ 0\ 0\ 0].$$

[1]) В этом примере все матрицы перестановок оказываются равными своим обратным.

Таким образом, МНОЖИТЕЛЬ $\left(\begin{bmatrix}0&0&0&1\\0&0&2&0\end{bmatrix}, 2, 4\right)$ выдает

$$L=\begin{bmatrix}1&0\\0&1\end{bmatrix}, \quad U=\begin{bmatrix}1&0&0&0\\0&2&0&0\end{bmatrix}, \qquad P=P_3P_1=\begin{bmatrix}0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0\end{bmatrix}.$$

Теперь возвращаемся к вызову МНОЖИТЕЛЬ$(M, 4, 4)$ в строке 6, причем роль L_1, U_1 и P_1 играют соответственно L, U и P. В строке 7 вычисляем

$$D=CP_1^{-1}=\begin{bmatrix}0&3&0&0\\4&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0\end{bmatrix}=\begin{bmatrix}0&0&3&0\\0&0&0&4\end{bmatrix},$$

в строке 8

$$E=\begin{bmatrix}1&0\\0&2\end{bmatrix}, \qquad F=\begin{bmatrix}0&0\\0&0\end{bmatrix},$$

так что после выполнения строки 9

$$G=D=\begin{bmatrix}0&0&3&0\\0&0&0&4\end{bmatrix}$$

и в строке 10

$$G'=\begin{bmatrix}3&0\\0&4\end{bmatrix}.$$

Предлагаем читателю проверить, что МНОЖИТЕЛЬ$(G', 2, 2)$ выдает

$$L_2=\begin{bmatrix}1&0\\0&1\end{bmatrix}, \quad U_2=\begin{bmatrix}3&0\\0&4\end{bmatrix}, \quad P_2=\begin{bmatrix}1&0\\0&1\end{bmatrix}.$$

Таким образом, в строке 12 $P_3=I_4$, а в строке 13

$$H=U_1P_3^{-1}=U_1=\begin{bmatrix}1&0&0&0\\0&2&0&0\end{bmatrix}.$$

Следовательно, в строках 14—16 вычисляем

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\end{bmatrix}, \quad \begin{bmatrix}1&0&0&0\\0&2&0&0\end{bmatrix}, \quad \begin{bmatrix}0&0&0&1\\0&0&1&0\end{bmatrix}$$

Теперь приступим к доказательству корректности алгоритма 6.1.

Теорема 6.3. *Для любой невырожденной матрицы A алгоритм* 6.1 *вычисляет такие матрицы L, U и P, что $A=LUP$.*

Доказательство. Детали, необходимые для доказательства того, что различные разложения, изображенные на рис. 6.2 и 6.3, корректны, оставляем в качестве упражнения. Покажем лишь, что

1) в строке 2 процедуры МНОЖИТЕЛЬ всегда можно найти ненулевой столбец и
2) в строке 9 всегда существует E^{-1}.

Пусть A будет $(m\times n)$-матрицей. Покажем индукцией по m, где m — степень числа 2, что если A имеет ранг m, то МНОЖИТЕЛЬ вычислит такие L, U и P, что $A=LUP$ и L, U, P — нижняя треугольная матрица, верхняя треугольная матрица и матрица перестановки рангов m, m, n соответственно. Кроме того, первые m столбцов матрицы U образуют подматрицу ранга m. Если $m=1$, то в A должен быть ненулевой элемент, так что базис индукции выполняется. Допустим, что $m=2^k$, $k\geqslant 1$. Так как A имеет m столбцов и ранг m, то каждая из матриц B и C, появляющихся в строке 5, имеет $m/2$ столбцов и ранг $m/2$. По предположению индукции вызов процедуры МНОЖИТЕЛЬ в строке 6 выдает требуемые матрицы L_1, U_1 и P_1, причем первые $m/2$ столбцов матрицы U_1 образуют матрицу ранга $m/2$. Поэтому матрица E^{-1}, необходимая в строке 9, существует.

Из рис. 6.2,*г* видно, что матрица A равна произведению трех матриц, у одной из которых в верхней части стоит U_1, а в нижней G. Ранг этой матрицы должен быть m, ибо матрица A имеет ранг m. Поэтому G имеет ранг $m/2$. Поскольку первые $m/2$ столбцов матрицы G состоят из нулей, а G' получается из G вычеркиванием ее первых $m/2$ столбцов, то ранг матрицы G' также равен $m/2$. Следовательно, по предположению индукции вызов процедуры МНОЖИТЕЛЬ в строке 11 дает нужные L_2, U_2, P_2. Отсюда непосредственно вытекает доказываемое утверждение. □

Прежде чем переходить к анализу времени работы, заметим, что матрицу перестановки можно представить в виде такого массива P, что $P[i]=j$ тогда и только тогда, когда 1 в столбце i стоит в строке j. Поэтому две $(n\times n)$-матрицы перестановок можно перемножить за время $O(n)$, положив $P_1P_2[i]=P_1[P_2[i]]$. При таком представлении можно вычислить за время $O(n)$ также и обращение матрицы перестановки.

Теорема 6.4. *Пусть для каждого n можно умножить две $(n\times n)$-матрицы за такое время $M(n)$, что при некотором $\varepsilon>0$ неравенство*

$M(2m) \geqslant 2^{2+\varepsilon} M(m)$ выполняется для всех m [1]*. Тогда найдется такая постоянная k, что алгоритм* 6.1 *тратит не более $kM(n)$ времени для любой невырожденной матрицы.*

Доказательство. Применим алгоритм 6.1 к $(n \times n)$-матрице. Пусть $T(m)$ — время, требуемое для выполнения процедуры МНОЖИТЕЛЬ(A, m, p), где A — это $(m \times p)$-матрица, $m \leqslant p \leqslant n$. В силу строк 1—4 этой процедуры $T(1) = bn$ для некоторой постоянной b. Рекурсивные вызовы в строках 6 и 11 занимают $T(m/2)$ времени каждый. В каждой из строк 7 и 13 вычисляется матрица, обратная к матрице перестановки (что занимает $O(n)$ времени), и произвольная матрица умножается на матрицу перестановки. Это умножение просто переставляет столбцы первой матрицы. Представляя матрицу перестановки в виде массива P, видим, что $P[i]$-й столбец первой матрицы становится i-м столбцом произведения. Таким образом, произведение можно найти за время $O(mn)$, и тем самым строки 7 и 13 выполняются за время $O(mn)$.

Строка 9 тратит $O(M(m/2))$ времени на вычисление E^{-1} (в силу теоремы 6.2); такое же время требуется для вычисления FE^{-1}. Так как матрица U_1 имеет не более $(m/2)$ строк и не более n столбцов, то произведение $(FE^{-1})U_1$ можно вычислить за время $O((n/m)M(m/2))$.

Заметим, что n делится на m, так как m и n являются степенями числа 2 и $m \leqslant n$. Легко видеть, что остальные шаги занимают $O(mn)$ времени в худшем случае. Таким образом, получили рекуррентные соотношения

$$T(m) \leqslant \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{cn}{m} M\left(\frac{m}{2}\right) + dmn, & \text{если } m > 1, \\ bn, & \text{если } m = 1, \end{cases} \tag{6.5}$$

где b, c и d — постоянные.

В силу условия теоремы и равенства $M(1) = 1$ справедливо неравенство $M(m/2) \geqslant (m/2)^2$. Поэтому можно объединить второе и третье слагаемые в (6.5). Для некоторой постоянной e

$$T(m) \leqslant \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{en}{m} M\left(\frac{m}{2}\right), & \text{если } m > 1, \\ bn, & \text{если } m = 1. \end{cases} \tag{6.6}$$

Из (6.6) выводим

$$T(m) \leqslant \frac{en}{4m}\left[4M\left(\frac{m}{2}\right) + 4^2 M\left(\frac{m}{2^2}\right) + \ldots + 4^{\log m} M(1)\right] + bnm \leqslant$$

$$\leqslant \frac{en}{4m} \sum_{i=1}^{\log m} 4^i M\left(\frac{m}{2^i}\right) + bnm.$$

[1]) Неформально: здесь требуется, чтобы значение $M(n)$ было заключено между $n^{2+\varepsilon}$ и n^3. Может оказаться, например, $M(n) = kn^2 \log n$ для некоторой постоянной k; тогда условие теоремы не удовлетворяется.

Из условия теоремы вытекает, что $4^iM(m/2^i)\leqslant(1/2^\varepsilon)^iM(m)$. Поэтому

$$T(m)\leqslant\frac{en}{4m}M(m)\sum_{i=1}^{\infty}\left(\frac{1}{2^\varepsilon}\right)^i+bnm.$$

Так как сумма в правой части сходится и $M(m)\geqslant m^2$, существует такая постоянная k, что $T(m)\leqslant(kn/m)M(m)$. Для алгоритма 6.1 $n=m$ и, значит, $T(n)\leqslant kM(n)$. □

Следствие. НВП-*разложение любой невырожденной* $(n\times n)$-*матрицы можно найти за* $O(n^{2,81})$ *шагов.*

Доказательство. В силу теорем 6.1, 6.3 и 6.4. □

6.5. ПРИЛОЖЕНИЯ НВП-РАЗЛОЖЕНИЯ

В этом разделе мы покажем, как использовать НВП-разложение для нахождения обратных матриц, вычисления определителей и решения систем линейных уравнений. Мы увидим, что каждая из этих задач сводится к задаче умножения двух матриц и, следовательно, любое улучшение асимптотической временной сложности умножения матриц приводит к улучшению асимптотической временной сложности этих задач. Обратно, умножение матриц, как мы покажем, сводится к обращению матриц и, следовательно, задачи умножения матриц и обращения их эквивалентны с вычислительной точки зрения.

Теорема 6.5. *Пусть* $\varepsilon>0$ *и* $a\geqslant1$. *Пусть* $M(n)$ — *время, требуемое для умножения двух матриц, и* $M(2m)\geqslant2^{2+\varepsilon}M(m)$ *для некоторого* $\varepsilon>0$. *Тогда матрицу, обратную к данной невырожденной матрице, можно найти за время* $O(M(n))$.

Доказательство. Пусть A — невырожденная $(n\times n)$-матрица. В силу теорем 6.3 и 6.4 можно найти НВП-разложение $A=LUP$ за время $O(M(n))$. Тогда $A^{-1}=P^{-1}U^{-1}L^{-1}$. Матрицу P^{-1} можно вычислить за $O(n)$ шагов. Матрицы U^{-1} и L^{-1} существуют, и их можно вычислить за $O(M(n))$ шагов в силу теоремы 6.2. Аналогично за $O(M(n))$ шагов можно вычислить произведение $P^{-1}U^{-1}L^{-1}$. □

Следствие. *Обращение* $(n\times n)$-*матрицы можно найти за* $O(n^{2,81})$ *шагов.*

Теорема 6.6. *Если функция* $M(n)$ *та же, что и в теореме* 6.5, *и* A *есть* $(n\times n)$-*матрица, то* $\det(A)$ *можно вычислить за* $O(M(n))$ *шагов.*

Доказательство. Применим алгоритм 6.1, чтобы найти НВП-разложение матрицы A. Если он не срабатывает из-за того,

что в строке 2 не удалось найти ненулевой столбец или в строке 9 не существует E^{-1}, то матрица A вырожденна, и $\det(A)=0$. В противном случае пусть $A=LUP$. Тогда $\det(A)=\det(L)\det(U)\det(P)$. Найдем $\det(L)$ и $\det(U)$, вычислив произведения их диагональных элементов. Так как L — нормированная нижняя треугольная матрица, то $\det(L)=1$. Так как U — верхняя треугольная, то можно вычислить $\det(U)$ за $O(n)$ шагов. Поскольку P — матрица перестановки, то $\det(A)=\pm 1$ в зависимости от того, представляет P четную или нечетную перестановку. Вопрос о четности или нечетности перестановки можно выяснить, построив ее из $(1, 2, \ldots, n)$ с помощью транспозиций. Потребуется не более $n-1$ транспозиций, и их число можно сосчитать во время выполнения. □

Следствие. *Определитель $(n\times n)$-матрицы можно вычислить за $O(n^{2,81})$ шагов.*

Пример 6.4. Вычислим определитель матрицы M из примера 6.3. Там мы нашли НВП-разложение

$$\begin{bmatrix}0&0&0&1\\0&0&2&0\\0&3&0&0\\4&0&0&0\end{bmatrix}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&2&0&0\\0&0&3&0\\0&0&0&4\end{bmatrix}\begin{bmatrix}0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0\end{bmatrix}.$$

Определители первого и второго сомножителей равны произведениям диагональных элементов, т. е. соответственно 1 и 24. Осталось установить, какую перестановку — четную или нечетную — представляет третья матрица P. Так как P представляет перестановку (4, 3, 2, 1) и ее можно получить двумя транспозициями $(1, 2, 3, 4)\Rightarrow$ $\Rightarrow(4, 2, 3, 1)\Rightarrow(4, 3, 2, 1)$, заключаем, что она четна и $\det(P)=1$. Таким образом, $\det(M)=24$. □

Теорема 6.7. *Пусть функция $M(n)$ та же, что и в теореме* 6.5, *A — невырожденная $(n\times n)$-матрица и $\mathbf{b}$ — вектор-столбец размерности n. Пусть $\mathbf{x}=[x_1, x_2, \ldots, x_n]^T$ — вектор-столбец неизвестных. Тогда систему линейных уравнений $A\mathbf{x}=\mathbf{b}$ можно решить за $O(M(n))$ шагов.*

Доказательство. С помощью алгоритма 6.1 построим НВП-разложение $A=LUP$. Тогда система $LUP\mathbf{x}=\mathbf{b}$ решается в два шага. Сначала решаем систему $L\mathbf{y}=\mathbf{b}$ относительно $\mathbf{y}$, а затем — систему $UP\mathbf{x}=\mathbf{y}$ относительно $\mathbf{x}$. Каждую из этих подзадач можно решить за $O(n^2)$ шагов, применив метод исключения, т. е. сначала найти значение y_1, подставить его вместо переменной y_1, затем найти значение y_2 и т. д. НВП-разложение можно построить за $O(M(n))$ шагов в силу теоремы 6.4, а систему $LUP\mathbf{x}=\mathbf{b}$ можно решить за $O(n^2)$ шагов. □

Следствие. *Систему из n уравнений с n неизвестными можно решить за $O(n^{2,81})$ шагов.*

В заключение покажем, что умножение матриц и обращение их имеют вычислительные сложности одного порядка.

Теорема 6.8. *Пусть для умножения двух $(n\times n)$-матриц и обращения $(n\times n)$-матрицы требуется соответственно $M(n)$ и $I(n)$ времени. Предположим, что $8M(m)\geqslant M(2m)\geqslant 2^{2+\varepsilon}M(m)$ при некотором $\varepsilon>0$ и аналогичные неравенства верны для $I(n)$. Тогда $M(n)$ и $I(n)$ совпадают с точностью до постоянного множителя.*

Доказательство. Теорема 6.5 показывает, что $I(n)\leqslant c_1M(n)$ для некоторой постоянной c_1. Чтобы установить неравенство $M(n)\leqslant c_2I(n)$ для некоторой постоянной c_2, рассмотрим произвольные $(n\times n)$-матрицы A и B. Так как

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix},$$

то можно вычислить произведение AB, обращая $(3n\times 3n)$-матрицу. Следовательно, $M(n)\leqslant I(3n)\leqslant I(4n)\leqslant 64\,I(n)$. □

6.6. УМНОЖЕНИЕ БУЛЕВЫХ МАТРИЦ

В разд. 5.9 изучалась задача умножения двух булевых матриц с элементами из замкнутого полукольца $\{0, 1\}$, в котором сложение и умножение определяются таблицами

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Было показано, что умножение двух булевых матриц эквивалентно вычислению транзитивного замыкания графа. К сожалению, замкнутое полукольцо $\{0, 1\}$ не является кольцом, и поэтому к умножению булевых матриц неприменимы ни алгоритм Штрассена для умножения матриц, ни другие результаты, изложенные ранее в этой главе.

Очевидно, что обычный алгоритм умножения матриц требует $O(n^3)$ шагов [1]). Тем не менее известно по крайней мере два способа

[1]) Если почти все элементы матрицы-произведения равны 1, то можно перемножить две булевы матрицы в среднем за время $O(n^2)$, вычисляя каждую сумму

умножения булевых матриц менее чем за $O(n^3)$ шагов. Первый из них асимптотически лучше, но второй, по-видимому, практичнее для умеренных n. Опишем сейчас первый способ.

Теорема 6.9. *Произведение двух булевых $(n\times n)$-матриц можно вычислить за $O_{\text{А}}(n^{2,81})$ шагов.*

Доказательство. Целые числа по модулю $n+1$ образуют кольцо $\mathbf{Z}_{n+1}$. Для умножения матриц A и B в $\mathbf{Z}_{n+1}$ можно воспользоваться алгоритмом Штрассена. Пусть C — произведение матриц A и B в $\mathbf{Z}_{n+1}$, а D — их произведение как булевых матриц. Легко показать, что если $D[i, j]=0$, то $C[i, j]=0$, и если $D[i, j]=1$, то $1\leqslant C[i, j]\leqslant n$. Следовательно, D легко получается из C. □

Следствие 1. *Если для умножения двух k-разрядных двоичных чисел требуется m битовых операций, то две булевы матрицы можно перемножить за $O_{\text{Б}}(n^{2,81}\, m \log n)$ шагов.*

Доказательство. Так как все арифметические операции можно проделать в $\mathbf{Z}_{n+1}$, для представления чисел достаточно $\lfloor \log n \rfloor + 1$ разрядов. Умножение двух таких чисел занимает не более $O_{\text{Б}}(m \log n)$ времени, а сложение и вычитание — не более $O_{\text{Б}}(\log n)$, что, разумеется, не превосходит $O_{\text{Б}}(m \log n)$. □

В гл. 7 мы обсудим алгоритм умножения чисел, для которого $m(k)=O_{\text{Б}}(k \log k \log\log k)$. Учитывая этот результат, получаем такое следствие.

Следствие 2. *Для умножения булевых матриц требуется не более $O_{\text{Б}}(n^{2,81} \log n \log\log n \log\log\log n)$ шагов.*

Второй метод, часто называемый алгоритмом четырех русских в честь его изобретателей, в какой-то мере "практичнее" алгоритма из теоремы 6.9. Кроме того, он легко переносится на вычисления с двоичными векторами, чего нельзя сказать об алгоритме из теоремы 6.9.

Пусть надо перемножить две булевы $(n\times n)$-матрицы A и B. Для простоты будем считать, что n делится на $\log n$. Можно разбить A на $(n\times(\log n))$-подматрицы, а B — на $((\log n)\times n))$-подмат-

---

$\sum_{k=1}^{n} a_{ik}b_{kj}$ лишь до слагаемого, равного 1. Сумма с таким слагаемым, как мы знаем, равна 1. Если, например, каждый элемент a_{ik} или b_{kj} равен 1 с вероятностью p и эти события независимы, то среднее число слагаемых, которое надо просмотреть, не превосходит $1/p^2$, что не зависит от n.

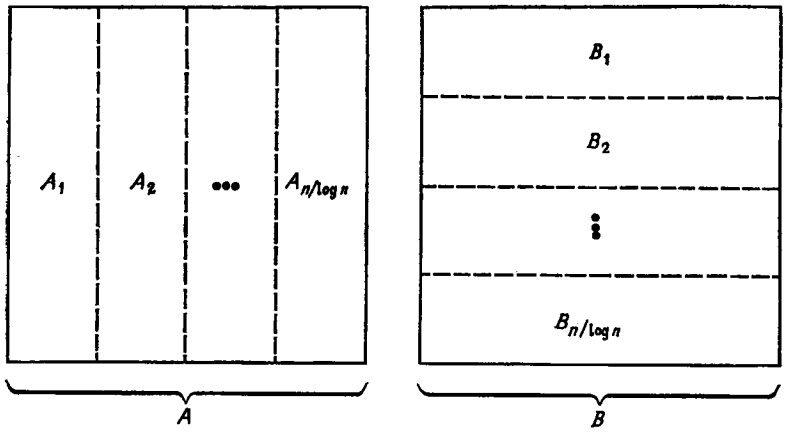


Рис. 6.5. Разбиение булевых матриц A и B.

рицы, как показано на рис. 6.5. Тогда

$$AB = \sum_{i=1}^{n/\log n} A_i B_i.$$

Заметим, что каждое произведение A_iB_i само является $(n \times n)$-матрицей. Если мы сможем вычислить каждое произведение A_iB_i за $O(n^2)$ шагов, то сможем вычислить AB за $O(n^3/\log n)$ шагов, ибо всего $n/\log n$ таких произведений.

Займемся вычислением произведений A_iB_i. Каждое такое произведение можно вычислить за $O(n^2 \log n)$ шагов. Для этого вычисляем $\mathbf{a}_jB_i$ для каждой строки $\mathbf{a}_j$ матрицы A_i. Чтобы найти $\mathbf{a}_jB_i$, берем в матрице B_i все строки с такими номерами k, что в $\mathbf{a}_j$ на k-м месте стоит 1. Затем складываем их, обращаясь с ними как с n-мерными двоичными векторами.

Например, пусть

$$A_i = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \qquad B_i = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Тогда

$$C_i = A_i B_i = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Первая строка в C_i есть в точности третья строка в B_i, поскольку в первой строке в A_i 1 стоит только в третьем столбце. Вторая строка в C_i равна сумме первой и третьей строк в B_i, поскольку во второй строке в A_i 1 стоит в первом и третьем столбцах, и т. д.

Вычисление $\mathbf{a}_j B_i$ для каждой строки $\mathbf{a}_j$ матрицы A_i занимает $O(n \log n)$ времени. Так как в A_i n строк, то общий объем работы при таком вычислении $A_i B_i$ составляет $O(n^2 \log n)$.

Чтобы быстрее вычислить $A_i B_i$, заметим, что в каждой строке матрицы A_i содержится $\log n$ элементов, каждый из которых равен 0 или 1. Поэтому все матрицы A_i могут содержать в общей сложности не более $2^{\log n} = n$ различных строк.

Таким образом, из строк матрицы B_i можно составить лишь n различных сумм. Можно заранее заготовить таблицу всех возможных сумм строк матрицы B_i и вместо вычисления $\mathbf{a}_j B_i$ находить в ней по $\mathbf{a}_j$ ответ.

Этот метод тратит лишь $O(n^2)$ времени на вычисление $A_i B_i$. Объясняется это так. Любое подмножество строк матрицы B_i или пусто, или состоит из одного элемента, или равно объединению одноэлементного множества и множества, меньшего исходного. Выбрав правильный порядок, можно вычислять каждую сумму строк, прибавляя одну строку матрицы B_i к уже сосчитанной сумме строк. Так можно получить все n сумм строк матрицы B_i за $O(n^2)$ шагов. После вычисления сумм и расположения их в виде массива, можно выбирать нужную сумму для каждой из n строк матрицы A_i.

Алгоритм 6.2. Алгоритм четырех русских для умножения булевых матриц

Вход. Две булевы $(n \times n)$-матрицы A и B.

Выход. Произведение $C = AB$.

Метод. Положим $m = \lfloor \log n \rfloor$. Разобьем A на матрицы $A_1, A_2, \ldots, A_{\lceil n/m \rceil}$, где A_i, $1 \leqslant i < \lceil n/m \rceil$, состоит из столбцов матрицы A с номерами от $m(i-1)+1$ до mi, а $A_{\lceil n/m \rceil}$ — из оставшихся последних столбцов, к которым добавлены столбцы из нулей, если это нужно, чтобы в $A_{\lceil n/m \rceil}$ было m столбцов. Разобьем B на матрицы $B_1, B_2, \ldots, B_{\lceil n/m \rceil}$, где B_i, $1 \leqslant i < \lceil n/m \rceil$, состоит из строк матрицы

begin

1. **for** $i \leftarrow 1$ **until** $\lceil n/m \rceil$ **do**
   begin
   comment Вычисляем суммы строк $\mathbf{b}_1^{(i)}, \ldots, \mathbf{b}_m^{(i)}$ матрицы B_i;
2. СУММАСТРОК[0] $\leftarrow [\underbrace{0, 0, \ldots, 0}_{n}]$;
3. **for** $j \leftarrow 1$ **until** $2^m - 1$ **do**
   begin
4. пусть k таково, что $2^k \leqslant j < 2^{k+1}$;
5. СУММАСТРОК[j] $\leftarrow$ СУММАСТРОК[$j - 2^k$] $+ \mathbf{b}_{k+1}^{(i)}$ [1])
   end
6. пусть C_i — матрица, j-я строка которой равна СУММАСТРОК[ЧИС($\mathbf{a}_j$)], где $\mathbf{a}_j$ есть j-я строка матрицы A_i, $1 \leqslant j \leqslant n$
   end;
7. пусть $C = \sum_{i=1}^{\lceil n/m \rceil} C_i$

end

[1]) Здесь, конечно, подразумевается поразрядная булева сумма.

Рис. 6.6. Алгоритм четырех русских.

B с номерами от $m(i-1)+1$ до mi, а $B_{\lceil n/m \rceil}$ — из оставшихся строк, к которым добавлены строки из нулей, если это необходимо для получения m строк. Эта ситуация, по существу, совпадает с изображенной на рис. 6.5. Вычисления приведены на рис. 6.6. ЧИС(**v**) обозначает целое число, представленное двоичным вектором **v**, записанным в двоичной системе счисления с обратным порядком разрядов. Например, ЧИС([0, 1, 1])=6. □

Теорема 6.10. *Алгоритм* 6.2 *вычисляет* $C=AB$ *за* $O(n^3/\log n)$ *шагов.*

Доказательство. Простая индукция по j показывает, что в строках 2—5 СУММАСТРОК[j] становится равной поразрядной булевой сумме таких строк $\mathbf{b}_k$ матрицы B_i, что в двоичном представлении числа j на k-м месте справа стоит 1. Отсюда вытекает, что $C_i = A_i B_i$ в строке 6 и, значит, $C=AB$ в строке 7.

Для подсчета временной сложности алгоритма сначала рассмотрим цикл, описанный строками 3—5. Оператор присваивания в

строке 5, очевидно, выполняется за $O(n)$ шагов. Вычисление значения k в строке 4 занимает время $O(m)$, меньшее $O(n)$, так что все тело цикла (строки 4, 5) занимает время $O(n)$. Цикл повторяется 2^m-1 раз, поэтому его сложность равна $O(n\,2^m)$. Так как $m \leqslant \log n$, то цикл в строках 3—5 занимает время $O(n^2)$.

В строке 6 вычисление ЧИС($\mathbf{a}_j$) имеет сложность $O(m)$, а копирование вектора СУММАСТРОК[ЧИС($\mathbf{a}_j$)] — сложность $O(n)$, так что строка 6 выполняется за $O(n^2)$ шагов. Так как $\lceil n/m \rceil \leqslant 2n/\log n$, то цикл в строках 1—6, который повторяется $\lceil n/m \rceil$ раз, занимает время $O(n^3/\log n)$. Аналогично в строке 7 надо найти не более $2n/\log n$ сумм $(n \times n)$-матриц, что дает сложность $O(n^3/\log n)$. Таким образом, весь алгоритм требует $O(n^3/\log n)$ шагов. □

Интереснее, по-видимому, то, что алгоритм 6.2 можно реализовать за $O_{\text{ДВ}}(n^2/\log n)$ вычислений с двоичными векторами, если в нашем распоряжении есть логические и арифметические операции над цепочками из 0 и 1.

Теорема 6.11. *Алгоритм* 6.2 *можно реализовать за* $O_{\text{ДВ}}(n^2/\log n)$ *операций с двоичными векторами.*

Доказательство. Для того чтобы узнавать, когда надо увеличить k, используется счетчик. Вначале значение счетчика равно 1, а $k=0$. Всякий раз, когда j увеличивается, счетчик уменьшается на 1, если его значение было отлично от 1; а в последнем случае значение счетчика полагается равным новому значению j, а k увеличивается на 1.

Присваивания в строках 2 и 5 рис. 6.6 занимают постоянное время. Следовательно, цикл в строках 3—5 имеет сложность $O_{\text{ДВ}}(n)$. Поскольку в РАМ двоичный вектор представляется целым числом [1]), на вычисление ЧИС($\mathbf{a}_i$) в строке 6 не уходит время, так что каждую строку матрицы C_i можно найти за фиксированное число операций над двоичными векторами и строка 6 имеет сложность $O_{\text{ДВ}}(n)$. Поэтому цикл в строках 1—6 занимает время $O_{\text{ДВ}}(n^2/\log n)$; такое же время тратится и на строку 7. □

УПРАЖНЕНИЯ

6.1. Покажите, что целые числа по модулю n образуют кольцо, т. е. что $\mathbf{Z}_n$— кольцо $(\{0, 1, \ldots, n-1\}, +, \cdot, 0, 1)$, где $a+b$ и $a \cdot b$ — обычные сложение и умножение по модулю n.

6.2. Покажите, что $(n \times n)$-матрицы с элементами из некоторого кольца R образуют кольцо.

---

[1]) Можно обойти ту деталь, что ЧИС($\mathbf{a}_i$) соответствует не самому вектору $\mathbf{a}_i$, а перестановке его элементов в обратном порядке, если в качестве j-й строки матрицы B взять j-ю строку снизу, а не сверху, как мы делали раньше.

6.3. Приведите пример, показывающий, что произведение матриц некоммутативно, даже если их элементы берутся из кольца с коммутативным умножением.

6.4. Примените алгоритм Штрассена для вычисления произведения

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

6.5. Еще один вариант алгоритма Штрассена использует для вычисления произведения двух (2×2)-матриц равенства

$$\begin{aligned}
s_1 &= a_{21} + a_{22}, & m_1 &= s_2 s_6, & t_1 &= m_1 + m_2, \\
s_2 &= s_1 - a_{11}, & m_2 &= a_{11} b_{11}, & t_2 &= t_1 + m_4. \\
s_3 &= a_{11} - a_{21}, & m_3 &= a_{12} b_{21}, & & \\
s_4 &= a_{12} - s_2, & m_4 &= s_3 s_7, & & \\
s_5 &= b_{12} - b_{11}, & m_5 &= s_1 s_5, & & \\
s_6 &= b_{22} - s_5, & m_6 &= s_4 b_{22}, & & \\
s_7 &= b_{22} - b_{12}, & m_7 &= a_{22} s_8, & & \\
s_8 &= s_6 - b_{21}, & & & &
\end{aligned}$$

Элементы произведения получаются так:

$$\begin{aligned}
c_{11} &= m_2 + m_3, & c_{21} &= t_2 - m_7, \\
c_{12} &= t_1 + m_5 + m_6, & c_{22} &= t_2 + m_5.
\end{aligned}$$

Покажите, что эти элементы равны соответствующим элементам из (6.1). Заметьте, что было сделано только 7 умножений и 15 сложений.

6.6. Докажите следующие соотношения для $(n\times n)$-матриц A, B и C:

(а) если $AB=I$ и $AC=I$, то $B=C$, (б) $A^{-1}A=I$,
(в) $(AB)^{-1}=B^{-1}A^{-1}$, (г) $(A^{-1})^{-1}=A$,
(д) $\det(AB)=\det(A)\det(B)$.

6.7. Теорема 6.2 показывает, что невырожденную верхнюю треугольную матрицу можно обратить за $cn^{2,81}$ арифметических операций, где c — постоянная. Найдите эту постоянную c в предположении, что матрицы умножаются по алгоритму Штрассена и n — степень числа 2.

6.8. Представим матрицу перестановки массивом P, для которого $P[i]=j$ тогда и только тогда, когда в i-м столбце на j-й строке стоит 1. Пусть P_1 и P_2 — такие представления $(n\times n)$-матриц перестановки.

(а) Докажите, что $P_1P_2[i]=P_1[P_2[i]]$.

(б) Постройте алгоритм, вычисляющий P_1^{-1} за время $O(n)$.

(c) Измените описанное представление так, чтобы $P[i]=j$ тогда и только тогда, когда на i-й строке в j-м столбце стоит 1. Напишите правильную формулу для P_1P_2 и постройте алгоритм для вычисления P_1^{-1}.

6.9. Примените алгоритм 6.1, чтобы найти НВП-разложение матрицы

$$M=\begin{bmatrix}0 & 0 & 1 & 2\\ 0 & 0 & 3 & 0\\ 1 & -1 & 0 & 1\\ 2 & 0 & -1 & 3\end{bmatrix}.$$

6.10. Мы показали, что нахождение НВП-разложения, обращение матрицы, вычисление определителя и решение системы линейных уравнений имеют сложность $O_A(n^{2,81})$. Найдите наилучшие мультипликативные постоянные для каждой из этих задач в предположении, что матрицы умножаются по алгоритму Штрассена, n — степень числа 2 и применяется техника алгоритма 6.1 и теорем 6.4—6.7.

6.11. Для матрицы M из упр. 6.9 найдите (а) обратную и (б) определитель, используя технику этой главы.

6.12. С помощью НВП-разложения решите систему

$$\begin{aligned} x_3+\ 2x_4&=7\\ 3x_3\qquad\ &=9\\ x_1-x_2\ +\ x_4&=3\\ 2x_1\ -x_3+3x_4&=10.\end{aligned}$$

6.13. Покажите, что каждая перестановка либо четна, либо нечетна, но не то и другое одновременно.

6.14. Вычислите произведение булевых матриц

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 1\\ 1 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}0 & 1 & 0 & 0\\ 1 & 1 & 0 & 0\\ 1 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix},$$

применяя (а) метод теоремы 6.9 и (б) алгоритм четырех русских.

6.15. Закончите доказательство теоремы 6.3, показав, что верны взаимосвязи, изображенные на рис. 6.2 и 6.3.

****6.16.** Рассмотрим поле [1]) F_2 целых чисел по модулю 2. Найдите алгоритм умножения $(n\times n)$-матриц над F_2 с асимптотической сложностью не более $O(n^{2,81}/(\log n)^{0,4})$. *Указание:* Разбейте матрицы на блоки размера $\sqrt{\log n}\times\sqrt{\log n}$.

[1]) Определение поля см. в разд. 12.1.

6.17. Оцените значение n, начиная с которого $n^{2,81}$ меньше $n^3/\log n$.

***6.18.** Пусть $L(n)$ — время умножения двух нижних треугольных $(n\times n)$-матриц, а $T(n)$ — время умножения двух произвольных матриц. Докажите, что найдется такая постоянная c, что $T(n)\leqslant cL(n)$.

6.19. Докажите, что матрица, обратная к верхней (нижней) треугольной, является верхней (нижней) треугольной.

***6.20.** Пусть $I(n)$ — число шагов, необходимое для обращения $(n\times n)$-матрицы, а $U(n)$ — для обращения верхней треугольной матрицы. Докажите, что найдется такая постоянная c, что $I(n)\leqslant cU(n)$ для всех n

****6.21.** Чтобы вычислить произведение матриц $C=AB$, можно было бы определить сначала $D=(PAQ)(Q^{-1}BR)$, а затем $C=P^{-1}DR^{-1}$. Если P, Q и R — специальные матрицы, например матрицы перестановок, то вычисление произведений PAQ, $Q^{-1}BR$ и $P^{-1}DR^{-1}$ не требует умножения элементов кольца. Воспользуйтесь этой идеей, чтобы найти другой метод перемножения (2×2)-матриц за 7 умножений элементов кольца.

6.22. Докажите, что НВ-разложение невырожденной матрицы A, если оно существует, единственно. *Указание:* Пусть $A=L_1U_1=L_2U_2$. Покажите, что $L_2^{-1}L_1=U_2U_1^{-1}=I$.

6.23. Докажите, что если матрица A невырожденна и каждая ее главная подматрица невырожденна, то A имеет НВ-разложение.

6.24. Может ли вырожденная матрица обладать НВП-разложением?

****6.25.** Пусть A будет $(n\times m)$-матрицей с вещественными элементами. Матрица A называется *положительно определенной*, если для каждого ненулевого вектора-столбца $\mathbf{x}$ выполнено неравенство $\mathbf{x}^TA\mathbf{x}>0$.

(а) Покажите, что лемму 6.5 можно применить для обращения произвольной невырожденной симметричной положительно определенной матрицы.

(б) Покажите, что если матрица A невырожденна, то матрица AA^T положительно определена и симметрична.

(в) Используя (а) и (б), постройте алгоритм сложности $O_A(M(n))$ для обращения любой невырожденной матрицы с вещественными элементами.

(г) Будет ли ваш алгоритм для (в) работать в случае поля целых чисел по модулю 2?

***6.26.** Матрица A размера $n\times n$ называется *тёплицевой*, если $A[i, j]=A[i-1, j-1]$, $2\leqslant i, j\leqslant n$.

(а) Найдите представление тёплицевых матриц, при котором сумму двух тёплицевых матриц можно найти за $O(n)$ шагов.

(б) С помощью приема "разделяй и властвуй" постройте алгоритм умножения тёплицевой $(n \times n)$-матрицы на вектор-столбец. Сколько арифметических операций он требует? *Указание:* С помощью приема "разделяй и властвуй" можно получить алгоритм сложности $O(n^{1,59})$. В гл. 7 будет развита техника, которую можно будет применить для улучшения этого результата.

(в) Постройте асимптотически эффективный алгоритм умножения двух тёплицевых $(n \times n)$-матриц. Сколько арифметических операций он требует? Заметьте, что произведение тёплицевых матриц может не быть тёплицевой матрицей.

Проблемы для исследования

6.27. Естественно попытаться непосредственно улучшить метод Штрассена. Хопкрофт, Керр [1971] показали, что для вычисления произведения (2×2)-матриц над произвольным кольцом необходимы семь умножений. Однако рекурсивный алгоритм может быть основан на умножении матриц какого-нибудь другого небольшого размера. Например, можно было бы улучшить порядок алгоритма Штрассена, если суметь перемножать (3×3)-матрицы за 21 умножение или (4×4)-матрицы за 48 умножений.

6.28. Можно ли находить кратчайшие пути меньше, чем за $O(n^3)$ шагов? Алгоритм Штрассена неприменим к замкнутым полукольцам, состоящим из неотрицательных вещественных чисел и $+\infty$, но, может быть, удастся свести операции в этом замкнутом полукольце к операциям в некотором кольце, как это было сделано для булевых матриц.

Замечания по литературе

Алгоритм Штрассена заимствован из работы Штрассена [1969]. Виноград [1973] уменьшил число необходимых сложений до 15, что улучшило мультипликативную постоянную, но не порядок сложности (см. упр. 6.5).

Штрассен [1969] также описал методы сложности $O(n^{2,81})$ для обращения матриц, вычисления определителей и решения систем линейных уравнений в предположении, что каждая матрица, встречающаяся в процессе вычислений, невырожденна. Банч, Хопкрофт [1974] показали, что НВП-разложение можно сделать за $O(n^{2,81})$ шагов при единственном предположении, что исходная матрица невырожденна. Шёнхаге независимо показал, что обращение любой невырожденной матрицы над упорядоченным полем можно найти за $O(n^{2,81})$ шагов (упр. 6.25).

Результат о том, что умножение матриц не сложнее обращения матрицы, получен Виноградом [1970б]. Алгоритм сложности $O_{\text{Д}}(n^{2,81})$ для умножения булевых матриц построен Фишером, Мейером [1971], а алгоритм четырех русских — Арлазаровым, Диницем, Кронродом, Фараджевым [1970]. Валиант [1974] применил алгоритм Штрассена для распознавания бесконтекстных языков $O(n^{2,81})$.

Дополнительные сведения по теории матриц можно найти у Хона [1958]. Алгебраические понятия, такие, как кольцо, изложены в книге Маклейна, Биркгофа [1967]. Решение упр. 6.13 приведено Биркгофом, Барти [1970], упр. 6.16 принадлежит Хопкрофту.

7 БЫСТРОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ И ЕГО ПРИЛОЖЕНИЯ

Преобразование Фурье естественно возникает во многих задачах теоретического и прикладного характера, и потому имеет смысл изучить эффективный алгоритм его вычисления. Вездесущность преобразования Фурье будет в дальнейшем продемонстрирована его применимостью к построению эффективных алгоритмов. Во многих приложениях бывает удобно преобразовать данную задачу в другую, более легкую. Примером служит вычисление произведения двух полиномов. С вычислительной точки зрения целесообразно сначала применить линейное преобразование к векторам коэффициентов полиномов, затем над образами коэффициентов выполнить операцию, более простую, чем свертка, и, наконец, к результату применить обратное преобразование, чтобы получить искомое произведение. В данном случае подходящим линейным преобразованием будет дискретное преобразование Фурье.

В этой главе мы изучим преобразование Фурье и обратное к нему и обсудим его роль в вычислении сверток и произведений различных типов. Будет изложен эффективный алгоритм, называемый быстрым преобразованием Фурье (БПФ). Он основан на технике вычисления полиномов с помощью деления, и в нем учитывается, что полиномы вычисляются для аргументов, равных корням из единицы.

Затем докажем теорему о свертке. Свертку будем интерпретировать как вычисление полиномов в корнях из единицы, умножение этих значений и последующую интерполяцию полиномов. С помощью быстрого преобразования Фурье разработаем эффективный алгоритм для свертки и применим его к формальному (символьному) умножению полиномов и умножению целых чисел. Получающийся алгоритм умножения целых чисел, называемый алгоритмом Шёнхаге — Штрассена, является асимптотически самым быстрым из известных способов умножения двух целых чисел.

7.1. ДИСКРЕТНОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ И ОБРАТНОЕ К НЕМУ

Обычно преобразование Фурье определяется над кольцом комплексных чисел. По причинам, которые станут ясными позже, мы будем определять преобразование Фурье над произвольным комму-

тативным кольцом $(R, +, \cdot, 0, 1)$ [1]). Элемент ω из R, обладающий свойствами

1) $\omega \neq 1$,
2) $\omega^n = 1$,
3) $\sum_{j=0}^{n-1} \omega^{jp} = 0$ для $1 \leqslant p < n$,

называется *примитивным корнем n-й степени из единицы*. Элементы $\omega^0, \omega^1, \ldots, \omega^{n-1}$ называются *корнями n-й степени из единицы*.

Например, $e^{2\pi i/n}$, где $i = \sqrt{-1}$, является примитивным корнем n-й степени из единицы в кольце комплексных чисел.

Пусть $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ будет n-мерным вектором (-столбцом) с элементами из R. Мы предполагаем, что элемент n обладает в этом кольце обратным относительно умножения [2]) и что ω — примитивный корень n-й степени из единицы в этом кольце. Пусть A — такая $(n \times n)$-матрица, что $A[i, j] = \omega^{ij}$ для $0 \leqslant i, j < n$. *Дискретным преобразованием Фурье* вектора $\mathbf{a}$ называется вектор $A\mathbf{a}$, обозначаемый также $F(\mathbf{a})$; его i-я компонента b_i, $0 \leqslant i < n$, равна $\sum_{k=0}^{n-1} a_k \omega^{ik}$. Матрица A невырожденна, и, значит, существует обратная к ней матрица A^{-1}; ее простой вид описывается в лемме 7.1.

Лемма 7.1. *Пусть R — коммутативное кольцо, ω — примитивный корень n-й степени из единицы и n как элемент кольца R имеет обратный. Пусть A — такая $(n \times n)$-матрица, что $A[i, j] = \omega^{ij}$, $0 \leqslant i, j < n$. Тогда существует матрица A^{-1} и (i, j)-й элемент ее равен $(1/n)\omega^{-ij}$.*

Доказательство. Пусть $\delta_{ij} = 1$, если $i = j$, и $\delta_{ij} = 0$ в противном случае. Достаточно показать, что если матрица A^{-1} определена, как выше, то $A \cdot A^{-1} = I_n$, т. е. (i, j)-й элемент матрицы $A \cdot A^{-1}$ удовлетворяет равенству

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-kj} = \delta_{ij} \quad \text{для} \quad 0 \leqslant i, j < n. \tag{7.1}$$

Если $i = j$, то левая часть равенства (7.1) превращается в

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^0 = 1.$$

[1]) Напомним, что коммутативным называется кольцо, в котором умножение (как и сложение) коммутативно.

[2]) Под элементом n кольца подразумевается $1+1+\ldots+1$ (n раз). В этом смысле целые числа появляются в любом кольце, даже конечном. Заметим также, что примитивный корень ω обладает обратным, так как $\omega \cdot \omega^{n-1} = 1$ (и, следовательно, $\omega^{-1} = \omega^{n-1}$), поэтому можно говорить об отрицательных степенях ω.

Пусть $i \neq j$ и $q = i - j$. Тогда левая часть в (7.1) равна

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk}, \quad -n < q < n, \quad q \neq 0.$$

Если $q > 0$, то

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk} = 0,$$

поскольку ω — примитивный корень n-й степени из единицы. Если $q < 0$, то, умножив левую часть на $\omega^{-q(n-1)}$, изменив порядок слагаемых и подставив $-q$ вместо q, получим

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk}, \qquad 0 < q < n.$$

Эта сумма тоже равна 0, поскольку ω — примитивный корень n-й степени из единицы. Отсюда сразу вытекает (7.1). □

Обратным дискретным преобразованием Фурье вектора $\mathbf{a}$ называется вектор $F^{-1}(\mathbf{a}) = A^{-1}\mathbf{a}$, i-я компонента которого, $0 \leqslant i < n$, равна

$$\frac{1}{n} \sum_{k=0}^{n-1} a_k \omega^{-ik}.$$

Очевидно, что в результате применения обратного преобразования к преобразованию вектора $\mathbf{a}$ получается сам вектор $\mathbf{a}$, т. е. $F^{-1}F(\mathbf{a}) = \mathbf{a}$.

Преобразование Фурье тесно связано с вычислением полиномов и их интерполяцией. Пусть

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

— полином $(n-1)$-й степени. Его можно однозначно представить двумя способами: списком его коэффициентов $a_0, a_1, \ldots, a_{n-1}$ и списком его значений в n различных точках $x_0, x_1, \ldots, x_{n-1}$. Процесс нахождения коэффициентов полинома по его значениям в точках $x_0, x_1, \ldots, x_{n-1}$ называется *интерполяцией*.

Вычисление преобразования Фурье вектора $[a_0, a_1, \ldots, a_{n-1}]^T$ эквивалентно превращению представления полинома $\sum_{i=0}^{n-1} a_i x^i$ списком его коэффициентов в представление его списком значений в точках $\omega^0, \omega^1, \ldots, \omega^{n-1}$. Точно так же вычисление обратного преобразования Фурье эквивалентно интерполяции полинома по его значениям в корнях n-й степени из единицы.

Можно было бы определить преобразование, которое вычисляло бы значения полинома на множестве точек, отличных от корней из единицы. Например, можно было бы использовать целые числа $1, 2, \ldots, n$. Однако мы увидим, что, выбирая степени корня ω, мы делаем вычисление значений и интерполяцию особенно простыми. В гл. 8 преобразование Фурье применяется для вычисления значений и интерполяции полиномов в произвольных точках.

Одно из основных приложений преобразования Фурье— вычисление свертки двух векторов. Пусть

$$\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T, \qquad \mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]^T$$

— два вектора-столбца. Их *сверткой* $\mathbf{a} \otimes \mathbf{b}$ называется такой вектор $\mathbf{c} = [c_0, c_1, \ldots, c_{2n-1}]^T$, что $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$. (Полагаем $a_k = b_k = 0$, если $k < 0$ или $k \geqslant n$.) Таким образом,

$$c_0 = a_0 b_0, \quad c_1 = a_0 b_1 + a_1 b_0, \quad c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0, \text{ и т. д.}$$

Заметим, что $c_{2n-1} = 0$; эта компонента включена только для симметрии.

Чтобы мотивировать рассмотрение свертки, снова обратимся к представлению полинома его коэффициентами. Произведение двух полиномов степени n—1

$$p(x) = \sum_{i=0}^{n-1} a_i x^i, \qquad q(x) = \sum_{j=0}^{n-1} b_j x^j$$

является полиномом степени $2n$—2

$$p(x)\, q(x) = \sum_{i=0}^{2n-2} \left[\sum_{j=0}^{i} a_j b_{i-j} \right] x^i.$$

Заметим, что коэффициенты произведения — это в точности компоненты свертки векторов $[a_0, a_1, \ldots, a_{n-1}]^T$ и $[b_0, b_1, \ldots, b_{n-1}]^T$, составленных из коэффициентов исходных полиномов (коэффициентом c_{2n-1}, равным 0, мы пренебрегаем).

Если два полинома степени n—1 представлены своими коэффициентами, то, чтобы вычислить коэффициенты их произведения, можно устроить свертку векторов их коэффициентов. С другой стороны, если $p(x)$ и $q(x)$ представлены своими значениями в корнях n-й степени из единицы, то, чтобы вычислить аналогичное представление для их произведения, можно просто перемножить пары значений $p(x)$ и $q(x)$ в соответствующих корнях. Отсюда следует, что свертка двух векторов $\mathbf{a}$ и $\mathbf{b}$ равна обратному преобразованию, примененному к покомпонентному произведению их образов. Формально это записывается так: $\mathbf{a} \otimes \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. Иными словами, свертку векторов можно вычислить, взяв их преобразования Фурье, вычислив покомпонентное произведение и затем сделав обратное преобра-

зование. Единственная трудность состоит в том, что произведение двух полиномов степени $n-1$, вообще говоря, является полиномом степени $2n-2$ и для его представления требуются его значения в $2n-2$ различных точках. В теореме 7.1 показывается, как преодолеть эту трудность. Можно рассматривать $p(x)$ и $q(x)$ как полиномы степени $2n-1$, у которых равны нулю коэффициенты при n старших степенях x (т. е. полиномы степени $n-1$ считать полиномами степени $2n-1$).

Теорема 7.1. (Теорема о свертке.) *Пусть*

$$\mathbf{a}=[a_0,\ a_1,\ \ldots,\ a_{n-1},\ 0,\ \ldots,\ 0]^T$$

и

$$\mathbf{b}=[b_0,\ b_1,\ \ldots,\ b_{n-1},\ 0,\ \ldots,\ 0]^T$$

— векторы-столбцы размерности $2n$, *а*

$$F(\mathbf{a})=[a'_0,\ a'_1,\ \ldots,\ a'_{2n-1}]^T$$

и

$$F(\mathbf{b})=[b'_0,\ b'_1,\ \ldots,\ b'_{2n-1}]^T$$

— их преобразования Фурье. Тогда $\mathbf{a}\circledast\mathbf{b}=F^{-1}(F(\mathbf{a})\cdot F(\mathbf{b}))$.

Доказательство. Так как $a_i=b_i=0$ для $n\leqslant i<2n$, то при $0\leqslant l<2n$

$$a'_l=\sum_{j=0}^{n-1}a_j\omega^{lj},\qquad b'_l=\sum_{k=0}^{n-1}b_k\omega^{lk}.$$

Следовательно,

$$a'_lb'_l=\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}a_jb_k\omega^{l(j+k)}. \tag{7.2}$$

Пусть $\mathbf{a}\circledast\mathbf{b}=[c_0,\ c_1,\ \ldots,\ c_{2n-1}]^T$ и $F(\mathbf{a}\circledast\mathbf{b})=[c'_0,\ c'_1,\ \ldots,\ c'_{2n-1}]^T$. Так как $c_p=\sum_{j=0}^{2n-1}a_jb_{p-j}$, то

$$c'_l=\sum_{p=0}^{2n-1}\sum_{j=0}^{2n-1}a_jb_{p-j}\omega^{lp}. \tag{7.3}$$

Меняя порядок суммирования в (7.3) и подставляя k вместо $p-j$, получаем

$$c'_l=\sum_{j=0}^{2n-1}\sum_{k=-j}^{2n-1-j}a_jb_k\omega^{l(j+k)}. \tag{7.4}$$

Так как $b_k=0$ для $k<0$, можно повысить нижний предел суммирования во внутренней сумме до $k=0$. Аналогично, поскольку $a_j=0$ для $j\geqslant n$, можно понизить верхний предел во внешней сумме до $n-1$. Верхний предел внутреннего суммирования не меньше n

независимо от значения j. Таким образом, можно заменить верхний предел на $n-1$, так как $b_k=0$ при $k\geqslant n$. После этих изменений (7.4) превратится в (7.2); отсюда следует, что $c'_l=a'_l\,b'_l$. Итак, $F(\mathbf{a}\circledast\mathbf{b})=F(\mathbf{a})\cdot F(\mathbf{b})$, откуда $\mathbf{a}\circledast\mathbf{b}=F^{-1}(F(\mathbf{a})\cdot F(\mathbf{b}))$. □

Свертка двух n-мерных векторов является $2n$-мерным вектором. Это требует, чтобы в теореме о свертке вектора **a** и **b** были "разбавлены" n нулями. Чтобы избежать этого "разбавления", введем понятие обернутой свертки.

Определение. Пусть $\mathbf{a}=[a_0, a_1, \ldots, a_{n-1}]^T$ и $\mathbf{b}=[b_0, b_1, \ldots, b_{n-1}]^T$ — два n-мерных вектора. *Положительно обернутой сверткой* векторов **a** и **b** называется такой вектор $\mathbf{c}=[c_0, c_1, \ldots, c_{n-1}]^T$, что

$$c_l=\sum_{j=0}^{l}a_jb_{l-j}+\sum_{j=l+1}^{n-1}a_jb_{n+l-j}.$$

Отрицательно обернутой сверткой векторов **a** и **b** называется такой вектор $\mathbf{d}=[d_0, d_1, \ldots, d_{n-1}]^T$, что

$$d_l=\sum_{j=0}^{l}a_jb_{l-j}-\sum_{j=l+1}^{n-1}a_jb_{n+l-j}.$$

В разд. 7.5 мы воспользуемся обернутой сверткой в алгоритме Шёнхаге — Штрассена — алгоритме быстрого умножения целых чисел. А пока отметим следующее. Вычислив значения двух полиномов степени $n-1$ в корнях n-й степени из единицы и перемножив пары значений в соответствующих точках, мы получим n значений, по которым сможем однозначно интерполировать полином степени $n-1$. Вектор коэффициентов этого единственного полинома как раз и будет положительно обернутой сверткой векторов коэффициентов исходных полиномов.

Теорема 7.2. *Пусть* $\mathbf{a}=[a_0, a_1, \ldots, a_{n-1}]^T$ *и* $\mathbf{b}=[b_0, b_1, \ldots, b_{n-1}]^T$ *— два n-мерных вектора, а* ω *— примитивный корень n-й степени из единицы. Пусть* $\psi^2=\omega$. *Предположим, что элемент n имеет обратный. Тогда*

1) *положительно обернутая свертка векторов* **a** *и* **b** *равна* $F^{-1}(F(\mathbf{a})\cdot F(\mathbf{b}))$,
2) *если* $\mathbf{d}=[d_0, d_1, \ldots, d_{n-1}]^T$ *— отрицательно обернутая свертка векторов* **a** *и* **b**, $\hat{\mathbf{a}}=[a_0, \psi a_1, \ldots, \psi^{n-1}a_{n-1}]^T$, $\hat{\mathbf{b}}=[b_0, \psi b_1, \ldots, \psi^{n-1}b_{n-1}]^T$, $\hat{\mathbf{d}}=[d_0, \psi d_1, \ldots, \psi^{n-1}d_{n-1}]^T$, *то* $\hat{\mathbf{d}}=F^{-1}(F(\hat{\mathbf{a}})\cdot F(\hat{\mathbf{b}}))$.

Доказательство. Теорема доказывается аналогично теореме 7.1 с учетом равенства $\psi^n=-1$. Детали оставляем в качестве упражнения. □

7.2. АЛГОРИТМ БЫСТРОГО ПРЕОБРАЗОВАНИЯ ФУРЬЕ

Очевидно, что преобразование Фурье вектора $\mathbf{a}$ из R^n и обратное к нему можно вычислить за время $O_A(n^2)$ в предположении, что каждая арифметическая операция над элементами кольца R выполняется за один шаг. Однако если n — степень числа 2, то можно сделать это быстрее: известен алгоритм сложности $O_A(n \log n)$, вычисляющий преобразование Фурье или обратное к нему, и мы думаем, что эта оценка неулучшаема. Приведем только алгоритм прямого преобразования Фурье. Алгоритм обратного преобразования аналогичен и предоставляется читателю. Основная идея быстрого преобразования Фурье (БПФ) является по своей природе алгебраической; мы находим подобие между частями тех n сумм, которые порождаются умножением $A\mathbf{a}$. Во всем этом разделе мы считаем, что $n=2^k$ для некоторого целого числа k.

Вспомним, что вычисление вектора $A\mathbf{a}$ эквивалентно вычислению полинома $p(x)=\sum_{j=0}^{n-1} a_j x^j$ в точках $x=\omega^0, \omega^1, \ldots, \omega^{n-1}$. Но вычисление $p(x)$ в точке $x=a$ равносильно нахождению остатка от деления $p(x)$ на $x-a$. (Чтобы увидеть это, можно записать $p(x)$ в виде $(x-a)\,q(x)+c$, где c — постоянная. Тогда $p(a)=c$.) Таким образом, вычисление преобразования Фурье сводится к нахождению остатка от деления полинома $p(x)=\sum_{j=0}^{n-1} a_j x^j$ степени $n-1$ на каждый из полиномов $x-\omega^0$, $x-\omega^1$, $\ldots$, $x-\omega^{n-1}$.

Обычное последовательное деление $p(x)$ на каждый из полиномов $x-\omega^i$ дает процесс сложности $O(n^2)$. Чтобы построить более быстрый алгоритм, перемножим попарно $x-\omega^i$, затем перемножим попарно получившиеся $n/2$ полиномов и т. д., пока не останется два полинома q_1 и q_2, каждый из которых равен произведению половины полиномов $x-\omega^i$. Далее делим $p(x)$ на q_1 и q_2. Соответствующие остатки $r_1(x)$ и $r_2(x)$ имеют степень не более $n/2-1$ каждый. Для каждого корня ω^i, для которого q_1 делится без остатка на $x-\omega^i$, нахождение остатка от деления $p(x)$ на $x-\omega^i$ равносильно нахождению остатка от деления $r_1(x)$ на $x-\omega^i$. Аналогичное утверждение верно для каждого корня ω^i, для которого q_2 делится без остатка на $x-\omega^i$. Поэтому вычисление остатков от деления $p(x)$ на каждый из полиномов $x-\omega^i$ равносильно вычислению остатков от деления $r_1(x)$ и $r_2(x)$ на каждый из $n/2$ подходящих полиномов $x-\omega^i$. Рекурсивное применение этой тактики "разделяй и властвуй" гораздо эффективнее прямолинейного метода деления $p(x)$ на каждый полином $x-\omega^i$.

При перемножении полиномов $x-\omega^i$ можно ожидать, что произведения будут содержать результаты перекрестных умножений одночленов. Однако при подходящем упорядочении полиномов $x-\omega^i$

можно добиться, чтобы все произведения имели вид $x^l - \omega^t$, и это еще уменьшит время, затрачиваемое на умножения и деления полиномов. Изложим все эти идеи точнее.

Пусть $c_0, c_1, \ldots, c_{n-1}$ — перестановка элементов $\omega^0, \omega^1, \ldots, \omega^{n-1}$, которую мы конкретизируем позже. Определим полиномы q_{lm}, где $0 \leqslant m \leqslant k$ и l является целым кратным числа 2^m, $0 \leqslant l \leqslant 2^k - 1$:

$$q_{lm} = \prod_{j=l}^{l+2^m-1} (x - c_j).$$

Таким образом, $q_{0k} = (x - c_0)(x - c_1) \ldots (x - c_{n-1})$, $q_{l0} = x - c_l$ и в общем случае

$$q_{lm} = q_{l,m-1} q_{l+2^{m-1},m-1}.$$

Существует 2^{k-m} полиномов со вторым нижним индексом m, и каждый полином $x - c_l$ делит в точности один из них. Полиномы q_{lm} изображены на рис. 7.1. (См. также разд. 8.4 и 8.5.)

Наша цель — вычислить остаток от деления $p(x)$ на $q_{l0}(x)$ для каждого l. Для этого вычислим остатки от деления $p(x)$ на $q_{lm}(x)$ для каждого q_{lm}, начиная с $m = k - 1$ и кончая $m = 0$.

Допустим, что уже вычислены полиномы r_{lm} степени $2^m - 1$, остающиеся от деления $p(x)$ на $q_{lm}(x)$. (Можно считать, что $r_{0k} = p(x)$.) Поскольку $q_{lm} = q'q''$, где $q' = q_{l,m-1}$ и $q'' = q_{l+2^{m-1},m-1}$, мы утверждаем, что остаток от деления $p(x)$ на $q'(x)$ равен остатку от деления r_{lm} на $q'(x)$ и аналогичное утверждение верно и для $q''(x)$. Для до-

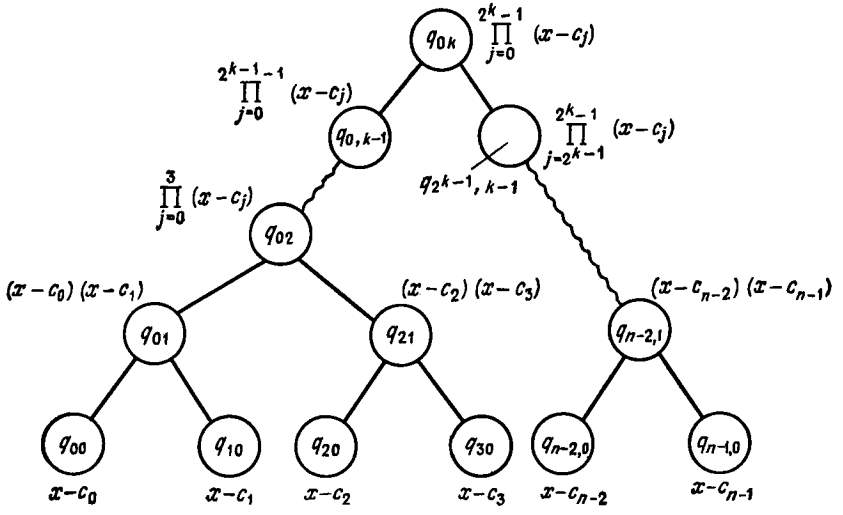


Рис. 7.1. Полиномы q_{lm}.

казательства положим

$$p(x) = h_1(x)\, q'(x) + r_{l,m-1},$$

где степень полинома $r_{l,m-1}$ не превосходит $2^{m-1}-1$. Так как

$$p(x) = h_2(x)\, q_{lm}(x) + r_{lm},$$

то

$$h_1(x)\, q'(x) + r_{l,m-1} = h_2(x)\, q_{lm}(x) + r_{lm}. \tag{7.5}$$

Разделим обе части равенства (7.5) на $q'(x)$; так как $h_1(x)\, q'(x)$ и $h_2(x)\, q_{lm}(x)$ делятся без остатка, то остаток от деления r_{lm} на $q'(x)$ равен $r_{l,m-1}$.

Таким образом, можно получить остатки от деления $p(x)$ на $q'(x)$ и $p(x)$ на $q''(x)$, разделив на $q'(x)$ и $q''(x)$ полином r_{lm} степени $2^{k-m}-1$, а не полином $p(x)$ степени 2^k-1. Этот метод выполнения делений сам по себе дает экономию времени. Но можно делать еще больше. Выбрав подходящий порядок элементов $c_0, c_1, \ldots, c_{n-1}$ для степеней ω, можно добиться, чтобы каждый полином q_{lm} имел вид $x^{2^m}-\omega^s$ при некотором s. Деление на такие полиномы выполняется особенно просто.

Лемма 7.2. *Пусть $n=2^k$ и ω — примитивный корень степени n из единицы. Пусть $[d_0 d_1 \ldots d_{k-1}]$ — двоичное представление целого числа j, где $0 \leqslant j < 2^k$[1]), а $\mathrm{rev}(j)$ — целое число с двоичным представлением $[d_{k-1}\, d_{k-2} \ldots d_0]$. Обозначим $c_j = \omega^{\mathrm{rev}(j)}$ и $q_{lm} = \prod_{j=l}^{l+2^m-1} (x - c_j)$. Тогда $q_{lm} = x^{2^m} - \omega^{\mathrm{rev}(l/2^m)}$.*

Доказательство. Доказательство проводится индукцией по m. Базис, т. е. случай $m=0$, тривиален, ибо по определению $q_{l0} = x - c_l = x - \omega^{\mathrm{rev}(l)}$. Для проведения шага индукции заметим, что при $m>0$

$$q_{lm} = q_{l,m-1} q_{l+2^{m-1},\, m-1} = \\ = (x^{2^{m-1}} - \omega^{\mathrm{rev}(l/2^{m-1})})\,(x^{2^{m-1}} - \omega^{\mathrm{rev}(l/2^{m-1}+1)}),$$

где $l/2^{m-1}$ — четное число между 0 и 2^k-1. Тогда

$$\omega^{\mathrm{rev}(l/2^{m-1}+1)} = \omega^{2^{k-1}+\mathrm{rev}(l/2^{m-1})} = -[\omega^{\mathrm{rev}(l/2^{m-1})}],$$

поскольку $\omega^{2^{k-1}} = \omega^{n/2} = -1$. Следовательно,

$$q_{lm} = x^{2^m} - \omega^{2\,\mathrm{rev}(l/2^{m-1})} = x^{2^m} - \omega^{\mathrm{rev}(l/2^m)},$$

поскольку $\mathrm{rev}(2l) = {}^1\!/_2\, \mathrm{rev}(l)$. □

Пример 7.1. Если $n=8$, то список $c_0, c_1, \ldots, c_7$ есть $\omega^0, \omega^4, \omega^2, \omega^6, \omega^1, \omega^5, \omega^3, \omega^7$. Полиномы q_{lm} иллюстрируются на рис. 7.2.

[1]) То есть $j = \sum_{i=0}^{k-1} d_{k-1-i} 2^i$.

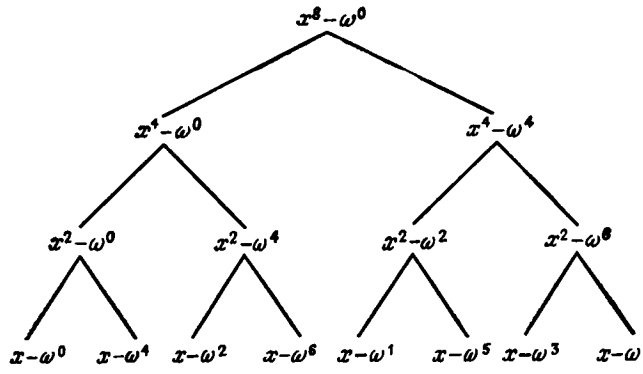


Рис. 7.2. Полиномы q_{lm} из леммы 7.2.

Покажем, как использовать полиномы q_{lm} для вычисления остатков. Вначале вычисляются остатки r_{02} и r_{42} от деления $p(x)$ на $x^4-\omega^0$ и $p(x)$ на $x^4-\omega^4$, где $p(x)=\sum_{j=0}^{7} a_j x^j$. Затем вычисляются остатки r_{01} и r_{21} от деления r_{02} на $x^2-\omega^0$ и r_{02} на $x^2-\omega^4$ и остатки r_{41} и r_{61} от деления r_{42} на $x^2-\omega^2$ и r_{42} на $x^2-\omega^6$. Наконец, вычисляются r_{00}, r_{10}, r_{20}, . . ., r_{70}, где r_{00} и r_{10}— остатки от деления r_{01} на $x-\omega^0$ и r_{01} на $x-\omega^4$, r_{20} и r_{30}— остатки от деления r_{21} на $x-\omega^2$ и r_{21} на $x-\omega^6$, и т. д.

Другие примеры применения этого подхода приведены в разд. 8.5. □

Доказав, что полиномы q_{lm} имеют вид x^s-c, покажем, что остаток от деления $p(x)$ на x^s-c найти легко.

Лемма 7.3. *Пусть*

$$p(x)=\sum_{j=0}^{2t-1} a_j x^j$$

и c — постоянная. Тогда остаток от деления $p(x)$ на x^t-c равен

$$r(x)=\sum_{j=0}^{t-1}(a_j+ca_{j+t})\,x^j.$$

Доказательство. Достаточно заметить, что $p(x)$ можно представить в виде

$$\left[\sum_{j=0}^{t-1} a_{j+t}x^j\right](x^t-c)+r(x). \qquad \square$$

Итак, остаток от деления произвольного полинома степени $2t-1$ на x^t-c можно найти за $O_A(t)$ шагов. Любой из известных алгорит-

begin

1. пусть $r_{0k} = \sum_{j=0}^{n-1} a_j x^j$;

   comment Полином представлен своими коэффициентами, так что в строке 1 не производится никаких вычислений. Остаток от деления полинома $\sum_{j=0}^{n-1} a_j x^j$ на q_{lm} будет представлен полиномом r_{lm};

2. **for** $m \leftarrow k-1$ **step** -1 **until** 0 **do**
3. **for** $l \leftarrow 0$ **step** 2^{m+1} **until** $n-1$ **do**
   begin
4. пусть $r_{l,m+1} = \sum_{j=0}^{2^{m+1}-1} a_j x^j$;

   comment Вычисляем остатки меньшей степени, используя коэффициенты полинома $r_{l,m+1}$;

5. $s \leftarrow \mathrm{rev}(l/2^m)$;
6. $r_{lm}(x) \leftarrow \sum_{j=0}^{2^m-1} (a_j + \omega^s a_{j+2^m}) x^j$;
7. $r_{l+2^m,m}(x) \leftarrow \sum_{j=0}^{2^m-1} (a_j + \omega^{s+n/2} a_{j+2^m}) x^j$

   end;

8. **for** $l \leftarrow 0$ **until** $n-1$ **do** $b_{\mathrm{rev}(l)} \leftarrow r_{l0}$

end

Рис. 7.3. Алгоритм быстрого преобразования Фурье.

мов тратит больше времени на вычисление остатка от деления произвольного полинома степени $2t-1$ на произвольный полином степени t.

Сформулируем полностью БПФ-алгоритм.

Алгоритм 7.1. Быстрое преобразование Фурье

Вход. Вектор $\mathbf{a}=[a_0, a_1, \ldots, a_{n-1}]^T$, где $n=2^k$ для некоторого целого числа k.

Выход. Вектор $F(\mathbf{a})=[b_0, b_1, \ldots, b_{n-1}]^T$, где $b_i = \sum_{j=0}^{n-1} a_j \omega^{ji}$ для $0 \leqslant i < n$.

Метод. См. рис. 7.3. □

Модификация алгоритма 7.1 для вычисления обратных преобразований состоит в замене ω на ω^{-1} (для этого в строках 6 и 7 меняются знаки показателей степеней элемента ω). Кроме того, в строке 8 $b_{\mathbf{rev}(l)}$ делится на n.

Пример 7.2. Пусть $n=8$ и, значит, $k=3$. При $m=2$ цикл в строках 3—7 выполняется только для $l=0$. В строке 4 $r_{03}=\sum_{j=0}^{7} a_j x^j$, в строке 5 $s=0$, в строках 6 и 7

$$r_{02}=(a_3+a_7)\,x^3+(a_2+a_6)\,x^2+(a_1+a_5)\,x+(a_0+a_4)$$

и

$$r_{42}=(a_3+\omega^4 a_7)\,x^3+(a_2+\omega^4 a_6)\,x^2+(a_1+\omega^4 a_5)\,x+(a_0+\omega^4 a_4).$$

Если $m=1$, то l принимает значения 0 и 4. Когда $l=0$, в строке 5 $s=0$. Тогда в строках 6 и 7

$$r_{01}=(a_1+a_3+a_5+a_7)\,x+(a_0+a_2+a_4+a_6)$$

и

$$r_{21}=(a_1+\omega^4 a_3+a_5+\omega^4 a_7)\,x+(a_0+\omega^4 a_2+a_4+\omega^4 a_6).$$

Когда $l=4$, в строке 5 $s=2$. В силу строк 6 и 7 и формулы для r_{42}, приведенной выше,

$$r_{41}=(a_1+\omega^2 a_3+\omega^4 a_5+\omega^6 a_7)\,x+(a_0+\omega^2 a_2+\omega^4 a_4+\omega^6 a_6),$$
$$r_{61}=(a_1+\omega^6 a_3+\omega^4 a_5+\omega^2 a_7)\,x+(a_0+\omega^6 a_2+\omega^4 a_4+\omega^2 a_6).$$

Наконец, если $m=0$, то l принимает значения 0, 2, 4 и 6. Например, при $l=4$ будет $s=1$, и, отправляясь от r_{41}, вычисляем

$$r_{40}=a_0+\omega a_1+\omega^2 a_2+\ldots+\omega^7 a_7.$$

К моменту входа в **for**-цикл в строке 8 полином r_{l0} всегда будет иметь степень 0, т. е. будет равен постоянной. Например, при $l=4$ будет $\mathbf{rev}(l)=1$ и r_{40} станет значением для b_1. Эта формула для b_1 согласуется с определением b_1. □

Покажем, что алгоритм 7.1 корректен.

Теорема 7.3. *Алгоритм* 7.1 *вычисляет дискретное преобразование Фурье.*

Доказательство. В строке 6 $r_{lm}=r_{l,m+1}/q_{lm}$, а в строке 7 $r_{l+2^m,m}=r_{l,m+1}/q_{l+2^m,m}$. Поэтому с помощью лемм 7.2 и 7.3 легко доказать индукцией по $k-m$, что r_{lm}— остаток от деления $\sum_{j=0}^{n-1} a_j x^j$ на q_{lm}. Тогда для $m=0$ лемма 7.3 гарантирует, что **for**-цикл в строке 8 присвоит всем b_i правильные значения (остатки, равные постоянным). □

Теорема 7.4. *Алгоритм* 7.1 *тратит время* $O_{\mathrm{A}}(n \log n)$.

Доказательство. Каждое выполнение строк 6 и 7 занимает $O_A(2^m)$ шагов. При фиксированном m цикл в строках 3—7 повторяется $n/2^{m+1}$ раз, занимая в целом время $O_A(n)$, не зависящее от m. Внешний цикл, начинающийся в строке 2, выполняется $\log n$ раз, занимая в целом время $O_A(n \log n)$. Цикл в строке 8 не требует выполнения арифметических операций. □

В теореме 7.4 мы предполагали, что число n фиксировано. Поэтому степени элемента ω, а также значения s и $\mathbf{rev}(l)$, получаемые из l в строках 5 и 8, можно вычислять заранее и использовать как постоянные в неветвящейся программе. Если же надо, чтобы n было параметром, можно вычислить степени элемента ω и запомнить их в виде таблицы за $O(n)$ шагов работы РАМ. Более того, хотя на вычисление s и $\mathbf{rev}(l)$ в строках 5 и 8 тратится $O(\log n)$ шагов, всего производится не более $3n$ таких вычислений, так что весь алгоритм при реализации на РАМ имеет временную сложность $O(n \log n)$.

Следствие 1. *Свертку* $\mathbf{a} \circledast \mathbf{b}$, *где* $\mathbf{a}$ *и* $\mathbf{b}$ — *векторы размерности* n, *можно вычислить за* $O_A(n \log n)$ *шагов.*

Доказательство. В силу теорем 7.1, 7.3 и 7.4. □

Следствие 2. *Положительно и отрицательно обернутые свертки векторов* $\mathbf{a}$ *и* $\mathbf{b}$ *можно вычислить за* $O_A(n \log n)$ *шагов.*

Алгоритм 7.1 был изложен для того, чтобы пояснить интуитивные соображения, лежащие в основе его конструкции. На самом деле при вычислении БПФ можно работать лишь с коэффициентами и тем самым упростить алгоритм. Это реализовано в алгоритме 7.2.

Алгоритм 7.2. Упрощенный БПФ-алгоритм

Вход. Вектор $\mathbf{a}=[a_0, a_1, \ldots, a_{n-1}]^T$, где $n=2^k$ для некоторого целого числа k.

Выход. Вектор $F(\mathbf{a})=[b_0, b_1, \ldots, b_{n-1}]^T$, где $b_i=\sum_{j=0}^{n-1} a_j \omega^{ij}$ при $0 \leqslant i < n$.

Метод. Применяем программу на рис. 7.4. Для облегчения понимания вводим временный массив S, чтобы хранить результаты предыдущего шага. На практике эти вычисления можно осуществлять на том же месте. □

Когда строка 3 выполняется первый раз, коэффициенты полинома $p(x)=\sum_{i=1}^{n-1} a_i x^i$ хранятся в массиве S. Во время первого выполнения строки 6 полином $p(x)$ делится на $x^{n/2}-1$ и $x^{n/2}-\omega^{n/2}$. Получаются остатки

$$\sum_{i=0}^{n/2-1} (a_i + a_{i+n/2}) x^i \quad \text{и} \quad \sum_{i=0}^{n/2-1} (a_i + \omega^{n/2} a_{i+n/2}) x^i.$$

```
   begin
1.   for i ← 0 until 2^k − 1 do R[i] ← a_i;
2.   for l ← 0 until k − 1 do
       begin
3.       for i ← 0 until 2^k − 1 do S[i] ← R[i];
4.       for i ← 0 until 2^k − 1 do
           begin
5.           пусть [d_0 d_1 ... d_{k-1}] — двоичное представление целого
               числа i;
6.           R[[d_0 ... d_{k-1}]] ← S[[d_0 ... d_{l-1} 0 d_{l+1} ... d_{k-1}]] +
               + ω^{[d_l d_{l-1} ... d_0 0 ... 0]} S[[d_0 ... d_{l-1} 1 d_{l+1} ... d_{k-1}]]
           end
       end;
7.   for i ← 0 until 2^k − 1 do
       b_{[d_0 ... d_{k-1}]} ← R[[d_{k-1} ... d_0]]
   end
```

Рис. 7.4. Упрощенный БПФ-алгоритм.

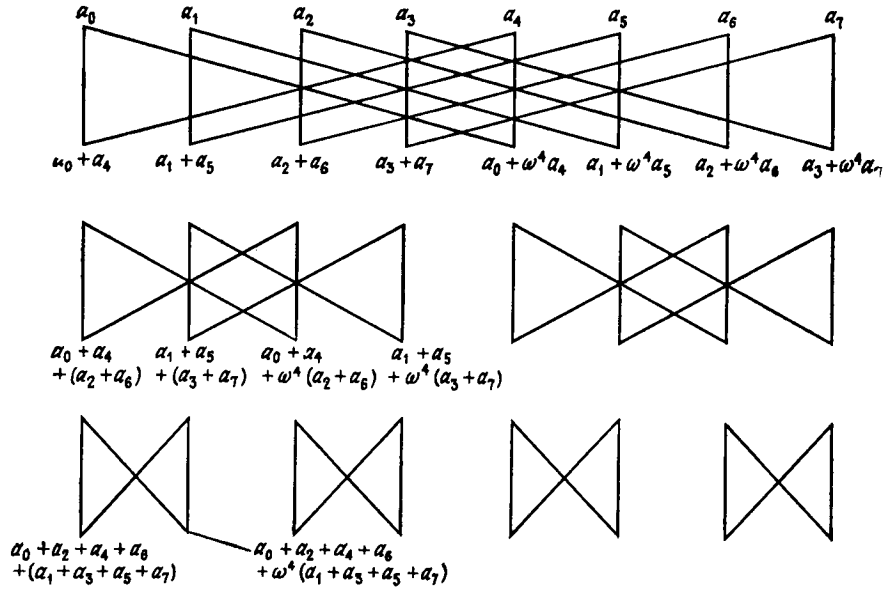


Рис. 7.5. Иллюстрация вычисления БПФ с помощью алгоритма 7.2. Некоторые полиномы-остатки опущены за недостатком места.

Их коэффициенты хранятся в массиве S при втором выполнении строки 3. Коэффициенты первого остатка занимают первую половину массива S, а второго остатка — вторую половину S.

При втором выполнении строки 6 каждый из этих двух полиномов-остатков делится на два полинома вида $x^{n/4}-\omega^s$. Это дает четыре остатка, каждый степени $n/4-1$. Их коэффициенты при третьем выполнении строки 3 запоминаются в S и т. д. Строка 7 реорган зует компоненты ответа так, чтобы они шли в правильном порядке. Она нужна потому, что корни из единицы были переставлены, чтобы устранить члены, получающиеся от перекрестных умножений при вычислении произведений полиномов $x-\omega^i$. Этот процесс при $n=8$ частично проиллюстрирован на рис. 7.5.

7.3. БПФ ПРИ ИСПОЛЬЗОВАНИИ БИТОВЫХ ОПЕРАЦИЙ

В тех приложениях, где преобразование Фурье проводится для упрощения вычисления свертки, часто нужен точный результат. Если элементы берутся из кольца вещественных чисел, их надо аппроксимировать с помощью конечного числа разрядов, и это порождает ошибки. Избежать этих ошибок можно, если производить вычисления в конечном поле [1]). Например, чтобы свернуть $\mathbf{a}=[a_0, a_1, a_2, 0, 0]^T$ и $\mathbf{b}=[b_0, b_1, b_2, 0, 0]^T$, можно взять 2 в качестве корня пятой степени из единицы и вычислять по модулю 31. Тогда преобразование векторов $\mathbf{a}$ и $\mathbf{b}$, попарные умножения и нахождение обратного преобразования дают точное значение $\mathbf{a}\circledast\mathbf{b}$ по модулю 31 [2]). При использовании конечного поля часто бывает трудно выбрать подходящее поле с подходящим корнем n-й степени из единицы. Поэтому мы будем пользоваться кольцом R_m целых чисел по модулю m, где m будет таким, чтобы в R_m был примитивный корень n-й степени из единицы [3]). Сразу не очевидно, что по данному n можно найти такие ω и m что ω — примитивный корень n-й степени из единицы в кольце вычетов по модулю m. Кроме того, не годятся слишком большие m, поскольку тогда вычисления по модулю m будут громоздкими. К счастью, если n — степень числа 2, то подходящее число m существует всегда, и оно равно примерно 2^n. В частности, мы покажем (теорема 7.5), что когда n и $\omega>1$ явля-

[1]) Определение поля дано в разд. 12.1.

[2]) Разумеется, компоненты ответа должны быть заключены между 0 и 30, ибо иначе нельзя будет восстановить их. В общем случае надо выбирать модули достаточно большими, чтобы можно было восстановить ответ.

[3]) До сих пор вы могли бы считать ω комплексным числом $e^{2\pi i/n}$, а арифметические операции — операциями в поле комплексных чисел. Но начиная отсюда, нужно считать ω целым числом, а все арифметические операции — операциями в конечном кольце вычетов по модулю m.

ются степенями числа 2, можно вычислять свертки в кольце вычетов по модулю $\omega^{n/2}+1$ с помощью преобразования Фурье, покомпонентного умножения и обратного преобразования. Сначала установим два предварительных результата — леммы 7.4 и 7.5. В них мы будем предполагать, что $R=(S, +, \cdot, 0, 1)$ — коммутативное кольцо и $n=2^k$, $k\geqslant 1$.

Лемма 7.4. *Для всякого* $a\in S$

$$\sum_{i=0}^{n-1} a^i = \prod_{i=0}^{k-1} (1+a^{2^i}).$$

Доказательство. Доказательство проводится индукцией по k. Базис, т. е. случай $k=1$, тривиален. Теперь заметим, что

$$\sum_{i=0}^{n-1} a^i = (1+a)\sum_{i=0}^{n/2-1} (a^2)^i. \tag{7.6}$$

Предположение индукции и подстановка a^2 вместо a дают

$$\sum_{i=0}^{n/2-1} (a^2)^i = \prod_{i=0}^{k-2} [1+(a^2)^{2^i}] = \prod_{i=1}^{k-1} [1+a^{2^i}]. \tag{7.7}$$

Подставляя (7.7) в правую часть равенства (7.6), получаем требуемое. □

Лемма 7.5. *Пусть* $m=\omega^{n/2}+1$, *где* $\omega\in S$, $\omega\neq 0$. *Тогда для* $1\leqslant p<n$

$$\sum_{i=0}^{n-1} \omega^{ip} \equiv 0 \pmod m.$$

Доказательство. В силу леммы 7.4 достаточно показать, что $1+\omega^{2^j p}\equiv 0 \pmod m$ для некоторого j, $0\leqslant j<k$. Пусть $p=2^s p'$, где p' нечетно. Очевидно, что $0\leqslant s<k$. Выберем j так, чтобы $j+s=k-1$. Тогда $1+\omega^{2^j p}=1+\omega^{2^{k-1}p'}=1+(m-1)^{p'}$. Но $m-1\equiv -1 \pmod m$ и p' нечетно, так что $(m-1)^{p'}\equiv -1 \pmod m$. Отсюда следует, что $1+\omega^{2^j p}\equiv 0 \pmod m$ для $j=k-1-s$. □

Теорема 7.5. *Пусть* n *и* ω *— положительные степени числа* 2 *и* $m=\omega^{n/2}+1$. *Пусть* R_m *— кольцо вычетов по модулю* m. *Тогда в кольце* R_m *элемент* n *имеет обратный (по модулю* m*) и* ω *— примитивный корень* n*-й степени из единицы.*

Доказательство. Так как n — степень числа 2, а m нечетно, то m и n взаимно просты. Поэтому n имеет обратный по модулю m [1]). Так как $\omega\neq 1$, то $\omega^n=\omega^{n/2}\,\omega^{n/2}\equiv(-1)(-1)=$

[1]) Это утверждение составляет одну из основных теорем теории чисел. В разд. 8.8 будет показано, что если a и b взаимно просты, то существуют такие целые числа x и y, что $ax+by=1$. Тогда $ax\equiv 1 \pmod b$. Полагая $b=m$ и $a=n$, получаем наше утверждение.

$=1 \pmod{(\omega^{n/2}+1)}$. Тогда из леммы 7.5 следует, что ω — примитивный корень n-й степени из единицы в R_m. □

Теорема 7.5 важна потому, что теорема о свертке верна для кольца целых чисел по модулю $2^{n/2}+1$. Если надо вычислить свертку двух n-мерных векторов с целыми компонентами, причем компоненты свертки заключены между 0 и $2^{n/2}$, мы можем быть уверены, что ответ будет точным. Если же компоненты свертки не заключены между 0 и $2^{n/2}$, то они точны по модулю $2^{n/2}+1$.

Сейчас мы уже почти готовы к тому, чтобы установить, сколько требуется битовых операций для вычисления свертки по модулю m. Сначала, однако, выясним, сколько битовых операций требуется для вычисления вычета целого числа по модулю m, так как это существенный шаг в подсчете числа битовых операций на основе числа арифметических операций по модулю m.

Пусть $m=\omega^p+1$ для некоторого целого числа p. Для вычисления a по модулю m применим обобщение приема "отбрасывания девяток" (нахождения вычета по модулю 9). Если число a разложено по основанию ω^p, т. е. записано в виде последовательности из l блоков по p цифр в каждом, то a по модулю m можно сосчитать, попеременно складывая и вычитая эти l блоков из p цифр.

Лемма 7.6. *Пусть* $m=\omega^p+1$ *и* $a=\sum_{i=0}^{l-1} a_i \omega^{pi}$, *где* $0 \leqslant a_i < \omega^p$ *для каждого* i. *Тогда*

$$a \equiv \sum_{i=0}^{l-1} a_i (-1)^i \pmod{m}.$$

Доказательство. Достаточно увидеть, что $\omega^p \equiv -1 \pmod{m}$. □

Заметим, что если число блоков l в лемме 7.6 фиксировано, то вычет числа a по модулю m можно найти за $O_Б(p \log \omega)$ битовых операций.

Пример 7.3. Пусть $n=4$, $\omega=2$ и $m=2^3+1$. Тогда, применяя лемму 7.6, положим $p=2$. Рассмотрим число a, запись которого по основанию 2 имеет вид 101100. В нашем случае $a_0=00$, $a_1=11$ и $a_2=10$. Вычисляем $a_0-a_1+a_2=-1$, а затем, прибавляя m, находим, что $a \equiv 4 \pmod 5$. Так как $a=44$, то результат верен. □

Лемма 7.6 дает эффективный метод вычисления a по модулю m. Она играет важную роль в следующей теореме, устанавливающей верхнюю границу на число битовых операций, требуемых для вычисления дискретного преобразования Фурье и обратного к нему.

Теорема 7.6. *Пусть* ω *и* n — *степени числа* 2 *и* $m=\omega^{n/2}+1$. *Пусть* $[a_0, a_1, \ldots, a_{n-1}]^T$ — *вектор с целочисленными компонентами, где* $0 \leqslant a_i < m$ *для каждого* i. *Тогда дискретное преобразование Фурье*

для $[a_0, a_1, \ldots, a_{n-1}]^T$ *и обратное к нему можно вычислить по модулю* m *за* $O_Б(n^2 \log n \log \omega)$ *шагов.*

Доказательство. Применить алгоритм 7.1 или 7.2. Для обратного преобразования подставить ω^{-1} вместо ω и умножить каждый результат на n^{-1}. Целое число по модулю m можно представить цепочкой из $b=((n/2)\log\omega)+1$ символов 0 и 1. Так как $m=2^{b-1}+1$, то вычеты по модулю m можно представить в виде двоичных чисел от 00. . .0 до 100. . .0.

В алгоритме 7.1 участвует сложение целых чисел по модулю m и умножение по модулю m целого числа на степень числа ω. Эти операции выполняются за время $O(n \log n)$. В силу леммы 7.6 сложение по модулю m занимает $O_Б(b)$ шагов, где $b=((n/2)\log\omega)+1$. Умножение на ω^p, $0\leqslant p<n$, эквивалентно сдвигу влево на $p \log \omega$ разрядов, поскольку ω — степень числа 2. Получающееся целое число содержит не более $3b$—2 разрядов, так что по лемме 7.6 сдвиг и последующее вычисление вычета занимают $O_Б(b)$ шагов. Таким образом, прямое преобразование Фурье можно найти за время $O_Б(bn \log n)$, или $O_Б(n^2 \log n \log \omega)$.

В обратном преобразовании участвует умножение на ω^{-p} и n^{-1}. Так как $\omega^p\omega^{n-p}\equiv 1 \pmod m$, то $\omega^{n-p}\equiv\omega^{-p} \pmod m$. Следовательно, вместо умножения на ω^{-p} можно умножать на ω^{n-p}, что эквивалентно сдвигу влево на $(n-p)\log\omega$ разрядов, причем получающееся целое число содержит не более $3b$—2 разрядов. Снова по лемме 7.6 вычеты можно найти за $O_Б(b)$ шагов. Наконец, рассмотрим умножение на n^{-1}. Если $n=2^k$, то оно сводится к сдвигу влево на $n \log \omega - k$ разрядов (в результате получается число не более чем с $3b$—2 двоичными разрядами) и вычислению вычета по лемме 7.6. Таким образом, нахождение обратного преобразования Фурье также требует $O_Б(n^2 \log n \log \omega)$ шагов. □

Пример 7.4. Пусть $\omega=2$, $n=4$ и $m=5$. Вычислим преобразование Фурье вектора $[a_0, a_1, a_2, a_3]^T$, где $a_i=i$. Так как $a_i<5$ для каждого i, то можно ожидать, что этот вектор можно будет восстановить, проведя вычисления по модулю m. Для представления чисел мы используем три бита, но в действительности у нас будут лишь 000, . . ., 100, если не считать промежуточных результатов.

В соответствии с алгоритмом 7.1 мы должны вычислить коэффициенты, указанные на рис. 7.6, где вместо ω подставлено число 2.

a_0	a_1	a_2	a_3
a_0+a_2	a_1+a_3	a_0+4a_2	a_1+4a_3
$a_0+a_2+(a_1+a_3)$ $=a_0+a_1+a_2+a_3$ $=b_0$	$a_0+a_2+4(a_1+a_3)$ $=a_0+4a_1+a_2+4a_3$ $=b_2$	$a_0+4a_2+2(a_1+4a_3)$ $=a_0+2a_1+4a_2+8a_3$ $=b_1$	$a_0+4a_2+8(a_1+4a_3)$ $=a_0+8a_1+4a_2+2a_3$ $=b_3$

Рис. 7.6. Вычисление быстрого преобразования Фурье для $n=4$.

Значения переменных и выражений на рис. 7.6 для нашего примера таковы:

$$\begin{array}{cccc} a_0 = 000 & a_1 = 001 & a_2 = 010 & a_3 = 011 \\ a_0 + a_2 = 010 & a_1 + a_3 = 100 & a_0 + 4a_2 = 011 & a_1 + 4a_3 = 011 \\ b_0 = 001 & b_2 = 011 & b_1 = 100 & b_3 = 010 \end{array}$$

Преобразованием вектора $[0, 1, 2, 3]^T$ будет, таким образом, вектор $[1, 4, 3, 2]^T$ по модулю 5. Рассмотрим последний элемент b_3 в нижней строке. Он вычисляется по двум последним элементам в средней строке:

Берем a_1+4a_3	011
Сдвигаем на три разряда влево (умножаем на 8)	11000
Расщепляем на три блока по 2 разряда	1 10 00
Складываем первый и третий блоки и вычитаем второй	—1
Прибавляем $m=5$	100
Прибавляем $a_0+4a_2=011$	111
Вычитаем m	010

Для нахождения обращения заметим, что $2^{-1} \equiv 8 \pmod 5$, $4^{-1} \equiv 4 \pmod 5$ и $8^{-1} \equiv 2 \pmod 5$. Таким образом, формулы для обратного преобразования можно вывести из рис. 7.6, поменяв местами a_i и b_i, а также 2 и 8. Это вычисление выглядит так:

$$\begin{array}{cccc} b_0 = 001 & b_1 = 100 & b_2 = 011 & b_3 = 010 \\ b_0 + b_2 = 100 & b_1 + b_3 = 001 & b_0 + 4b_2 = 011 & b_1 + 4b_3 = 010 \\ 4a_0 = 000 & 4a_2 = 011 & 4a_1 = 100 & 4a_3 = 010 \end{array}$$

Наконец, делим каждый ответ на 4 (умножаем на 4, ибо $4^{-1} \equiv 4 \pmod 5$), и получаем $[0, 1, 2, 3]^T$ в качестве $[a_0, a_1, a_2, a_3]^T$. □

Следствие теоремы 7.6. *Пусть на вычисление произведения двух k-разрядных двоичных целых чисел тратится $O_Б(M(k))$ шагов. Пусть* **a** *и* **b** *будут n-мерными векторами длины n с целочисленными компонентами между* 0 *и* ω^n, *где n и ω — степени числа* 2. *Тогда свертку* $\mathbf{a} \circledast \mathbf{b}$, *а также положительно и отрицательно обернутые свертки векторов* **a** *и* **b** *по модулю* ω^n+1 *можно вычислить за время*

$$O_Б(\text{MAX}[n^2 \log n \log \omega,\ nM(n \log \omega)]).$$

Первая величина здесь представляет время вычисления преобразований, а вторая — выполнения $2n$ умножений $(n \log \omega + 1)$-разрядных двоичных целых чисел. Наилучшее известное значение $M(k)$ равно $k \log k \log \log k$ (разд. 7.5). При этом значении вторая величина больше первой, так что для вычисления соответствующей свертки требуется $O_Б(n^2 \log n \log \log n \log \omega \log \log \omega \log \log \log \omega)$ шагов.

7.4. ПРОИЗВЕДЕНИЕ ПОЛИНОМОВ

Задача умножения двух полиномов от одной переменной по существу совпадает с задачей нахождения свертки двух последовательностей, а именно

$$\left(\sum_{i=0}^{n-1} a_i x^i\right)\left(\sum_{j=0}^{n-1} b_j x^j\right) = \sum_{k=0}^{2n-2} c_k x^k, \text{ где } c_k = \sum_{m=0}^{n-1} a_m b_{k-m}.$$

Как и раньше, a_p и b_p считаются нулями, если $p<0$ или $p\geqslant n$. Напомним, что коэффициент c_{2n-1} должен быть нулем. Поэтому имеем дополнительные следствия теоремы 7.4.

Следствие 3 теоремы 7.4. *Коэффициенты произведения двух полиномов степени n можно вычислить за $O_A(n \log n)$ шагов.*

Доказательство. В силу следствия 1 теоремы 7.4 и соображений, приведенных выше. □

Следствие 4 теоремы 7.4. *Допустим, что произведение двух k-разрядных двоичных целых чисел можно вычислить за $M(k)$ шагов. Пусть*

$$p(x) = \sum_{i=0}^{n-1} a_i x^i, \quad q(x) = \sum_{j=0}^{n-1} b_j x^j,$$

где a_i и b_j — целые числа между 0 и $\omega^{n/2}/\sqrt{n}$ для всех i и j, а n и ω — степени числа 2. Тогда коэффициенты полинома $p(x)q(x)$ можно вычислить за $O_Б(\text{MAX}[n^2 \log n \log \omega, nM(n \log \omega)])$ шагов.

Доказательство. В силу теоремы 7.4 и следствия теоремы 7.6. □

Снова заметим, что в следствии 4 доминирует вторая величина.

Фактически теорема 7.1 допускает такую интерпретацию. Предположим, что $p(x)$ и $q(x)$ — полиномы степени n—1. Можно вычислить полиномы p и q в любых $2n$—1 или более точках, скажем $c_0, c_1, \ldots$, и затем перемножить $p(c_j)$ и $q(c_j)$, чтобы получить значения pq в тех же точках. По этим значениям можно интерполяцией построить единственный полином степени $2n$—2. Он и будет произведением $p(x)\,q(x)$.

Когда преобразование Фурье применяют к вычислению свертки (или, что эквивалентно, к умножению полиномов), выбирают $c_j = \omega^j$, где ω — примитивный корень степени $2n$ из единицы. Далее находят преобразования Фурье полиномов p и q (т. е. вычисляют p и q в точках $c_0, c_1, \ldots$), попарно перемножают результаты этих преобразований (т. е. умножают $p(c_j)$ на $q(c_j)$, чтобы получить значение произведения в точке c_j), и вычисляют pq, применяя обратное

преобразование. Лемма 7.1 гарантирует, что это обращение на самом деле дает формулу для интерполяции. Иными словами, мы действительно восстанавливаем полином по его значениям в точках $\omega^0, \omega^1, \ldots, \omega^{2n-1}$.

7.5. АЛГОРИТМ ШЁНХАГЕ — ШТРАССЕНА ДЛЯ УМНОЖЕНИЯ ЦЕЛЫХ ЧИСЕЛ

Теперь изучим важное приложение теоремы о свертке — алгоритм быстрого битового умножения целых чисел. В разд. 2.6 мы видели, как умножить два n-разрядных двоичных числа за $O(n^{\log_3})$ шагов, разбивая каждое двоичное число на два $(n/2)$-разрядных числа. Этот метод можно обобщить, разбивая числа на b блоков по l разрядов в каждом. Если рассматривать эти b блоков как коэффициенты полинома, получатся выражения, аналогичные тем, что встречались в (2.4). Чтобы определить коэффициенты произведения полиномов, вычисляем полиномы на некотором подходящем множестве точек, перемножаем найденные значения и интерполируем. Выбрав в качестве точек, в которых вычисляются полиномы, примитивные корни из единицы, можно воспользоваться преобразованием Фурье и теоремой о свертке. Сделав b функцией от n и применив рекурсию, можно умножить два n-разрядных двоичных числа за $O_Б(n \log n \log\log n)$ шагов.

Для простоты ограничимся случаем, когда n — степень числа 2. В случае когда n не является степенью числа 2, нужно добавить к началам входных векторов подходящее количество нулей, чтобы n стало степенью числа 2 (это увеличивает только мультипликативную постоянную). Кроме того, мы вычислим произведение двух n-разрядных двоичных чисел по модулю 2^n+1. Чтобы найти точное значение произведения двух n-разрядных двоичных чисел, надо добавить нули к началам и умножать $2n$-разрядные числа по модулю $2^{2n}+1$, тем самым увеличивая время опять не более чем в постоянное число раз.

Пусть u и v — двоичные целые числа между 0 и 2^n, которые надо перемножить по модулю 2^n+1. Заметим, что двоичное представление числа 2^n занимает $n+1$ разрядов. Если число u или v равно 2^n, то оно представляется специальным символом -1, и в этом особом случае умножение выполняется легко: если $u=2^n$, то uv по модулю 2^n+1 получается путем вычисления 2^n+1-v по модулю 2^n+1.

Допустим, что $n=2^k$, и положим $b=2^{k/2}$, если k четно, и $b=2^{(k-1)/2}$ в противном случае. Пусть $l=n/b$. Заметим, что $l \geqslant b$ и l делится на b без остатка. Первый шаг состоит в разбиении u и v на b блоков по l битов в каждом. Таким образом,

$$u = u_{b-1}2^{(b-1)l} + \ldots + u_1 2^l + u_0,$$
$$v = v_{b-1}2^{(b-1)l} + \ldots + v_1 2^l + v_0.$$

Произведение uv представимо в виде

$$uv = y_{2b-2}2^{(2b-2)l} + \ldots + y_1 2^l + y_0, \qquad (7.8)$$

где

$$y_i = \sum_{j=0}^{b-1} u_j v_{i-j}, \qquad 0 \leqslant i < 2b.$$

(Для $j<0$ или $j>b-1$ берем $u_j=v_j=0$. Член y_{2b-1} равен 0 и включен только для симметрии.)

Произведение uv можно вычислить с помощью теоремы о свертке. Перемножение преобразований Фурье требует $2b$ умножений. Применяя обернутую свертку, можно уменьшить число умножений до b. Именно по этой причине мы вычисляем uv по модулю 2^n+1. Так как $bl=n$, то $2^{bl} \equiv -1 \pmod{2^n+1}$. Отсюда в силу (7.8) и леммы 7.6, взяв uv по модулю 2^n+1, находим, что

$$uv \equiv (w_{b-1}2^{(b-1)l} + \ldots + w_1 2^l + w_0) \pmod{2^n+1},$$

где $w_i = y_i - y_{b+i}$, $0 \leqslant i < b$.

Так как произведение двух l-разрядных двоичных чисел меньше 2^{2l}, а y_i и y_{b+i} — это суммы, составленные из $i+1$ и $b-(i+1)$ таких произведений соответственно, то $w_i = y_i - y_{b+i}$ удовлетворяет неравенствам $-(b-1-i)\,2^{2l} < w_i < (i+1)\,2^{2l}$. Следовательно, w_i может принимать не более $b2^{2l}$ значений. Если мы сможем вычислить все w_i по модулю $b2^{2l}$, то сможем вычислить uv по модулю 2^n+1, сделав $O(b \log (b2^{2l}))$ дополнительных шагов для сложения b экземпляров w_i с необходимыми сдвигами.

Чтобы вычислить все w_i по модулю $b2^{2l}$, вычисляем эти w_i дважды — по модулю b и по модулю $2^{2l}+1$. Пусть w_i' — это w_i по модулю b, а w_i'' — это w_i по модулю $2^{2l}+1$. Так как b — степень числа 2, а число $2^{2l}+1$ нечетно, то b и $2^{2l}+1$ взаимно просты. Поэтому w_i можно получить из w_i'' и w_i' по формуле

$$w_i = (2^{2l}+1)\,((w_i' - w_i'') \bmod b) + w_i''\,^{1}),$$

учитывая, что w_i лежит между $-(b-1-i)\,2^{2l}$ и $(i+1)\,2^{2l}$. Вычисление w_i по w_i' и w_i'' требует $O(l+\log b)$ шагов для каждого w_i, что дает в целом $O(bl+b \log b)$, или $O(n)$ шагов.

Значения w_i по модулю b вычисляются так: берем $u_i' = u_i$ по модулю b и $v_i' = v_i$ по модулю b и формируем два $(3b \log b)$-разрядных числа $\hat{u}$ и $\hat{v}$, как показано на рис. 7.7. Произведение $\hat{u}\hat{v}$ вычисляется с помощью алгоритма из разд. 2.6 не более чем за $O((3b \log b)^{1,6})$ шагов, т. е. менее чем за $O(n)$ шагов. Далее, $\hat{u}\hat{v} =$

---

[1]) Если для взаимно простых p_1 и p_2 выполнены сравнения $w \equiv q_1 \pmod{p_1}$, $w \equiv q_2 \pmod{p_2}$ и $0 \leqslant w < p_1p_2$, то $w = p_2(p_2^{-1} \bmod p_1)(q_1 - q_2 \bmod p_1) + q_2$. Пусть $p_1 = b$ и $p_2 = 2^{2l}+1$. Поскольку b — степень числа 2 и $b \leqslant 2^{2l}$, то b делит 2^{2l} и, значит, элементом, обратным к $2^{2l}+1$ по модулю b, будет 1.

$$\hat{u} = u'_{b-1}00\ldots0u'_{b-2}00\ldots0u'_{b-3}\ldots00\ldots0u'_0$$
$$\hat{v} = v'_{b-1}00\ldots0v'_{b-2}00\ldots0v'_{b-3}\ldots00\ldots0v'_0$$

Рис. 7.7. Изображение составных чисел, используемых при вычислении w по модулю b. В каждом блоке, состоящем из нулей, содержатся $2\log b$ нулей.

$=\sum_{i=0}^{2b-1} y'_i 2^{(3\log b)i}$, где $y'_i=\sum_{j=0}^{2b-1} u'_j v'_{i-j}$. Заметим, что $y'_i < 2^{3\log b}$, так что все y_i легко восстановить по произведению $\hat{u}\hat{v}$. Тогда w_i по модулю b можно найти, вычисляя $y'_i - y'_{b+i}$ по модулю b.

Вычислив все w_i по модулю b, вычисляем w_i по модулю $2^{2l}+1$ с помощью обернутой свертки. Для этого надо взять преобразование Фурье, выполнить попарные умножения и найти обратное преобразование. Пусть $\omega=2^{4l/b}$ и $m=2^{2l}+1$. По теореме 7.5 элементы ω и b имеют обратные по модулю m, а ω — примитивный корень b-й степени из единицы. Поэтому отрицательно обернутая свертка векторов $[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]$ и $[v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]$, где $\psi=2^{2l/b}$ (ψ — корень степени $2b$ из единицы), равна $[(y_0-y_b), \psi(y_1-y_{b+1}), \ldots, \psi^{b-1}(y_{b-1}-y_{2b-1})] \pmod{2^{2l}+1}$, где $y_i=\sum_{j=0}^{b-1} u_j u_{i-j}$ для $0\leqslant i\leqslant 2b-1$. Значения w_i по модулю $2^{2l}+1$ можно получить соответствующим сдвигом. Весь алгоритм таков:

Алгоритм 7.3. Алгоритм Шёнхаге — Штрассена для умножения целых чисел

Вход. Два n-разрядных двоичных целых числа u и v, где $n=2^k$.

Выход. $(n+1)$-разрядное произведение uv по модулю 2^n+1.

Метод. Если n мало, умножьте u на v по модулю 2^n+1 вашим любимым алгоритмом. Для большого n положим $b=2^{k/2}$, если k четно, и $b=2^{(k-1)/2}$, если k нечетно. Пусть $l=n/b$. Представим u и v в виде $\sum_{i=0}^{b-1} u_i 2^{il}$ и v в виде $\sum_{i=0}^{b-1} v_i 2^{il}$, где u_i и v_i — целые числа между 0 и 2^l-1 (т. е. числа u_i — это блоки, составленные из l разрядов числа u; аналогично v_i — блоки из l разрядов числа v).

1. Вычисляем преобразование Фурье по модулю $2^{2l}+1$ векторов

$$[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]^T \quad \text{и} \quad [v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]^T,$$

где $\psi=2^{2l/b}$ и ψ^2 берется в качестве примитивного корня b-й степени из единицы.

2. Вычисляем по модулю $2^{2l}+1$ покомпонентное произведение преобразований Фурье, полученных на шаге 1, при помощи алгоритма 7.3, который рекурсивно применяется для вычисления про-

изведения каждой пары соответствующих компонент. (Ситуация, когда одно из чисел равно 2^{2l}, рассматривается как легкий частный случай.)

3. Вычисляем обратное преобразование Фурье по модулю $2^{2l}+1$ вектора, равного покомпонентному произведению, полученному на шаге 2. Результатом будет вектор $[w_0, \psi w_1, \ldots, \psi^{b-1} w_{b-1}]^T$ по модулю $2^{2l}+1$, где w_i обозначает i-ю компоненту отрицательно обернутой свертки векторов $[u_0, u_1, \ldots, u_{b-1}]^T$ и $[v_0, v_1, \ldots, v_{b-1}]^T$. Вычисляем $w_i'' = w_i$ по модулю $2^{2l}+1$, умножая $\psi^i w_i$ на ψ^{-i} по модулю $2^{2l}+1$.

4. Вычисляем $w_i' = w_i \bmod b$ следующим образом.

   (а) Пусть $u_i' = u_i$ по модулю b и $v_i' = v_i$ по модулю b для $0 \leqslant i < b$.

   (б) Построим числа $\hat{u}$ и $\hat{v}$, выписывая в цепочки числа u_i' и v_i' и вставляя между ними $2 \log b$ нулей, т. е.

$$\hat{u} = \sum_{i=0}^{b-1} u_i' \, 2^{(3 \log b) i} \quad \text{и} \quad \hat{v} = \sum_{i=0}^{b-1} v_i' \, 2^{(3 \log b) i}.$$

   (в) Вычисляем произведение $\hat{u}\hat{v}$ с помощью алгоритма из разд. 2.6.

   (г) Произведение $\hat{u}\hat{v}$ имеет вид $\sum_{i=0}^{2b-1} y_i' \, 2^{(3 \log b) i}$, где $y_i' = \sum_{j=0}^{2b-1} u_j' \, v_{i-j}'$.

   Числа w_i по модулю b можно восстановить по этому произведению, вычисляя $w_i' = y_i' - y_{b+i}'$ по модулю b для $0 \leqslant i < b$.

5. Вычисляем точные значения w_i по формуле

$$w_i = (2^{2l}+1)\left((w_i' - w_i'') \bmod b\right) + w_i'',$$

учитывая, что w_i лежит между $-(b-1-i)\, 2^{2l}$ и $(i+1)\, 2^{2l}$.

6. Вычисляем $\sum_{j=0}^{b-1} w_j \, 2^{lj}$ по модулю (2^n+1). Это и есть искомый результат. □

Теорема 7.7. *Алгоритм* 7.3 *вычисляет* uv *по модулю* (2^n+1).

Доказательство. По теореме 7.2 шаги 1—3 алгоритма 7.3 правильно вычисляют значения w_i по модулю $2^{2l}+1$. В качестве упражнения предлагаем доказать, что шаг 4 вычисляет w_i по модулю b, а шаг 5 — w_i по модулю $b(2^{2l}+1)$, т. е. точное значение w_i. □

Теорема 7.8. *Алгоритм* 7.3 *тратит* $O_{\text{Б}}(n \log n \log \log n)$ *времени.*

Доказательство. В силу следствия теоремы 7.6 шаги 1—3 выполняются за время $O_{\text{Б}}[bl \log b + bM(2l)]$, где $M(m)$ —

время умножения двух m-разрядных двоичных целых чисел путем рекурсивного применения этого алгоритма. На шаге 4 формируются и умножаются числа $\hat{u}$ и $\hat{v}$ длины $3b \log b$ за $O_{\text{Б}}[(3b \log b)^{1,59}]$ шагов. Так как $(3b \log b)^{1,59} < b^2$ для достаточно большого b, то временем, затрачиваемым на шаг 4, можно пренебречь, ибо шаги 1 и 3 занимают $O_{\text{Б}}(b^2 \log b)$ времени. Шаги 5 и 6 занимают оба $O(n)$ времени, так что и ими можно пренебречь.

Поскольку $n = bl$ и $b \leqslant \sqrt{n}$, то

$$M(n) \leqslant cn \log n + bM(2l) \tag{7.9}$$

для некоторой постоянной c и достаточно больших n. Пусть $M'(n) = M(n)/n$. Тогда (7.9) можно записать в виде

$$M'(n) \leqslant c \log n + 2M'(2l). \tag{7.10}$$

Поскольку $l \leqslant 2\sqrt{n}$, то

$$M'(n) \leqslant c \log n + 2M'(4\sqrt{n}), \tag{7.11}$$

откуда следует, что $M'(n) \leqslant c' \log n \log \log n$ при подходящем выборе постоянной c'. Чтобы убедиться в этом, подставим в (7.11) $c' \log (4\sqrt{n}) \log \log (4\sqrt{n})$ вместо $M'(4\sqrt{n})$. Очевидные преобразования дадут неравенство

$$M'(n) \leqslant c \log n + 4c' \log \left(2 + \tfrac{1}{2} \log n\right) + c' \log n \log \left(2 + \tfrac{1}{2} \log n\right).$$

Для больших n выполняется $2 + {}^1/_2 \log n \leqslant {}^3/_4 \log n$, и потому

$$M'(n) \leqslant c \log n + 4c' \log \frac{2}{3} + 4c' \log \log n + c' \log \frac{2}{3} \log n + c' \log n \log \log n. \tag{7.12}$$

Для больших n и для достаточно большой постоянной c' первые три слагаемых не больше абсолютной величины четвертого, а оно отрицательно. Таким образом, $M'(n) \leqslant c' \log n \log \log n$. Отсюда заключаем, что $M(n) \leqslant c' n \log n \log \log n$. □

УПРАЖНЕНИЯ

7.1. Чему равны примитивные корни n-й степени из единицы в кольце комплексных чисел при $n = 3, 4, 5$?

7.2. Покажите, как реализовать алгоритм 7.2 без временного массива S.

7.3. Вычислите дискретное преобразование Фурье следующих последовательностей в кольце комплексных чисел:

(а) [0, 1, 2, 3],

(б) [1, 2, 0, 2, 0, 0, 0, 1].

7.4. Обобщите алгоритм быстрого преобразования Фурье на случай, когда n не является степенью числа 2.

Определение. Тройка целых чисел (ω, n, m) называется *допустимой*, если ω и n имеют обратные и ω — примитивный корень n-й степени из единицы в кольце целых чисел по модулю m.

7.5. Какие из следующих троек допустимы?
(а) (3, 4, 5), (б) (2, 6, 21), (в) (2, 6, 7).

7.6. Покажите, что если (ω, n, m) — допустимая тройка, то $\omega^n \equiv 1 \pmod m$ и $\omega^p \not\equiv 1$, если $1 \leqslant p < n$.

***7.7.** Покажите, что если n — степень числа 2, $2^n \equiv 1 \pmod m$ и числа $2^p - 1$ и m взаимно просты при $1 \leqslant p < n$, то $(2, n, m)$ — допустимая тройка.

****7.8.** Покажите, что если m — простое число, а ω — произвольное положительное целое число, то найдется такое число n, что (ω, n, m) — допустимая тройка.

***7.9.** Покажите, что если a и b взаимно просты, то $ac \equiv 1 \pmod b$ для некоторого числа c, и обратно. Докажите, что c по модулю b единственно.

7.10. Найдите $(10101110011110)_2$ по модулю 2^5+1.

7.11. Пусть t — целое число, заданное своим десятичным представлением. Покажите, что если сложить все цифры числа t, затем сложить все цифры результата и т. д. до тех пор, пока не останется одна цифра; то в конце концов получится t по модулю 9.

7.12. С помощью теоремы о свертке вычислите свертку последовательностей [1, 2, 3, 4] и [4, 3, 2, 1] по модулю 17, используя допустимую тройку (2, 8, 17).

7.13. Вычислите по алгоритму 7.3 произведение двоичных чисел $(1011011)_2$ и $(10001111)_2$.

7.14. Покажите, что в результате извлечения квадратного корня из $n = 2^{2^k}$ получится $\log \log n$ раз число 2.

****7.15.** Разработайте алгоритм быстрого умножения целых чисел на основе свертки, а не обернутой свертки. Каково асимптотическое время работы вашего алгоритма?

***7.16.** *Циклической разностью* $\Delta\mathbf{a}$ вектора $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ называется вектор $[a_0 - a_{n-1}, a_1 - a_0, a_2 - a_1, \ldots, a_{n-1} - a_{n-2}]^T$. Пусть

$$F(\mathbf{a}) = [a'_0, a'_1, \ldots, a'_{n-1}]^T.$$

Покажите, что $F(\Delta\mathbf{a}) = [0, a'_1(1-\omega), a'_2(1-\omega^2), \ldots, a'_{n-1}(1-\omega^{n-1})]$, где ω — соответствующий корень n-й степени из единицы.

****7.17.** С помощью упр. 7.16 покажите, что из $X(n) - X(n-1) = n$ и $X(0) = 0$ следует $X(n) = n(n+1)/2$.

*7.18. *Циркулянтом* называется матрица, в которой каждая строка получается из строки, стоящей над ней, циклическим сдвигом на одну позицию вправо. Например, матрица

$$\begin{bmatrix} a & b & c \\ c & a & b \\ b & c & a \end{bmatrix}$$

есть (3×3)-циркулянт. Покажите, что вычисление дискретного преобразования Фурье n-мерного вектора, где n — простое число, эквивалентно умножению на $((n-1)\times(n-1))$-циркулянт.

**7.19. Покажите, что преобразование Фурье над конечным полем для простого числа n можно вычислить за $O_{\text{А}}(n \log n)$ шагов.

*7.20. Рассмотрите представление полинома значениями всех его производных в некоторой точке. Линейно ли преобразование коэффициентов полинома в значения его производных?

**7.21. Дайте "физическое" объяснение обернутой свертки в терминах операций над полиномами.

*7.22. Используйте БПФ для построения алгоритма сложности $O(n \log n)$, который умножал бы тёплицеву матрицу на вектор. Сравните ваш алгоритм с решением упр. 6.26(б).

Проблема для исследования

7.23. Найдите более быстрые алгоритмы умножения целых чисел и дискретного преобразования Фурье. Или, наоборот, покажите, что алгоритм Шёнхаге — Штрассена или БПФ — наилучший из возможных при некоторых ограничениях на модель вычислений. Развивая идеи Кука, Аандераа [1969], Патерсон, Фишер, Мейер [1974] показали, что при определенных ограничениях битовое умножение целых чисел требует $O_{\text{Б}}((n \log n)/(\log \log n))$ шагов. Аналогично Моргенштерн [1973] показал, что при определенных ограничениях дискретное преобразование Фурье требует $O_{\text{А}}(n \log n)$ шагов.

Замечания по литературе

Кули, Льюис, Уелч [1967] указывают, что способ быстрого преобразования Фурье описан еще в книге Рунге, Кёнига [1924]. Его применяли Даниелсон, Ланцош [1942] и Гуд [1958, 1960]. В фундаментальной работе Кули, Тьюки [1965] проясняется природа этого метода. Николсон [1971] дал алгебраическое описание для него. Трактовка БПФ как задачи деления полиномов, принятая нами, принадлежит Фидуччиа [1972]. Ввиду важности этого алгоритма для вычислений много внимания уделялось его эффективной реализации (см., например, работу Джентльмена, Санде [1966] и многочисленные статьи в сборнике под редакцией Рабинера, Рейдера [1972]). Алгоритм умножения целых чисел разработан Шёнхаге, Штрассеном [1971]. Упр. 7.18 и 7.19 взяты у Рейдера [1968].

8 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ ЧИСЛАМИ И ПОЛИНОМАМИ

Арифметические операции над целыми числами и полиномами целесообразно изучать вместе потому, что многие алгоритмы, работающие с целыми числами, по существу совпадают с алгоритмами, работающими с полиномами от одной переменной. Это верно не только для таких операций, как умножение и деление, но также и для более сложно описываемых операций. Например, нахождение вычета целого числа по модулю, задаваемому другим целым числом, эквивалентно вычислению полинома в точке. Представление целого числа его вычетами эквивалентно представлению полинома его значениями в нескольких точках. Восстановление целого числа по его вычетам ("китайская теорема об остатках") эквивалентно интерполированию полинома.

В этой главе мы покажем, что для некоторых операций над целыми числами и полиномами, таких, как деление и возведение в квадрат, требуется время того же порядка, что и для умножения. Время выполнения других операций, таких, как упомянутые выше операции с вычетами или вычисление наибольших общих делителей, может превосходить время умножения не более чем в $\log n$ раз, где n — длина двоичного представления целого числа или степень полинома. Наша стратегия будет состоять в том, чтобы чередовать результаты для целых чисел с соответствующими результатами для полиномов, причем обычно мы будем доказывать эти результаты только для чего-то одного, а доказательство для другого будем оставлять в качестве упражнения. Как и в остальных главах, основное внимание будет уделяться алгоритмам, асимптотически наиболее эффективным среди известных.

В конце главы кратко обсудим различие между моделью полиномов, в которой предполагается, что большинство коэффициентов отличны от нуля (плотная модель), и моделью, где большинство коэффициентов равны нулю (разреженная модель). Разреженная модель особенно полезна в случае полиномов от многих переменных (этот случай мы не рассматриваем).

8.1. АНАЛОГИИ МЕЖДУ ЦЕЛЫМИ ЧИСЛАМИ И ПОЛИНОМАМИ

Наиболее очевидная аналогия между неотрицательными целыми числами и полиномами от одной переменной заключается в возможности представлять те и другие конечными степенными рядами $\sum_{i=0}^{n-1} a_i x^i$. В случае целых чисел коэффициенты a_i можно выбирать из множества $\{0, 1\}$ при $x=2$. В случае полиномов эти a_i можно выбирать из некоторого множества коэффицентов [1]), считая x переменной.

Существует естественная мера "размера" — это по существу длина степенного ряда, представляющего целое число или полином. В случае двоичного целого числа размером служит число битов, необходимых для его представления; в случае полинома размер — это число его коэффициентов. Таким образом, мы приходим к следующему определению.

Определение. Если i — неотрицательное целое число, то РАЗМЕР$(i)=\lfloor \log i \rfloor+1$. Если $p(x)$ — полином, то РАЗМЕР$(p)=$ $=$СТ$(p)+1$, где СТ(p) — степень полинома p, т. е. наибольшая степень переменной x с ненулевым коэффициентом.

Над целыми числами и полиномами можно выполнять приближенное деление. Если a и b — два целых числа и $b\neq 0$, то найдется единственная пара целых чисел q и r, для которых

1) $a=bq+r$,
2) $r<b$,

где q и r — соответственно частное и остаток от деления a на b.

Аналогично, если a и b — полиномы, причем b отличен от постоянной, то можно найти такие полиномы q и r что

1) $a=bq+r$,
2) СТ$(r)<$СТ(b).

Другая аналогия между целыми числами и полиномами — существование для тех и других удивительно быстрых алгоритмов умножения. В предыдущей главе мы показали, что два полинома степени n с вещественными коэффициентами можно перемножить с помощью БПФ за время $O_A(n \log n)$. Здесь разумно измерять сложность числом арифметических операций, поскольку на практике мы представили бы полиномы их коэффициентами с фиксированной точностью и оперировали бы с полиномами, выполняя арифметические операции над такими коэффициентами.

[1]) Всюду ниже можно считать множество коэффициентов полем (см. разд. 12.1) вещественных чисел, хотя результаты верны для любого поля коэффициентов, если измерять вычислительную сложность числом операций в этом поле. Таким образом, в нашей модели размер коэффициентов не принимается во внимание.

Алгоритмом Шёнхаге — Штрассена из разд. 7.5 можно умножить два n-разрядных двоичных целых числа за время $O_Б(n \log n \log \log n)$. Мы утверждаем, что в случае целых чисел интересны лишь битовые операции. Фактически только в двух ситуациях не стóит рассматривать умножение целых чисел как основную (первичную) операцию. Это прежде всего разработка аппаратной реализации умножения, где число битовых операций соответствует числу элементов, необходимых для схемы умножения, и кроме того, разработка алгоритмов любой точности для операций с фиксированной запятой, реализуемых на вычислительных машинах со словами фиксированной длины, где число битовых операций соответствует числу машинных команд, необходимых для умножения с точностью до n разрядов.

Таким образом, результаты арифметических операций над полиномами и целыми числами окажутся очень похожими, если пользоваться двумя различными мерами сложности (арифметической и битовой). Эти две меры аналогичны в том смысле, что битовые операции — это операции над коэффициентами степенных рядов, представляющих целые числа, а арифметические — это операции над коэффициентами полиномов.

8.2. УМНОЖЕНИЕ И ДЕЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ

Покажем, что время (как число битовых операций) умножения целых чисел равно с точностью до постоянного множителя времени деления и оно так же связано с операциями возведения в квадрат и обращения. В данном разделе мы будем употреблять такие обозначения:

$M(n)$ — время умножения двух целых чисел размера n,
$D(n)$ — время деления целого числа размера не более $2n$ на целое число размера n,
$S(n)$ — время возведения в квадрат целого числа размера n,
$R(n)$ — время обращения целого числа размера n.

Здесь время измеряется числом битовых операций. Мы будем предполагать (что вполне разумно), что $M(n)$ удовлетворяет неравенствам

$$a^2 M(n) \geqslant M(an) \geqslant aM(n)$$

для $a \geqslant 1$. Предполагаем, что остальные три функции также обладают этим свойством.

Сначала покажем, что n-разрядное двоичное целое число можно обратить по существу за то же время, что и умножить два n-разрядных двоичных числа. Так как $1/i$ не является целым числом для $i > 1$, то под "обратным" к числу i мы на самом деле понимаем приближение к дроби $1/i$, имеющее n значащих двоичных разрядов (битов).

Поскольку предполагается, что на изменение масштаба (сдвиг двоичной запятой) время не тратится, мы можем с тем же успехом сказать, что "обратное" к числу i — это частное от деления 2^{2n-1} на i. В остальной части этого раздела термин "обратное" употребляется именно в этом смысле.

Сначала рассмотрим, как найти последовательные приближения к числу $A=1/P$, где P — целое. Пусть A_i будет i-м приближением к $1/P$. Тогда точное значение A можно представить в виде

$$A = A_i + (1/P)(1 - A_iP). \tag{8.1}$$

Если в (8.1) взять в качестве приближения к $1/P$ число A_i, то получим формулу

$$A_{i+1} = A_i + A_i(1 - A_iP) = 2A_i - A_i^2P, \tag{8.2}$$

которую можно использовать для нахождения $(i+1)$-го приближения числа A по его i-му приближению. Заметим, что если $A_iP=1-S$, то

$$A_{i+1}P = 2A_iP - A_i^2P^2 = 2(1-S) - (1-S)^2 = 1 - S^2.$$

Это показывает, что итерация по формуле (8.2) дает квадратичную скорость сходимости. Если $S \leqslant 1/2$, то число правильных двоичных разрядов удваивается при каждой итерации.

Поскольку число P предположительно содержит много двоичных разрядов, то для первых приближений не обязательно брать все его цифры, потому что на правильные цифры числа A_{i+1} влияют только старшие разряды в P. Если в A_i первые k цифр справа от запятой верны, то $2k$ цифр числа A_{i+1} можно найти по формуле (8.2). Иными словами, для вычисления $A_{i+1}=2A_i-A_i^2P$ берем в $2A_i$ только k цифр справа от запятой и находим A_i^2P, используя $2k$-разрядное приближение числа P и отбрасывая затем все разряды правее $2k$-го после запятой. Подобное использование приближенных значений может, конечно, повлиять на сходимость, так как ранее предполагалось, что число P точно.

Применим эти соображения в алгоритме, вычисляющем частное $\lfloor 2^{2n-1}/P \rfloor$, где $P=p_1\,p_2\ldots p_n$ — это n-разрядное двоичное целое с $p_1=1$. Метод по существу определяется формулой (8.2), к которой добавлено изменение масштаба (перенос запятой), чтобы можно было работать только с целыми числами. Таким образом, не надо заботиться о положении запятой.

Алгоритм 8.1. Обращение целых чисел

Вход. n-разрядное двоичное целое число $P=[p_1\,p_2\ldots p_n]$ с $p_1=1$. Для удобства предполагаем, что n — степень числа 2, а $[x]$ — целое число с двоичной записью x (например, $[110]=6$).

Выход. Целое число $A=[a_0\,a_1\ldots a_n]$, равное $A=\lfloor 2^{2n-1}/P \rfloor$.

Метод. Вызываем ОБРАТНОЕ ($[p_1p_2\ldots p_n]$), где ОБРАТНОЕ — рекурсивная процедура, приведенная на рис. 8.1. Она вычисляет

```
   procedure ОБРАТНОЕ ([p_1 p_2 ... p_k]):
1. if k = 1 then return [10]
   else
     begin
2.     [c_0 c_1 ... c_{k/2}] ← ОБРАТНОЕ ([p_1 p_2 ... p_{k/2}]);
3.     [d_1 d_2 ... d_{2k}] ← [c_0 c_1 ... c_{k/2}] * 2^{3k/2} − [c_0 c_1 ... c_{k/2}]^2 * [p_1 p_2 ... p_k];
       comment Хотя правая часть оператора в строке 3 служит для получения (2k+1)-разрядного числа, старший (2k+1)-й разряд всегда равен нулю;
4.     [a_0 a_1 ... a_k] ← [d_1 d_2 ... d_{k+1}];
       comment [a_0 a_1 ... a_k] — хорошее приближение к 2^{2k-1}/[p_1 p_2 ... p_k]. Следующий цикл улучшает это приближение, прибавляя, если необходимо, трехзначное число к младшим разрядам;
5.     for i ← 2 step −1 until 0 do
6.       if ([a_0 a_1 ... a_k] + 2^i) * [p_1 p_2 ... p_k] ⩽ 2^{2k-1} then
7.         [a_0 a_1 ... a_k] ← [a_0 a_1 ... a_k] + 2^i;
8.     return [a_0 a_1 ... a_k]
     end
```

Рис. 8.1. Процедура для обращения целых чисел.

приближение к $\lfloor 2^{2k-1}/[p_1 p_2 \ldots p_k] \rfloor$ для любого k, являющегося степенью числа 2. Заметим, что в результате обычно получается k-разрядное число. Исключение составляет случай, когда P — степень числа 2; здесь в результате получится $(k+1)$-разрядное число.

По данным k битам числа, обратного к $[p_1 p_2 \ldots p_k]$, строки 2—4 вычисляют $2k$—3 битов числа, обратного к $[p_1 p_2 \ldots p_{2k}]$. Строки 5—7 корректируют последние три бита. На практике можно было бы перескочить через строки 5—7 и получить нужную точность дополнительным применением формулы (8.2) в конце. Мы предпочли включить в программу цикл в строках 5—7, чтобы упростить понимание алгоритма и доказательство правильности его работы. □

Пример 8.1. Вычислим $2^{15}/153$. Здесь $n=8$ и $153=[p_1 p_2 \ldots p_8]=[10011001]$. Вызываем ОБРАТНОЕ ([10011001]), которое по очереди рекурсивно вызывает ОБРАТНОЕ с аргументами [1001], [10] и [1]. В строке 1 находим ОБРАТНОЕ([1])=[10] и возвращаемся к ОБРАТНОЕ ([10]) в строке 2, где полагаем $[c_0 c_1] \leftarrow [10]$. Затем в строке 3 вычисляем $[d_1 \ldots d_4] \leftarrow [10]*2^3 - [10]^2*[10]=[1000]$.

Далее в строке 4 полагаем $[a_0a_1a_2]=[100]$. Цикл в строках 5—7 ничего не меняет. Возвращаемся к вызову ОБРАТНОЕ ([1001]) с [100] в качестве приближения к $2^3/[10]$.

В строке 2 при $k=4$ имеем $[c_0c_1c_2]=[100]$. Тогда $[d_1\ldots d_8]=[01110000]$ и $[a_0\ldots a_4]=[01110]$. Снова цикл в строках 5—7 не приводит к изменениям. Возвращаемся к ОБРАТНОЕ ([10011001]) в строке 2, причем $[c_0\ldots c_4]=[01110]$. Далее

$$[d_1\ldots d_{16}]=[0110101011011100].$$

Следовательно, в строке 4 $[a_0\ldots a_8]=[011010101]$. В строке 6 находим, что $[011010101]*[10011001]$ (т. е. 213*153 в десятичной записи) равно 32589, тогда как $2^{15}=32768$. Поэтому цикл в строках 5—7 добавляет 1 к 213, что дает 214, или [011010110] в двоичной записи. □

Теорема 8.1. *Алгоритм* 8.1 *находит такое число* $[a_0\,a_1\ldots a_k]$, *что*

$$[a_0a_1\ldots a_k]*[p_1p_2\ldots p_k]=2^{2k-1}-S$$

и $0\leqslant S<[p_1\,p_2\ldots p_k]$.

Доказательство. Доказательство проводится индукцией по k. Базис, т. е. случай $k=1$, тривиален в силу строки 1. Для проведения шага индукции положим $C=[c_0c_1\ldots c_{k/2}]$, $P_1=[p_1p_2\ldots p_{k/2}]$ и $P_2=[p_{k/2+1}\,p_{k/2+2}\ldots p_k]$. Тогда $P=[p_1p_2\ldots p_k]=P_12^{k/2}+P_2$. По предположению индукции

$$CP_1=2^{k-1}-S,$$

где $0\leqslant S<P_1$. В силу строки 3 $D=[d_1\,d_2\ldots d_{2k}]$ определяется равенством

$$D=C2^{3k/2}-C^2(P_12^{k/2}+P_2). \tag{8.3}$$

Так как $p_1=1$, то $P_1\geqslant 2^{k/2-1}$ и, значит, $C\leqslant 2^{k/2}$. Отсюда следует, что $D<2^{2k}$, и потому для представления D достаточно $2k$ битов.

Рассмотрим произведение $PD=(P_12^{k/2}+P_2)\,D$, равное в силу (8.3)

$$PD=CP_12^{2k}+CP_22^{3k/2}-(CP_12^{k/2}+CP_2)^2. \tag{8.4}$$

Подставив $2^{k-1}-S$ вместо CP_1 в (8.4) и сделав некоторые алгебраические упрощения, получаем

$$PD=2^{3k-2}-(S2^{k/2}-CP_2)^2. \tag{8.5}$$

Разделив (8.5) на 2^{k-1} видим, что

$$2^{2k-1}=\frac{PD}{2^{k-1}}+T, \tag{8.6}$$

где $T=(S2^{k/2}-CP_2)^22^{-(k-1)}$. По предположению индукции и в силу неравенства $P_1<2^{k/2}$ имеем $S<2^{k/2}$. Так как $C\leqslant 2^{k/2}$ и $P_2<2^{k/2}$, то $0\leqslant T<2^{k+1}$.

В строке 4 $A=[a_0a_1\ldots a_k]=\lfloor D/2^{k-1}\rfloor$. Далее,

$$P\left\lfloor\frac{D}{2^{k-1}}\right\rfloor > P\left(\frac{D}{2^{k-1}}-1\right),$$

так что из (8.6) вытекает

$$2^{2k-1}\geqslant\frac{PD}{2^{k-1}}\geqslant P\left\lfloor\frac{D}{2^{k-1}}\right\rfloor>\frac{PD}{2^{k-1}}-P=2^{2k-1}-T-P>$$
$$>2^{2k-1}-2^{k+1}-2^k.$$

Поэтому

$$P\left\lfloor\frac{D}{2^{k-1}}\right\rfloor=2^{2k-1}-S',$$

где $0\leqslant S'<2^{k+1}+2^k$. Так как $P\geqslant 2^{k-1}$, то прибавление к $\lfloor D/2^{k-1}\rfloor$ числа, не превосходящего 6, дает число, удовлетворяющее предположению индукции для k. Поскольку именно это и делается в строках 5—7, то шаг индукции выполнен. □

Теорема 8.2. *Существует такая постоянная c, что $R(n)\leqslant cM(n)$.*

Доказательство. Достаточно показать, что алгоритм 8.1 тратит $O_Б(M(n))$ времени. На строку 2 уходит $R(k/2)$ битовых операций. Строка 3 состоит из возведения в квадрат и умножения, что требует соответственно $M(k/2+1)$ и $M(k+2)$ времени, а также вычитания, расходующего $O_Б(k)$ времени. Заметим, что умножение на степени числа 2 не тратит битовых операций; мы считаем, что разряды сомножителей просто занимают новые положения, т. е. они будто бы сдвигаются. В силу нашего предположения о M справедливо неравенство $M(k/2+1)\leqslant{}^1/_2M(k+2)$. Кроме того, $M(k+2)-M(k)=O_Б(k)$ (см., например, разд. 2.6) и, значит, сложность строки 3 ограничена величиной ${}^3/_2M(k)+c'k$ для некоторой постоянной c'. Сложность строки 4 равна, очевидно, $O_Б(k)$.

Может показаться, что цикл в строках 5—7 требует три умножения, но можно проделать необходимые вычисления за одно умножение $[a_0a_1\ldots a_k]*[p_1p_2\ldots p_k]$ и несколько сложений и вычитаний не более чем $2k$-разрядных двоичных целых чисел. Поэтому сложность строк 5—7 ограничена величиной $M(k)+c''k$ для некоторой постоянной c''. Объединяя все эти сложности, заключаем, что

$$R(k)\leqslant R\left(\frac{k}{2}\right)+\frac{5}{2}M(k)+c_1k \qquad (8.7)$$

для некоторой постоянной c_1.

Мы утверждаем, что найдется такая постоянная c, что $R(k)\leqslant cM(k)$. Можно выбрать c так, чтобы было $c\geqslant R(1)/M(1)$ и $c\geqslant 5+2c_1$. Докажем наше утверждение индукцией по k. Базис, т. е. случай $k=1$, очевиден. Шаг индукции получается в силу (8.7), по-

скольку

$$R(k) \leqslant cM\left(\frac{k}{2}\right) + \frac{5}{2}M(k) + c_1 k. \tag{8.8}$$

Так как $M(k/2) \leqslant {}^1/_2 M(k)$ в силу нашего предположения о M, а оценка $k \leqslant M(k)$ очевидна, можно переписать (8.8) в виде

$$R(k) \leqslant \left(\frac{c}{2} + \frac{5}{2} + c_1\right) M(k). \tag{8.9}$$

Поскольку $c \geqslant 5 + 2c_1$, то из (8.9) следует, что $R(k) \leqslant cM(k)$. □

Ясно, что алгоритмом 8.2 можно вычислить $1/P$ с n значащими двоичными цифрами, если P имеет столько же цифр, при этом неважно, где расположена запятая. Например, если ${}^1/_2 < P < 1$ и P имеет n битов, можно очевидным образом изменить масштаб и представить $1/P$ как 1 и последующие $n-1$ битов дробной части.

Теперь покажем, что время $S(n)$, необходимое для возведения в квадрат целого числа размера n, не превышает по порядку времени $R(n)$ обращения целого числа размера n. Метод основан на тождестве

$$P^2 = \frac{1}{\frac{1}{P} - \frac{1}{P+1}} - P. \tag{8.10}$$

Приведем алгоритм, использующий (8.10) с подходящим изменением масштаба.

Алгоритм 8.2. Возведение в квадрат с помощью обратных величин

Вход. n-разрядное целое число P в двоичной записи.

Выход. Двоичная запись числа P^2.

Метод.

1. Применяем алгоритм 8.1 для вычисления $A = \lfloor 2^{4n-1}/P \rfloor$. Для этого добавляем $2n$ нулей к P, применяем процедуру ОБРАТНОЕ [1]) для вычисления $\lfloor 2^{6n-1}/P 2^{2n} \rfloor$ и сдвигаем результат.

2. Аналогично вычисляем $B = \lfloor 2^{4n-1}/(P+1) \rfloor$.

3. Полагаем $C = A - B$. Заметим, что $C = 2^{4n-1}/(P^2+P) + T$, где $|T| \leqslant 1$. Слагаемое T возникает из-за того, что отбрасывание знаков при вычислении A и B может привести к ошибке вплоть до 1. Так как $2^{2n-2} < P^2 + P < 2^{2n}$, то $2^{2n+1} \geqslant C \geqslant 2^{2n-1}$.

4. Вычисляем $D = \lfloor 2^{4n-1}/C \rfloor - P$.

5. Пусть Q — четыре последние бита числа P. Увеличиваем или уменьшаем D на минимально возможную величину так, чтобы последние четыре бита результата совпали с последними четырьмя битами в Q^2. □

[1]) Процедура ОБРАТНОЕ определена в алгоритме 8.1 только для целых чисел, длина которых равна степени числа 2. Обобщение на любые целые числа должно быть очевидным: добавляем нули и, если надо, изменяем масштаб.

Теорема 8.3. *Алгоритм* 8.2 *вычисляет* P^2.

Доказательство. В силу способа, которым отбрасываются цифры на шагах 1 и 2, можно ручаться лишь за то, что

$$C = \frac{2^{4n-1}}{P^2+P} + T, \qquad \text{где } |T| \leqslant 1.$$

Так как $2^{2n-1} \leqslant C \leqslant 2^{2n+1}$ и ошибка в C заключена между -1 и 1, то связанная с ней ошибка в $2^{4n-1}/C$ не превосходит

$$\left|\frac{2^{4n-1}}{C} - \frac{2^{4n-1}}{C-1}\right| = \left|\frac{2^{4n-1}}{C^2-C}\right|.$$

Так как $C^2-C \geqslant 2^{4n-3}$, то эта ошибка не превосходит 4. Отбрасывание цифр в строке 4 может увеличить ошибку до 5. Таким образом, $|P^2-D| \leqslant 5$. Следовательно, вычисление последних четырех цифр в P^2 на шаге 5 гарантирует, что D корректируется так, что становится равным P^2. □

Пример 8.2. Путь $n=4$ и $P=[1101]$. Тогда

$$A = \lfloor 2^{15}/[1101] \rfloor = [100111011000]$$

и

$$B = \lfloor 2^{15}/[1110] \rfloor = [100100100100].$$

Далее,

$$C = A - B = [10110100].$$

Тогда

$$D = [2^{15}/C] - P = [10110110] - [1101] = [10101001].$$

Таким образом, $D=169$ — квадрат числа 13, и на шаге 5 не нужна никакая коррекция. □

Теорема 8.4. *Найдется такая постоянная* c, *что* $S(n) \leqslant cR(n)$.

Доказательство. В алгоритме 8.2 трижды вычисляются обратные к числам, задаваемым цепочками длины не более $3n$. Вычитания на шагах 3 и 4 требуют $O_Б(n)$ времени, а на шаге 5 выполняется работа фиксированного объема, не зависящего от n. Следовательно,

$$S(n) \leqslant 3R(3n) + c_1 n \tag{8.11}$$

для некоторой постоянной c_1.

Таким образом, $S(n) \leqslant 27R(n) + c_1 n$. Так как $R(n) \geqslant n$, то, полагая $c = 27 + c_1$, получаем нужное неравенство. □

Теорема 8.5. $M(n)$, $R(n)$, $D(n)$ *и* $S(n)$ *равны с точностью до постоянных множителей.*

Доказательство. Мы уже показали, что $R(n) \leqslant c_1 M(n)$ и $S(n) \leqslant c_2 R(n)$ для некоторых постоянных c_1 и c_2. Легко вывести

неравенство $M(n) \leqslant c_3 S(n)$, заметив, что

$$AB = \frac{1}{2}[(A+B)^2 - A^2 - B^2].$$

Таким образом, M, R и S равны с точностью до постоянных множителей.

Когда мы обсуждаем деление n-разрядных двоичных чисел, мы фактически подразумеваем деление числа, содержащего до $2n$ битов, на число, содержащее в точности n битов, причем ответ содержит не более $n+1$ битов. Очевидно, что $R(n) \leqslant D(n)$, так что $M(n) \leqslant c_2 c_3 D(n)$. Более того, с помощью тождества $A/B = A*(1/B)$ можно показать, изменяя подходящим образом масштаб, что для некоторой постоянной c

$$D(n) \leqslant M(2n) + R(2n) + cn. \tag{8.12}$$

Поскольку $R(2n) \leqslant c_1 M(2n)$ и, как легко показать, $M(2n) \leqslant 4M(n)$, можно переписать (8.12) в виде

$$D(n) \leqslant 4(1+c_1) M(n) + cn. \tag{8.13}$$

Так как $M(n) \geqslant n$, то в силу (8.13) справедливо неравенство $D(n) \leqslant c_4 M(n)$, где $c_4 = 4 + 4c_1 + c$. Итак, мы доказали, что все рассматриваемые функции заключены между $dM(n)$ и $eM(n)$ для некоторых положительных постоянных d и e. □

Следствие. *Деление $2n$-разрядного двоичного целого числа на n-разрядное можно выполнить за время $O_Б(n \log n \log \log n)$.*

Доказательство. В силу теорем 7.8 и 8.5. □

8.3. УМНОЖЕНИЕ И ДЕЛЕНИЕ ПОЛИНОМОВ

Вся техника предыдущего раздела переносится на арифметические операции над полиномами от одной переменной. В этом разделе функции $M(n)$, $D(n)$, $R(n)$ и $S(n)$ обозначают времена, требуемые соответственно для умножения, деления, обращения и возведения в квадрат полиномов степени n. Как и раньше, мы предполагаем, что $a^2 M(n) \geqslant M(an) \geqslant aM(n)$ для $a \geqslant 1$ и подобные неравенства справедливы и для других функций.

Под "обратным" к полиному $p(x)$ степени n мы понимаем полином $\lfloor x^{2n}/p(x) \rfloor$[1]). $D(n)$ — это время нахождения полинома $\lfloor s(x)/p(x) \rfloor$, где $p(x)$ имеет степень n, а $s(x)$ — степень, не превосходящую $2n$. Заметим, что подобно тому, как в предыдущем разделе мы изменяли

[1]) По аналогии с обозначением для целых чисел для обозначения частного от деления полиномов употребляется «функция дна». Иными словами, если $p(x)$ не постоянная, то $\lfloor s(x)/p(x) \rfloor$ — это единственный полином $q(x)$, для которого $s(x) = p(x)q(x) + r(x)$ и $\mathrm{CT}(r(x)) < \mathrm{CT}(p(x))$.

масштаб целых чисел с помощью степеней числа 2, можно "изменять масштаб" полиномов, умножая и деля их на степени переменной x.

Так как результаты настоящего раздела очень похожи на результаты для целых чисел, изложим в деталях только один из них: алгоритм обращения полиномов, аналогичный алгоритму 8.1 для целых чисел. Алгоритмы для полиномов в какой-то мере проще соответствующих алгоритмов для целых чисел —в основном благодаря тому, что в степенных рядах в отличие от целых чисел нет переносов. Поэтому в алгоритмах для полиномов не надо корректировать младшие значащие разряды, как это требовалось, например, в строках 5—7 алгоритма 8.1.

Алгоритм 8.3. Обращение полиномов

Вход. Полином $p(x)$ степени n—1, где n — степень числа 2 (т. е. $p(x)$ имеет 2^t членов, где t — некоторое целое число).

Выход. Обратный полином $\lfloor x^{2n-2}/p(x) \rfloor$.

Метод. На рис. 8.2 приведена новая процедура

$$\text{ОБРАТНЫЙ}\left(\sum_{i=0}^{k-1} a_i x^i\right),$$

где k — степень числа 2 и $a_{k-1} \neq 0$. Эта процедура вычисляет

$$\left\lfloor \frac{x^{2k-2}}{\sum_{i=0}^{k-1} a_i x^i} \right\rfloor.$$

Заметим, что при $k=1$ аргументом будет постоянная a_0, а ее обратным — другая постоянная $1/a_0$. Предполагается что каждую операцию над коэффициентами можно выполнить за один шаг, и поэто-

---

procedure ОБРАТНЫЙ $\left(\sum_{i=0}^{k-1} a_i x^i\right)$:

1. **if** $k=1$ **then return** $1/a_0$
   else
   begin
2. $q(x) \leftarrow$ ОБРАТНЫЙ $\left(\sum_{i=k/2}^{k-1} a_i x^{i-k/2}\right)$;
3. $r(x) \leftarrow 2q(x)\, x^{(3/2)k-2} - (q(x))^2 \left(\sum_{i=0}^{k-1} a_i x^i\right)$;
   return $\lfloor r(x)/x^{k-2} \rfloor$
   end

Рис. 8.2. Процедура для обращения полиномов.

му для вычисления $1/a_0$ нет необходимости вызывать процедуру ОБРАТНОЕ.

Сам алгоритм состоит в вызове процедуры ОБРАТНЫЙ с аргументом $p(x)$. □

Пример 8.3. Вычислим $\lfloor x^{14}/p(x) \rfloor$, где $p(x)=x^7-x^6+x^5+2x^4-x^3-3x^2+x+4$. В строке 2 обращаем полином x^3-x^2+x+2, т. е. находим $q(x)=\lfloor x^6/(x^3-x^2+x+2) \rfloor$. Проверьте, что $q(x)=x^3+x^2-3$. Поскольку $k=8$, строка 3 вычисляет $r(x)=2q(x)\,x^{10}-(q(x))^2\,p(x)=x^{13}+x^{12}-3x^{10}-4x^9+3x^8+15x^7+12x^6-42x^5-34x^4+39x^3+51x^2-9x-36$. Затем в строке 4 получаем результат $s(x)=x^7+x^6-3x^4-4x^3+3x^2+15x+12$. Заметим, что $s(x)\,p(x)=x^{14}+$полином степени 6. □

Теорема 8.6. *Алгоритм* 8.6 *правильно вычисляет полином, обратный к данному.*

Доказательство. Докажем индукцией по k, где k — степень числа 2, что если $s(x)=$ОБРАТНЫЙ$(p(x))$ и $p(x)$ имеет степень $k-1$, то $s(x)\,p(x)=x^{2k-2}+t(x)$, где $t(x)$ — полином степени, меньшей $k-1$. Базис, т. е. случай $k=1$, тривиален, ибо $p(x)=a_0$, $s(x)=1/a_0$ и слагаемого $t(x)$ нет.

Для шага индукции положим $p(x)=p_1(x)x^{k/2}+p_2(x)$, где $\mathrm{CT}(p_1)=k/2-1$ и $\mathrm{CT}(p_2)\leqslant k/2-1$. Тогда по предположению индукции, если $s_1(x)=$ОБРАТНЫЙ$(p_1(x))$, то $s_1(x)p_1(x)=x^{k-2}+t_1(x)$, где $\mathrm{CT}(t_1)<k/2-1$. В строке 3 вычисляем

$$r(x)=2s_1(x)\,x^{(3/2)k-2}-(s_1(x))^2\,(p_1(x)\,x^{k/2}+p_2(x)). \qquad (8.14)$$

Достаточно показать, что $r(x)\,p(x)=x^{3k-4}+$полином степени, меньшей $2k-3$. Тогда деление на x^{k-2} в строке 4 дает искомый результат.

В силу (8.14) и равенства $p(x)=p_1(x)\,x^{k/2}+p_2(x)$ имеем

$$r(x)\,p(x)=2s_1(x)\,p_1(x)\,x^{2k-2}+2s_1(x)\,p_2(x)\,x^{(3/2)k-2}-(s_1(x)\,p_1(x)\,x^{k/2}+s_1(x)\,p_2(x))^2. \qquad (8.15)$$

Подставив $x^{k-2}+t_1(x)$ вместо $s_1(x)\,p_1(x)$ в (8.15), получим

$$r(x)\,p(x)=x^{3k-4}-(t_1(x)\,x^{k/2}+s_1(x)\,p_2(x))^2. \qquad (8.16)$$

Так как $\mathrm{CT}(t_1)<k/2-1$ и степени полиномов $s_1(x)$ и $p_2(x)$ не больше $k/2-1$, то степени всех членов в (8.16), отличных от x^{3k-4}, не превосходят $2k-4$. □

Ясно, что время работы алгоритмов 8.3 и 8.1 оценивается схожим образом, если рассматривать две меры сложности (соответственно арифметическую и битовую). Аналогично можно показать, что и другие оценки времени, установленные в разд. 8.2, переносятся

на полиномы, если вместо битовых шагов рассматривать арифметические. Таким образом, верна следующая теорема.

Теорема 8.7. *Пусть $M(n)$, $D(n)$, $R(n)$ и $S(n)$ — арифметические сложности соответственно умножения, деления, обращения и возведения в квадрат полиномов от одной переменной. Все эти функции равны с точностью до постоянных множителей.*

Доказательство. Аналогично доказательству теоремы 8.5 и результатов, на которые оно опирается. □

Следствие. *Полином степени $2n$ можно разделить на полином степени n за время $O_A(n \log n)$.*

Доказательство. В силу теоремы 8.7 и следствия 3 теоремы 7.4. □

8.4. МОДУЛЬНАЯ АРИФМЕТИКА

В некоторых приложениях удобно выполнять арифметические операции над целыми числами в "модульном" представлении (т. е. в системе классов вычетов). Это значит, что вместо того, чтобы представлять целое число в системе счисления с фиксированным основанием, его представляют вычетами по модулям из множества попарно взаимно простых чисел. Если $p_0, p_1, \ldots, p_{k-1}$ — попарно взаимно простые числа и $p = \prod_{i=0}^{k-1} p_i$, то любое целое число u, $0 \leqslant u < p$, можно однозначно представить множеством его вычетов $u_0, u_1, \ldots, u_{k-1}$, где $u_i = u$ по модулю p_i, $0 \leqslant i < k$. Когда $p_0, p_1, \ldots, p_{k-1}$ фиксированы, пишут $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Сложение, вычитание и умножение легко выполняются, если их результаты заключены между 0 и $p-1$ (другими словами, если эти вычисления можно рассматривать как вычисления по модулю p). Пусть

$$u \leftrightarrow (u_0, u_1, \ldots, u_{k-1}), \qquad v \leftrightarrow (v_0, v_1, \ldots, v_{k-1}).$$

Тогда

$$u + v \leftrightarrow (w_0, w_1, \ldots, w_{k-1}), \quad \text{где} \quad w_i = (u_i + v_i) \bmod p_i, \tag{8.17}$$

$$u - v \leftrightarrow (x_0, x_1, \ldots, x_{k-1}), \quad \text{где} \quad x_i = (u_i - v_i) \bmod p_i, \tag{8.18}$$

$$uv \leftrightarrow (y_0, y_1, \ldots, y_{k-1}), \quad \text{где} \quad y_i = u_i v_i \bmod p_i. \tag{8.19}$$

Пример 8.4. Пусть $p_0=5$, $p_1=3$ и $p_2=2$. Тогда $4 \leftrightarrow (4, 1, 0)$, поскольку $4 = 4 \bmod 5$, $1 = 4 \bmod 3$ и $0 = 4 \bmod 2$. Аналогично $7 \leftrightarrow (2, 1, 1)$ и $28 \leftrightarrow (3, 1, 0)$. Заметим, что в силу (8.19) $4 \times 7 \leftrightarrow (3, 1, 0)$, а это — представление числа 28. Первая компонента произведения 4×7 равна $4 \times 2 \bmod 5$, т. е. 3; вторая компонента равна $1 \times 1 \bmod 3$, т. е. 1; третья равна $0 \times 1 \bmod 2$, т. е. 0. Кроме того, $4 * 7 \leftrightarrow (1, 2, 1)$

(представление числа 11), $7—4 \leftrightarrow (3, 0, 1)$ (представление числа 3). □

Однако неясно, как в модульной арифметике экономно выполнять деление. Заметим, что отношение u/v может не быть целым числом, а если бы и было, то в общем случае нельзя найти его модульное представление, вычисляя u_i/v_i по модулю p_i для каждого i. Действительно, если p_i не является простым числом, то между 0 и $p_i—1$ может оказаться несколько целых чисел w, равных u_i/v_i по модулю p_i в том смысле, что $wv_i \equiv u_i \pmod{p_i}$. Например, если $p_i=6$, $v_i=3$ и $u_i=3$, то в качестве w можно было бы взять 1, 3 или 5, поскольку $1\times3\equiv3\times3\equiv5\times3\equiv3 \pmod 6$. Поэтому $(u_i/v_i) \bmod p_i$ может не иметь смысла.

Преимущество модульного представления в основном в том, что арифметические операции можно реализовать с меньшими аппаратными затратами, чем при обычном представлении, поскольку вычисления выполняются независимо для каждого модуля. В отличие от обычного (позиционного) представления чисел, здесь не нужны никакие переносы. К сожалению, проблемы эффективного деления и контроля переполнений (т. е. выхода результата за пределы области, заключенной между 0 и $p—1$) оказываются непреодолимыми, и поэтому такие системы редко реализуются в машинных блоках общего назначения.

Тем не менее содержащиеся здесь идеи находят применение, главным образом при рассмотрении полиномов, поскольку делить полиномы скорее всего не потребуется. Кроме того, как мы увидим в следующем разделе, вычисление полиномов и их вычетов (по модулю других полиномов) тесно связаны. Сейчас покажем, что модульная арифметика целых чисел "работает" так, как нужно.

Первая часть доказательства состоит в том, чтобы доказать, что соотношения (8.17) — (8.19) выполняются. Эти соотношения очевидны, и мы оставляем их в качестве упражнения. Вторая часть доказательства — показать, что соответствие $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ взаимно однозначно (т. е. является изоморфизмом). Хотя этот результат несложен, сформулируем его в виде леммы.

Лемма 8.1. *Пусть* $p_0, p_1, \ldots, p_{k-1}$ — *попарно взаимно простые целые числа,* $p=\prod_{i=0}^{k-1} p_i$ *и* $u_i = u \bmod p_i$. *Тогда соответствие* $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ *между целыми числами и в интервале* $[0, p)$ *и наборами вида*

$$(u_0, u_1, \ldots, u_{k-1}), \quad 0 \leqslant u_i < p_i \quad \text{при} \quad 0 \leqslant i < k,$$

взаимно однозначно.

Доказательство. Очевидно, что для каждого u найдется соответствующий k-членный кортеж. Так как в интервале

$[0, p)$ заключено ровно p значений переменной u и допустимых k-членных кортежей также ровно p, достаточно показать, что каждый такой кортеж соответствует не более одному целому числу u. Допустим, что два числа u и v, $0 \leqslant u < v < p$, соответствуют кортежу $(u_0, u_1, \ldots, u_{k-1})$. Тогда разность $v-u$ должна делиться на каждое число p_i. Поскольку все p_i попарно взаимно просты, разность $v-u$ должна делиться и на p. Но $u \neq v$ и $v-u$ делится на p, так что u и v должны разниться не менее чем на p и, значит, не могут оба лежать между 0 и $p-1$. □

Для того чтобы можно было пользоваться модульной арифметикой, нужны алгоритмы, осуществляющие переход от позиционного представления к модульному и обратно. Один из методов перехода от позиционного представления целого числа u к его модульному представлению состоит в том, чтобы разделить u на каждое из чисел p_i, $0 \leqslant i < k$.

Допустим, что каждое из чисел p_i содержит b разрядов в двоичном представлении. Тогда для представления

$$p = \prod_{i=0}^{k-1} p_i$$

требуется, грубо говоря, bk битов (двоичных разрядов), а деление u на каждое из чисел p_i, где $0 \leqslant u < p$, могло бы потребовать k делений kb-битового числа на b-битовое число. Разбив каждое деление на k делений $2b$-битовых чисел на b-битовые, можно перейти к модульному представлению за время $O_{\text{Б}}(k^2 D(b))$, где $D(n)$ — время деления целых чисел (не превосходящее $O_{\text{Б}}(n \log n \log\log n)$ в силу следствия теоремы 8.5).

Однако можно проделать эту работу за значительно меньшее время, если применить метод, напоминающий метод деления полиномов из разд. 7.2. Вместо того чтобы делить число u на каждый из k модулей $p_0, p_1, \ldots, p_{k-1}$, сначала вычисляем произведения $p_0 p_1, p_2 p_3, \ldots, p_{k-2} p_{k-1}$, затем $p_0 p_1 p_2 p_3$, $p_4 p_5 p_6 p_7, \ldots$ и т. д. Далее вычисляем вычеты с помощью приема "разделяй и властвуй". Деля, получаем вычеты u_1 и u_2 числа u по модулям $p_0 \ldots p_{k/2-1}$ и $p_{k/2} \ldots p_{k-1}$ соответственно. Теперь задача вычисления $u \bmod p_i$, $0 \leqslant i < k$, сведена к двум подзадачам половинного размера, а именно $u \bmod p_i = u_1 \bmod p_i$ для $0 \leqslant i < k/2$, и $u \bmod p_i = u_2 \bmod p_i$ для $k/2 \leqslant i < k$.

Алгоритм 8.4. Вычисление вычетов

Вход. Модули $p_0, p_1, \ldots, p_{k-1}$ и такое целое число u, что $0 \leqslant u < p = \prod_{i=0}^{k-1} p_i$.

Выход. Числа u_i, $0 \leqslant i < k$, такие, что $u_i = u \bmod p_i$.

Метод. Допустим, что k — степень числа 2, скажем $k = 2^t$.

(Если нужно, добавим ко входу лишние модули, равные 1, чтобы сделать k степенью числа 2.) Начинаем с вычисления определенных произведений модулей, аналогичных полиномам q_{lm} из разд. 7.2.

Пусть $0 \leqslant j < t$, число i кратно числу 2^j и $0 \leqslant i < k$; положим

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m.$$

Таким образом, $q_{i0}=p_i$ и $q_{ij}=q_{i,j-1}q_{i+2^{j-1},j-1}$.

Сначала вычисляем числа q_{ij}, затем находим остатки u_{ij} от деления u на каждое из чисел q_{ij}. Искомым ответом будут числа u_{i0}. Детали приведены в программе на рис. 8.3. □

Теорема 8.8. *Алгоритм* 8.4 *правильно вычисляет числа* u_i.

Доказательство. Доказательство следует плану доказательства теоремы 7.3, где вычислялись значения полинома в корнях n-й степени из единицы. Легко показать индукцией по j, что строка 4 правильно вычисляет числа q_{ij}. Затем возвратной индукцией по j в строке 6 доказываем, что $u_{ij}=u \bmod q_{ij}$. Строки 8 и 9 позволяют сделать это легко. Полагая $j=0$, получаем $u_i=u \bmod p_i$. Детали оставляем в качестве упражнения. □

Теорема 8.9. *Алгоритм* 8.4 *тратит* $O_Б(M(bk)\log k)$ *времени, если для представления каждого из чисел* p_i *достаточно* b *битов.*

Доказательство. Легко видеть, что больше всего времени требуют циклы в строках 3, 4 и 7—9. Каждый из них занимает

```
    begin
1.    for i ← 0 until k − 1 do q_{i0} ← p_i;
2.    for j ← 1 until t − 1 do
3.      for i ← 0 step 2^j until k − 1 do
4.        q_{ij} ← q_{i,j-1} * q_{i+2^{j-1},j-1};
5.      u_{0t} ← u;
6.      for j ← t step −1 until 1 do
7.        for i ← 0 step 2^j until k − 1 do
            begin
8.            u_{i,j-1} ← ОСТАТОК (u_{ij}/q_{i,j-1});
9.            u_{i+2^{j-1},j-1} ← ОСТАТОК (u_{ij}/q_{i+2^{j-1},j-1})
            end;
10.     for i ← 0 until k − 1 do u_i ← u_{i0}
    end
```

Рис. 8.3. Вычисление вычетов.

$O_Б(2^{t-j}M(2^{j-1}b))$ шагов[1]). Поскольку мы предположили, что $M(an) \geqslant aM(n)$ для $a \geqslant 1$, то сложность выполнения этих циклов ограничена величиной $O_Б(M(2^t b)) = O_Б(M(kb))$. Так как каждый из циклов повторяется не более $t = \log k$ раз, получаем нужный результат. □

Следствие. *Если для представления каждого из модулей* p_0, p_1, . . ., p_{k-1} *требуется* b *битов, то вычеты по этим модулям можно вычислить за время* $O_Б(bk \log k \log bk \log\log bk)$.

8.5. МОДУЛЬНАЯ АРИФМЕТИКА ПОЛИНОМОВ И ВЫЧИСЛЕНИЕ ИХ ЗНАЧЕНИЙ

Результаты, аналогичные результатам для целых чисел, справедливы и для полиномов. Пусть $p_0, \ldots, p_{k-1}$ — полиномы и $p = \prod_{i=0}^{k-1} p_i$. Тогда каждый полином u можно представить последовательностью $u_0, u_1, \ldots, u_{k-1}$ остатков от деления u на каждый полином p_i. Полином u_i — это тот единственный полином, для которого $СТ(u_i) < СТ(p_i)$ и $u = p_i q_i + u_i$ для некоторого полинома q_i. В такой ситуации мы будем писать $u_i = u \bmod p_i$, что совершенно аналогично модульной записи целых чисел.

По аналогии с леммой 8.1 можно показать, что соответствие $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ взаимно однозначно, если полиномы p_i попарно взаимно просты и степень полинома u меньше степени полинома p, т. е. $РАЗМЕР(u) < РАЗМЕР(p)$. Более того, алгоритм 8.4 для вычисления вычетов применим к полиномам, если в качестве p_i взять полиномы, а не целые числа. Вместо b (числа битов в каждом p_i) надо рассматривать наибольшую степень полиномов p_i. Разумеется, сложность измеряется числом арифметических, а не битовых операций. Тогда справедлива теорема, аналогичная теореме 8.9.

Теорема 8.10. *Перенеся алгоритм* 8.4 *на полиномы, можно найти вычеты полинома* $u(x)$ *относительно полиномов* $p_0, p_1, \ldots, p_{k-1}$ *за время* $O_A(M(dk) \log k)$, *где* d *— верхняя граница степеней полиномов* p_i *и степень полинома* u *меньше степени полинома* $\prod_{i=0}^{k-1} p_i$.

Доказательство. Аналогично теореме 8.9; оставим его в качестве упражнения. □

Следствие 1. *Для нахождения вычетов полинома* u *относительно полиномов* $p_0, p_1, \ldots, p_{k-1}$ *требуется не более* $O_A(dk \log k \log dk)$

[1]) Напомним, что $D(n)$ и $M(n)$ по существу совпадают.

времени, где d — верхняя граница степеней полиномов p_i и степень полинома и меньше степени полинома $\prod_{i=0}^{k-1} p_i$.

Пример 8.5. Рассмотрим четыре полиномиальных модуля

$$p_0 = x-3,$$
$$p_1 = x^2+x+1,$$
$$p_2 = x^2-4,$$
$$p_3 = 2x+2$$

и положим $u=x^5+x^4+x^3+x^2+x+1$. Сначала вычисляем произведения

$$p_0p_1 = x^3-2x^2-2x-3,$$
$$p_2p_3 = 2x^3+2x^2-8x-8.$$

Затем вычисляем

$$u' = u \bmod p_0p_1 = 28x^2+28x+28,$$
$$u'' = u \bmod p_2p_3 = 21x+21.$$

В самом деле, $u = p_0p_1(x^2+3x+9)+28x^2+28x+28$ и $u = p_2p_3\left(\frac{1}{2}x^2+\frac{5}{2}\right)+21x+21$.

Далее,

$$u \bmod p_0 = u' \bmod p_0 = 364,$$
$$u \bmod p_1 = u' \bmod p_1 = 0,$$
$$u \bmod p_2 = u'' \bmod p_2 = 21x+21,$$
$$u \bmod p_3 = u'' \bmod p_3 = 0. \quad \square$$

Заметим, что алгоритм БПФ из разд. 7.2 в действительности является реализацией этого алгоритма, когда в качестве $p_0, p_1, \ldots, p_{k-1}$ берутся $x-\omega^0, x-\omega^1, \ldots, x-\omega^{n-1}$. Алгоритм БПФ извлекает выгоду из выбора в качестве p_i полинома $x-\omega^i$. Благодаря подходящему упорядочению множества полиномов p_i каждое произведение равно разности степени x и степени ω и потому деление выполняется особенно легко.

Как было отмечено в разд. 7.2, если p_i — полином первой степени $x-a$, то $u \bmod p_i = u(a)$. Поэтому случай, когда все полиномы p_i имеют степень 1, особенно важен. Из теоремы 8.10 вытекает следующий результат.

Следствие 2. *Значения полинома n-й степени в n точках можно вычислить за время $O_A(n \log^2 n)$.*

Доказательство. Чтобы вычислить $u(x)$ в n точках $a_0, a_1, \ldots, a_{n-1}$, вычисляем $u(x) \bmod (x-a_i)$ для $0 \leqslant i < n$. Это занимает время $O_A(n \log^2 n)$ в силу следствия 1, ибо здесь $d=1$. $\square$

8.6. ПРИМЕНЕНИЕ КИТАЙСКОЙ ТЕОРЕМЫ ОБ ОСТАТКАХ

Изучим задачу преобразования модульного представления целого числа в его позиционное представление [1]). Пусть даны попарно взаимно простые модули $p_0, p_1, \ldots, p_{k-1}$ и вычеты $u_0, u_1, \ldots, u_{k-1}$, где $k=2^t$; надо найти такое целое число u, что $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$. Это можно сделать с помощью целочисленного аналога *интерполяционной формулы Лагранжа* для полиномов.

Лемма 8.2. *Пусть* c_i — *произведение всех* p_j, *кроме* p_i (*т. е.* $c_i = p/p_i$, *где* $p=\prod_{j=0}^{k-1} p_j$). *Положим* $d_i = c_i^{-1} \bmod p_i$ (*т. е.* $d_i c_i \equiv 1 \pmod{p_i}$ *и* $0 \leqslant d_i < p_i$). *Тогда*

$$u \equiv \sum_{i=0}^{k-1} c_i d_i u_i \pmod{p}. \tag{8.20}$$

Доказательство. Поскольку числа p_j попарно взаимно просты, то, как известно (упр. 7.9), числа d_i существуют и единственны. Кроме того, c_i делится на p_j при $j \neq i$, так что $c_i d_i u_i \equiv 0 \pmod{p_j}$ при $j \neq i$. Поэтому

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv c_j d_j u_j \pmod{p_j}.$$

Так как $c_j d_j \equiv 1 \pmod{p_j}$, то

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv u_j \pmod{p_j}.$$

Так как p_i делит p, то эти соотношения выполняются даже тогда, когда все арифметические операции производятся по модулю p. Таким образом, (8.20) верно. □

Наша задача состоит в эффективном вычислении (8.20). Сразу трудно увидеть, как можно вычислять d_i по p_i, не прибегая к методу проб и ошибок. Позже мы увидим, что эта задача не так уж трудна, если использовать алгоритм Евклида из разд. 8.8 и его быструю реализацию из разд 8.10. В данном разделе мы изучим только ту форму китайской задачи об остатках, которая основана на "предварительной обработке". Если часть входных данных фиксирована для ряда задач, то все величины, зависящие только от этой фиксированной части, можно вычислить заранее и считать частью входа. Алгоритм, в котором сделаны такие предварительные вычисления, называют алгоритмом *с предварительной обработкой данных*

[1]) Процесс восстановления числа по его остаткам был известен китайцам более 2000 лет назад, и потому соответствующую теорему называют *китайской теоремой об остатках.*

Для китайского алгоритма с предварительной обработкой данных входом будут служить не только вычеты и модули p_i, но и обратные к ним (числа d_i). Эта ситуация не является нереальной. Если часто применять китайский алгоритм для перевода чисел, представленных вычетами по фиксированному множеству модулей, то разумно заранее вычислить те функции от этих модулей, значения которых используются в алгоритме. В следствии теоремы 8.21 мы увидим, что на оценку сложности наличие предварительной обработки (или ее отсутствие) влияет весьма скромно.

Анализируя (8.20), замечаем, что слагаемые $c_i d_i u_i$ при разных i имеют много общих сомножителей. Например, $c_i d_i u_i$ имеет сомножителем число $p_0 p_1 \ldots p_{k/2-1}$ если $i \geqslant k/2$, и число $p_{k/2}\, p_{k/2+1} \ldots p_{k-1}$, если $i < k/2$. Поэтому можно записать (8.20) в виде

$$u = \left(\sum_{i=0}^{k/2-1} c_i' d_i u_i\right) \times \prod_{i=k/2+1}^{k-1} p_i + \left(\prod_{i=k/2+1}^{k-1} c_i'' d_i u_i\right) \times \prod_{i=0}^{k/2-1} p_i,$$

где c_i' — это произведение $p_0 p_1 \ldots p_{k/2-1}$ без p_i, а c_i'' — произведение $p_{k/2} p_{k/2+1} \ldots p_{k-1}$ без p_i. Это наблюдение должно подсказать прием "разделяй и властвуй", аналогичный тому, что применялся при вычислении вычетов. Находим произведения

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$$

(как в алгоритме 8.4), а затем — целые числа

$$s_{ij} = \prod_{m=i}^{i+2^j+1} q_{ij} d_m u_m / p_m.$$

Если $j=0$, то $s_{i0} = d_i u_i$. Если $j>0$, то вычисляем s_{ij} по формуле

$$s_{ij} = s_{i,j-1} q_{i+2^{j-1},j-1} + s_{i+2^{j-1},j-1}\, q_{i,j-1}.$$

Наконец, получаем $s_{0t} = u$, что и нужно для (8.20).

Алгоритм 8.5. Быстрый китайский алгоритм с предварительной обработкой данных

Вход.

1. Попарно взаимно простые целочисленные модули $p_0, p_1, \ldots, p_{k-1}$, где $k = 2^t$ для некоторого t.

2. Множество "обратных" к ним чисел $d_0, d_1, \ldots, d_{k-1}$, т. е. таких, что $d_i = (p/p_i)^{-1} \bmod p_i$, где $p = \prod_{i=0}^{k-1} p_i$.

3. Последовательность вычетов $(u_0, u_1, \ldots, u_{k-1})$.

Выход. Единственное целое число u, $0 \leqslant u < p$, для которого $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

```
    begin
1.  for i ← 0 until k − 1 do s_{i0} ← d_i * u_i;
2.  for j ← 1 until t do
3.      for i ← 0 step 2^j until k − 1 do
4.          s_{ij} ← s_{i,j−1} * q_{i+2^{j−1},j−1} + s_{i+2^{j−1},j−1} * q_{i,j−1};
5.  write s_{0t} mod q_{0t}
    end
```

Рис. 8.4. Программа вычисления целого числа по его модульному представлению.

Метод. Сначала вычисляем $q_{ij}=\prod_{m=i}^{i+2^j-1} p_m$, как в алгоритме 8.4 [1]). Затем запускаем программу на рис. 8.4 в предположении, что в ней

$$s_{ij}=\sum_{m=i}^{i+2^j-1}\frac{q_{ij}d_m u_m}{p_m}. \qquad \square$$

Пример 8.6. Пусть p_0, p_1, p_2, p_3 равны соответственно 2, 3, 5, 7 и $(u_0, u_1, u_2, u_3)=(1, 2, 4, 3)$. Тогда $q_{i0}=p_i$ для $0\leqslant i\leqslant 4$, $q_{01}=6$, $q_{21}=35$ и $q_{02}=p=210$. Далее,

$$\begin{aligned}
d_0&=(3*5*7)^{-1} \bmod 2=1, && \text{поскольку } 1*105\equiv 1 \pmod 2,\\
d_1&=(2*5*7)^{-1} \bmod 3=1, && \text{поскольку } 1*70\equiv 1 \pmod 3,\\
d_2&=(2*3*7)^{-1} \bmod 5=3, && \text{поскольку } 3*42\equiv 1 \pmod 5,\\
d_3&=(2*3*5)^{-1} \bmod 7=4, && \text{поскольку } 4*30\equiv 1 \pmod 7.
\end{aligned}$$

Таким образом, в строке 1 вычисляются

$$\begin{aligned}
s_{00}&=1*1=1, & s_{10}&=1*2=2,\\
s_{02}&=3*4=12, & s_{30}&=4*3=12.
\end{aligned}$$

Затем выполняется цикл в строках 3, 4 с $j=1$. Здесь i принимает значения 0 и 2; следовательно,

$$\begin{aligned}
s_{01}&=s_{00}*q_{10}+s_{10}*q_{00}=1*3+2*2=7,\\
s_{21}&=s_{20}*q_{30}+s_{30}*q_{20}=12*7+12*5=144.
\end{aligned}$$

Далее выполняется цикл в строках 3, 4 с $j=2$, а i принимает только значение 0. Получаем

$$s_{02}=s_{01}*q_{21}+s_{21}*q_{01}=7*35+144*6=1109.$$

[1]) Заметим, что числа q_{ij} зависят только от чисел p_i. Можно включить их во входные данные, а не вычислять, ибо разрешается предварительная обработка. Но легко показать, что на порядок времени работы не влияет, вычислены заранее эти q_{ij} или нет.

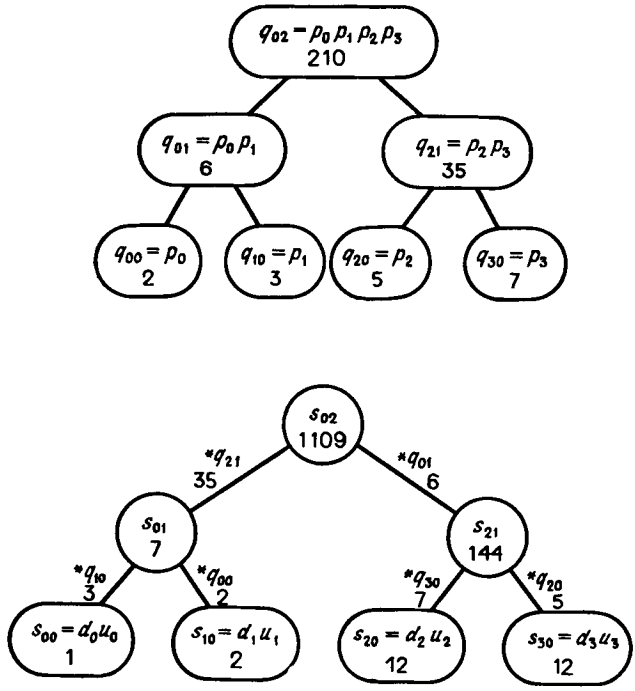


Рис. 8.5. Вычисления из примера 8.6.

Таким образом, результатом строки 5 будет 1109 mod 210, т. е. 59. Можете проверить, что вычеты числа 59 по модулям 2, 3, 5 и 7 равны 1, 2, 4 и 3 соответственно. На рис. 8.5 эти вычисления изображены графически. □

Теорема 8.11. *Алгоритм* 8.5 *правильно вычисляет целое число* u, *для которого* $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Доказательство. Элементарная индукция по j показывает, что s_{ij} принимает нужное значение, т. е.

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

Корректность алгоритма непосредственно следует из леммы 8.2, т. е. из справедливости формулы (8.20). □

Теорема 8.12. *Пусть даны* k *попарно взаимно простых целочисленных модулей* $p_0, p_1, \ldots, p_{k-1}$ *и вычеты* $(u_0, u_1, \ldots, u_{k-1})$. *Если каждое из чисел* p_i *содержит не более* b *битов, то существует алгоритм с предварительной обработкой данных, вычисляющих число*

и, для которого $0 \leqslant u < p = \prod_{i=0}^{k-1} p_i$ *и* $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$, *за время* $O_Б(M(bk)\log k)$, *где* $M(n)$ — *время умножения двух n-битовых чисел.*

Доказательство. Вычисление чисел q_{ij} занимает $O_Б(M(bk)\log k)$ времени [1]). Для анализа тела алгоритма заметим, что s_{ij} содержит не более $b2^j+b+j$ битов, ибо является суммой 2^j слагаемых, каждое из которых равно произведению 2^j+1 целых чисел, состоящих не более чем из b битов. Поэтому каждое слагаемое содержит не более $b(2^j+1)$ битов, а сумма 2^j таких слагаемых — не более $b(2^j+1)+\log(2^j)=b2^j+b+j$ битов. Таким образом, строка 4 занимает время $O_Б(M(b2^j))$. Цикл в строках 3, 4 повторяется $k/2^j$ раз для фиксированного j, так что весь цикл выполняется за

$$O_Б\left(\frac{k}{2^j} M(b2^j)\right)$$

шагов, а в силу обычного предположения о росте функции $M(n)$ это не превосходит $O_Б(M(bk))$. Так как цикл в строках 2—4 итерируется $\log k$ раз, то общая сложность составляет $O_Б(M(bk)\log k)$ шагов. Легко показать, что сложность строки 5 меньше. □

Следствие. *Китайский алгоритм с предварительной обработкой данных, примененный к k модулям по b битов в каждом, работает не более* $O_Б(bk \log k \log bk \log\log bk)$ *времени.*

8.7. КИТАЙСКАЯ ТЕОРЕМА ОБ ОСТАТКАХ И ИНТЕРПОЛЯЦИЯ ПОЛИНОМОВ

Должно быть ясно, что все результаты предыдущего раздела справедливы и для полиномиальных модулей, а не только для целочисленных. Поэтому верны следующая теорема и ее следствие.

Теорема 8.13. *Пусть* $p_0(x), p_1(x), \ldots, p_{k-1}(x)$ — *полиномы степени не больше d и* $M(n)$ — *число арифметических операций, необходимых для умножения двух полиномов степени n. Пусть* $u_0(x), u_1(x), \ldots, u_{k-1}(x)$ — *такие полиномы, что степень полинома* $u_i(x)$ *меньше степени полинома* $p_i(x)$, $0 \leqslant i < k$. *Тогда существует алгоритм с предварительной обработкой данных, вычисляющий за время* $O_A(M(dk)\log k)$ *тот единственный полином* $u(x)$ *степени, меньшей степени полинома* $p(x)=\prod_{i=0}^{k-1} p_i(x)$, *для которого*

$$u(x) \leftrightarrow (u_0(x), u_1(x), \ldots, u_{k-1}(x)).$$

Доказательство. Применяется аналог алгоритма 8.5 и доказательство следует доказательству теоремы 8.12. □

[1]) Поскольку $D(n)$ и $M(n)$ по существу равны, мы предпочитаем везде употреблять $M(n)$.

Следствие. *Существует алгоритм восстановления полиномов по остаткам (согласно китайской теореме) с временной сложностью* $O_A(dk \log k \log dk)$.

Рассмотрим один важный частный случай: все модули имеют степень 1. Если $p_i(x)=x-a_i$, $0\leqslant i<k$, то вычеты (числа u_i) постоянны, т. е. являются полиномами степени 0. Если $u(x)\equiv u_i \pmod{(x-a_i)}$, то $u(x)=q(x)(x-a_i)+u_i$ и, значит, $u(a_i)=u_i$. Таким образом, единственным полиномом $u(x)$ степени, меньшей k, для которого $u(x)\leftrightarrow (u_0, u_1, \ldots, u_{k-1})$, будет тот единственный полином степени, меньшей k, для которого $u(a_i)=u_i$ для каждого i, $0\leqslant i<k$. Другими словами, $u(x)$ — это полином, интерполируемый по значениям u_i в точках a_i, $0\leqslant i<k$.

Поскольку при работе с полиномами интерполяция очень важна, мы с удовольствием отметим, что интерполяцию по значениям в k точках можно выполнить за время $O_A(k \log^2 k)$, даже если нет предварительной обработки данных. Это объясняется тем, что здесь, как мы увидим из следующей леммы, легко найти значения коэффициентов d_i из (8.20) [1]).

Лемма 8.3. *Пусть* $p_i(x)=x-a_i$ *для* $0\leqslant i<k$, *где все* a_i *различны (т. е. все* $p_i(x)$ *взаимно просты). Пусть* $p(x)=\prod_{j=0}^{k-1} p_j(x)$, $c_i(x)=p(x)/p_i(x)$ *и* $d_i(x)$ *— постоянный полином, причем* $d_i(x)c_i(x)\equiv 1 \pmod{p_i(x)}$. *Тогда* $d_i(x)=1/b$, *где* $b=\frac{d}{dx}p(x)\big|_{x=a_i}$.

Доказательство. Запишем $p(x)=c_i(x)\,p_i(x)$, так что

$$\frac{d}{dx}p(x)=p_i(x)\frac{d}{dx}c_i(x)+c_i(x)\frac{d}{dx}p_i(x). \tag{8.21}$$

Далее, $dp_i(x)/dx=1$ и $p_i(a_i)=0$, поэтому

$$\left.\frac{dp(x)}{dx}\right|_{x=a_i}=c_i(a_i). \tag{8.22}$$

Заметим, что $d_i(x)$ обладает тем свойством, что $d_i(x)\,c_i(x)\equiv 1 \pmod{(x-a_i)}$, и, значит, $d_i(x)c_i(x)=q_i(x)(x-a_i)+1$ при некотором $q_i(x)$. Таким образом, $d_i(a_i)=1/c_i(a_i)$. Теперь лемма немедленно следует из (8.22), поскольку $d_i(x)$ — постоянная. □

Теорема 8.14. *Интерполяцию полинома по значениям в* k *точках можно выполнить за время* $O_A(k \log^2 k)$ *без предварительной обработки данных.*

Доказательство. В силу леммы 8.3 вычисление полиномов d_i эквивалентно вычислению значений производной некоторого

[1]) Как упоминалось в разд. 8.6, эта задача в действительности несложна и в общем случае. Но в общем случае нужна техника следующего раздела.

полинома степени $k-1$ в k точках. Полином $p(x)=\prod_{i=0}^{k-1} p_i(x)$ можно получить за время $O_A(k \log^2 k)$, если сначала вычислить p_0p_1, p_2p_3, затем $p_0p_1p_2p_3$, $p_4p_5p_6p_7$, ... и т. д. Производную полинома $p(x)$ можно найти за $O_A(k)$ шагов. Вычисление значений производной занимает $O_A(k \log^2 k)$ времени в силу следствия 2 теоремы 8.10. Теорема вытекает теперь из следствия теоремы 8.13 при $d=1$. □

Пример 8.7. Проинтерполируем полином по следующим парам (точка, значение): (1, 2); (2, 7); (3, 4); (4, 8). Здесь $a_i=i+1$ для $0\leqslant i<4$, $u_0=2$, $u_1=7$, $u_2=4$ и $u_3=8$. Тогда $p_i(x)=x-i-1$, а полином $p(x)=\prod_{i=0}^{3} p_i(x)$ равен $x^4-10x^3+35x^2-50x+24$. Далее, $dp(x)/dx=4x^3-30x^2+70x-50$, и в точках 1, 2, 3, 4 этот полином принимает соответственно значения -6, 2, -2, 6. Таким образом, d_0, d_1, d_2 и d_3 равны соответственно $-1/6$, $1/2$, $-1/2$ и $1/6$. С помощью быстрого китайского алгоритма 8.5, переделанного для полиномов, получаем

$$s_{01}=d_0u_0p_1+d_1u_1p_0=\left(-\frac{1}{6}\right)(2)(x-2)+\left(\frac{1}{2}\right)(7)(x-1)=$$
$$=\frac{19}{6}x-\frac{17}{6},$$
$$s_{21}=d_2u_2p_3+d_3u_3p_2=\left(-\frac{1}{2}\right)(4)(x-4)+\left(\frac{1}{6}\right)(8)(x-3)=$$
$$=-\frac{2}{3}x+4$$

и затем

$$s_{02}=u(x)=s_{01}q_{21}+s_{21}q_{01}=$$
$$=\left(\frac{19}{6}x-\frac{17}{6}\right)(x^2-7x-12)+\left(-\frac{2}{3}x+4\right)(x^2-3x+2),$$
$$u(x)=\frac{5}{2}x^3-19x^2+\frac{89}{2}x-26.$$ □

Как отмечалось в гл. 7, такие арифметические операции над полиномами, как сложение и умножение, можно выполнять, вычисляя полиномы в n точках, производя над найденными значениями арифметические операции и затем интерполируя полином по полученным в результате значениям. Если на выходе должен быть полином степени не более $n-1$, то этот способ приведет к правильному ответу.

Метод БПФ именно это и делает в случае, когда в качестве n точек берутся числа ω^0, ω^1, ..., ω^{n-1}. Здесь алгоритмы вычисления значений и интерполяции оказались особенно простыми благодаря ствойствам степеней числа ω и специального упорядочения их. Однако стоит заметить, что вместо степеней числа ω можно было бы взять любой набор точек. Тогда у нас было бы такое "преобразова-

ние", что на всю задачу (преобразование, вычисления и обратное преобразование) потребовалось бы $O_A(n \log^2 n)$, а не $O_A(n \log n)$ времени.

8.8. НАИБОЛЬШИЕ ОБЩИЕ ДЕЛИТЕЛИ И АЛГОРИТМ ЕВКЛИДА

Определение. Пусть a_0 и a_1— положительные целые числа. Положительное целое число g называется *наибольшим общим делителем* чисел a_0 и a_1 (его часто обозначают через НОД(a_0, a_1)), если

1) g делит a_0 и a_1,
2) всякий общий делитель чисел a_0 и a_1 делит g.

Легко показать, что для положительных целых чисел a_0 и a_1 такое число g единственно. Например, НОД(57, 33)=3.

Алгоритм Евклида для вычисления НОД(a_0, a_1) состоит в вычислении *последовательности остатков* $a_0, a_1, \ldots, a_k$, где a_i, $2 \leqslant i \leqslant k$,— ненулевой остаток от деления a_{i-2} на a_{i-1} и a_k нацело делит a_{k-1} (т. е. $a_{k+1}=0$). В этой ситуации НОД$(a_0, a_1)=a_k$.

Пример 8.8. В примере, приведенном выше, $a_0=57$, $a_1=33$, $a_2=24$, $a_3=9$, $a_4=6$ и $a_5=3$. Поэтому $k=5$ и НОД(57, 33)=3. □

Теорема 8.15. *Алгоритм Евклида правильно находит значение* НОД(a_0, a_1).

Доказательство. Этот алгоритм вычисляет $a_{i+1}=a_{i-1}-q_i a_i$ для $1 \leqslant i < k$, где $q_i = \lfloor a_{i-1}/a_i \rfloor$. Так как $a_{i+1} < a_i$ при $i \geqslant 1$, то алгоритм, очевидно, заканчивает работу. Кроме того, любой общий делитель чисел a_{i-1} и a_i делит a_{i+1} и любой общий делитель чисел a_i и a_{i+1} делит a_{i-1}. Следовательно, НОД$(a_0, a_1)=$ =НОД$(a_1, a_2)=\ldots=$НОД(a_{k-1}, a_k). Очевидно, НОД$(a_{k-1}, a_k)=a_k$, так что теорема доказана. □

Алгоритм Евклида можно расширить так, чтобы он находил не только наибольший общий делитель чисел a_0 и a_1, но также и целые числа x и y, для которых $a_0x+a_1y=$НОД(a_0, a_1). Сформулируем этот алгоритм.

Алгоритм 8.6. Расширенный алгоритм Евклида

Вход. Положительные целые числа a_0 и a_1.

Выход. НОД(a_0, a_1) и такие целые числа x и y, что $a_0x+a_1y=$ =НОД(a_0, a_1).

Метод. Применяем программу, приведенную на рис. 8.6. □

```
   begin
1.   x_0 ← 1; y_0 ← 0; x_1 ← 0; y_1 ← 1; i ← 1;
2.   while a_i не делит a_{i-1} do
       begin
3.       q ← ⌊a_{i-1}/a_i⌋;
4.       a_{i+1} ← a_{i-1} − q * a_i;
5.       x_{i+1} ← x_{i-1} − q * x_i;
6.       y_{i+1} ← y_{i-1} − q * y_i;
7.       i ← i + 1
       end;
8.   write a_i; write x_i; write y_i
   end
```

Рис. 8.6. Расширенный алгоритм Евклида.

Пример 8.9. Если $a_0=57$ и $a_1=33$, то для чисел a_i, x_i, и y_i получаем значения

i	a_i	x_i	y_i
0	57	1	0
1	33	0	1
2	24	1	−1
3	9	−1	2
4	6	3	−5
5	3	−4	7

Заметим, что $57\times(-4)+33\times 7=3$. □

Ясно, что алгоритм 8.6 правильно вычисляет НОД(a_0, a_1), поскольку очевидно, что числа a_i образуют требуемую последовательность остатков. В следующей лемме описывается важное свойство чисел x_i и y_i, вычисляемых алгоритмом 8.6.

Лемма 8.4. *В алгоритме* 8.6

$$a_0x_i+a_1y_i=a_i \quad при \quad i\geqslant 0. \tag{8.23}$$

Доказательство. Равенство (8.23) справедливо для $i=0$ и $i=1$ в силу строки 1 алгоритма 8.6. Допустим, что (8.23) справедливо для $i-1$ и i. Тогда $x_{i+1}=x_{i-1}-qx_i$ в силу строки 5 и $y_{i+1}=y_{i-1}-qy_i$ в силу строки 6. Следовательно,

$$a_0x_{i+1}+a_1y_{i+1}=a_0x_{i-1}+a_1y_{i-1}-q\,(a_0x_i+a_1y_i). \tag{8.24}$$

По предположению индукции и в силу (8.24)

$$a_0x_{i+1}+a_1y_{i+1}=a_{i-1}-qa_i.$$

Так как $a_{i-1}-qa_i=a_{i+1}$ в силу строки 4, то лемма доказана. □

Заметим, что лемма 8.4 даже не зависит от способа вычисления q в строке 3, хотя строка 3 существенна, чтобы гарантировать, что алгоритм 8.6 действительно вычисляет НОД(a_0, a_1). Суммируем эти замечания в следующей теореме.

Теорема 8.16. *Алгоритм* 8.6 *вычисляет* НОД(a_0, a_1) *и такие числа* x *и* y, *что* $a_0x+a_1y=$НОД(a_0, a_1).

Доказательство. Элементарное упражнение на применение леммы 8.4. □

Введем некоторые обозначения, которые понадобятся нам в следующем разделе.

Определение. Пусть a_0 и a_1— целые числа и $a_0, a_1, \ldots, a_k$— соответствующая последовательность остатков. Пусть $q_i=\lfloor a_{i-1}/a_i \rfloor$, $1\leqslant i\leqslant k$. Определим (2×2)-матрицу $R_{ij}^{(a_0,\ a_1)}$ (или R_{ij}, если ясно, каковы a_0 и a_1) для $0\leqslant i\leqslant j\leqslant k$ условиями

1) $R_{ii}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$ для $i\geqslant 0$,

2) если $j>i$, то $R_{ij}=\begin{bmatrix}0 & 1\\ 1 & -q_j\end{bmatrix}*\begin{bmatrix}0 & 1\\ 1 & -q_{j-1}\end{bmatrix}*\ldots*\begin{bmatrix}0 & 1\\ 1 & -q_{i+1}\end{bmatrix}$.

Пример 8.10. Пусть $a_0=57$ и $a_1=33$, последовательность остатков есть 57, 33, 24, 9, 6, 3, а частные q_i, $1\leqslant i\leqslant 4$, равны 1, 1, 2, 1. Тогда □ □

$$R_{04}^{(57,33)}=\begin{bmatrix}0 & 1\\ 1 & -1\end{bmatrix}*\begin{bmatrix}0 & 1\\ 1 & -2\end{bmatrix}*\begin{bmatrix}0 & 1\\ 1 & -1\end{bmatrix}*\begin{bmatrix}0 & 1\\ 1 & -1\end{bmatrix}=\begin{bmatrix}3 & -5\\ -4 & 7\end{bmatrix}.$$ □

Два интересных свойства этих матриц описаны в следующей лемме.

Лемма 8.5.

(а) $\begin{bmatrix}a_j\\ a_{j+1}\end{bmatrix}=R_{ij}\begin{bmatrix}a_i\\ a_{i+1}\end{bmatrix}$ *для* $i<j<k$,

(б) $R_{0j}=\begin{bmatrix}x_j & y_j\\ x_{j+1} & y_{j+1}\end{bmatrix}$ *для* $0\leqslant j<k$,

где числа a_i, x_i, y_i *определяются расширенным алгоритмом Евклида.*

Доказательство. Элементарное упражнение на применение индукции. □

Заметим, что все построения этого раздела годятся и для полиномов от одной переменной. Надо лишь учесть, что, в то время как наибольший общий делитель двух целых чисел определяется однозначно, для полиномов с коэффициентами из некоторого поля наибольший общий делитель единствен только с точностью до умножения на элемент поля. Иными словами, если $g(x)$ делит полиномы $a_0(x)$ и $a_1(x)$ и всякий другой их делитель тоже делит $g(x)$, то полином $cg(x)$, где c — отличная от нуля постоянная, также обладает этим свойством. Нас удовлетворит любой полином, который делит $a_0(x)$ и $a_1(x)$ и делится на любой их делитель. Чтобы обеспечить единственность, мы могли бы (но не будем) требовать, чтобы наибольший общий делитель был нормированным, т. е. чтобы коэффициент при его старшем члене был равен 1.

8.9. АСИМПТОТИЧЕСКИ БЫСТРЫЙ АЛГОРИТМ НАХОЖДЕНИЯ НОД ПОЛИНОМОВ

При построении алгоритмов для нахождения наибольшего общего делителя мы меняем нашу схему и рассматриваем сначала случай полиномов, поскольку в случае целых чисел приходится преодолевать дополнительные трудности. Пусть $a_0(x)$ и $a_1(x)$ — два полинома, наибольший общий делитель которых надо вычислить. Мы предполагаем, что $\mathrm{CT}(a_1(x))<\mathrm{CT}(a_0(x))$. Легко добиться выполнения этого условия, а именно: если $\mathrm{CT}(a_0)=\mathrm{CT}(a_1)$, то заменяем полиномы a_0 и a_1 полиномами a_1 и $a_0 \bmod a_1$, т. е. вторым и третьим членами последовательности остатков, и работаем, отправляясь от них.

Разобьем нашу задачу на две части. Первая состоит в построении алгоритма, отыскивающего последний член последовательности остатков, степень которого больше половины степени полинома a_0. Формально, пусть $l(i)$ — то единственное целое число, для которого $\mathrm{CT}(a_{l(i)})>i$ и $\mathrm{CT}(a_{l(i)+1})\leqslant i$. Заметим, что если a_0 имеет степень n, то $l(i)\leqslant n-i-1$ в предположении, что $\mathrm{CT}(a_1)<\mathrm{CT}(a_0)$, поскольку $\mathrm{CT}(a_i)\leqslant\mathrm{CT}(a_{i-1})-1$ для всех $i\geqslant 1$.

Введем рекурсивную процедуру ПНОД (полуНОД), которая по полиномам a_0 и a_1, таким, что $\mathrm{CT}(a_0)>\mathrm{CT}(a_1)$, формирует матрицу R_{0j} (см. разд. 8.8), где $j=l(n/2)$, т. е. a_j— последний член последовательности остатков, степень которого больше половины степени полинома a_0.

В основе ПНОД-алгоритма лежит тот факт, что частные от деления полиномов степеней d_1 и d_2, $d_1>d_2$, зависят только от $2(d_1-d_2)+1$ старших членов делимого и d_1-d_2+1 старших членов делителя. Процедура ПНОД приведена на рис. 8.7.

Пример 8.11. Пусть

$$p_1(x)=x^5+x^4+x^3+x^2+x+1$$

и

$$p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7.$$

Допустим, мы пытаемся вычислить ПНОД(p_1, p_2). Если $a_0 = p_1$ и $a_1 = p_2$, то в строках 2 и 3 $m = 2$ и

$$b_0 = x^3 + x^2 + x + 1, \quad c_0 = x + 1,$$
$$b_1 = x^2 - 2x + 3, \quad c_1 = -x - 7.$$

Затем вызываем ПНОД(b_0, b_1) в строке 4. Можно проверить, что на этом шаге

$$R = \begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix}.$$

---

procedure ПНОД(a_0, a_1):

1. **if** $\mathrm{CT}(a_1) \leqslant \mathrm{CT}(a_0)/2$ **then return** $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

   else

   begin

2. пусть $a_0 = b_0 x^m + c_0$, где $m = \lfloor \mathrm{CT}(a_0)/2 \rfloor$ и $\mathrm{CT}(c_0) < m$;
3. пусть $a_1 = b_1 x^m + c_1$, где $\mathrm{CT}(c_1) < m$;

   comment b_0 и b_1 — старшие члены полиномов a_0 и a_1. Имеем $\mathrm{CT}(b_0) = \lceil \mathrm{CT}(a_0)/2 \rceil$ и $\mathrm{CT}(b_0) - \mathrm{CT}(b_1) = \mathrm{CT}(a_0) - \mathrm{CT}(a_1)$;

4. $R \leftarrow$ ПНОД(b_0, b_1);
5. $\begin{bmatrix} d \\ e \end{bmatrix} \leftarrow R \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$;
6. $f \leftarrow d \bmod e$;

   comment e и f — соседние члены последовательности остатков, их степень не превосходит $\lceil 3m/2 \rceil$, т. е. $^3/_4$ степени полинома a_0;

7. пусть $e = g_0 x^{\lfloor m/2 \rfloor} + h_0$, где $\mathrm{CT}(h_0) < \lfloor m/2 \rfloor$;
8. пусть $f = g_1 x^{\lfloor m/2 \rfloor} + h_1$, где $\mathrm{CT}(h_1) < \lfloor m/2 \rfloor$;

   comment степени полиномов g_0 и g_1 не превосходят $m + 1$;

9. $S \leftarrow$ ПНОД(g_0, g_1);
10. $q \leftarrow \lfloor d/e \rfloor$;
11. **return** $S * \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} * R$

    end

Рис. 8.7. Процедура ПНОД.

В строках 5 и 6 вычисляем

$$d = x^4 - 2x^3 + 3x^2 - x - 7,$$
$$e = 4x^3 - 7x^2 + 11x + 22,$$
$$f = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}.$$

Находим, что $\lfloor m/2 \rfloor = 1$ и, значит, в строках 7 и 8

$$g_0 = 4x^2 - 7x + 11, \qquad h_0 = 22,$$
$$g_1 = -\frac{3}{16}x - \frac{93}{16}, \qquad h_1 = -\frac{45}{8}.$$

Таким образом, в строке 9

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

В строке 10 обнаруживаем, что $q(x)$, т. е. частное $\lfloor d(x)/e(x) \rfloor$, равно [1] $^1/_4x - ^1/_{16}$. Следовательно, в строке 11 получаем результат

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -\left(\frac{1}{4}x - \frac{1}{16}\right) \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix} =$$
$$= \begin{bmatrix} 1 & -(x+3) \\ -\left(\frac{1}{4}x - \frac{1}{16}\right) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Заметим, что

$$T \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix};$$

это верно, ибо в последовательности остатков для p_1 и p_2 член e является последним полиномом, степень которого больше половины степени p_1. □

Рассмотрим матрицу R, вычисляемую в строке 4 процедуры ПНОД. Предположительно, R есть

$$\prod_{j=1}^{l(\lfloor m/2 \rfloor)} \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix},$$

где $q_j(x)$ обозначает j-е частное из последовательности остатков для b_0 и b_1, т. е. $R = R_{0,\, l(\lfloor m/2 \rfloor)}^{(b_0,\, b_1)}$. Однако в строке 5 мы использовали R как матрицу $R_{0,\, l(\lfloor 3m/2 \rfloor)}^{(a_0,\, a_1)}$, чтобы получить d и e, где d — последний член последовательности остатков, степень которого больше $3m/2$. Надо показать, что обе эти интерпретации матрицы R корректны, т. е. $R_{0,\, l(\lfloor 3m/2 \rfloor)}^{(a_0,\, a_1)} = R_{0,\, l(\lfloor m/2 \rfloor)}^{(b_0,\, b_1)}$.

[1] Это вычисление, конечно, можно было бы выполнить сразу после строки 5.

Точно так же мы должны показать, что матрица S, вычисляемая в строке 9, может играть предназначенную ей роль, т. е.

$$S = R^{(g_0, g_1)}_{0, l(\lceil m/2 \rceil)} = R^{(e, f)}_{0, l(m)}.$$

Эти результаты вытекают из следующих лемм.

Лемма 8.6. *Пусть*

$$f(x) = f_1(x)\, x^k + f_2(x), \tag{8.25}$$

где $\mathrm{CT}(f_2) < k$ *и*

$$g(x) = g_1(x)\, x^k + g_2(x), \tag{8.26}$$

где $\mathrm{CT}(g_2) < k$. *Пусть* q *и* q_1 — *частные* $\lfloor f(x)/g(x) \rfloor$ *и* $\lfloor f_1(x)/g_1(x) \rfloor$, r *и* r_1 — *остатки* $f(x)$—$q(x)\, g(x)$ *и* $f_1(x)$—$q_1(x) g_1(x)$ *соответственно. Если* $\mathrm{CT}(f) > \mathrm{CT}(g)$ *и* $k \leqslant 2\,\mathrm{CT}(g) - \mathrm{CT}(f)$ (*т. е.* $\mathrm{CT}(g_1) \geqslant {}^1\!/_2 \mathrm{CT}(f_1)$), то

(а) $q(x) = q_1(x)$,

(б) *в полиномах* $r(x)$ *и* $r_1(x)x^k$ *все члены степени* $k + \mathrm{CT}(f) - \mathrm{CT}(g)$ *и выше совпадают.*

Доказательство. Рассмотрим процесс деления $f(x)$ на $g(x)$ с помощью обычного алгоритма, который делит первый член полинома $f(x)$ на первый член полинома $g(x)$, чтобы получить первый член частного. Первый член частного умножается на $g(x)$ и вычитается из $f(x)$ и т. д. Первые $\mathrm{CT}(g) - k$ членов, полученных таким способом, не зависят от $g_2(x)$. Но частное содержит лишь члены степени $\mathrm{CT}(f) - \mathrm{CT}(g)$. Поэтому если $\mathrm{CT}(f) - \mathrm{CT}(g) \leqslant \mathrm{CT}(g) - k$, т. е. $k \leqslant 2\mathrm{CT}(g) - \mathrm{CT}(f)$, то частное не зависит от $g_2(x)$. Если $\mathrm{CT}(f) - \mathrm{CT}(g) \leqslant \mathrm{CT}(f) - k$, то частное не зависит от $f_2(x)$. Но неравенство $\mathrm{CT}(f) - \mathrm{CT}(g) \leqslant \mathrm{CT}(f) - k$ следует из $k \leqslant 2\mathrm{CT}(g) - \mathrm{CT}(f)$ и $\mathrm{CT}(f) > \mathrm{CT}(g)$. Таким образом, утверждение (а) доказано. Что касается утверждения (б), то аналогичные рассуждения показывают, что члены остатка, имеющие степень $\mathrm{CT}(f) - (\mathrm{CT}(g) - k)$ и выше, не зависят от $g_2(x)$. Аналогично члены остатка, имеющие степень k и выше, не зависят от $f_2(x)$. Но $\mathrm{CT}(f) - \mathrm{CT}(g) + k > k$. Таким образом, в $r(x)$ и $r_1(x)x^k$ все члены степени $\mathrm{CT}(f) - \mathrm{CT}(g) + k$ и выше совпадают. □

Лемма 8.7. *Пусть* $f(x) = f_1(x)x^k + f_2(x)$ *и* $g(x) = g_1(x)x^k + g_2(x)$, *где* $\mathrm{CT}(f_2) < k$ *и* $\mathrm{CT}(g_2) < k$. *Пусть* $\mathrm{CT}(f) = n$ *и* $\mathrm{CT}(g) < \mathrm{CT}(f)$. *Тогда*

$$R^{(f, g)}_{0, l(\lceil (n+k)/2 \rceil)} = R^{(f_1, g_1)}_{0, l(\lceil (n-k)/2 \rceil)},$$

т. е. частные, входящие в последовательности остатков для (f, g) *и* (f_1, g_1), *совпадают по крайней мере до того места, где во второй последовательности появляется остаток степени, не большей половины степени полинома* f_1.

Доказательство. Лемма 8.6 гарантирует совпадение частных и достаточного количества старших членов в соответствую-

щих остатках, входящих в рассматриваемые последовательности. □

Теорема 8.17. *Пусть* $a_0(x)$ *и* $a_1(x)$ *— такие полиномы, что* $\mathrm{CT}(a_0)=n$ *и* $\mathrm{CT}(a_1)<n$. *Тогда* $\mathrm{НОД}(a_0, a_1)=R_{0,\,l(n/2)}$.

Доказательство. Теорема доказывается простой индукцией по n, в которой учитывается лемма 8.7, чтобы гарантировать, что матрица R в строке 4 равна $R^{(a_0,\,a_1)}_{0,\,l(\lceil 3m/2\rceil)}$, а S в строке 9 равна $R^{(a_0,\,a_1)}_{l(\lceil 3m/2\rceil)\,+\,1,\,l\,(m)}$. □

Теорема 8.18. *Процедура* ПНОД *выполняется за* $O_{\mathrm{A}}(M(n)\log n)$ *шагов, если ее аргументы имеют степени не выше* n; *здесь* $M(n)$ *— время умножения двух полиномов степени* n.

Доказательство. Докажем утверждение для случая, когда n степень числа 4. Поскольку время выполнения процедуры ПНОД, очевидно, не убывает с ростом n, то наша теорема будет тогда верна и для произвольного n. Если $\mathrm{CT}(a_0)$ — степень числа 4, то

$$\mathrm{CT}(b_0)=\frac{1}{2}\mathrm{CT}(a_0), \quad \mathrm{CT}(g_0)\leqslant\frac{1}{2}\mathrm{CT}(a_0).$$

Таким образом, время $T(n)$ выполнения процедуры ПНОД для входа степени n удовлетворяет условию

$$T(n)\leqslant 2T\left(\frac{n}{2}\right)+cM(n) \tag{8.27}$$

для некоторой постоянной c. Другими словами, тело процедуры ПНОД включает в себя два вызова той же процедуры для аргументов половинного размера и постоянное число других операций с временной сложностью $O_{\mathrm{A}}(n)$ или $O_{\mathrm{A}}(M(n))$. Решение неравенства (8.27) должно быть известно читателю; оно ограничено сверху функцией $c_1M(n)\log n$ для некоторой постоянной c_1. □

Перейдем к изложению полного алгоритма нахождения наибольших делителей. В нем участвует процедура ПНОД, вычисляющая $R_{0,n/2}$, затем $R_{0,3n/4}$, $R_{0,7n/8}$ и т. д., где n — степень входа.

Алгоритм 8.7. НОД-алгоритм

Вход. Полиномы $p_1(x)$ и $p_2(x)$, для которых $\mathrm{CT}(p_2)<\mathrm{CT}(p_1)$.
Выход. Наибольший общий делитель $\mathrm{НОД}(p_1, p_2)$ для p_1 и p_2.
Метод. Вызываем процедуру $\mathrm{НОД}(p_1, p_2)$, где НОД — рекурсивная процедура, приведенная на рис. 8.8. □

Пример 8.12. Продолжим пример 8.11. Там $p_1(x)=x^5+x^4+x^3+x^2+1$ и $p_2(x)=x^4-2x^3+3x^2-x-7$. Мы уже нашли, что

$$\mathrm{ПНОД}(p_1, p_2)=\begin{bmatrix} 1 & -(x+3) \\ -\left(\frac{1}{4}x-\frac{1}{16}\right) & \frac{1}{4}x^2+\frac{11}{16}x+\frac{13}{16} \end{bmatrix}.$$

```
   procedure НОД (a_0, a_1):
1. if a_1 делит a_0 then return a_1
   else
     begin
2.     R ← ПНОД (a_0, a_1);
3.     [b_0; b_1] ← R * [a_0; a_1];
4.     if b_1 делит b_0 then return b_1
       else
         begin
5.         c ← b_0 mod b_1;
6.         return НОД (b_1, c)
         end
     end
```

Рис. 8.8. Процедура НОД.

Поэтому в строке 3 вычисляем

$$b_0 = 4x^3 - 7x^2 + 11x + 22,$$
$$b_1 = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}.$$

Обнаруживаем, что b_1 не делит b_0. В строке 5 находим, что

$$b_0 \bmod b_1 = 3952x + 3952.$$

Так как последний полином делит b_1, то вызов процедуры НОД в строке 6 завершает работу в строке 1 и выдает в качестве ответа $3952x+3952$. Разумеется, $x+1$ также является наибольшим общим делителем для p_1 и p_2. □

Доказательство корректности алгоритма 8.7 тривиально, если доказать, что он заканчивает свою работу. Таким образом, корректность алгоритма вытекает из анализа времени его работы, что составляет содержание следующей теоремы.

Теорема 8.19. *Если* $\mathrm{CT}(p_1)=n$, *то алгоритм* 8.7 *выполняется за* $O_A(M(n) \log n)$ *шагов, где* $M(n)$ *— время умножения двух полиномов степени* n.

Доказательство. Неравенство

$$T(n) \leqslant T\left(\frac{n}{2}\right) + c_1 M(n) + c_2 M(n) \log n, \tag{8.28}$$

где c_1 и c_2 — постоянные, описывает время работы алгоритма 8.7. Действительно, степень полинома b_1 меньше половины степени

полинома a_0, так что первое слагаемое в (8.28) отражает рекурсивный вызов в строке 6. Слагаемое $c_1M(n)$ отражает деления и умножения в строках 1, 3—5, $c_2M(n)\log n$ — вызов процедуры ПНОД в строке 2. Легко видеть, что решение неравенства (8.28) ограничено сверху функцией $kM(n)\log n$, где k — некоторая постоянная. □

Следствие. НОД *двух полиномов степени не выше* n *можно вычислить за время* $O_A(n\log^2 n)$.

8.10. НОД ЦЕЛЫХ ЧИСЕЛ

Обсудим кратко изменения, которые надо произвести в процедурах ПНОД и НОД, чтобы они годились для целых чисел. Чтобы понять, где возникают трудности, вернемся к лемме 8.6, показывающей, что при нахождении частных от деления полиномов степеней n и $n-d$ нам ни в одном из них не нужны члены степени, меньшей $n-2d$.

По аналогии с леммой 8.6 можно рассмотреть два целых числа f и g, $f>g$, и записать их в виде $f=f_1 2^k+f_2$ и $g=g_1 2^k+g_2$, где $f_2<2^k$ и $g_2<2^k$. Вместо условия $\mathrm{CT}(g_1(x))\geqslant{}^1/_2\mathrm{CT}(f_1(x))$ можно предполагать, что $f_1\leqslant(g_1)^2$. Тогда можно считать, что $f=qg+r$ и $f_1=q_1g_1+r_1$. Комбинируя эти формулы, получаем

$$g_1 2^k\ (q-q_1)=f_2+r_1 2^k-qg_2-r. \tag{8.29}$$

Так как $r_1<g_1$, $f_2<2^k$ и все эти целые числа неотрицательны, легко вывести из (8.29), что $q-q_1\leqslant 0$. Пусть $q=q_1-m$ для некоторого $m\geqslant 0$. Тогда из (8.29) заключаем, что

$$mg_1 2^k\leqslant qg_2+r=(q_1-m)\,g_2+r.$$

Следовательно,

$$mg=mg_1 2^k+mg_2\leqslant q_1g_2+r. \tag{8.30}$$

Поскольку $f_1\leqslant(g_1)^2$, то $q_1\leqslant g_1$. Кроме того, по условию $g_2<2^k$ и $r<g$, так что из (8.30) вытекает, что

$$mg<g_1 2^k+g\leqslant 2g. \tag{8.31}$$

Теперь из (8.31) непосредственно следует, что $m<2$. Отсюда заключаем, что или $q=q_1$, или $q=q_1-1$.

В первом случае нет трудностей. Если же $q=q_1-1$, то нельзя рассчитывать, что процедура ПНОД сработает правильно. К счастью, можно показать, что $q_1\neq q$, только если q — последнее частное в последовательности остатков, для которого процедура ПНОД строит матрицу. Иными словами, подставляя $q-q_1=-1$ в (8.29), получаем

$$r=f_2+r_1 2^k-qg_2+g_1 2^k. \tag{8.32}$$

Так как $r<g=g_1 2^k+g_2$, то из (8.32) вытекает $r_1 2^k+f_2<g_2(1+q)$ и тем более $r_1<1+q$, а значит, $r_1<q_1$.

Итак, число r_1, которое стоит в последовательности остатков после f_1 и g_1, должно быть меньше f_1/g_1. Поскольку $g_1\geqslant\sqrt{f_1}$, то $r_1<\sqrt{f_1}$, а это означает, что процедура ПНОД выдала бы матрицу, содержащую частные вплоть до f_1/g_1, но не далее. Если эта матрица используется в строке 5 процедуры ПНОД, то может оказаться, что число f, вычисляемое в строке 6, будет не меньше $a_0^{3/4}$. Но поскольку ошибка в последнем частном равна всего 1, можно показать, что продолжение последовательности остатков на ограниченное число членов (не зависящее от размера a_0) после выполнения строки 6 будет достаточно для того, чтобы один из очередных членов этой последовательности стал меньше $a_0^{3/4}$. Аналогичный прием надо применить и после строки 5 процедуры НОД.

Еще один круг трудностей связан с леммой 8.6. В случае полиномов мы смогли показать, что в последовательности остатков, образованной с привлечением старших членов полиномов, определенные члены высокого порядка совпадают с соответствующими членами в последовательности, образованной по всем членам полиномов. Аналогичный результат приближенно выполняется и для целых чисел при условии $q=q_1$. Однако можно только ограничить разность между r и $r_1 2^k$ числом $2^k(q+1)$; нельзя гарантировать, что какие-нибудь конкретные разряды чисел r и $r_1 2^k$ будут совпадать. Тем не менее такой источник "ошибок округления" приведет к тому, что после выполнения строки 6 процедуры ПНОД и строки 5 процедуры НОД в последовательности остатков понадобится лишь ограниченное число дополнительных членов.

Из-за продолжения последовательности остатков на ограниченное число членов, сложность алгоритма увеличивается на $O_Б(M(n))$, где $M(n)$ — время умножения n-разрядных двоичных чисел, так что анализ времени работы процедур ПНОД и НОД, по существу, не изменится. Поэтому справедлива следующая теорема.

Теорема 8.20. *Если $M(n)$ — время умножения двух n-разрядных двоичных целых чисел, то существует алгоритм, отыскивающий их наибольший общий делитель за $O_Б(M(n)\log n)$ шагов.*

Доказательство. Упражнение на предложенную выше модификацию процедур ПНОД и НОД. □

Следствие. НОД *двух целых чисел можно найти за время* $O_Б(n\log^2 n\log\log n)$. □

8.11. ЕЩЕ РАЗ О ПРИМЕНЕНИИ КИТАЙСКОЙ ТЕОРЕМЫ ОБ ОСТАТКАХ

Мы обещали показать, как применить НОД-алгоритм для построения асимптотически быстрого алгоритма, который восстанавливал бы целое число по его вычетам без предварительной обработки данных. Вспомним, что алгоритм 8.5, использующий предварительную обработку, тратит $O_Б(M(bk)\log k)$ времени на восстановление числа u по вычетам, взятым по k модулям, содержащим по b двоичных разрядов. Задача состоит в вычислении $d_i=(p/p_i)^{-1} \bmod p_i$, где $p_0, p_1, \ldots, p_{k-1}$ — модули, а p — их произведение.

С помощью очевидного алгоритма, основанного на приеме "разделяй и властвуй", можно вычислить p, найдя сначала произведения пар чисел p_i, затем четверок чисел p_i и т. д., за время $O_Б(M(bk)\log k)$. Техника алгоритма 8.5 позволяет вычислить $e_i=(p/p_i) \bmod p_i$ при $0\leqslant i<k$ за $O_Б(M(bk)\log k)$ шагов без предварительной обработки данных. Остается оценить время, необходимое для вычисления $d_i=e_i^{-1} \bmod p_i$.

Так как p/p_i — произведение всех модулей, отличных от p_i, оно должно быть взаимно просто с p_i. Если представить p/p_i в виде qp_i+e_i, где q — некоторое целое число, то в силу предыдущего e_i и p_i будут взаимно просты, т. е. НОД$(e_i, p_i)=1$. Поэтому, если x и y таковы, что $e_ix+p_iy=1$, то $e_ix\equiv 1 \pmod{p_i}$. Следовательно, $x\equiv e_i^{-1}=d_i \bmod p_i$. Но такие числа x и y вычисляются расширенным алгоритмом Евклида.

Хотя процедура НОД была разработана, чтобы вычислять только НОД(p_1, p_2), мы строили процедуру ПНОД так, чтобы она вычисляла матрицу $R_{0,l(n/2)}$. Поэтому простая модификация НОД-алгоритма могла бы вычислять и $R_{0,l(0)}$. Тогда можно было бы получить x, поскольку это левый верхний элемент матрицы $R_{0,l(0)}$. Сформулируем результат о времени работы китайского алгоритма без предварительной обработки данных.

Теорема 8.21. *В случае целых чисел китайский алгоритм имеет сложность* $O_Б(M(bk)\log k)+O_Б(kM(b)\log b)$, *где* k *— число используемых модулей, содержащих* b *двоичных разрядов каждый.*

Доказательство. В силу анализа, проведенного выше, первый член выражения для сложности соответствует вычислению чисел e_i и выполнению алгоритма 8.5. Второй соответствует вычислению чисел d_i, поскольку вычисление чисел x и y можно вести по $\bmod p_i$, используя везде b-битовые операции. □

Следствие. *Китайский алгоритм без предварительной обработки данных имеет временную сложность* $O_Б(bk\log^2 bk\log\log bk)$ [1]).

[1])Интересно заметить, что при $M(n)=n\log n\log\log n$ эта величина наилучшая, которую можно получить из теоремы 8.21 независимо от того, как соотносятся b и k.

8.12. РАЗРЕЖЕННЫЕ ПОЛИНОМЫ

Представляя полином от одной переменной в виде $\sum_{i=0}^{n-1} a_i x^i$, мы предполагали до сих пор, что он *плотный*, т. е. отличны от нуля почти все его коэффициенты. Для многих приложений полезно предполагать, что полином *разрежен*, т. е. число ненулевых коэффициентов много меньше его степени. В такой ситуации логично представлять полином списком пар (a_i, j_i), состоящих из ненулевого коэффициента и соответствующего ему показателя степени переменной x.

Мы не можем углубляться в разнообразную технику, известную для выполнения арифметических операций над разреженными полиномами; упомянем лишь два интересных аспекта этой теории. Во-первых, поскольку неразумно применять преобразование Фурье для выполнения умножения, мы дадим один разумный алгоритм умножения разреженных полиномов. Во-вторых, на примере вычисления $[p(x)]^4$ продемонстрируем удивительное различие между способом, которым следует выполнять арифметические операции над плотными полиномами и аналогичным способом для разреженных полиномов.

Наиболее разумная из известных стратегий умножения разреженных полиномов состоит в том, что полином $\sum_{i=1}^{n} a_i x^{j_i}$ задается списком пар $(a_1, j_1), (a_2, j_2), \ldots, (a_n, j_n)$, где все j_i различны и расположены в порядке убывания, т. е. $j_i > j_{i+1}$ для $1 \leqslant i < n$. Чтобы умножить два полинома, представленных таким образом, находим произведения пар и располагаем их по величине показателей (по вторым компонентам пар), объединяя все члены с одинаковыми показателями. Если это не делать, то придется расплачиваться ростом числа членов с одинаковыми показателями. Поэтому по мере выполнения арифметических операций сложность начнет значительно превосходить ту, которая была бы, если бы приведение подобных членов выполнялось на каждом шаге.

При умножении упорядоченных разреженных полиномов можно извлечь пользу из их упорядоченности и из того, что в одном из них может оказаться много больше членов, чем в другом, чтобы максимально упростить упорядочение множества мономов произведения. Приведем неформальный алгоритм умножения разреженных полиномов таким способом.

Алгоритм 8.8. Умножение упорядоченных разреженных полиномов

Вход. Полиномы $f(x) = \sum_{i=1}^{m} a_i x^{j_i}$ и $g(x) = \sum_{i=1}^{n} b_i x^{k_i}$, представленные списками пар

$$(a_1, j_1), (a_2, j_2), \ldots, (a_m, j_m)$$

и

$$(b_1, k_1), (b_2, k_2), \ldots, (b_n, k_n),$$

где последовательности $j_1, \ldots, j_m$ и $k_1, \ldots, k_n$ монотонно убывают.

Выход. Полином $\sum_{i=1}^{p} c_i x^{l_i} = f(x)g(x)$, представленный списком пар, в котором последовательность $l_1, \ldots, l_p$ монотонно убывает.

Метод. Не умаляя общности, предполагаем, что $m \geqslant n$.

1. Строим последовательности S_i, $1 \leqslant i \leqslant n$, в которых r-й член, $1 \leqslant r \leqslant m$, равен $(a_r b_i, j_r + k_i)$. Таким образом, S_i представляет произведение полинома $f(x)$ на i-й член полинома $g(x)$.

2. Сливаем S_{2i-1} с S_{2i} для $1 \leqslant i \leqslant n/2$, приводя подобные члены. Затем попарно сливаем полученные последовательности, приводя подобные члены, и повторяем процесс, пока не получим одну упорядоченную последовательность. □

Теорема 8.22. *Алгоритм* 8.8 *занимает* $O(mn \log n)$ *времени* [1]) *в предположении, что* $m \geqslant n$.

Доказательство. Шаг 1, очевидно, имеет сложность $O(m, n)$. Шаг 2 надо повторить $\lceil \log n \rceil$ раз, и ясно, что при каждом выполнении вся работа займет $O(mn)$ времени. □

Посмотрим теперь, как алгоритм 8.8 и его временная сложность влияют на выбор способа выполнения арифметических операций над разреженными полиномами.

Пример 8.13. Вычислим $p^4(x)$, где $p(x)$ — полином с n членами в случаях, когда он плотный и когда разреженный. Если полином p плотный, то легко показать, что лучше всего вычислять $p^4(x)$, дважды возводя в квадрат. Иными словами, пусть $M(n) = cn \log n$, где $M(n)$ — время умножения плотных полиномов. Тогда можно вычислить $p^2(x)$ за $cn \log n$ шагов и результат возвести в квадрат за $2cn \log 2n$ шагов, всего затратив $3cn \log n + 2cn$ шагов. Для сравнения укажем, что если вычислять $p^4 = p \times (p \times (p \times p))$, то, как легко видеть, требуется $6cn \log n$ шагов в предположении, что на $p \times p^2$ тратится $2M(n)$ шагов, а на $p \times p^3$ тратится $3M(n)$ шагов. Таким образом, для плотных полиномов получаем, как и ожидали, что p^4 надо вычислять повторным возведением в квадрат.

Теперь пусть $p(x)$ — разреженный полином с n членами. Если вычислять $(p^2)^2$ по алгоритму 8.8, то на первое возведение в квадрат уходит $cn^2 \log n$ времени, а в предположении, что приводятся немного подобных членов, второе возведение в квадрат занимает

[1]) Заметим, что здесь фигурирует сложность на RAM, а не арифметическая, поскольку программа для алгоритма 8.8 неизбежно содержит разветвления, даже если m и n фиксированы.

$cn^4 \log n^2$ времени, так что всего тратится $c(2n^4+n^2)\log n$ шагов. С другой стороны, $p\times(p\times(p\times p))$ вычисляется за $cn^2\log n+cn^3\log n+cn^4\log n=c(n^4+n^3+n^2)\log n$ шагов. Эта величина меньше времени повторного возведения в квадрат. Таким образом, повторное возведение в квадрат разреженного полинома не всегда дает хороший способ вычисления p^4. Особенно этот эффект проявляется при вычислении p^{2^t} для больших значений t. □

УПРАЖНЕНИЯ

8.1. Используйте алгоритм 8.1 для нахождения числа, "обратного" к 429.

8.2. Используйте алгоритм 8.2 для нахождения 429^2.

8.3. Используйте алгоритм 8.3 для нахождения полинома, "обратного" к

$$x^7-x^6+x^5-3x^4+x^3-x^2+2x+1.$$

*8.4. Опишите алгоритм, аналогичный алгоритму 8.2, для вычисления квадратов полиномов.

8.5. Используйте ваш алгоритм из упр. 8.4, чтобы вычислить $(x^3-x^2+x-2)^2$.

8.6. Используйте алгоритм 8.4, чтобы найти представление для числа 1 000 000 по модулям 2, 3, 5, 7, 11, 13, 17, 19.

8.7. Полностью напишите алгоритм нахождения вычетов полинома по набору полиномиальных модулей.

8.8. Найдите вычеты полинома $x^7+3x^6+x^5+3x^4+x^2+1$ по модулям $x+3$, x^3-3x+1, x^2+x-2 и x^2-1.

8.9. Полиномы в упр. 8.8 были тщательно подобраны, чтобы можно было легко провести вычисления вручную. Случайно выберите четыре полинома степеней 1, 2, 3 и 4 и найдите относительно них вычеты полинома x^9-4. Что произойдет?

8.10. Пусть 5, 6, 7, 11 — четыре модуля. Найдите такое число u, меньшее их произведения, что $u \leftrightarrow (1, 2, 3, 4)$.

*8.11. Обобщите лемму 8.3 так, чтобы применить ее к произвольным полиномиальным модулям, корни которых известны или могут быть найдены (например, полиномы степени меньше 5). Какова сложность восстановления полиномов по их вычетам, если использовать ваш алгоритм (сложность рассматривается как функция от числа и наибольшей степени модулей)?

8.12. Найдите полином, значения которого в точках 0, 1, 2, 3 равны соответственно 1, 1, 2, 2.

8.13. Найдите наибольший общий делитель полиномов

$$x^6+3x^5+3x^4+x^3-x^2-x-1,$$
$$x^6+2x^5+x^4+2x^3+2x^2+x+1.$$

8.14. Полиномы в упр. 8.13 были тщательно подобраны, чтобы можно было легко провести вычисления вручную. Выберите два произвольных полинома степеней 7 и 8 и попробуйте найти их НОД. Что произойдет? Каким по вашему предположению окажется НОД?

***8.15.** Полностью напишите алгоритм нахождения НОД целых чисел, который работал бы время $O_{\text{Б}}(n\log^2 n\log\log n)$ на n-разрядных двоичных целых числах.

8.16. Используйте ваш алгоритм из упр. 8.15, чтобы найти НОД (377, 233).

***8.17.** Пусть $p(x)$ — разреженный полином с n мономами. Найдите как функцию от n сложность наилучшего метода вычисления $p^8(x)$ при условии, что умножения выполняются по алгоритму 8.8.

****8.18.** Предлагается следующий метод вычисления $f(x)/g(x)$ по полиномам f и g в предположении, что g делит f. Пусть f и g имеют степень $n-1$ или меньше.

(1) Вычислить дискретные преобразования Фурье F и G полиномов f и g соответственно.
(2) Разделить члены полинома F на соответствующие члены полинома G; получится последовательность H.
(3) Взять обратное преобразование от H. Считаем, что результатом будет f/g. Будет ли работать этот алгоритм?

****8.19.** Пусть $M(n)$ — время умножения двух n-разрядных двоичных чисел и $Q(n)$ — время нахождения $\lfloor\sqrt{i}\rfloor$ для n-разрядного двоичного целого числа i. Предположим, что $M(an)\geqslant aM(n)$ при $a\geqslant 1$ и аналогичное неравенство верно для $Q(n)$. Покажите, что $M(n)$ и $Q(n)$ равны с точностью до постоянного множителя.

***8.20.** Обобщите упр. 8.19 на (а) полиномы, (б) корни r-й степени при фиксированном r.

****8.21.** Напишите алгоритм сложности $O_{\text{А}}(n\log n)$, который вычислял бы полином степени $n-1$ и все его производные в данной точке.

****8.22.** Плотный полином от r переменных [1]) можно представить в виде

$$\sum_{i_1=0}^{n-1}\sum_{i_2=0}^{n-1}\cdots\sum_{i_r=0}^{n-1}a_{i_1i_2\ldots i_r}x_1^{i_1}x_2^{i_2}\ldots x_r^{i_r}.$$

[1]) По мере роста r предположение о плотности полинома становится все менее полезным.

Покажите, что можно умножать такие полиномы за время $O_A(n^r \log n)$, если для вычисления значений и интерполяции использовать точки $x_1=\omega^{j_1}$, $x_2=\omega^{j_2}$, . . . , $x_r=\omega^{j_r}$, $0 \leqslant j_k < 2n$, где ω — примитивный корень степени $2n$ из единицы.

**8.23. Покажите, что при разумных допущениях о гладкости функций $M(n)$ и $D(n)$ время умножения и время деления плотных полиномов от r переменных, т. е. $M(n)$ и $D(n)$, равны с точностью постоянного множителя.

**8.24. Найдите алгоритм сложности $O_Б(n \log^2 n \log\log n)$, который переводил бы (а) n-разрядные двоичные числа в десятичные; (б) n-разрядные десятичные числа в двоичные.

**8.25. *Наименьшим общим кратным* (НОК) целых чисел (полиномов) x и y — называется целое число (некоторый полином), которое делится на x и y и делит всякое другое целое число (всякий другой полином), делящееся (делящийся) на x и y. Покажите, что время нахождения НОК двух n-разрядных двоичных чисел не меньше времени их умножения.

8.26. Порождает ли метод умножения целых чисел из разд. 2.6 алгоритм умножения полиномов за время $O_A(n^{1.59})$?

**8.27. Покажите, что за $O_A(n \log n)$ шагов можно вычислить полином степени $n-1$ в точках a^0, a^1, . . ., a^{n-1}. *Указание*: Покажите, что $c_j=\sum_{i=0}^{n-1} b_i a^{ij}$ можно записать как $c_j=\sum_{i=0}^{n-1} f(i)g(j-i)$ для некоторых функций f и g.

**8.28. Покажите, что за $O_A(n \log n)$ шагов можно вычислить полином степени $n-1$ в n точках $ba^{2j}+ca^j+d$, $0 \leqslant j \leqslant n$.

8.29. Укажите рекурсивные варианты алгоритмов 8.4 и 8.5.

Проблемы для исследования

8.30. В этой главе мы показали, что много задач о полиномах и целых числах

(1) имеют, по существу, ту же сложность, что и умножение, или

(2) сложнее умножения не более чем в логарифмический множитель.

Некоторые из этих задач приводятся в упражнениях к данной главе. Упр. 9.9 дает еще одну задачу из группы (2) — задачу о (и/или)-умножении.

Разумно попытаться пополнить множество задач группы (1) или (2). Другая область исследования — показать, что некоторые задачи группы (2) должны иметь одинаковые сложности. Например,

можно предположить, что это верно для задач нахождения НОД и НОК.

8.31. Еще одна проблема, которая кажется обманчиво простой: выяснить, является ли алгоритм 8.8 наилучшим алгоритмом умножения упорядоченных разреженных полиномов и упорядочения результата. Некоторые аспекты этой проблемы рассмотрел Джонсон [1974].

8.32. Значение n, при котором многие из описанных алгоритмов становятся практичными, огромно. Однако их можно было бы тщательно скомбинировать для небольших n с $O(n^{1,59})$-методами, упомянутыми в разд. 2.6 и упр. 8.26, и для самых малых n с очевидными методами сложности $O(n^2)$. Получите таким путем верхние оценки на те значения n, для которых задачи этой главы обладают алгоритмами, лучшими очевидных. Некоторая работа на этом пути уже проделана Кунгом [1973].

Замечания по литературе

Теорема 8.2, утверждающая, что обращение не сложнее умножения, взята у Кука [1966]. Любопытно, что существование полиномиального аналога алгоритма 8.1 не осознавали в течение нескольких лет. Моенк, Бородин [1972] описали алгоритм сложности $O_Б(n \log^2 n \log\log n)$ для деления, а вскоре затем алгоритм деления сложности $O_Б(n \log n \log\log n)$ независимо изложили несколько авторов, в том числе Зивекинг [1972].

Построение алгоритмов для модульной арифметики, а также для вычисления значений и интерполяции полиномов следует работе Моенка, Бородина [1972]. Хейндел, Хоровиц [1971] нашли алгоритм сложности $O_Б(n \log^2 n \log\log n)$, восстанавливающий числа по китайской теореме об остатках и использующий предварительную обработку данных. Бородин, Мунро [1971] описали алгоритм сложности $O(n^{1,91})$ для вычисления значений полинома в нескольких точках, а Хоровиц [1972] обобщил его на интерполяцию. Кунг [1973] построил алгоритм сложности $O_A(n \log^2 n)$ для вычисления значений полинома без применения быстрого (сложности $n \log n$) алгоритма деления. Единство восстановления по вычетам, интерполяции, вычисления значений полиномов и вычисления вычетов целых чисел установил Липсон [1971].

Алгоритм сложности $O_Б(n \log^2 n \log\log n)$ для нахождения НОД целых чисел принадлежит Шёнхаге [1971]. На полиномы и общие евклидовы области его перенес Моенк [1973].

Обзор классической техники для нахождения НОД дан Кнутом [1969]. Некоторую подборку материала о сложности арифметических операций над разреженными полиномами можно найти у Брауна [1971], Джентльмена [1972], Джентльмена, Джонсона [1973]. Дополнительные результаты о реализации арифметических операций над полиномами на вычислительных машинах приведены Холлом [1971], Брауном [1973] и Коллинзом [1973]. Алгоритм 8.8 со структурой данных в виде сортирующего дерева реализовал С. Джонсон на языке ALTRAN (см. [Браун, 1973]).

Упр. 8.19 и 8.20 принадлежат Карпу и Ульману. Упр. 8.21 о вычислении значений полинома и его производных можно найти в работах Вари [1974] и Ахо, Стейглица, Ульмана [1974]. Из последней работы взято также упр. 8.28. Упр. 8.27 приведено Блюстейном [1970] и Рабинером, Шэфером, Рейдером [1969]. Подробно арифметические операции над полиномами и целыми числами обсуждаются в книге Бородина, Мунро [1975].

9 АЛГОРИТМЫ ИДЕНТИФИКАЦИИ

Идентификация цепочек символов входит как составная часть во многие задачи, связанные с редактированием текстов, поиском данных и символьной обработкой. В типичной задаче идентификации цепочек задаются цепочка-текст x и множество цепочек-образов $\{y_1, y_2, \ldots\}$. Требуется найти либо одно, либо все вхождения цепочек-образов в x. Множество образов часто является регулярным множеством, заданным регулярным выражением. В настоящей главе мы обсудим несколько приемов решения такого рода задач идентификации цепочек.

Начнем с краткого обзора регулярных выражений и конечных автоматов. Затем изложим алгоритм, выявляющий в данной цепочке вхождение какой-нибудь цепочки из множества, заданного регулярным выражением. Время работы алгоритма равно по порядку произведению длин цепочки x и данного регулярного выражения. Потом приведем алгоритм линейной сложности, который распознает, является ли данная цепочка y подцепочкой данной цепочки x. Далее докажем сильный теоретический результат, состоящий в том, что любую проблему распознавания вхождения цепочек, разрешимую на двустороннем детерминированном магазинном автомате, можно решить на РАМ за линейное время. Это замечательный результат, поскольку магазинный автомат может потребовать для решения задачи квадратичное или даже экспоненциальное время. Наконец, введем понятие дерева позиций (позиционного дерева) и применим его к другим задачам идентификации цепочек, таким, как поиск в данной цепочке самого длинного повторения.

9.1. КОНЕЧНЫЕ АВТОМАТЫ И РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Многие задачи идентификации (распознавания образов) и их решения можно выразить в терминах регулярных выражений и конечных автоматов. Поэтому начнем с обзора соответствующего материала.

Определение. *Алфавитом* называется конечное множество символов. *Цепочка* в алфавите I — это конечная последовательность символов из I. *Пустая цепочка* ε — это цепочка, не содержащая символов. *Конкатенацией* цепочек x и y называется цепочка xy. Если цепочка имеет вид xyz, то x — ее *префикс (начало)*, y — *подцепочка*, а z — *суффикс (конец)*. *Длиной* $|x|$ цепочки x называется число различных вхождений символов в x. Например, длина цепочки aab равна 3; ε имеет длину 0.

Языком над алфавитом I называют произвольное множество цепочек в I. Пусть L_1 и L_2 — два языка. Язык L_1L_2, называемый *конкатенацией* L_1 и L_2, есть множество $\{xy | x \in L_1 \text{ и } y \in L_2\}$.

Пусть L — язык. Тогда положим $L^0=\{\varepsilon\}$ и $L^i=LL^{i-1}$ при $i \geqslant 1$. *Замыканием Клини (итерацией)* L^* языка L называют язык $L^*=\bigcup_{i=0}^{\infty} L^i$, а *позитивной итерацией* L^+ — язык $L^+=\bigcup_{i=1}^{\infty} L^i$.

Регулярные выражения и языки, которые они представляют *(регулярные множества)*, полезны во многих областях науки о вычислениях. В этой главе мы увидим, что они полезны для описания идентифицируемых образов.

Определение. Пусть I — алфавит. *Регулярные выражения над* I и языки, представляемые ими, рекурсивно определяются следующим образом.

1. $\varnothing$ является регулярным выражением и представляет пустое множество.
2. ε является регулярным выражением и представляет множество $\{\varepsilon\}$.
3. Для каждого $a \in I$ символ a является регулярным выражением и представляет множество $\{a\}$.
4. Если p и q — регулярные выражения, представляющие множества P и Q соответственно, то $(p+q)$, (pq) и (p^*) являются регулярными выражениями и представляют множества $P \cup Q$, PQ и P^* соответственно.

При записи регулярного выражения можно опустить многие скобки, если предположить, что операция * имеет более высокий приоритет, чем конкатенация и +, а конкатенация — более высокий приоритет, чем +. Например, $((0(1^*))+0)$ можно записать как 01^*+0. Кроме того, вместо pp^* будем для краткости писать p^+.

Пример 9.1.

1. 01 — регулярное выражение, представляющее множество $\{01\}$.
2. $(0+1)^*$ представляет $\{0, 1\}^*$.
3. $1(0+1)^*1+1$ представляет множество всех цепочек, начинающихся и кончающихся символом 1. □

Язык называется *регулярным*, если его можно представить регулярным выражением. Два регулярных выражения α и β называют *эквивалентными* (и пишут $\alpha=\beta$), если они представляют одно и то же множество. Например, $(0+1)^*=(0^*1^*)^*$.

Понятие детерминированного конечного автомата было введено в гл. 4. Его можно воспринимать как устройство, состоящее из блока управления, который всегда находится в одном из конечного числа состояний, и входной ленты, которая просматривается слева направо своей головкой (входной головкой). Детерминированный конечный автомат выполняет "шаги", определяемые текущим состоянием его блока управления и входным символом, обозреваемым входной головкой. Каждый шаг состоит из перехода в новое состояние и сдвига входной головки на одну клетку вправо. Оказывается, что язык представим регулярным выражением тогда и только тогда, когда он допускается некоторым конечным автоматом.

Важным обобщением рассматриваемого понятия является недетерминированный конечный автомат. Для каждого состояния и каждого входного символа недетерминированный автомат имеет нуль или более вариантов выбора следующего шага. Он может также выбирать, сдвигать ему входную головку при изменении состояния или нет.

Определение. *Недетерминированным конечным автоматом* (НКА) называется пятерка (S, I, δ, s_0, F), где

1) S — конечное *множество состояний* устройства управления;
2) I — *алфавит* входных символов;
3) δ — *функция переходов*, отображающая $S\times(I\cup\{\varepsilon\})$ в множество подмножеств множества S;
4) $s_0\in S$ — *начальное состояние* устройства управления;
5) $F\subseteq S$ — множество *заключительных (допускающих)* состояний.

Мгновенным описанием (МО) НКА M называется пара (s, w), где $s\in S$ — состояние блока управления и $w\in I^*$ — неиспользованная часть входной цепочки (т. е. символ, обозреваемый входной головкой, и все символы справа от него). *Начальным* МО автомата M называется МО, у которого первой компонентой служит начальное состояние, а второй — распознаваемая цепочка, т. е. (s_0, w) для некоторой цепочки $w\in I^*$. *Допускающие* МО — это МО вида (s, ε), где $s\in F$.

Представим шаги НКА бинарным отношением $\vdash$ на множестве мгновенных описаний. Если $\delta(s, a)$ содержит s', то пишут $(s, aw)\vdash(s', w)$ для всех $w\in I^*$. Заметим, что a — это или ε, или символ из I. Если $a=\varepsilon$, то состояние изменяется независимо от обозреваемого входного символа. Если $a\neq\varepsilon$, то символ a должен стоять в очередной клетке входной ленты, а входная головка сдвигается на одну клетку вправо.

Состояние \ Вход	a	b	ε
s_1	$\{s_1, s_2\}$	$\{s_1\}$	$\varnothing$
s_2	$\varnothing$	$\{s_3\}$	$\varnothing$
s_3	$\{s_4\}$	$\varnothing$	$\{s_1\}$
s_4	$\varnothing$	$\varnothing$	$\{s_2\}$

Рис. 9.1. Функция переходов δ.

Рефлексивное и транзитивное замыкание отношения $\vdash$ обозначается через $\vdash^*$. Говорят, что *цепочка w допускается автоматом M*, если $(s_0, w) \vdash^* (s, \varepsilon)$ для некоторого $s \in F$. Иными словами, входная цепочка w допускается автоматом M, если найдется последовательность из нуля или более шагов, переводящая M из начального МО (s_0, w) в допускающее МО (s, ε). Множество цепочек $L(M)$, допускаемых автоматом M, называют *языком, допускаемым автоматом M*.

Пример 9.2. Рассмотрим недетерминированный конечный автомат M, допускающий все те цепочки из символов a и b, которые оканчиваются цепочкой aba, т. е. $L(M)=(a+b)^*aba$. Пусть $M=(\{s_1, s_2, s_3, s_4\}, \{a, b\}, \delta, s_1, \{s_4\})$, где функция δ определена на рис. 9.1. (Здесь без ε-переходов можно обойтись.)

Пусть на вход автомата M подается цепочка $ababa$. Тогда M может сработать в соответствии с последовательностями МО, показанными на рис. 9.2. Так как $(s_1, ababa) \vdash^* (s_4, \varepsilon)$ и s_4 — заключительное состояние, то цепочка $ababa$ допускается автоматом M. □

С каждым НКА связан ориентированный граф, естественным образом представляющий функцию переходов этого автомата.

Определение. Пусть $M=(S, I, \delta, s_0, F)$ — НКА. *Графом переходов* (или *диаграммой*) автомата M называют ориентированный граф $G=(S, E)$ с помеченными ребрами. Множество ребер E и метки определяются следующим образом. Если $\delta(s, a)$ содержит s' для некоторого $a \in I \cup \{\varepsilon\}$, то ребро (s, s') принадлежит E. Меткой ребра

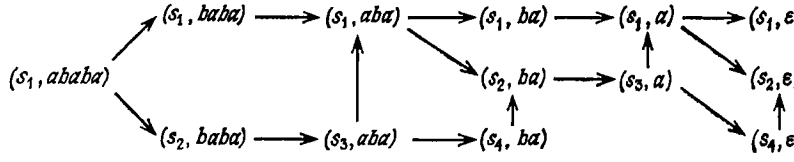


Рис. 9.2. Последовательность МО для входа $ababa$.

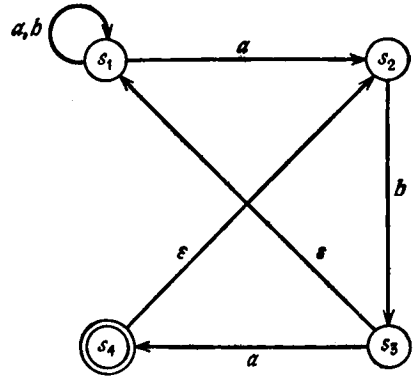


Рис. 9.3. Граф переходов для примера 9.2.

(s, s') служит множество тех $b \in I \cup \{\varepsilon\}$, для которых $\delta(s, b)$ содержит s'.

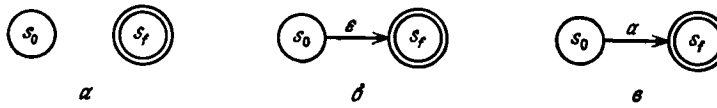
Пример 9.3. Граф переходов для НКА M из примера 9.2 изображен на рис. 9.3. Заключительное состояние обведено двойным кружком. □

Диаграммы НКА и задачи о путях на графах можно связать с помощью определенного замкнутого полукольца. Пусть I — алфавит; положим $S_I=(\mathcal{P}(I^*), \cup, \cdot, \varnothing, \{\varepsilon\})$. Из разд. 5.6 известно, что S_I — замкнутое полукольцо, в котором $\mathcal{P}(I^*)$ — множество всех языков над I, $\varnothing$ — единичный элемент относительно объединения $\cup$ и $\{\varepsilon\}$ — единичный элемент относительно конкатенации $\cdot$

Теорема 9.1. *Всякий язык, допускаемый недетерминированным конечным автоматом, регулярен.*

Доказательство. Пусть $M=(S, I, \delta, s_0, F)$ — НКА и $G=(S, E)$ — его диаграмма. Алгоритм 5.5 может найти для каждой пары узлов s и s' диаграммы язык $L_{ss'}$, являющийся множеством всех цепочек, помечающих пути из s в s'. Метка каждого ребра диаграммы является регулярным выражением. Более того, если множества C_{ij}^{k-1}, вычисленные алгоритмом 5.5, представимы регулярными выражениями, то в силу строки 5 этого алгоритма множества C_{ij}^{k} также представимы регулярными выражениями. Следовательно, каждый язык $L_{ss'}$ можно представить регулярным выражением, а потому он регулярен. Тогда $L(M)=\bigcup_{s \in F} L_{s_0 s}$ — регулярное множество, ибо по определению объединение регулярных множеств регулярно. □

Теорема, обратная к теореме 9.1, также верна. Иными словами, для данного регулярного выражения найдется НКА, допускающий язык, представленный этим регулярным выражением.



а *б* *в*

Рис. 9.4. Конечные автоматы, допускающие языки, представленные регулярными выражениями длины 1: (*а*) $\varnothing$, (*б*) $\{\varepsilon\}$, (*в*) $\{a\}$.

С точки зрения сложности вычислений наиболее важно, что можно найти НКА, у которого число состояний не больше удвоенной длины данного регулярного выражения и который из каждого своего состояния может перейти не более чем в два других.

Теорема 9.2. *Пусть* α *— регулярное выражение. Тогда найдется* НКА $M=(S, I, \delta, s_0, \{s_f\})$, *допускающий язык, представленный* α, *и обладающий такими свойствами:*

1) $\|S\| \leqslant 2|\alpha|$, *где* $|\alpha|$ *— длина выражения* α [1]),
2) *для каждого* $a \in I \cup \{\varepsilon\}$ *множество* $\delta(s_f, a)$ *пусто,*
3) *для каждого* $s \in S$ *сумма чисел* $\|\delta(s, a)\|$ *по всем* a *из* $I \cup \{\varepsilon\}$ *не превосходит* 2.

Доказательство. Доказательство проведем индукцией по длине выражения α. Рассмотрим базис, т. е. случай $|\alpha|=1$. Тогда выражение α должно быть одного из трех видов: (а) $\varnothing$, (б) ε или (в) a для некоторого $a \in I$. На рис. 9.4 изображены три автомата, имеющие по два состояния каждый, допускающие указанные языки и удовлетворяющие условиям теоремы.

Для шага индукции рассмотрим четыре возможных вида α: (а) $\beta+\gamma$, (б) $\beta\gamma$, (в) β^*, (г) (β), где β и γ — регулярные выражения. В случае (г) α и β представляют один и тот же язык, так что индукция очевидна. Рассмотрим другие случаи. Пусть M' и M'' — НКА, допускающие языки, представленные выражениями β и γ соответственно, причем их множества состояний не пересекаются. Пусть s_0' и s_0'' — начальные состояния этих автоматов, а s_f' и s_f'' — их заключительные состояния. Графы переходов для случаев (а), (б) и (в) показаны на рис. 9.5.

Пусть длины α, β и γ равны $|\alpha|$, $|\beta|$ и $|\gamma|$ соответственно, а n, n' и n'' — числа состояний автоматов M, M' и M''. В случае (а) $|\alpha| = |\beta|+|\gamma|+1$ и $n=n'+n''+2$. По предположению индукции $n' \leqslant 2|\beta|$ и $n'' \leqslant 2|\gamma|$, откуда $n \leqslant 2|\alpha|$. Кроме того, в случае (а) добавляются только два ребра, выходящие из s_0 (которые удовлетворяют тому ограничению, что из любого узла выходит не более двух ребер), да еще по одному ребру, выходящему из каждого узла s_f' и s_f''. Так

[1]) Длиной регулярного выражения α называется число вхождений символов в цепочку α. Например, $|\varepsilon^*|=2$. Можно было бы усилить свойство 1, игнорируя скобки при подсчете длины выражения α (например, регулярное выражение $(a^*b^*)^*$ при таком условии имело бы длину 5, а не 7).

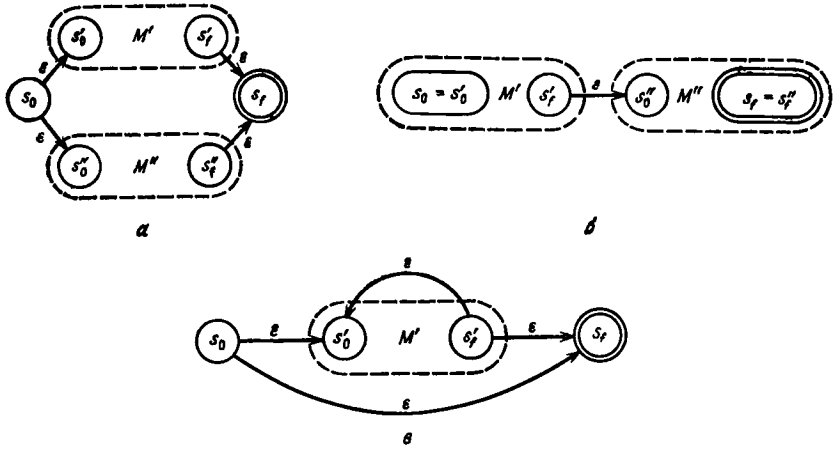


Рис. 9.5. Графы переходов автоматов, допускающие языки, представленные регулярными выражениями (*а*) $\beta+\gamma$, (*б*) $\beta\gamma$, (*в*) β^*.

как по предположению из каждого узла s_f' и s_f'' раньше не выходило ни одного ребра, то это же ограничение на число выходящих ребер выполнится и для s_f' и s_f''. Наконец, из s_f, очевидно, никакие ребра не выходят. Случай (б) проверяется аналогично [1]).

В случае (в) $|\alpha|=|\beta|+1$ и $n=n'+2$. Поскольку $n'\leqslant 2|\beta|$, то отсюда следует, что $n\leqslant 2|\alpha|$. Ограничение на число ребер, выходящих из любого узла, легко проверяется. □

Пример 9.4. Построим НКА для регулярного выражения ab^*+c, длина которого равна 5. НКА для a, b и c изображены на рис. 9.4, *в*. С помощью конструкции рис. 9.5,*в* построим автомат для b^*, как показано на рис. 9.6,*а*. Затем с помощью конструкции рис. 9.5,*б* построим автомат для ab^*, как показано на рис. 9.6,*в*. Наконец, с помощью конструкции рис. 9.5,*а* построим НКА для ab^*+c. Этот автомат, имеющий 10 состояний, изображен на рис. 9.6,*в*. □

Необходимо упомянуть один дополнительный результат. По данному НКА можно найти эквивалентную "детерминированную" машину. Однако детерминированный конечный автомат, эквивалентный данному НКА с n состояниями, может иметь вплоть до 2^n состояний. Поэтому переход к детерминированному автомату не всегда дает эффективный способ моделирования недетерминированного конечного автомата. Тем не менее детерминированные автоматы оказыва-

[1]) На самом деле ребро из s_f' в s_0'' не нужно. Вместо него можно было бы просто отождествить s_f' и s_0''. Точно так же на рис. 9.5,*а* можно было бы отождествить s_f' и s_f'' в одно заключительное состояние.

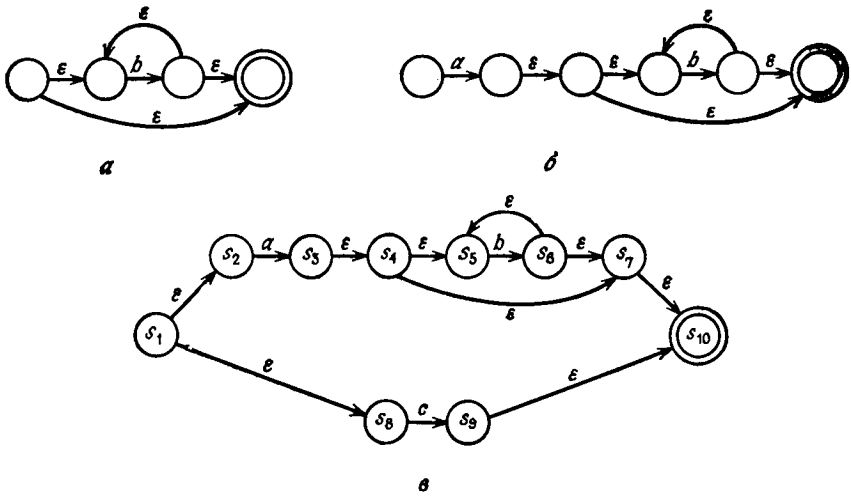


Рис. 9.6. Построение НКА для ab^*+c: (*а*) для b^*; (*б*) для ab^*; (*в*) для $ab^*+\varepsilon$.

ются полезными в распознавании образов, и сейчас мы напомним их определение.

Определение. *Детерминированным конечным автоматом* (ДКА) называется недетерминированный конечный автомат (S, I, δ, s_0, F), в котором

1) $\delta(s, \varepsilon)=\varnothing$ для всех $s \in S$,
2) $\|\delta(s, a)\| \leqslant 1$ для всех $s \in S$ и $a \in I$.

Теорема 9.3. *Если L — регулярное множество, оно допускается некоторым* ДКА.

Доказательство. По теореме 9.2 L допускается некоторым НКА $M=(S, I, \delta, s_0, \{s_t\})$. Превратим M в ДКА следующим образом. Сначала найдем такие пары состояний (s, t), что $(s, \varepsilon) \vdash_M^* (t, \varepsilon)$. Чтобы сделать это, построим ориентированный граф $G=(S, E)$, у которого $(s, t) \in E$ тогда и только тогда, когда $\delta(s, \varepsilon)$ содержит t. Затем вычислим рефлексивное и транзитивное замыкание $G'=(S, E')$ графа G. Мы утверждаем, что $(s, \varepsilon) \vdash_M^* (t, \varepsilon)$ тогда и только тогда, когда (s, t) принадлежит E'.

Теперь построим такой НКА $M'=(S', I, \delta', s_0, F)$, что $L(M')=L(M)$ и в M' нет ε-переходов.

1) $S'=\{s_0\} \cup \{t \mid \delta(s, a)$ содержит t для некоторого $s \in S$ и некоторого $a \in I\}$.
2) Для каждого $s \in S'$ и каждого $a \in I$

$$\delta'(s, a)=\{u \mid (s, t) \in E' \text{ и } \delta(t, a) \text{ содержит } u\}.$$

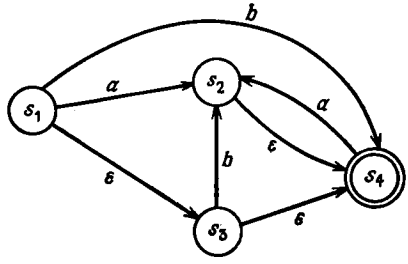


Рис. 9.7. НКА M из примера 9.5.

3) $F'=\{s|(s, f)\in E'$ и $f\in F\}$.

В качестве упражнения предлагаем доказать, что $L(M)=L(M')$. Разумеется, в M' нет переходов по ε.

Далее по M' построим ДКА M'', состояния которого образуют множество-степень для S'. Другими словами, $M''=(\mathcal{P}(S'), I, \delta'', \{s_0\}, F'')$, где

1) для каждого подмножества S множества S' и каждого $a\in I$ $\delta''(S, a)=\{t|\delta'(s, a)$ содержит t для некоторого $s\in S\}$,

2) $F''=\{S|S\cap F\neq\varnothing\}$.

В качестве упражнения предлагаем доказать индукцией по $|w|$, что $(\{s_0\}, w)\vdash_{M'}^{*}(S, \varepsilon)$ тогда и только тогда, когда $S=\{t|(s_0, w)\vdash_{M'}^{*}(t, \varepsilon)\}$. Таким образом, $L(M)=L(M')=L(M'')$. □

Пример 9.5. Рассмотрим НКА M на рис. 9.7. Из начального состояния s_1 можно достичь s_3 и заключительное состояние s_4 по путям, помеченным символом ε. Поэтому для вычисления рефлексивного и транзитивного замыкания G' ориентированного графа G, о котором шла речь в доказательстве теоремы 9.3, надо добавить ребро (s_1, s_4). Весь граф G' изображен на рис. 9.8. По M и G' построим НКА M' (рис. 9.9). Так как в узел s_4 входят ребра из всех узлов графа G', объявляем все состояния в M' заключительными. Так как единст-

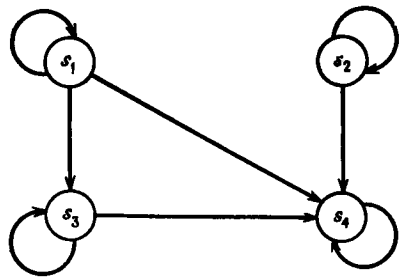


Рис. 9.8. Граф G'.

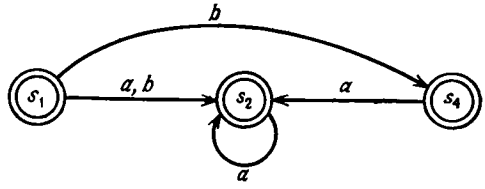


Рис. 9.9. НКА M'.

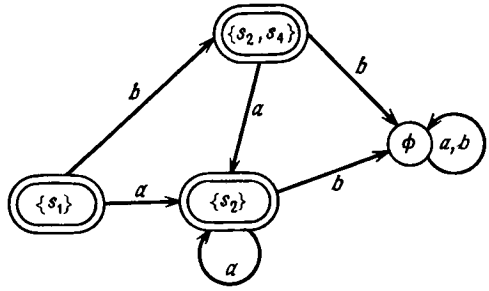


Рис. 9.10. ДКА M''.

венное ребро, входящее в узел s_3 в диаграмме для M, помечено символом ε, то s_3 не входит в M'.

При построении ДКА M'' по автомату M' образуется восемь состояний. Но только четырех из них можно достичь из начального состояния, так что остальные четыре можно выбросить. ДКА M'' изображен на рис. 9.10. □

9.2. РАСПОЗНАВАНИЕ ОБРАЗОВ, ЗАДАВАЕМЫХ РЕГУЛЯРНЫМИ ВЫРАЖЕНИЯМИ

Изучим задачу распознавания образов, в которой дана *цепочка-текст* $x=a_1a_2 \ldots a_n$ и регулярное выражение α, называемое *образом*. Мы хотим найти такой наименьший индекс j, что для некоторого i подцепочка $a_ia_{i+1} \ldots a_j$ цепочки x принадлежит языку, представленному выражением α.

Вопросы такого рода часто возникают при редактировании текстов. Многие программы для редактирования текстов разрешают пользователю задавать типы замен в цепочке-тексте. Например, пользователь может сказать, что он хочет заменить слово y каким-то другим словом в куске текста x. Чтобы выполнить такую команду, программа редактирования текста должна суметь найти вхождение слова y в x. Некоторые искусные редактирующие программы разрешают пользователю в качестве множества заменяемых цепочек указывать регулярное множество. Например, пользователь может

сказать: "Заменить $[I^*]$ в x пустой цепочкой", имея в виду, что в x следует стереть пару квадратных скобок и символы между ними.

Поставленную выше задачу можно переформулировать, заменив данное регулярное выражение α выражением $\beta = I^*\alpha$, где I — алфавит цепочки-текста. Можно найти первое вхождение цепочки из $L(\alpha)$ в $x = a_1a_2 \ldots a_n$, обнаружив кратчайший префикс цепочки x, принадлежащий языку выражения β. Эту задачу можно решить, сначала построив НКА M для распознавания множества, представленного выражением β, а затем применив алгоритм 9.1 (см. ниже) для определения последовательности множеств состояний S_i, в которые может перейти НКА M после прочтения цепочки $a_1a_2 \ldots a_i$ при $i = 1, 2, \ldots, n$. Как только в S_j попадает заключительное состояние, мы можем сказать, что у цепочки $a_1a_2 \ldots a_j$ есть такой суффикс $a_ia_{i+1} \ldots a_j$, что $a_ia_{i+1} \ldots a_j$ принадлежит языку, представленному выражением α, при некотором $1 \leqslant i \leqslant j$. Техника нахождения i, т. е. левого конца образа, обсуждается в упр. 9.6—9.8.

Один из способов моделирования поведения НКА M на цепочке-тексте x — превратить M в детерминированный конечный автомат, как в теореме 9.3. Но такой путь может оказаться слишком дорогим, поскольку от регулярного выражения β можно перейти к НКА с $2|\beta|$ состояниями, а затем к ДКА с почти $2^{2|\beta|}$ состояниями. Уже само построение ДКА может вызвать непреодолимые трудности.

Другой способ моделирования поведения НКА M на цепочке x — сначала исключить ε-переходы в M и тем самым построить НКА M' без ε, как это было сделано в теореме 9.3. Затем моделировать НКА M' на входной цепочке $x = a_1a_2 \ldots a_n$, вычислив для каждого i, $1 \leqslant i \leqslant n$, множество состояний S_i, в которые мог бы попасть M' после прочтения $a_1a_2 \ldots a_i$. На самом деле каждое множество S_i — это то состояние, в которое пришел бы ДКА M'', построенный в доказательстве теоремы 9.3, после прочтения $a_1a_2 \ldots a_i$.

Применяя эту технику, можно не строить автомат M'', а вычислить только те его состояния, которые появляются при обработке цепочки x. Чтобы объяснить, как найти множество S_i, надо показать, как построить S_i по S_{i-1}. Легко видеть, что

$$S_i = \bigcup_{s \in S_{i-1}} \delta'(s, a_i),$$

где δ' — функция переходов автомата M'. Тогда S_i — объединение вплоть до $2|\beta|$ множеств, каждое из которых содержит не больше $2|\beta|$ членов. Так как при объединении надо исключить повторяющиеся члены (иначе представление множеств может стать громоздким), то очевидное моделирование автомата M' занимает $O(|\beta|^2)$ шагов на входной символ, т. е. в общей сложности $O(n|\beta|^2)$ шагов.

Как это ни удивительно, но во многих практических ситуациях гораздо эффективнее не исключать ε-переходы, а работать прямо с НКА M, построенным по теореме 9.2 из регулярного выражения β.

```
1.  for i ← 0 until n do
      begin
2.      if i = 0 then S_i ← {s_0}
3.      else S_i ← ⋃_{s ∈ S_{i-1}} δ(s, a_i);
        comment S_i еще не достигло своего конечного значения. Сейчас оно соответствует множеству T_i, упомянутому выше;
4.      пометить каждое состояние t ∈ S_i как "рассмотренное";
5.      пометить каждое состояние t ∈ S − S_i как "нерассмотренное";
6.      ОЧЕРЕДЬ ← S_i;
7.      while список ОЧЕРЕДЬ не пуст do
          begin
8.          найти и удалить первый элемент t, входящий в ОЧЕРЕДЬ;
9.          for u ∈ δ(t, ε) do
10.           if u — "нерассмотренное" состояние then
                begin
11.               пометить u как "рассмотренное";
12.               добавить u в ОЧЕРЕДЬ и в S_i
                end
          end
      end
```

Рис. 9.11. Моделирование недетерминированного конечного автомата.

Решающее свойство в теореме 9.2 заключалось в том, что на графе переходов из каждого состояния автомата M выходит не более двух ребер. Это свойство позволяет показать, что на входной цепочке $x=a_1a_2\ldots a_n$ алгоритм 9.1 (см. ниже) тратит на моделирование НКА, построенного из β, $O(n|\beta|)$ шагов.

Алгоритм 9.1. Моделирование недетерминированного конечного автомата

Вход. НКА $M=(S, I, \delta, s_0, F)$ и цепочка $x=a_1a_2\ldots a_n$ из I^*.

Выход. Последовательность $S_0, S_1, S_2, \ldots, S_n$, где

$$S_i=\{s \mid (s_0, a_1a_2\ldots a_i) \vdash^* (s, \varepsilon)\}, \quad 0 \leqslant i \leqslant n.$$

Метод. Чтобы получить S_i по S_{i-1}, сначала найдем множество состояний $T_i=\{t \mid \delta(s, a_i)$ содержит t для некоторого $s \in S_{i-1}\}$. Затем

вычислим "замыкание" множества T_i, добавив к T_i все такие состояния u, что $\delta(t, \varepsilon)$ содержит u для некоторого t, ранее оказавшегося в T_i. Это замыкание (оно и будет множеством S_i) строится с помощью очереди состояний $t \in T_i$, для которых множество $\delta(t, \varepsilon)$ еще не рассматривалось. Алгоритм приведен на рис. 9.11. □

Пример 9.6. Пусть M — НКА, изображенный на рис. 9.6, *в*. Допустим, что на вход подана цепочка $x=ab$. Тогда $S_0=\{s_1\}$ в строке 2. В строках 9—12 рассмотрение состояния s_1 приводит к добавлению s_2 и s_8 в ОЧЕРЕДЬ и в S_0. Рассмотрение s_2 и s_8 не добавляет ничего, поэтому $S_0=\{s_1, s_2, s_8\}$. Затем в строке 3 $S_1=\{s_3\}$. Рассмотрение s_3 добавляет s_4 в S_1, а рассмотрение s_4 добавляет s_5 и s_7. Рассмотрение s_5 не добавляет ничего, а рассмотрение s_7 добавляет s_{10}. Таким образом, $S_1=\{s_3, s_4, s_5, s_7, s_{10}\}$. Затем в строке 3 $S_2=\{s_6\}$. Рассмотрение s_6 добавляет s_5 и s_7 в S_2. Рассмотрение s_7 добавляет s_{10}. Итак, $S_2=\{s_5, s_6, s_7, s_{10}\}$. □

Теорема 9.4. *Алгоритм* 9.1 *правильно вычисляет последовательность* $S_0, S_1, \ldots, S_n$, *где* $S_i=\{s \mid (s_0, a_1 a_2 \ldots a_i) \vdash^* (s, \varepsilon)\}$.

Доказательство. Простое упражнение на применение индукции, которое мы оставляем читателю. □

Теорема 9.5. *Пусть в диаграмме автомата* M *с* m *состояниями из каждого узла выходит не более* e *ребер. Тогда на входной цепочке длины* n *алгоритм* 9.1 *тратит* $O(emn)$ *шагов.*

Доказательство. Исследуем построение множества S_i для одного конкретного значения i. Строки 8—12 на рис. 9.11 занимают $O(e)$ шагов. Поскольку для данного i никакое состояние не попадает в ОЧЕРЕДЬ дважды, то цикл в строках 7—12 требует $O(em)$ времени. Поэтому легко показать, что тело основного цикла, т. е. строки 2—12, занимает $O(em)$ времени. Таким образом, весь алгоритм занимает $O(emn)$ времени. □

Отсюда вытекает важный результат, связывающий распознавание вхождения элементов регулярного множества с алгоритмом 9.1.

Следствие. *Если* β — *произвольное регулярное выражение и* $x=a_1a_2 \ldots a_n$ — *цепочка длины* n, *то найдется* НКА M, *допускающий язык, представленный выражением* β, *причем алгоритм* 9.1 *тратит на построение последовательности* $S_0, S_1, \ldots, S_n$, *где* $S_i=\{s \mid (s_0, a_1a_2 \ldots a_i) \vdash^* (s, \varepsilon)\}$ *для* $0 \leqslant i \leqslant n$, *время* $O(n|\beta|)$.

Доказательство. По теореме 9.2 можно построить автомат M, у которого не более $2|\beta|$ состояний и из каждого из них выходит не более двух ребер. Поэтому e в теореме 9.5 не превосходит 2. □

Исходя из алгоритма 9.1, можно построить различные алгоритмы распознавания образов. Например, пусть даны регулярное выражение α и цепочка-текст $x=a_1a_2\ldots a_n$ и надо найти наименьшее число k, для которого существует такое $j<k$, что $a_ja_{j+1}\ldots a_k$ принадлежит множеству, представленному выражением α. С помощью теоремы 9.2 можно построить по α НКА M, допускающий язык $I^*\alpha$. Чтобы найти наименьшее число k, для которого $a_1a_2\ldots a_k$ принадлежит $L(M)$, можно после блока, состоящего из строк 2—12 на рис. 9.11, вставить тест, проверяющий, содержит ли множество S_i состояние из F. В силу теоремы 9.2 можно сделать F одноэлементным, так что такой текст не займет много времени; достаточно $O(m)$ шагов, где m — число состояний в M. Если S_i содержит состояние из F, то прерываем основной цикл, поскольку обнаружено, что $a_1a_2\ldots a_i$ — кратчайший префикс цепочки x, который входит в $L(M)$.

Алгоритм 9.1 можно модифицировать и так, чтобы он для каждого упомянутого k находил такое наибольшее $j<k$ (или наименьшее j), что $a_ja_{j+1}\ldots a_k$ входит в множество, представленное выражением α. Это делается приписыванием целого числа каждому состоянию из множеств S_i. Число, приписанное состоянию $s\in S_k$, указывает такое наибольшее (или наименьшее) j, что $(s_0, a_ja_{j+1}\ldots a_k)\vdash^*(s,\varepsilon)$. Детали приписывания этих целых чисел с помощью алгоритма 9.1 оставляем в качестве упражнения.

9.3. РАСПОЗНАВАНИЕ ПОДЦЕПОЧЕК

Важным частным случаем общей проблемы, описанной в предыдущем разделе, является случай, когда регулярное выражение α имеет вид $y_1+y_2+\ldots+y_k$, где каждый член y_i — цепочка в алфавите I. Из следствия теоремы 9.5 вытекает, что первое вхождение цепочки-образа y_i в цепочку-текст $x=a_1a_2\ldots a_n$ можно найти за $O(ln)$ шагов, где l — сумма длин цепочек y_i. Однако возможно решение и сложности $O(l+n)$. Сначала рассмотрим случай, когда дана только одна цепочка-образ $y=b_1b_2\ldots b_l$, где каждый символ b_i принадлежит I.

По цепочке y построим детерминированную машину M_y для идентификации цепочек, которая распознает кратчайшую цепочку из I^*y, входящую в x. Чтобы построить M_y, построим сначала *скелетный* ДКА с $l+1$ состояниями $0, 1, \ldots, l$, который переходит из состояния $i-1$ в состояние i на входном символе b_i, как показано на рис. 9.12. Состояние 0 имеет также переход в себя при всех $b\neq b_1$. Состояние i можно считать указателем на i-ю позицию цепочки-образа y.

Машина идентификации цепочек M_y работает как детерминированный конечный автомат с той лишь разницей, что, просматривая один входной символ, она может сделать несколько переходов (изме-

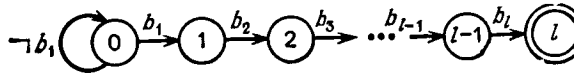


Рис. 9.12. Скелетная машина.

нений состояния). M_y имеет то же множество состояний, что и скелетная машина. Поэтому состояние j машины M_y соответствует префиксу $b_1b_2\ldots b_j$ цепочки-образа y.

M_y начинает работу в состоянии 0 и с входным указателем на a_1, т. е. на первый символ цепочки-текста $x=a_1a_2\ldots a_n$. Если $a_1=b_1$, то M_y переходит в состояние 1 и сдвигает входной указатель на вторую позицию цепочки-текста. Если $a_1\neq b_1$, то M_y остается в состоянии 0 и сдвигает входной указатель на вторую позицию.

Допустим, что M_y после прочтения $a_1a_2\ldots a_k$ находится в состоянии j. Это означает, что последние j символов цепочки $a_1a_2\ldots a_k$ совпадают с $b_1b_2\ldots b_j$, а последние m символов в $a_1a_2\ldots a_k$ не являются префиксом цепочки $b_1b_2\ldots b_l$ для $m>j$. Если a_{k+1}, т. е. следующий входной символ, совпадает с b_{j+1}, то M_y переходит в состояние $j+1$ и сдвигает входной указатель на a_{k+2}. Если $a_{k+1}\neq b_{j+1}$, то M_y переходит в наибольшее состояние i, для которого $b_1b_2\ldots b_i$ — суффикс цепочки $a_1a_2\ldots a_{k+1}$.

Чтобы облегчить нахождение состояния i, с машиной M_y связывается функция f, принимающая целочисленные значения. Она называется *функцией отказов* и задается так: $f(j)$ — наибольшее число $s<j$, для которого $b_1b_2\ldots b_s$ — суффикс цепочки $b_1b_2\ldots b_j$, т. е. $b_1b_2\ldots b_s=b_{j-s+1}b_{j-s+2}\ldots b_j$. Если такого $s\geqslant 1$ нет, то $f(j)=0$.

Пример 9.7. Пусть $y=aabbaab$. Функция f принимает значения

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

Например, $f(6)=2$, ибо aa — самый длинный собственный префикс цепочки $aabbaa$, который является ее суффиксом. □

Алгоритм вычисления функции отказов мы изложим несколько позже. Сейчас, чтобы увидеть, как функция отказов используется машиной M_y, определим функцию $f^{(m)}(j)$:

1) $f^{(1)}(j)=f(j)$,

2) $f^{(m)}(j)=f(f^{(m-1)}(j))$ для $m>1$.

Иными словами, $f^{(m)}(j)$ — эта та же функция f, примененная m раз к j. (В примере 9.7 $f^2(6)=1$.)

Снова предположим, что M_y после прочтения $a_1a_2\ldots a_k$ находится в состоянии j и $a_{k+1}\neq b_{j+1}$. В этот момент M_y итерирует

применение функции отказов к j до тех пор, пока не обнаруживается наименьшее значение m, для которого либо

1) $f^{(m)}(j)=u$ и $a_{k+1}=b_{u+1}$, либо
2) $f^{(m)}(j)=0$ и $a_{k+1}\neq b_1$.

Таким образом, M_y движется назад через состояния $f^{(1)}(j)$, $f^{(2)}(j)$, . . . до тех пор, пока не встретит такое m, что условие 1 или 2 будет выполнено для $f^{(m)}(j)$, но не для $f^{(m-1)}(j)$. Если выполнилось условие 2, то M_y переходит в состояние 0. В любом случае входной указатель сдвигается на a_{k+2}.

В случае 1 легко проверить, что поскольку $b_1b_2. . .b_j$ — самый длинный префикс цепочки y, который являлся суффиксом цепочки $a_1a_2. . .a_k$, то $b_1b_2. . .b_{f^{(m)}(j)+1}$ — самый длинный префикс цепочки y, который является суффиксом цепочки $a_1a_2. . .a_{k+1}$. В случае 2 никакой префикс цепочки y не является суффиксом цепочки $a_1a_2. . . . . .a_{k+1}$.

Затем M_y обрабатывает входной символ a_{k+2} и продолжает работать по такой схеме до момента, когда либо попадет в заключительное состояние l (и тогда l последних просмотренных входных символов образуют вхождение цепочки $y=b_1b_2. . .b_l$), либо обработает последний входной символ из x и не попадет в состояние l (и тогда y не является подцепочкой цепочки x).

Пример 9.8. Пусть $y=aabbaab$. Машина идентификации цепочек M_y изображена на рис. 9.13. Штриховые стрелки указывают значения функции отказов для всех состояний. На входе $x=abaabaabbaab$ машина M_y пройдет через такую последовательность состояний:

Вход:		*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*b*	*a*	*a*	*b*	
Состояние:	0	1	0	1	2	3	1	2	3	4	5	6	7	
		0				0								

Например, вначале машина M_y находится в состоянии 0. Прочитав первый символ из x, она переходит в состояние 1. Так как из состояния 1 нет перехода по второму входному символу b, то M_y переходит в состояние 0, т. е. в значение функции отказов для состояния 1, при этом входной указатель не сдвигается. Так как первый символ в y отличен от b, то выполнено условие 2, и M_y остается в состоянии 0 и сдвигает входной указатель на позицию 3.

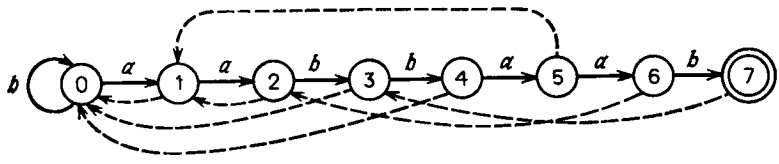


Рис. 9.13. Машина идентификации цепочек.

После прочтения двенадцатого входного символа M_y попадает в заключительное состояние 7. Таким образом, дойдя до позиции 12 в цепочке x, машина M_y нашла вхождение цепочки-образа y. □

Функцию f можно итеративно вычислить почти по той же схеме, по какой работает M_y. По определению $f(1)=0$. Допустим, что вычислены $f(1), f(2), \ldots, f(j)$. Пусть $f(j)=i$. Чтобы вычислить $f(j+1)$, исследуем b_{j+1} и b_{i+1}. Если $b_{j+1}=b_{i+1}$, то $f(j+1)=f(j)+1$, поскольку

$$b_1 b_2 \ldots b_i b_{i+1} = b_{j-i+1} b_{j-i+2} \ldots b_j b_{j+1}.$$

Если $b_{j+1} \neq b_{i+1}$, то находим наименьшее m, для которого либо

1) $f^{(m)}(j)=u$ и $b_{j+1}=b_{u+1}$, либо
2) $f^{(m)}(j)=0$ и $b_{j+1} \neq b_1$.

В случае 1 полагаем $f(j+1)=u+1$. В случае 2 полагаем $f(j+1)=0$. Детали даны в следующем алгоритме.

Алгоритм 9.2. Вычисление функции отказов

Вход. Цепочка-образ $y=b_1 b_2 \ldots b_l$, $l \geqslant 1$.
Выход. Функция отказов f для y.
Метод. Выполнение программы на рис. 9.14. □

Пример 9.9. Рассмотрим поведение алгоритма 9.2 на входе $y=aabbaab$. Формирование начальных данных дает $f(1)=0$. Поскольку $b_2=b_1$, то $f(2)=1$. Но $b_3 \neq b_2$ и $b_3 \neq b_1$, так что $f(3)=0$. Продолжая в том же духе, получаем значения f, указанные в примере 9.7. □

Докажем, что алгоритм 9.2 правильно вычисляет f за время $O(|y|)$. Сначала докажем корректность алгоритма.

```
    begin
1.    f(1) ← 0;
2.    for j ← 2 until l do
        begin
3.        i ← f(j − 1);
4.        while b_j ≠ b_{i+1} и i > 0 do i ← f(i);
5.        if b_j ≠ b_{i+1} и i = 0 then f(j) ← 0
6.        else f(j) ← i + 1
        end
    end
```

Рис. 9.14. Вычисление функции отказов.

Теорема 9.6. *Алгоритм* 9.2 *вычисляет* f.

Доказательство. Докажем индукцией по j, что $f(j)$ — такое наибольшее целое $i<j$, что $b_1b_2\ldots b_i=b_{j-i+1}b_{j-i+2}\ldots b_j$. Если такого i нет, то $f(j)=0$.

По определению $f(1)=0$. Допустим, что предположение индукции верно для всех $k<j$. При вычислении $f(j)$ алгоритм 9.2, выполняя строку 4, сравнивает b_j с $b_{f(j-1)+1}$.

Случай 1. Пусть $b_j=b_{f(j-1)+1}$. Поскольку $f(j-1)$ — это такое наибольшее i, что $b_1\ldots b_i=b_{j-i}\ldots b_{j-1}$, равенство $f(j)=i+1$ выполняется. Таким образом, в строках 5 и 6 $f(j)$ вычисляется правильно.

Случай 2. Пусть $b_j\neq b_{f(j-1)+1}$. Тогда надо найти наибольшее значение i, для которого $b_1\ldots b_i=b_{j-i}\ldots b_{j-1}$ и $b_{i+1}=b_j$, если такое i существует. Если такого i нет, то очевидно, что $f(j)=0$, и $f(j)$ правильно вычисляется в строке 5. Пусть i_1, i_2, . . .— наибольшее, второе по величине и т. д. значения i, для которых

$$b_1b_2\ldots b_i=b_{j-i}\ldots b_{j-1}.$$

С помощью простой индукции убеждаемся, что $i_1=f(j-1)$, $i_2=f(i_1)=f^{(2)}(j-1)$,. . . , $i_k=f(i_{k-1})=f^{(k)}(j-1)$, поскольку i_{k-1} — это $(k-1)$-е по величине значение i, для которого $b_1\ldots b_i=b_{j-i}\ldots b_{j-1}$, а i_k — наибольшее значение $i<i_{k-1}$, для которого $b_1\ldots b_i=b_{i_{k-1}-i+1}\ldots b_{i_{k-1}}=b_{j-1}\ldots b_{j-1}$. Строка 4 просматривает i_1, i_2, . . . по очереди, пока не найдет такое i, что $b_1\ldots b_i=b_{j-i}\ldots b_{j-1}$ и $b_{i+1}=b_j$, если такое i существует. По окончании выполнения **while**-оператора будет $i=i_m$, если такое i_m существует, и, значит, $f(j)$ правильно вычисляется в строке 5.

Таким образом, $f(j)$ правильно вычисляется для всех j. □

Теорема 9.7. *Алгоритм* 9.2 *вычисляет* f *за* $O(l)$ *шагов.*

Доказательство. Строки 3 и 5 имеют фиксированную сложность. Сложность **while**-оператора пропорциональна числу уменьшений значения i оператором $i\leftarrow f(i)$, который стоит после **do** в строке 4. Единственный способ увеличить i — это присвоить $f(j)=i+1$ в строке 6, затем увеличить j на 1 в строке 2 и положить $i=f(j-1)$ в строке 3. Поскольку вначале $i=0$, а строка 6 выполняется не более $l-1$ раз, заключаем, что **while**-оператор в строке 4 не может выполняться более l раз. Поэтому строка 4 требует $O(l)$ времени. Остальная часть алгоритма, очевидно, имеет сложность $O(l)$, и потому весь алгоритм тратит $O(l)$ времени. □

С помощью тех же рассуждений, что и в теореме 9.6, можно доказать, что после прочтения слова $a_1a_2\ldots a_k$ машина идентификации образов M_y будет находиться в состоянии i тогда и только тогда, когда $b_1b_2\ldots b_i$ — самый длинный префикс цепочки y, который является суффиксом цепочки $a_1a_2\ldots a_k$. Поэтому машина M_y правиль-

но находит самое левое вхождение цепочки y в цепочку-текст $x=a_1a_2\ldots a_n$.

С помощью тех же рассуждений, что и в теореме 9.7, можно доказать, что при обработке входной цепочки x машина M_y изменит свое состояние не более $2|x|$ раз. Поэтому можно узнать, является ли y подцепочкой цепочки x, проследив изменения состояния машины M_y на входе x [1]). Для этого надо лишь знать значение функции отказов на y. По теореме 9.7 эти значения функции f можно найти за время $O(|y|)$. Следовательно, узнать, является ли y подцепочкой цепочки x, можно за время $O(|x|+|y|)$, не зависящее от размера алфавита. Если же алфавит цепочки-образа мал, а цепочка-текст значительно длиннее образа, то можно смоделировать некоторый ДКА, допускающий язык I^*y. Этот ДКА в точности один раз меняет состояние на каждом входном символе.

Алгоритм 9.3. Построение ДКА для I^*y

Вход. Цепочка-образ $y=b_1b_2\ldots b_l$ в алфавите I. Для удобства вводим новый символ $b_{l+1}\notin I$.

Выход. ДКА M, для которого $L(M)=I^*y$.

Метод.

1. Алгоритмом 9.2 строим функцию отказов f для y.

2. Пусть $M=(S, I, \delta, 0, \{l\})$, где $S=\{0, 1, \ldots, l\}$, а δ определяется так:

begin
  for $j=1$ **until** l **do** $\delta(j-1, b_j)\leftarrow j$;
  for $b\in I$, $b\neq b_1$ **do** $\delta(0, b)\leftarrow 0$;
  for $j=1$ **until** l **do**
    for $b\in I$, $b\neq b_{j+1}$ **do** $\delta(j, b)\leftarrow\delta(f(j), b)$
end □

Теорема 9.8. *Алгоритм* 9.3 *строит такой* ДКА M, *что*

$$(0, a_1a_2\ldots a_k)\vdash^*(j, \varepsilon),$$

тогда и только тогда, когда $b_1b_2\ldots b_j$ *— суффикс цепочки* $a_1a_2\ldots a_k$, *но* $b_1b_2\ldots b_i$ *при* $i>j$ *не является суффиксом для* $a_1a_2\ldots a_k$.

Доказательство. Доказательство проводится индукцией по k с помощью тех же рассуждений, что и в теореме 9.6. Оставляем его читателю. □

Пример 9.10. ДКА M для $y=aabbaab$, построенный алгоритмом 9.3, изображен на рис. 9.15.

На входе $x=abaabaabbaab$ автомат M делает такие переходы:

[1]) Напомним, что состояние машины M_y есть на самом деле указатель позиции в цепочке-образе y. Поэтому изменение состояния машины M_y можно реализовать, непосредственно перемещая указатель по y.

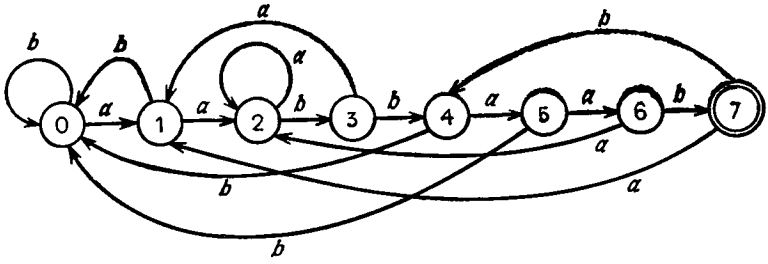


Рис. 9.15. Детерминированный конечный автомат, допускающий $(a+b)^*aabbaab$.

Вход:		a	b	a	a	b	a	a	b	b	a	a	b
Состояние:	0	1	0	1	2	3	1	2	3	4	5	6	7

Единственное отличие его от M_y состоит в том, что M заранее вычисляет состояние, в которое следует переходить в случае несовпадения. Поэтому он делает в точности один переход на каждом входном символе. □

Основные результаты раздела суммируем в следующей теореме.

Теорема 9.9. *За время $O(|x|+|y|)$ можно выяснить, является ли y подцепочкой цепочки x.*

Теперь разберем случай, когда даны несколько цепочек-образов $y_1, y_2, \ldots, y_k$. Наша задача — распознать, входит ли одна из цепочек y_i в данную цепочку $x=a_1a_2\ldots a_n$. К этой задаче можно также применить методы данного раздела. Сначала построим скелетную машину для $y_1, y_2, \ldots, y_k$. Она будет деревом. На этом дереве вычислим функцию отказов за время, пропорциональное $l=|y_1|+|y_2|+\ldots+|y_k|$. Потом тем же способом, что и раньше, построим машину идентификации образов. Тогда за $O(l+n)$ шагов мы узнаем, является ли какая-нибудь цепочка y_i подцепочкой цепочки x. Детали оставляем в качестве упражнения.

9.4. ДВУСТОРОННИЙ ДЕТЕРМИНИРОВАННЫЙ МАГАЗИННЫЙ АВТОМАТ

Как только мы заподозрили, что существует алгоритм сложности $O(|x|+|y|)$, распознающий, входит ли y в x, его уже нетрудно построить. Но что может заставить нас подозревать о существовании такого алгоритма? Одна из возможных причин возникает при изучении двусторонних детерминированных магазинных автоматов (2ДМА для краткости).

2ДМА представляет собой специальный тип машины Тьюринга, допускающей язык. Многие задачи распознавания образов можно переформулировать в терминах задач распознавания языков. Например, пусть L — язык $\{xcy | x, y \in I^*, c \notin I$ и y — подцепочка це-

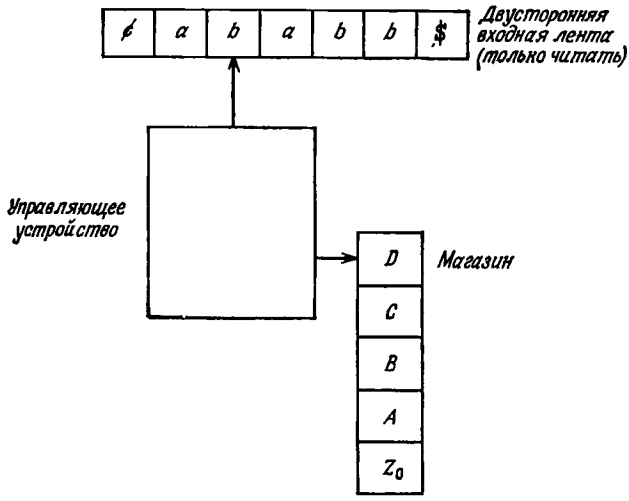


Рис. 9.16. Двусторонний детерминированный магазинный автомат.

почки $x\}$. Тогда распознавание того, входит ли y в x, эквивалентно распознаванию принадлежности цепочки xcy языку L. В этом разделе мы покажем, что существует 2ДМА, способный распознать L. Хотя этот 2ДМА может затратить $O(n^2)$ времени, но известна мощная техника моделирования, позволяющая промоделировать поведение данного 2ДМА на входной цепочке длины n на РАМ, которая затратит на это $O(n)$ шагов. В настоящем разделе мы подробно изучим эту технику моделирования.

2ДМА можно рассматривать как двухленточную машину Тьюринга, одна из лент которой используется в качестве магазинной памяти (рис. 9.16). 2ДМА имеет входную ленту, которую можно только читать и у которой самая левая клетка помечена концевым маркером ¢, а самая правая — концевым маркером \$. Распознаваемая входная цепочка располагается между этими двумя концевыми маркерами по одному символу в клетке. Входные символы берутся из входного алфавита I, который по предположению не содержит ¢ и \$. Входная головка (на входной ленте) может за один шаг прочесть один символ и сдвинуться на одну клетку влево, вправо или остаться на месте. Мы предполагаем, что головка не может сойти ни с одного конца входной ленты; наличие концевых маркеров ¢ и \$ позволяет так писать функцию переходов, чтобы она никогда не сдвигала входную головку влево от ¢ и вправо от \$.

Магазин — это стек, содержащий символы из магазинного алфавита T. Дно (нижняя клетка) магазина содержит специальный символ Z_0, отмечающий основание стека. Мы предполагаем, что Z_0 не принадлежит T.

Управляющее устройство всегда находится в одном из состояний, принадлежащих конечному множеству S. Работа машины определяется функцией переходов δ, которая для всякого $s \in S$, $a \in I \cup \{¢, \$\}$ и $A \in T \cup \{Z_0\}$ указывает действие машины в случае, когда управляющее устройство находится в состоянии s, входная головка обозревает символ a и в вершине магазина расположен символ A. Возможны три типа действий машины:

$$\delta(s, a, A) = \begin{cases} (s', d, \textbf{push } B), & \text{где } B \neq Z_0, \\ (s', d), & \\ (s', d, \textbf{pop}), & \text{если } A \neq Z_0. \end{cases}$$

При любом из этих действий машина переходит в состояние s' и сдвигает входную головку в направлении d (здесь $d=-1$, $+1$ или 0, что означает сдвиг на одну клетку влево, вправо или отсутствие сдвигов); **push** B означает добавление символа B в вершину магазина; **pop** — удаление самого верхнего символа из магазина.

Определение. Формально 2ДМА определяется как семерка

$$P = (S, I, T, \delta, s_0, Z_0, s_f),$$

где

1) S — конечное множество *состояний управляющего устройства*,
2) I — *входной алфавит* (не содержащий $¢$ и \$),
3) T — *магазинный алфавит* (не содержащий Z_0),
4) δ — отображение множества $(S-\{s_f\}) \times (I \cup \{¢, \$\}) \times (T \cup \{Z_0\})$. Значение $\delta(s, a, A)$, если определено, имеет один из следующих типов: $(s', d,$ **push** $B)$, (s', d) или $(s', d,$ **pop**$)$, где $s' \in S$, $B \in T$ и $d \in \{-1, 0, +1\}$. Мы считаем, что в заключительном состоянии s_f 2ДМА не делает никаких шагов, для некоторых других состояний шаги работы могут быть не определены. Кроме того, для любых s и A во второй компоненте у $\delta(s, ¢, A)$ не стоит $d=-1$, во второй компоненте у $\delta(s, \$, A)$ не стоит $d=+1$. Наконец, в третьей компоненте у $\delta(s, a, Z_0)$ не стоит **pop**.
5) $s_0 \in S$ — *начальное состояние* управляющего устройства,
6) Z_0 — *маркер дна (нижний маркер)* магазина,
7) s_f — выделенное *заключительное состояние*.

Мгновенным описанием (МО) 2ДМА P на входе $w = a_1 a_2 \ldots a_n$ называется тройка (s, i, α), где

1) s — состояние из S,
2) i — целое число, $0 \leqslant i \leqslant n+1$, указывающее положение входной головки (считаем, что $a_0 = ¢$ и $a_{n+1} = \$$),
3) α — цепочка, представляющая содержимое магазина, причем самый левый символ в α расположен в вершине магазина.

Шаг 2ДМА на входе $a_1a_2\ldots a_n$ определим с помощью бинарного отношения $\vdash$ на МО:

1) $(s, i, A\alpha) \vdash (s', i+d, BA\alpha)$, если $\delta(s, a_i, A)=(s', d, \textbf{push } B)$,
2) $(s, i, A\alpha) \vdash (s', i+d, A\alpha)$, если $\delta(s, a_i, A)=(s', d)$,
3) $(s, i, A\alpha) \vdash (s', i+d, \alpha)$, если $\delta(s, a_i, A)=(s', d, \textbf{pop})$.

Заметим, что добавление, считывание и удаление символов происходит только в вершине магазина. Кроме того, требуется, чтобы в любой момент времени в магазине 2ДМА был только один маркер дна. Для обозначения последовательностей из нуля или более шагов 2ДМА используется знак рефлексивного и транзитивного замыкания $\vdash^*$ отношения $\vdash$.

Начальное мгновенное описание 2ДМА P для $w=a_1a_2\ldots a_n$ (без концевых маркеров) имеет вид $(s_0, 1, Z_0)$, показывающий, что P находится в своем начальном состоянии s_0, входная головка обозревает самый левый символ из $a_1a_2\ldots a_na_{n+1}$[1]) и магазин содержит только нижний маркер Z_0.

Допускающее мгновенное описание для $w=a_1\ldots a_n$ имеет вид (s_f, i, Z_0) для некоторого $0\leqslant i\leqslant n+1$, показывающий, что автомат P перешел в заключительное состояние s_f и магазин содержит только нижний маркер.

Говорят, что 2ДМА P *допускает* входную цепочку w, если при входе w

$$(s_0, 1, Z_0)\vdash^*(s_f, i, Z_0)$$

для некоторого $0\leqslant i\leqslant |w|+1$. *Языком* $L(P)$, *допускаемым* автоматом P, называется множество всех цепочек, допускаемых P.

Рассмотрим несколько примеров 2ДМА и допускаемых ими языков. 2ДМА будем описывать неформально — в терминах их поведения, а не как семерки.

Пример 9.11. Рассмотрим язык $L=\{xcy \mid x,y\in\{a, b\}^*$ и y — подцепочка цепочки $x\}$. 2ДМА P может распознать L следующим образом. Допустим, что автомату P дана входная цепочка w вида xcy, где $x=a_1a_2\ldots a_n$ и $a_i\in\{a, b\}$, $1\leqslant i\leqslant n$.

1. P сдвигает свою входную головку вправо, пока не встретит c.

2. P сдвигает свою входную головку влево, переписывая $a_na_{n-1}\ldots a_1$ в магазин, пока входная головка не достигнет левого концевого маркера. В этот момент в магазине записано $xZ_0=a_1a_2\ldots\ldots a_nZ_0$ (самый левый символ цепочки xZ_0 расположен в вершине магазина).

[1]) Если $n=0$, т. е. $w=e$, то P обозревает $a_{n+1}=\$$, т. е. правый концевой маркер.

3. Затем P сдвигает свою входную головку к первому символу, стоящему справа от c (т. е. к началу цепочки y), не трогая магазин, и готовится сравнивать y с магазинным списком.

4. **while** символ в вершине магазина совпадает с символом, обозреваемым входной головкой **do**

begin
удалить верхний символ из магазина;
сдвинуть входную головку на одну клетку вправо
end

5. В этот момент возможны два случая.
   (а) Входная головка достигла правого концевого маркера \$, и тогда P допускает вход.
   (б) Символ в вершине магазина не совпадает с текущим входным символом. Поскольку цепочка y и магазинный список совпадали до сих пор, P может восстановить магазинный список, сдвигая входную головку влево и переписывая вход в магазин, пока не встретит c. В этот момент в магазине автомата P записана цепочка $a_i a_{i+1} \ldots a_n Z_0$ при некотором $1 \leqslant i \leqslant n$ [1]). Если $i=n$, то P останавливается в незаключительном состоянии. В противном случае P удаляет a_i из магазина и сдвигает входную головку вправо к первому символу цепочки $y\$$. Затем P возвращается к шагу 4, пытаясь установить, не является ли y префиксом цепочки $a_{i+1} \ldots a_n$.

Мы видим, что P распознает, является ли y подцепочкой цепочки $x=a_1 a_2 \ldots a_n$, вполне естественным способом. P пытается по очереди идентифицировать y с префиксом цепочки $a_i a_{i+1} \ldots a_n$ для $i=1, 2, \ldots, n$. Если это ему удается, он допускает вход. Если нет, отвергает. Заметим, что на эту процедуру P может затратить $O(|x| \cdot |y|)$ шагов. □

Пример 9.12. Рассмотрим язык $L=\{w \mid w \in \{a, b, c\}^*, w=xy$, где $|x| \geqslant 2$ и $x^R=x\}$ (т. е. w имеет префикс длины не менее 2, являющийся палиндромом). Здесь верхний индекс R обозначает обращение цепочки; например, $(abb)^R=bba$. Требование $|x| \geqslant 2$ наложено потому, что всякая цепочка из $\{a, b, c\}^+$ начинается с одного из тривиальных палиндромов a, b или c. Рассмотрим 2ДМА P, который на данной входной цепочке $a_1 a_2 \ldots a_n$ ведет себя следующим образом.

1. P сдвигает входную головку к правому концевому маркеру, переписывая вход в магазин. Теперь магазин содержит $a_n \ldots a_2 a_1 Z_0$. Если $n<2$, то P сразу отвергает вход.

---

[1]) Когда мы попадаем в такую ситуацию первый раз, $i=1$, но впоследствии i будет возрастать каждый раз на 1.

2. Затем P сдвигает входную головку к левому концевому маркеру, не трогая магазинный список. P устанавливает входную головку на символ, стоящий непосредственно справа от левого концевого маркера.

3. **while** символ в вершине магазина совпадает с символом, обозреваемым входной головкой **do**

   begin

   удалить верхний символ из магазина;

   сдвинуть входную головку на одну клетку вправо

   end

4. В этот момент возможны два случая.

   (а) Магазин содержит только Z_0, и тогда P останавливается и допускает вход.

   (б) Символ в вершине магазина не совпадает с текущим входным символом. В этом случае P сдвигает входную головку влево, переписывая вход в магазин, пока головка не достигнет левого концевого маркера. Если первоначальная входная цепочка имела вид $a_1a_2 \ldots a_n$, то в этот момент в магазине автомата P записано $a_i \ldots a_2a_1Z_0$ для некоторого i. Если $i=2$, то P останавливается и отвергает вход. В противном случае P удаляет a_i из магазина и сдвигает входную головку на одну клетку вправо — к символу, стоящему непосредственно справа от левого концевого маркера. Затем P возвращается к шагу 3.

Таким образом, P устанавливает, начинается ли входная цепочка с палиндрома, пытаясь последовательно идентифицировать $a_i \ldots a_1$ с префиксом цепочки $a_1 \ldots a_n$ для каждого i, $2 \leqslant i \leqslant n$. На эту процедуру P, возможно, затратит $O(n^2)$ шагов. □

Покажем, что за время $O(|w|)$ можно распознать, допускает ли 2ДМА P входную цепочку w, независимо от того, сколько шагов в действительности делает P. На протяжении последующего обсуждения мы считаем фиксированными 2ДМА $P=(S, I, T, \delta, s_0, Z_0, s_f)$ и входную цепочку w длины n.

Определение. *Поверхностной конфигурацией* (для краткости *конфигурацией*) автомата P называют такую тройку (s, i, A), что $s \in S$, i — положение входной головки, $0 \leqslant i \leqslant n+1$, и A — один из магазинных символов из $T \cup \{Z_0\}$.

Заметим, что конфигурация является МО, но не всякое МО является конфигурацией. В некотором смысле каждая поверхностная конфигурация представляет многие МО — те, у которых "поверхность", т. е. вершина магазина, совпадает с третьей компонентой

этой конфигурации. Говорят, что конфигурация $C'=(s', i', A)$ *выводима* из конфигурации $C=(s, i, A)$, и пишут $C \Rightarrow C'$, если найдется такая последовательность шагов автомата P, что при $m \geqslant 0$

$$\begin{aligned} C &\vdash (s_1, i_1, \alpha_1) \\ &\vdash (s_2, i_2, \alpha_2) \\ &\vdots \\ &\vdash (s_m, i_m, \alpha_m) \\ &\vdash C', \end{aligned}$$

где $|\alpha_i| \geqslant 2$ для каждого i [1]). В этой последовательности шагов промежуточные МО (s_j, i_j, α_j) имеют в магазине не менее двух символов. Мы будем оперировать с конфигурациями, а не с МО, потому что конфигураций только $O(n)$, а число различных МО может быть экспоненциальным.

Определение. Конфигурацию (s, i, A) называют *терминальной*, если значение $\delta(s, a_i, A)$ не определено или имеет вид $(s', d,$ **pop**$)$. Иными словами, терминальная конфигурация приводит или к остановке автомата P, или к удалению символа из магазина. *Терминатором* конфигурации C называется такая единственная терминальная конфигурация C' (если она существует), что $C \Rightarrow^* C'$, где $\Rightarrow^*$ — рефлексивное и транзитивное замыкание отношения $\Rightarrow$.

Пример 9.13. Если изобразить длину магазинного списка как функцию числа шагов, сделанных автоматом P, можно получить кривую типа кривой на рис. 9.17. На этом рисунке каждая точка помечена конфигурацией (не МО) автомата P после каждого шага.

Из рис. 9.17 видно, например, что в конфигурациях C_0 и C_{11} в вершине стека находится Z_0. Поскольку между этими конфигурациями нет конфигурации с Z_0 в вершине стека, то $C_0 \Rightarrow C_{11}$. Если C_{11} — заключительная конфигурация, то C_{11} служит терминатором как для себя, так и для C_0. Из этого рисунка можно также вывести, что $C_1 \Rightarrow C_2$, $C_2 \Rightarrow C_7$, $C_1 \Rightarrow^* C_{10}$ и C_{10} — терминальная конфигурация, поскольку она приводит к тому, что P удаляет символ из магазина. □

Ключевые соображения для алгоритма моделирования содержатся в следующих двух простых леммах.

Лемма 9.1. *P допускает w тогда и только тогда, когда некоторая конфигурация вида (s_i, i, Z_0), $0 \leqslant i \leqslant |w|+1$, служит терминатором начальной конфигурации $(s_0, 1, Z_0)$.*

Доказательство. Результат вытекает прямо из определения того, что значит "2ДМА допускает входную цепочку". □

---

[1]) Если $m=0$, то $C \vdash C'$.

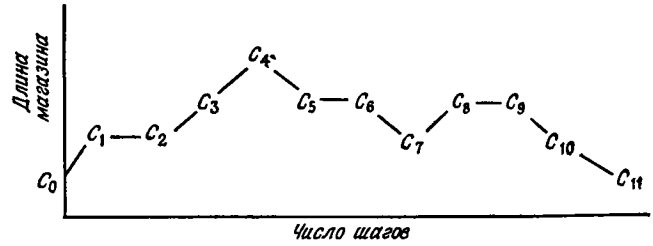


Рис. 9.17. Последовательность конфигураций.

Определение. Конфигурацию C называют *предшественницей* конфигурации D, если $C \Rightarrow^* D$, и *непосредственной предшественницей* для D, если $C \Rightarrow D$. Говорят, что пара конфигураций (C, D) *лежит ниже* пары (C', D') (необязательно различных) конфигураций, если

(1) $C=(s, i, A)$, $D=(t, j, A)$,
(2) $C'=(s', i', B)$, $D'=(t', j', B)$,
(3) $(s, i, A) \vdash (s', i', BA)$,
(4) $(t', j', BA) \vdash (t, j, A)$,

т. е. если P может дойти из C в C' с помощью шага **push**, а из D' в D с помощью шага **pop**.

Лемма 9.2. *Если* (C, D) *лежит ниже* (C', D') *и* $C' \Rightarrow^* D'$, *то* $C \Rightarrow D$.

Доказательство. Легкое упражнение. □

Пример 9.14. На рис. 9.17 пара (C_3, C_5) лежит ниже пары (C_4, C_4), а (C_7, C_{10}) — ниже (C_8, C_9). Но нельзя сказать с уверенностью, что (C_2, C_{10}) лежит ниже (C_3, C_9), поскольку символы в вершинах магазинов для C_3 и C_9 могут быть различными. □

Работа моделирующего алгоритма основана на поиске терминатора для каждой конфигурации 2ДМА P при входе w. Как только найден терминатор начальной конфигурации $(s_0, 1, Z_0)$, цель достигнута.

Для хранения терминатора каждой конфигурации используется массив, называемый ТЕРМ. Мы предполагаем, что множество конфигураций линейно упорядочено (с помощью каких-то лексикографических условий). Тогда можно обращаться с именем C конфигурации как с целым числом и считать ТЕРМ[C] терминатором для C.

Используется также массив списков ПРЕД. Он индексируется конфигурациями, и ПРЕД[D] — список конфигураций C, для которых $C \Rightarrow D$.

Кроме массивов ТЕРМ и ПРЕД используются два дополнительных списка НОВ и ВРЕМ. Список НОВ содержит пары еще не рас-

смотренных конфигураций (C, D), для которых ТЕРМ[C]=D. Список ВРЕМ нужен в процедуре КОРРЕКТИР(C, D), чтобы хранить предшественниц конфигурации C.

Мы поступаем следующим образом. Сначала полагаем ТЕРМ[C]=C, если C — терминальная конфигурация. (Каждая терминальная конфигурация является своим терминатором.) Добавляем (C, C) к НОВ. Затем рассматриваем такие пары (C, D), что $C \vdash D$ за один шаг работы P. (Заметим, что при таком шаге магазин не меняется.)

Если терминатор для D уже известен, полагаем ТЕРМ[E]=ТЕРМ[D] для всех E, о которых в этот момент известно, что они — предшественницы конфигурации C, включая саму C. (Собственные предшественницы находятся в ПРЕД[C].) Добавляем также пару (E, ТЕРМ[D]) к списку НОВ.

Если терминатор для D еще не известен, то C помещается в ПРЕД[D], т. е. в список непосредственных предшественниц конфигурации D.

В этот момент для каждой конфигурации C известна такая единственная конфигурация D, что $C \Rightarrow^* D$ без изменения магазина, и либо D — терминатор для C, либо $D \vdash (s, i, \alpha)$ при $|\alpha|=2$. Теперь рассматриваем все пары конфигураций, уже добавленные к списку НОВ. В общем случае НОВ содержит нерассмотренные пары конфигураций (A, B), в которых $A \Rightarrow^* B$ и B терминальна. Допустим, что НОВ содержит пару (A, B). Удаляем (A, B) из НОВ и рассматриваем все пары (C, D) конфигураций, лежащих ниже (A, B). Если терминатор для D уже вычислен, полагаем ТЕРМ[C]=ТЕРМ[D] и добавляем к списку НОВ пару (C, ТЕРМ[D]). Для каждой конфигурации E из ПРЕД[C] полагаем ТЕРМ[E]=ТЕРМ[D] и добавляем (E, ТЕРМ[D]) к списку НОВ. Но если терминатор для D еще не вычислен, помещаем C в список ПРЕД[D]. Продолжаем эту процедуру, пока НОВ не опустеет. В этот момент найден терминатор (если он существует) для каждой конфигурации C.

Исчерпав весь список НОВ, рассматриваем ТЕРМ[C_0], где C_0 — начальная конфигурация. Если ТЕРМ[C_0] — допускающая конфигурация, то 2ДМА P допускает w. В противном случае P отвергает w.

Дадим более точное описание.

Алгоритм 9.4. Моделирование 2ДМА

Вход. 2ДМА $P=(S, I, T, \delta, s_0, Z_0, s_f)$ и входная цепочка $w \in I^*$, $|w|=n$.

Выход. Ответ "да", если $w \in L(P)$, и "нет" в противном случае.

Метод.

1. Произведем начальную загрузку массивов и списков следующим образом. Для каждой конфигурации C положим ТЕРМ[C]=$\varnothing$ и ПРЕД[C]=$\varnothing$. Положим НОВ=$\varnothing$ и ВРЕМ=$\varnothing$.

```
   procedure КОРРЕКТИР (C, D):
   begin
     comment Всякий раз, когда вызывается КОРРЕКТИР(C, D),
       имеем C ⇒ D;
1.   if ТЕРМ[D]=∅ then добавить C к ПРЕД[D]
     else
       begin
2.       ВРЕМ ← {C};
3.       while ВРЕМ ≠ ∅ do
           begin
4.           выбрать и удалить конфигурацию B из ВРЕМ;
5.           ТЕРМ[B] ← ТЕРМ[D];
6.           добавить (B, ТЕРМ[D]) к НОВ;
7.           for A ∈ ПРЕД[B] do добавить A к ВРЕМ
           end
       end
   end
```

Рис. 9.18. Процедура КОРРЕКТИР.

2. Для каждой терминальной конфигурации C полагаем ТЕРМ[C]=C и добавляем к НОВ пару (C, C).

3. Для каждой конфигурации C проверяем, существует ли такая конфигурация D, что $C \vdash D$ за один шаг. Если да, вызываем КОРРЕКТИР(C, D). Процедура КОРРЕКТИР приведена на рис. 9.18.

4. Пока список НОВ не пуст, удаляем пару (C', D') из НОВ. Для каждой пары (C, D), лежащей ниже (C', D'), вызываем КОРРЕКТИР(C, D).

5. Если ТЕРМ[C_0], где C_0 — начальная конфигурация, является заключительной конфигурацией, получаем ответ "да". В противном случае "нет". □

Пример 9.15. Рассмотрим некоторые вычисления, выполняемые алгоритмом 9.4, когда он применяется к 2ДМА, работающему, как показано на рис. 9.17.

На шаге 2 обнаруживаем, что C_4, C_8, C_9, C_{10} и C_{11} — терминальные конфигурации и потому свои же терминаторы. Добавляем к НОВ пары (C_4, C_4), (C_8, C_8), (C_9, C_9), (C_{10}, C_{10}) и (C_{11}, C_{11}).

На шаге 3 вызываем КОРРЕКТИР(C_1, C_2). Поскольку в этот момент ТЕРМ[C_2]=∅, то КОРРЕКТИР всего лишь помещает C_1 в список ПРЕД[C_2]. На шаге 3 вызываем также КОРРЕКТИР(C_5, C_6). Так как ТЕРМ[C_6]=C_6, то КОРРЕКТИР полагает ТЕРМ[C_5]=C_6

и добавляет (C_5, C_6) к НОВ. Кроме того, на шаге 3 вызываем КОРРЕКТИР(C_8, C_9), и этот вызов полагает ТЕРМ$[C_8]=C_9$ и добавляет (C_8, C_9) к НОВ. Поэтому после шага 3 НОВ содержит пары

(C_4, C_4), (C_6, C_6), (C_9, C_9), (C_{10}, C_{10}), (C_{11}, C_{11}), (C_5, C_6), (C_8, C_9).

На шаге 4 удаляем (C_4, C_4) из НОВ и вызываем КОРРЕКТИР(C_3, C_5), поскольку (C_3, C_5) лежит ниже (C_4, C_4). Поскольку в этот момент ТЕРМ$[C_5]=C_6$, КОРРЕКТИР полагает ТЕРМ$[C_3]=C_6$ и добавляет (C_3, C_6) к НОВ. Затем на шаге 4 удаляем (C_6, C_6) из НОВ и, поскольку (предположим, что это так) ниже (C_6, C_6) не лежит никакая пара, не вызываем КОРРЕКТИР [1]). Аналогично не вызываем КОРРЕКТИР для пар (C_9, C_9), (C_{10}, C_{10}), (C_{11}, C_{11}) и (C_5, C_6). Когда (C_8, C_9) удаляется из НОВ, вызываем КОРРЕКТИР(C_7, C_{10}), и этот вызов полагает ТЕРМ$[C_7]=C_{10}$ и добавляет (C_7, C_{10}) к НОВ. В этот момент НОВ содержит (C_3, C_6) и (C_7, C_{10}).

Удалив (C_3, C_6) из НОВ, вызываем КОРРЕКТИР(C_2, C_7), что приводит к ТЕРМ$[C_2]=C_{10}$ и ТЕРМ$[C_1]=C_{10}$, поскольку ПРЕД$[C_2]$ содержит C_1. Добавляем (C_2, C_{10}) и (C_1, C_{10}) к НОВ.

Предлагаем читателю завершить это моделирование. □

Теорема 9.10. *Алгоритм* 9.4 *правильно отвечает на вопрос "Принадлежит ли слово* w *языку* $L(P)$*?" за время* $O(|w|)$.

Доказательство. Можно показать, что каждая конфигурация попадает в список ВРЕМ не более одного раза. Поэтому каждый вызов процедуры КОРРЕКТИР закончится. Также можно показать, что никакая пара конфигураций не попадает в список НОВ более одного раза, следовательно, и сам алгоритм закончит работу. Подробное доказательство этих двух утверждений оставляем в качестве упражнения.

Покажем, что по окончании работы алгоритма 9.4 ТЕРМ$[C_0]$ будет заключительной конфигурацией тогда и только тогда, когда $w \in L(P)$. Легко показать индукцией по числу шагов работы алгоритма 9.4, что

1) если ТЕРМ$[C]$ полагается равным D, то D — терминатор для C,
2) если C добавляется к ПРЕД$[D]$, то $C \Rightarrow D$,
3) если (C, D) помещается в НОВ, то ТЕРМ$[C]=D$.

Таким образом, если алгоритм 9.4 обнаруживает, что ТЕРМ$[C_0]$— заключительная конфигурация, то в силу леммы 9.1 утверждение "$w \in L(P)$" верно.

Кроме того, надо показать, что если D — терминатор для C, то ТЕРМ$[C]$ в конце концов становится равным D. Доказательство

[1]) Для простоты мы считаем, что P не делает шагов, которые не следуют из рис. 9.17.

проводится индукцией по числу шагов в последовательности $C \vdash^* D$. Базис, т. е. случай нулевого числа шагов, тривиален, поскольку $C=D$ и ТЕРМ[C] делается равным D на шаге 2 алгоритма 9.4.

Для шага индукции предположим, что $C \vdash E \vdash^* D$, и рассмотрим отдельно два случая.

Случай 1. E — конфигурация, т. е. шаг $C \vdash E$ не является ни шагом **push**, ни шагом **pop**. Тогда на шаге 3 вызывается КОРРЕКТИР(C, E). Если к этому моменту ТЕРМ[E] было присвоено значение D, то конфигурация C будет помещена в список ВРЕМ в строке 2 и в конце концов значение ТЕРМ[C] сделается равным D в строке 5. Если значение ТЕРМ[E] еще не равно D, то добавляем C к ПРЕД[E] в строке 1 процедуры КОРРЕКТИР(C, E). По предположению индукции мы в конце концов получим, что ТЕРМ[E]=D. Если это случится в строке 5 процедуры КОРРЕКТИР, то C добавится к ВРЕМ в строке 7, а значение ТЕРМ[C] сделается равным D во время этого вызова процедуры КОРРЕКТИР. Значение ТЕРМ[E] не может стать равным D на шаге 2 алгоритма 9.4, поскольку $E \neq D$. (Если бы оказалось, что $E=D$, то ТЕРМ[E] уже было бы присвоено значение D к тому моменту, когда мы стали рассматривать $C \vdash E$ на шаге 3.) Итак, в случае 1 значение ТЕРМ[C] оказывается равным D.

Случай 2. E — это такое МО, что $C \vdash E$ является шагом **push**. Тогда можно найти такие конфигурации A, B и F, что (C, F) лежит ниже (A, B), $A \vdash^* B$ и $F \vdash^* D$, причем каждый из этих переходов совершается за меньшее число шагов, чем переход $C \vdash^* D$. (Конфигурация A служит "поверхностью" МО E.) По предположению индукции, значение ТЕРМ[A] полагается равным B, а значение ТЕРМ[F] — равным D.

Допустим, что последнее происходит раньше первого. Тогда (A, B) со временем помещается в НОВ, а на шаге 4 вызывается КОРРЕКТИР(C, F). Так как в этот момент ТЕРМ[F]=D, то в строке 5 полагаем ТЕРМ[C]=D.

Допустим теперь, что ТЕРМ[A] присвоено значение B до того, как ТЕРМ[F] присвоено значение D. Тогда при вызове КОРРЕКТИР(C,F) ТЕРМ[F]=$\varnothing$ и C добавляется к ПРЕД[F]. Но тогда D становится значением ТЕРМ[C] при вычислении ТЕРМ[F]. Это завершает индукцию и доказательство корректности алгоритма 9.4.

Проанализируем время работы алгоритма 9.4. Шаги 1 и 2 занимают $O(n)$ времени, поскольку всего конфигураций $O(n)$. Так как для каждой конфигурации 2ДМА делает не более одного шага, то пар конфигураций (C, D), в которых $C \vdash D$, всего $O(n)$. Поэтому процедура КОРРЕКТИР вызывается на шаге 3 не более $O(n)$ раз.

Пара (C', D') попадает в список НОВ, когда $C' \Rightarrow^* D'$ и уже найдено, что D' — терминатор для C'. Так как каждая конфигурация имеет лишь один терминатор (если она его вообще имеет), то никакая пара не помещается в список НОВ более одного раза. Следователь-

но, общее число пар, помещаемых в список НОВ, не превосходит $O(n)$. Ниже каждой пары, попавшей в НОВ, лежит ограниченное число пар, ибо если (C, D) лежит ниже (C', D'), то положения головок конфигураций C и C', а также D и D', различаются не более чем на 1. Поэтому КОРРЕКТИР вызывается $O(n)$ раз.

Теперь оценим общее время, затрачиваемое на подпрограмму КОРРЕКТИР. Можно показать, что в массиве ПРЕД каждая конфигурация появляется не более одного раза и в список ВРЕМ никакая конфигурация не попадает более одного раза. Общее время выполнения строк 4—6 процедуры КОРРЕКТИР пропорционально количеству конфигураций B, удаляемых из ВРЕМ, а время выполнения строки 7 — количеству конфигураций A, добавляемых к ВРЕМ. Так как КОРРЕКТИР вызывается не более $O(n)$ раз, то общее время, занимаемое процедурой КОРРЕКТИР, без учета времени на строки 4—6 и 7 составляет $O(n)$. Следовательно, алгоритм 9.4 работает линейное время. □

Результаты настоящего раздела применяются главным образом при доказательстве существования алгоритмов линейной сложности, решающих определенные задачи. Мы уже видели, что некоторые задачи идентификации образов можно сформулировать как задачи распознавания языков. Если мы сможем построить 2ДМА, распознающий язык, соответствующий задаче идентификации образов, то сможем утверждать, что существует алгоритм линейной сложности, решающий ту же задачу. Так как на входе длины n 2ДМА может делать n^2 или даже 2^n шагов, то часто бывает проще найти алгоритм для распознавания языка на 2ДМА, чем алгоритм линейной сложности для решения задачи идентификации образов прямо на РАМ.

9.5. ПОЗИЦИОННЫЕ ДЕРЕВЬЯ И ИДЕНТИФИКАТОРЫ ПОЗИЦИЙ

В предыдущем разделе мы показали, что если задачу идентификации образов можно сформулировать как задачу распознавания языка, для которой можно найти решающий ее 2ДМА, то исходную задачу идентификации образов можно решить за линейное время. Можно было бы взять прямо алгоритм 9.4 как алгоритм линейной сложности, решающий исходную задачу. Но мультипликативная постоянная, возникающая при таком моделировании, сделала бы этот подход в лучшем случае непривлекательным. В настоящем разделе мы изучим структуру данных, которую можно использовать для идентификации образов с помощью более практичных алгоритмов, работающих линейное время. Эта структура данных применима, в частности, к следующим задачам.

1. По данным цепочке-тексту x и цепочке-образу y найти все вхождения y в x.

2. По данной цепочке-тексту x найти самую длинную повторяющуюся подцепочку в x.
3. По данным двум цепочкам x и y найти самую длинную подцепочку, входящую как в x, так и в y.

Определение. *Позицией* в цепочке длины n считается любое число от 1 до n. Говорят, что символ a *входит в цепочку* x *в позиции* i, если $x=yaz$ и $|y|=i-1$. Говорят, что подцепочка u *идентифицирует* позицию i в цепочке x, если $x=yuz$, где $|y|=i-1$, и цепочку x можно представить в виде $y'uz'$ только при $y'=y$, т. е. в позиции i начинается единственное вхождение цепочки u в x. Например, подцепочка bba идентифицирует позицию 2 в цепочке $abbabb$. Подцепочка bb не идентифицирует позицию 2.

В оставшейся части этой главы положим $x=a_1a_2\ldots a_n$ — цепочка в некотором алфавите I и $a_{n+1}=\$\notin I$. Тогда каждая позиция i цепочки $x\$=a_1a_2\ldots a_na_{n+1}$ идентифицируется по крайней мере одной цепочкой, а именно $a_ia_{i+1}\ldots a_{n+1}$. Кратчайшая цепочка, идентифицирующая позицию i в $x\$$, называется *идентификатором* (*для*) *позиции* i *в* x и обозначается через $s(i)$.

Пример 9.16. Рассмотрим цепочку $abbabb\$$. Идентификаторы позиций от 1 до 7 приведены на рис. 9.19. □

Идентификаторы позиций в цепочке $x\$$ удобно представлять с помощью дерева, называемого I-деревом, в котором помечены ребра и некоторые узлы.

Определение. *I-деревом* называют помеченное дерево T, ребра которого, выходящие из любого внутреннего узла, помечены различными символами из I. Если ребро $(v, w)\in T$ помечено символом a, то w называют *a-сыном* узла v.

Позиция	Идентификатор
1	$abba$
2	bba
3	ba
4	$abb\$$
5	$bb\$$
6	$b\$$
7	$\$$

Рис. 9.19. Идентификаторы для $abbabb\$$.

Деревом позиций (или *позиционным деревом*) (для) цепочки $x\$ = a_1 \ldots a_n a_{n+1}$, где $a_i \in I$, $1 \leqslant i \leqslant n$, и $a_{n+1} = \$$, называют $(I \cup \{\$\})$-дерево, для которого выполнены следующие условия.

1) T имеет $n+1$ листьев, помеченных числами $1, 2, \ldots, n+1$. Листья дерева T взаимно однозначно соответствуют позициям в $x\$$.
2) Последовательность реберных меток, стоящих на пути из корня в лист с меткой i, служит идентификатором $s(i)$ позиции i.

Пример 9.17. Позиционное дерево для цепочки *abbabb*\$ изображено на рис. 9.20. Например, путь из корня в лист с меткой 2 помечен цепочкой *bba*, являющейся идентификатором позиции 2. □

Сформулируем некоторые основные свойства идентификаторов.

Лемма 9.3. *Пусть $s(i)$ — идентификатор позиции i цепочки $x\$ = a_1 a_2 \ldots a_{n+1}$.*

(а) *Если $s(i)$ имеет длину j, то длина подцепочки $s(i-1)$ не превосходит $j+1$.*
(б) *Никакой идентификатор не является собственным префиксом другого.*

Доказательство. (а) Если длина подцепочки $s(i-1)$ больше $j+1$, то найдется такая позиция $k \neq i-1$, что $a_{i-1} a_i \ldots a_{i+j-1} = a_k a_{k+1} \ldots a_{k+j}$. Поэтому $a_i a_{i+1} \ldots a_{i+j-1} = a_{k+1} a_{k+2} \ldots a_{k+j}$, и $a_i a_{i+1} \ldots a_{i+j-1}$ не идентифицирует позицию i; получили противоречие.

(б) Легкое упражнение. □

Лемма 9.3(б) гарантирует, что для каждой цепочки $x\$$ действительно существует позиционное дерево.

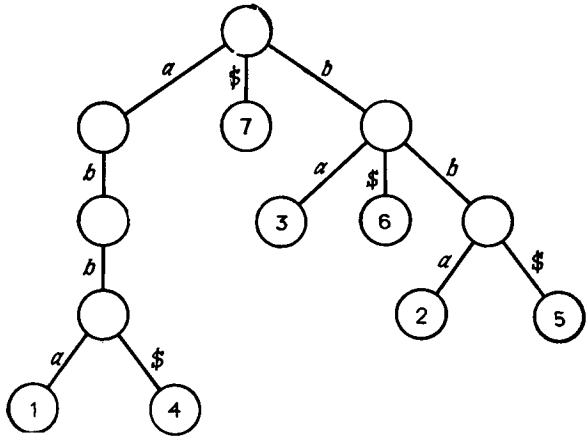


Рис. 9.20. Позиционное дерево.

С помощью позиционных деревьев можно решить многие задачи идентификации образов, включая те, что были упомянуты в начале раздела.

Пример 9.18. Исследуем основную задачу идентификации образов: "Является ли $y=b_1b_2\ldots b_p$ подцепочкой цепочки $x=a_1a_2\ldots a_n$?" Допустим, что мы уже построили для $x\$$ позиционное дерево T. Чтобы узнать, входит ли $y=b_1b_2\ldots b_p$ в x, рассмотрим его как граф переходов некоторого детерминированного конечного автомата. Иными словами, отправляясь из корня дерева T, проследим максимально длинный возможный путь в позиционном дереве, которому приписана цепочка $b_1b_2\ldots b_j$ для некоторого $0\leqslant i\leqslant p$. Пусть этот путь оканчивается в узле v. Возможны несколько случаев.

1. Если $j<p$ и v не лист, то ответом будет "нет". В этом случае $b_1b_2\ldots b_j$ — подцепочка цепочки x, а $b_1b_2\ldots b_jb_{j+1}$ нет.

2. Если $j\leqslant p$ и v — лист с меткой i, то $b_1b_2\ldots b_j$ совпадает с j символами цепочки x, начиная с позиции i. Тогда надо сравнить $b_{j+1}b_{j+2}\ldots b_p$ с $a_{i+j}a_{i+j+1}\ldots a_{i+p-1}$. Если они не совпадают, то ответом будет "нет"; в противном случае "да", причем y входит в x, начиная с позиции i.

3. Если $j=p$ и v не лист, то ответом будет "да". В этом случае y — подцепочка с двумя или более вхождениями в x. Начальные позиции этих вхождений — метки листьев, принадлежащих поддереву узла v позиционного дерева. □

Пример 9.19. С помощью позиционного дерева можно найти самую длинную повторяющуюся подцепочку данной цепочки $x=a_1a_2\ldots a_n$ (два вхождения этой подцепочки могут перекрываться). Длиннейшая повторяющаяся подцепочка соответствует внутреннему узлу позиционного дерева с наибольшей глубиной. Такой узел можно обнаружить очевидным способом.

Рассмотрим нахождение длиннейшей повторяющейся подцепочки в *abbabb*. В позиционном дереве для *abbabb*\$ (рис. 9.20) есть один внутренний узел глубины 3 и ни одного внутреннего узла большей глубины. Поэтому соответствующая этому узлу цепочка *abb* — самая длинная повторяющаяся подцепочка в *abbabb*. Узлы с метками 1 и 4 указывают, где начинаются два вхождения цепочки *abb*. В нашем случае они не перекрываются. □

Изучим подробно задачу построения позиционного дерева. В оставшейся части этой главы $x\$$ будет цепочкой $a_1\ldots a_na_{n+1}$, где a_{n+1} — единственный правый концевой маркер \$. Через x_i, $1\leqslant i\leqslant n+1$, будем обозначать суффикс $a_i\ldots a_na_{n+1}$, а через $s_i(j)$ — идентификатор позиции j в x_i. Все позиции нумеруются по исходной цепочке $a_1\ldots a_na_{n+1}$.

Мы изложим алгоритм построения позиционного дерева для $a_1\ldots a_{n+1}$ за время, пропорциональное числу узлов окончательного

дерева. Заметим, что позиционное дерево для $a_1 \ldots a_n a_{n+1}$ может содержать $O(n^2)$ узлов. Например, позиционное дерево для $a^n b^n a^n b^n \$$ [1]) содержит n^2+6n+2 узлов, в чем легко убедиться самим. Но при разумных предположениях о том, что такое "случайная" цепочка (например, символы выбираются из алфавита с равной вероятностью и независимо), можно показать, что "среднее" позиционное дерево для цепочки длины n содержит $O(n)$ узлов. Мы не будем показывать это здесь. Отметим лишь, что существует алгоритм сложности $O(n)$ в худшем случае, который строит компактную форму позиционного дерева прямо по данной цепочке. Работы, обсуждающие этот алгоритм, указаны в замечаниях по литературе.

Рассмотрим различия между множествами S_i и S_{i+1} идентификаторов для x_i и x_{i+1} соответственно, поскольку в алгоритме, описываемом ниже, S_i строится из S_{i+1}. Одно очевидное различие состоит в том, что S_i содержит идентификатор $s_i(i)$ первой позиции в x_i. Из-за того, что S_i содержит эту дополнительную цепочку, может оказаться, что идентификатор некоторой позиции k, содержащийся в S_{i+1}, не является идентификатором позиции k в S_i. Это происходит тогда и только тогда, когда идентификатор $s_{i+1}(k)$ позиции k в S_{i+1} служит префиксом для $s_i(i)$. В этой ситуации надо удлинить цепочку $s_{i+1}(k)$, чтобы сделать из нее $s_i(k)$, т. е. идентификатор позиции k в S_i. Два идентификатора из S_{i+1} удлинять не придется. В самом деле, если бы надо было удлинять цепочки $s_{i+1}(k_1)$ и $s_{i+1}(k_2)$, то они обе были бы префиксами цепочки $s_i(i)$ и, значит, одна из них была бы префиксом другой вопреки лемме 9.3(б).

Пример 9.20. Пусть $a_1 \ldots a_n a_{n+1} = abbabb\$$. Идентификаторы для $x_4 = abb\$$, $x_3 = babb\$$, $x_2 = bbabb\$$ и $x_1 = abbabb\$$ приведены на рис. 9.21. Заметим, что S_3 получается из S_4 добавлением одной цепочки $s_3(3) = ba$. С другой стороны, для построения S_2 из S_3 потребовалось два изменения. Мы добавили $s_2(2) = bba$ к S_2 и дописали $\$$ к концу $s_3(5)$, чтобы получить $s_2(5) = bb\$$. Чтобы построить S_1, добавили $s_1(1) = abba$ к S_2 и дописали $bb\$$ к $s_2(4)$, чтобы получить $s_1(4) = abb\$$. □

Рассмотрим, что же нужно для построения позиционного дерева T_i, представляющего S_i, по позиционному дереву T_{i+1}, представляющему S_{i+1}. Пусть T_{i+1} построено. Надо добавить к T_{i+1} лист с меткой i, соответствующий цепочке $s_i(i)$. Если для некоторого k, $i < k \leqslant n$, цепочка $s_{i+1}(k)$ служит префиксом цепочки $s_i(i)$, то надо также удлинить путь в T_{i+1}, идущий из корня в лист с меткой k, чтобы новый лист с меткой k в T_i соответствовал цепочке $s_i(k)$. Таким образом, ключом к эффективному построению по T_{i+1} позиционного дерева T_i для x_i является умение быстро находить такую цепочку y, что $a_i y$ — самый длинный префикс цепочки x_i, входящий также в

[1]) a^n обозначает n-кратную конкатенацию символа a.

Позиция	$s_4(p)$	$s_3(p)$	$s_2(p)$	$s_1(p)$
7	\$	\$	\$	\$
6	*b*\$	*b*\$	*b*\$	*b*\$
5	*bb*	*bb*	*bb*\$	*bb*\$
4	*a*	*a*	*a*	*abb*\$
3	—	*ba*	*ba*	*ba*
2	—	—	*bba*	*bba*
1	—	—	—	*abba*

Рис. 9.21. Идентификаторы некоторых суффиксов цепочки *abbabb*\$.

x_{i+1}, и, кроме того, умение узнавать, служит ли $a_i y$ префиксом идентификатора из T_{i+1}.

Такое построение требует добавления к позиционному дереву трех новых структур. Первая из них — двоичный вектор (массив), который приписывается каждому узлу позиционного дерева T_i. Вектор, приписываемый узлу v, будет обозначаться B_v. Для каждого символа из I в этом векторе будет своя компонента. Если v — узел в T_i, соответствующий цепочке y, и $a \in I$, то $B_v[a]=1$, если ay — подцепочка в x_i. В противном случае $B_v[a]=0$.

Далее каждому узлу приписывается его глубина в позиционном дереве. Эту информацию легко корректировать по мере роста дерева, и впредь мы будем предполагать, не оговаривая это особо, что она вычисляется.

Последнее добавление к позиционному дереву — новая древовидная структура, расположенная на его узлах. Мы будем называть ее "вспомогательным деревом", но на самом деле это будет всего лишь другое множество ребер, соединяющих узлы нашего позиционного дерева.

Определение. Пусть T_i — позиционное дерево для $x_i=a_i a_{i+1}\ldots a_n$\$. *Вспомогательным деревом* для позиционного дерева T_i назовем $(I \cup \{\$\})$-дерево A_i, обладающее следующими свойствами:

1) A_i имеет то же множество узлов, что и T_i,
2) узел w является a-сыном узла v в A_i, если ay — цепочка, соответствующая w в T_i, и y — цепочка, соответствующая v в T_i. Таким образом, пути в A_i, идущему из корня в w, приписана цепочка $y^R a$, а пути в T_i, идущему из корня в w, приписана цепочка ay.

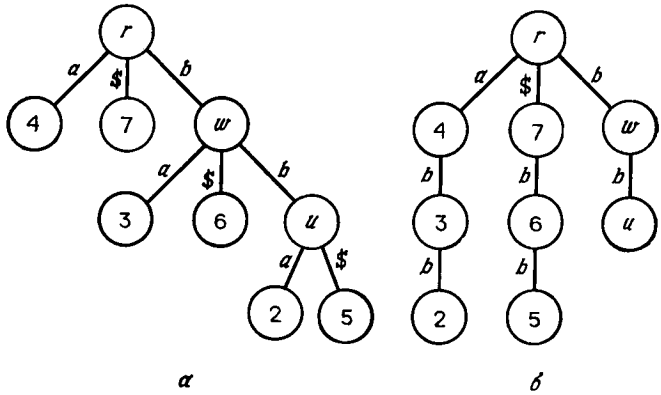


Рис. 9.22. Позиционное (*а*) и вспомогательное (*б*) деревья для *bbabb*\$.

Пример 9.21. Позиционное и вспомогательное деревья для $x_2 = bbabb\$$ изображены на рис. 9.22. (Числа на листьях указывают позиции относительно $x = abbabb\$$.) Заметим, что листья вспомогательного дерева не обязательно являются листьями позиционного дерева, хотя множество узлов у обоих деревьев одно и то же. □

Из определения не видно, всегда ли у позиционного дерева есть вспомогательное; действительно, произвольное I-дерево может не иметь вспомогательного. В следующей теореме сформулированы условия, при которых I-дерево имеет вспомогательное дерево.

Теорема 9.11. *Для того чтобы у данного I-дерева T было вспомогательное дерево, необходимо и достаточно, чтобы T обладало таким свойством: если в T есть узел, соответствующий цепочке ax, где $a \in I$ и $x \in I^*$, то в нем есть также узел, соответствующий цепочке x.*

Доказательство. Упражнение. □

Следствие. *Всякое позиционное дерево имеет вспомогательное дерево.*

Доказательство. Если ax — префикс идентификатора позиции i, то в силу леммы 9.3(а) x — префикс идентификатора позиции $i+1$. □

Теперь мы готовы описать алгоритм, который строит позиционное дерево T_i для x_i из позиционного дерева T_{i+1} для x_{i+1}.

Алгоритм 9.5. Построение T_i из T_{i+1}

Вход. Цепочка $a_1 \ldots a_n a_{n+1}$, позиционное T_{i+1} и вспомогательное A_{i+1} деревья для $x_{i+1} = a_{i+1} \ldots a_n a_{n+1}$.

Выход. Позиционное T_i и вспомогательное A_i деревья для x_i.
Метод.

1. Находим лист с меткой $i+1$ в T_{i+1}. (Это последний добавленный к T_{i+1} лист.)

2. Проходим по пути, ведущему из этого листа в корень дерева T_{i+1}, до такого узла u_y, что $B_{u_y}[a_i]=1$. (Узел u_y соответствует такой самой длинной цепочке y, что a_iy — префикс цепочки x_i и подцепочка в x_{i+1}, начинающаяся в некоторой позиции k, $i<k\leqslant n$.) Если такого узла нет, доходим до корня.

3. Полагаем $B_v[a_i]=1$ для каждого узла v на пути, идущем из узла с меткой $i+1$ в сына узла u_y, расположенного на том же пути, или в корень, если такого u_y нет. (Каждый узел v на этом пути соответствует некоторому префиксу z цепочки x_{i+1}. Поэтому ясно, что a_iz — подцепочка цепочки a_ix_{i+1}.)

4. 1) Если узла u_y нет, переходим к случаю 1 (ниже).
   2) Если узел u_y существует, но у него нет a_i-сына во вспомогательном дереве A_{i+1}, переходим к случаю 2.
   3) Если у u_y есть a_i-сын v_y во вспомогательном дереве, переходим к случаю 3. Узел v_y соответствует цепочке a_iy в позиционном дереве T_{i+1}.

Случай 1. a_i — символ, не входящий в x_{i+1}. Тогда идентификатором позиции i и будет a_i. Чтобы построить T_i из T_{i+1}, выполняем следующее:

(а) образуем новый лист с меткой i, являющийся a_i-сыном корня дерева T_{i+1},
(б) полагаем $B_i[a]=0$ для всех $a\in I$.

Чтобы построить A_i из A_{i+1}, делаем новый узел с меткой i a_i-сыном корня дерева A_{i+1}.

Случай 2. a_i входит в x_{i+1}, но в T_{i+1} есть лишь собственный префикс цепочки a_iy (приписанный пути из корня дерева T_{i+1} в некоторый лист v_1). Эта ситуация возникает тогда, когда нужно удлинять идентификатор некоторой позиции k, $i<k\leqslant n$, чтобы он стал идентификатором позиции k в T_i. Допустим, что $y=a_{i+1}a_{i+2}\ldots a_j$ и узел v_1 соответствует цепочке $a_ia_{i+1}\ldots a_p$ для некоторого $p<j$. Тогда k — метка листа v_1 и $a_ia_{i+1}\ldots a_p$ — идентификатор позиции k в T_{i+1} (т. е. $a_ia_{i+1}\ldots a_p=a_ka_{k+1}\ldots a_l$).

Чтобы построить T_i из T_{i+1}, добавляем к узлу v_1 поддерево с двумя новыми листьями, помеченными числами i и k. Пути, идущему из v_1 в k в этом добавленном поддереве, приписана цепочка $a_{l+1}a_{l+2}\ldots a_{m+1}$, а пути из v_1 в i — цепочка $a_{p+1}a_{p+2}\ldots a_{j+1}$, при этом $a_{l+1}a_{l+2}\ldots a_m=a_{p+1}a_{p+2}\ldots a_j$. Таким образом, пути из корня дерева T_i в лист k будет приписана цепочка $a_ka_{k+1}\ldots a_ma_{m+1}$, являющаяся новым идентификатором позиции k в x_i, а пути из корня в

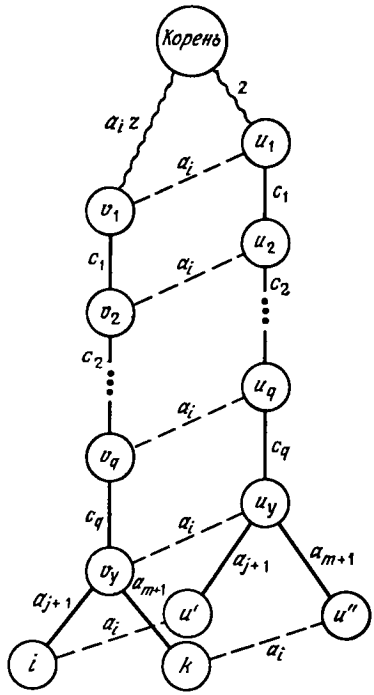


Рис. 9.23. Части дерева T_i, важные для случая 2. Штриховыми линиями изображены ребра дерева A_i.

лист i — цепочка $a_ia_{i+1}\ldots a_ja_{j+1}$, являющаяся идентификатором позиции i в x_i.

Точнее, для построения T_i из T_{i+1} выполняем следующее.

1) Двигаясь по пути в T_i, идущему из u_y в корень, находим узел u_1, являющийся для узла u_y первым предком с a_i-сыном в A_{i+1}. Пусть v_1 — этот a_i-сын. В T_{i+1} узел v_1 — лист с меткой k для некоторого $i<k\leqslant n$. Стираем у v_1 метку k.
2) Пусть $u_1, u_2, \ldots, u_q, u_y$ — последовательность узлов на пути в T_{i+1}, идущем из u_1 в u_y. Предположим, что ребро из u_h в u_{h+1} помечено символом c_h, $1\leqslant h<q$, а ребро из u_q в u_y — символом c_q, как показано на рис. 9.23 ($c_1c_2\ldots c_q=a_{p+1}a_{p+2}\ldots a_j=a_{l+1}a_{l+2}\ldots a_m$, где символы a_i те же, что и выше).
3) В T_i образуем q новых узлов $v_2, v_3, \ldots, v_q, v_y$. Делаем v_{h+1} c_h-сыном узла v_h для $1\leqslant h<q$. Делаем v_y c_q-сыном узла v_q.
4) Пусть $j=i+$ глубина узла u_y и $m=k+$ глубина узла u_y. Добавляем к T_i два новых узла с метками i и k. Лист i делаем a_{j+1}-сыном узла v_y, а лист k делаем a_{m+1}-сыном узла v_y.

5) Для всех $a \in I$ полагаем $B_v[a]=B_{v_1}[a]$ для каждого $v \in \{v_2, v_3, \ldots, v_q, v_y, k\}$.
6) Полагаем $B_i[a]=0$ для всех $a \in I$.

Чтобы построить A_i из A_{i+1}, выполняем следующее.

7) Делаем v_y a_i-сыном узла u_y в A_i.
8) Пусть u' будет a_{j+1}-сыном узла u_y в T_{i+1}. Новый лист с меткой i делаем a_i-сыном узла u' в A_i.
9) Пусть u'' будет a_{m+1}-сыном узла u_y в T_{i+1}. Новый лист с меткой k делаем a_i-сыном узла u'' в A_i.
10) Делаем v_h a_i-сыном узла u_h в A_i для $2 \leqslant h \leqslant q$.

Случай 3. Узел u_y имеет a_i-сына v_y в A_{i+1}. В этом случае надо рассмотреть две ситуации.

(а) v_y — лист с меткой k в T_{i+1}. Такая ситуация возникает, когда $s_i(i)=a_ia_{i+1}. \ . \ .a_ja_{j+1}$ и $s_{i+1}(k)=a_ka_{k+1}. \ . \ .a_m$, причем $a_ka_{k+1}\ldots$ $\ldots .a_m=a_ia_{i+1}. \ . \ .a_j$; это частный вид случая 2, когда $v_1=v_y$. Чтобы построить T_i из T_{i+1}, удаляем метку k из узла v_y и образуем для узла v_y a_{j+1}-сына с меткой i и a_{m+1}-сына с меткой k. Детали этого построения те же, что и в случае 2 (без шагов 3, 7 и 10).

(б) v_y — внутренний узел дерева T_{i+1}. Такая ситуация возникает, когда $S_i=S_{i+1} \cup \{s_i(i)\}$. Чтобы построить T_i из T_{i+1}, просто образуем для узла v_y a_{j+1}-сына (которого у него нет) и помечаем этот лист меткой i. Подробный алгоритм таков:

1 Пусть $j=i+$ глубина узла u_y.
2. Новый узел с меткой i делаем a_{j+1}-сыном узла v_y в T_i. (Заметим, что у v_y не может быть a_{j+1}-сына в T_{i+1} в силу максимальности y.)
3. Полагаем $B_i[a]=0$ для всех $a \in I$. Поскольку a_iya_{j+1} не является подцепочкой цепочки x_{i+1}, то, разумеется, aa_iya_{j+1} не является подцепочкой цепочки x_i ни для какого a.
4. Чтобы построить A_i из A_{i+1}, берем узел u', являющийся a_{j+1}-сыном узла u_y в T_{i+1}. (В T_{i+1} узел u' соответствует ya_{j+1}.) Новый узел i делаем a_i-сыном узла u' в A_{i+1}. (В A_i узел i будет соответствовать цепочке $a_{j+1}y^Ra_i$.)

Связи между узлами, упомянутыми в случае 3(б), показаны на рис. 9.24.

Пример 9.22. Пусть $a_1. \ . \ .a_na_{n+1}=abbabb\$$, а T_2 и A_2 — деревья с рис. 9.22. Алгоритмом 9.5 строим T_1 и A_1 из T_2 и A_2. Здесь $i=1$ и $a_1=a$.

Прежде всего отыскиваем лист с меткой 2 и, поскольку $B_2[a]=0$, полагаем $B_2[a]=1$ и поднимаемся в узел, обозначенный u на рис.

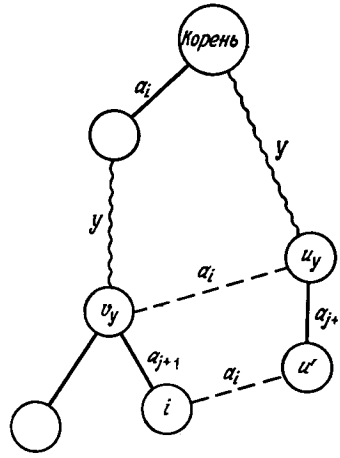


Рис. 9.24. Части дерева T_i, важные для случая 3(б). Двумя штриховыми линиями с метками a_i изображены ребра дерева A_i.

9.22. В узле u находим, что $B_u[a]=1$, поскольку abb — подцепочка цепочки $bbabb\$$. Таким образом, узел u — это u_y. Теперь ищем узел v_y, т. е. a-сына узла u в A_2. Узла v_y нет, так что здесь подходит случай 2.

Узел w, т. е. отец узла u в T_2, не имеет a-сына в A_2. Но узел r, отец узла w, имеет. Им будет узел 4. Поэтому на шаге 1 случая 2 мы находим, что v_1 — узел с меткой 4 в T_2 и A_2. На шаге 2 находим, что $q=2$, $u_1=r$ и $u_2=w$. Кроме того, $c_1=c_2=b$. Поэтому на шаге 3 образуем узел v, являющийся b-сыном узла с меткой 4 в T_2 (в T_1 он утратил свою метку). Образуем также узел v_y и делаем его b-сыном узла v.

На шаге 4 получаем $j=3$. Поскольку бывшая метка k узла v_1 равна 4, заключаем, что $m=6$. Находим $a_{j+1}=a$ и $a_{m+1}=\$$. Поэтому новые узлы с метками 1 и 4 становятся соответственно a-сыном и $-сыном узла v_y. Дерево T_1 изображено на рис. 9.25. Остальные шаги оставляем в качестве упражнения. □

Лемма 9.4. *Если T_{i+1} и A_{i+1} — позиционное и вспомогательное деревья для x_{i+1}, то деревья T_i и A_i, построенные алгоритмом* 9.5,— *позиционное и вспомогательное деревья для x_i.*

Доказательство. Допустим, что T_{i+1} представляет множество S_{i+1} идентификаторов позиций в x_{i+1}, а A_{i+1} — вспомогательное дерево для T_{i+1}. Здесь могут быть две возможности:

1) $S_i=S_{i+1}\cup\{s_i(i)\}$,
2) $S_i=(S_{i+1}-\{s_{i+1}(k)\})\cup\{s_i(k), s_i(i)\}$ для некоторого $i<k\leqslant n$.

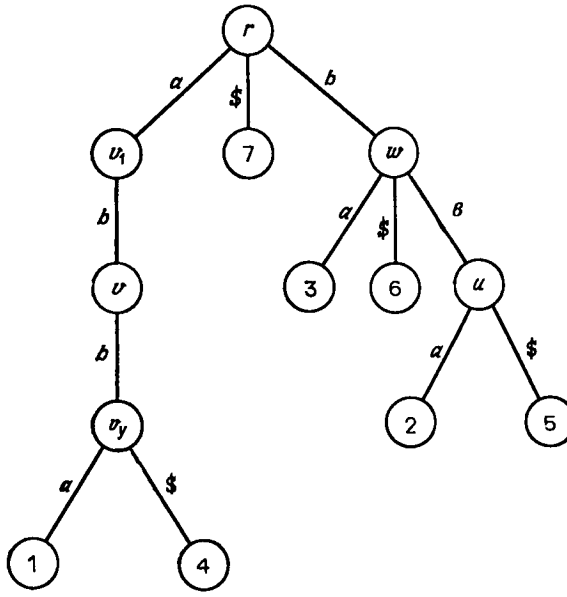


Рис. 9.25. Окончательное позиционное дерево T_1.

Первая возможность покрывается случаями 1 и 3(б) алгоритма 9.5, вторая — случаями 2 и 3(а). Рассуждения, необходимые для доказательства корректности алгоритма 9.5 в случаях 1—3, включены в его описание. □

Таким образом, позиционное дерево для произвольной цепочки можно построить следующим алгоритмом.

Алгоритм 9.6. Построение позиционного дерева

Вход. Цепочка $x\$ = a_1 \ldots a_n a_{n+1}$, где $a_i \in I$ для $1 \leqslant i \leqslant n$ и $a_{n+1} = \$$.

Выход. Позиционное дерево T_i для $x\$$.

Метод.

1. Пусть T_{n+1} — позиционное дерево на рис. 9.26 и $B_r[a] = B_{n+1}[a] = 0$ для всех $a \in I$.

2. Пусть A_{n+1} — вспомогательное дерево, совпадающее с деревом на рис. 9.26.

3. **for** $i \leftarrow n$ **step** -1 **until** 1 **do** алгоритм 9.5 для построения T_i и A_i из T_{i+1} и A_{i+1}. □

Теорема 9.12. *Алгоритм* 9.6 *строит позиционное дерево для цепочки* $x\$$ *за время, пропорциональное числу узлов в окончательном позиционном дереве* T_1.

Рис. 9.26. Начальное позиционное дерево.

Доказательство. На шаге 1 алгоритма 9.5 можно найти лист $i+1$ за фиксированное время с помощью указателя на этот лист, установленного в момент добавления листа к T_{i+1}. Когда к T_i добавляется i, этот указатель переводится на i.

Работа, производимая при каждом выполнении шагов 2 и 3, очевидно, пропорциональна длине пути из $i+1$ в u_y, а при каждом выполнении шага 1 постоянна. Легко проверить, что время, затрачиваемое на выполнение любого из случаев 1—3 алгоритма 9.5, пропорционально числу узлов, добавляемых к дереву. Таким образом, общее время, которое тратится всеми частями алгоритма 9.5, кроме шагов 2 и 3, пропорционально размеру дерева T_1.

Осталось показать, что сумма расстояний между узлами $i+1$ и u_y (или корнем, если узла u_y нет) в T_{i+1} для $1 \leqslant i \leqslant n$ не превосходит размера дерева T_1. Обозначим эти расстояния через $d_2, d_3, \ldots, d_{n+1}$, и пусть e_i, $1 \leqslant i \leqslant n+1$,— глубина узла i в T_i. Простой анализ случаев 1—3 показывает, что

$$e_i \leqslant e_{i+1} - d_{i+1} + 2. \tag{9.1}$$

Просуммируем обе части неравенства (9.1) по i от 1 до n:

$$\sum_{i=2}^{n+1} d_i \leqslant 2n + e_{n+1} - e_1. \tag{9.2}$$

Из (9.2) непосредственно следует, что время, затрачиваемое шагами 2 и 3 алгоритма 9.5, составляет $O(n)$ и, значит, по порядку не превосходит размера дерева T_1. □

Как уже отмечалось, позиционное дерево для цепочки длины n может содержать $O(n^2)$ узлов. Поэтому любой алгоритм идентификации, в котором строится такое дерево, должен иметь временную сложность $O(n^2)$. Однако можно "уплотнить" позиционное и вспомогательное деревья, сжав все цепи в позиционном дереве в один узел. *Цепью* называется путь, каждый узел которого обладает в точности одним сыном. Нетрудно показать, что уплотненное позиционное дерево для цепочек длины n содержит не более $4n-2$ узлов. Уплот-

ненное дерево может давать ту же информацию, что и исходное позиционное дерево, и потому его можно использовать в тех же алгоритмах идентификации.

Алгоритм 9.5 можно модифицировать так, чтобы он строил уплотненные позиционное и вспомогательное деревья за линейное время. Мы не приводим здесь этот модифицированный алгоритм потому, что он аналогичен алгоритму 9.5, а также потому, что во многих приложениях можно ожидать, что размер позиционного дерева будет пропорционален длине входной цепочки. В замечаниях по литературе указаны работы, в которых излагается алгоритм с уплотнением позиционного дерева и другие его приложения.

УПРАЖНЕНИЯ

9.1. Постройте регулярные выражения и графы переходов конечных автоматов для следующих регулярных множеств цепочек в алфавите $I=\{a, b\}$.

(а) Все цепочки, начинающиеся и кончающиеся символом a.
(б) Все цепочки, не содержащие двух символов a подряд.
*(в) Все цепочки, содержащие нечетное число символов a и четное число символов b.
*(г) Все цепочки, не содержащие подцепочки aba.

9.2. Докажите, что множество, допускаемое НКА на рис. 9.1, имеет вид $(a+b)^*aba$.

9.3. Постройте НКА, допускающие регулярные множества

(а) $(a+b)^*(aa+bb)$,
(б) $a^*b^*+b^*a^*$,
(в) $(a+\varepsilon)(b+\varepsilon)(c+\varepsilon)$.

9.4. Покажите, что дополнение регулярного множества регулярно.

***9.5.** Покажите, что множества цепочек

(а) $\{a^nb^n|n\geqslant 1\}$,
(б) $\{ww|w\in\{a, b\}^*\}$,
(в) $\{w|w\in\{a, b\}^*$ и $w=w^R\}$ (т. е. множество палиндромов)

не регулярны.

9.6. Пусть $x=a_1a_2\ldots a_n$ — данная цепочка и α — регулярное выражение. Модифицируйте алгоритм 9.1 так, чтобы он находил наименьшее число k, а по нему (а) наименьшее j и (б) наибольшее j, такое, что $a_ja_{j+1}\ldots a_k$ принадлежит множеству, представленному выражением α. *Указание*: Каждому состоянию из S_i поставьте в соответствие целое число j.

***9.7.** Пусть x и α те же, что и в упр. 9.6. Модифицируйте алгоритм 9.1 так, чтобы он находил такое наименьшее j и по нему наи-

большее k, что $a_j a_{j+1} \ldots a_k$ принадлежит множеству, представленному выражением α.

9.8. Пусть x и α те же, что и в упр. 9.6. Постройте алгоритм, который находил бы все подцепочки в x, принадлежащие множеству, представленному выражением α. Какова временная сложность вашего алгоритма?

***9.9.** Пусть I — алфавит и $\varnothing$ — символ, не принадлежащий I. *Цепочкой с несущественными символами* (ЦСНС) назовем цепочку в алфавите $I \cup \{\varnothing\}$. Будем говорить, что ЦСНС $b_1 b_2 \ldots b_n$ *идентифицируется* с ЦСНС $a_1 a_2 \ldots a_n$ в позиции i, если для всех j, $i \leqslant j \leqslant n$, либо $a_j = b_{j-i+1}$, либо один из символов a_j или b_{j-i+1} есть $\varnothing$. Покажите, что для фиксированного алфавита I проблема определения позиций, в которых одна ЦСНС идентифицируется с другой, эквивалентна по асимптотической временной сложности (*и/или*)-*умножению*, т. е. операции вида $c_j = \bigvee_{i=1}^{j} d_i \wedge e_{j-i+1}$, где все c_i, d_i и e_i булевы

****9.10.** Покажите, что можно определить позиции, в которых одна ЦСНС идентифицируется с другой, за время $O_A(n \log^2 n \log \log n)$. *Указание*: Используйте упр. 9.9 и алгоритм Шёнхаге — Штрассена для умножения целых чисел.

9.11. Напишите алгоритм сложности $O(n)$, который по данным цепочкам $a_1 a_2 \ldots a_n$ и $b_1 b_2 \ldots b_n$ узнавал бы, существует ли такое k, что $a_i = b_{(k+i) \bmod n}$ для $1 \leqslant i \leqslant n$.

9.12. С помощью алгоритма 9.3 постройте машины для цепочек y, равных

(а) *abbbbabbbbabbaba*,

(б) *abcabcab*.

9.13. Докажите теорему 9.8.

9.14. Пусть $S = \{y_1, y_2, \ldots, y_m\}$ — множество цепочек. Напишите алгоритм, который находил бы все вхождения цепочек из S в цепочку-текст x. Какова временная сложность вашего алгоритма?

***9.15.** Постройте 2ДМА, допускающие языки

(а) $\{xcy_1cy_2 \ldots cy_m | m \geqslant 1,\ x \in \{a, b\}^*,\ y_i \in \{a, b\}^*$ для $1 \leqslant i \leqslant m$ и хотя бы одна из цепочек $y_1, y_2, \ldots, y_m$ входит в $x\}$,

(б) $\{xyy^R | x, y \in I^*$ и $|y| \geqslant 1\}$,

(в) $\{x_1 c x_2 \ldots c x_n | x_i \in \{a, b\}^*$ для $1 \leqslant i \leqslant n$ и все x_i различны$\}$,

(г) $\{x_1 c x_2 \ldots c x_k d y_1 c y_2 \ldots c y_l |$ все x_i и y_i принадлежат $\{a, b\}^*$ и $x_i = y_j^R$ для некоторых i и $j\}$.

9.16. Рассмотрим 2ДМА P с правилами:

$$\delta(s_0, a, Z_0) = \delta(s_0, a, A) = (s_0, +1, \textbf{push } A),$$
$$\delta(s_0, \$, A) = (s_1, -1),$$

$$\delta(s_1, a, A) = \delta(s_1, a, Z_0) = (s_1, -1, \text{pop}),$$
$$\delta(s_1, ¢, Z_0) = (s_f, 0).$$

(а) Запишите все поверхностные конфигурации автомата P для входа aa.

(б) Примените алгоритм 9.4 для вычисления массива ТЕРМ.

***9.17.** Предположим, что 2ДМА может из позиции i перейти за один шаг в любую из позиций

(а) $n-i$,
(б) $i/2$,
(в) $n/2$,
(г) $\log n$,

где n — длина входа. Покажите, что ни одна из этих модификаций не увеличивает распознающей силы 2ДМА. Какие операции вы можете добавить, чтобы не увеличить при этом допускающей силы 2ДМА?

****9.18.** Постройте 2ДМА, допускающий язык $\{w_1 w_1^R w_2 w_2^R w_3 w_3^R u \mid w_1, w_2, w_3 \in \{a, b\}^+$ и $u \in \{a, b\}^*\}$. *Указание*: Пусть x — входная цепочка. Напишите подпрограмму для нахождения такой кратчайшей цепочки w_1, что $w_1 w_1^R y = x$. Затем примените вашу подпрограмму к y и т. д.

****9.19.** Постройте алгоритмы, распознающие за линейное время языки

(а) $\{ww^R \mid w \in I^*\}^*$,
(б) $\{ww^R u \mid w, u \in I^*, |w| \geqslant 1\}$,
(в) $\{w_1 w_2 \ldots w_n \mid w_i \in \{a, b\}^*$ и каждая цепочка w_i является непустым палиндромом четной длины$\}$.

9.20. Напишите алгоритм, который по двум цепочкам $a_1 a_2 \ldots a_n$ и $b_1 b_2 \ldots b_p$ отыскивает за линейное время такое наибольшее число l, что $a_1 a_2 \ldots a_l$ — подцепочка цепочки $b_1 b_2 \ldots b_p$. Как с помощью этого алгоритма проверить, является ли данная цепочка палиндромом?

В следующих четырех упражнениях обсуждаются некоторые варианты магазинных автоматов.

***9.21.** *k-головчатым* 2ДМА называют 2ДМА с k независимыми читающими головками на входной ленте. Покажите, как распознать за время $O(n^k)$, допускается ли входная цепочка длины n k-головчатым 2ДМА.

9.22. *Односторонним* магазинным автоматом называют магазинный автомат, никогда не сдвигающий свою входную головку влево. *Недетерминированным* магазинным автоматом (НМА) называют автомат, у которого есть нуль или более возможностей для выбора

очередного шага. Таким образом, различаются четыре семейства магазинных автоматов: 1ДМА, 1НМА, 2ДМА и 2НМА. Язык, допускаемый 1НМА, называется *бесконтекстным*.

(а) Покажите, что $\{wcw^R | w \in \{a, b\}^*\}$ допускается некоторым 1ДМА.

(б) Покажите, что $\{ww^R | w \in \{a, b\}^*\}$ допускается некоторым 1НМА. (Этот язык не может допускаться никаким 1ДМА.)

(в) Покажите, что $\{wwx | w, x \in \{a, b\}^*, |w| \geqslant 1\}$ допускается некоторым 2НМА. (Этот язык не может допускаться никаким 1НМА.)

***9.23.** Покажите, что язык L порождается бесконтекстной грамматикой в нормальной форме Хомского [1]) тогда и только тогда, когда он допускается некоторым 1НМА.

***9.24.** Покажите, что за время $O(n^3)$ можно распознать, допускается ли входная цепочка длины n некоторым 2НМА.

9.25. Постройте позиционные деревья для цепочек

(а) *baaaab*\$,

(б) *abababa*\$.

9.26. Покажите, что позиционное дерево для $a^nb^na^nb^n\$$ содержит n^2+6n+1 узлов.

***9.27.** Покажите, что позиционное дерево для случайной входной цепочки x содержит $O(|x|)$ узлов при условии, что символы во всех позициях выбираются из фиксированного алфавита равновероятно и независимо.

9.28. Для позиционных деревьев из упр. 9.25 найдите

(а) вспомогательные деревья,

(б) двоичные векторы B_v для каждого узла v.

9.29. Покажите, что для каждого позиционного дерева существует вспомогательное дерево.

9.30. Завершите доказательство леммы 9.4, показав, что T_i и A_i правильно строятся по T_{i+1} и A_{i+1}.

***9.31.** Покажите, что с помощью позиционного дерева для x можно проверить, является ли какая-то цепочка из $y_1, y_2, \ldots, y_m$ подцепочкой в x, за время порядка

$$|y_1| + |y_2| + \ldots + |y_m|.$$

9.32. Напишите алгоритм, который находил бы длиннейшую общую подцепочку двух данных цепочек x и y. Какова временная сложность вашего алгоритма?

---

[1]) См. определение перед упр. 2.34.

9.33. Напишите эффективный алгоритм, который по данным цепочкам x и $b_1b_2\ldots b_p$ находил бы для каждого i, $1\leqslant i\leqslant p$, длиннейшую подцепочку в x, являющуюся префиксом цепочки $b_ib_{i+1}\ldots b_p$.

9.34. Напишите эффективный алгоритм, который по данным двум цепочкам $x=a_1a_2\ldots a_n$ и $y=b_1b_2\ldots b_n$ в алфавите I находил бы кратчайшее представление для y в виде $c_1c_2\ldots c_m$, где c_i — символ из I или символ, обозначающий подцепочку цепочки x. Например, если $x=abbabb$ и $y=ababbaa$, то $[1:2][4:6]aa$ — представление для y длины 4 ($[i:j]$ обозначает подцепочку $a_ia_{i+1}\ldots a_j$ цепочки x). *Указание*: Воспользуйтесь упр. 9.33.

9.35. Докажите, что уплотненное позиционное дерево для цепочки длины n содержит не более $4n-2$ узлов.

****9.36.** Напишите алгоритм, который за время $O(n)$ находил бы уплотненное позиционное дерево для цепочки длины n.

9.37. Цепочка $x=a_1a_2\ldots a_n$ называется *подпоследовательностью* цепочки $y=b_1b_2\ldots b_p$, если $a_1a_2\ldots a_n=b_{i_1}b_{i_2}\ldots b_{i_n}$ для некоторых $i_1<i_2<\ldots<i_n$ (т. е. x получается из y вычеркиванием нуля или более символов). Напишите алгоритм сложности $O(|x|\cdot|y|)$ для нахождения самой длинной общей подпоследовательности цепочек x и y.

9.38. Напишите алгоритм, который по двум данным цепочкам x и y находил бы кратчайшую последовательность вставок и удалений одного символа, превращающую x в y.

Проблемы для исследования

9.39. Можно ли улучшить оценку времени в упр. 9.10?

9.40. Необходимо ли время $O(|\alpha|\cdot|x|)$ для распознавания вхождения в x цепочки из языка, представленного регулярным выражением α?

9.41. k-головчатый 2ДМА можно промоделировать на РАМ за время $O(n^k)$ (упр. 9.21). Может ли каждый бесконтекстный язык допускаться некоторым 2-головчатым 2ДМА? Если да, то каждый бесконтекстный язык можно было бы распознать на некоторой РАМ за время $O(n^2)$.

9.42. Существует ли бесконтекстный язык, не допускаемый никаким 2ДМА?

9.43. Существует ли язык, допускаемый некоторым 2НМА, но не допускаемый никаким 2ДМА?

9.44. Можно ли улучшить оценку времени в упр. 9.37? (См. Ахо, Хиршберг, Ульман [1974], где для получения оценок использована модель дерева решений.)

Замечания по литературе

Эквивалентность конечных автоматов и регулярных выражений установлена Клини [1956]. Недетерминированные конечные автоматы изучались Рабиным, Скоттом [1959], которые доказали их эквивалентность детерминированным. Алгоритм идентификации цепочек, задаваемых регулярным выражением (алгоритм 9.1), построен на основе алгоритма Томпсона [1968]. Эренфойхт, Цайгер [1974] обсуждают сложность НКА как устройства для классификации образов. Распознавание вхождения одной цепочки в другую за линейное время (алгоритм 9.3) взято у Морриса, Пратта [1970] [1]). Моделирование 2ДМА за линейное время, а также обобщение в упр. 9.21 принадлежат Куку [1971а]. Свойства 2ДМА изучали Грей, Харрисон, Ибарра [1967]. Моделирование 2НМА за время $O(n^3)$ (упр. 9.24) описано у Ахо, Хопкрофта, Ульмана [1968]. Упр. 9.15(б) принадлежит Честеру, а упр. 9.19(в) — Пратту. Решение упр. 9.19(в) приведено в работе Кнута, Пратта [1971].

Материал разд. 9.5, касающийся позиционных деревьев, а также понятие уплотненного позиционного дерева можно найти у Вайнера [1973]. Там же содержатся решения упр. 9.20 и 9.31—9.36 вместе с некоторыми другими приложениями; см. также Кнут [1973б]. Карп, Миллер, Розенберг [1972] дают несколько алгоритмов идентификации, касающихся повторяющихся цепочек. Упр. 9.9 и 9.10 о совпадении цепочек с несущественными символами принадлежат Фишеру, Патерсону [1974]. Вагнер, Фишер [1974] приводят решение упр. 9.38. В их статье и статье Хиршберга [1973] обсуждаются алгоритмы для задачи об общей подпоследовательности (см. упр. 9.37).

---

[1]) Более сильный результат о распознавании вхождения одной цепочки в другую был получен Ю. В. Матиясевичем в 1969 г. (См. его статью [1971].) Для обычной постановки задачи вопрос о сложности идентификации цепочек символов и близких задач решен в работе Слисенко [1977].— *Прим. перев.*

10 NP-ПОЛНЫЕ ЗАДАЧИ

Сколько вычислений должна требовать задача, чтобы мы сочли ее действительно трудной? Общепринято, что если задачу нельзя решить быстрее, чем за экспоненциальное время, то ее следует рассматривать как безусловно трудно разрешимую. В этой "схеме классификации" задачи, решаемые алгоритмами полиномиальной сложности, будут легко разрешимыми. Но нужно иметь в виду, что, хотя экспоненциальная функция (такая, как 2^n) растет быстрее любой полиномиальной функции от n, для небольших значений n алгоритм, требующий $O(2^n)$ времени, может оказаться эффективнее многих алгоритмов с полиномиально ограниченным временем работы. Например, сама функция 2^n не превосходит n^{10} до значения n, равного 59. Тем не менее скорость роста экспоненциальной функции столь стремительна, что мы будем называть задачу *трудно разрешимой*, если у всех алгоритмов, решающих ее, сложность по меньшей мере экспоненциальна.

В этой главе мы приведем соображения, показывающие, что некоторый класс задач, а именно задач, полных для недетерминированного полиномиального времени (называемых NP-полными), вероятно, содержит только трудно разрешимые задачи. Этот класс включает в себя многие "классические" задачи из комбинаторики (такие, как задачу о коммивояжере, о гамильтоновом цикле, задачу целочисленного линейного программирования), и можно показать, что все задачи из этого класса "эквивалентны" в том смысле, что если хотя бы одна из них оказалась легко разрешимой, то таковы же и все остальные. Поскольку многие из этих задач изучались математиками и специалистами по вычислениям в течение десятилетий и ни для одной из них не удалось найти алгоритма полиномиальной сложности, естественно предположить, что таких полиномиальных алгоритмов и не существует, и, значит, можно рассматривать все задачи из этого класса как трудно разрешимые.

Мы исследуем также еще один класс задач, называемых "полными для полиномиальной емкости", которые по меньшей мере столь же сложны, как NP-полные задачи, но и про них еще не доказано, что

они трудно разрешимы. В гл. 11 будут изложены некоторые задачи, относительно которых действительно можно доказать, что они трудно разрешимы.

10.1. НЕДЕТЕРМИНИРОВАННЫЕ МАШИНЫ ТЬЮРИНГА

По причинам, которые скоро станут ясными, ключевое понятие в теории NP-полных задач — недетерминированная машина Тьюринга [1]). Мы уже обсуждали недетерминированные конечные автоматы, у которых каждый шаг может перевести автомат в несколько различных состояний. По аналогии недетерминированная машина Тьюринга имеет конечное число возможных шагов, из которых в очередной момент выбирается какой-то один. Входная цепочка x допускается, если по крайней мере одна последовательность шагов для входа x приводит к допускающему мгновенному описанию (МО).

При данной входной цепочке x можно считать, что недетерминированная машина Тьюринга M параллельно выполняет все возможные последовательности шагов, пока не достигнет допускающего МО или пока не окажется, что дальнейшие шаги невозможны. Иначе говоря, после i шагов можно считать, что существуют много экземпляров M. Каждый экземпляр представляет МО, в котором M может оказаться после i шагов. На $(i+1)$-м шаге экземпляр C порождает j своих экземпляров, если машина Тьюринга, находясь в МО C, может выбрать следующий шаг j способами.

Таким образом, возможные последовательности шагов машины M на входе x можно организовать в виде дерева мгновенных описаний. Каждый путь из корня в лист этого дерева представляет некоторую последовательность возможных шагов. Если σ — кратчайшая последовательность шагов, которая оканчивается допускающим МО, то, как только машина M сделает $|\sigma|$ шагов, она остановится и допустит вход x. Время, затрачиваемое на обработку входа x, равно длине последовательности σ. Если на входе x никакая последовательность шагов не приводит к допускающему МО, то M отвергает x, а время, затрачиваемое на обработку x, не определено.

Часто удобно представлять себе, что M "угадывает" только шаги из последовательности σ и проверяет, что σ на самом деле оканчивается допускающим МО. Но поскольку детерминированная машина обычно не может заранее угадать допускающую последовательность шагов, то детерминированное моделирование машины M потребовало бы просмотра в некотором порядке дерева всех возможных последовательностей шагов на входе x и этот просмотр продолжался бы до обнаружения кратчайшей последовательности, окан-

[1]) Читателю, недостаточно хорошо знакомому с машинами Тьюринга, предлагаем освежить в памяти разд. 1.6.

чивающейся допускающим МО. Если никакая последовательность шагов не приводит к допусканию входа, то детерминированное моделирование машины M могло бы продолжаться бесконечно долго, если только нет какой-нибудь априорной границы на длину кратчайшей допускающей последовательности. Поэтому естественно ожидать, что недетерминированные машины Тьюринга способны выполнять задания, которые никакие детерминированные машины Тьюринга не могут выполнить за то же время или с той же памятью. Тем не менее все еще открыт важный вопрос о существовании языков, допускаемых недетерминированной машиной Тьюринга с данной временной или емкостной сложностью, но не допускаемых никакой детерминированной машиной Тьюринга с той же сложностью.

Определение. *k-ленточной недетерминированной машиной Тьюринга* (для краткости НМТ) M называют семерку $(Q, T, I, \delta, \mathrm{b}, q_0, q_f)$, где значения всех компонент те же, что и для обычной детерминированной машины Тьюринга, за исключением того, что здесь функция переходов δ представляет собой отображение множества $Q\times T^k$ в множество подмножеств в $Q\times(T\times\{\mathrm{L}, \mathrm{R}, \mathrm{S}\})^k$. (L означает сдвиг головки влево, R — вправо, S — головка остается на месте.) Иными словами, по данному состоянию и списку из k ленточных символов функция δ выдает конечное множество вариантов следующего шага; каждый вариант состоит из нового состояния, k новых ленточных символов и k сдвигов головок. Заметим, что НМТ M может выбрать любой из этих вариантов шага, но не может выбрать следующее состояние из одного шага, а новые ленточные символы — из другого или устроить какую-нибудь еще комбинацию вариантов шага.

Мгновенные описания для НМТ определяются точно так же, как и для детерминированной машины Тьюринга (ДМТ). Данная НМТ $M=(Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ делает *шаг*, исходя из своего текущего состояния, скажем q, и символов, обозреваемых каждой из k головок, скажем $X_1, X_2, \ldots, X_k$. Из множества $\delta(q, X_1, X_2, \ldots, X_k)$ она выбирает некоторый элемент $(r, (Y_1, D_1), \ldots, (Y_k, D_k))$. Этот элемент указывает, что новым состоянием должно стать r, на i-й ленте следует напечатать Y_i вместо X_i и головку сдвинуть в направлении, обозначенном знаком D_i, $1\leqslant i\leqslant k$. Если при некотором выборе следующего шага МО C переходит в МО D, то пишут $C\vdash_M D$ (или $C\vdash D$, если ясно, о какой машине M идет речь). Заметим, что для данной НМТ M может быть несколько таких D, что $C\vdash D$, но если машина M детерминированна, то для каждого C существует не более одного такого D.

Запись $C_1\vdash^* C'$ означает, что для некоторого k выполняется $C_1\vdash C_2\vdash \ldots\vdash C_k=C'$ или $C_1=C'$. НМТ M *допускает* цепочку w, если $(q_0w, q_0, q_0, \ldots, q_0)\vdash^* (\alpha_1, \alpha_2, \ldots, \alpha_k)$, где α_1 (и, значит, $\alpha_2, \ldots$

. . ., α_k) содержит (где-то внутри себя) символ заключительного состояния q_1. *Языком* $L(M)$, *допускаемым машиной* M, называют множество всех цепочек, допускаемых M.

Пример 10.1. Построим НТМ, допускающую такие цепочки вида

$$10^{i_1}10^{i_2}\ldots10^{i_k},$$

что $\sum_{j\in I} i_j = \sum_{j\notin I} i_j$ для некоторого множества $I \subseteq \{1, 2, \ldots, k\}$. Иными словами, цепочка w должна допускаться, если список [1]) целых чисел $i_1, i_2, \ldots, i_k$, представленный ею, можно разбить на два подсписка так, чтобы суммы чисел в них были равны. Эта задача известна как задача о разбиении. Доказано, что она NP-полна, если целые числа i_j представлены своими двоичными кодами и размером задачи считается длина списка этих двоичных целых чисел [2]).

Мы построим трехленточную НМТ M, распознающую этот язык. Она просматривает свою входную ленту слева направо, и каждый раз, когда она доходит до блока вида 0^{i_j}, на вторую или третью ленту добавляются эти i_j нулей, причем выбор ленты недетерминирован. После того как машина дойдет до конца входа, она проверит, совпадают ли числа нулей на лентах 2 и 3, и, если да, допустит вход. Поэтому, если какая-нибудь последовательность выборов, указывающая, в какое из множеств помещать очередное число i_j — в первое (на ленту 2) или во второе (на ленту 3), приводит к равным суммам, то эта НТМ допустит вход. Последовательности шагов, приводящие к неравным цепочкам на лентах 2 и 3, не принимаются во внимание, если хотя бы одна последовательность выборов срабатывает. Формально пусть

$$M = (\{q_0, q_1, \ldots, q_5\}, \{0, 1, \mathrm{b}, \$\}, \{0, 1\}, \delta, \mathrm{b}, q_0, q_5),$$

где функция переходов δ задана на рис. 10.1.

На рис. 10.2 показаны две из многих последовательностей шагов, которые может сделать на входе 1010010 эта НМТ. Первая приводит к допускающему состоянию, вторая нет. Поскольку по крайней мере одна последовательность шагов приводит к допускающему состоянию, эта НМТ принимает 1010010. □

Определение. Говорят, что НМТ M имеет *временную сложность* $T(n)$, если для всякой допускаемой входной цепочки длины n найдется последовательность, состоящая не более чем из $T(n)$ шагов,

---

[1]) Это список, а не множество из-за возможных повторений.

[2]) Как мы увидим, способ кодирования данных задачи очень важен. Нетрудно показать, что если для распознавания используется ДМТ, то время решения задачи из примера 10.1 составляет в действительности $O_{\text{МТ}}(n^2)$, где n — длина входа. Но если вход закодировать двоичными числами, то длина входа будет равна сумме логарифмов чисел $i_1, \ldots, i_k$ и та же стратегия даст алгоритм сложности $O_{\text{МТ}}(c^n)$, где n — это уже длина двоичного входа и $c>1$.

Состояние	Текущий символ на ленте			Новый символ, сдвиг головки на ленте			Новое состояние	Примечания
	1	2	3	1	2	3		
q_0	1	b	b	1, S	\$, R	\$, R	q_1	Машина помечает левые концы лент 2 и 3 знаком \$ и переходит в состояние q_1.
q_1	1	b	b	1, R 1, R	b, S b, S	b, S b, S	q_2 q_3	Здесь машина выбирает, писать очередной блок на ленте 2 (q_2) или 3 (q_3).
q_2	0 1 b	b b b	b b b	0, R 1, S b, S	0, R b, S b, L	b, S b, S b, L	q_2 q_1 q_4	Машина переписывает блок из нулей на ленту 2, затем по достижении единицы на ленте 1 возвращается в состояние q_1. Если же на ленте 1 достигнут символ b, она переходит в состояние q_4, чтобы сравнить длины лент 2 и 3.
q_3	0 1 b	b b b	b b b	0, R 1, S b, S	b, S b, S b, L	0, R b, S b, L	q_3 q_1 q_4	То же, что в состоянии q_2, но запись на ленту 3.
q_4	b b	0 \$	0 \$	b, S b, S	0, L \$, S	0, L \$, S	q_4 q_5	Сравнение длин лент 2 и 3.
q_5								Машина допускает вход.

Рис. 10.1. Переходы (шаги) НМТ. Каждая строка представляет один вариант выбора.

$(q_0 1010010,\ q_0,\ q_0)$	$(q_0 1010010,\ q_0,\ q_0)$
$\vdash (q_1 1010010,\ \$q_1,\ \$q_1)$	$\vdash (q_1 1010010,\ \$q_1,\ \$q_1)$
$\vdash (1q_2 010010,\ \$q_2,\ \$q_2)$	$\vdash (1q_3 010010,\ \$q_3,\ \$q_3)$
$\vdash (10q_2 10010,\ \$0q_2,\ \$q_2)$	$\vdash (10q_3 10010,\ \$q_3,\ \$0q_3)$
$\vdash (10q_1 10010,\ \$0q_1\ \$q_1)$	$\vdash (10q_1 10010,\ \$q_1,\ \$0q_1)$
$\vdash (101q_3 0010,\ \$0q_3,\ \$q_3)$	$\vdash (101q_3 0010,\ \$q_3,\ \$0q_3)$
$\vdash (1010q_3 010,\ \$0q_3,\ \$0q_3)$	$\vdash (1010q_3 010,\ \$q_3,\ \$00q_3)$
$\vdash (10100q_3 10,\ \$0q_3,\ \$00q_3)$	$\vdash (10100q_3 10,\ \$q_3,\ \$000q_3)$
$\vdash (10100q_1 10,\ \$0q_1,\ \$00q_1)$	$\vdash (10100q_1 10,\ \$q_1,\ \$000q_1)$
$\vdash (101001q_2 0,\ \$0q_2,\ \$00q_2)$	$\vdash (101001q_3 0,\ \$q_3,\ \$000q_3)$
$\vdash (1010010q_2,\ \$00q_2,\ \$00q_2)$	$\vdash (1010010q_3,\ \$q_3,\ \$0000q_3)$
$\vdash (1010010q_4,\ \$0q_4 0,\ \$0q_4 0)$	$\vdash (1010010q_4,\ q_4\$,\ \$000q_4 0)$
$\vdash (1010010q_4,\ \$q_4 00,\ \$q_4 00)$	Остановка — нет следующего МО
$\vdash (1010010q_4,\ q_4\$00,\ q_4\$00)$	
$\vdash (1010010q_5,\ q_5\$00,\ q_5\$00)$	
Вход допускается	

Рис. 10.2. Две возможные последовательности шагов для НМТ.

которая приводит в допускающее состояние, и *емкостную сложность* $S(n)$, если для всякой допускаемой входной цепочки длины n найдется последовательность шагов, приводящая в допускающее состояние, в которой число просмотренных головкой клеток на каждой ленте не превосходит $S(n)$.

Пример 10.2. НМТ из примера 10.1 имеет временную сложность $2n+2$ (худший случай — вход из n единиц) и емкостную сложность $n+1$. Другие разумные кодирования задачи о разбиении также дают эту сложность. Например, пусть $B(i)$ — двоичное представление целого числа i и

$$L_1 = \Big\{ \#B(i_1)\#B(i_2)\ldots\#B(i_k) \mid \text{существует такое множество } I \subseteq \{1, 2, \ldots, k\}, \text{ что } \sum_{i \in I} i_i = \sum_{j \in I} i_j \Big\},$$

где $\#$ — разделительный маркер. Чтобы распознать L_1, можно построить новую НМТ M_1, работа которой аналогична работе НМТ с рис. 10.1. Отличие состоит в том, что вместо копирования нулей на ленту 2 или 3 M_1 запоминает на лентах двоичные числа. Каждое новое двоичное число, встречающееся во входной цепочке, прибавляется к числу на той или другой ленте.

Чтобы обработать список целых чисел $i_1, i_2, \ldots, i_k$, машина M_1 просматривала бы входную цепочку $x=\#B(i_1)\#B(i_2)\ldots\#B(i_k)$. Машина M при решении той же задачи просматривала бы входную цепочку $w=10^{i_1}10^{i_2}\ldots10^{i_k}$, которая может оказаться экспоненциально длиннее цепочки x. Таким образом, хотя M_1 в некотором смысле экспоненциально быстрее, чем НМТ на рис. 10.1, ее временная и емкостная сложности, поскольку вход соответственно укорочен, все равно равны $O(n)$, где n — длина входа. □

С помощью моделирования можно показать, что любой язык, допускаемый какой-нибудь НМТ, допускается также и некоторой ДМТ, но за это, по-видимому, придется расплачиваться сильным увеличением временной сложности. Наименьшая верхняя граница, которую можно установить для такого моделирования, экспоненциальна. Другими словами, если $T(n)$ — разумная функция, задающая временную сложность (разумная в том смысле, что она "конструируема по времени", этот термин определяется в гл. 11), то для каждой НМТ M, время работы которой ограничено функцией $T(n)$, можно найти такие постоянную c и ДМТ M', что $L(M)=L(M')$ и время работы машины M' составляет $O(c^{T(n)})$.

Этот результат можно доказать, построив ДМТ M', моделирующую машину M путем полного перебора возможных вычислений. Существует постоянная d, ограничивающая число выборов очередного шага машины M во всех ситуациях. Поэтому последовательность, содержащую вплоть до $T(n)$ шагов работы машины M, можно представить цепочкой в алфавите $\Sigma=\{0, 1, \ldots, d-1\}$, длина которой не превосходит $T(n)$. M' моделирует поведение машины M на входе x длины n следующим образом. M' последовательно порождает в лексикографическом порядке все цепочки из Σ^*, длины которых не превосходят $T(n)$. Таких цепочек не более $(d+1)^{T(n)}$. Когда новая цепочка w порождена, машина M' моделирует последовательность σ_w шагов машины M, представленную цепочкой w. Если σ_w приводит к тому, что M допускает x, то M' также допускает x. Если σ_w не представляет возможную последовательность шагов машины M или σ_w не приводит к допусканию x машиной M, то M' повторяет этот процесс со следующей цепочкой из Σ^*.

M' может промоделировать последовательность σ_w за время $O(T(n))$. Порождение очередной цепочки w занимает $O(T(n))$ шагов. Поэтому все моделирование работы НМТ M может занять время $O(T(n)(d+1)^{T(n)})$, которое не превосходит $O(c^{T(n)})$ для некоторой постоянной c. Детали моделирования оставляем в качестве упражнения.

Следует, однако, подчеркнуть, что не известно никаких нетривиальных нижних оценок, касающихся сложности моделирования НМТ с помощью ДМТ. А именно не известен никакой язык L, который может допустить какая-то НМТ с временной сложностью

$T(n)$, но про который можно доказать, что он не допускается никакой ДМТ с временной сложностью $T(n)$. Ситуация, связанная с емкостной сложностью, много приятнее.

Определение. Функцию $S(n)$ назовем *конструируемой по емкости (памяти)*, если некоторая ДМТ M, начав работу над данным входом длины n, поместит специальный маркер на $S(n)$-ю клетку одной из своих лент, просмотрев не более $S(n)$ клеток на каждой ленте. Самые обычные функции, например полиномы с целыми коэффициентами, 2^n, $n!$, $\lceil n \log(n+1) \rceil$, конструируемы по емкости.

Можно показать, что если $S(n)$ — конструируемая по емкости функция, а M — НМТ с емкостной сложностью, ограниченной функцией $S(n)$, то найдется такая ДМТ M' с емкостной сложностью $O(S^2(n))$, что $L(M)=L(M')$.

Одна из стратегий, с помощью которой M' может моделировать машину M, заключается в интересном приложении приема "разделяй и властвуй". Если емкостная сложность k-ленточной НМТ $M=(Q, T, I, \delta, b, q_0, q_f)$ ограничена функцией $S(n)$, конструируемой по емкости, то для некоторой постоянной c число различных МО, в которые M может попасть, начав работу с входа длины n, не превосходит $c^{S(n)}$, а точнее $\|Q\| \cdot (\|T\|+1)^{kS(n)} \cdot (S(n))^k$, где первый множитель представляет число состояний, второй ограничивает число возможных состояний лент, а последний — число возможных положений головок. Поэтому если $C_1 \vdash_M^* C_2$, то M может дойти из МО C_1 в МО C_2 по некоторому пути, состоящему не более чем из $c^{S(n)}$ шагов. Можно выяснить, осуществим ли переход $C_1 \vdash_M^* C_2$ за $2i$ шагов, проверив для всех C_3, осуществимы ли переходы $C_1 \vdash_M^* C_3$ и $C_3 \vdash_M^* C_2$ за i шагов.

Таким образом, стратегия, на которой основана работа M', состоит в том, чтобы установить с помощью описанного ниже алгоритма, приведет ли начальное МО C_0 к какому-нибудь допускающему МО.

Алгоритм 10.1. Детерминированное моделирование НМТ

Вход. НМТ M с границей $S(n)$ на емкостную сложность, где $S(n)$ — конструируемая по емкости функция, и входная цепочка w длины n.

Выход. "Да", если $w \in L(M)$, и "нет" в противном случае.

Метод. Рекурсивная процедура ПРОВЕРКА(C_1, C_2, i), приведенная на рис. 10.3, распознает, осуществим ли переход $C_1 \vdash_M^* C_2$ за i шагов. Если да, то она принимает значение **true**, в противном случае **false**. В этом алгоритме C_1 и C_2 обозначают МО, в которых на каждой ленте использовано не более $S(n)$ клеток.

Полный алгоритм состоит в вызове ПРОВЕРКА(C_0, C_f, $c^{S(n)}$) для каждого допускающего МО C_f, где C_0 — начальное МО машины M для входа w. Если обнаружится, что один из таких

```
procedure ПРОВЕРКА (C1, C2, i):
if i = 1 then
  if C1 ⊢ C2 или C1 = C2 then return true
  else return false
else
  begin
    for каждого МО C3, в котором на каждой ленте использовано не более S(n) клеток do
      if ПРОВЕРКА (C1, C3, ⌈i/2⌉) и ПРОВЕРКА (C3, C2, ⌊i/2⌋)
        then return true;
    return false
  end
```

Рис. 10.3. Процедура ПРОВЕРКА.

вызовов выдал значение **true**, то ответом алгоритма будет "да", в противном случае "нет". □

Теорема 10.1. *Если M — НМТ с конструируемой по емкости емкостной сложностью $S(n)$, то найдется такая ДМТ M' с емкостной сложностью $O(S^2(n))$, что $L(M)=L(M')$.*

Доказательство. Доказательство состоит в реализации алгоритма 10.1 на машине Тьюринга. Мы знаем из разд. 2.5, как промоделировать на РАМ рекурсивную процедуру ПРОВЕРКА. Ту же стратегию со стеком можно применить на машине Тьюринга. Поскольку аргументами процедуры ПРОВЕРКА, занимающими существенную память, служат МО длины $O(S(n))$, то на ленте надо расположить фрагменты стека того же размера; это можно сделать потому, что функция $S(n)$ конструируема по емкости. Анализируя процедуру ПРОВЕРКА, видим, что при каждом ее рекурсивном вызове третий аргумент уменьшается, по существу, вдвое. Поэтому можно показать, что в любой момент число фрагментов стека не превосходит $1+\log\lceil c^{S(n)}\rceil$, т. е. $O(S(n))$. Поскольку на каждый фрагмент приходится $O(S(n))$ ячеек, всего для стека наша машина Тьюринга использует $O(S^2(n))$ клеток. Остальные детали конструкции машины Тьюринга оставляем в качестве упражнения. □

Чтобы упростить доказательства, часто желательно ограничиться одноленточными машинами Тьюринга. Следующая лемма позволяет это сделать, если мы готовы платить увеличением времени вычислений.

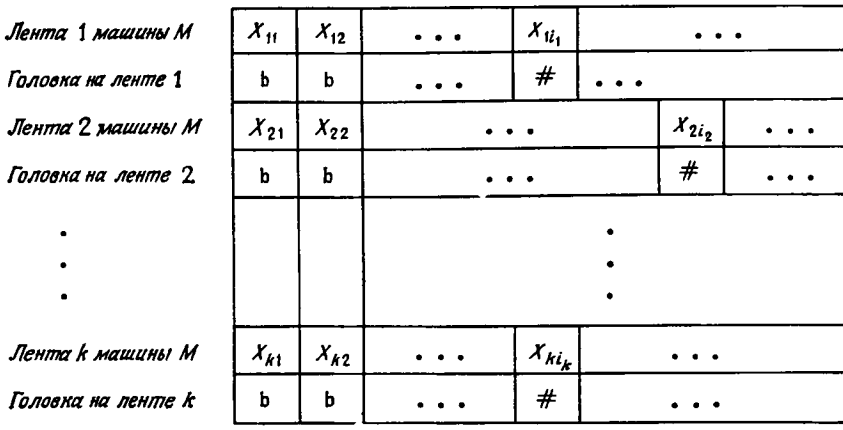


Рис. 10.4. Лента машины M_1.

Лемма 10.1. *Если язык L допускается k-ленточной* НМТ $M=(Q, T, I, \delta, \mathrm{b}, q_0, q_1)$ *с временной сложностью $T(n)$, то он допускается одноленточной* НМТ *с временной сложностью $O(T^2(n))$.*

Доказательство. Построим одноленточную НМТ M_1, допускающую L с временной сложностью $O(T^2(n))$. Представим себе (в соответствии с применяемой техникой моделирования), что лента машины M_1 имеет $2k$ "дорожек" (рис. 10.4). Другими словами, ленточными символами машины M_1 являются $2k$-членные кортежи, в которых на нечетных местах стоят символы алфавита T, а на четных — либо символ пробела b, либо специальный маркер $\#$. Дорожки с нечетными номерами соответствуют k лентам машины M, а каждая дорожка с четным номером содержит символ b во всех клетках, кроме одной, в которой стоит $\#$. Символ $\#$ на дорожке с номером $2j$ отмечает положение головки машины M на ленте j, которой соответствует дорожка с номером $2j-1$. На рис. 10.4 головка j-й ленты обозревает клетку i_j на j-й ленте для всех j, $1 \leqslant j \leqslant k$.

M_1 моделирует один шаг машины M следующим образом. Предположим, что вначале головка машины M_1 обозревает клетку, содержащую самую левую головку машины M.

1. Головка машины M_1 движется вправо, пока не минует все k маркеров положений головок на дорожках с четными номерами. Во время этого движения M_1 запоминает в своем состоянии символы, обозреваемые каждой из головок машины M.
2. Действия машины M_1 на шаге 1 детерминированны. Теперь M_1 делает недетерминированное "разветвление". А именно, исходя из состояния машины M, которое машина M_1 запомнила в своем состоянии, и из символов на лентах, обозреваемых ма-

шиной M, которые M_1 также нашла, машина M_1 недетерминированно выбирает возможный шаг для M. Для каждого шага, возможного для M в этой ситуации, машина M_1 имеет состояние, в которое может перейти на следующем шаге.

3. Выбрав для моделирования шаг машины M, машина M_1 изменяет в соответствии с ним состояние машины M, которое она помнит в своем состоянии. Затем M_1 сдвигает свою головку влево, пока не пройдет через все k маркеров головок. Всякий раз, когда M_1 находит такой маркер, она изменяет ленточный символ, стоящий на дорожке над маркером, и сдвигает этот маркер не более чем на одну клетку влево или вправо в соответствии с выбранным шагом.

В этот момент машина M_1 промоделировала один шаг работы M. Ее головка находится правее левого маркера головок не больше чем на две клетки, так что можно найти этот маркер и повторить цикл.

M_1 может допустить вход, когда его допускает M, поскольку M_1 помнит состояние машины M. Если M допускает цепочку w длины n, то делает это с помощью последовательности, состоящей не более чем из $T(n)$ шагов. Очевидно, в последовательности из $T(n)$ шагов головки машины M не могут разойтись больше, чем на $T(n)$ клеток, и, значит, M_1 может смоделировать один шаг этой последовательности не более чем за $O(T(n))$ своих шагов. Таким образом, M_1 допускает цепочку w, выполняя не более $O(T^2(n))$ шагов. Отсюда следует, что M_1 допускает язык L и имеет временную сложность $O(T^2(n))$. □

Следствие 1. *Если язык L допускается k-ленточной* ДМТ *с временной сложностью $T(n)$, то он допускается однолен точной* ДМТ *с временной сложностью $O(T^2(n))$.*

Доказательство. Если в приведенном выше доказательстве машина M детерминированна, то M_1 тоже будет детерминированной. □

Следствие 2. *Если язык L допускается k-ленточной* НМТ *с емкостной сложностью $S(n)$, то он допускается однолен точной* НМТ *с емкостной сложностью $S(n)$.*

Следствие 3. *Если язык L допускается k-ленточной* ДМТ *с емкостной сложностью $S(n)$, то он допускается однолен точной* ДМТ *с емкостной сложностью $S(n)$.*

10.2. КЛАССЫ $\mathcal{P}$ И $\mathcal{NP}$

Введем два важных класса языков.

Определение. Определим $\mathcal{P}$-TIME как множество всех языков, допускаемых ДМТ с полиномиальной временной сложностью, т. е.

$\mathcal{P}$-TIME $= \{L \mid$ существуют такие ДМТ M и полином $p(n)$, что временная сложность машины M равна $p(n)$ и $L(M) = L\}$.

Определим $\mathcal{NP}$-TIME как множество всех языков, допускаемых НМТ с полиномиальной временной сложностью. Для краткости мы будем часто писать $\mathcal{P}$ и $\mathcal{NP}$ вместо $\mathcal{P}$-TIME и $\mathcal{NP}$-TIME соответственно.

Прежде всего заметим, что, хотя классы $\mathcal{P}$ и $\mathcal{NP}$ определены в терминах машин Тьюринга, можно было бы использовать любую из многих других моделей вычислений. Интуитивно можно представлять себе $\mathcal{P}$ как класс языков, распознаваемых за полиномиальное время. Например, мы показали, что если язык L допускается машиной Тьюринга с временной сложностью $T(n)$, то временная сложность его распознавания на РАМ или РАСП при логарифмическом весовом критерии будет лежать между $k_1 T(n)$ и $k_2 T^4(n)$, где k_1 и k_2 — некоторые положительные постоянные. Таким образом, язык L допускается машиной Тьюринга с полиномиальной временной сложностью тогда и только тогда, когда существует алгоритм его распознавания, реализуемый на РАМ или РАСП с полиномиальной сложностью при логарифмическом весовом критерии.

Можно также определить *недетерминированную* РАМ или РАСП, добавив к системе команд команду CHOICE(L_1, L_2,. . ., L_k), означающую, что недетерминированны выбор и последующее выполнение одного из операторов L_1, L_2,. . ., L_k. Таким образом, класс $\mathcal{NP}$ можно также определить с помощью недетерминированных РАМ или РАСП с полиномиально ограниченной временной сложностью при логарифмическом весовом критерии.

Следовательно, можно представить себе недетерминированную вычислительную машину вроде РАМ или РАСП, способную выполнить много различных возможных последовательностей шагов, начинающихся с данного МО. Оказывается, такое устройство может распознать за полиномиальное время многие языки, по-видимому нераспознаваемые за полиномиальное время детерминированными алгоритмами. Разумеется, любая попытка прямого моделирования недетерминированного устройства N детерминированным устройством D, выполняющим все возможные последовательности шагов, занимает гораздо больше времени, чем любая единичная реализация последовательности шагов устройства N, поскольку D должно прослеживать работу огромного количества копий N. На основе результатов предыдущего раздела мы можем утверждать лишь, что если язык L принадлежит $\mathcal{NP}$, то он допускается некоторой ДМТ с временной сложностью $k^{p(n)}$, где k — постоянная и p — полином, зависящие от L.

С другой стороны, еще никому не удалось доказать, что существует язык из $\mathcal{NP}$, не принадлежащий $\mathcal{P}$. Таким образом, неизвестно, является ли $\mathcal{P}$ собственным подклассом класса $\mathcal{NP}$. Однако можно

доказать, что некоторые языки не менее "трудны", чем любой язык из $\mathcal{NP}$, в том смысле, что если бы у нас был детерминированный алгоритм, распознающий один из этих "не менее трудных" языков за полиномиальное время, то можно было бы для каждого языка из $\mathcal{NP}$ найти детерминированный алгоритм, распознающий его за полиномиальное время. Такие языки называются NP-полными.

Определение. Язык L_0 из $\mathcal{NP}$ называется *полным для недетерминированного полиномиального времени* (или NP-*полным*), если выполнено следующее условие: по данному детерминированному алгоритму распознавания L_0 с временной сложностью $T(n) \geqslant n$ и любому языку L из $\mathcal{NP}$ можно эффективно найти детерминированный алгоритм, распознающий L за время $T(p_L(n))$, где p_L — некоторый полином, зависящий от L. Говорят, что L *сводится к* L_0.

NP-полноту языка L_0 можно доказать, показав, что L_0 принадлежит $\mathcal{NP}$ и каждый язык $L \in \mathcal{NP}$ можно "полиномиально трансформировать" в L_0.

Определение. Язык L называется *полиномиально трансформируемым* в L_0, если некоторая детерминированная машина Тьюринга M с полиномиально ограниченным временем работы преобразует каждую цепочку w в алфавите языка L в такую цепочку w_0 в алфавите языка L_0, что $w \in L$ тогда и только тогда, когда $w_0 \in L_0$.

Если язык L трансформируем в L_0 и L_0 распознается некоторым детерминированным алгоритмом A с временной сложностью $T(n) \geqslant n$, то можно выяснить принадлежность цепочки w языку L, преобразовав w в w_0 с помощью машины M и затем применив A для выяснения принадлежности w_0 языку L_0. Если время работы машины M ограничено полиномом $p(n)$, то $|w_0| \leqslant p(|w|)$. Таким образом, существует алгоритм, выясняющий принадлежность w языку L за время $p(|w|)+T(p(|w|)) \leqslant 2T(p(|w|))$. Если бы функция T была полиномом (т. е. язык L_0 принадлежал бы $\mathcal{P}$), то этот алгоритм, распознающий L, работал бы полиномиально ограниченное время, и язык L также принадлежал бы $\mathcal{P}$.

Некоторые авторы, действительно, определяют язык L_0 как NP-полный, если он принадлежит $\mathcal{NP}$ и каждый язык из $\mathcal{NP}$ полиномиально трансформируем в L_0. Это определение представляется более узким, чем предыдущее, хотя не известно, приводят ли указанные два определения NP-полных языков к разным классам. Определение, основанное на сводимости, означает, что если M_0 — детерминированная машина Тьюринга, распознающая NP-полный язык L_0 за время $T(n)$, то всякий язык из $\mathcal{NP}$ можно распознать за время $T(p(n))$, где p — некоторый полином, с помощью детерминированной машины Тьюринга, которая обращается к M_0 как к подпрограмме нуль или более раз. Определение, основанное на трансфор-

мируемости, означает, что к M_0 можно обратиться лишь один раз и затем использовать результат ее работы заранее фиксированным образом. Хотя мы примем более широкое определение, все наши доказательства NP-полноты проходят и для узкого определения.

Какое бы из этих определений ни взять, ясно, что если некоторый детерминированный алгоритм распознает L_0 за полиномиальное время, то все языки из $\mathcal{NP}$ можно распознать за полиномиальное время. Таким образом, либо все NP-полные языки принадлежат $\mathcal{P}$, либо ни один из них не принадлежит $\mathcal{P}$. Первое верно тогда и только тогда, когда $\mathcal{P}=\mathcal{NP}$. К сожалению, в настоящее время мы можем лишь выдвинуть гипотезу о том, что $\mathcal{P}$ — собственный подкласс класса $\mathcal{NP}$.

10.3. ЯЗЫКИ И ЗАДАЧИ

Мы определили $\mathcal{P}$ и $\mathcal{NP}$ как классы языков. Причина этого двоякая. Во-первых, это упрощает систему обозначений. Во-вторых, задачи из разных областей, таких, как теория графов и теория чисел, часто можно сформулировать как задачи распознавания языков. Например, рассмотрим задачу, которая при каждом значении входных данных требует ответа "да" или "нет". Можно закодировать все возможные значения входных данных такой задачи в виде цепочек и переформулировать ее как задачу о распознавании языка, состоящего из всех цепочек, представляющих входные данные нашей задачи, которые приводят к ответу "да". Такие способы кодирования уже встречались нам в гл. 9, где задачи идентификации формулировались в терминах задач распознавания языков. Однако надо аккуратно выбирать эти кодирования, потому что от них может зависеть временная сложность задачи.

Чтобы связь задача — язык стала более явной, зададим единые "стандартные" представления задач. В частности, примем следующие соглашения.

1. Целые числа будем представлять в десятичной системе счисления.
2. Узлы графа будем представлять целыми числами 1, 2,..., n, закодированными как десятичные числа в соответствии с соглашением 1. Ребро будем представлять цепочкой (i_1, i_2), где i_1 и i_2 — десятичные представления пары узлов, определяющей это ребро.
3. Булевы выражения (формулы) с n пропозициональными переменными будем представлять цепочками, в которых $*$ представляет "и", $+$ представляет "или", $\neg$ представляет "не" [1]), а целые числа 1, 2,..., n представляют пропозицио-

[1]) Мы часто будем писать $\bar{\alpha}$ вместо $\neg(\alpha)$. Если α состоит из единственного *литерала* (переменной или ее отрицания), то скобки можно опускать.

нальные переменные. Знак *, когда можно, опускается. Если нужно, будем ставить скобки.

Теперь мы можем сказать, что задача принадлежит $\mathcal{P}$ или $\mathcal{NP}$, если ее стандартный код принадлежит $\mathcal{P}$ или $\mathcal{NP}$ соответственно.

Пример 10.3. Рассмотрим задачу о клике для неориентированных графов. *k-кликой* в графе G называют k-узельный полный подграф (каждая пара различных узлов в этом подграфе соединена ребром) графа G. Задача о клике формулируется так: содержит ли данный граф G k-клику, где k — данное целое число?

Пример задачи о клике для графа G на рис. 10.5 при $k=3$ можно закодировать цепочкой

$$3\ (1,2)\ (1,4)\ (2,3)\ (2,4)\ (3,4)\ (3,5)\ (4,5).$$

Первое число представляет значение k. За ним идут пары узлов, соединенных ребрами, причем узел v_i представлен числом i. Таким образом, ребра в порядке перечисления выглядят так: (v_1, v_2), (v_1, v_4),. . ., (v_4, v_5).

Язык L, представляющий задачу о клике, образуют такие цепочки вида

$$k\ (i_1, j_1)\ (i_2, j_2) \ldots (i_m, j_m),$$

что граф с ребрами (i_r, j_r), $1 \leqslant r \leqslant m$, содержит k-клику. Задачу о клике могут представлять и другие языки. Например, можно было бы потребовать, чтобы постоянная k стояла за, а не перед графом. Или можно было бы использовать двоичные числа вместо десятичных. Но для любых двух таких языков должен существовать такой полином p, что цепочку w, представляющую частный случай задачи о клике в одном языке, можно за время $p(|w|)$ преобразовать в цепочку, представляющую тот же частный случай этой задачи в другом языке. Таким образом, когда речь идет о принадлежности классу $\mathcal{P}$ или $\mathcal{NP}$, точный выбор языка для представления задачи о клике неважен, если применяется "стандартное" кодирование. □

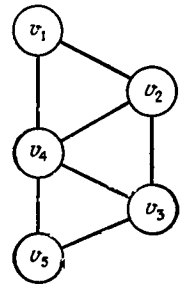


Рис. 10.5. Неориентированный граф.

Задача о клике принадлежит классу $\mathcal{NP}$. Недетерминированная машина Тьюринга сначала может "догадаться", какие k узлов составляют полный подграф, а затем проверить, что любая пара этих узлов соединена ребром, при этом для проверки достаточно $O(n^3)$ шагов, где n — длина кода задачи о клике. Это демонстрирует "силу" недетерминизма, ибо все подмножества из k узлов проверяются "параллельно" независимыми экземплярами распознающего устройства. Граф на рис. 10.5 содержит три 3-клики, а именно $\{v_1, v_2, v_4\}$, $\{v_2, v_3, v_4\}$ и $\{v_3, v_4, v_5\}$.

Мы увидим позже, что задача о клике NP-полна. В настоящее время не известны способы решения задачи о клике за детерминированное полиномиальное время.

Пример 10.4. Булеву формулу $(p_1+p_2)*p_3$ можно представить цепочкой $(1+2)\,3$, где число i представляет переменную p_i. Рассмотрим язык L, образованный всеми цепочками, представляющими *выполнимые* булевы формулы (т. е. формулы, принимающие значение 1 на некотором наборе 0-1-значений переменных). Мы утверждаем, что L принадлежит $\mathcal{NP}$. Недетерминированный алгоритм, распознающий L, начинает с того, что "угадывает" выполняющий набор 0-1-значений пропозициональных переменных во входной цепочке, если такой набор существует. Затем угаданное значение (0 или 1) каждой переменной подставляется вместо всех вхождений этой переменной во входную цепочку. Чтобы закрыть дыры, образующиеся в цепочке в результате подстановки одного символа 0 или 1 вместо десятичного представления пропозициональной переменной, потребуются некоторые сдвиги цепочки. Далее вычисляется значение полученного выражения, чтобы проверить его совпадение с 1.

Это вычисление осуществляется за время, пропорциональное длине выражения, многими алгоритмами синтаксического анализа (см. Ахо, Ульман [1972]). Но даже и без них не трудно вычислить значение булевой формулы за $O(n^2)$ шагов. Следовательно, некоторая недетерминированная машина Тьюринга допускает выполнимые булевы формулы с полиномиальной временной сложностью, и потому задача распознавания выполнимости булевых формул принадлежит $\mathcal{NP}$. Снова отметим, что недетерминизм пригодился, чтобы "параллелизовать" задачу, поскольку "угадывание" правильного решения в действительности означает параллельную проверку всех возможных решений. Как будет показано позже, эта задача также NP-полна. □

Часто нас интересуют задачи оптимизации, например нахождение наибольшей клики в графе. Многие такие задачи можно преобразовать за полиномиальное время в задачи распознавания языка. Например, наибольшую клику в графе G можно найти так:

Пусть n — длина представления графа G. Для каждого k от 1 до n выясняем, содержит ли граф клику размера k. Установив таким образом размер m наибольшей клики, удаляем по одному узлу до тех пор, пока удаление очередного узла v не разрушит все оставшиеся клики размера m. Затем рассмотрим подграф G', состоящий из всех узлов графа G, смежных с v. Рекурсивно вызывая этот процесс, находим клику C размера m—1 в G'. Наибольшей кликой для G будет $C \cup v$.

Предлагаем читателю убедиться в том, что время нахождения наибольшей клики таким методом полиномиально зависит от длины n представления графа G и от времени выяснения, содержит ли граф G клику размера k.

С помощью двоичного поиска (разд. 4.3) часто можно решить задачу оптимизации вида "найти такое наибольшее k, что $P(k)$ истинно", где P — некоторое условие, а число допустимых k оценивается экспонентой от длины n описания задачи. Если из истинности $P(k)$ следует истинность $P(i)$ для $i<k$, а k изменяется от 0 до c^n, где c — некоторая постоянная, то наибольшее k, для которого истинно $P(k)$, можно найти двоичным поиском, проведя $\log c^n = n \log c$ проверок условия $P(k)$. Читателю снова предлагаем убедиться в том, что оптимальное значение k можно найти за время, полиномиально зависящее от n и от наибольшего времени проверки $P(k)$.

Разработку техники подобного рода предоставляем изобретательности читателя и в дальнейшем будем заниматься только задачами, требующими ответа "да" или "нет".

10.4. NP-ПОЛНОТА ЗАДАЧИ ВЫПОЛНИМОСТИ

Можно показать, что задача, а точнее ее представление в виде языка L_0, NP-полна, доказав, что L_0 принадлежит $\mathcal{NP}$ и всякий язык из $\mathcal{NP}$ полиномиально трансформируем в L_0. Благодаря "силе" недетерминизма принадлежность данной задачи классу $\mathcal{NP}$ обычно доказывается легко. Примеры 10.3 и 10.4 служат типичными иллюстрациями этого шага. Трудности вызывает доказательство того, что всякая задача из $\mathcal{NP}$ полиномиально трансформируема в данную задачу. Однако, раз уж установлена NP-полнота некоторой задачи L_0, можно доказать NP-полноту новой задачи L, показав, что L принадлежит $\mathcal{NP}$ и задача L_0 полиномиально трансформируема в L.

Мы покажем, что задача выполнимости булевых формул NP-полна. Затем докажем NP-полноту некоторых фундаментальных задач из пропозиционального исчисления и теории графов, показав, что они принадлежат $\mathcal{NP}$ и в них полиномиально трансформируема задача выполнимости (или какая-нибудь задача, NP-полнота которой уже доказана).

Для изучения важных NP-полных задач, касающихся неориентированных графов, нам понадобятся следующие определения.

Определения. Пусть $G=(V, E)$ — неориентированный граф.

1. *Узельным покрытием* графа G называется такое подмножество $S \subseteq V$, что каждое ребро графа G инцидентно некоторому узлу из S.
2. *Гамильтоновым циклом* называется цикл в графе G, содержащий все узлы из V.
3. Граф G называется *k-раскрашиваемым*, если существует такое приписывание целых чисел $1, 2, \ldots, k$, называемых *цветами*, узлам графа G, что никаким двум смежным узлам не приписан один и тот же цвет. *Хроматическим числом* графа G называется такое наименьшее число k, что граф G k-раскрашиваем.

Для представления важных NP-полных задач об ориентированных графах нам понадобятся следующие определения.

Определения. Пусть $G=(V, E)$ — ориентированный граф.

1. *Множеством узлов, разрезающих циклы*, называется такое подмножество $S \subseteq V$, что каждый цикл в G содержит узел из S.
2. *Множеством ребер, разрезающих циклы*, называется такое подмножество $F \subseteq E$, что каждый цикл в G содержит ребро из F.
3. *Ориентированным гамильтоновым циклом* называется цикл в графе G, содержащий все узлы из V.

Теорема 10.2. *Следующие задачи принадлежат классу* $\mathcal{NP}$:

1. (*Выполнимость*) Выполнима ли данная булева формула?
2. (*Клика*) Содержит ли данный неориентированный граф k-клику?
3. (*Узельное покрытие*) Имеет ли данный неориентированный граф узельное покрытие размера k?
4. (*Гамильтонов цикл*) Имеет ли данный неориентированный граф гамильтонов цикл?
5. (*Раскрашиваемость*) Является ли данный неориентированный граф k-раскрашиваемым?
6. (*Множество узлов, разрезающих циклы*) Имеет ли данный ориентированный граф k-элементное множество узлов, разрезающих циклы?
7. (*Множество ребер, разрезающих циклы*) Имеет ли данный ориентированный граф k-элементное множество ребер, разрезающих циклы?
8. (*Ориентированный гамильтонов цикл*) Имеет ли данный ориентированный граф ориентированный гамильтонов цикл?
9. (*Покрытие множествами*) Существует ли для данного семей-

ства множеств $S_1, S_2, \ldots, S_n$[1]) такое подсемейство из k множеств $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$, что

$$\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{j=1}^{n} S_j?$$

10. *(Точное покрытие)* Существует ли для данного семейства множеств $S_1, S_2, \ldots, S_n$ подсемейство попарно непересекающихся множеств, являющееся покрытием?

Доказательство. Принадлежность задач 1 и 2 классу $\mathcal{NP}$ показана соответственно в примерах 10.4 и 10.3. Аналогично для каждой из остальных задач надо построить недетерминированную машину Тьюринга (или, если читателю угодно, недетерминированную РАМ) полиномиальной сложности, которая "угадывает" решение и проверяет, что это действительно решение. Детали оставляем в качестве упражнения. □

Докажем, что всякий язык из $\mathcal{NP}$ полиномиально трансформируем в задачу выполнимости; тем самым будет установлено, что задача выполнимости булевых формул NP-полна.

Теорема 10.3. *Задача выполнимости булевой формулы* NP-*полна.*

Доказательство. Мы уже знаем, что задача выполнимости принадлежит $\mathcal{NP}$. Осталось показать, что всякий язык L из $\mathcal{NP}$ полиномиально трансформируем в задачу выполнимости. Пусть M — недетерминированная машина Тьюринга, распознающая L за полиномиальное время, и пусть на ее вход подана цепочка w. Из M и w можно построить булеву формулу w_0, выполнимую тогда и только тогда, когда M допускает w. Основная трудность доказательства — показать, что для каждой машины M найдется алгоритм, который построит булеву формулу w_0 из w за время, ограниченное полиномом. Этот полином зависит от M.

В силу леммы 10.1 каждый язык из $\mathcal{NP}$ распознается недетерминированной одноленточной машиной Тьюринга за полиномиальное время. Поэтому можно считать, что M имеет всего одну ленту. Пусть состояниями M будут $q_1, q_2, \ldots, q_s$, а символами на ленте — $X_1, X_2, \ldots, X_m$. Пусть $p(n)$ — временная сложность машины M.

Предположим, что цепочка w, поданная на вход машины M, имеет длину n. Таким образом, если M допускает w, то делает это не более чем за $p(n)$ шагов. Если M допускает w, то найдется хотя бы одна такая последовательность МО $Q_0, Q_1, \ldots, Q_q$, что Q_0 — начальное МО, $Q_{i-1} \vdash Q_i$ для $1 \leqslant i \leqslant q$, Q_q — допускающее МО, $q \leqslant p(n)$ и ни одно МО не занимает более $p(n)$ клеток на ленте.

[1]) Конечные множества мы кодируем цепочками вида $\{i_1, i_2, \ldots, i_m\}$, где i_j — десятичные (или двоичные) целые числа, представляющие элементы этих множеств. Если во всех множествах всего n разных элементов, мы представляем j-й элемент целым числом j.

Построим булеву формулу w_0, которая "моделирует" последовательность МО, проходимых машиной M. Любое присвоение значений *истина* (число 1) и *ложь* (число 0) переменным формулы w_0 соответствует самое большее одной последовательности МО, возможно незаконной (т. е. такой, которая не может на самом деле реализоваться). Булева формула w_0 примет значение 1 тогда и только тогда, когда это присваивание значений переменным соответствует последовательности МО $Q_0, Q_1, \ldots, Q_q$, ведущей к допусканию входа. Иными словами, булева формула w_0 будет выполнима тогда и только тогда, когда M допускает w. В качестве пропозициональных переменных в w_0 употребляются следующие переменные. Перечислим их вместе с их подразумеваемой интерпретацией.

1. $C\langle i, j, t\rangle=1$ тогда и только тогда, когда i-я клетка на входной ленте машины M содержит символ X_j в момент времени t. Здесь $1\leqslant i\leqslant p(n)$, $1\leqslant j\leqslant m$ и $0\leqslant t\leqslant p(n)$.
2. $S\langle k, t\rangle=1$ тогда и только тогда, когда M в момент времени t находится в состоянии q_k. Здесь $1\leqslant k\leqslant s$ и $0\leqslant t\leqslant p(n)$.
3. $H\langle i, t\rangle=1$ тогда и только тогда, когда головка в момент времени t обозревает i-ю клетку. Здесь $1\leqslant i\leqslant p(n)$ и $0\leqslant t\leqslant p(n)$.

Таким образом, пропозициональных переменных всего $O(p^2(n))$ и их можно представить двоичными числами, содержащими не более $c\log n$ разрядов, где c — постоянная, зависящая от p. В дальнейшем удобно будет представлять себе пропозициональную переменную как один символ, а не $c\log n$ символов. Эта потеря множителя $c\log n$ не может отразиться на полиномиальной ограниченности времени, затрачиваемого на вычисление той или иной функции.

Из этих пропозициональных переменных построим булеву формулу w_0, следуя структуре последовательности МО $Q_0, Q_1, \ldots, Q_q$. В этом построении мы воспользуемся предикатом $U(x_1, x_2, \ldots, x_r)$, принимающим значение 1, когда в точности один из аргументов $x_1, x_2, \ldots, x_r$ принимает значение 1. Предикат U можно выразить булевой формулой вида

$$U(x_1, x_2, \ldots, x_r)=(x_1+x_2+\ldots+x_r)\left(\prod_{\substack{i,j\\ i\neq j}}(\neg x_i+\neg x_j)\right). \quad (10.1)$$

Первый множитель в (10.1) утверждает, что по крайней мере одна из переменных x_i истинна. Остальные $r(r-1)/2$ множителей утверждают, что никакие две переменные не являются истинными одновременно. Заметим, что формальная запись формулы $U(x_1, x_2, \ldots, x_r)$ имеет длину $O(r^2)$ [1]).

[1]) Напомним, что мы считаем переменную одним символом. Строго говоря, потребуется $O(r^2\log r)$ и, значит, не более $O(r^3)$ символов.

Если M допускает w, то найдется допускающая последовательность МО $Q_0, Q_1, \ldots, Q_q$, проходимая машиной M в процессе обработки цепочки w. Для упрощения рассмотрений мы, не умаляя общности, будем считать, что машина M модифицирована так, что она делает ровно $p(n)$ шагов, а все ее МО содержат $p(n)$ клеток. Этого можно добиться, заставив M, если нужно, работать после перехода в допускающее состояние, но не изменяя его и не сдвигая головку, и дополнив все МО нужным числом пустых символов. Поэтому будем строить w_0 как произведение семи формул $A, B, \ldots, G$, которые утверждают, что $Q_0, Q_1, \ldots, Q_q$ — допускающая последовательность МО, причем каждое МО Q_i имеет длину $p(n)$ и $q=p(n)$. Утверждение о том, что $Q_0, Q_1, \ldots, Q_{p(n)}$ — допускающая последовательность МО, равносильно совокупности утверждений:

1) в каждом МО головка обозревает ровно одну клетку;
2) в каждом МО в каждой клетке ленты стоит ровно один символ;
3) каждое МО содержит в точности одно состояние;
4) при переходе от одного МО к следующему изменяется содержимое разве что клетки, обозреваемой головкой;
5) изменение состояния, положения головки и содержимого клетки при переходе от МО к следующему происходит в соответствии с функцией переходов машины M;
6) первое МО является начальным;
7) состояние в последнем МО — заключительное.

Построим булевы формулы $A, \ldots, G$, соответствующие утверждениям 1—7.

1. A утверждает, что в каждый момент времени машина M обозревает в точности одну клетку. Пусть A_t утверждает, что в момент t обозревается в точности одна клетка. Тогда $A=A_0\, A_1 \ldots A_{p(n)}$, где

$$A_t = U(H\langle 1, t\rangle, H\langle 2, t\rangle, \ldots, H\langle p(n), t\rangle).$$

Заметим, что если раскрыть выражение, обозначенное через U, то окажется, что формула A имеет длину $O(p^3(n))$ и ее можно записать за такое же время.

2. B утверждает, что каждая клетка ленты в каждый момент времени содержит в точности один символ. Пусть B_{it} утверждает, что i-я клетка ленты в момент времени t содержит ровно один символ. Тогда

$$B = \prod_{i,t} B_{it},$$

где

$$B_{it} = U(C\langle i, 1, t\rangle, C\langle i, 2, t\rangle, \ldots, C\langle i, m, t\rangle).$$

Длина каждой формулы B_{it} не зависит от n, поскольку размер m ленточного алфавита зависит только от машины Тьюринга M. Таким образом, B имеет длину $O(p^2(n))$.

3. C утверждает, что в каждый момент времени t машина M находится в одном и только одном состоянии:

$$C = \prod_{0 \leqslant t \leqslant p(n)} U(S\langle 1, t\rangle, S\langle 2, t\rangle, \ldots, S\langle s, t\rangle).$$

Так как число состояний s машины M постоянно, то C имеет длину $O(p(n))$.

4. D утверждает, что в любой момент времени t можно изменить содержимое не более чем одной клетки [1]):

$$D = \prod_{i, j, t} [(C\langle i, j, t\rangle \equiv C\langle i, j, t+1\rangle) + H\langle i, t\rangle].$$

Формула $(C\langle i, j, t\rangle \equiv C\langle i, j, t+1\rangle) + H\langle i, t\rangle$ утверждает, что либо

(а) в момент t головка обозревает клетку i, либо
(б) j-й символ записан в клетке i в момент $t+1$ тогда и только тогда, когда он был записан там в момент t.

Поскольку A и B утверждают, что в момент t головка обозревает только одну клетку и клетка i содержит лишь один символ, то в момент времени t либо головка обозревает клетку i, либо содержимое клетки i не меняется. Если раскрыть значение знака $\equiv$, то длина D будет $O(p^2(n))$.

5. E утверждает, что очередное МО машины M получается из предыдущего одним переходом, разрешаемым функцией переходов δ машины M. Пусть E_{ijkt} означает одно из следующих утверждений:

(а) в момент t клетка i не содержит символа j,
(б) в момент t головка не обозревает клетку i,
(в) в момент t машина M не находится в состоянии k,
(г) очередное МО машины M получается из предыдущего переходом, разрешаемым функцией переходов машины M.

Тогда

$$E = \prod_{i, j, k, t} E_{ijkt},$$

где

$$E_{ijkt} = \neg C\langle i, j, k\rangle + \neg H\langle i, t\rangle + \neg S\langle k, t\rangle + \\ + \sum_{l} [C\langle i, j_l, t+1\rangle S\langle k_l, t+1\rangle H\langle i_l, t+1\rangle].$$

Здесь l пробегает по всем шагам машины M, возможным в случае, когда M обозревает символ X_j в состоянии q_k. Иными словами, для каждой тройки $\langle q, X, d\rangle$ из $\delta(q_k, X_j)$ существует одно значение l, для которого $X_{j_l}=X$, $q_{k_l}=q$ и число i_l равно $i-1$, i или $i+1$ в зависимости от d: сдвигается ли головка влево ($d=\mathrm{L}$, $i_l=i-1$),

[1]) $x \equiv y$ означает $xy + \bar{x}\bar{y}$ (x тогда и только тогда, когда y).

вправо (d=R, i_t=i+1) или остается на месте (d=S, i_t=i). Здесь δ — функция переходов для M. Поскольку машина M недетерминированна, то таких троек (q, X, d) может быть несколько, но во всяком случае их число конечно. Таким образом, E_{ijkt} имеет ограниченную длину, не зависящую от n. Поэтому E имеет длину $O(p^2(n))$.

6. F утверждает, что выполнены начальные условия:

$$F = S\langle 1, 0\rangle H\langle 1, 0\rangle \prod_{1 \leqslant i \leqslant n} C\langle i, j_i, 0\rangle \prod_{n < i \leqslant p(n)} C\langle i, 1, 0\rangle,$$

где $S\langle 1, 0\rangle$ утверждает, что M в момент t=0 находится в состоянии q_1, которое мы считаем начальным; $H\langle 1, 0\rangle$ утверждает, что M в момент t=0 обозревает самую левую клетку ленты; $\prod_{1 \leqslant i \leqslant n} C\langle i, j_i, 0\rangle$ утверждает, что первые n клеток вначале содержат входную цепочку w; $\prod_{n < i \leqslant p(n)} C\langle i, 1, 0\rangle$ утверждает, что остальные клетки вначале пусты. Мы считаем, что пустым символом является X_1. Очевидно, что F имеет длину $O(p(n))$.

7. G утверждает, что M в конце концов приходит в заключительное состояние. Поскольку мы потребовали, чтобы машина M оставалась в заключительном состоянии после того, как оно достигнуто, то G=$S\langle s, p(n)\rangle$. Мы считаем, что заключительным состоянием является q_s.

Булева формула w_0 — это произведение $ABCDEFG$. Поскольку каждый из семи сомножителей состоит не более чем из $O(p^3(n))$ символов, сама формула w_0 также состоит не более чем из $O(p^3(n))$ символов. Так как мы считали каждую пропозициональную переменную за один символ, а в действительности переменные представляются цепочками длины $O(\log n)$, то на самом деле границей на $|w_0|$ будет $O(p^3(n) \log n)$, а это не превосходит $cnp^3(n)$, где c — некоторая постоянная. Таким образом, длина цепочки w_0 полиномиально зависит от длины цепочки w. Должно быть ясно, что если даны цепочка w и полином p, то формулу w_0 можно записать за время, пропорциональное ее длине.

По данной допускающей последовательности МО $Q_0, Q_1, \ldots, Q_q$ можно, очевидно, найти такое присваивание значений 0 и 1 пропозициональным переменным $C\langle i, j, t\rangle$, $S\langle k, t\rangle$ и $H\langle i, t\rangle$, что w_0 примет значение *истина*. Обратно, если дано присваивание значений переменным формулы w_0, при котором она становится истинной, то легко найти допускающую последовательность МО. Таким образом, формула w_0 выполнима тогда и только тогда, когда M допускает цепочку w.

Мы не наложили на язык L, допускаемый машиной M, никаких ограничений, кроме того, что он принадлежит классу $\mathcal{NP}$. Поэтому мы показали, что любой язык из $\mathcal{NP}$ полиномиально трансформи-

руем в задачу выполнимости булевых формул. Отсюда заключаем, что задача выполнимости NP-полна. □

На самом деле мы доказали больше, чем утверждается в теореме 10.3. Говорят, что булева формула находится в *конъюнктивной нормальной форме* (КНФ), если она представляет собой произведение сумм литералов, где каждый *литерал* имеет вид x или $\neg x$ для некоторой переменной x. Например, $(x_1+x_2)(x_2+\bar{x}_1+x_3)$ находится в КНФ, а $x_1x_2+x_3$ нет. Формула w_0, построенная в доказательстве теоремы 10.3, практически находится в КНФ — ее можно представить в КНФ, увеличив длину не более чем в постоянное число раз.

Следствие. *Задача выполнимости булевых формул, находящихся в* КНФ, NP-*полна.*

Доказательство. Достаточно показать, что каждая из формул $A,\ldots, G$, построенных в доказательстве теоремы 10.3, либо уже находится в КНФ, либо может быть преобразована в такую с помощью правил булевой алгебры, причем ее длина увеличится не более чем в постоянное число раз. Формула U, определенная равенством (10.1), уже находится в КНФ. Отсюда следует, что A, B и C тоже находятся в КНФ. F и G тривиально находятся в КНФ, поскольку F и G — произведения литералов.

D — формула вида $(x\equiv y)+z$. Если раскрыть знак $\equiv$, получится выражение

$$xy+\bar{x}\bar{y}+z, \tag{10.2}$$

эквивалентное

$$(x+\bar{y}+z)(\bar{x}+y+z). \tag{10.3}$$

Подставив (10.3) вместо всех вхождений (10.2) в D, получим формулу, эквивалентную исходной, но находящуюся в НКФ и превосходящую исходную формулу по длине не более чем в 2 раза.

Наконец, E — произведение формул E_{ijkt}. Поскольку длины всех E_{ijkt} ограничены независимо от n, каждую формулу E_{ijkt} можно преобразовать в формулу в КНФ, причем ее длина не будет зависеть от n. Поэтому преобразование формулы E в формулу в КНФ увеличивает ее длину не более чем в постоянное число раз.

Итак, формулу w_0 можно представить в КНФ, увеличив ее длину не более чем в постоянное число раз. □

Мы только что показали, что задача выполнимости формул в КНФ NP-полна. Можно показать, что даже при более жестких ограничениях на вид формулы задача выполнимости булевых формул также NP-полна. Говорят, что формула находится в *k-конъюнктивной нормальной форме* (k-КНФ), если она представляет собой произведение сумм, состоящих не более чем из k литералов. Задача k-выполнимости состоит в выяснении выполнимости формулы, нахо-

дящейся в k-КНФ. Для $k=1$ и 2 известны детерминированные алгоритмы полиномиальной сложности, проверяющие k-выполнимость. Ситуация (по-видимому) изменяется при $k=3$, как видно из следующей теоремы.

Теорема 10.4. *Задача 3-выполнимости* NP-*полна.*

Доказательство. Покажем, что выполнимость формул в КНФ полиномиально трансформируема в 3-выполнимость. В данном произведении сумм заменим каждую сумму $(x_1+x_2+\ldots+x_k)$, $k\geqslant 4$, на

$$(x_1+x_2+y_1)(x_3+\bar{y}_1+y_2)(x_4+\bar{y}_2+y_3)\ldots \\ \ldots(x_{k-2}+\bar{y}_{k-4}+y_{k-3})(x_{k-1}+x_k+\bar{y}_{k-3}), \quad (10.4)$$

где $y_1, y_2, \ldots, y_{k-3}$ — новые переменные. Например, для $k=4$ выражение (10.4) принимает вид $(x_1+x_2+y_1)(x_3+x_4+\bar{y}_1)$.

Присвоить значения новым переменным так, чтобы подставляемая формула приняла значение 1, можно тогда и только тогда, когда один из литералов $x_1, x_2, \ldots, x_k$ имеет значение 1, т. е. тогда и только тогда, когда исходная формула принимает значение 1. Допустим, что $x_i=1$. Тогда положим $y_j=1$ для $j\leqslant i-2$ и $y_j=0$ для $j>i-2$. Подставляемая формула принимает значение 1. Обратно, допустим, что при некоторых значениях переменных y_i подставляемая формула принимает значение 1. Если $y_1=0$, то одна из переменных x_1 и x_2 должна иметь значение 1. Если $y_{k-3}=1$, то одна из переменных x_{k-1} и x_k должна иметь значение 1. Если $y_1=1$ и $y_{k-3}=0$, то для некоторого i, $1\leqslant i\leqslant k-4$, выполнены оба равенства $y_i=1$ и $y_{i+1}=0$, откуда следует, что переменная x_{i+2} должна иметь значение 1. В любом случае некоторая переменная x_i должна иметь значение 1.

Длина формулы, получаемой в результате описанной выше замены, превосходит длину исходной формулы не более чем в постоянное число раз. Таким образом, по данной формуле E в КНФ можно найти, применяя описанное выше преобразование к каждой сумме, формулу E' в 3-КНФ, выполнимую тогда и только тогда, когда выполнима исходная формула. Более того, легко показать, что E' можно найти за время, пропорциональное длине формулы E, выполняя преобразования очевидным образом. □

10.5. ЕЩЕ НЕСКОЛЬКО NP-ПОЛНЫХ ЗАДАЧ

Теперь приступим к доказательству того, что каждая из задач, упомянутых в теореме 10.2, NP-полна. Для этого покажем, что в нее прямо или косвенно преобразуется выполнимость. Дерево на рис. 10.6 иллюстрирует последовательность преобразований, которые мы в действительности будем применять. Если P — отец для

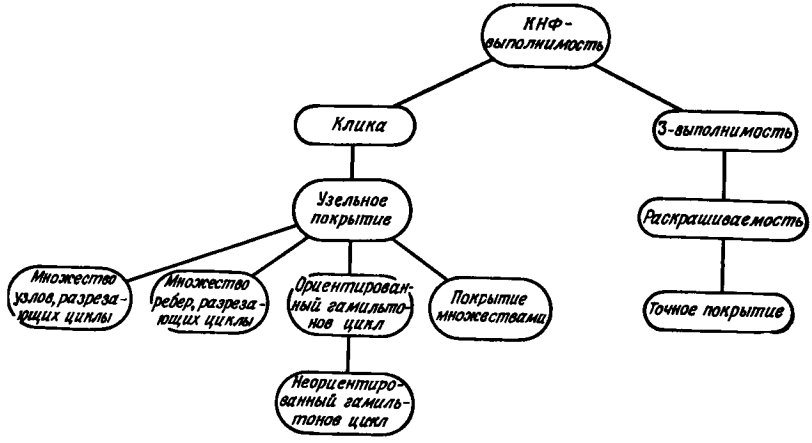


Рис. 10.6. Последовательность преобразований для различных трудных задач.

P' на рис. 10.6, то мы покажем, что задача P полиномиально трансформируема в P'.

Теорема 10.5. *Задача* КНФ-*выполнимости полиномиально трансформируема в задачу о клике. Поэтому задача о клике* NP-*полна.*

Доказательство. Пусть $F=F_1F_2\ldots F_q$ — формула в КНФ, где F_i — сомножители; каждая формула F_i имеет вид $(x_{i1}+x_{i2}+\ldots+x_{ik_i})$, где x_{ij} — литерал. Построим неориентированный граф $G=(V, E)$, узлами которого служат пары целых чисел $[i, j]$ для $1\leqslant i\leqslant q$ и $1\leqslant j\leqslant k_i$. Первая компонента такой пары представляет сомножитель, а вторая — литерал, входящий в него. Таким образом, каждый узел графа естественным образом соответствует конкретному вхождению в конкретный сомножитель.

Ребрами графа G служат пары $([i, j], [k, l])$, для которых $i\neq k$ и $x_{ij}\neq x_{kl}$. Неформально, узлы $[i, j]$ и $[k, l]$ смежны в G, если они соответствуют различным сомножителям и можно так присвоить значения переменным из литералов x_{ij} и x_{kl}, чтобы оба литерала приняли значения 1 (тем самым давая значение 1 формулам F_i и F_k). Иными словами, либо $x_{ij}=x_{kl}$, либо переменные, входящие в литералы x_{ij} и x_{kl}, различны.

Число узлов в G, очевидно, меньше длины формулы F, а число ребер не превосходит квадрата числа узлов. Поэтому граф G можно закодировать в виде цепочки, длина которой ограничена полиномом от длины формулы F, и, что еще важнее, такой код можно найти за время, ограниченное полиномом от длины F. Мы покажем, что G содержит q-клику тогда и только тогда, когда формула F выполнима. Отсюда будет следовать, что по данному алгоритму, решающему задачу о клике за полиномиально ограниченное время, можно по-

строить алгоритм с полиномиально ограниченным временем работы для задачи КНФ-выполнимости, построив по F граф G, содержащий q-клику тогда и только тогда, когда формула F выполнима. Итак, покажем, что для того, чтобы граф G содержал q-клику, необходимо и достаточно, чтобы формула F была выполнима.

Достаточность. Пусть формула F выполнима. Тогда существует набор значений переменных, состоящий из нулей и единиц, при котором $F=1$. При этом наборе каждый сомножитель формулы F принимает значение 1. Каждый сомножитель F_i содержит по меньшей мере один литерал, принимающий значение 1. Пусть таким литералом в F_i будет x_{im_i}.

Мы утверждаем, что множество узлов $\{[i, m_i] | 1 \leqslant i \leqslant q\}$ образует q-клику. Если бы это было не так, то нашлись бы такие i и j, что $i \neq j$ и узлы $[i, m_i]$ и $[j, m_j]$ не соединены ребром. Отсюда следовало бы, что $x_{im_i}=x_{jm_j}$ (по определению множества ребер графа G). Но это невозможно, поскольку $x_{im_i}=x_{jm_j}=1$ в силу выбора переменных x_{im_i}.

Необходимость. Пусть G содержит q-клику. Первые компоненты узлов, составляющих такую клику, должны быть различны, поскольку узлы с одинаковыми первыми компонентами не соединяются ребрами. Так как в этой клике в точности q узлов, то узлы клики взаимно однозначно соответствуют сомножителям формулы F. Пусть узлы клики имеют вид $[i, m_i]$, $1 \leqslant i \leqslant q$. Пусть $S_1=\{y | x_{im_i}=y$, где $1 \leqslant i \leqslant q$ и y — переменная$\}$ и $S_2=\{y | x_{im_i}=\bar{y}$, где $1 \leqslant i \leqslant q$ и y — переменная$\}$. Иными словами, S_1 и S_2 — множества переменных и отрицаний переменных соответственно, представленных узлами клики. Тогда $S_1 \cap S_2=\varnothing$, ибо в противном случае какие-то узлы $[s, m_s]$ и $[t, m_t]$, для которых $x_{sm_s}=\bar{x}_{tm_t}$, соединялись бы ребром. Если положить переменные из S_1 равными 1, а из S_2 равными 0, то каждая формула F_i примет значение 1. Поэтому формула F выполнима. □

Пример 10.5. Рассмотрим формулу $F=(y_1+\bar{y}_2)(y_2+\bar{y}_3)(y_3+\bar{y}_1)$. Литералы таковы:

$$x_{11}=y_1, \quad x_{21}=y_2, \quad x_{31}=y_3,$$
$$x_{12}=\bar{y}_2, \quad x_{22}=\bar{y}_3, \quad x_{32}=\bar{y}_1.$$

Конструкция из теоремы 10.5 дает граф, изображенный на рис. 10.7. Например, узел [1, 1] не соединен с [1, 2], потому что первые компоненты одинаковы, и с [3, 2], потому что $x_{11}=y_1$ и $x_{32}=\bar{y}_1$, а с остальными тремя узлами соединен.

В формуле F три сомножителя, и оказывается, что в графе на рис. 10.7 две 3-клики, а именно $\{[1,1], [2,1], [3,1]\}$ и $\{[1,2], [2,2], [3,2]\}$. В первой клике представлены три литерала y_1, y_2 и y_3.

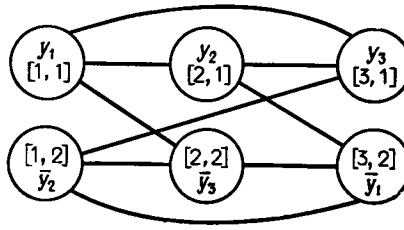


Рис. 10.7. Граф, построенный по теореме 10.5.

Это переменные без отрицаний, и первая клика соответствует присвоению значений $y_1=y_2=y_3=1$, выполняющему F. Вторая клика соответствует другому набору значений, обращающему формулу F в истинную, а именно $y_1=y_2=y_3=0$. □

Теорема 10.6. *Задача о клике полиномиально трансформируема в задачу об узельном покрытии. Поэтому задача об узельном покрытии* NP*-полна.*

Доказательство. Пусть дан неориентированный граф $G=(V, E)$. Рассмотрим его дополнение $\bar{G}=(V, \bar{E})$, где $\bar{E}=\{(v, w) | v, w \in V, v \neq w$ и $(v, w) \notin E\}$. Мы утверждаем, что множество $S \subseteq V$ является кликой в G тогда и только тогда, когда $V-S$ является узельным покрытием графа $\bar{G}$. Действительно, если S — клика в G, то никакое ребро в $\bar{G}$ не соединяет никакие два узла в S. Поэтому всякое ребро в $\bar{G}$ инцидентно по меньшей мере одному узлу из $V-S$, откуда следует, что $V-S$ есть узельное покрытие графа $\bar{G}$. Аналогично, если $V-S$ есть узельное покрытие графа $\bar{G}$, то каждое ребро из $\bar{G}$ инцидентно по меньшей мере одному узлу из $V-S$. Поэтому никакое ребро из $\bar{G}$ не соединяет два узла из S. Следовательно, каждая пара узлов из S соединена в G, и, значит, S — клика в G.

Чтобы узнать, существует ли k-клика, построим граф $\bar{G}$ и выясним, содержит ли он узельное покрытие размера $||V||-k$. Разумеется, по данному стандартному представлению графа $G=(V, E)$ и числа k можно найти представление графа $\bar{G}$ и числа $||V||-k$ за время, полиномиально зависящее от длины представления G и k. □

Пример 10.6. Граф G на рис. 10.8,*а* содержит две 3-клики $\{1, 2, 5\}$ и $\{1, 4, 5\}$. Граф $\bar{G}$ на рис. 10.8,*б* содержит соответствующие узельные покрытия $\{3, 4\}$ и $\{2, 3\}$ размера 2. Граф G содержит (среди других) 2-клики $\{2, 3\}$ и $\{3, 4\}$, а граф $\bar{G}$ — соответствующие узельные покрытия $\{1, 4, 5\}$ и $\{1, 2, 5\}$ размера 3. □

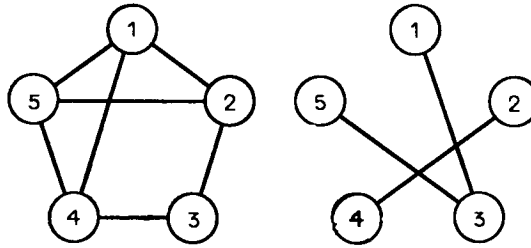


Рис. 10.8. *а* — граф G; *б* — его дополнение $\overline{G}$.

Теорема 10.7. *Задача об узельном покрытии полиномиально трансформируема в задачу о множестве узлов, разрезающих циклы. Поэтому задача о множестве узлов, разрезающих циклы,* NP*-полна.*

Доказательство. Пусть $G=(V, E)$ — неориентированный граф, а D — ориентированный граф, полученный заменой каждого ребра графа G двумя ориентированными ребрами. Точнее, пусть $D=(V, E')$, где $E'=\{(v, w), (w, v) | (v, w) \in E\}$. Поскольку каждое ребро из E заменено циклом графа D, то множество $S \subseteq V$ разрезает циклы в D (каждый цикл графа D содержит узел из S) тогда и только тогда, когда S — узельное покрытие для G. Кроме того, представление графа D легко найти по G за полиномиальное время. □

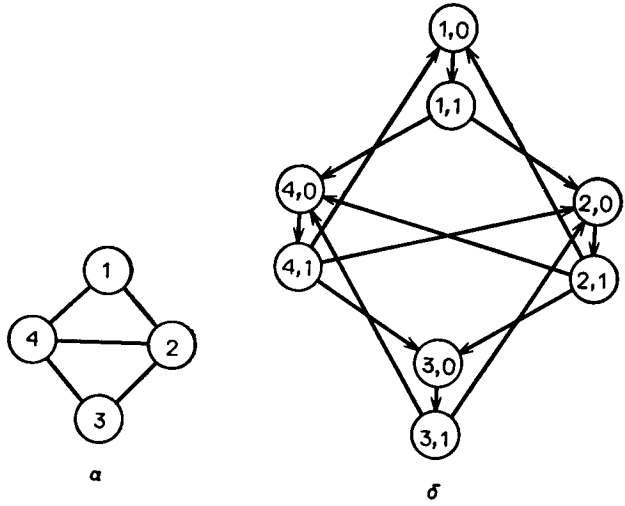
Теорема 10.8. *Задача об узельном покрытии полиномиально трансформируема в задачу о множестве ребер, разрезающих циклы. Поэтому задача о множестве ребер, разрезающих циклы,* NP*-полна.*

Доказательство. Пусть $G=(V, E)$ — неориентированный граф, а $D=(V \times \{0, 1\}, E')$ — ориентированный граф, где E' состоит из (см. рис. 10.9)

1) $[v, 0] \to [v, 1]$[1]) для каждого $v \in V$ и
2) $[v, 1] \to [w, 0]$ и $[w, 1] \to [v, 0]$ для каждого неориентированного ребра $(v, w) \in E$.

Пусть $F \subseteq E'$ — множество ребер графа D, содержащих по крайней мере одно ребро из каждого цикла в D. Заменим каждое ребро из F, имеющее вид $[v, 1] \to [w, 0]$, ребром $[w, 0] \to [w, 1]$. Полученное множество обозначим через F'. Мы утверждаем, что $||F'|| \leqslant ||F||$ и F' содержит по крайней мере одно ребро из каждого цикла. (Единственное ребро, выходящее из $[w, 0]$, идет в $[w, 1]$, так что $[w, 0] \to [w, 1]$ принадлежит любому циклу, содержащему $[v, 1] \to [w, 0]$.)

[1]) Здесь и далее в этой главе ориентированное ребро (x, y) обозначается через $x \to y$. Это соглашение облегчает чтение.



а *б*

Рис. 10.9. *а* — неориентированный граф G; *б* — соответствующий ориентированный граф D. Узельное покрытие $\{2, 4\}$ соответствует множеству $\{[2, 0]\to[2, 1], [4, 0]\to[4, 1]\}$ ребер, разрезающих циклы.

Не умаляя общности, будем считать, что

$$F' = \{[v_i, 0]\to[v_i, 1] \mid 1 \leqslant i \leqslant k\}$$

для некоторого k. Тогда каждый цикл в D содержит ребро $[v_i, 0]\to[v_i, 1]$ для некоторого i, $1 \leqslant i \leqslant k$. Однако заметим, что если (x, y) — произвольное ребро в G, то $[x, 1]$, $[y, 0]$, $[y, 1]$, $[x, 0]$, $[x, 1]$ образуют цикл в D. Поэтому каждое ребро в G инцидентно некоторому узлу v_i, $1 \leqslant i \leqslant k$, и, значит, $\{v_1, \ldots, v_k\}$ — узельное покрытие графа G.

Обратно, легко показать, что если S — узельное покрытие размера k, то $\{[v, 0]\to[v, 1] \mid v \in S\}$ — множество ребер, разрезающих циклы в D. Чтобы узнать, содержит ли G узельное покрытие размера k, надо построить за полиномиальное время граф D и выяснить, есть ли в нем k-элементное множество ребер, разрезающих циклы. □

Теорема 10.9. *Задача об узельном покрытии полиномиально трансформируема в задачу о гамильтоновом цикле для ориентированных графов. Поэтому задача о гамильтоновом цикле в ориентированном графе* NP-*полна.*

Доказательство. Пусть дан неориентированный граф $G=(V, E)$ и целое число k. Покажем, как за время, полиномиальное по $\|V\|$, построить ориентированный граф $G_D=(V_D, E_D)$, содержа-

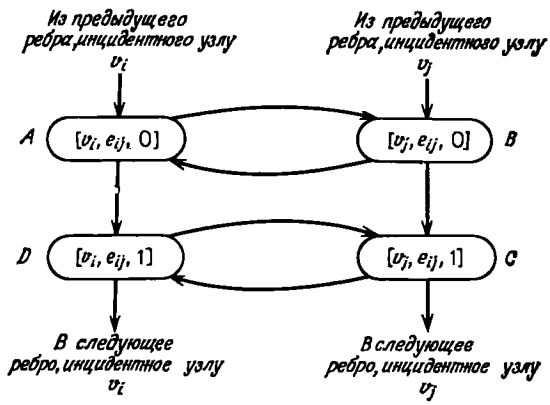


Рис. 10.10. Представление ребра (v_i, v_j).

щий гамильтонов цикл тогда и только тогда, когда некоторое k-элементное подмножество множества V покрывает ребра графа G.

Пусть $V=\{v_1, v_2, \ldots, v_n\}$. Обозначим ребро (v_i, v_j) через e_{ij}[1]). Пусть $a_1, a_2, \ldots, a_k$ — новые символы. Множество V_D будет включать в себя по одному узлу для каждого a_i и еще по четыре узла для каждого ребра графа G. Точнее,

$$V_D = \{a_1, a_2, \ldots, a_k\} \cup \{[v, e, b] \mid v \in V,\ e \in E,\ b \in \{0, 1\} \text{ и ребро } e \text{ инцидентно } v\}.$$

Прежде чем формально описывать ребра графа G_D, дадим объяснение на интуитивном уровне. Ребро (v_i, v_j) графа G представляется подграфом с четырьмя узлами (рис. 10.10).

Если гамильтонов цикл входит в подграф, представляющий ребро e_{ij}, в узле A, то он должен выйти из него в узле D, ибо если он выйдет в узле C, то $[v_j, e_{ij}, 0]$ или $[v_i, e_{ij}, 1]$ не сможет оказаться в этом цикле. Идя из A в D, цикл может посетить либо все четыре узла подграфа, либо только два самых левых. В последнем случае, гамильтонов цикл должен в какой-то момент пройти из B в C, посетив два самых правых узла.

Граф G_D можно представлять себе состоящим из $||V||$ списков, по одному списку для каждого узла. Список для узла v содержит все ребра, инцидентные узлу v и расположенные в специальном порядке. Из каждого узла a_j, $1 \leqslant j \leqslant k$, идут ребра к таким узлам $[v_i, e_{il}, 0]$, что e_{il} — первое ребро в списке для v_i. Далее, в каждый узел a_j идут ребра из $[v_i, e_{il}, 1]$, если e_{il} — последнее ребро в списке для v_i. Из $[v_i, e_{il}, 1]$ идет ребро в $[v_i, e_{im}, 0]$, если ребро e_{im} непосредственно следует за e_{il} в списке для v_i. Эти ребра вместе с ребра-

[1]) Заметим, что e_{ij} и e_{ji} обозначают одно и то же ребро, и их следует рассматривать как один и тот же символ.

ми, необходимыми в силу рис. 10.10, образуют множество E_D. Покажем, что для того, чтобы в графе G_D был гамильтонов цикл, необходимо и достаточно, чтобы в G было множество из k узлов, покрывающее все ребра.

Достаточность. Допустим, что узлы $v_1, v_2, \ldots, v_k$ покрывают все ребра графа G. Пусть $f_{i1}, f_{i2}, \ldots, f_{il}$ — список ребер, инцидентных v_i, $1 \leqslant i \leqslant k$. Рассмотрим цикл, идущий из узла a_1 в $[v_1, f_{11}, 0]$, $[v_1, f_{11}, 1]$, $[v_1, f_{12}, 0]$, $[v_1, f_{12}, 1], \ldots, [v_1, f_{1l_1}, 0]$, $[v_1, f_{1l_1}, 1]$ и далее в a_2, затем в $[v_2, f_{21}, 0]$, $[v_2, f_{21}, 1], \ldots$ и т. д. по ребрам списков для $v_3, v_4, \ldots, v_k$ и, наконец, обратно в a_1. Этот цикл проходит через каждый узел графа G_D, кроме узлов, содержащихся в списках ребер для узлов, отличных от $v_1, \ldots, v_k$. Для каждой пары узлов графа G_D, имеющей вид $[v_j, e, 0]$, $[v_j, e, 1]$, $j > k$, можно расширить этот цикл, поскольку ребро e инцидентно некоторому узлу v_i, $1 \leqslant i \leqslant k$. Заменим ребро, идущее из $[v_l, e, 0]$ в $[v_i, e, 1]$ в этом цикле, путем

$$[v_l, e, 0], [v_j, e, 0], [v_j, e, 1], [v_i, e, 1].$$

Цикл, исправленный описанным выше способом для каждого узла v_j и ребра e, содержит каждый узел графа G_D и, значит, является гамильтоновым.

Необходимость. Пусть в графе G_D есть гамильтонов цикл. Этот цикл можно разбить на k путей, каждый из которых начинается в некотором узле a_i и кончается в каком-то узле a_j. Из наших предыдущих рассмотрений возможных путей через подграф с четырьмя узлами, представляющий какое-то ребро (как на рис. 10.10), вытекает, что существует такой узел v, что каждый узел на пути из a_i в a_j имеет вид либо $[v, e, 0]$ или $[v, e, 1]$, где e — ребро графа G, либо $[w, e, 0]$ или $[w, e, 1]$, где e — ребро (v, w). Поэтому первая компонента каждого узла, лежащего на пути из a_i в a_j, представляет собой либо v, либо узел, смежный с v. Пути из a_i в a_j можно, таким образом, поставить в соответствие некоторый узел v. Для каждого узла $[w, e, b]$ графа G_D ребро e должно быть инцидентно одному из k узлов графа G, поставленных в соответствие путям. Следовательно, эти k узлов образуют узельное покрытие графа G. Отсюда заключаем, что G_D содержит гамильтонов цикл тогда и только тогда, когда в G существуют такие k узлов, что каждое ребро из G инцидентно одному из этих k узлов. □

Пример 10.7. Пусть G — граф с узлами v_1, v_2, v_3 и ребрами e_{12}, e_{13}, e_{23} (рис. 10.11,*а*). Граф G_D, построенный в теореме 10.9, изображен на рис. 10.11,*б*. Гамильтонов цикл в G_D, основанный на узельном покрытии $\{v_1, v_2\}$, указан жирными линиями. □

Теорема 10.10. *Задача о гамильтоновом цикле для ориентированных графов полиномиально трансформируема в задачу о гамильтоновом цикле для неориентированных графов. Поэтому задача существования гамильтонова цикла в неориентированном графе* NP*-полна.*

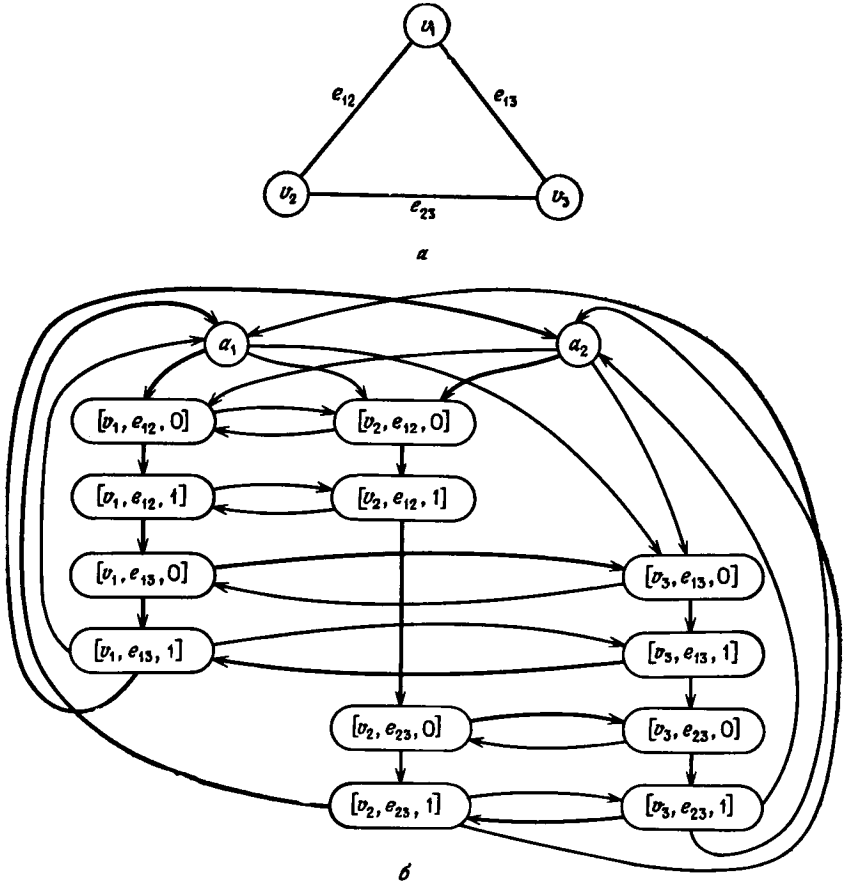


Рис. 10.11. Граф с гамильтоновым циклом: *а* — неориентированный граф G; *б* — ориентированный граф G_D.

Доказательство. Пусть $G=(V, E)$ — ориентированный граф, а $U=(V\times\{0, 1, 2\}, E')$ — неориентированный граф, где E' состоит из ребер

1) $([v, 0], [v, 1])$ для $v\in V$,
2) $([v, 1], [v, 2])$ для $v\in V$,
3) $([v, 2], [w, 0])$ тогда и только тогда, когда ребро $v\to w$ принадлежит E.

Отметим, что каждый узел из V разложен в путь, состоящий из трех узлов. Так как гамильтонов цикл в U должен включать в себя все узлы, то последняя компонента узлов, составляющих этот цикл,

должна изменяться в порядке 0, 1, 2, 0, 1, 2,. . . или обратном к нему. Будем считать, что реализуется первый из этих двух случаев; тогда ребра типа 3, у которых вторая компонента меняется с 2 на 0, образуют ориентированный гамильтонов цикл в G. Обратно, гамильтонов цикл в G можно превратить в неориентированный гамильтонов цикл в U, заменив каждое ребро $v \rightarrow w$ путем $[v, 0]$, $[v, 1]$, $[v, 2]$, $[w, 0]$ в U. □

Теорема 10.11. *Задача об узельном покрытии полиномиально трансформируема в задачу о покрытии множествами. Поэтому задача о покрытии множествами* NP-*полна.*

Доказательство. Пусть $G=(V, E)$ — неориентированный граф. Для $1 \leqslant i \leqslant \|V\|$ обозначим через S_i множество всех ребер, инцидентных узлу v_i. Очевидно, что $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ образуют покрытие множествами для S_i тогда и только тогда, когда $\{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ — узельное покрытие графа G. □

Теорема 10.12. *Задача* 3-*выполнимости полиномиально трансформируема в задачу раскрашиваемости.*

Доказательство. Пусть дана формула F в 3-КНФ с n переменными и t сомножителями. Покажем, как построить за время, ограниченное полиномом от MAX(n, t), неориентированный граф $G=(V, E)$ с $3n+t$ узлами, который можно раскрасить в $n+1$ цветов тогда и только тогда, когда формула F выполнима.

Пусть $x_1, x_2, \ldots, x_n$ и $F_1, F_2, \ldots, F_t$ — соответственно переменные и сомножители формулы F. Пусть $v_1, v_2, \ldots, v_n$ — новые символы. Без потери общности будем считать, что $n \geqslant 4$, поскольку любую формулу в 3-КНФ, число различных переменных которой не превосходит 3, можно проверить на выполнимость за время, линейно зависящее от ее длины, не прибегая к раскрашиваемости. Узлы графа G таковы:

1) x_i, $\bar{x}_i$, v_i для $1 \leqslant i \leqslant n$,
2) F_i для $1 \leqslant i \leqslant t$.

Ребра графа G —

1) все (v_i, v_j), для которых $i \neq j$,
2) все (v_i, x_j) и $(v_i, \bar{x}_j)$, для которых $i \neq j$,
3) $(x_i, \bar{x}_i)$ для $1 \leqslant i \leqslant n$,
4) (x_i, F_j), если x_i не входит в F_j, и $(\bar{x}_i, F_j)$, если $\bar{x}_i$ не входит в F_j.

Узлы $v_1, v_2, \ldots, v_n$ образуют полный подграф с n узлами, так что для их раскраски требуется n различных цветов. Каждый из узлов x_j и $\bar{x}_j$ соединен с каждым v_i, $i \neq j$, и, значит, x_j и $\bar{x}_j$ не могут

быть того же цвета, что и v_i, если $i \neq j$. Так как узлы x_j и $\bar{x}_j$ смежны, то они не могут быть одинакового цвета, и потому граф G можно раскрасить в $n+1$ цветов только тогда, когда один из узлов x_j и $\bar{x}_j$ имеет тот же цвет, что и v_j, а другой имеет новый цвет, который мы назовем *специальным*.

Представим себе, что тому из узлов x_j и $\bar{x}_j$, который раскрашен в специальный цвет, приписано значение 0. Теперь рассмотрим цвет, приписанный узлам F_j. Узел F_j смежен по крайней мере с $2n-3$ из $2n$ узлов $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$. Так как мы предположили, что $n \geqslant 4$, то для каждого j найдется такое i, что узел F_j смежен как с x_i, так и с $\bar{x}_i$. Поскольку один из узлов x_i или $\bar{x}_i$ раскрашен в специальный цвет, то F_j не может быть раскрашен в специальный цвет.

Если формула F_j содержит такой литерал y, что узлу $\bar{y}$ приписан специальный цвет, то узел F_j не смежен ни с каким узлом, раскрашенным так же, как y, и, значит, ему можно приписать тот же цвет, что и у y. В противном случае нужен новый цвет. Таким образом, все F_i можно раскрасить без дополнительных цветов тогда и только тогда, когда литералам можно так приписать специальный цвет, чтобы каждый сомножитель содержал такой литерал y, что литералу $\bar{y}$ приписан специальный цвет, т. е. тогда и только тогда, когда переменным можно так присвоить значения, чтобы в каждом сомножителе оказался y со значением 1 ($\bar{y}$ со значением 0), т. е. тогда и только тогда, когда формула F выполнима. □

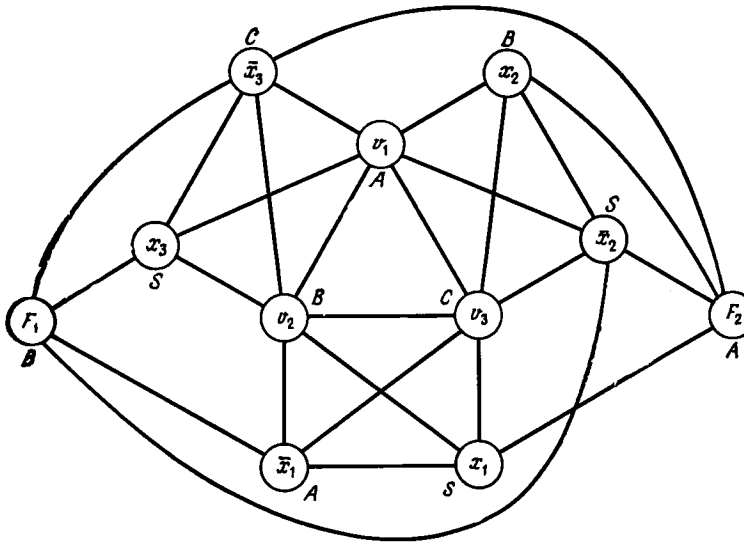


Рис. 10.12. Граф с хроматическим числом 4.

Пример 10.8. Пусть $F=(x_1+x_2)(\bar{x}_1+x_3)$. Заметим, что здесь в каждом сомножителе по два, а не по три члена. Это изменение не влияет на конструкцию графа G в доказательстве теоремы 10.12, но позволяет рассматривать формулы, содержащие только три переменные, и графы, раскрашиваемые в четыре цвета. (Аналог теоремы 10.12 для 2-КНФ верен, но неинтересен. Так как существует алгоритм для 2-КНФ с полиномиально ограниченным временем работы, то выполнимость для 2-КНФ трансформируема за полиномиальное время в любую задачу.) Результирующий граф показан на рис. 10.12. В качестве цветов взяты A, B, C и S (специальный). Эта 4-раскраска соответствует решению $x_1=\bar{x}_2=x_3=0$. □

Теорема 10.13. *Задача раскрашиваемости полиномиально трансформируема в задачу о точном покрытии. Поэтому задача о точном покрытии* NP-*полна.*

Доказательство. Пусть $G=(V, E)$ — неориентированный граф и k — целое число. Будем строить множества, элементы которых выбираются из

$$S=V\cup\{[e, i]\,|\,e\in E \text{ и } 1\leqslant i\leqslant k\}.$$

Для каждого узла $v\in V$ и $1\leqslant i\leqslant k$ зададим множества

$$S_{vi}=\{v\}\cup\{[e, i]|\text{ ребро } e \text{ инцидентно } v\}. \tag{10.5}$$

Для каждого ребра $e\in E$ и $1\leqslant i\leqslant k$ зададим множества

$$T_{ei}=\{[e, i]\}. \tag{10.6}$$

Установим соответствие между k-раскрасками графа G и точными покрытиями набора множеств, определенных равенствами (10.5) и (10.6). Предположим, что G имеет k-раскраску, в которой узлу v приписан цвет c_v, где c_v можно считать целым числом между 1 и k. Тогда легко проверить, что набор множеств, состоящий из S_{vc_v} для каждого v и тех одноэлементных множеств T_{ei}, для которых $[e, i]\notin S_{vc_v}$ при всех v, образует точное покрытие. Для доказательства заметим, что если $e=(v, w)$ — ребро, то $c_v\neq c_w$, поэтому $S_{vc_v}\cap S_{wc_w}=\varnothing$. Ясно, что если x и y не смежны, то S_{xc_x} и S_{yc_y} не пересекаются, а множество T_{ei} выбирается, только если оно не пересекается ни с одним из выбранных множеств.

Обратно, допустим, что набор множеств, определенных в (10.5) и (10.6), имеет точное покрытие $\mathcal{S}$. Тогда для каждого v найдется такое единственное число c_v, что $S_{vc_v}\in\mathcal{S}$. Это вытекает из того, что каждый узел v должен принадлежать точно одному множеству из $\mathcal{S}$. Мы утверждаем, что для получения k-раскраски графа G можно раскрасить каждый узел v в цвет c_v. Допустим, что это не так.

Тогда существует такое ребро $e=(v, w)$, что $c_v=c_w=c$. А тогда каждое из множеств S_{vc_v} и S_{wc_w} содержит $[e, c]$, и потому $\mathcal{S}$ не является точным покрытием вопреки предположению. □

10.6. ЗАДАЧИ С ПОЛИНОМИАЛЬНО ОГРАНИЧЕННОЙ ПАМЯТЬЮ

Класс $\mathcal{NP}$ содержится в другом естественном классе трудных задач. Этот класс, обозначаемый $\mathcal{P}$-SPACE, образован языками, допускаемыми детерминированными машинами Тьюринга с полиномиально ограниченной памятью (т. е. емкостной сложностью).

Вспомним, что по теореме 10.1, если L допускается некоторой НМТ с емкостной сложностью $p(n)$, где p — полином, то он допускается некоторой ДМТ с емкостной сложностью $p^2(n)$. Поэтому $\mathcal{P}$-SPACE включает в себя также все языки, допускаемые НМТ с полиномиально ограниченной памятью. Так как каждая НМТ с полиномиально ограниченным временем имеет полиномиально ограниченную память, ясно, что $\mathcal{NP}$-TIME$\subseteq\mathcal{P}$-SPACE (рис. 10.13). Кроме того, поскольку $\mathcal{P}$-TIME$\subseteq\mathcal{NP}$-TIME$\subseteq\mathcal{P}$-SPACE, то из равенства $\mathcal{P}$-TIME$=\mathcal{P}$-SPACE следует, что $\mathcal{P}$-TIME$=\mathcal{NP}$-TIME, а потому распознавание детерминированным алгоритмом с полиномиально ограниченным временем работы любого языка из $\mathcal{P}$-SPACE еще менее вероятно, чем распознавание таким алгоритмом любого языка из $\mathcal{NP}$-TIME.

Можно указать довольно естественный язык L_r, *полный для* $\mathcal{P}$-SPACE. Так называется язык, который принадлежит $\mathcal{P}$-SPACE и удовлетворяет следующему условию: если он допускается какой-то детерминированной МТ с ограниченным функцией $T(n)$ временем работы, то для каждого языка $L\in\mathcal{P}$-SPACE найдется детерминированная МТ с временем работы, ограниченным функцией $T(p_L(n))$, где p_L — полином, зависящий от L. (Здесь слово "детерминированная" можно заменить на "недетерминированная".) Поэтому если $L_r\in\mathcal{P}$-TIME, то $\mathcal{P}$-TIME$=\mathcal{NP}$-TIME$=\mathcal{P}$-SPACE. Если $L_r\in\mathcal{NP}$-TIME, то $\mathcal{NP}$-TIME$=\mathcal{P}$-SPACE. Язык L_r — это множество регулярных выражений, дополнения которых представляют непустые множества.

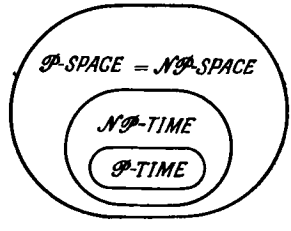


Рис. 10.13. Взаимосвязь памяти и времени.

Лемма 10.2. *Пусть* $M=(Q, T, I, \delta, b, q_0, q_f)$ *— одноленточная* ДМТ *с памятью, ограниченной полиномом* p. *Тогда найдется такой полином* p', *что если* $x \in I^*$ *и* $|x|=n$, *то за время* $p'(n)$ *можно построить регулярное выражение* R_x, *дополнение которого представляет пустое множество тогда и только тогда, когда* M *не допускает* x.

Доказательство. Доказательство очень похоже на доказательство теоремы 10.3. Там для описания вычислений машины Тьюринга использовались булевы функции. Здесь у нас будет язык регулярных выражений. Регулярные выражения можно записывать в компактной форме, и с их помощью удается выразить факты о последовательностях МО, существенно более длинных, чем сами регулярные выражения.

Регулярное выражение R_x можно представлять себе как способ описания последовательностей МО машины M, которые не допускают цепочку x. Для наших целей понадобится алфавит $\Delta = T \cup \{[qX] | q \in Q \text{ и } X \in T\}$. Символ $[qX]$ из $Q \times T$ представляет клетку входной ленты машины M, обозреваемую головкой, причем клетка содержит X, а M находится в состоянии q. Если M допускает x, то в соответствующей последовательности шагов на каждом шаге используется не более $p(|x|)$ клеток на ленте. Поэтому можно представлять МО цепочкой w_i, состоящей ровно из $p(n)$ символов, причем все они, кроме одного, взяты из T; этот один символ имеет вид $[qX]$. Последовательность шагов машины M можно представить цепочкой $\#w_1\#w_2\#\ldots w_k\#$ для некоторого $k \geqslant 1$, где $\#$ — новый разделительный символ и каждая цепочка w_i из Δ^* представляет некоторое МО, $|w_i| = p(n)$. При необходимости в это МО добавляются пустые символы (чтобы сделать длину цепочки w_i равной $p(n)$).

Мы хотим построить регулярное выражение R_x так, чтобы оно представляло те цепочки из $(\Delta \cup \{\#\})^*$, которые не описывают допускающие вычисления машины M для входа x. Поэтому R_x будет представлять все цепочки из $(\Delta \cup \{\#\})^*$ тогда и только тогда, когда M не допускает x. Цепочка $y \in (\Delta \cup \{\#\})^*$ не будет представлять допускающее вычисление машины M только тогда, когда выполнено одно или более из следующих четырех условий. Мы считаем, что $|x|=n$.

1. Цепочка y ни для какого $k \geqslant 1$ не имеет вида $\#w_1\#w_2\#\ldots\ldots w_k\#$, где $|w_i| = p(n)$ для каждого i, $1 \leqslant i \leqslant k$, и все символы цепочки w_i принадлежат T, за исключением одного, который имеет вид $[qX]$.
2. Начальное МО w_1 неправильно, т. е. цепочка y не начинается с $\#[q_0a_1]a_2\ldots a_n bb\ldots b\#$, где $x=a_1\ldots a_n$, а число пустых символов равно $p(n)-n$.
3. M никогда не попадает в заключительное состояние, т. е. никакой символ вида $[q_fX]$ не входит в y.

4. В y есть такие два соседних МО, что второе не получается из первого за один шаг работы машины M.

R_x будет иметь вид $A+B+C+D$, где A, B, C, D — регулярные выражения, представляющие множества, определяемые условиями 1—4 соответственно.

Полезно ввести сокращения для записи регулярных выражений. Если мы пользуемся алфавитом $S=\{c_1, c_2, \ldots, c_m\}$, то S будет обозначать регулярное выражение $c_1+c_2+\ldots+c_m$, а S^i — регулярное выражение $(S)(S)\ldots(S)$ (i раз), т. е. цепочки длины i в алфавите S. Заметим, что длина регулярного выражения, обозначенного через S, равна $2m-1$, а через S^i равна $i(2m+1)$. Аналогично, если $S_1—S_2=(d_1, d_2, \ldots, d_r)$, то $S_1—S_2$ будет обозначать регулярное выражение $d_1+d_2+\ldots+d_r$.

При таких соглашениях регулярное выражение A можно записать в виде

$$\begin{aligned} A = {} & \Delta^* + \Delta^* \# \Delta^* + \Delta(\Delta + \#)^* + (\Delta + \#)^* \Delta \\ & + (\Delta + \#)^* \# T^* \# (\Delta + \#)^* \\ & + (\Delta + \#)^* \# \Delta^* (Q \times T) \Delta^* (Q \times T) \Delta^* \# (\Delta + \#)^* \\ & + (\Delta + \#)^* \# \Delta^{p(n)+1} \Delta^* \# (\Delta + \#)^* \\ & + A_0 + A_1 + \ldots + A_{p(n)-1}, \end{aligned} \tag{10.7}$$

где A_i будут определены позже. Первые семь слагаемых в (10.7) представляют соответственно цепочки

1) без $\#$,
2) лишь с одним символом $\#$,
3) не начинающиеся с $\#$,
4) не кончающиеся на $\#$,
5) не содержащие символа из $Q\times T$ между двумя $\#$,
6) содержащие более одного символа из $Q\times T$ между двумя $\#$,
7) содержащие более $p(n)$ символов из Δ между символами $\#$.

Остальные слагаемые A_i определяются равенствами $A_i=(\Delta+\#)^* \#\Delta^i\#(\Delta+\#)^*$ и в совокупности представляют множество цепочек, содержащих менее $p(n)$ символов из Δ между двумя символами $\#$.

Предлагаем читателю проверить, что регулярное выражение (10.7) на самом деле представляет цепочки, удовлетворяющие условию 1. Кроме того, первые шесть слагаемых имеют фиксированную длину, зависящую только от M. Длина седьмого, очевидно, есть $O(p(n))$. Слагаемое A_i имеет длину $O(i)$, так что сумма длин всех A_i и, значит, длина выражения (10.7) есть $O(p^2(n))$, причем мультипликативная постоянная зависит только от M, но не от x. Далее, легко проверить, что выражение (10.7) легко выписать за время, полиномиально зависящее от n.

Теперь заметим, что для B, C и D достаточно написать выражения, представляющие все цепочки, которые удовлетворяют условию

2, 3 или 4, но нарушают условие 1, т. е. цепочки вида $\#w_1\#\ldots\ldots w_k\#$, где $|w_i|=p(n)$ и $w_i \in T^*(Q\times T)T^*$. Однако для простоты мы выпишем выражения, определяющие множества цепочек, которые удовлетворяют условию 2, 3 или 4 и нарушают условие 1, а также некоторых цепочек, удовлетворяющих не только условию 2, 3 или 4, но и условию 1. Так как цепочки последнего типа уже представлены в R_x в силу определения выражения A, их наличие или отсутствие в B, C и D несущественно.

Считая, что входная цепочка имеет вид $x=a_1a_2\ldots a_n$, полагаем $B=B_1+B_2+\ldots+B_{p(n)}$, где

$$B_1 = \#(\Delta - [q_0a_1])\,\Delta^{p(n)-1}\#(\Delta+\#)^*,$$

$$B_i = \begin{cases} \#\Delta^{i-1}(T-\{a_i\})\,\Delta^{p(n)-i}\#(\Delta+\#)^*, & 1<i\leqslant n, \\ \#\Delta^{i-1}(T-\{\mathrm{b}\})\,\Delta^{p(n)-i}\#(\Delta+\#)^*, & n<i\leqslant p(n). \end{cases}$$

Таким образом, B_i отличается от начального МО по крайней мере в i-м входном символе. Очевидно, B имеет длину $O(p^2(n))$, причем мультипликативная постоянная зависит только от M.

Положим $C=((\Delta+\#)-(\{q_f\}\times T))^*$. Длина этого выражения, конечно, зависит только от M.

Наконец, построим D. Пусть дана цепочка y, представляющая правильное (т. е. позволяющее фактически реализовать) вычисление машины M. Тогда y имеет вид $\#w_1\#w_2\#\ldots w_k\#$, где $|w_i|=p(n)$ для каждого i и w_{i+1} — это МО, которое получается из w_i за один шаг работы машины M. Заметим, что по данным трем последовательным символам $c_1c_2c_3$ цепочки y можно однозначно определить символ, назовем его $f(c_1c_2c_3)$, который должен появиться в цепочке на $p(n)+1$ символов правее c_2. Например, если c_1, c_2 и c_3 все принадлежат T, то c_2 не может измениться в следующем МО, так что $f(c_1c_2c_3)=c_2$. Если c_1 и c_2 принадлежат T, $c_3=[qX]$ — символ из $Q\times T$ и $\delta(q, X)=(p, X, L)$, то $f(c_1c_2c_3)=[pc_2]$. Остальные правила для определения f, включая те, что соответствуют случаю, когда c_1 или c_2 принадлежит $Q\times T$ или один из символов c_i равен $\#$, должны быть очевидны; оставляем их в качестве упражнения.

D представляет собой сумму членов

$$D_{c_1c_2c_3} = (\Delta+\#)^* c_1c_2c_3\,(\Delta+\#)^{p(n)-1}\,\bar{f}(c_1c_2c_3)\,(\Delta+\#)^*$$

для всех c_1, c_2 и c_3 из $\Delta\cup\{\#\}$, где $\bar{f}(c_1c_2c_3)=\Delta\cup\{\#\}-f(c_1c_2c_3)$. Ясно, что выражение D имеет длину $O(p(n))$ и его можно выписать за время $O(p(n))$, где мультипликативная постоянная зависит лишь от M.

Таким образом, регулярное выражение R_x можно построить за время $p'(n)$, где p' — полином, растущий по порядку как p^2. Более того, R_x представляет множество $(\Delta\cup\{\#\})^*$ тогда и только тогда, когда x не допускается машиной M. □

Итак, в лемме 10.2 построено регулярное выражение над алфавитом $\Delta \cup \{\#\}$, размер которого зависит от M. Мы хотим рассуждать о "языке регулярных выражений, дополнения которых (относительно их алфавитов) представляют непустые множества". Так как язык должен определяться над конечным алфавитом, мы будем кодировать регулярное выражение над конечным алфавитом Σ следующим образом:

1) $+$, $*$, $\varnothing$, ε и скобки обозначают сами себя,
2) i-й символ алфавита Σ (в произвольном порядке) обозначается через $\#w\#$, где w — десятичное представление числа i.

Таким образом, для кодирования любого регулярного выражения над любым алфавитом достаточно 17 символов из алфавита

$$\Gamma = \{+, *, \varnothing, \varepsilon, (,), \#, 0, 1, \ldots, 9\}.$$

Определим язык $L_r \subseteq \Gamma^*$ как множество кодов тех регулярных выражений R, дополнения которых представляют непустые множества. Если R — выражение над алфавитом Σ, то дополнение берется относительно Σ^*. Заметим, что если выражение R имеет длину $n \geqslant 2$, то длина его кода не превосходит $3n \log n$.

Лемма 10.3. *L_r принадлежит классу* $\mathcal{P}$-SPACE.

Доказательство. По теореме 10.1 достаточно построить НМТ M, которая допускает язык L_r, работая с полиномиально ограниченной памятью. Пусть R — регулярное выражение, код которого подан на вход машины M. В силу теоремы 9.2 по данному регулярному выражению длины n можно построить недетерминированный конечный автомат A с не более чем $2n$ состояниями, распознающий язык, представленный этим регулярным выражением. НМТ M сначала строит автомат A по коду выражения R, поданному машине M. Заметим, что по теореме 9.2 можно построить функцию переходов автомата A за время, не превосходящее $O(n^2)$.

Допустим, что дополнение множества, представленного выражением R, непусто. Тогда существует цепочка $x = a_1 \ldots a_m$, не принадлежащая языку, представленному выражением R. M угадывает ее символ за символом. Для хранения множества S_i состояний, в которых A может оказаться после прочтения $a_1 \ldots a_i$, $1 \leqslant i \leqslant m$, машина M использует массив из $2n$ битов. Она начинает с множества S_0 состояний, достижимых из начального состояния автомата A только с помощью ε-переходов. После того как она угадает $a_1 \ldots a_i$, автомат A может находиться в одном из состояний множества S_i. Когда M угадывает следующий входной символ a_{i+1}, она прежде всего вычисляет $T_{i+1} = \{s' \mid s' \in \delta_A(s, a_{i+1})$ и $s \in S_i\}$, где δ_A — функция переходов автомата A, а затем строит S_{i+1}, добавляя к T_{i+1} все состояния из $\delta_A(s', \varepsilon)$ для s' из T_{i+1}. (Обратите внимание на аналогию с алгоритмом 9.1.)

Когда M угадает $a_1a_2\ldots a_m$, она выяснит, что S_m не содержит заключительного состояния автомата A. Обнаружив это, машина M допускает свой вход, т. е. код регулярного выражения R. Таким образом, M допускает код выражения R тогда и только тогда, когда дополнение множества, представленного выражением R, непусто. □

Теперь покажем, что язык L_r, состоящий из кодов тех регулярных выражений, которые представляют множества с непустыми дополнениями, полон для полиномиальной памяти.

Теорема 10.14. *Если L_r допускается* ДМТ *с временной сложностью $T(n)\geqslant n$, то для каждого $L\in\mathcal{P}$-SPACE найдется такой полином p_L, что L допускается за время $T(p_L(n))$.*

Доказательство. Пусть M будет ДМТ, допускающей L. В силу леммы 10.2 и рассуждений, проведенных выше, существует алгоритм с полиномиально ограниченным временем работы, скажем с временной сложностью $p_1(n)$, который по входной цепочке x строит регулярное выражение R_x. Это выражение в фиксированном 17-символьном алфавите, и, разумеется, оно не длиннее $p_1(|x|)$. По лемме 10.2 его дополнение пусто тогда и только тогда, когда $x\in L$. По предположению можно проверить пустоту этого дополнения за $T(|R_x|)=T(p_1(|x|))$ шагов. Построение R_x занимает $p_1(|x|)$ шагов, а проверка пустоты — $T(p_1(|x|))$ шагов. Так как $T(n)\geqslant n$, то для завершения доказательства достаточно взять $p_L(n)=2p_1(n)$[1]). □

Следствие. *L_r принадлежит $\mathcal{P}$-*TIME *тогда и только тогда, когда $\mathcal{P}$-*TIME$=\mathcal{P}$-SPACE.

Доказательство. Если $\mathcal{P}$-TIME$=\mathcal{P}$-SPACE, то L_r принадлежит $\mathcal{P}$-TIME по лемме 10.3. Обратное следует из теоремы 10.14 с полиномиальной функцией $T(n)$. □

Теорема 10.15. *Если L_r допускается* НМТ *с временной сложностью $T(n)\geqslant n$, то для каждого $L\in\mathcal{P}$-SPACE найдется такой полином p_L, что L допускается* НМТ *за время $T(p_L(n))$.*

Доказательство. Аналогично доказательству теоремы 10.14. □

Следствие. *L_r принадлежит $\mathcal{NP}$-*TIME *тогда и только тогда, когда $\mathcal{NP}$-*TIME$=\mathcal{P}$-SPACE.

---

[1]) В действительности при сделанных предположениях о росте T мы не можем получить требуемое неравенство $m+T(m)\leqslant T(2m)$. Для завершения доказательства можно использовать упр. 11.9.— *Прим. ред.*

УПРАЖНЕНИЯ

10.1. Выпишите все правильные последовательности шагов НМТ, изображенной на рис. 10.1, для входа 10101. Допускает ли эта НМТ указанный вход?

10.2. Неформально опишите НМТ или недетерминированную программу на Упрощенном Алголе, допускающую

(а) множество таких цепочек $10^{i_1}10^{i_2}\ldots10^{i_k}$, что $i_r=i_s$ для некоторых $1\leqslant r<s\leqslant k$,

(б) множество таких цепочек xcy, что x и y принадлежат $\{a, b\}^*$ и x — подцепочка в y,

(в) множество цепочек, как в (б), но x — подпоследовательность в y.

10.3. Опишите РАМ-программу, моделирующую недетерминированную РАМ-программу.

10.4. Пусть M обозначает $(m\times n)$-матрицу из нулей и единиц. Напишите программу на Упрощенном Алголе, которая находила бы такую наименьшую $(s\times n)$-подматрицу S матрицы M, что если $M[i, j]=1$, то $S[k, j]=1$ для некоторого $1\leqslant k\leqslant s$. Какова временная сложность вашей программы?

10.5. Покажите, что следующие функции конструируемы по емкости, неформально описав для этого ДМТ, помещающую маркер в нужную клетку одной из своих лент:

(а) n^2, (в) 2^n,
(б) n^3-n^2+1, (г) $n!$.

10.6. Покажите, что если одноленточная НМТ с емкостной сложностью $S(n)$, имеющая s состояний и t ленточных символов, допускает слово длины n, то она допускает его не более чем за $sS(n)t^{S(n)}$ шагов.

***10.7.** Покажите, как вычислить на РАМ значение булевой формулы за время $O(n)$.

10.8. Покажите, что язык, состоящий из булевых функций, не являющихся тавтологиями [1]), NP-полон.

10.9. Покажите, что *задача об изоморфизме подграфу* ("Изоморфен ли данный неориентированный граф G некоторому подграфу данного неориентированного графа G'?") NP-полна. *Указание*: Воспользуйтесь NP-полнотой задачи о клике или о гамильтоновом цикле.

10.10. Покажите, что *задача о ранце* ("Существует ли для данной последовательности целых чисел $S=i_1, i_2,\ldots, i_n$ и целого

[1]) *Тавтологией* называется булева формула, принимающая значение 1 на всех наборах значений ее переменных.

числа k подпоследовательность в S, сумма членов которой равна k?") NP-полна. *Указание*: Используйте задачу о точном покрытии.

***10.11.** Покажите, что *задача о разбиении* (задача о ранце из упр. 10.10 с $\sum_{j=1}^{n} i_j = 2k$) NP-полна.

10.12. Покажите, что *задача о коммивояжере* ("По данному графу G с целочисленными весами ребер найти цикл, который включает каждый узел и сумма весов ребер которого не превосходит k") NP-полна.

****10.13.** Покажите, что NP-полна задача распознавания следующего свойства: регулярное выражение без * (т. е. такое, в котором употребляются только операции + и ·) не представляет всех цепочек некоторой фиксированной длины. *Указание*: Трансформируйте в эту задачу о регулярных выражениях задачу КНФ-выполнимости.

****10.14.** Покажите, что NP-полна задача распознавания следующего свойства: регулярное выражение над алфавитом {0} не представляет 0^*.

****10.15.** Пусть $G=(V, E)$ — неориентированный граф и v_1, v_2 — два различных узла. *Разрезом* для v_1 и v_2 называется такое подмножество $S \subseteq E$, что любой путь между v_1 и v_2 содержит элемент из S. Покажите, что NP-полна задача распознавания того, есть ли в графе такой разрез размера k для v_1 и v_2, что обе части графа содержат равные количества узлов.

****10.16.** *Одномерная задача об упаковке* состоит в следующем. Пусть $G=(V, E)$ — неориентированный граф и k — положительное целое число. Существует ли такая нумерация $v_1, v_2, \ldots, v_n$ узлов из V, что

$$\sum_{(v_i, v_j) \in E} |i-j| \leqslant k?$$

Покажите, что эта задача NP-полна.

****10.17.** Докажите, что задача о раскраске остается NP-полной, даже если ограничить k числом 3, а наибольшую степень каждой вершины — числом 4.

****10.18.** Выполните упр. 10.17 для планарных графов. *Указание*: Покажите, что планарный граф на рис. 10.14 можно раскрасить в три цвета только тогда, когда v_1 и v_1' раскрашены одинаково, а также v_2 и v_2' раскрашены одинаково. Скомбинируйте этот результат с вашим ответом на упр. 10.17.

***10.19.** Докажите NP-полноту задачи о клике, непосредственно представляя вычисления НМТ вместо того, чтобы трансформировать ее из 3-КНФ-выполнимости.

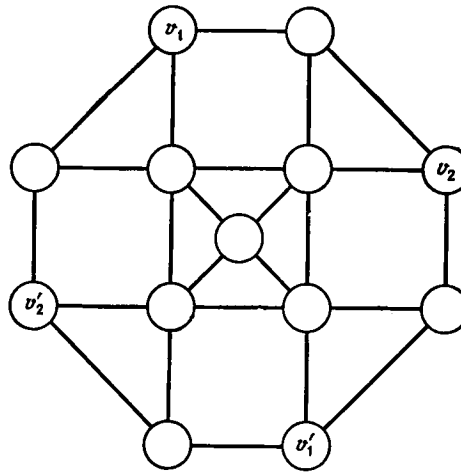


Рис. 10.14. Планарный граф.

10.20. Рассмотрим ориентированный граф с двумя выделенными узлами s и t. Припишем каждому ребру целочисленную "пропускную способность". Можно построить максимальный поток из s в t, многократно находя путь из s в t и увеличивая поток вдоль него до максимума, разрешаемого пропускными способностями его ребер. Покажите, что задача нахождения наименьшего множества путей, на которых следует увеличивать поток, NP-полна. *Указание*: Рассмотрите граф типа изображенного на рис. 10.15 и свяжите эту задачу с задачей о ранце.

****10.21.** *Задача о расписании работ с равными временами выполнения* состоит в следующем. Даны множество работ $S=(J_1,\ldots, J_n)$, лимит времени t, число процессоров p и частичный порядок $<$ на S. Существует ли *расписание* работ из S, требующее не более t единиц времени, т. е. такое отображение f из S в $\{1, 2,\ldots, t\}$, что в каждое целое число отображается не более p работ, и если $J<J'$, то $f(J)<f(J')$? Покажите, что эта задача NP-полна.

10.22. Покажите, что каждая из задач, перечисленных в теореме 10.2, принадлежит классу $\mathcal{NP}$-TIME.

10.23. Пусть $p_1(x)$ и $p_2(x)$ — полиномы. Покажите, что некоторый полином для каждого x принимает значение, большее $p_1(p_2(x))$.

10.24. Выпишите выражение E_{ijkt}, введенное в доказательстве теоремы 10.3, для $\delta(q_k, x_j)=\{(q_3, X_4, L), (q_9, X_2, R)\}$.

****10.25.** Постройте алгоритм полиномиальной сложности для проверки 2-выполнимости.

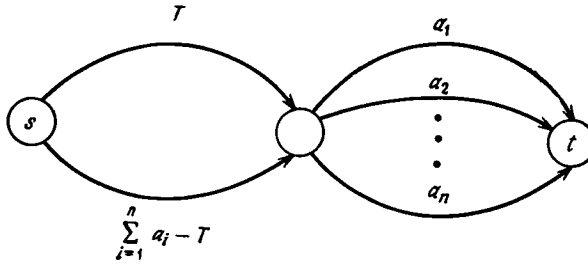


Рис. 10.15.

***10.26.** Известно, что некоторые частные случаи задачи об изоморфизме графов (например, изоморфизм планарных графов) легкие. Другие столь же трудны, как и общая задача. Покажите, что задача об изоморфизме корневых ориентированных ациклических графов имеет ту же сложность, что и задача об изоморфизме произвольных графов.

***10.27.** *Контекстным* называется язык, допускаемый НМТ с емкостной сложностью $n+1$ (эта машина называется *линейно ограниченным автоматом;* см. Хопкрофт, Ульман [1969]). Покажите, что задача выяснения, допускает ли данный линейно ограниченный автомат данный вход, полна для полиномиальной памяти, т. е. задача принадлежности цепочки произвольному контекстному языку полна для $\mathcal{P}$-SPACE.

10.28. Покажите, что задача эквивалентности двух регулярных выражений полна для полиномиальной памяти.

****10.29.** Опишите алгоритм, допускающий множество L, кодов всех регулярных выражений R, представляющих непустое множество, который можно реализовать на ДМТ с линейно ограниченной памятью.

Проблемы для исследования

10.30. Одна из очевидных открытых проблем — выяснить, выполняются ли равенства $\mathcal{P}$-TIME$=\mathcal{NP}$-TIME и $\mathcal{NP}$-TIME$=\mathcal{P}$-SPACE. Если учесть объем работы, проделанный в поисках алгоритмов, решающих NP-полные задачи за полиномиальное время, то похоже, что эта проблема по меньшей мере столь же сложна, как некоторые классические проблемы математики, такие, как гипотеза Ферма ("Имеет ли уравнение $x^n+y^n=z^n$ решение в целых числах для $n\geqslant 3$?") или проблема четырех красок [1]).

[1]) В *Bull. Amer. Math. Soc.*, 82, № 5 (1976), 711—712, опубликовано сообщение о решении проблемы четырех красок, которое нашли с помощью ЭВМ американские математики Аппель и Хакен. Более подробное изложение их результатов см. в *Illinois J. Math.*, 21, № 3 (1977), 429—567.— *Прим. ред.*

10.31. В случае неудачи с 10.30 было бы интересно получить нетривиальный результат, дающий такую функцию $T(n)$, что всякий язык из $\mathcal{NP}$-TIME распознается детерминированной машиной с временной сложностью не более $T(n)$. Это не показано даже для $T(n)=2^n$.

Замечания по литературе

Дополнительную информацию о недетерминированных машинах Тьюринга можно найти в книге Хопкрофта, Ульмана [1969]. Теорему 10.1, устанавливающую связь между емкостными сложностями детерминированных и недетерминированных машин, доказал Сэвич [1970].

Ключевая теорема о NP-полноте задачи выполнимости принадлежит Куку [1971б]. Много классических NP-полных задач привел Карп [1972], ясно продемонстрировав важность понятия NP-полноты. С тех пор к семейству известных NP-полных задач были добавлены новые, найденные Сахни [1972], Сети [1973], Ульманом [1973], Раундзом [1973], Ибаррой, Сахни [1973], Хантом, Розенкранцем [1974], Гэри, Джонсоном, Стокмейером [1974], Бруно, Сети [1974] и многими другими. Интересно, что до Кука в нескольких работах была показана полиномиальная взаимосвязь между некоторыми NP-полными задачами, но объем всего класса не был осознан. Например, Данциг, Блаттнер, Рао [1966] установили связь между задачами о коммивояжере и о кратчайшем пути с неотрицательными весами. Диветти, Грасселли [1968] описали связь между задачами о покрытии множествами и о множестве ребер, разрезающих циклы.

Задачи, полные для $\mathcal{P}$-SPACE, впервые рассмотрели Мейер, Стокмейер [1972]. Первый язык, полный для памяти, неявно описал Сэвич [1971], определив язык (множество проходимых лабиринтов), полный для $\log n$-памяти. Джоунс [1973] и Стокмейер, Мейер [1973] рассмотрели ограниченные формы сводимости задач за полиномиальное время. Бук [1972, 1974] доказал несравнимость некоторых классов сложности.

Упр. 10.8 и 10.9 взяты у Кука [1971б], упр. 10.10 и 10.12 — у Карпа [1972], упр. 10.13 — у Стокмейера, Мейера [1973] и Ханта [1973а], упр. 10.14 и 10.28— у Стокмейера, Мейера [1973], а упр. 10.15—10.18 — у Гэри, Джонсона, Стокмейера [1974]; рис. 10.14 представляет собой улучшенный вариант, предложенный Фишером. Доказательство NP-полноты задачи 3-раскрашиваемости планарных графов появилось в работе Стокмейера [1973]. Упр. 10.20 заимствовано из работы Ивена [1973], упр. 10.21 — Ульмана [1973], упр. 10.25 — Кука [1971б], а редукция, нужная для решения упр. 10.27, взята у Карпа [1972]. Доказательство теоремы 10.13 предложил нам Ивен.

11 НЕКОТОРЫЕ ДОКАЗУЕМО ТРУДНО РАЗРЕШИМЫЕ ЗАДАЧИ

В этой главе мы приводим доказательства того, что для двух классов расширенных регулярных выражений задача пустоты дополнения трудно разрешима, т. е. любой алгоритм, решающий ее для любого из двух классов, тратит по меньшей мере экспоненциальное время. Для одного из этих классов устанавливается нижняя оценка, существенно превышающая экспоненту. В частности, будет показано, что эта задача требует времени, большего чем

$$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$$

для любой конечной башни из двоек. Прежде чем устанавливать нижние оценки, мы рассмотрим результаты о иерархиях, показывающие, что "чем больше времени или памяти можно использовать для вычислений, тем больше языков можно распознать".

11.1. ИЕРАРХИИ ПО СЛОЖНОСТИ

В гл. 10 мы видели, что некоторые задачи полны для недетерминированного полиномиального времени или же для полиномиальной памяти. Чтобы доказать полноту конкретных задач, мы показывали, как в терминах данной конкретной задачи представляется произвольная задача из класса $\mathcal{NP}$-TIME или $\mathcal{P}$-SPACE.

Техника доказательств представляла собой, по существу, технику моделирования. Например, мы показали, что задача выполнимости булевых формул NP-полна, а задача пустоты дополнения для регулярных выражений полна для полиномиальной памяти, причем в обоих случаях результат получался прямым вложением вычислений на машинах Тьюринга в частные случаи этих задач. Полноту других задач мы показали, сводя к ним какую-нибудь задачу, о которой уже известно, что она полна для соответствующего класса задач. Таким образом, мы показали, что оба класса $\mathcal{NP}$-TIME и $\mathcal{P}$-SPACE

содержат "самые трудные" задачи, т. е. задачи, сложность которых по меньшей мере столь же велика, как и сложность любой задачи из этого класса.

Однако сколь ни сильны косвенные доводы, еще никому не удалось найти в $\mathcal{NP}$-TIME или $\mathcal{P}$-SPACE задачу, о которой можно было бы доказать, что она не принадлежит $\mathcal{P}$-TIME. Более того, похоже, что техники гл. 10 хватает лишь для доказательства того, что данная задача по меньшей мере столь же трудна, как и любая другая задача из некоторого класса. Чтобы действительно доказать, что данная задача не принадлежит $\mathcal{P}$-TIME, нужна техника, которая позволила бы показать, что существует хотя бы один язык, не допускаемый никакой детерминированной машиной Тьюринга за полиномиальное время. Чаще всего для подобной цели применяется диагонализация. Хотя этот способ, по-видимому, не достаточно силен, чтобы доказать, что $\mathcal{P}$-TIME$\neq\mathcal{NP}$-TIME, с его помощью удалось получить результаты о иерархии как по емкостной, так и по временной сложностям для обоих типов машин Тьюринга — детерминированных и недетерминированных. Каждая теорема о иерархии имеет следующий вид: для данных "хороших" функций $f(n)$ и $g(n)$, таких, что $f(n)$ растет "быстрее" $g(n)$, существует язык со сложностью $f(n)$, но не $g(n)$.

11.2. ИЕРАРХИЯ ПО ЕМКОСТНОЙ СЛОЖНОСТИ ДЛЯ ДЕТЕРМИНИРОВАННЫХ МАШИН ТЬЮРИНГА

Здесь мы докажем теорему о иерархии по емкостной сложности (иерархии по памяти) для детерминированных машин Тьюринга. Можно доказать аналогичные результаты для времени и для недетерминированных машин Тьюринга, но поскольку для наших целей они не нужны, мы оставляем их в качестве упражнений. Вспомним, что в силу следствия 3 леммы 10.1, если язык L допускается k-ленточной ДМТ с емкостной сложностью $S(n)$, он допускается и одноленточной ДМТ с емкостной сложностью $S(n)$. Таким образом, можно ограничиться рассмотрением одноленточных ДМТ.

Чтобы получить результат о иерархии по емкостной сложности, надо *перечислить* ДМТ, т. е. каждому неотрицательному целому числу i поставить в соответствие единственную ДМТ. Кроме того, каждая ДМТ должна появляться в этом перечислении бесконечно много раз. Мы будем перечислять только одноленточные ДМТ с входным алфавитом $\{0, 1\}$, ибо это все, что нам нужно. Для перечисления всех машин Тьюринга достаточно просто обобщить этот метод.

Не умаляя общности, можно принять следующие соглашения о способе представления одноленточных ДМТ.

1. Состояния имеют имена $q_1, q_2, \ldots, q_s$ для некоторого s, причем q_1 — начальное и q_s — допускающее состояния.

2. Входным алфавитом служит {0, 1}.
3. Ленточным алфавитом служит $\{X_1, X_2, \ldots, X_t\}$ для некоторого t, где X_1=b, X_2=0 и X_3=1.
4. Функция переходов δ представляет собой список пятерок вида $(q_i, X_j, q_k, X_l, D_m)$, означающих, что $\delta(q_i, X_j)=(q_k, X_l, D_m)$, где q_i и q_k — состояния, X_j и X_l — ленточные символы и D_m — направление сдвига головки: L (влево), R (вправо) и S (остается на месте) при m=0, 1 и 2 соответственно. Считаем, что такая пятерка кодируется цепочкой $10^i10^j10^k10^l10^m1$.
5. Сама машина Тьюринга кодируется цепочкой, получающейся конкатенацией в любом порядке кодов пятерок, представляющих ее функцию переходов. Спереди можно добавлять, если надо, дополнительные единицы. Результатом такого кодирования будет начинающаяся с единицы цепочка из нулей и единиц, которую можно интерпретировать как целое число.

Любое целое число, которое нельзя декодировать, считается представлением тривиальной машины Тьюринга с пустой функцией переходов. Каждая одноленточная ДМТ будет появляться в этом перечислении бесконечно много раз, поскольку, имея какую-то ДМТ, можно к началу ее кода приписывать единицы и получать все бóльшие и бóльшие целые числа, представляющие одно и то же множество пятерок.

Построим четырехленточную ДМТ M_0, рассматривающую свою входную цепочку x одновременно как код одноленточной ДМТ M и как вход для M. Мы построим M_0 так, чтобы для каждой ДМТ M с данной сложностью нашлась хотя бы одна входная цепочка, которую допускает M_0, но отвергает M, или наоборот. Одна из способностей, которыми наделена машина M_0, состоит в возможности моделировать произвольную машину Тьюринга по ее описанию. Машина M_0 будет распознавать, допускает ли машина Тьюринга M входную цепочку x так, что при этом используется не более $S_1(|x|)$ клеток, где S_1 — некоторая функция. Если M допускает x, укладываясь в $S_1(|x|)$ клеток, то M_0 не допускает x. В противном случае M_0 допускает x. Таким образом, либо поведение машины M_0 отличается от поведения i-й ДМТ на входе x, являющемся двоичным представлением числа i, либо i-я ДМТ на входе x использует более $S_1(|x|)$ клеток.

Говорят, что M_0 представляет *диагонализацию* по всем ДМТ с емкостной сложностью $S_1(n)$, ибо если вообразить двумерную таблицу, (i, j)-й элемент которой показывает, допустила ли i-я машина Тьюринга вход j, то поведение машины M_0 будет отличаться от поведения некоторых машин Тьюринга на диагонали этой таблицы. В частности, M_0 будет отличаться от машин Тьюринга, допускающих свой вход с емкостной сложностью, не большей $S_1(n)$. В силу след-

ствия 3 леммы 10.1 существует одноленточная ДМТ M_0', эквивалентная M_0 и имеющая ту же емкостную сложность Поскольку сама машина M_0' тоже представлена в таблице (скажем, это k-я ДМТ при некотором k) и она не может отличаться от себя, то можно заключить, что емкостная сложность машин M_0 и M_0' не есть $S_1(n)$. Фактическая конструкция M_0 усложнена из-за нашего желания построить M_0 так, чтобы она имела такую емкостную сложность $S_2(n)$, что $S_1(n)$ и $S_2(n)$ почти одинаковы.

Определение. Пусть $f(n)$ — произвольная функция. Обозначим через $\inf\limits_{n\to\infty} f(n)$ предел при $n\to\infty$ наибольшей нижней грани последовательности $f(n)$, $f(n+1)$,. . ..

Пример 11.1. Так как $(n^2+1)/n^2$ монотонно убывает, то

$$\inf_{n\to\infty}\frac{n^2+1}{n^2}=\lim_{n\to\infty}\frac{n^2+1}{n^2}=1.$$

В качестве второго примера рассмотрим $f(n)=1-1/n$, если n не есть степень числа 2, и $f(n)=n$ в противном случае. Тогда $\inf\limits_{n\to\infty} f(n)=1$, поскольку наибольшая нижняя грань последовательности $f(n)$, $f(n+1)$, . . ., равна либо $1-1/n$, если n не есть степень числа 2, либо $1-1/(n+1)$, если n — степень числа 2. □

Теорема 11.1. *Пусть $S_1(n)\geqslant n$ и $S_2(n)\geqslant n$ — две функции, конструируемые по памяти, причем*

$$\inf_{n\to\infty}\frac{S_1(n)}{S_2(n)}=0.$$

Тогда существует язык L, допускаемый некоторой ДМТ *с емкостной сложностью $S_2(n)$, но не допускаемый никакой* ДМТ *с емкостной сложностью $S_1(n)$*[1]).

Доказательство. Пусть M_0 — четырехленточная ДМТ, которая работает на входной цепочке x длины n следующим образом.

1. M_0 отмечает $S_2(n)$ клеток на каждой ленте. После этого всякий раз, когда любая ее головка пытается выйти за отмеченные клетки, M_0 останавливается, не допуская вход.

[1]) В литературе часто приводится определение емкостной сложности машины Тьюринга, при котором не учитывается число клеток, просмотренных на входной ленте,— ведь на входной ленте нельзя изменять символы. При таком определении можно также рассматривать и функции для емкостной сложности, меньшие n, и из посылки теоремы убрать условия $S_1(n)\geqslant n$ и $S_2(n)\geqslant n$. Поскольку здесь нас интересуют лишь большие емкостные сложности, то результат для сложностей, меньших n, не стóит дополнительных деталей, нужных для него, и мы его опускаем. В этой теореме можно ослабить ограничение на конструируемость $S_1(n)$ по памяти.

2. Если x не является кодом никакой одноленточной ДМТ, M_0 останавливается, не допуская вход.
3. В прочих случаях пусть x — код ДМТ M. M_0 определяет число t используемых машиной M символов на ленте и число s ее состояний. Третья лента машины M_0 может играть роль "промежуточной" памяти при вычислении t. Затем M_0 разбивает свою вторую ленту на $S_1(n)$ блоков по $\lceil \log t \rceil$ клеток в каждом, эти блоки отделяются друг от друга клетками, содержащими маркер #; таким образом, всего клеток $(1+\lceil \log t \rceil)S_1(n)$, если $(1+\lceil \log t \rceil)S_1(n) \leqslant S_2(n)$. Каждый символ на ленте машины M будет закодирован двоичным числом в соответствующем блоке на второй ленте машины M_0. Вначале M_0 помещает свой вход в виде его двоичного кода в блоки ленты 2, а неиспользованные блоки заполняет кодом пустого символа.
4. На ленте 3 M_0 образует блок из $\lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil S_1(n)$ клеток и вначале записывает в них нули; здесь снова предполагается, что число требуемых клеток не превосходит $S_2(n)$. Лента 3 играет роль счетчика, в который можно помещать числа вплоть до $sS_1(n)t^{S_1(n)}$.
5. M_0 моделирует машину M, используя ленту 1 (т. е. свою входную ленту) для определения шагов машины M и ленту 2 для моделирования ленты машины M. Номера шагов машины M записываются в виде двоичных чисел в блок на ленте 3, а на ленте 4 хранится состояние машины M. Если M допускает свой вход, то M_0 останавливается, не допуская свой. M_0 допускает вход, если M останавливается, не допуская свой вход, или если для моделирования работы M машина M_0 пытается занять больше клеток на ленте 2, чем отведено, или если переполняется счетчик на ленте 3, т. е. число шагов, сделанное машиной M, превосходит $sS_1(n)t^{S_1(n)}$.

Машина Тьюринга M_0, описанная выше, обладает емкостной сложностью $S_2(n)$ и допускает некоторый язык L. Допустим, что L допускается какой-то ДМТ M_i с емкостной сложностью $S_1(n)$. В силу следствия 3 леммы 10.1 можно считать машину M_i одноленточной. Пусть M_i имеет s состояний и t символов на ленте. Записывая одну за другой пятерки, изображающие команды машины M_i и добавляя при необходимости единицы к началу полученной цепочки, можно построить цепочку w длины n, представляющую машину M_i, причем n столь велико, что

$$S_2(n) > \mathrm{MAX}\,[(1+\lceil \log t \rceil)S_1(n),\ \lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil S_1(n)].$$

Равенство $\inf\limits_{n\to\infty} [S_1(n)/S_2(n)]=0$ гарантирует, что такое n можно найти.

Теперь, когда на вход машины M_0 подается цепочка w, у M_0 достаточно места для моделирования работы M_i. Она допускает вход w тогда и только тогда, когда M_i не допускает его. Но мы предположили, что машина M_i допускает L, т. е. ее выход совпадает с выходом M на всех входах. Заключаем отсюда, что машина M_i невозможна, т. е. язык L не допускается никакой ДМТ с емкостной сложностью $S_1(n)$. □

Типичное приложение теоремы 11.1 состоит, например, в доказательстве существования языка, допускаемого ДМТ с емкостной сложностью $n^2 \log n$, но не допускаемого никакой ДМТ с емкостной сложностью n^2. Некоторые другие приложения будут даны в оставшейся части этой главы.

11.3. ЗАДАЧА, ТРЕБУЮЩАЯ ЭКСПОНЕНЦИАЛЬНЫХ ВРЕМЕНИ И ПАМЯТИ

В разд. 10.6 мы изучили задачу пустоты дополнения для регулярных выражений и показали, что она полна для полиномиальной емкости. Поскольку класс регулярных множеств замкнут относительно пересечения и дополнения, то добавление знаков операций пересечения $\cap$ и дополнения $\neg$ к системе обозначений для регулярных выражений не увеличивает класса множеств, которые можно описать такими выражениями. Однако использование этих знаков сильно сокращает длину выражений, необходимых для описания некоторых регулярных множеств. Из-за возможности такого сокращения для решения ряда задач с расширенными регулярными выражениями требуется еще больше времени, чем для решения задач с "обычными" регулярными выражениями (при условии, что время измеряется как функция длины данного выражения).

Определение. *Расширенное регулярное выражение* над алфавитом Σ определяется следующим образом.

1. ε, $\varnothing$ и a из Σ являются расширенными регулярными выражениями, представляющими соответственно $\{\varepsilon\}$, пустое множество и $\{a\}$.
2. Если R_1 и R_2 — расширенные регулярные выражения, представляющие соответственно языки L_1 и L_2, то (R_1+R_2), $(R_1 \cdot R_2)$, (R_1^*), $(R_1 \cap R_2)$ и $(\neg R_1)$ — тоже расширенные регулярные выражения, представляющие соответственно языки $L_1 \cup L_2$, L_1L_2, L_1^*, $L_1 \cap L_2$ и $\Sigma^* - L_1$.

Можно в расширенных регулярных выражениях опускать пары скобок, если принять соглашение, что приоритеты операций возрастают в порядке

$$+ \quad \cap \quad \neg \quad \cdot \quad *.$$

Далее, знак операции · обычно не пишется. Например $a\neg b^*+c\cap d$ означает

$$((a\cdot(\neg(b^*)))+(c\cap d)).$$

Если расширенное регулярное выражение не содержит знаков дополнения, то его называют *полурасширенным*. *Задача пустоты дополнения* для полурасширенных регулярных выражений состоит в выяснении, пусто ли дополнение множества, представленного данным выражением R (или, что эквивалентно, представляет ли R все цепочки во входном алфавите). Например, регулярное выражение

$$b^*+b^*aa^*(\varepsilon+b(a+b)(a+b)^*)+b^*aa^*b$$

представляет множество $(a+b)^*$, поскольку

$$b^*+b^*aa^*(\varepsilon+b(a+b)(a+b)^*)=\neg b^*aa^*b.$$

Приступим к доказательству того, что задача пустоты дополнения для полурасширенных регулярных выражений не принадлежит классу $\mathcal{P}$-SPACE и, значит, не принадлежит классу $\mathcal{NP}$-TIME, ибо $\mathcal{NP}$-TIME содержится в $\mathcal{P}$-SPACE. Фактически мы покажем, что существуют такие полурасширенные регулярные выражения длины n, что решение задачи пустоты дополнения для них требует память (и, следовательно, время) по меньшей мере $c'c^{\sqrt{n/\log n}}$, где $c'>0$ и $c>1$— некоторые постоянные.

В разд. 11.4 мы увидим, что задача пустоты дополнения для всего класса расширенных регулярных выражений гораздо труднее, чем для полурасширенных выражений. В предвидении этих дальнейших результатов мы будем излагать многие результаты настоящего раздела в терминах всего класса расширенных регулярных выражений.

Кроме сокращений, введенных в разд. 10.6, будем использовать $\sum_{i=1}^{k}R_i$ для обозначения расширенного регулярного выражения $R_1+R_2+\ldots+R_k$. Мы также будем писать R^+ вместо RR^*. Когда мы будем говорить о длине выражения, то будем подразумевать длину исходного выражения, записанного без этих сокращений.

Теперь введем понятие измерителя. Пусть Σ — алфавит и x — произвольная цепочка в Σ^*. Положим ЦИКЛ$(x)=\{zy|x=yz$, где $y\in\Sigma^*$ и $z\in\Sigma^*\}$. Пусть $\#$ — специальный маркер, не принадлежащий Σ. Множество ЦИКЛ$(x\#)$ называется *измерителем* с длиной $|x\#|$. Таким образом, измеритель — это множество всех циклических перестановок цепочки $x\#$. Когда это не будет вызывать недоразумений, мы будем называть измерителем саму цепочку $x\#$.

С помощью измерителя мы будем определять длину МО в правильном вычислении машины Тьюринга. Сначала покажем, как относительно короткими полурасширенными регулярными выражениями можно представить некоторые длинные измерители.

Лемма 11.1. *Для каждого $k\geqslant 1$ существует измеритель с длиной, большей 2^k, который можно представить таким полурасширенным регулярным выражением R, что $|R|\leqslant ck^2$ для некоторой постоянной c, не зависящей от k.*

Доказательство. Пусть $A=\{a_0, a_1, \ldots, a_k\}$ — алфавит из $k+1$ различных символов. Положим $x_0=a_0a_0$ и $x_i=x_{i-1}a_ix_{i-1}a_i$ для $1\leqslant i<k$. Тогда

$$x_l=(\ldots((a_0^2a_1)^2a_2)^2\ldots a_l)^2.$$

Длина цепочки x_0 равна 2, а длина цепочки x_l больше удвоенной длины цепочки x_{i-1}. Поэтому длина цепочки x_{k-1} не меньше 2^k для $k\geqslant 1$.

ЦИКЛ($x_{k-1}a_k$) — искомый измеритель. Осталось показать, что некоторое короткое полурасширенное регулярное выражение представляет множество ЦИКЛ($x_{k-1}a_k$).

Для $0\leqslant i\leqslant k$ пусть $A_i=(a_0+a_1+\ldots+a_i)$ и $A-A_i=(a_{i+1}+a_{i+2}+\ldots+a_k)$. Положим

$$\begin{aligned}R_0=a_0a_0&[(A-A_0)^+a_0^2]^*(A-A_0)^+\\&+a_0[(A-A_0)^+a_0^2]^*(A-A_0)^+a_0\\&+\ [(A-A_0)^+a_0^2]^*(A-A_0)^+a_0a_0(A-A_0)^*\end{aligned}$$

и

$$\begin{aligned}R_i=A_{i-1}^+a_iA_{i-1}^+a_i&[(A-A_i)^+(A_{i-1}^+a_i)^2]^*(A-A_i)^+A_{i-1}^*\\&+a_iA_{i-1}^+a_i[(A-A_i)^+(A_{i-1}^+a_i)^2]^*(A-A_i)^+A_{i-1}^+\\&+\ A_{i-1}^+a_i[(A-A_i)^+(A_{i-1}^+a_i)^2]^*(A-A_i)^+A_{i-1}^+a_iA_{i-1}^*\\&+\quad a_i[(A-A_i)^+(A_{i-1}^+a_i)^2]^*(A-A_i)^+A_{i-1}^+a_iA_{i-1}^+\\&+\quad\ [(A-A_i)^+(A_{i-1}^+a_i)^2]^*(A-A_i)^+A_{i-1}^+a_iA_{i-1}^+a_i(A-A_i)^*\end{aligned}$$

для $1\leqslant i\leqslant k-1$.

Мы утверждаем, что полурасширенное регулярное выражение

$$R=R_0\cap R_1\cap\ldots\cap R_{k-1}\cap A_{k-1}^*a_kA_{k-1}^*$$

представляет ЦИКЛ($x_{k-1}a_k$). Для доказательства покажем индукцией по i, что любая цепочка из множества, представленного выражением $R_0\cap R_1\cap\ldots\cap R_i$, имеет вид

$$\begin{aligned}&y_2[(A-A_i)^+x_i]^*(A-A_i)^+y_1\\&+[(A-A_i)^+x_i]^*(A-A_i)^+x_i(A-A_i)^*,\end{aligned}$$

где $y_1y_2=x_i$.

Базис индукции. Для $i=0$ видно, что R_0 представляет в точности цепочки вида

$$\begin{aligned}&a_0^j[(A-A_0)^+a_0^2]^*(A-A_0)^+a_0^{2-j}\\&+[(A-A_0)^+a_0^2]^*(A-A_0)^+a_0a_0(A-A_0)^*\end{aligned}$$

для некоторого $0\leqslant j\leqslant 2$. Поскольку $x_0=a_0^2$, утверждение верно.

Шаг индукции. Предположим, что $R_0 \cap R_1 \cap \ldots \cap R_{i-1}$ представляет все цепочки вида

$$y_2'[(A-A_{i-1})^+ x_{i-1}]^* (A-A_{i-1})^+ y_1' + [(A-A_{i-1})^+ x_{i-1}]^* (A-A_{i-1})^+ x_{i-1} (A-A_{i-1})^*, \quad (11.1)$$

где $y_1'y_2'=x_{i-1}$.

Рассмотрим цепочку z из пересечения множества цепочек (11.1) и множества, представленного выражением R_i. Каждая подцепочка цепочки z, принадлежащая A_{i-1}^+ и имеющая максимальную длину, должна совпадать с x_{i-1}, кроме тех случаев, когда она является началом или концом этой цепочки (тогда она равна y_2' или y_1'). Поэтому каждое вхождение A_{i-1}^+ в R_i можно заменить на x_{i-1} (если только оно не в начале и не в конце этого регулярного выражения), не изменяя пересечения множеств (11.1) и R_i. Вхождения A_{i-1}^+ и A_{i-1}^* в начале и конце выражения R_i можно заменить на y_2' и y_1' соответственно. Замечая, что $x_{i-1}a_ix_{i-1}a_i=x_i$, получаем

$$\begin{aligned} R_0 \cap R_1 \cap \ldots \cap R_i = {} & \\ = {} & y_2'a_ix_{i-1}a_i[(A-A_i)^+ x_i]^* (A-A_i)^+ y_1' \\ + {} & a_ix_{i-1}a_i[(A-A_i)^+ x_i]^* (A-A_i)^+ x_{i-1} \\ + {} & y_2'a_i[(A-A_i)^+ x_i]^* (A-A_i)^+ x_{i-1}a_iy_1' \\ + {} & a_i[(A-A_i)^+ x_i]^* (A-A_i)^+ x_{i-1}a_ix_{i-1} \\ + {} & [(A-A_i)^+ x_i]^* (A-A_i)^+ x_{i-1}a_ix_{i-1}a_i (A-A_i)^*. \end{aligned} \quad (11.2)$$

Наконец, (11.2) можно переписать в виде

$$y_2[(A-A_i)^+ x_i]^* (A-A_i)^+ y_1 + [(A-A_i)^+ x_i]^* (A-A_i)^+ x_i (A-A_i)^*,$$

где $y_1y_2=x_i$. Отсюда вытекает утверждение индукции. Полагая $i=k-1$ и пересекая с $A_{k-1}^* a_k A_{k-1}^*$, заключаем, что R представляет ЦИКЛ$(x_{k-1}a_k)$.

Осталось показать, что длина расширенного регулярного выражения R не превосходит ck^2. Это непосредственно следует из существования такой постоянной c_1, что длина каждого регулярного выражения R_i не превосходит c_1k. Иными словами, A_i и $A-A_i$ обозначают расширенные регулярные выражения длины $O(k)$, и потому каждое слагаемое суммы, определяющей R_i, обозначает расширенное регулярное выражение длины $O(k)$. Следовательно, длина полурасширенного регулярного выражения R ограничена сверху числом ck^2 для некоторой новой постоянной c. □

Конструкция, аналогичная конструкции из леммы 11.1, позволяет написать короткое регулярное выражение, представляющее множество всех цепочек из A^*, кроме $x=x_{k-1}$. Это регулярное выражение понадобится нам вместе с полурасширенным регулярным

выражением R, только что построенным для представления множества ЦИКЛ($x\#$).

Лемма 11.2. *Пусть алфавит* $A=\{a_0, a_1, \ldots, a_k\}$ *и цепочки* x_i *те же, что и в лемме* 11.1. *Тогда найдется регулярное выражение длины* $O(k^2)$, *представляющее* $\neg x_{k-1}$.

Доказательство. Цепочка x_{k-1} обладает такими свойствами:

(1) она непуста и начинается с a_0 (символы a_0 входят в нее парами и за каждым вхождением такой пары идет a_1),
(2) для $1 \leqslant i \leqslant k-2$ символы a_i также входят парами (в каждой паре два символа a_i отделены друг от друга цепочкой из $a_0 A_{i-1}^*$ и за вторым a_i этой пары идет a_{i+1}, т. е. за символом a_i, слева от которого стоят исключительно символы из A_{i-1} или непосредственно слева от которого стоит цепочка из A_{i-1}^*, имеющая перед собой символ из $A-A_i$, идет символ a_0),
(3) символ a_{k-1} входит в точности два раза (за первым вхождением идет a_0, а второе является самым правым символом).

Еще важнее то, что x_{k-1} — единственная цепочка, обладающая этими свойствами. Можно доказать индукцией по j, что если $b_1 b_2 \ldots b_j$ — префикс цепочки, обладающей этими свойствами, то $b_1 b_2 \ldots b_j$ — префикс цепочки x_{k-1}. Например, в силу (1) первым символом является a_0. Также в силу (1) изолированный символ a_0 не может быть последним в этой цепочке и за ним может идти только a_0, следовательно, цепочка начинается с $a_0 a_0$. Далее, опять по свойству (1) за $a_0 a_0$ должен идти символ a_1. В силу свойства (2) при $i=1$ за цепочкой $a_0 a_0 a_1$ должен идти символ a_0 и т. д. Формальную индукцию оставляем в качестве упражнения. Поскольку x_{k-1} на самом деле обладает этими свойствами, эта цепочка и есть единственная, обладающая ими.

Легко написать регулярное выражение, представляющее все цепочки, не обладающие хотя бы одним из этих свойств. Цепочки, не обладающие свойством (1), представляются выражением [1])

$$S_1 = \varepsilon + (A-a_0)\,A^* + A^* a_0 a_0 [(A-a_1)\,A^* + \varepsilon] + \\ + [\varepsilon + A^*(A-a_0)]\,a_0\,[(A-a_0)\,A^* + \varepsilon].$$

Первое слагаемое в S_1 обозначает пустую цепочку, второе — те цепочки, которые не начинаются с a_0, третье — те цепочки, в которых после пары символов a_0 не идет a_1, и последнее — те цепочки, которые содержат изолированный символ a_0. Цепочки, не обладаю-

[1]) Через $A - a_i$ обозначена сумма $\sum_{\substack{j=0 \\ j \neq i}}^{k} a_j$.

щие свойством (2), представляются выражением

$$S_2 = \sum_{i=1}^{k-2} [A^* a_i A_{i-1}^* a_i [(A - a_{i+1}) A^* + \varepsilon] + \\ + [\varepsilon + A^* (A - A_i)] A_{i-1}^* a_i [(A - a_0) A^* + \varepsilon]].$$

Наконец, цепочки, не обладающие свойством (3), представляются выражением

$$S_3 = A^* a_{k-1} A^* a_{k-1} A^+ + (A - a_{k-1})^* a_{k-1} (A - a_{k-1})^* + \\ + (A - a_{k-1})^* + (A - a_{k-1})^* a_{k-1} (A - a_0) A^*.$$

В S_3 первое слагаемое обозначает все цепочки, имеющие более двух вхождений символа a_{k-1}, а также цепочки с ровно двумя вхождениями символа a_{k-1}, второе из которых не приходится на правый конец цепочки. Таким образом, $S_1 \cup S_2 \cup S_3$ представляет $\neg x_{k-1}$. Легко видеть, что длина выражения $S_1 \cup S_2 \cup S_3$ есть $O(k^2)$. □

Прежде чем переходить к доказательству отсутствия алгоритма с полиномиальной памятью или с полиномиальным временем работы, который решал бы задачу пустоты дополнения для полурасширенных регулярных выражений, введем два определения.

Определение. *Гомоморфизмом* из Σ_1^* в Σ_2^* называется такая функция h, что $h(xy) = h(x)\, h(y)$ для любых цепочек x и y. Отсюда непосредственно следует, что $h(\varepsilon) = \varepsilon$ и $h(a_1 a_2 \ldots a_n) = h(a_1) h(a_2) \ldots h(a_n)$. Таким образом, гомоморфизм h однозначно определяется значением $h(a)$ для каждого $a \in \Sigma_1$.

Говорят, что гомоморфизм h *сохраняет длину*, если $h(a)$ — однобуквенное слово из Σ_2 для каждого $a \in \Sigma_1$. Гомоморфизм, сохраняющий длину, просто переименовывает символы, возможно отождествляя несколько символов путем переименования их в один символ. Если $w \in \Sigma_2^*$, то $h^{-1}(w) = \{x \mid h(x) = w\}$. Если $L \subseteq \Sigma_2^*$, то $h^{-1}(L) = \{x \mid h(x) \in L\}$.

Пример 11.2. Пусть $\Sigma_1 = \{a, b, c\}$ и $\Sigma_2 = \{0, 1\}$. Определим гомоморфизм h равенствами $h(a) = 010$, $h(b) = 1$ и $h(c) = \varepsilon$. Тогда $h(abc) = 0101$. Заметим, что h не сохраняет длину. Пусть L — язык, представляемый расширенным регулярным выражением $1 + \neg 1^*$, или, в эквивалентной форме, обычным регулярным выражением $1 + (0 + 1)^* 0 (0 + 1)^*$. Тогда $h^{-1}(L)$ представляется расширенным регулярным выражением $c^* bc^* + \neg(b + c)^*$ или обычным регулярным выражением $c^* bc^* + (a + b + c)^* a (a + b + c)^*$. □

Лемма 11.3. *Пусть h — сохраняющий длину гомоморфизм из Σ_1^* в Σ_2^*, а R_2 — расширенное регулярное выражение, представляющее множество $S \subseteq \Sigma_2^*$. Можно построить такое расширенное регулярное выражение R_1, представляющее $h^{-1}(S)$, что его длина будет*

Верхняя дорожка	#	C_0	#	C_1	#	...	#	C_f	#
Нижняя дорожка	#	x	#	x	#	...	#	x	#

Рис. 11.1. Последовательность МО с измерителем.

ограничена длиной выражения R_2, умноженной на постоянную (зависящую только от h), и оно будет содержать знак дополнения только тогда, когда его содержит R_2.

Доказательство. Заменим каждое вхождение в R_2 символа из Σ_2 регулярным выражением, представляющим множество символов, отображаемых в этот символ гомоморфизмом h. Например, если $\{a_1, a_2, \ldots, a_r\}$ — множество символов, отображаемых в a, заменим a на $(a_1+a_2+\ldots+a_r)$. Пусть R_1 — расширенное регулярное выражение, получающееся в результате такой замены. Доказать, что R_1 представляет $h^{-1}(S)$, можно индукцией по числу вхождений в R_2 знаков $+$, $\cdot$, *, $\cap$ и $\neg$. □

Теперь мы можем представить последовательности МО машин Тьюринга во многом так же, как делали это в лемме 10.2. Иными словами, для данной машины Тьюринга $M=(Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ представляем ее мгновенное описание последовательностью символов из T и еще одним символом вида $[qX]$, где $q \in Q$ и $X \in T$, который указывает состояние и положение головки. При необходимости можно дополнить МО пустыми символами, чтобы все МО одного и того же вычисления были одинаковой длины.

Чтобы облегчить сравнение символов одного МО с "соответствующими" символами в следующем МО, приложим к самим МО измеритель, длина которого равна длине этих МО. Формально мы используем "двухдорожечную" цепочку символов, в которой верхняя дорожка содержит последовательность МО, а нижняя — повторяющийся измеритель. Здесь "двухдорожечными" символами служат пары $[a, b]$, где a — символ на верхней дорожке, а b — на нижней. Это изображено на рис. 11.1, где использован измеритель ЦИКЛ($x\#$) (x может быть любой цепочкой, не содержащей $\#$).

Определение. Для данной МТ M положим $\Delta_1=T \cup (Q\times T) \cup \{\#\}$, а Δ_2 пусть будет множеством символов, входящих в $x\#$, т. е. Δ_1— множество символов, которые могут появляться на верхней дорожке, а Δ_2 — на нижней. "Двухдорожечным" алфавитом будет $\Delta_1\times\Delta_2$. *Правильным вычислением машины M с измерителем* ЦИКЛ($x\#$) называют цепочку из множества $(\Delta_1\times\Delta_2)^*$, имеющую вид, как на рис. 11.1, где $C_i \vdash C_{i+1}$ за один шаг машины M для $0 \leqslant i < f$, C_0— начальное МО и состоянием последнего МО C_f является q_f. В МО дописаны пустые символы, чтобы все МО имели длину $|x|$.

Лемма 11.4. *Пусть R_1 — расширенное регулярное выражение для множества* ЦИКЛ($x\#$) *и R_1' — регулярное выражение над алфавитом цепочки $x\#$, представляющее все цепочки, кроме x. Можно построить такое расширенное регулярное выражение R_2, представляющее все неправильные вычисления машины M с измерителем* ЦИКЛ($x\#$), *что его длина будет линейной по $|R_1|+|R_1'|$, причем коэффициент пропорциональности будет зависеть только от M, и оно будет содержать знак дополнения только тогда, когда его содержит R_1 или R_1'.*

Доказательство. Цепочка не является правильным вычислением с измерителем ЦИКЛ ($x\#$) тогда и только тогда, когда либо

1) нижняя дорожка не содержится в $\#(x\#)^*$, либо
2) нижняя дорожка содержится в $\#(x\#)^*$, но верхняя дорожка не является правильным вычислением.

Если Δ_1 и Δ_2 — алфавиты верхней и нижней дорожек соответственно, то обозначим через h_1 и h_2 гомоморфизмы, отображающие $(\Delta_1\times\Delta_2)^*$ в Δ_1^* и Δ_2^* соответственно, так что $h_1([a,\ b])=a$ и $h_2([a,\ b])=b$. Тогда

$$h_2^{-1}[\Delta_2^*\#(R_1'\cap(\Delta_2-\#)^*)\#\Delta_2^*+(\Delta_2-\#)\Delta_2^*+\Delta_2^*(\Delta_2-\#)]+\varepsilon$$

представляет те и только те цепочки, у которых нижняя дорожка не принадлежит $\#(x\#)^*$. Первое слагаемое аргумента в h_2^{-1} представляет цепочки, у которых между двумя знаками $\#$ стоит нечто, отличное от x, а остальные представляют цепочки, которые не начинаются или не кончаются символом $\#$. По лемме 11.3 найдется расширенное регулярное выражение с длиной, пропорциональной $|R_1'|$, представляющее это множество.

Данная цепочка удовлетворяет условию 2, когда два символа, отделенные друг от друга символами в количестве, равном длине измерителя, не соответствуют шагу машины M либо когда происходит одно или более из следующих нарушений структуры:

1) цепочка на верхней дорожке не начинается или не кончается символом $\#$,
2) в первом МО или совсем нет состояния, или есть два или более состояний,
3) в первом МО начальное состояние не является компонентой первого символа,
4) допускающее состояние не появляется в качестве компоненты никакого символа,
5) первое МО не имеет корректной длины, т. е. первые два символа $\#$ на верхней дорожке стоят не в тех же клетках, что и первые два $\#$ на нижней дорожке.

Покажем, как написать расширенное регулярное выражение для последовательностей в $(\Delta_1 \times \Delta_2)^*$, по какой-то причине не отражающих правильные шаги машины M. Как и в лемме 10.2, замечаем, что в правильном вычислении три последовательных символа $c_1c_2c_3$ на верхней дорожке однозначно определяют символ, стоящий на $|x\#|$ позиций правее c_2. Пусть, как и раньше, этим символом будет $f(c_1c_2c_3)$. Положим

$$R_{c_1c_2c_3} = (\Delta_1 \times \Delta_2)^* [h_1^{-1}(c_1c_2c_3\Delta_1^*) \cap h_2^{-1}(R_1)][h_1^{-1}(\Delta_1(\Delta_1 - f(c_1c_2c_3))\Delta_1)](\Delta_1 \times \Delta_2)^* \quad (11.3)$$

и

$$R = \sum_{c_1c_2c_3} R_{c_1c_2c_3}.$$

Заметим, что $h_1^{-1}(c_1c_2c_3\Delta_1^*)$ представляет все цепочки из $(\Delta_1 \times \Delta_2)^*$, у которых верхняя дорожка начинается с $c_1c_2c_3$, а $h_2^{-1}(R_1)$ — все цепочки длины $|x\#|$, у которых нижняя дорожка корректна. Поэтому их пересечение содержит все цепочки длины $|x\#|$, у которых на нижней дорожке стоит цепочка из ЦИКЛ($x\#$), а верхняя начинается с $c_1c_2c_3$. Теперь должно быть понятно, что R включает все цепочки, удовлетворяющие условию 2, поскольку в них есть шаг, незаконный для M. Кроме них, туда будут включены некоторые цепочки, у которых на нижней дорожке не стоит $\#(x\#)^*$; они удовлетворяют и условию 1, и их присутствие или отсутствие в R несущественно. Далее, длина выражения R ограничена длиной выражения R_1, умноженной на постоянную (зависящую от M). Чтобы убедиться в истинности этого утверждения, достаточно заметить, что h_2^{-1} увеличивает длину регулярных выражений в постоянное число раз, зависящее от M. Аналогично, h_1^{-1} увеличивает длину выражений не более чем в $3||\Delta_2||$ раз, ибо если $h_1(b)=a$ ровно для k значений b, то $h_1^{-1}(a)=(b_1+b_2+\ldots+b_k)$ имеет длину $2k+1$. Так как $1 \leqslant k \leqslant ||\Delta_2||$, то $|h_1^{-1}(a)| \leqslant 3||\Delta_2||$. Более того, должно быть ясно, что $||\Delta_2|| \leqslant |R_1|$, поскольку каждый символ из Δ_2 появляется в R_1. Следовательно, $h_1^{-1}(c_1c_2c_3\Delta_1^*)$ и $h_1^{-1}(\Delta_1(\Delta_1 - f(c_1c_2c_3))\Delta_1)$ из (11.3) имеют длину $O(|R_1|)$. Наконец, $h_2^{-1}(R_1)$ и $(\Delta_1 \times \Delta_2)^*$, очевидно, имеют длину $O(|R_1|)$, где постоянная зависит только от M. Поэтому (11.3) и, значит, R имеют длину $O(|R_1|+|R_1'|)$.

Регулярные выражения, отражающие упомянутые выше нарушения структуры, строятся весьма просто, и это построение мы оставляем читателю. Изложенным способом можно построить такое расширенное регулярное выражение R_2, что его длина будет линейна по $|R_1|+|R_1'|$ и оно будет содержать знак дополнения только тогда, когда его содержит R_1 или R_1'. □

Теорема 11.2. *Любой алгоритм, выясняющий, представляет ли данное полурасширенное регулярное выражение* [1]) *все цепочки в своем*

[1]) Предполагается, что кодировка аналогична использованной в лемме 10.3.

алфавите, имеет емкостную (и, следовательно, временную) сложность, которая для бесконечно многих n не меньше $c'c^{\sqrt{n/\log n}}$, где $c'>0$ и $c>1$ — постоянные.

Доказательство. Пусть L — произвольный язык с емкостной сложностью 2^n, но не $2^n/n$ [1]). (Из теоремы 11.1 мы знаем, что такой язык существует.) Пусть M — ДМТ, допускающая L.

Предположим, что у нас есть ДМТ M_0 с емкостной сложностью $f(n)$, выясняющая, пусто ли дополнение множества, представленного данным полурасширенным регулярным выражением. Тогда можно применить M_0 для распознавания языка L следующим образом. Пусть $w=a_1a_2\ldots a_n$ — входная цепочка длины n.

1. Построим полурасширенное регулярное выражение R_1 для измерителя с длиной 2^n+1 или более. По лемме 11.1 (для $k=n$) R_1 существует и имеет длину $O(n^2)$. Более того, чтобы найти R_1, достаточно $O(n^2)$ памяти. Аналогично построим R_1', представляющее $\neg x$, где x таково, что $R_1=$ЦИКЛ$(x\#)$. В силу леммы 11.2 выражение R_1' имеет длину $O(n^2)$ и, как легко видеть, его можно построить, заняв $O(n^2)$ клеток памяти.
2. Построим полурасширенное регулярное выражение R_2, представляющее все неправильные вычисления машины M с измерителем R_1. В силу леммы 11.4 можно построить R_2 так, чтобы его длина была не больше c_1n^2, где c_1 — постоянная, зависящая только от M.
3. Построим регулярное выражение R_3, представляющее все цепочки из $(\Delta_1\times\Delta_2)^*$, которые не начинаются на верхней дорожке с $\#\,[q_0a_1]a_2\ldots a_n\mathrm{b}\ldots\mathrm{b}\#$, где q_0 — начальное состояние машины M. Очевидно, что существует такое выражение R_3 длины $O(n)$. Таким образом, $|R_2+R_3|\leqslant c_2n^2$ для некоторой постоянной c_2.
4. Применим M_0 к множеству, представленному R_2+R_3, чтобы выяснить, пусто ли его дополнение. Если оно непусто, то существует правильное вычисление машины M с входом w, так что w принадлежит L. В противном случае w не принадлежит L.

Можно построить ДМТ M' с емкостной сложностью $f(c_2n^2\log n)$, реализующую этот алгоритм распознавания L. Множитель $\log n$ в аргументе функции f появился из-за того, что регулярное выражение R_2+R_3 над n-символьным алфавитом нужно закодировать цепочкой в фиксированном алфавите. Так как мы предположили, что L имеет емкостную сложность 2^n, но не $2^n/n$, то неравенство

[1]) Выбор функции 2^n несуществен. Можно было бы вместо 2^n взять любую экспоненциальную функцию $f(n)$, а вместо $2^n/n$ — любую функцию, которая растет чуть медленнее, чем $f(n)$, при условии, что эти функции конструируемы по памяти.

$f(c_2n^2 \log n) > 2^n/n$ должно выполняться по крайней мере для некоторых n. Но если бы $f(c_2n^2 \log n) > 2^n/n$ выполнялось лишь для конечного числа n, то можно было бы построить модифицированный вариант описанного выше алгоритма распознавания, который сначала проверял бы по конечной таблице, совпадает ли $|w|$ с одним из тех n, для которых $f(c_2n^2 \log n) > 2^n/n$, и если да, то узнавал бы из той же таблицы, принадлежит ли w языку L. Поэтому значение $f(c_2n^2 \log n)$ должно превосходить $2^n/n$ для бесконечно многих n, так что $f(m) \geqslant 2^{c_3\sqrt{m/\log m}}/\sqrt{m/\log m} \geqslant c(c')^{\sqrt{m/\log m}}$ для бесконечно многих m и некоторых постоянных $c_3 > 0$, $c > 0$ и $c' > 1$. □

Следствие. *Задача эквивалентности полурасширенных регулярных выражений требует* $c'c^{\sqrt{n/\log n}}$ *памяти и времени при некоторых постоянных* $c' > 0$, $c > 1$ *и бесконечно многих* n.

Доказательство. Легко показать, что задача пустоты дополнения полиномиально сводима к задаче эквивалентности, поскольку регулярное выражение, представляющее все цепочки, легко пишется и имеет небольшую длину. □

11.4. НЕЭЛЕМЕНТАРНАЯ ЗАДАЧА

Исследуем весь класс расширенных регулярных выражений. Поскольку теперь у нас есть операция дополнения, достаточно рассматривать задачу пустоты, а именно: пусто ли множество, представленное данным регулярным выражением? Мы увидим, что, располагая операцией дополнения, можно представлять регулярные множества еще более короткими выражениями и задача пустоты для расширенных регулярных выражений значительно труднее задачи пустоты дополнения для полурасширенных регулярных выражений.

Определим функцию $g(m, n)$ равенствами

1) $g(0, n) = n$,
2) $g(m, n) = 2^{g(m-1, n)}$ для $m > 0$.

Тогда $g(1, n) = 2^n$, $g(2, n) = 2^{2^n}$ и

$$g(m, n) = 2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}},$$

где справа стоит башня из m двоек и последняя возводится в степень n. Функция $f(n)$ называется *элементарной*, если для некоторого m_0 она ограничена сверху функцией $g(m_0, n)$ для всех, кроме конечного числа, значений n.

С помощью техники разд. 11.3 можно доказать, что задача пустоты для всего класса расширенных регулярных выражений не может иметь емкостную сложность $S(n)$ ни для какой элементарной

Верхняя дорожка	C_0	#	C_1	#	...	#	C_f	\$
Нижняя дорожка	x	#	x	#	...	#	x	\$

Рис. 11.2. Новая форма правильных вычислений с измерителем.

функции $S(n)$. Прежде чем приступить к доказательству, изменим немного определение правильного вычисления машины Тьюринга $M=(Q, T, I, \delta, b, q_0, q_f)$ с измерителем ЦИКЛ($x\#$). Удалим первый маркер $\#$ и заменим последний маркер новым символом \$ (рис. 11.2). Обозначим через C_i МО (дополненные, если надо, пустыми символами, чтобы их длина стала равной длине цепочки x); как и раньше, $C_i \vdash C_{i+1}$ за один шаг машины M для $0 \leqslant i < f$, C_0 — начальное МО и C_f содержит состояние q_f.

Будем использовать следующие обозначения (предполагаем, что $x \in \Sigma^*$):

1) $\Delta_1 = T \cup (Q \times T) \cup \{\#, \$\}$ — алфавит верхней дорожки,
2) $\Delta_2 = \Sigma \cup \{\#, \$\}$ — алфавит нижней дорожки,
3) h_1: $\Delta_1 \times \Delta_2 \to \Delta_1$, причем $h_1([a, b]) = a$,
4) h_2: $\Delta_1 \times \Delta_2 \to \Delta_2$, причем $h_2([a, b]) = b$.

Лемма 11.5. *Пусть R_1 — расширенное регулярное выражение для множества* ЦИКЛ($x\#$). *Можно построить такое расширенное регулярное выражение R_2, представляющее множество циклических перестановок правильных вычислений машины Тьюринга M с измерителем* ЦИКЛ($x\#$), *что $|R_2|$ есть $O(|R_1|)$, где постоянная зависит только от M.*

Доказательство. Цепочка является циклической перестановкой правильного вычисления с измерителем ЦИКЛ($x\#$) тогда и только тогда, когда

1) нижняя дорожка — циклическая перестановка цепочки вида $(x\#)^* x\$$,
2) верхняя дорожка — циклическая перестановка правильного вычисления машины M и
3) верхняя и нижняя дорожки циклически переставлены одинаково.

Пусть R_1' — это выражение R_1, в котором вместо $\#$ стоит символ \$. Цепочки, удовлетворяющие условию 1, можно представить выражением $h_2^{-1}(U_1 \cap U_2 \cap U_3)$, где

$$U_1 = \Sigma^* (\# + \$) [(R_1 + R_1') \cap (\Sigma^* \# + \Sigma^* \$)]^* \Sigma^*,$$
$$U_2 = (\Sigma + \#)^* \$ (\Sigma + \#)^*,$$
$$U_3 = (R_1 + R_1')^*.$$

Подвыражение $[(R_1+R_1')\cap(\Sigma^*\#+\Sigma^*\$)]$ в U_1 представляет две цепочки $x\#$ и $x\$$. Поэтому U_1 представляет $\Sigma^*(\#+\$)(x\#+x\$)^*\Sigma^*$. Выражение U_2 содержит цепочки с ровно одним вхождением \$. Поэтому всякая цепочка из $U_1\cap U_2$ имеет вид

$$y_2\#x\#x\#\ldots x\#x\$x\#\ldots x\#y_1,$$

где y_1 и y_2 — некоторые цепочки из Σ^*. Выражение U_3 приводит к тому, что $|y_2|+|y_1|=|x|$, откуда легко следует, что $y_1y_2=x$ для цепочек из $U_1\cap U_2\cap U_3$. Таким образом, $S_1=U_1\cap U_2\cap U_3$ представляет нижнюю дорожку всех цепочек, удовлетворяющих условию 1.

Что касается условия 2, то в лемме 11.4 мы видели, как написать полурасширенное регулярное выражение для неправильных вычислений машины M с измерителем ЦИКЛ($x\#$). Эта техника легко переносится на форму, показанную на рис. 11.2, и мы предлагаем читателю самому построить выражение E, представляющее циклические перестановки неправильных вычислений машины M с корректным измерителем на нижней дорожке. Применение операции дополнения к E дает расширенное регулярное выражение S_2. Это единственное место, где применяется дополнение. Должно быть ясно, что все цепочки, представленные выражением $S_1\cap S_2$, удовлетворяют обоим условиям 1 и 2.

Выражение для условия 3 в предположении, что условия 1 и 2 выполнены, строится без особого труда. Надо лишь проверить, что один символ имеет \$ на обеих дорожках. Пересекая это выражение с $S_1\cap S_2$, получаем требуемое. □

Теперь покажем, как с помощью леммы 11.5 выражать все более и более длинные измерители, не слишком увеличивая длины задающих их выражений.

Интуитивно мы строим регулярное выражение для какого-то измерителя с небольшой длиной, скажем с длиной n. Затем используем его, чтобы построить новое регулярное выражение для всех циклических перестановок правильных вычислений с этим измерителем для некоторой машины Тьюринга, допускающей в точности одну цепочку длины n. Машина Тьюринга специально выбирается так, чтобы на входе длины n она делала по меньшей мере 2^{n+2} шагов, поэтому множество циклических перестановок ее правильных вычислений само будет измерителем с длиной 2^n или более. Повторяя этот процесс с новым измерителем, получаем измерители все с бóльшими и бóльшими длинами, но не меньшими 2^n, 2^{2^n} и т. д.

Пусть M_0 — конкретная одноленточная машина Тьюринга, которая ведет себя так:

1) проверяет, что ее входная лента начинается с последовательности символов a, для чего просматривает эту ленту до первого пустого символа,
2) если входная лента содержит цепочку из m символов a, за которыми стоит пустой символ, считает в двоичной системе

от 0 до $2^{m+1}-1$ на части своей ленты, которая была занята символами a и следующим за ними пустым символом,

3) досчитав до $2^{m+1}-1$, останавливается и допускает свой вход.

Итак, M_0 обладает следующими свойствами. Для каждого n она допускает в точности одну цепочку длины n, а именно a^n. Она делает это, выполняя единственное правильное вычисление, состоящее не менее чем из 2^{n+2} шагов, поскольку половина ее шагов добавляет биты, а другая половина осуществляет переносы. Наконец, M_0 просматривает только $n+1$ клеток ленты, когда ей подан вход длины n.

Исследуем теперь правильные вычисления машины M_0 с измерителем ЦИКЛ($x\#$), имеющие вид, показанный на рис. 11.2. Если $|x|=n+1$, то существует единственное правильное вычисление машины M_0 с измерителем ЦИКЛ($x\#$), у которого верхняя дорожка начинается с $[q_0a]a\ldots a\,\mathrm{b}\#$. По лемме 11.5 для всех его циклических перестановок можно построить регулярное выражение R_2. Оно будет представлять множество, состоящее из циклических перестановок фиксированной цепочки $w\in(\Delta_1\times\Delta_2)^*$. Цепочка $h_1(w)$ будет правильным вычислением машины M_0 на входе a^n, где, напомним, $|x|=n+1$. Так как M_0 делает на входе a^n не менее 2^{n+2} шагов, R_2 можно взять в качестве измерителя с длиной не менее 2^{n+2}.

Итак, пусть дан измеритель ЦИКЛ($x\#$), представленный выражением R_1, где $x\in\Sigma^*$. Тогда по R_1 и описанной выше ДМТ M_0 можно построить измеритель с длиной, не меньшей $2^{|x\#|}$. По лемме 11.5 для него найдется регулярное выражение R_2 длины не более $c_1|R_1|$, где постоянная c_1 зависит только от M_0.

Лемма 11.6. *Для всех $i\geqslant 1$ и $m\geqslant 2$ существует измеритель с длиной, не меньшей $g(i, m)$* [1]*, который можно представить расширенным регулярным выражением длины не более $c(c_1)^{i-1}m^2$, где c_1 — постоянная, зависящая от M_0 и введенная в предыдущем абзаце, а c — постоянная из леммы* 11.1.

Доказательство. Докажем индукцией по i, что найдется расширенное регулярное выражение R_1, представляющее ЦИКЛ($x\#$) для некоторого x, где $|x\#|\geqslant g(i, m)$ и $|R_1|\leqslant c(c_1)^{i-1}m^2$. Базис ($i=1$) следует из леммы 11.1 при $k=m$, поскольку можно построить расширенное регулярное выражение длины cm^2, представляющее ЦИКЛ($x\#$).

Для шага индукции допустим, что построено расширенное регулярное выражение R_1 длины $|R_1|\leqslant c(c_1)^{i-2}m^2$, представляющее ЦИКЛ($x\#$), где $|x\#|\geqslant g$ ($i-1$, m). Тогда на основе проведенного выше анализа, касающегося ДМТ M_0, заключаем, что можно по-

[1]) Функция g определена в самом начале раздела.

строить регулярное выражение R длины не более $c_1|R_1|=c(c_1)^{i-1}m^2$, представляющее ЦИКЛ($y\$$), где $|y\$|\geqslant 2^{|x\#|}\geqslant 2^{g(i-1,\,m)}=g(i,\,m)$. □

Теорема 11.3. *Пусть $S(n)$ — любая элементарная функция. Тогда не существует* ДМТ *с емкостной (и, следовательно, временной) сложностью, ограниченной сверху функцией $S(n)$, которая могла бы установить, пусто ли множество, представленное произвольным расширенным регулярным выражением.*

Доказательство. Допустим противное, т. е. что нашлась ДМТ M_1 с памятью, ограниченной функцией $g(k_0,\,n)$, которая выясняет, пусто ли множество, представленное расширенным регулярным выражением. По теореме 11.1 существует язык L, допускаемый некоторой ДМТ M с памятью, ограниченной функцией $g(k_0+1,\,n)$, но не допускаемый никакой ДМТ с памятью, не большей $g(k_0+1,\,n)/n$. Предположив, что M_1 существует, можно построить ДМТ M_2, распознающую язык L следующим образом.

1. По данной входной цепочке $w=a_1a_2\ldots a_n$ длины n машина M_2 строит расширенное регулярное выражение R_1 длины d_1n^2 (где d_1 — постоянная, не зависящая от n), представляющее измеритель с длиной, не меньшей $g(k_0+1,\,n)$. Для построения R_1 используется лемма 11.6.
2. Далее M_2 строит расширенное регулярное выражение R_2, представляющее правильные вычисления с измерителем R_1 для ДМТ M, которая допускает L и требует для этого не более $g(k_0+1,\,n)$ клеток памяти. По лемме 11.5 можно добиться, чтобы было $|R_2|\leqslant d_2|R_1|$ для некоторой постоянной d_2.
3. Затем M_2 строит расширенное регулярное выражение R_3, представляющее правильные вычисления машины M с измерителем R_1 и начальным МО $C_0=[q_0a_1]a_2\ldots a_n\mathrm{bb}\ldots$, где q_0 — начальное состояние машины M, а именно

$$R_3=R_2\cap h^{-1}([q_0a_1]a_2\ldots a_n\mathrm{b}^*\#)\,(\Delta_1\times\Delta_2)^*,$$

где h_1 — гомоморфизм из леммы 11.5 и $\Delta_1\times\Delta_2$ — алфавит выражения R_2. По лемме 11.3 $|R_3|\leqslant|R_2|+d_3n$ для некоторой постоянной $d_3>0$. Поэтому

$$|R_3|\leqslant d_1d_2n^2+d_3n. \tag{11.4}$$

4. Наконец, M_2 кодирует R_3 в фиксированном алфавите, как в теореме 11.2, и использует M_1 для проверки пустоты множества, представленного расширенным регулярным выражением R_3. Если оно пусто, что M_2 отвергает вход w, а если нет, то допускает его. Таким образом, M_2 допускает L.

Теперь нетрудно видеть, что наибольшей памяти требует шаг 4. На этом шаге машине M_1 нужна память $g(k_0,\,|R_3|\log|R_3|)$, чтобы выяснить, пусто ли множество, представленное выражением R_3.

Следовательно, и машине M_2 нужно столько же памяти. Учитывая границу (11.4) для $|R_3|$, заключаем, что M_2 имеет емкостную сложность $S(n)=d_4g(k_0, d_5n^2\log n)$ для некоторых постоянных d_4 и d_5, поскольку для всех, кроме конечного числа, значений n первое слагаемое в правой части (11.4), а именно $d_1d_2n^2$, больше второго, т. е. d_3n. Однако, рассуждая, как в приведенном доказательстве теоремы 11.2, можно показать, что емкостная сложность машины M_2 должна быть больше $g(k_0+1, n)/n$ для бесконечно многих n. Таким образом, если существует M_2, то

$$g(k_0+1, n)/n < d_4g(k_0, d_5n^2\log n) \qquad (11.5)$$

для бесконечно многих n. Но независимо от выбора d_4 и d_5 лишь для конечного числа значений n справедливо (11.5) (это легко проверить). Итак, машины M_2 не существует, и, значит, не существует M_1. Поэтому проблема пустоты для расширенных регулярных выражений неразрешима никакой машиной Тьюринга с емкостной сложностью, ограниченной элементарной функцией. □

УПРАЖНЕНИЯ

11.1. Функцию $T(n)$ называют *конструируемой по времени*, если некоторая ДМТ M при данном входе длины n делает ровно $T(n)$ шагов до своей остановки. Покажите, что функции

(а) n^2,
(б) 2^n,
(в) $n!$

конструируемы по времени.

***11.2.** Покажите, что всякая функция, конструируемая по времени, конструируема по памяти, и если функция $S(n)$ конструируема по памяти, то $c^{S(n)}$ конструируема по времени для некоторого целого числа c.

***11.3.** Покажите, что если язык L допускается за время $T(n)$ какой-нибудь k-ленточной ДМТ (НМТ), то он допускается некоторой одноленточной ДМТ (НМТ) с временной сложностью $O(T^2(n))$.

***11.4.** (*Иерархия по времени для* ДМТ) Покажите, что если функции $T_1(n)$ и $T_2(n)$ конструируемы по времени и

$$\inf_{n\to\infty} \frac{T_1^2(n)}{T_2(n)} = 0,$$

то некоторый язык допускается ДМТ за время $T_2(n)$, но не $T_1(n)$. *Указание*: В силу упр. 11.3 достаточно диагонализировать по одноленточным ДМТ с временной сложностью не выше $T_1(n)$. Саму диагонализацию можно выполнить с помощью многоленточной ДМТ.

В следующих двух упражнениях сформулированы слабые варианты результатов об иерархиях для недетерминированных машин Тьюринга.

**11.5. Покажите, что для каждого целого числа $k \geqslant 1$ существует язык, допускаемый некоторой НМТ с емкостной сложностью n^{k+1}, но не допускаемый никакой НМТ с емкостной сложностью n^k.

**11.6. Докажите тот же результат, что и в упр. 11.5, для временной сложности.

В следующих двух упражнениях иерархия по времени для ДМТ уплотняется.

**11.7. Покажите, что всякий язык L, допускаемый некоторой k-ленточной ДМТ с временной сложностью $T(n)$, допускается двухленточной ДМТ с временной сложностью $O(T(n) \log T(n))$.

11.8. С помощью результата упр. 11.7 докажите, что если функции $T_1(n)$ и $T_2(n)$ конструируемы по времени и

$$\inf_{n \to \infty} \frac{T_1(n) \log T_1(n)}{T_2(n)} = 0,$$

то некоторый язык допускается ДМТ за время $T_2(n)$, но не $T_1(n)$.

*11.9. Покажите, что если L допускается ДМТ (НМТ) M с емкостной сложностью $S(n)$ и временной сложностью $T(n)$, а $c > 0$ — произвольная постоянная, то L допускается некоторой ДМТ (НМТ) M' с емкостной сложностью $\mathrm{MAX}(cS(n), n+1)$ и временной сложностью $\mathrm{MAX}(cT(n), 2n)$. *Указание*: M' должна начинать свою работу с сжимания блоков клеток машины M так, чтобы каждый такой блок помещался в одну клетку машины M'.

11.10. Покажите, что если L допускается НМТ с временной сложностью $T(n)$, то найдется такая постоянная c, что L допускается ДМТ с временной сложностью $c^{T(n)}$.

*11.11. Пусть T_1 и T_2 — такие функции, что

$$\inf_{n \to \infty} \frac{T_1(n)}{T_2(n)} = 0.$$

Покажите, что существует язык, допускаемый РАМ за время $T_2(n)$, но не $T_1(n)$ (время определяется по логарифмическому весовому критерию).

11.12. Закончите доказательство леммы 11.2.

*11.13. По данному расширенному регулярному выражению R_1 для множества ЦИКЛ($x\#$) постройте расширенное регулярное

выражение R_2 для ЦИКЛ($(x\#)^*$) [1]. Длина выражения R_2 должна быть не больше длины выражения R_1, умноженной на постоянную.

**11.14. Постройте недетерминированный алгоритм с емкостной сложностью $O(n)$, выясняющий, пусто ли множество, представленное произвольным полурасширенным регулярным выражением. Почему ваш алгоритм не срабатывает, когда вы пытаетесь узнать, представляет ли данное регулярное выражение все цепочки? Почему он должен не срабатывать?

11.15. Постройте детерминированный алгоритм, решающий за экспоненциальное время задачу пустоты дополнения для полурасширенных регулярных выражений.

11.16. Напишите регулярное выражение для "нарушений структуры" в доказательстве леммы 11.4.

11.17. Элементарна ли функция, определяемая равенствами $F(0)=1$ и $F(n)=2^{F(n-1)}$ для $n\geqslant 1$? (Эта функция введена в разделе 4.7.)

*11.18. Постройте алгоритм, выясняющий, представляет ли произвольное расширенное регулярное выражение пустое множество. Какова временная и емкостная сложности вашего алгоритма?

**11.19. Покажите, что задача пустоты для расширенных регулярных выражений, использующих только знаки операций $+$, $\cdot$ и $\neg$, неэлементарна.

Проблемы для исследования

11.20. Естественная область исследований, подсказываемая материалом этой главы,— поиск практически важных задач, про которые можно доказать, что они трудно разрешимые. Работу в этом направлении проделали Фишер, Рабин [1974], исследовавшие сложность задач с элементарными арифметическими операциями, Кук, Рекхау [1974], изучившие процедуры доказательства теорем [2], Хант [1974], рассмотревший ряд задач из теории языков, и некоторые другие авторы, упомянутые в замечаниях по литературе.

11.21. Интересен также вопрос о том, насколько плотнее могут быть иерархия по времени для ДМТ и иерархии по времени и емкости для НМТ по сравнению с иерархиями, указанными в упр. 11.5, 11.6 и 11.8. Например, существуют ли конструируемые по времени функции $T(n)$, для которых нет языков, распознаваемых за время $T(n)\log T(n)$, но не $T(n)$? Читателю советуем обратиться к работе Сайфераса, Фишера, Мейера [1973], где приводится самая плотная из известных иерархий для НМТ.

[1]) Результатом применения операции ЦИКЛ к множеству цепочек служит объединение ее значений на каждой отдельной цепочке из этого множества.

[2]) Здесь уместно сослаться еще на статью Цейтина [1968].— *Прим. перев.*

Замечания по литературе

Первое обширное исследование иерархий по емкости и времени для машин Тьюринга выполнили Хартманис, Стирнз [1965] и Хартманис, Льюис, Стирнз [1965]. Статья Рабина [1963] была одной из первых работ о временной сложности, и она заслуживает изучения. Улучшения иерархии по времени, данные в упр. 11.7, 11.8, заимствованы у Хенни, Стирнза [1966]. Что касается иерархий для НМТ, то упр. 11.5 (память) взято у Ибарры [1972], а 11.6 (время) — у Кука [1973]. Иерархия для РАМ в упр. 11.11 принадлежит Куку, Рекхау [1973], упр. 11.14 — Хопкрофту, упр. 11.19 — Мейеру, Стокмейеру [1973].

Установление экспоненциальных нижних оценок для «естественных» задач началось с работ Мейера [1972] и Мейера, Стокмейера [1972]. Теорема 11.2 о полурасширенных регулярных выражениях взята у Ханта [1973б], теорема 11.3 о расширенных регулярных выражениях — у Мейера, Стокмейера [1973]. Другие результаты о трудно разрешимых задачах приводят Бук [1972], Хант [1973а, 1974], Стокмейер, Мейер [1973], Мейер [1972], Хант, Розенкранц [1974], Раундз [1973] и Констейбл, Хант, Сахни [1974].

12 НИЖНИЕ ОЦЕНКИ ЧИСЛА АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

При разработке алгоритма для решения данной задачи основной вопрос, ответ на который нам хотелось бы знать,— это "Какова ее подлинная вычислительная сложность?". Зная теоретическую нижнюю границу для эффективности алгоритма, можно лучше оценить имеющиеся алгоритмы и определить, сколь много усилий следует тратить на поиск лучшего решения. Например, если известно, что задача трудно разрешима, можно довольствоваться эвристическими способами нахождения приближенных решений.

К сожалению, выяснить подлинную вычислительную сложность данной задачи обычно очень трудно. Для большинства практических задач, чтобы судить о качестве алгоритма, приходится полагаться на опыт. Однако иногда можно довольно точно оценить снизу число арифметических операций, требуемых для выполнения вычислений. В этой главе мы изложим некоторые важные результаты этого рода. Например, покажем, что умножение $(n \times p)$-матрицы на p-вектор требует np умножений, а вычисление значения полинома n-й степени требует n умножений. Много дополнительных результатов о нижних оценках содержится в упражнениях. Читателю, интересующемуся нижними оценками, советуем рассматривать эти упражнения как неотъемлемую часть данной главы.

12.1. ПОЛЯ

Чтобы получить точные нижние границы, надо определить, какие операции считать основными. Для определенности будем предполагать, что все вычисления ведутся в некотором поле, таком, как поле вещественных чисел, где основные операции — сложение и умножение элементов поля.

Определение. *Полем* называется такая алгебраическая система $(A, +, \cdot, 0, 1)$, что

1) она — кольцо с единицей 1 относительно умножения,
2) умножение коммутативно,

3) каждый элемент a из $A-\{0\}$ обладает обратным a^{-1} относительно умножения, т. е. $aa^{-1}=a^{-1}a=1$[1]).

Пример 12.1. Вещественные числа с обычными операциями сложения и умножения образуют поле. Целые числа образуют кольцо, но не поле, поскольку целые числа, отличные от ± 1, не имеют обратных, являющихся целыми числами. Но для простого числа p целые числа по модулю p образуют (конечное) поле. □

Рассмотрим задачу вычисления произвольного полинома $p(x)=\sum_{i=0}^{n} a_i x^i$ в некоторой точке x. Мы хотим найти алгоритм, который по значениям a_i и x в качестве входов строит соответствующее значение $p(x)$ в качестве выхода. Этот алгоритм должен работать на всех возможных значениях своих входов, принадлежащих некоторому полю. Максимальное число сложений, вычитаний и умножений, выполняемых алгоритмом, где максимум берется по всем допустимым входам, называется *арифметической сложностью* этого алгоритма.

Заметим, что одни полиномы n-й степени вычислить оказывается проще, чем другие. Например, на вычисление x^n+2 тратится порядка $\log n$ операций, тогда как интуитивно мы чувствуем, что на вычисление "произвольного" полинома n-й степени должно расходоваться линейное число операций. Поэтому алгоритм для вычисления значений, пригодный только для конкретного полинома, может работать много быстрее, чем алгоритм, применимый ко всем полиномам. Чтобы сформулировать понятие алгоритма, пригодного для целого класса задач, введем для представления входных переменных формальные переменные.

Определение. *Формальной переменной* над алгебраической системой называется символ, не принадлежащий множеству элементов этой системы. Пусть $F=(A, +, \cdot, 0, 1)$ — поле и $x_1, \ldots, x_n$ — формальные переменные над F. *Расширением* $F[x_1, \ldots, x_n]$ поля F этими переменными $x_1, \ldots, x_n$ называется такое наименьшее коммутативное [2]) кольцо $(B, +, \cdot, 0, 1)$, что B содержит $A \cup \{x_1, \ldots, x_n\}$.

Заметим, что эти формальные переменные не связаны никакими тождествами. Всякий полином с коэффициентами из F и "неизвестными" $x_1, \ldots, x_n$ представляет некоторый элемент из $F[x_1, \ldots, x_n]$. Два полинома обозначают один и тот же элемент из $F[x_1, \ldots, x_n]$, если один из них можно превратить в другой с помощью аксиом коммутативного кольца. Единица в F будет также единицей в $F[x_1, \ldots, x_n]$.

---

[1]) Как обычно, знак $\cdot$ опускается, если это не приводит к недоразумениям.
[2]) Кольцо с коммутативным умножением.

Пример 12.2. Пусть F — поле вещественных чисел. Тогда кольцо $F[x, y]$ включает в себя x, y и все вещественные числа. Так как оно замкнуто относительно сложения, то $x+y$ и $x+4$ принадлежат $F[x, y]$. Так как оно замкнуто относительно умножения, то оно содержит элемент $(x+y)(x+4)$, эквивалентный $x^2+xy+4x+4y$ в силу закона дистрибутивности для кольца. □

12.2. ЕЩЕ РАЗ О НЕВЕТВЯЩИХСЯ ПРОГРАММАХ

Зададим снова вопрос. "Сколько арифметических операций требуется для вычисления значений произвольного полинома?" На самом деле вопрос касается числа операций $+$ и $\cdot$, необходимых для построения выражения $\sum_{i=0}^{n} a_i x^i$ (или эквивалентного ему) от формальных переменных $a_0, \ldots, a_n$ и x. Это мотивирует следующую модель вычислений, которая, по существу, есть модель неветвящейся программы, введенной в разд. 1.5.

Определение. Пусть F — поле. *Вычисление* относительно F состоит из

1) множества формальных переменных, называемых *входами*,
2) множества *имен переменных*,
3) последовательности *шагов* вида $a \leftarrow b\theta c$, где θ — один из знаков $+$, $-$ или $*$, a — *переменная*, не участвующая на предыдущих шагах, b и c — либо входы, либо элементы из F, либо имена переменных, появившиеся слева от стрелки на одном из предыдущих шагов.

Краткости ради будем писать $a \leftarrow b$ вместо $a \leftarrow b+0$ и $a \leftarrow -b$ вместо $a \leftarrow 0-b$. Элемент поля F, входящий в вычисление, называется *постоянной*.

Для определения результата вычисления нам нужно понятие *значения* переменной в вычислении. Говоря неформально, мы считаем, что вычисление производится шаг за шагом; на каждом шаге новой переменной присваивается элемент из $F[x_1, \ldots, x_n]$, где $x_1, \ldots, x_n$ — входы. *Значение* $v(a)$ переменной или входа a определяется следующим образом. Если a — вход или элемент из F, то $v(a)=a$. Если a — переменная и $a \leftarrow b\theta c$ — шаг, в котором a стоит слева, то $v(a)=v(b)\theta v(c)$.

Данное вычисление *вычисляет* множество E выражений из $F[x_1, \ldots, x_n]$ относительно поля F, если для каждого выражения $e \in E$ найдется в этом вычислении такая переменная f, что $v(f)=e$.

Заметим, что вычисление рассматривается относительно данного основного поля. Например, чтобы вычислить x^2+y^2 относительно поля F вещественных чисел, требуются два умножения, даже если

не считать умножений на постоянную. Но если F — поле комплексных чисел, достаточно одного умножения (не считая умножений на постоянные), а именно $(x+iy)$ $(x-iy)$. В качестве основного поля обычно берется поле вещественных чисел, хотя можно было бы взять и поле комплексных или рациональных чисел или какое-нибудь конечное поле. Какое поле взять, зависит от того, какие операции считать основными. Если предполагается, что мы умеем представлять вещественные числа и выполнять сложение и умножение их как основные операции, то в качестве F берется поле вещественных чисел.

Пример 12.3. Вспомним, что вычисление произведения двух комплексных чисел $a+ib$ и $c+id$ относительно поля вещественных чисел можно рассматривать как вычисление выражений $ac-bd$ и $ad+bc$. Очевидное вычисление выглядит так:

$$\begin{aligned} f_1 &\leftarrow a*c \\ f_2 &\leftarrow b*d \\ f_3 &\leftarrow f_1 - f_2 \\ f_4 &\leftarrow a*d \\ f_5 &\leftarrow b*c \\ f_6 &\leftarrow f_4 + f_5. \end{aligned}$$

Значением f_1 является ac. Аналогично $v(f_2)=bd$ и $v(f_3)=ac-bd$. Таким образом, значение f_3 равно первому выражению. Значение f_6 равно $ad+bc$, т. е. второму выражению. Следовательно, это вычисление вычисляет произведение двух комплексных чисел.

Другое вычисление для комплексного умножения использует только три вещественных умножения:

$$\begin{aligned} f_1 &\leftarrow a + b \\ f_2 &\leftarrow f_1 * c \\ f_3 &\leftarrow d - c \\ f_4 &\leftarrow a * f_3 \\ f_5 &\leftarrow f_4 + f_2 \\ f_6 &\leftarrow d + c \\ f_7 &\leftarrow b * f_6 \\ f_8 &\leftarrow f_2 - f_7. \end{aligned}$$

Легко показать, что $v(f_5)=ad+bc$ и $v(f_8)=ac-bd$. □

Должно быть ясно, что данное вычисление вычисляет некоторое выражение тогда и только тогда, когда, подставляя вместо входов произвольные значения из поля F, получаем вычисление того частного случая выражения, который получается, если подставить эти входные значения вместо формальных переменных рассматриваемого выражения.

В оставшейся части главы нас будут интересовать прежде всего нижние оценки числа умножений, необходимых для вычисления данного множества выражений, поскольку в некоторых ситуациях

умножения имеют гораздо большую сложность, чем сложения или вычитания.

Например, в гл. 6 мы видели, что умножение двух (2×2)-матриц за 7 арифметических умножений дает алгоритм умножения произвольных матриц за время $O_A(n^{\log 7})$. Использование же 18 сложений в алгоритме Штрассена не влияет на асимптотику. На первом уровне рекурсии для алгоритма Штрассена 7 умножений $((n/2)\times(n/2))$-матриц по мере роста n становятся гораздо сложнее, чем 18 (или любое другое число) сложений и вычитаний матриц того же размера.

На самом деле если бы можно было умножать две (2×2)-матрицы с помощью только шести умножений в некоммутативном кольце, то у нас мог бы быть алгоритм умножения матриц со сложностью $O_A(n^{\log 6})=O_A(n^{2,59})$ независимо от того, сколько тратится на умножение (2×2)-матриц сложений и вычитаний. (Можно, однако, показать, что для того, чтобы такой алгоритм работал для произвольного кольца, для умножения (2×2)-матриц необходимы 7 умножений; см. Хопкрофт, Керр [1971].)

Вот другой пример. В разд. 2.6 мы видели, что можно умножить два n-разрядных двоичных целых числа за $O_Б(n^{1,59})$ шагов, потому что достаточно трех умножений, чтобы вычислить выражения ac, bd и $ad+bc$. Если бы можно было вычислять эти выражения с помощью только двух умножений, то было бы $M(n)\leqslant 2M(n/2)+cn$ для некоторой постоянной c. Такое вычисление, примененное рекурсивно, привело бы к алгоритму умножения целых чисел со сложностью $O_Б(n \log n)$, который был бы лучше всех известных. К сожалению, как мы покажем, для вычисления этих выражений в любом поле требуется не менее трех умножений.

12.3. МАТРИЧНАЯ ФОРМУЛИРОВКА ЗАДАЧ

Многие задачи можно сформулировать в терминах вычисления произведения матрицы на вектор-столбец. Элементы матрицы берутся из $F[a_1, \ldots, a_n]$, где F — поле и $a_1, \ldots, a_n$ — формальные переменные. Компонентами вектора-столбца служат формальные переменные, отличные от $a_1, \ldots, a_n$.

Пример 12.4. Задачу умножения двух комплексных чисел $a+ib$ и $c+id$ можно записать так:

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac-bd \\ bc+ad \end{bmatrix}.$$

Здесь F — поле вещественных чисел, а элементы матрицы взяты из $F[a, b]$. Вектор состоит из формальных переменных c и d. □

Пример 12.5. Вычисление значений полинома $\sum_{i=0}^{n} a_i x^i$ можно выразить как

$$[1,\ x,\ x^2,\ \ldots,\ x^n]\begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{bmatrix}.$$

Здесь F — поле вещественных чисел, а элементы $(1\times(n+1))$-матрицы принадлежат $F[x]$. □

12.4. НИЖНЯЯ ГРАНИЦА ДЛЯ ЧИСЛА УМНОЖЕНИЙ, СВЯЗАННАЯ С РАНГОМ ПО СТРОКАМ

Определение. Пусть F — поле и $a_1, \ldots, a_n$ — формальные переменные. Обозначим через $F^m[a_1, \ldots, a_n]$ m-мерное пространство векторов с компонентами из $F[a_1, \ldots, a_n]$, а F^m пусть будет m-мерным пространством векторов с компонентами из F. Множество векторов $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ из $F^m[a_1, \ldots, a_n]$ называется *линейно независимым по модулю* F^m, если для любых $u_1, \ldots, u_k$ из F соотношение $\sum_{i=1}^{k} u_i \mathbf{v}_i \in F^m$ влечет за собой, что все u_i равны нулю. Если векторы $\mathbf{v}_i$ не являются линейно независимыми по модулю F^m, то их называют *зависимыми по модулю* F^m.

Линейную независимость по модулю F^m можно трактовать иначе, рассмотрев факторпространство $F^m[a_1, \ldots, a_n]/F^m$ классов эквивалентности векторов из $F^m[a_1, \ldots, a_n]$. (Векторы $\mathbf{v}_1$ и $\mathbf{v}_2$ из $F^m[a_1, \ldots, a_n]$ *эквивалентны* тогда и только тогда, когда разность $\mathbf{v}_1 - \mathbf{v}_2$ принадлежит F^m.) Линейная независимость по модулю F^m означает линейную независимость классов эквивалентности из $F^m[a_1, \ldots, a_n]/F^m$.

Пример 12.6. Пусть F — поле вещественных чисел, а a и b — формальные переменные. Тогда три вектора

$$\mathbf{v}_1 = \begin{bmatrix} 2a \\ 4b \\ 2a-2b \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} -a \\ -b \\ b+3 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} \pi \\ b-1 \\ a \end{bmatrix}$$

из $F^3[a, b]$ зависимы по модулю F^3, поскольку

$$\frac{1}{2}\mathbf{v}_1 + \mathbf{v}_2 - \mathbf{v}_3 = \begin{bmatrix} -\pi \\ 1 \\ 3 \end{bmatrix}$$

принадлежит F^3.

С другой стороны, векторы

$$\mathbf{v}_1 = \begin{bmatrix} b \\ a \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} a \\ 0 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 0 \\ b \end{bmatrix}$$

линейно независимы по модулю F^2. В самом деле, пусть $\sum_{i=1}^{3} u_i \mathbf{v}_i \in F^2$. Тогда $u_1 b + u_2 a \in F$. Поскольку ни a, ни b не принадлежит F, то $u_1 = u_2 = 0$. Кроме того, соотношение $u_1 a + u_3 b \in F$ влечет за собой $u_1 = u_3 = 0$. Следовательно, $\mathbf{v}_1$, $\mathbf{v}_2$ и $\mathbf{v}_3$ линейно независимы по модулю F^2. □

Пусть M обозначает $(r \times p)$-матрицу с элементами из F. Число линейно независимых строк в матрице M равно числу ее линейно независимых столбцов. Но если элементы матрицы M принадлежат $F[a_1, \ldots, a_n]$, то число линейно независимых строк по модулю F^p, вообще говоря, не равно числу линейно независимых столбцов в M по модулю F^r. Например, в (1×3)-матрице $[a_1 \ a_2 \ a_3]$ одна строка линейно независима по модулю F^3 и три столбца линейно независимы по модулю F. *Рангом по строкам* матрицы M по модулю F^p называется число строк в M, линейно независимых по модулю F^p. *Ранг по столбцам* матрицы M по модулю F^r определяется аналогично.

В оставшейся части раздела и в двух следующих мы будем употреблять такие обозначения:

1) F — поле,
2) $\{a_1, \ldots, a_n\}$ и $\{x_1, \ldots, x_p\}$ — непересекающиеся множества формальных переменных,
3) M есть $(r \times p)$-матрица с элементами из $F[a_1, \ldots, a_n]$,
4) $\mathbf{x}$ — вектор-столбец $[x_1, \ldots, x_p]^T$ [1]).

Теорема 12.1. *Пусть M — матрица с элементами из $F[a_1, \ldots, a_n]$ и $\mathbf{x}$ — вектор-столбец $[x_1, \ldots, x_p]^T$. Предположим, что ранг по строкам матрицы M равен r. Тогда любое вычисление произведения $M\mathbf{x}$ требует не менее r умножений.*

Доказательство. Предположим, не умаляя общности, что M имеет r строк. (В противном случае можно вычеркнуть все строки матрицы M, кроме r линейно независимых, и получить некоторую матрицу M'. Любое вычисление произведения $M\mathbf{x}$ будет вычислением и для $M'\mathbf{x}$. Отсюда следует, что если $M'\mathbf{x}$ требует r умножений, то и $M\mathbf{x}$ требует r умножений.)

Допустим, что в данном вычислении произведения $M\mathbf{x}$ необходимы s умножений. Пусть $e_1, \ldots, e_s$ — выражения, вычисляемые

[1]) Так как неудобно записывать векторы-столбцы в виде столбцов, мы будем часто записывать их в виде строк, как в гл. 7, и указывать транспонирование верхним индексом T.

на каждом из шагов с умножением. Тогда $M\mathbf{x}$ можно представить в виде линейной функции от $e_1, \ldots, e_s$ и формальных переменных, т. е.

$$M\mathbf{x} = N\mathbf{e} + \mathbf{f}, \tag{12.1}$$

где $\mathbf{e}$ — вектор $[e_1, \ldots, e_s]$, а компоненты вектора $\mathbf{f}$ представляют собой линейные функции только от $a_1, \ldots, a_n$ и $x_1, \ldots, x_p$. Все элементы $(r \times s)$-матрицы N принадлежат полю F.

Теперь допустим, что $r > s$. Это значит, что строки матрицы N линейно зависимы (элементарный факт из линейной алгебры) [1]). Таким образом, существует такой вектор $\mathbf{y} \neq \mathbf{0}$ [2]) из F^r, что $\mathbf{y}^T N = \mathbf{0}^T$. Умножив (12.1) на $\mathbf{y}^T$, получим

$$\mathbf{y}^T M\mathbf{x} = \mathbf{y}^T N\mathbf{e} + \mathbf{y}^T\mathbf{f} = \mathbf{y}^T\mathbf{f}. \tag{12.2}$$

Из (12.2) видно, что произведение $(\mathbf{y}^T M)\mathbf{x}$ равно $\mathbf{y}^T\mathbf{f}$, т. е. является линейной функцией от формальных переменных. Так как компоненты вектора $\mathbf{x}$ — различные формальные переменные, то вектор-строка $\mathbf{y}^T M$ должна состоять только из элементов поля F, а иначе в $\mathbf{y}^T M\mathbf{x}$ нашлось бы слагаемое, состоящее из произведения формальных переменных, и, значит, такое слагаемое было бы и в $\mathbf{y}^T\mathbf{f}$, а это противоречит тому, что $\mathbf{f}$ — линейная функция от формальных переменных. Поскольку $\mathbf{y} \neq \mathbf{0}$ и $\mathbf{y}^T M \in F^p$, то строки матрицы M зависимы по модулю F^p вопреки условию. Поэтому $s \geqslant r$, и теорема доказана. □

Пример 12.7. В разд. 12.2 мы рассмотрели вычисление трех выражений ac, bd и $ad+bc$ простым рекурсивным алгоритмом умножения целых чисел. Все три умножения можно записать в виде умножения матрицы на вектор, а именно

$$\begin{bmatrix} a & 0 \\ 0 & b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}. \tag{12.3}$$

Пусть $\mathbf{R}$ — поле вещественных чисел. В примере 12.6 было показано, что три строки матрицы в (12.3) линейно независимы по модулю $\mathbf{R}^2$, и, следовательно, любое вычисление этих трех выражений над вещественным полем требует трех умножений вещественных чисел. □

---

[1]) Для читателей, незнакомых с этим фактом, укажем, что доказательство *основной теоремы* (нашей леммы 12.1) позволяет доказать и этот более легкий результат.

[2]) $\mathbf{0}$ — это вектор подходящей размерности, состоящий из одних нулей.

12.5. НИЖНЯЯ ГРАНИЦА ДЛЯ ЧИСЛА УМНОЖЕНИЙ, СВЯЗАННАЯ С РАНГОМ ПО СТОЛБЦАМ

Теорема, аналогичная теореме 12.1, связанная с рангом по столбцам, верна, но доказывается с бóльшим трудом. Нам понадобится предварительный результат о множествах линейно независимых векторов.

Лемма 12.1. *Пусть* $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ — *множество, состоящее из* $k \geqslant 1$ *m-мерных векторов с компонентами из* $F[a_1, \ldots, a_n]$. *Если в* $\{\mathbf{v}_i \mid 1 \leqslant i \leqslant k\}$ *есть подмножество из* q *векторов, линейно независимых по модулю* F^m, *то для любых элементов* $b_2, \ldots, b_k$ *из* F *множество* $\{\mathbf{v}'_i \mid \mathbf{v}'_i = \mathbf{v}_i + b_i \mathbf{v}_1,\ 2 \leqslant i \leqslant k\}$ *содержит подмножество, состоящее из* $q-1$ *векторов, линейно независимых по модулю* F^m.

Доказательство. Перенумеровав при необходимости векторы $\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_k$, можно считать, что линейно независимы [1]) либо векторы из $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$, либо векторы из $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$.

Случай 1. Пусть множество $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$ состоит из линейно независимых векторов. Мы утверждаем, что векторы из $\{\mathbf{v}'_2, \mathbf{v}'_3, \ldots, \mathbf{v}'_q\}$ также линейно независимы. Чтобы доказать это, допустим, что $\sum_{i=2}^{q} c_i \mathbf{v}'_i \in F^m$ для некоторого множества элементов $c_i \in F$. Тогда $\sum_{i=1}^{q} c_i \mathbf{v}_i \in F^m$, где $c_1 = \sum_{i=2}^{q} c_i b_i$. Но так как векторы из $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$ линейно независимы, то $c_i = 0$ для $1 \leqslant i \leqslant q$. Поэтому множество $\{\mathbf{v}'_2, \mathbf{v}'_3, \ldots, \mathbf{v}'_q\}$ также состоит из линейно независимых векторов.

Случай 2. Пусть множество $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$ состоит из линейно независимых векторов. Если векторы из $\{\mathbf{v}'_2, \ldots, \mathbf{v}'_q\}$ линейно независимы, то лемма верна; поэтому допустим, что это не так. Тогда найдутся такие элементы $c_2, \ldots, c_q$ из F, не все равные 0, что $\sum_{i=2}^{q} c_i \mathbf{v}'_i \in F^m$. Пусть $c_1 = \sum_{i=2}^{q} c_i b_i$. Тогда вектор $\mathbf{w} = \sum_{i=1}^{q} c_i \mathbf{v}_i$ также принадлежит F^m.

Заметим, что $c_1 \neq 0$, ибо в противном случае вектор $\sum_{i=2}^{q} c_i \mathbf{v}_i$ оказался бы в F^m вопреки предположению, что векторы из $\{\mathbf{v}_2, \ldots, \mathbf{v}_{q+1}\}$ линейно независимы. Поэтому можно написать

$$\mathbf{v}_1 = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i \mathbf{v}_i\right).$$

Поскольку не все из $c_2, \ldots, c_q$ равны 0, можно, не умаляя общности, считать, что $c_2 \neq 0$. Повторим наше рассуждение примени-

[1]) Всюду в этом доказательстве мы опускаем слова «по модулю F^m».

тельно к $\{\mathbf{v}'_3, \mathbf{v}'_4, \ldots, \mathbf{v}'_{q+1}\}$. Если векторы из этого множества линейно независимы, то все в порядке, так что считаем, что $\sum_{i=3}^{q+1} d_i \mathbf{v}'_i \in F^m$, где $d_3, \ldots, d_{q+1}$ — элементы из F, не все равные 0. Пусть $d_1 = \sum_{i=3}^{q+1} d_i b_i$. Тогда вектор $\mathbf{x} = d_1\mathbf{v}_1 + \sum_{i=3}^{q+1} d_i \mathbf{v}_i$ принадлежит F^m. Если $d_1 = 0$, то получаем противоречие с линейной независимостью векторов из $\{\mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$. Если $d_1 \neq 0$, можно написать

$$\mathbf{v}_1 = d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i \mathbf{v}_i\right).$$

Приравняем два выражения для $\mathbf{v}_1$:

$$d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i \mathbf{v}_i\right) = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i \mathbf{v}_i\right).$$

Умножим на $c_1 d_1$ и перегруппируем члены:

$$d_1 c_2 \mathbf{v}_2 + \sum_{i=3}^{q} (d_1 c_i - c_1 d_i)\,\mathbf{v}_i - c_1 d_{q+1} \mathbf{v}_{q+1} = d_1 \mathbf{w} - c_1 \mathbf{x}.$$

Поскольку $\mathbf{w}$ и $\mathbf{x}$ принадлежат F^m, то $d_1\mathbf{w} - c_1\mathbf{x}$ также принадлежит F^m. Так как c_2 и d_1 отличны от нуля, то векторы из $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$ линейно зависимы; получили противоречие. □

Пусть M будет $(r \times p)$-матрицей с элементами из $F[a_1, \ldots, a_n]$, как и ранее. Покажем теперь, что если q столбцов матрицы M линейно независимы по модулю F^r, $q \geqslant 1$, то любое вычисление вектора $M\mathbf{x}$, где $\mathbf{x} = [x_1, \ldots, x_p]^T$, занимает по крайней мере q умножений.

Определение. Умножение называют *активным*, если в значение одного из его операндов входит одна из формальных переменных x_i, а значение другого не принадлежит полю F. Например, умножение $b*c$ активно, если $v(b) = 3 + a_2$ и $v(c) = x_1 + 2*x_3$, но неактивно, если $v(b) = 3$ и $v(c) = x_1 + 2*x_3$ или $v(b) = 3 + a_2$ и $v(c) = a_1 + 2*a_3$.

Теорема 12.2. *Пусть* $\mathbf{y}$ *будет* r*-мерным вектором с элементами из* $F[a_1, \ldots, a_n]$. *Если ранг по столбцам матрицы* M *равен* q, *то любое вычисление вектора* $M\mathbf{x} + \mathbf{y}$ *содержит не менее* q *активных умножений, где* $q \geqslant 1$.

Доказательство. Доказательство проводится индукцией по q.

Базис: $q = 1$. Существуют столбец матрицы M, не принадлежащий F^r, и, значит, элемент e матрицы M, принадлежащий $F[a_1, \ldots, a_n]$, но не F. Поэтому некоторая компонента вектора $M\mathbf{x}$ содержит произведение ex_j для некоторого j. Тогда то же верно и для $M\mathbf{x} + \mathbf{y}$. Вычисление без активных умножений может вычислять

только выражения вида $P(a_1, \ldots, a_n)+L(x_1, \ldots, x_p)$, где P — полином и L — линейная функция, оба с коэффициентами из F. Таким образом, вычисление вектора $M\mathbf{x}+\mathbf{y}$ должно включать хотя бы одно активное умножение.

Шаг индукции. Допустим, что $q>1$ и теорема верна для $q-1$. Пусть C — вычисление вектора $M\mathbf{x}+\mathbf{y}$. По предположению индукции C содержит не менее $q-1\geqslant 1$ активных умножений. Пусть $f \leftarrow g*h$ — первое такое умножение. Тогда без потери общности можно считать, что

$$v(g) = P(a_1, \ldots, a_n) + \sum_{i=1}^{p} c_i x_i, \tag{12.4}$$

где P — полином с коэффициентами из F и все c_i также принадлежат F. Более того, не умаляя общности, можно считать, что $c_1\neq 0$.

Наша стратегия состоит в том, чтобы построить из C и множества выражений $M\mathbf{x}+\mathbf{y}$ новое множество выражений $M'\mathbf{x}'+\mathbf{y}'$ с вычислением C', содержащим на одно активное умножение меньше, чем C. Кроме того, $q-1$ столбцов матрицы M' будут линейно независимы по модулю F^r. Тогда по предположению индукции C' будет содержать не менее $q-1$ активных умножений, а значит, C — не менее q активных умножений.

Конкретно, мы заменим x_1 в C некоторым выражением e, которое сделает g из (12.4) равным 0. Выражение e будет вычислимо без активных умножений. Вычисление C' начинается с вычисления выражения e, при этом значение e присваивается формальной переменной x_1 (она уже больше не будет входной переменной). Остальная часть вычисления C' состоит из C, где $f \leftarrow g*h$ заменено на $f \leftarrow 0$. Затем мы покажем, что множество выражений, вычисляемых с помощью C', можно выразить в виде $M'\mathbf{x}'+\mathbf{y}'$, где $q-1$ столбцов матрицы M' линейно независимы по модулю F^r.

Итак, приступим к изложению деталей доказательства.

Из (12.4) и допущения $c_1\neq 0$ заключаем, что $g=0$, если

$$x_1 = -c_1^{-1}\left[P(a_1, \ldots, a_n) + \sum_{i=2}^{p} c_i x_i\right]. \tag{12.5}$$

Правая часть равенства (12.5) и есть выражение e, упоминаемое выше. Вычисление C', получаемое из C описанным выше способом, вычисляет $M\mathbf{x}+\mathbf{y}$, куда вместо x_1 подставлено e. Поэтому $M'\mathbf{x}'+\mathbf{y}'$ можно записать как

$$M\begin{bmatrix} e \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_p \end{bmatrix} + \mathbf{y},$$

а это переписать в виде

$$M\begin{bmatrix} -c_1^{-1}\sum_{i=2}^{p} c_i x_i \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_p \end{bmatrix} + M\begin{bmatrix} -c_1^{-1} P(a_1, \ldots, a_n) \\ 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} + \mathbf{y}. \qquad (12.6)$$

Рассмотрим первое слагаемое в (12.6). Пусть $\mathbf{m}_i$ — это i-й столбец матрицы M. Для $2 \leqslant i \leqslant p$ положим $\mathbf{m}_i = \mathbf{m}_i - (c_i/c_1)\mathbf{m}_1$. Пусть M' — матрица, состоящая из $p-1$ столбцов, у которой i-й столбец есть $\mathbf{m}'_{i+1}$, и пусть $\mathbf{x}' = [x_2, \ldots, x_p]^T$. Таким образом, первое слагаемое в (12.6) равно $M'\mathbf{x}'$.

Второе слагаемое — вектор с компонентами из $F[a_1, \ldots, a_n]$, ибо элементы матрицы M принадлежат $F[a_1, \ldots, a_n]$. Поэтому второе и третье слагаемые в (12.6) можно объединить в новый вектор $\mathbf{y}'$ с компонентами из $F[a_1, \ldots, a_n]$. Таким образом, C' вычисляет $M'\mathbf{x}' + \mathbf{y}'$. Из леммы 12.1 непосредственно вытекает, что не менее $q-1$ столбцов матрицы M' линейно независимы по модулю F^r. Следовательно, C' содержит не менее $q-1$ активных умножений, и потому C содержит не менее q активных умножений. □

Приведем два примера применения теоремы 12.2.

Пример 12.8. Мы утверждаем, что умножение $(n \times p)$-матрицы A на p-мерный вектор $\mathbf{v}$ требует np умножений элементов. Формально, пусть a_{ij} и v_j — формальные переменные, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant p$. Тогда произведение $A\mathbf{v}$ матрицы на вектор имеет вид, показанный на рис. 12.1.

Непосредственно применяя теоремы 12.1 и 12.2 к $A\mathbf{v}$, заключаем лишь, что требуется $\mathrm{MAX}(n, p)$ умножений. Однако произведение матрицы на вектор можно также выразить в виде $M\mathbf{x}$, где в i-й строке матрицы M в столбцах с $((i-1)p+1)$-го до (ip)-го стоят $v_1, \ldots, v_p$, а в остальных столбцах — нули. Вектор $\mathbf{x}$ — это вектор-стол-

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1p} \\ a_{21} & a_{22} & \ldots & a_{2p} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{n1} & a_{n2} & \ldots & a_{np} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_p \end{bmatrix}$$

Рис. 12.1. Общий вид произведения матрицы на вектор.

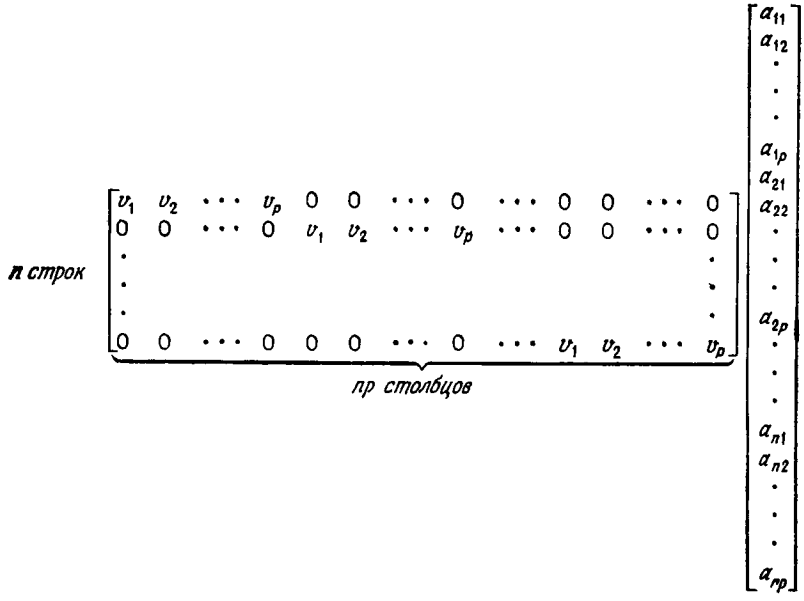


Рис. 12.2. Эквивалентная форма произведения матрицы на вектор.

бец, равный конкатенации строк матрицы A, записанной сверху вниз. M и x изображены на рис. 12.2.

Легко показать, что столбцы матрицы M линейно независимы по модулю F^n. Поэтому в силу теоремы 12.2 любое вычисление для $A\mathbf{v}$ требует не менее np умножений. □

Пример 12.9. Рассмотрим задачу вычисления полинома n-й степени $\sum_{i=0}^{n} a_i x^i$. Эту задачу можно представить как вычисление произведения матрицы M на вектор x:

$$[1,\ x,\ x^2,\ \ldots,\ x^n]\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Элементы матрицы M принадлежат $F[x]$ и $\mathbf{x}=[a_0, a_1, \ldots, a_n]^T$. Легко показать, что все столбцы матрицы M, кроме первого, линейно независимы по модулю F. Следовательно, в M содержатся n линейно независимых столбцов, и вычисление произвольного полинома n-й степени требует не менее n умножений.

Правило Горнера, вычисляющее полином по схеме

$$(\ldots((a_nx+a_{n-1})x+a_{n-2})x+\ldots+a_1)x+a_0,$$

требует в точности n умножений и, значит, оптимально в том смысле, что использует наименьшее возможное число умножений. Аналогично можно показать, что для вычисления $\sum_{i=0}^{n} a_i x^i$ нужны n сложений и вычитаний. Таким образом, правило Горнера использует наименьшее число арифметических операций, необходимых для вычисления значения полинома в одной точке. □

12.6. ГРАНИЦА ДЛЯ ЧИСЛА УМНОЖЕНИЙ, СВЯЗАННАЯ С РАССМОТРЕНИЕМ СТРОК И СТОЛБЦОВ

Объединим теоремы 12.1 и 12.2, чтобы доказать более сильный результат, чем тот, который получается, если рассматривать независимость строк и столбцов отдельно. Пусть M будет $(r\times p)$-матрицей с элементами из $F[a_1, \ldots, a_n]$, как и раньше, и $\mathbf{x}=[x_1, \ldots, x_p]^T$.

Теорема 12.3. *Пусть M содержит такую подматрицу S с q строками и с столбцами, что для любых векторов $\mathbf{u}$ и $\mathbf{v}$ из F^q и F^c соответственно элемент $\mathbf{u}^TS\mathbf{v}$ принадлежит полю F тогда и только тогда, когда $\mathbf{u}=0$ или $\mathbf{v}=0$. Тогда любое вычисление произведения $M\mathbf{x}$ требует не менее $q+c-1$ умножений.*

Доказательство. Не умаляя общности, с самого начала считаем, что в матрице M только q строк, а S состоит из ее первых c столбцов. Допустим, что $M\mathbf{x}$ можно вычислить за s умножений. Пусть $\mathbf{e}$ — вектор, компонентами которого служат s выражений, вычисляемых этими умножениями. Предположим, кроме того, что i-я компонента вектора $\mathbf{e}$ вычислена раньше j-й для $i<j$. Тогда, как в теореме 12.1, можно написать

$$M\mathbf{x}=N\mathbf{e}+\mathbf{f}, \tag{12.7}$$

где N — это $(q\times s)$-матрица с элементами из F и $\mathbf{f}$ — вектор, компоненты которого представляют собой линейные комбинации из $a_1, \ldots, a_n$ и $x_1, \ldots, x_p$.

Как и в теореме 12.1, можно заключить, что $s\geqslant q$. Если бы это было не так, то нашелся бы такой вектор $\mathbf{y}\neq 0$ из F^q, что $\mathbf{y}^TN=0^T$, откуда следовало бы, что компоненты вектора $\mathbf{y}^TM$ принадлежат F. Тогда компоненты вектора $\mathbf{y}^TS$ принадлежали бы F вопреки условию (возьмите $\mathbf{u}=\mathbf{v}$ и $\mathbf{v}^T=[1, \ldots, 1]$).

Поскольку $s\geqslant q$, можно разбить N на две матрицы N_1 и N_2, где N_1 состоит из первых $s-q+1$ столбцов матрицы N, а N_2 — из ее последних $q-1$ столбцов. Далее, пусть $\mathbf{e}_1$ и $\mathbf{e}_2$ — векторы, составленные из первых $s-q+1$ и последних $q-1$ элементов вектора $\mathbf{e}$ со-

ответственно. Тогда можно записать (12.7) в виде

$$M\mathbf{x} = N_1\mathbf{e}_1 + N_2\mathbf{e}_2 + \mathbf{f}. \tag{12.8}$$

Так как N_2 имеет размер $q\times(q-1)$, то найдется такой вектор $\mathbf{y}\neq\mathbf{0}$ из F^q, что $\mathbf{y}^TN_2=\mathbf{0}^T$. Умножим (12.8) на $\mathbf{y}^T$:

$$\mathbf{y}^TM\mathbf{x} = \mathbf{y}^TN_1\mathbf{e}_1 + \mathbf{y}^T\mathbf{f}. \tag{12.9}$$

Положим $M'=\mathbf{y}^TM$. Заметим, что M' — это $(1\times p)$-матрица, равная линейной комбинации строк матрицы M. Поскольку произведения, входящие в $\mathbf{e}_1$, можно вычислять, не обращаясь к произведениям из $\mathbf{e}_2$ (мы предположили, что первые вычисляются раньше вторых), то очевидно, что выражение $M'\mathbf{x}$ можно вычислить с помощью (12.9) за $s-q+1$ умножений. Если теперь мы сможем показать, что c столбцов матрицы M' линейно независимы по модулю F, то в силу теоремы 12.2 сможем утверждать, что $s-q+1\geqslant c$. Тогда $s\geqslant q+c-1$, что и требовалось доказать.

Итак, покажем, что первые c столбцов матрицы $M'=\mathbf{y}^TM$ линейно независимы по модулю F. Пусть $\mathbf{y}^T=[y_1, \ldots, y_q]$. Первые c элементов матрицы M' имеют вид $\sum_{i=1}^{q} y_iM_{ij}$ для $1\leqslant j\leqslant c$, где M_{ij} — это (i, j)-й элемент матрицы M. Допустим, что нашелся такой вектор $\mathbf{z}\neq\mathbf{0}$ с компонентами $z_1, \ldots, z_c$ из F, что $\sum_{j=1}^{c} z_j\sum_{i=1}^{q} y_iM_{ij}$ принадлежит F, т. е. первые c столбцов матрицы M' зависимы по модулю F. Тогда элемент $\mathbf{y}^TS\mathbf{z}$ оказался бы в F вопреки условию теоремы относительно S. Поэтому M' содержит c линейно независимых по модулю F столбцов, и теорема доказана. □

Применим теорему 12.3 к умножению двух комплексных чисел $a+ib$ и $c+id$, чтобы показать, что оно требует трех умножений вещественных чисел. Заметим, что каждая из теорем 12.1 и 12.2 в отдельности не достаточно сильна для этого.

Пример 12.10. Рассмотрим задачу умножения двух комплексных чисел $a+ib$ и $c+id$, а именно вычисления

$$M\mathbf{x} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}\begin{bmatrix} c \\ d \end{bmatrix}.$$

Пусть S — сама матрица M, а F — поле вещественных чисел и

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \neq \mathbf{0}, \qquad \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}$$

— векторы из F^2. Допустим, что

$$[y_1 \;\; y_2]\begin{bmatrix} a & -b \\ b & a \end{bmatrix}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

— элемент поля F. Это произведение равно $y_1z_1a+y_2z_1b+y_2z_2a-$ $-y_1z_2b$. Если оно принадлежит полю F, то коэффициенты при a и b должны равняться нулю. Поэтому

$$y_1z_1+y_2z_2=0 \tag{12.10}$$

и

$$y_2z_1-y_1z_2=0. \tag{12.11}$$

Допустим, что $y_1\neq 0$. Тогда из (12.11) получаем $z_2=y_2z_1/y_1$. Подставив это в (12.10) и умножив на y_1, находим, что $(y_1^2+y_2^2)z_1=0$. Поскольку $y_1\neq 0$, то $y_1^2+y_2^2\neq 0$ и, значит, $z_1=0$ [1]). Тогда в силу (12.11) $z_2=0$. Но равенства $z_1=z_2=0$ противоречат условию

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}.$$

Теперь допустим, что $y_1=0$. Если $y_2=0$, то противоречие возникает сразу. Если $y_2\neq 0$, то, поменяв ролями y_1 и y_2 в рассуждении, приведенном выше, получаем, что $z_1=z_2=0$; снова противоречие.

Таким образом, теорема 12.3 применима к Mx при $q=c=2$, так что три вещественных умножения необходимы. Программа в примере 12.3 показывает, что три умножения и достаточны. □

Теоремы 12.1—12.3 можно обобщить на случай, когда в вычислении допускаются деления. В этом случае надо допускать такие шаги в вычислении, которые при некоторых значениях входов приводят к делению на 0. Следовательно, теория, рассматривающая мультипликативные операции (т. е. умножение и деление), предполагает, что поле F бесконечно. Если бы оно было конечным, то мы могли бы столкнуться с вычислениями, не работающими ни для каких входов из-за того, что любой возможный вход приводит к делению на 0.

С учетом этих изменений каждую из теорем 12.1—12.3 можно перенести на случай обеих мультипликативных операций вместо одного умножения. В теореме 12.2 определение *активной мультипликативной операции* $f \leftarrow g*h$ или $f \leftarrow g/h$ должно быть следующим: или значение одной из переменных g и h включает в себя одну из формальных переменных x_i, а другой операнд не принадлежит F, или g принадлежит F, h включает в себя одну из формальных переменных x_i и операцией является деление.

12.7. ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА

В примере 12.9 мы показали, что вычисление значения полинома n-й степени в одной точке требует n умножений. Однако мы исходили из предположения, что полином представлен своими коэффи-

[1]) Здесь существенно предположение о том, что F — поле вещественных чисел.

циентами. Сейчас изучим задачу минимизации числа умножений, необходимых для вычисления одного полинома, когда разрешается представлять полином любым множеством параметров, вычисляемых по коэффициентам. Если бы нам было нужно вычислять данный полином несколько раз, то разумно было бы потратить некоторое время на вычисление другого представления полинома, при условии что новое представление даст более быстрое вычисление. Такое изменение представления называется предварительной обработкой (коэффициентов) [1]). В примере 12.9 умножение было активным, если один из сомножителей включал в себя коэффициенты $a_0, a_1, \ldots, a_n$, а другой не принадлежал F. Таким образом, умножение, в котором оба сомножителя включали коэффициенты и не включали переменную x, считалось за умножение. В случае предварительной обработки мы получаем "даром" все умножения, в которых ни один сомножитель не включает в себя переменную.

Определение. *Степенью* полинома от нескольких переменных называют наибольшую степень переменных этого полинома. Например, степень полинома $p(x, y)=x^3+x^2y^2$ равна 3.

Следующая лемма утверждает, что для любых $v+1$ полиномов от v переменных существует нетривиальный полином от $v+1$ переменных, тождественно равный нулю на данных полиномах. Например, пусть $p_1=x^2$ и $p_2=x+x^3$. Тогда $p_1+2p_1^2+p_1^3-p_2^2=0$ для всех x.

Лемма 12.2. *Пусть* $p_i(x_1, \ldots, x_r)$, $1\leqslant i\leqslant n$,— *полиномы. Если* $n>r$, *то существует полином* $g(y_1, \ldots, y_n)$, *не все коэффициенты которого равны нулю, но для которого* $g(p_1, \ldots, p_n)$ *как функция от* $x_1, \ldots, x_r$ *равна тождественно нулю.*

Доказательство. Пусть S_d — множество всех полиномов вида $p_1^{i_1}p_2^{i_2}\ldots p_n^{i_n}$, где $0\leqslant i_j\leqslant d$ для всех j. Заметим, что $\|S_d\|=(d+1)^n$. Пусть m — наибольшая из степеней полиномов p_i. Тогда каждый полином $q\in S_d$ — это полином от $x_1, x_2, \ldots, x_r$ степени, не большей dmn.

Любой полином степени dmn от r переменных можно представить вектором длины $(dmn+1)^r$. Компонентами этого вектора служат коэффициенты при описанных выше членах, расположенных в фиксированном порядке. Поэтому можно говорить о линейной независимости полиномов и, в частности, утверждать, что (по основной теореме линейной алгебры) любое множество, состоящее из $(dmn+1)^r+1$ полиномов из S_d, должно быть линейно зависимо, т. е. некоторая их нетривиальная сумма должна равняться нулю.

[1]) Следует отметить различие между этой ситуацией и ситуацией из разд. 8.5, где были известны все точки, в которых нужно вычислять значения полинома. Здесь же перед вычислением значений мы можем работать только с коэффициентами, а вычисления значений производятся по одному за раз, т. е. в префиксном, а не в свободном режиме.

В частности, если m, n и r фиксированы и $n>r$, то нетрудно показать, что $(d+1)^n \geqslant (dmn+1)^r$ для достаточно большого d, поскольку $(d+1)^n$— полином более высокой степени по d, чем $(dmn+1)^r$. Поэтому найдется такое d, что нетривиальная сумма полиномов из S_d будет тождественно равна нулю как функция от $x_1, \ldots, x_r$. Эта сумма и будет искомым полиномом g. □

С помощью леммы 12.2 покажем, что любое множество параметров полинома, через которое его коэффициенты выражаются в виде полиномиальных функций, должно содержать $n+1$ параметров, чтобы представлять произвольный полином n-й степени.

Лемма 12.3. *Пусть M — взаимно однозначное отображение n-мерного векторного пространства C^n коэффициентов на r-мерное векторное пространство D^r параметров. Если M^{-1} таково, что каждая компонента вектора в C^n является полиномиальной функцией компонент соответствующего вектора в D^r, то $r \geqslant n$.*

Доказательство. Пусть $r<n$ и M^{-1} задается равенствами $c_i = p_i(d_1, \ldots, d_r)$, где p_i — полиномы, d_i — параметры и c_i — коэффициенты. В силу леммы 12.2 найдется такой нетривиальный полином g, что $g(c_1, \ldots, c_n)$ — тождественный нуль для всех d_i, $1 \leqslant i \leqslant r$ [1]). Но так как сам полином g не равен тождественно нулю, то найдутся такие полиномы с коэффициентами $c_1, \ldots, c_n$, что $g(c_1, \ldots, c_n) \neq 0$. Эти полиномы нельзя представить в терминах параметров d_i; получили противоречие. □

Теперь покажем, что вычисление значения полинома n-й степени в одной точке даже при наличии предварительной обработки требует не менее $n/2$ умножений.

Теорема 12.4. *Вычисление значения произвольного полинома n-й степени в одной точке требует не менее $n/2$ умножений, даже если не считать входящие в него умножения, которые включают в себя только коэффициенты полинома.*

Доказательство. Допустим, что существует вычисление с m умножениями, включающими в себя неизвестную x. Пусть результаты этих умножений — выражения $f_1, f_2, \ldots, f_m$. Тогда каждое выражение f_i можно записать в виде

$$f_i = [L_{2i-1}(f_1, \ldots, f_{i-1}, x) + \beta_{2i-1}] * [L_{2i}(f_1, \ldots, f_{i-1}, x) + \beta_{2i}],$$

$1 \leqslant i \leqslant m$, где каждая функция L_i линейна по f_j и x, а каждая функция β_i полиномиальна по коэффициентам. Полином $p(x)$, который вычисляется, можно записать в виде

$$p(x) = L_{2m+1}(f_1, \ldots, f_m, x) + \beta_{2m+1}.$$

[1]) Заметим, что g — полином по c_i и, значит, по d_i, поскольку c_i — полиномы по параметрам d_i.

Следовательно, $p(x)$ можно представить новым множеством из $2m+1$ параметров, а именно множеством функций β_i. Более того, коэффициенты полинома $p(x)$ — полиномиальные функции параметров β_i. По лемме 12.3 $2m+1 \geqslant n+1$. Таким образом, $m \geqslant n/2$. □

Теорема 12.4 дает нижнюю оценку числа умножений, необходимых для вычисления значения полинома в одной точке, когда проводится предварительная обработка (коэффициентов). Значительный интерес вызывает проблема представления полинома, при котором его можно вычислить за $n/2$ умножений. Но надо не только находить параметры для такого представления полинома, при котором легко вычисляются его значения, но и уметь быстро вычислять эти параметры по коэффициентам полинома. Поэтому хотелось бы, чтобы параметры были рациональными функциями коэффициентов. Техника нахождения таких параметров приведена в упражнениях.

УПРАЖНЕНИЯ

12.1. Пусть F — поле и x — формальная переменная. Покажите, что множество полиномов от x с коэффициентами из F образует коммутативное кольцо $F[x]$ и единица в F является единицей в $F[x]$.

12.2 Покажите, что любое вычисление для выражений

$$ac+bd,$$
$$ac-bd,$$
$$bc+ad,$$
$$bc-ad$$

требует не менее четырех умножений, где a, b, c, d — элементы поля F.

12.3. Покажите, что любое вычисление произведения вектор-столбца на вектор-строку

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} [b_1, b_2, \ldots, b_n]$$

требует не менее mn умножений.

12.4. Полином n-й степени от двух переменных x и y имеет в общем случае вид $\sum_{i=0}^{n}\sum_{j=0}^{n} a_{ij}x^iy^j$. Покажите, что для вычисления значения этого полинома с предварительной обработкой коэффициентов необходимо и достаточно $(n+1)(n+2)/2-1$ умножений.

12.5. Покажите, что умножение тёплицевой (2×2)-матрицы (см. упражнения к гл. 6) на вектор, т. е.

$$\begin{bmatrix} b & c \\ a & b \end{bmatrix}\begin{bmatrix} d \\ e \end{bmatrix},$$

требует трех умножений.

12.6. Обобщите упр. 12.5, показав, что для умножения тёплицевой $(n\times n)$-матрицы на вектор необходимо $2n-1$ умножений. Как близко к этой границе вы можете подойти при $n=3$?

***12.7.** Можно обобщить алгоритм умножения из разд. 2.6, разбивая каждый из сомножителей на три части и затем вычисляя произведение матрицы на вектор

$$\begin{bmatrix} a & 0 & 0 \\ b & a & 0 \\ c & b & a \\ 0 & c & b \\ 0 & 0 & c \end{bmatrix}\begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

за минимально возможное число умножений. Сколько умножений требуется для решения этой задачи? Приводит ли этот подход к улучшению алгоритма из разд. 2.6?

****12.8.** Пусть F — поле и $a_1, \ldots, a_n$ и $x_1, \ldots, x_p$ — непересекающиеся множества формальных переменных. Пусть $\mathbf{x}=[x_1, \ldots, x_p]^T$, а M — $(r\times p)$-матрица с элементами из $F[a_1, \ldots, a_n]$, у которой q столбцов линейно независимы по модулю F^r. Покажите, что при предварительной обработке вектора $\mathbf{x}$, т. е. когда не учитываются произведения, включающие только $x_1, \ldots, x_p$, вычисление произведения $M\mathbf{x}$ все же требует $q/2$ умножений.

****12.9.** Пусть ровно k выражений из множества P, никакая нетривиальная линейная комбинация элементов которого не равна постоянной, представляют собой одиночные умножения (например, они имеют вид $a*b$, где a и b — формальные переменные); допустим, что q умножений достаточно, чтобы вычислить все выражения из P. Покажите, что существует вычисление для P с q умножениями, k из которых совпадают с теми выражениями из P, которые можно вычислить с помощью одного умножения.

***12.10.** Покажите, что для вычисления выражений ac и $bc+ad$ требуется не менее трех умножений. *Указание*: Воспользуйтесь упр. 12.9.

***12.11.** Покажите, что для вычисления множества из $2k$ выражений a_ic_i и $b_ic_i+a_id_i$, $1\leqslant i\leqslant k$, требуется не менее $3k$ умножений.

которые входят в один n-членный кортеж, содержащий a_{ij}. Пусть

$$f=\sum_{i=1}^{n}\sum_{j=1}^{n}\log s_{ij}.$$

Рассмотрите изменение f по мере выполнения шагов вычисления.

**12.21. Наложим следующее ограничение на вид шагов вычисления: в $a \leftarrow b\theta c$ операция θ состоит в выборе $n/2$ формальных переменных из циклического сдвига n-членного кортежа b и $n/2$ формальных переменных из дополнительных позиций в c. Найдите вычисление для множества n-членных кортежей из упр. 12.20, состоящее из $O(n \log n)$ шагов.

**12.22. Пусть G — ориентированный ациклический граф с n выделенными *истоками* (узлами, в которые не входят никакие ребра) и n выделенными *выходами* (узлами, из которых не выходят никакие ребра). Пусть X и Y — подмножества истоков и выходов соответственно, а $G(X, Y)$ — подграф, состоящий из всех ориентированных ребер, лежащих на путях из узлов множества X в узлы множества Y. *Пропускной способностью* графа $G(X, Y)$ называется наименьшее число узлов, удаление которых (вместе с входящими и выходящими ребрами) отделяет каждый узел из X от каждого узла из Y. Допустим, что для любых X и Y граф $G(X, Y)$ обладает пропускной способностью MIN ($||X||$, $||Y||$). Покажите, что для каждого n существует такой граф G с $cn \log n$ ребрами, где c — некоторая постоянная.

**12.23. *Сдвигающей схемой* называется такой ориентированный граф с n истоками, занумерованными числами от 0 до $n-1$, и n выходами, занумерованными числами от 0 до $n-1$, что для каждого s, $0 \leqslant s \leqslant n-1$, найдется множество путей, непересекающихся по узлам, которое для каждого i содержит путь из истока i в выход $(i+s)$ по модулю n.

(а) Покажите, что для каждого n существует сдвигающая схема с $2n \log n$ ребрами.

(б) Покажите, что для сдвигающей схемы асимптотически требуется $n \log n$ ребер.

(в) Покажите, что с помощью сдвигающей схемы можно вычислить транспонированную матрицу.

Определение. Пусть F — поле и $x_1, \ldots, x_n$ — формальные переменные. *Линейным* вычислением называются последовательность шагов вида $a \leftarrow \lambda_1 b + \lambda_2 c$, где a — переменная, b и c — переменные или формальные переменные, а λ_1 и λ_2 — элементы из F (называемые *постоянными*).

**12.24. Пусть F — поле комплексных чисел, A — матрица с элементами из F и $\mathbf{x}=[x_1, \ldots, x_n]^T$. Покажите, что любое линейное вычисление для $A\mathbf{x}$ требует $\log |\det(A)|/\log(2c)$ шагов, где c — наи-

(б) Покажите, что существует вычисление, минимальное по числу умножений, в котором каждое умножение производится между линейной функцией от $a_1, a_2, \ldots$ и линейной функцией от $b_1, b_2, \ldots$. (Заметьте, что это утверждение не верно, если ограничиться случаем, когда основное кольцо коммутативно.)

(в) Пусть разрешаются более общие вычисления, а именно вычисления с делениями. Покажите, что существует вычисление, минимальное по числу мультипликативных операций, которое не использует делений.

**12.18. Пусть R — некоммутативное кольцо и $\mathbf{a}$, $\mathbf{b}$, $\mathbf{x}$ — векторы-столбцы из формальных переменных $[a_1, \ldots, a_m]^T$, $[b_1, \ldots, b_n]^T$, $[x_1, \ldots, x_p]^T$ соответственно. Пусть X — матрица, элементами которой служат линейные суммы из $x_1, \ldots, x_p$. Из упр. 12.17 следует, что вычисление множества билинейных форм $(\mathbf{a}^T X)^T$ можно представить в виде

$$M(P\mathbf{a} \cdot Q\mathbf{x}),$$

где M, P и Q — матрицы с элементами из F. Определим *левое двойственное* множество выражений для $(\mathbf{a}^T X)^T$ как $(\mathbf{b}^T X^T)^T$ и определим вычисление, *P-двойственное* к вычислению $M(P\mathbf{a} \cdot Q\mathbf{x})$, как $P^T(M^T\mathbf{b} \cdot Q\mathbf{x})$. Докажите, что вычисление, P-двойственное к любому вычислению для $(\mathbf{a}^T X)^T$, вычисляет левое двойственное множество выражений для $(\mathbf{a}^T X)^T$.

*12.19. Докажите, что минимальное число умножений, необходимое для умножения $(m \times n)$-матрицы на $(n \times p)$-матрицу над некоммутативным кольцом, то же, что и для умножения $(n \times m)$-матрицы на $(m \times p)$-матрицу. *Указание*: Воспользуйтесь упр. 12.18.

До сих пор в упражнениях содержался материал, относящийся к нижним границам для числа арифметических операций. В последующих упражнениях содержится материал более общего характера. Упр. 12.20 — 12.23 касаются транспонирования матриц. Для $1 \leqslant i, j \leqslant n$ пусть a_{ij} — формальные переменные. Рассмотрим модель вычислений, в которой каждая переменная представляет собой n-членный кортеж формальных переменных. Шаг $a \leftarrow b\theta c$ такого вычисления присваивает n-членному кортежу a формальные переменные, выбранные из b или c. Эти n-членные кортежи b и c не меняются.

*12.20. Докажите, что любое вычисление множества n-членных кортежей

$$\{(a_{1i}, \ldots, a_{ni}) \mid 1 \leqslant i \leqslant n\}$$

по множеству входов $\{(a_{i1}, \ldots, a_{in}) \mid 1 \leqslant i \leqslant n\}$ требует не менее $n \log n$ шагов. *Указание*: Для каждых i и j пусть s_{ij} обозначает максимальное число формальных переменных с нижним индексом j,

которые входят в один n-членный кортеж, содержащий a_{ij}. Пусть

$$f=\sum_{i=1}^{n}\sum_{j=1}^{n}\log s_{ij}.$$

Рассмотрите изменение f по мере выполнения шагов вычисления.

**12.21. Наложим следующее ограничение на вид шагов вычисления: в $a \leftarrow b\theta c$ операция θ состоит в выборе $n/2$ формальных переменных из циклического сдвига n-членного кортежа b и $n/2$ формальных переменных из дополнительных позиций в c. Найдите вычисление для множества n-членных кортежей из упр. 12.20, состоящее из $O(n \log n)$ шагов.

**12.22. Пусть G — ориентированный ациклический граф с n выделенными *истоками* (узлами, в которые не входят никакие ребра) и n выделенными *выходами* (узлами, из которых не выходят никакие ребра). Пусть X и Y — подмножества истоков и выходов соответственно, а $G(X, Y)$ — подграф, состоящий из всех ориентированных ребер, лежащих на путях из узлов множества X в узлы множества Y. *Пропускной способностью* графа $G(X, Y)$ называется наименьшее число узлов, удаление которых (вместе с входящими и выходящими ребрами) отделяет каждый узел из X от каждого узла из Y. Допустим, что для любых X и Y граф $G(X, Y)$ обладает пропускной способностью $\mathrm{MIN}(\|X\|, \|Y\|)$. Покажите, что для каждого n существует такой граф G с $cn \log n$ ребрами, где c — некоторая постоянная.

**12.23. *Сдвигающей схемой* называется такой ориентированный граф с n истоками, занумерованными числами от 0 до $n-1$, и n выходами, занумерованными числами от 0 до $n-1$, что для каждого s, $0 \leqslant s \leqslant n-1$, найдется множество путей, непересекающихся по узлам, которое для каждого i содержит путь из истока i в выход $(i+s)$ по модулю n.

(а) Покажите, что для каждого n существует сдвигающая схема с $2n \log n$ ребрами.

(б) Покажите, что для сдвигающей схемы асимптотически требуется $n \log n$ ребер.

(в) Покажите, что с помощью сдвигающей схемы можно вычислить транспонированную матрицу.

Определение. Пусть F — поле и $x_1, \ldots, x_n$ — формальные переменные. *Линейным* вычислением называются последовательность шагов вида $a \leftarrow \lambda_1 b + \lambda_2 c$, где a — переменная, b и c — переменные или формальные переменные, а λ_1 и λ_2 — элементы из F (называемые *постоянными*).

**12.24. Пусть F — поле комплексных чисел, A — матрица с элементами из F и $\mathbf{x}=[x_1, \ldots, x_n]^T$. Покажите, что любое линейное вычисление для $A\mathbf{x}$ требует $\log |\det(A)|/\log(2c)$ шагов, где c — наи-

больший из модулей [1]) постоянных, входящих в шаги этого вычисления.

*12.25. Покажите, что для преобразования Фурье вектора $[x_1, \ldots, x_n]$ линейное вычисление с постоянными, по модулю не превосходящими 1, требует $\frac{1}{2} n \log n$ шагов.

Следующие 6 упражнений касаются минимального числа логических элементов с двумя входами, необходимых для реализации булевой функции.

*12.26. Покажите, что каждую булеву функцию от n переменных можно реализовать с помощью не более $n2^n$ логических элементов с двумя входами.

*12.27. Покажите, что для каждого n существует булева функция от n переменных, наименьшая реализация которой требует примерно $2^n/n$ логических элементов с двумя входами.

**12.28. О сложности реализации конкретных булевых функций известно мало. Однако, если ограничиться реализациями на схемах, являющихся деревьями, то можно показать, что для некоторых функций требуется $n^2/\log n$ логических элементов. Докажите, что следующее условие достаточно для того, чтобы булева функция от n переменных требовала для своей реализации не менее $n^2/\log n$ логических элементов.

Условие. Пусть $f(x_1, \ldots, x_n)$ — булева функция от n переменных. Разобьем множество переменных x_i на $b=n/\log n$ блоков по $\log n$ переменных в каждом. Присвоим значения (0 или 1) всем переменным в $b-1$ блоках. Это можно сделать $2^n/n$ способами. Результатом будет функция от $\log n$ переменных, совпадающая с одной из 2^n функций от $\log n$ переменных. Функция, получаемая в действительности, зависит от значений, присвоенных переменным в $b-1$ блоках. Пусть B_i — блок переменных, которым значения не присвоены. Если присваивание подходящих значений переменным не из B_i дает порядка $2^n/n$ различных функций от переменных из B_i, причем это выполняется для каждого i, $1 \leqslant i \leqslant \log n$, то любая древовидная схема для f должна содержать $n^2/\log n$ логических элементов.

**12.29. Пусть $m=2+\log n$ и $\{t_{ij} | 1 \leqslant i \leqslant n/m, \ 1 \leqslant j \leqslant m\}$ — такое множество m-мерных векторов из 0 и 1, что каждый вектор t_{ij} имеет среди своих компонент не менее двух единиц. Пусть

---

[1]) *Модуль* числа $a+bi$ равен $\sqrt{a^2+b^2}$.

$$f(x_{11}, x_{12}, \ldots, x_{n/m,m}) = \bigoplus_{\substack{1\leqslant i\leqslant n/m \\ 1\leqslant j\leqslant m}} x_{ij} \cdot \left[\bigoplus_{\substack{1\leqslant k\leqslant n/2 \\ k\neq i}} \prod_{\substack{1\leqslant l\leqslant m \\ \text{такие, что} \\ t^l_{ij}=1}} x_{kl} \right],$$

где t^l_{ij} — это l-я компонента вектора $\mathbf{t}_{ij}$. Докажите, что в любой реализации функции f с помощью древовидной схемы не менее $n^2/\log n$ логических элементов.

*12.30. Постройте другие булевы функции, для реализации которых с помощью древовидных схем требуется (а) $n^{3/2}$ и (б) $n^2/\log n$ логических элементов.

12.31. Пусть $S_i(x_1, \ldots, x_n)$ — симметрическая булева функция, принимающая значение 1 тогда и только тогда, когда в точности i переменных x_i имеют значение 1.

(а) Найдите реализацию функции $S_3(x_1, \ldots, x_n)$ с минимально возможным числом логических элементов.

(б) Найдите древовидную реализацию для $S_3(x_1, \ldots, x_n)$ с минимально возможным числом логических элементов.

**12.32. В примере 12.9 мы доказали, что любая схема, пригодная для вычисления значений произвольного полинома n-й степени, требует n умножений. Постройте конкретный полином n-й степени с вещественными коэффициентами, для вычисления значений которого требуется n умножений.

**12.33. Докажите, что умножение матриц в полукольце положительных целых чисел с операциями MIN и $+$ требует cn^3 операций, где c — постоянная, $0<c<1$.

**12.34. Докажите, что умножение булевых матриц с операциями И и ИЛИ требует n^3 операций.

Упр. 12.35 и 12.36 касаются такого представления полинома, при котором значение полинома в точке можно вычислить примерно за $n/2$ умножений.

**12.35. Пусть $p(x)$ — полином n-й степени, где n — нечетное целое число, большее 1. Запишем его в виде

$$p(x) = (x^2 - \alpha)\, q(x) + bx + c.$$

(а) Покажите, что при подходящем α можно взять $b=0$.

(б) Покажите, что существуют такие α_i и β_i, что $p(x)$ можно представить в виде

$$p(x) = (x^2-\alpha_1)[(x^2-\alpha_2)[\ \ldots\ ]+\beta_2]+\beta_1,$$

и, значит, полином $p(x)$, представленный этими α_i и β_i, можно вычислить за $\lfloor n/2 \rfloor + 2$ умножений.

(в) Параметры α_i в (б) могут оказаться комплексными числами. Как преодолеть эту трудность? *Указание*: Подставьте $y+c$ вместо x в $p(x)$ при подходящем c и вместо вычисления значения $p(x)$ в точке $x=x_0$ вычислите некоторый полином $p'(y)$ в точке $y=x_0-c$.

Упр. 12.35 не дает метода для вычисления α_i. Более того, параметры α_i не обязательно рациональные числа. В следующем упражнении делается попытка преодолеть эти трудности.

**12.36. Пусть m — целое число. Для $1\leqslant l\leqslant m$ положим

$$r_{l0}=\frac{(m-l)^2+(l-1)^2+3m+1}{2}$$

и $r_{lj}=m-l+j+1$, $1\leqslant j\leqslant l$. Определим цепь с параметрами α_{ij} и β_{kl} как вычисление вида

$$\begin{aligned}
c_{l1}(x) &\leftarrow (x^{r_{l0}}+\alpha_{l1})(x^{r_{i1}}+\beta_{i1}),\\
c_{i2}(x) &\leftarrow (c_{i1}+\alpha_{i2})(x^{r_{i2}}+\beta_{i2}),\\
&\vdots\\
c_{ii}(x) &\leftarrow (c_{i,i-1}+\alpha_{ii})(x^{r_{ii}}+\beta_{ii}).
\end{aligned}$$

Пусть $p(x)=\sum_{i=1}^{l} c_{ii}(x)$.

(а) Докажите, что показатель наибольшей степени переменной x, коэффициент при которой зависит от α_{ij}, равен $\sum_{k=j}^{l} r_{ik}$, а показатель наибольшей степени переменной x, коэффициент при которой зависит от β_{ij}, равен $\sum_{k=0}^{l} r_{ik}-r_{ij}$.

(б) Докажите, что всякий полином степени, не превосходящей $m(m+1)+1$, со старшим коэффициентом, равным 1, можно представить $m(m+1)+1$ параметрами, так что (i) этот полином можно вычислить по параметрам за $m(m+1)/2+O(m)$ умножений и (ii) параметры являются рациональными функциями от коэффициентов.

Проблемы для исследования

Каждой неветвящейся программе можно следующим образом поставить в соответствие ориентированный ациклический граф. Узлы графа представляют входы и переменные. Если $f\leftarrow g*h$ — шаг, то из g в f и из h в f идут ориентированные ребра. Заметим, что число шагов программы ровно вдвое меньше числа ребер. Поэтому нижняя оценка числа ребер в любом ориентированном ациклическом графе, соответствующем вычислению для конкретной задачи, дает нижнюю оценку числа шагов в любой неветвящейся программе для этой задачи.

12.37. Докажите или опровергните, что для больших n всякий ориентированный ациклический граф с n истоками и n выходами, удовлетворяющий условию на пропускные способности из упр. 12.22, содержит не менее $n \log n$ ребер.

12.38. Удовлетворяет ли каждый граф, соответствующий неветвящейся программе для умножения двух n-разрядных двоичных целых чисел (предполагается, что используются битовые операции), условию на пропускные способности из упр. 12.22?

Замечания по литературе

Теорема 12.2 и общая формулировка задачи, изучаемой в этом разделе, принадлежат Винограду [1970а]. Теоремы 12.1 и 12.3 взяты из работы Фидуччиа [1971]. Тот факт, что без использования делений для вычисления значения полинома n-й степени требуется n умножений, приписывается (Кнутом [1969]) Гарсиа [1]). Пан [1966] распространил этот результат на мультипликативные операции. Моцкин [1955] показал, что для вычисления значения полинома с предварительной обработкой коэффициентов необходимо $\lfloor n/2 \rfloor + 1$ умножений. Одной из первых в этом направлении была работа Островского [1954]. Белага [1961] усилил результат Моцкина, а также получил оценку для числа сложений — вычитаний. Виноград [1970б] показал, что для умножения комплексных чисел необходимы три умножения.

Упр. 12.7 обсуждается Виноградом [1973]. Материал о нижних оценках для умножения матриц (упр. 12.9—12.16) основан на работе Хопкрофта, Керра [1971]. Результаты упр. 12.17 независимо получены несколькими авторами. Часть (в) Виноград (частное сообщение) приписывает Унгару. Упр. 12.18 и 12.19 заимствованы у Хопкрофта, Мусинского [1973]. Упр. 12.20—12.22, 12.37 и 12.38 основаны на беседах с Флойдом. Упр. 12.24 и 12.25 взяты у Моргенштерна [1973], упр. 12.28 и 12.29 — у Нечипорука [1966], 12.30(а) — у Харпера, Сэвиджа [1972], упр. 12.32 — у Штрассена [1974], упр. 12.33 — у Керра [1970], упр. 12.34 — у Пратта [1974], упр. 12.35 — у Ива [1964] и упр. 12.36 — у Рабина, Винограда [1971].

Еще некоторые попытки получения нижних оценок для числа арифметических операций были сделаны Бородиным, Куком [1974], Кедемом [1974] и Киркпатриком [1972, 1974].

---

[1]) Этот результат был впервые доказан В. Я. Паном в 1960 г. в более сильной форме и изложен в его статье [1962].

Читатель, интересующийся алгебраической трактовкой вопросов сложности вычисления наборов полиномов, может обратиться к многочисленным работам Ф. Штрассена, например [1973, 1976].— *Прим. перев.*

СПИСОК ЛИТЕРАТУРЫ [1]

Адельсон-Вельский Г. М., Ландис Е. М.
[1962] Один алгоритм организации информации, *Доклады АН СССР*, **146**, № 2, 263—266.

Арлазаров В. Л., Диниц Е. А., Кронрод М. А., Фараджев И. А.
[1970] Об экономном построении транзитивного замыкания графа, *Доклады АН СССР*, **194**, № 3, 487—488.

Ахо (ред.) (Aho A. V.)
[1973] Currents in the theory of computing, Prentice-Hall, Englewood Cliffs, N.J.

Ахо, Гэри, Ульман (Aho A. V., Garey M. R., Ullman J. D.)
[1972] The transitive reduction of a directed graph, *SIAM J. Comput.*, **1**, № 2, 131-137.

Ахо, Стейглиц, Ульман (Aho A. V., Steiglitz K., Ullman J. D.)
[1974] Evaluating polynomials at fixed sets of points, Bell Laboratories, Murray Hill, N.J. См. также *SIAM J. Comput.*, **4**, № 4 (1975), 533—539.

Ахо, Ульман (Aho A. V., Ullman J. D.)
[1972] The theory of parsing, translation and compiling. Vol. 1: Parsing, Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Ахо А., Ульман Дж., Теория синтаксического анализа, перевода и компиляции. Том 1: Синтаксический анализ, М., Мир, 1978.)
[1973] The theory of parsing, translation and compiling. Vol. 2: Compiling, Prentice-Hall, Englewood Cliffs, N.J. (Русский перевод: Ахо А., Ульман Дж., Теория синтаксического анализа, перевода и компиляции. Том 2: Компиляция, М., Мир, 1978.)

Ахо, Хиршберг, Ульман (Aho A. V., Hirschberg D. S., Ullman J. D.)
[1974] Bounds on the complexity of the maximal common subsequence problem, IEEE 15th Annual Symposium on Switching and Automata Theory, New Orleans, 104—109.

Ахо, Хопкрофт, Ульман (Aho A. V., Hopcroft J. E., Ullman J. D.)
[1968] Time and tape complexity of pushdown automaton languages. *Inform. and Control.*, **13**, № 3, 186—206. (Русский перевод в сб. «Языки и автоматы», М., Мир, 1975, стр. 185—197.)
[1974] On finding lowest common ancestors in trees, *SIAM J. Comput.*, **5**, № 1, 115—132.

Банч, Хопкрофт (Bunch J., Hopcroft J. E.)
[1974] Triangular factorization and inversion by fast matrix multiplication, *Math. Comput.*, **28**, № 125, 231—236.

Белага Э. Г.
[1961] О вычислении значений многочленов от одного переменного с предва-

[1] Работы, отмеченные знаком °, добавлены при переводе. — *Прим. перев.*

рительной обработкой коэффициентов. Проблемы кибернетики, вып. 5, М., Физматгиз, 7—15.

Беллман (Bellman R. E.)
[1957] Dynamic programming, Princeton University Press, Princeton, N.J. (Русский перевод: Беллман Р., Динамическое программирование, М., ИЛ, 1960.)

Берж (Berge C.)
[1958] The theory of graphs and its applications, Wiley, New York. (Русский перевод: Берж С., Теория графов и ее применение, М., ИЛ, 1962.)

Бёркхард (Burkhard W. E.)
[1973] Nonrecursive tree traversal algorithms, Proc. 7th Annual Princeton Conference on Information Sciences and Systems, 403—405.

Биркгоф, Барти (Birkhoff G., Bartee T. C.)
[1970] Modern applied algebra, McGraw-Hill, New York. (Русский перевод: Биркгоф Г., Барти Т., Современная прикладная алгебра, М., Мир, 1976.)

Блюм (Blum M.)
[1967] A machine independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.*, 14, № 2, 322—336. (Русский перевод в сб. «Проблемы математической логики», М., Мир, 1970, стр. 401—422.)

Блюм, Флойд, Пратт, Ривест, Тарьян (Blum M., Floyd R. W., Pratt V. R., Rivest R. L., Tarjan R. E.)
[1972] Time bounds for selection, *J. Comput. and Syst. Sci.*, 7, № 4, 448—461.

Блюстейн (Bluestein L. J.)
[1970] A linear filtering approach to the computation of the discrete Fourier transform, *IEEE Trans. Electroacoustics*, **AU-18**, № 4, 451—455. См. также Рабинер, Ридер [1972], 317—321.

Бородин (Borodin A. B.)
[1973a] Computational complexity — theory and practice, см. Ахо [1973], 35—89.
[1973б] On the number of arithmetics required to compute certain functions — circa May 1973, см. Трауб [1973], 149—180.

Бородин, Кук (Borodin A. B., Cook S. A.)
[1974] On the number of additions to compute specific polynomials, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 342—347.

Бородин, Мунро (Borodin A. B., Munro I.)
[1971] Evaluating polynomials at many points, *Inform. Process. Lett.*, 1, № 2, 66—68.
[1975] The computational complexity of algebraic and numeric problems, American Elsevier, N. Y.

Браун (Brown W. S.)
[1971] On Euclid's algorithm and greatest common divisors, *J. Assoc. Comput. Mach.*, 18, № 4, 478—504.
[1973] ALTRAN user's manual (3rd edition), Bell Laboratories, Murray Hill, N.J.

Бруно, Сети (Bruno J. L., Sethi R.)
[1974] Register allocation for a one-register machine, Proc. 8th Annual Princeton Conference on Information Sciences and Systems.

Бук (Book R. V.)
[1972] On languages accepted in polynomial time, *SIAM J. Comput.* 1, № 4, 281—287.
[1974] Comparing complexity classes, *J. Comput. and Syst. Sci.*, 9, № 2, 213—229.

Вагнер (Wagner R. A.)
[1974] A shortest path algorithm for edge-sparse graphs, Dept. Syst. and Inform. Sci., Vanderbilt University, Nashville, Tennessee.

Вагнер, Фишер (Wagner R. A., Fischer M. J.)
[1974] The string-to-string correction problem, *J. Assoc. Comput. Mach.*, **21**, № 1, 168—173.
Вайнер (Weiner P.)
[1973] Linear pattern matching algorithms, IEEE 14th Annual Symposium on Switching and Automata Theory, 1—11.
Валиант (Valiant L. G.)
[1974] General context-free recognition in less than cubic time, Dept. Comput. Sci., Carnegie-Mellon University, Pittsburg, Pa. См. также *J. Comput. and Syst. Sci.*, **10**, № 2 (1975), 308—315.
Вари (Vari T. M.)
[1974] Some complexity results for a class of Toeplitz matrices, неопубликованное сообщение, York Univ., Toronto, Ontario, Canada.
Виноград (Winograd S.)
[1965] On the time required to perform addition, *J. Assoc. Comput. Mach.*, **12**, № 2, 277—285. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 6, М., Мир, 1969, стр. 41—54.)
[1967] On the time required to perform multiplication, *J. Assoc. Comput. Mach.*, **14**, № 4, 793—802 (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 6, М., Мир, 1969, стр. 55—71.)
[1970а] On the number of multiplications necessary to compute certain functions, *Comm. Pure and Appl. Math.*, **23**, 165—179.
[1970б] On the multiplication of 2×2 matrices, IBM Res. Rept. RC 267, January 1970.
[1970в] The algebraic complexity of functions, Proc. Intern. Congress of Mathematicians (1970), vol. 3, Gauthier-Villars, Paris, 1971, 283—288.
[1973] Some remarks on fast multiplication of polynomials, см. Трауб [1973], 181—196.
Галлер, Фишер (Galler B. A., Fischer M. J.)
[1964] An improved equivalence algorithm, *Comm. ACM*, **7**, № 5, 301—303.
Гейл, Карп (Gale D., Karp R. M.)
[1970] A phenomenon in the theory of sorting, IEEE 11th Annual Symposium on Switching and Automata Theory, 51—59.
Гилберт, Мур (Gilbert E. N., Moore E. F.)
[1959] Variable length encodings, *Bell. Syst. Techn. J.*, **38**, № 4, 933—963.
Годбоул (Godbole S. S.)
[1973] On efficient computation of matrix chain products, *IEEE Trans. Comput.*, **C-22**, № 9, 864—966.
Грей, Харрисон, Ибарра (Gray J. N., Harrison M. A., Ibarra O. H.)
[1967] Two way pushdown automata, *Inform. and Contr.*, **11**, № 3, 30—70.
Грэхем (Graham R. L.)
[1972] An efficient algorithm for determining the convex hull of a planar set, *Inform. Process. Lett.*, **1**, № 4, 132—133.
Гуд (Good I. J.)
[1958] The interaction algorithm and practical Fourier series, *J. Roy. Statist. Soc.*, Ser. B, **20**, 361—372; Addendum, *там же*, **22** (1960), 372—375.
Гэри, Джонсон, Стокмейер (Garey M. R., Johnson D. S., Stockmeyer L. J.)
[1974] Some simplified polynomial complete problems, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 47—63.
Данциг, Блаттнер, Рао (Dantzig G. B., Blattner W. O., Rao M. R.)
[1966] All shortest routes from a fixed origin in a graph, Theory of graphs, Intern. Symp., Rome, July 1966. (Труды симпозиума опубликованы изд-вом Gordon and Breach, New York, 1967, 85—90.)
Дейкстра (Dijkstra E. W.)
[1959] A note on two problems in connection with graphs, *Numer. Math.*, **1**, 269—271.

Джентльмен (Gentleman W. M.)
[1972] Optimal multiplication chains for computing powers of polynomials, *Math. Comput.*, **26**, № 120, 935—940.
Джентльмен, Джонсон (Gentleman W. M., Johnson S. C.)
[1973] Analysis of algorithms, a case study: determinants of polynomials, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 135—141.
Джентльмен, Санд (Gentleman W. M., Sande G.)
[1966] Fast Fourier transforms for fun and profit, Proc. AFIPS 1966 Fall Joint Computer Conf., vol. 29, Spartan, Washington, D. C., 563—578.
Джонсон (Johnson D. B.)
[1973] Algorithms for shortest paths, Ph. D. Thesis, Dept. Comput. Sci., Cornell University, Ithaca, New York.
Джонсон (Johnson S. C.)
[1974] Sparse polynomial arithmetic, Bell Laboratories, Murrary Hill, N.J.,
Джоунс (Jones N. D.)
[1973] Reducibility among combinatorial problems in $\log n$ space, Proc. 7th Annual Princeton Conference on Information Sciences and Systems, 547—551.
Диветти, Грасселли (Divetti L., Grasselli A.)
[1968] On the determination of minimum feedback arc and vertex sets, *IEEE Trans. Circuit Theory*, **CT-15**, № 1, 86—89.
Дэниелсон, Ланцош (Danielson G. C., Lanczos C.)
[1942] Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids, *J. Franklin Inst.*, **233**, 365—380, 435—452.
Дэвис (Davis M.)
[1958] Computability and unsolvability, McGraw-Hill, New York.
Зивекинг (Sieveking M.)
[1972] An algorithm for division of power series, *Computing*, **10**, 153—156.
Ибарра (Ibarra O. H.)
[1972] A note concerning nondeterministic tape complexities, *J. Assoc. Comput. Mach.*, **19**, № 4, 608—612.
Ибарра, Сахни (Ibarra O. H., Sahni S. K.)
[1973] Polynomial complete fault detection problems, Techn. Rept. 74-3, Comput. Inform. and Control Sci., University of Minnesota, Minneapolis. См. также *IEEE Trans. Comput.*, **C-24**, № 3, (1975), 242—249.
Ив (Eve J.)
[1964] The evaluation of polynomials, *Numer. Math.*, **6**, 17—21.
Ивен (Even S.)
[1973] An algorithm for determining whether the connectivity of a graph is at least k, TR-73-184, Dept. Computer Sci., Cornell University, Ithaca, N.Y.
Карацуба А. А., Офман Ю. П.
[1961] Умножение многозначных чисел на автоматах, *Доклады АН СССР*, **145**, № 2, 293—294.
Карп (Karp R. M.)
[1972] Reducibility among combinatorial problems, см. Миллер, Тэчер [1972], 85—104. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 12, М., Мир, 1975, стр. 16—38.)
Карп, Миллер, Розенберг (Karp R. M., Miller R. E., Rosenberg A. L.)
[1972] Rapid identification of repeated patterns in strings, trees and arrays, Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, Colorado, 125—136.
Касами (Kasami T.)
[1965] An efficient recognition and syntax algorithm for context-free languages, Sci. Rept. AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Mass.
Кедем (Kedem Z.)
[1974] On the number of multiplications and divisions required to compute certain rational functions, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 334—341.

Керр (Kerr L. R.)
[1970] The effect of algebraic structure on the computational complexity of matrix multiplications, Ph. D. Thesis, Cornell University, Ithaca, N.Y.

Киркпатрик (Kirkpatrick D.)
[1972] On the additions necessary to compute certain functions, Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, Colorado, 94—101.
[1974] Determining graph properties from matrix representations, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 84—90.

Кислицын С. С.
[1964] О выделении k-го элемента упорядоченной совокупности путем попарных сравнений, *Сиб. матем. журнал*, 5, № 3, 557—564.

Клини (Kleene S. C.)
[1966] Representation of events in nerve nets and finite automata, в сб. «Automata Studies», под ред. Shannon C., McCarthy J., Princeton University Press, 3—40. (Русский перевод в сб. «Автоматы», М., ИЛ, 1956, стр. 15—67.)

Кнут (Knuth D. E.)
[1968] The art of computer programming. Vol. 1: Fundamental algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д., Искусство программирования для ЭВМ. Том 1: Основные алгоритмы, М., Мир, 1976.)
[1969] The art of computer programming. Vol. 2: Seminumerical algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д., Искусство программирования для ЭВМ. Том 2: Получисленные алгоритмы, М., Мир, 1977.)
[1971] Optimum binary search trees, *Acta Inform.*, 1, 14—25.
[1973a] The art of computer programming. Vol. 3: Searching and sorting, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д., Искусство программирования для ЭВМ. Том 3: Поиск и сортировка, М., Мир, 1978.)
[1973б] Notes on pattern matching, University of Trondheim, Norway.

Кнут, Пратт (Knuth D. E., Pratt V. R.)
[1971] Automata theory can be useful, Stanford University, Stanford, California.

Коллинз (Collins G. E.)
[1973] Computer algebra of polynomials and rational functions, *Amer. Math. Monthly*, 80, № 7, 725—754.

Констейбл, Хант, Сахни (Constable R. L., Hunt H. B. III, Sahni S. K.)
[1974] On the computational complexity of scheme equivalence. Proc. 8th Annual Princeton Conference on Information Sciences and Systems.

Кохави, Пац (ред.) (Kohavi Z., Paz A.)
[1971] Theory of machines and computations, Academic Press, New York.

Крейн (Crane C. A.)
[1972] Linear lists and priority queues as balanced binary trees, Ph. D. Thesis, Stanford University.

Крускал (Kruskal J. B. Jr)
[1956] On the shortest spanning subtree of a graph and the travelling salesman problem, *Proc. Amer. Math. Soc.*, 7, № 1, 48—50.

Кук (Cook S. A.)
[1966] On the minimum computation time of functions, Doctoral Thesis, Harvard University, Cambridge, Mass..
[1971a] Linear time simulation of deterministic two-way pushdown automata, Proc. IFIP Congr. 71, TA-2, North-Holland, Amsterdam, 172—179.
[1971б] The complexity of theorem proving procedures, Proc. 3d Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, 151—159. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 12, М., Мир, 1975, стр. 5—15.)
[1973] A hierarchy for nondeterministic time complexity, *J. Comput. and Syst. Sci.*, 7, № 4, 343—353.

Кук, Аандераа (Cook S. A., Aanderaa S. O.)
[1969] On the minimum complexity of functions, *Trans. Amer. Math. Soc.*, **142**, 291—314. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 8, М., Мир, 1971, стр. 168—200.)
Кук, Рекхау (Cook S. A., Reckhow R. A.)
[1973] Time-bounded random access machines, *J. Comput. and Syst. Sci.*, **7**, № 4, 354—375.
[1974] On the length of proofs in the propositional calculus, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 135—148.
Кули, Льюис, Уелч (Cooley J. M., Lewis P. A., Welch P. D.)
[1967] History of the fast Fourier transform, *Proc. IEEE*, **55**, 1675—1677.
Кули Тьюки (Cooley J. M., Tukey J. W.)
[1965] An algorithm for the machine calculation of complex Fourier series, *Math. Comput.*, **19**, 297—301.
Кунг (Kung H. T.)
[1973] Fast evaluation and interpolation, Dept. Comput. Sci., Carnegie-Mellon University, Pittsburgh.
Липсон (Lipson J.)
[1971] Chinese remainder and interpolation algorithms, Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, 372—391.
Льюис, Стирнз, Хартманис (Lewis P. M. II, Stearns R. E., Hartmanis J.)
[1965] Memory bounds for recognition of context-free and context-sensitive languages, IEEE 6th Annual Symposium on Switching Circuit Theory and Logic Design, 191—202. (Русский перевод в сб. «Проблемы математической логики», М., Мир, 1970, стр. 320—338.)
Лю (Liu C. L.)
[1968] Introduction to combinatorial mathematics, McGraw-Hill, New York.
[1972] Analysis and synthesis of sorting algorithms, *SIAM J. Comput.*, **1**, № 4, 290—304.
Мак-Лейн, Биркгоф (MacLane S., Birkhoff G.)
[1967] Algebra, Macmillan, New York.
Мак-Нотон, Ямада (McNaughton R., Yamada H.)
[1960] Regular expressions and state graphs for automata, *IRE Trans. Electron. Comput.*, **9**, № 1, 39—47. См. также Мур [1964], 157—174.
Матиясевич Ю. В.
[1971] О распознавании в реальное время отношения вхождения, Записки научн. семинаров Ленингр. отд. Матем. ин-та АН СССР, т. 20, 104—114.
Мейер (Meyer A. R.)
[1972] Weak monadic second order theory of successor is not elementary recursive, Proj. MAC Rept., MIT, Cambridge, Mass. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 12, М., Мир, 1975, стр. 62—77.)
Мейер, Стокмейер (Meyer A. R., Stockmeyer L. J.)
[1972] The equivalence problem for regular expressions with squaring requires exponential space, IEEE 13th Annual Symposium on Switching and Automata Theory, 125—129.
[1973] Nonelementary word problems in automata and logic, Proc. AMS Symposium on Complexity of Computation, April 1973.
Миллер, Тэчер (ред.) (Miller R. E., Thatcher J. W.)
[1972] Complexity of computer computations, Plenum Press, New York.
Минский (Minsky M.)
[1967] Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Минский М., Вычисления и автоматы, М., Мир, 1971.)
Моенк (Moenck R.)
[1973] Fast computations of GCD's, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 142—151.

Моенк, Бородин (Moenck R., Borodin A. B.)
[1972] Fast modular transforms via division, IEEE 13th Annual Symposium on Switching and Automata Theory, 90—96.
Моргенштерн (Morgenstern J.)
[1973] Note on a lower bound of the linear complexity of the fast Fourier transform, *J. Assoc. Comput. Mach.*, **20**, № 5, 305—306.
Моррис (Morris R.)
[1968] Scatter storage techniques, *Comm. ACM*, **11**, № 1, 35—44.
Моррис, Пратт (Morris J. H. Jr., Pratt V. R.)
[1970] A linear pattern matching algorithm, Tech. Rept. № 40, Comput. Centre, University of California, Berkeley.
Моцкин (Motzkin T. S.)
[1955] Evaluation of polynomials and evaluation of rational functions, *Bull. Amer. Math. Soc.*, **61**, 163.
Мунро (Munro J.)
[1971] Efficient determination of the transitive closure of a directed graph, *Inform. Process. Lett.*, **1**, № 2, 56—58.
Мур (ред.) (Moore E. F.)
[1964] Sequential machines: selected papers, Addison-Wesley, Reading, Mass.
Мураока, Кукк (Muraoka Y., Kuck D. J.)
[1973] On the time required for a sequence of matrix products, *Comm. ACM*, **16**, № 1, 22—26.
Нечипорук Э. И.
[1966] Об одной булевской функции, *Доклады АН СССР*, **169**, № 4, 765—766.
Нивергельт, Рейнгольд (Nievergelt J., Reingold E. M.)
[1973] Binary search trees of bounded balance, *SIAM J. Comput.*, **2**, № 1, 33—43.
Николсон (Nicholson P. J.)
[1971] Algebraic theory of finite Fourier transforms, *J. Comput. and Syst. Sci.*, **5**, № 5, 524—547.
Островский (Ostrowski A. M.)
[1954] On two problems in abstract algebra connected with Horner's rule, Studies presented to R. von Mises. Academic Press, N. Y.
Офман Ю. П.
[1962] Об алгорифмической сложности дискретных функций, *Доклады АН СССР*, **145**, № 1, 48—51.
Пан В. Я.
°[1962] О некоторых способах вычисления значений многочленов, Проблемы кибернетики, вып. 7, 21—30.
[1966] О способах вычисления значений многочленов, *Успехи матем. наук*, **21**, № 1(127), 103—134.
Патерсон, Фишер, Мейер (Paterson M. S., Fischer M. J., Meyer A. R.)
[1974] An improved overlap argument for on-line multiplication, Complexity Comput. (SIAM-AMS Proc., Vol. 7), Providence, N.J., 97—112. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 14, М., Мир, 1977, стр. 78—94.)
Пол (Pohl J.)
[1972] A sorting problem and its complexity, *Comm. ACM*, **15**, № 6, 462—466.
Пратт В. (Pratt V. R.)
[1974] The power of negative thinking in multiplying Boolean matrices, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 80—83. См. также *SIAM J. Comput.*, **4**, № 3, (1975), 326—330.
Пратт Т. (Pratt T. W.)
[1975] Programming languages. Design and implementation, Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод готовится к изданию в изд-ве «Мир».)

Пратт, Яо (Pratt V. R., Yao F. F.)
[1973] On lower bounds for computing the *i*th largest element, IEEE 14th Annual Symposium on Switching and Automata Theory, 70—81.
Прим (Prim R. C.)
[1957] Shortest connection networks and some generalizations, *Bell System Tehn. J.*, 1389—1401.
Рабин (Rabin M. O.)
[1963] Real-time computation, *Isr. J. Math.*, **1**, № 4, 203—211. (Русский перевод в сб. «Проблемы математической логики», М., Мир, 1970, стр. 156—167.)
[1972] Proving simultaneous positivity of linear forms, *J. Comput. and Syst. Sci.*, **6**, № 6, 639—650.
Рабин, Виноград (Rabin M. O., Winograd S.)
[1971] Fast evaluation of polynomials by rational preparation, IBM. Techn. Rept. RC 3645, Yorktown Heights, N.Y. (См. также *Математика*, **18**: 4, (1974), 98—120.)
Рабин, Скотт (Rabin M. O., Scott D.)
[1959] Finite automata and their decision problems, *IBM J. Res. and Devel.*, **3**, № 2, 114—125. (Русский перевод в Кибернетическом сборнике, вып. 4, М., ИЛ, 1962, стр. 56—91.)
Рабинер, Рейдер (ред.) (Rabiner L. R., Rader C. M.)
[1972] Digital signal processing, IEEE Press, New York.
Рабинер, Шэфер, Рейдер (Rabiner L. R., Schafer R. W., Rader C. M.)
[1969] The chirp *z*-transform and its application, *Bell Syst. Techn. J.*, **48**, № 3, 1249—1292. См. также Рабинер, Рейдер [1972], 322—328.
Раундз (Rounds W. C.)
[1973] Complexity of recognition in intermediate level languages, IEEE 14th Annual Symposium on Switching and Automata Theory, 145—158.
Рейнгольд (Reingold E. M.)
[1972] On the optimality of some set algorithms, *J. Assoc. Comput. Mach.*, **19**, № 4, 649—659.
Рейдер (Rader C. M.)
[1968] Discrete Fourier transform when the number of data points is prime, *Proc. IEEE*, **56**, 1107—1108.
Роджерс (Rogers H. Jr.)
[1967] Theory of recursive functions and effective computability, McGraw-Hill, New York. (Русский перевод: Роджерс Х., Теория рекурсивных функций и эффективная вычислимость, М., Мир, 1972.)
Рунге, Кёниг (Runge C., Kőnig H.)
[1924] Die Grundlehren der mathematischen Wissenschaften, B. 11, Springer, Berlin.
Сайферас, Фишер, Мейер (Seiferas J. J., Fischer M. J., Meyer A. R.)
[1973] Refinements of nondeterministic time and space hierarchies, IEEE 14th Annual Symposium on Switching and Automata Theory, 130—137.
Сахни (Sahni S. K.)
[1972] Some related problems from network flows, game theory and integer programming, IEEE 13th Annual Symposium on Switching and Automata Theory, 130—138.
Сети (Sethi R.)
[1973] Complete register allocation problems, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 182—195.
Синглтон (Singleton R. C.)
[1969] Algorithm 347: an algorithm for sorting with minimal storage, *Comm. ACM*, **12**, № 3, 185—187.
Слисенко А. О.
°[1977] Распознавание предиката вхождения в реальное время, препринт

ЛОМИ Р-7-77, Ленинград.

Слоун (Sloane N. J. A.)
[1973] A handbook of integer sequences, Academic Press, New York.

Спира (Spira P. M.)
[1973] A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log n)$, *SIAM J. Comput.*, **2**, № 1, 28—32.

Спира, Пан (Spira P. M., Pan A.)
[1973] On finding and updating shortest paths and spanning trees, IEEE 14th Annual Symposium on Switching and Automata Theory, 82—84.

Стирнз, Розенкранц (Stearns R. E., Rosenkrantz D. J.)
[1969] Table machine simulation, IEEE 10th Annual Symposium on Switching and Automata Theory, 118—128.

Стокмейер (Stockmeyer L. J.)
[1973] Planar 3-colorability is polynomial complete, *SIGACT News*, **5**, № 3, 19—25.

Стокмейер, Мейер (Stockmeyer L. J., Meyer A. R.)
[1973] Word problems requiring exponential time, Proc. 5th Annual ACM Symposium on Theory of Computing, 1—9.

Стоун (Stone H. S.)
[1972] Introduction to computer organization and data structures, McGraw-Hill, New York.

Сэвич (Savitch W. J.)
[1970] Relationship between nondeterministic and deterministic tape complexities, *J. Comput. and Syst. Sci.*, **4**, № 2, 177—192.
[1971] Maze recognizing automata, Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, Colorado, 151—156.

Тарьян (Tarjan R. E.)
[1972] Depth first search and linear graph algorithms, *SIAM J. Comput.* **1**, № 2, 146—160.
[1973a] Finding dominators in directed graphs, Proc. 7th Anual Princeton Conference on Information Sciences and Systems, 414—418.
[1973б] Testing flow graph reducibility, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 96—107.
[1975] On the efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.*, **22**, № 2, 215—224.
°[1977] Reference machines require non-linear time to maintain disjoint sets, Proc. 9th Annual ACM Symposium on Theory of Computing, Boulder, Colorado, 18—29.

Томпсон (Thompson K.)
[1968] Regular expression search algorithm, *Comm. ACM*, **11**, № 6, 419—422.

Трауб (ред.) (Traub J. F.)
[1973] Complexity of sequential and parallel numerical algorithms, Academic Press, New York.

Трахтенброт Б. А.
°[1956] Сигнализирующие функции и табличные операторы, Уч. записки Пензенского гос. пед. ин-та, IV, 75—87.

Тьюринг (Turing A. M.)
[1936] On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.*, ser. 2, **42**, 230—265. Corrections, *там же* **43**, 1937, 544—546.

Уильямс (Williams J. W. J.)
[1964] Algorithm 232: Heapsort, *Comm. ACM*, **7**, № 6, 347—348.

Ульман (Ullman J. D.)
[1973] Polynomial complete scheduling problems, Proc. 4th Symposium on Operating System Principles, 96—101.
[1974] Fast algorithms for the elimination of common subexpressions, *Acta Inform.*, **2**, № 3, 191—213.

Уоршолл (Warshall S.)
[1962] A theorem on Boolean matrices, *J. Assoc. Comput. Mach.*, 9, № 1, 11—12.
Фидуччиа (Fiduccia C. M.)
[1971] Fast matrix multiplication, Proc. 3rd Annual ACM Symposium on Theory of Computing, Shaker Hights, Ohio, 45—49.
[1972] Polynomial evaluation via the division algorithm—the fast Fourier transform revisited, Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, Colorado, 88—93.
Фишер (Fischer M. J.)
[1972] Efficiency of equivalence algorithms, см. Миллер, Тэчер [1972], 153—168.
Фишер, Мейер (Fischer M. J., Meyer A. R.)
[1971] Boolean matrix multiplication and transitive closure, IEEE 12th Annual Symposium on Switching and Automata Theory, 129—131.
Фишер, Патерсон (Fischer M. J., Paterson M. S.)
[1974] String-matching and other products, Complexity Comput. (SIAM—AMS Proc., Vol. 7), Providence, R. I. 113—125.
Фишер, Рабин (Fischer M. J., Rabin M. O.)
[1974] Super-exponential complexity of Presburger arithmetic, Complexity Comput. (SIAM-AMS Proc., Vol. 7), Providence, R. I., 27—42.
Флойд (Floyd R. W.)
[1962] Algorithm 97: shortest path, *Comm. ACM*, 5, № 6, 345.
[1964] Algorithm 245: treesort 3, *Comm. ACM*, 7, № 12, 701.
Флойд, Ривест (Floyd R. W., Rivest R. L.)
[1973] Expected time bounds for selection, Computer Sci. Dept., Stanford University.
Форд, Джонсон (Ford L. R., Johnson S. M.)
[1959] A tournament problem, *Amer. Math. Monthly*, 66, 387—389.
Фрэзер, Мак-Келлар (Frazer W. D., McKellar A. C.)
[1970] Samplesort: a smapling approach to minimal storage tree sorting, *J. Assoc. Comput. Mach.*, 17, № 3, 496—507.
Фурман М. Е.
[1970] О применении метода быстрого перемножения матриц в задаче нахождения транзитивного замыкания графа, *Доклады АН СССР*, 194, № 3, 524.
Хант (Hunt H. B. III)
[1973a] On the time and tape complexity of languages, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 10—19. См. также TR-73-182, Dept. Comput. Sci., Cornell University, Ithaca, N. Y.
[1973б] The equivalence problem for regular expressions with intersection is not polynomial in tape, TR 73-156, Dept. Comput. Sci., Cornell University, Ithaca, N. Y.
[1974] Stack languages, Rice's theorem and a natural exponential complexity gap, TR-142, Comput. Sci. Lab., Dept. Electr. Engineering, Princeton University, Princeton, N. J.
Хант, Розенкранц (Hunt H. B. III, Rosenkrantz D. J.)
[1974] Computational parallels between the regular and context-free languages, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 64—74.
Харари (Harary F.)
[1969] Graph theory, Addison-Wesley, Reding Mass. (Русский перевод: Харари Ф., Теория графов, М., Мир, 1973.)
Харпер, Сэвидж (Harper L. H., Savage J. E.)
[1972] On the complexity of the marriage problem, *Advan. Math.*, 9, № 3, 299—312.
Хартманис (Hartmanis J.)
[1971] Computational complexity of random access stored program machines, *Math. Syst. Theory.*, 5, № 3, 232—245.

Хартманис, Льюис, Стирнз (Hartmanis J., Lewis P. M. II, Stearns R. E.)
[1965] Classification of computations by time and memory requirements, Proc. IFIP Congress 65, Spartan, N. Y., 31—35.
Хартманис, Стирнз (Hartmanis J., Stearns R. E.)
[1965] On computational complexity of algorithms, *Trans. Amer. Math. Soc.*, **117**, 285—306. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 4, М., Мир, 1967, стр. 57—85.)
Хартманис, Хопкрофт (Hartmanis J., Hopcroft J. E.)
[1971] An overview of the theory of computational complexity, *J. Assoc. Comput. Mach.*, **18**, № 3, 444—475. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 11, М., Мир, 1974, стр. 131—176.)
Хейндел, Хоровиц (Heindel L. E., Horowitz E.)
[1971] On decreasing the computing time for modular arithmetic, IEEE 12th Annual Symposium on Switching and Automata Theory, 126—128.
Хект (Hecht M. S.)
[1973] Global data flow analysis of computer programs, Ph. D. Thesis, Dept. Electr. Engineering, Princeton University.
Хенни, Стирнз (Hennie F. C., Stearns R. E.)
[1966] Two tape simulation of multitape machines, *J. Assoc. Comput. Mach.*, **13**, № 4, 533—546. (Русский перевод в сб. «Проблемы математической логики», М., Мир, 1970, стр. 194—211.)
Хиршберг (Hirschberg D. S.)
[1973] A linear space algorithm for computing maximal common subsequences, TR-138, Comput. Sci. Lab., Dept. Electr. Engineering, Princeton University, Princeton, N. J. См. также *Comm. ACM*, **18**, № 6 (1975), 341—343.
Холл (Hall A. D.)
[1971] The ALTRAN system for rational manipulation—a survey, *Comm. ACM*, **14**, № 8, 517—521.
Хон (Hohn F. E.)
[1958] Elementary matrix algebra, Macmillan, New York.
Хоор (Hoare C. A. R.)
[1962] Quicksort, *Comput. J.*, **5**, № 1, 10—15.
Хопкрофт (Hopcroft J. E.)
[1971] An $n \log n$ algorithm for minimizing states in a finite automata, см. Кохави, Пац [1971], 189—196 (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 11, М., Мир, 1974, стр. 177—184.)
Хопкрофт, Карп (Hopcroft J. E., Karp R. M.)
[1971] An algorithm for testing the equivalence of finite automata, TR-71-114, Dept. Comput. Sci., Cornell University, Ithaca, N.Y.
Хопкрофт, Керр (Hopcroft J. E., Kerr L. R.)
[1971] On minimizing the number of multiplications necessary for matrix multiplication, *SIAM J. Appl. Math.*, **20**, № 1, 30—36.
Хопкрофт, Мусински (Hopcroft J. E., Musinski J.)
[1973] Duality in determining the complexity of noncommutative matrix multiplication, Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, 73—87.
Хопкрофт, Тарьян (Hopcroft J. E., Tarjan R. E.)
[1973а] Efficient planarity testing, TR-73-165, Dept. Comput. Sci., Cornell University, Ithaca, N. Y. См. также *J. Assoc. Comput. Mach.* **21**, № 4 (1974).
[1973б] Dividing a graph into triconnected components, *SIAM J. Comput.*, **2**, № 3, 135—157.
[1973в] Efficient algorithms for graph manipulation, *Comm. ACM*, **16**, № 6, 372—378.
Хопкрофт, Ульман (Hopcroft J. E., Ullman J. D.)
[1969] Formal languages and their relation to automata, Addison-Wesley, Reading, Mass.
[1973] Set merging algorithms, *SIAM J. Comput.*, **2**, № 4, 294—303.

Хопкрофт, Уонг (Hopcroft J. E., Wong J. K.)
[1974] A linear time algorithm for isomorphism of planar graphs, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 172—184.
Хорват (Horvath E. C.)
[1974] Some efficient stable sorting algorithms, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 194—215.
Хоровиц (Horowitz E.)
[1972] A fast method for interpolation using preconditioning, *Inform. Process. Lett.*, **1**, № 4, 157—163.
Ху (Hu T. C.)
[1968] A decomposition algorithm for shortest paths in a network., *Operat. Res.*, **16**, 91—102.
Ху, Таккер (Hu T. C., Tucker A. C.)
[1971] Optimum binary search trees, *SIAM J. Appl. Math.*, **21**, № 4, 514—532.
Хэдиан, Соубел (Hadian A., Sobel M.)
[1969] Selecting the tth largest using binary errorless comparisons, Tech. Rept. 121, Dept. of Statistics, University of Minnesota, Minneapolis.
Цейтин Г. С.
°[1968] О сложности вывода в исчислении высказываний, Записки научн. семинаров Ленингр. отд. Матем. ин-та АН СССР, т. 8, 234—259.
Шёнхаге (Schönhage A.)
[1971] Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Inform.*, **1**, 139—144.
Шёнхаге, Штрассен (Schönhage A., Strassen V.)
[1971] Schnelle Multiplikation grosser Zahlen, *Computing*, **7**, №3—4, 281—292. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 10, М., Мир, 1973, стр. 87—98.)
Шепердсон, Стерджис (Shepherdson J. C., Sturgis H. E.)
[1963] Computability of recursive functions, *J. Assoc. Comput. Mach.*, **10**, № 2, 217—255.
Штрассен (Strassen V.)
[1969] Gaussian elimination is not optimal, *Numer. Math.*, **13**, № 4, 354—356. (Русский перевод в Кибернетическом сборнике, нов. сер., вып. 7, М., Мир, 1970, стр. 67—70.)
°[1973] Vermeidung von Divisionen, *J. Reine Angew. Math.*, **264**, 184—202.
[1974] Polynomials with rational coefficients which are hard to compute, *SIAM J. Comput.*, **3**, № 2, 128—149.
°[1976] Computational complexity over finite fields, *SIAM J. Comput.* **5**, № 2, 234—331.
Элгот, Робинсон (Elgot C. C., Robinson A.)
[1964] Random access stored program machines, *J. Assoc. Comput. Mach.*, **11**, № 4, 365—399.
Эренфойхт, Цайгер (Ehrenfeucht A., Zeiger H. P.)
[1974] Complexity measures for regular expressions, Proc. 6th Annual ACM Symposium on Theory of Computing, Seattle, Washington, 75—79.
Янгер (Younger D. H.)
[1967] Recognition of context-free languages in time n^3, *Inform. and Contr.*, **10**, № 2, 189—208. (Русский перевод в сб. «Проблемы математической логики», М., Мир, 1970, стр. 344—362.)

ГЛОССАРИЙ

ADD — сложить (команда сложения)
and — и (конъюнкция)
begin — начало
b — сокращение от blank — пустой (символ, обозначающий пустую клетку ленты)
CHOICE — выбор
comment — комментарий
DIV — сокращение от divide — разделить (команда деления)
do — делать
else — иначе, в противном случае
end — конец
exclusive or — исключающее или (разделительная дизъюнкция, или сложение по модулю два)
false — ложь, ложный
for — для
goto — то же, что **go to** — перейти к
HALT — остановиться (команда остановки)
if ... then — если ... то
in — в
JGTZ — сокращение от jump on greater than zero — перейти (к указанной команде), если (содержимое сумматора) больше нуля.
JUMP — перейти (безусловный переход)
JZERO — сокращение от jump on zero (разъясняется аналогично JGTZ, но переход при равенстве содержимого сумматора нулю)
L — сокращение от left — влево (сдвиг головки влево)
LOAD — загрузить (вызов в сумматор)
MULT — сокращение от multiply — умножить (команда умножения)
not — не (отрицание)
otherwise — в противном случае

pop — вытолкнуть (из стека)
print — напечатать (на выходной ленте)
procedure — процедура
push — затолкнуть (в стек)
or — или (дизъюнкция)
R — сокращение от right — вправо (сдвиг головки вправо)
read — прочитать (со входа)
rev — сокращение от reversed — обращенный
repeat — повторить
return — выдать (результат)
S — сокращение от stationary — неподвижный (головка остается на месте)
SPACE — место (занимаемое чем-либо), объем или размер памяти машины
step — шаг (изменения параметра цикла)
STORE — поместить (команда запоминания)
SUB — сокращение от subtract — вычесть (команда вычитания)
then — см. **if ... then**
TIME — время
true — истина, истинный
until — вплоть до
while — до тех пор, пока
wlg — сокращение от without loss of generality — без потери общности
write — записать (на выходную ленту)

ИМЕННОЙ УКАЗАТЕЛЬ

Аандераа (Aanderaa S. O.) 310
Адельсон-Вельский Г. М. 193, 196
Аппель (Appel K.) 449
Арлазаров В. Л. 283
Ахо (Aho A. V.) 195, 196, 225, 239, 254, 353, 402, 403, 419

Банч (Bunch J.) 283
Барти (Bartee T.) 283
Белага Э. Г. 501
Беллман (Bellman R. E.) 92
Берж (Berge C.) 92, 254
Беркхард (Burkhard W. A.) 92
Биркгоф (Birkhoff G.) 283
Блаттнер (Blattner W. O.) 254, 450
Блюм (Blum M.) 56, 127
Блюстейн (Bluestein L. J.) 353
Бородин (Borodin A. B.) 56, 353, 501
Браун (Brown W. S.) 353
Бруно (Bruno J. L.) 450
Бук (Book R. V.) 450, 474

Вагнер (Wagner R. A.) 92, 253, 403
Вайнер (Weiner P.) 403
Валиант (Valiant L. G.) 283
Вари (Vari T. M.) 353
Виноград (Winograd S.) 92, 283, 501

Галлер (Galler B. A.) 196
Гарсиа (Garsia A.) 501
Гейл (Gale D.) 127
Гилберт (Gilbert E. N.) 196
Годбоул (Godbole S. S.) 92
Грасселли (Grasselli A.) 450
Грей (Gray J. N.) 403
Грэхем (Graham R. L.) 127
Гуд (Good I. J.) 310
Гэри (Garey M. R.) 254, 450

Даниелсон (Danielson G. C.) 310
Данциг (Dantzig G. B.) 254, 450
Дейкстра (Dijkstra E. W.) 254
Джентльмен (Gentleman W. M.) 310, 353
Джонсон Д. Б. (Johnson D. B.) 253, 254
Джонсон Д. С. (Johnson D. S.) 450
Джонсон С. К. (Johnson S. C.) 353
Джонсон С. М. (Johnson S. M.) 126
Джоунс (Jones N. D.) 450
Диветти (Divetti L.) 450
Диниц Е. А. 283
Дэвис (Davis M.) 19

Зивекинг (Sieveking M.) 353

Ибарра (Ibarra O. H.) 403, 450, 474
Ив (Eve J.) 501
Ивен (Even S.) 254, 450

Карацуба А. А. 92
Карп (Karp R. M.) 127, 196, 353, 403, 450
Касами (Kasami T.) 92
Кедем (Kedem Z.) 501
Кёниг (Kōnig H.) 310

Керр (Kerr L. R.) 253, 283, 479, 510
Киркпатрик (Kirkpatrick D.) 501
Кислицын С. С. 127
Клини (Kleene S. C.) 254, 403
Кнут (Knuth D. E.) 92, 127, 195, 196, 353, 403, 501
Кок (Cocke J.) 92
Коллинз (Collins G. E.) 353
Констейбл (Constable R. L.) 474
Крейн (Crane C. A.) 196
Кронрод М. А. 283
Крускал (Kruskal J. B., Jr.) 254
Кук (Cook S. A.) 56, 353, 403, 450, 473, 474, 501
Кукк (Kuck D. J.) 92
Кули (Cooley J. M.) 310
Кунг (Kung H. T.) 353

Ландис Е. М. 193, 196
Ланцош (Lanczos C.) 310
Липсон (Lipson J.) 353
Льюис П. А. (Lewis P. A.) 310
Льюис П. М. (Lewis P. M. II) 56, 196, 474
Лю (Liu C. L.) 92, 127

Мак-Илрой (McIlroy M. D.) 196
Мак-Келлар (McKellar A. C.) 127
Мак-Лейн (MacLane S.) 283
Мак-Нотон (McNaughton R.) 254
Матиясевич Ю. В. 403
Мейер (Meyer A. R.) 254, 283, 310, 450, 473, 474
Миллер (Miller R. E.) 403
Минский (Minsky M.) 56
Моенк (Moenck R.) 353
Моргенштерн (Morgenstern J.) 310, 501
Моррис Дж. (Morris J. H.) 403
Моррис Р. (Morris R.) 195, 196
Моцкин (Motzkin T. S.) 501
Мунро (Munro I.) 254, 353
Мур (Moore E. F.) 196
Мураока (Muraoka Y.) 92
Мусинский (Musinski J.) 501

Нечипорук Э. И. 501
Нивергельт (Nievergelt J.) 196
Николсон (Nicholson P. J.) 310
Островский (Ostrowski A. M.) 501
Офман Ю. П. 92

Пан А. (Pan A.) 253, 254
Пан В. Я. 501
Патерсон (Paterson M. S.) 196, 310, 403
Поль (Pohl I.) 92
Пратт В. (Pratt V. R.) 126, 127, 403, 501
Пратт Т. (Pratt T. W.) 92
Прим (Prim R. C.) 254

Рабин (Rabin M. O.) 56, 196, 253, 403, 473, 474, 501
Рабинер (Rabiner L. R.) 310, 353
Рао (Rao M. R.) 254, 450
Раундз (Rounds W. C.) 450, 474
Рейдер (Rader C. M.) 310, 353
Рейнгольд (Reingold E. M.) 196
Рекхау (Reckhow R. A.) 56, 473, 474
Ривест (Rivest R. L.) 127
Робинсон (Robinson A.) 56
Роджерс (Rogers H., Jr.) 19
Розенберг (Rosenberg A. L.) 403
Розенкранц (Rosenkrantz D. J.) 196, 450, 474
Рунге (Runge C.) 310

Сайферас (Seiferas J. J.) 473,
Санде (Sande G.) 310
Сахни (Sahni S.K.) 450, 474
Сети (Sethi R.) 450
Синглтон (Singleton R. C.) 127
Скотт (Scott D.) 403
Слисенко А. О. 403
Слоун (Sloane N. J. A.) 92
Соубел (Sobel M.) 127
Спира (Spira P. M.) 253, 254
Стейглиц (Steiglitz K.) 353
Стерджис (Sturgis H. E.) 56
Стирнз (Stearns R. E.) 56, 196, 474
Стокмейер (Stockmeyer L.) 450, 474
Стоун (Stone H. S.) 92
Сэвидж (Savage J. E.) 501
Сэвич (Savitch W. J.) 450

Таккер (Tucker A. C.) 196
Тарьян (Tarjan R. E.) 127, 196, 253, 254

Томпсон (Thompson K.) 403
Трахтенброт Б. А. 56
Триттер (Tritter A.) 196
Тьюки (Tukey J. W.) 310
Тьюринг (Turing A. M.) 56

Уелч (Welch P. D.) 310
Уильямс (Williams J. W. J.) 127
Ульман (Ullman J. D.) 20, 56, 195, 196, 225, 239, 254, 353, 402, 403, 419, 449, 450
Унгар (Ungar P.) 501
Уоршол (Warshall S.) 254

Фараджев И. А. 283
Фидуччиа (Fiduccia C. M.) 310, 501
Фишер (Fischer M. J.) 92, 196, 254, 283, 310, 403, 450, 473
Флойд (Floyd R. W.) 127, 254, 501
Форд (Ford L. R.) 126
Фрэйзер (Frazer W. D.) 127
Фурман М. Е. 254

Хакен (Haken W.) 449
Хант (Hunt H. B. III) 450, 473, 474
Харари (Harary F.) 92, 254
Харпер (Harper L. H.) 501
Харрисон (Harrison M. A.) 403
Хартманис (Hartmanis J.) 56, 196, 474
Хейдиан (Hadian A.) 127
Хейндел (Heindel L. E.) 353
Хект (Hecht M. S.) 239
Хенни (Hennie F. C.) 474
Хиршберг (Hirschberg D. S.) 402, 403
Холл (Hall A. D.) 353
Хон (Hohn F. E.) 283
Хоор (Hoare C. A. R.) 127
Хопкрофт (Hopcroft J. E.) 20, 56, 196, 225, 253, 254, 283, 403, 449, 450, 474, 479, 501
Хорват (Horvath E. C.) 127
Хоровиц (Horowitz E.) 353
Ху (Hu T. C.) 196, 254

Цайгер (Zeiger H. P.) 403
Цейтин Г. С. 56, 473

Честер (Chester D.) 403

Шёнхаге (Schönhage A.) 283, 310, 353
Шепердсон (Shepherdson J. C.) 56
Штрассен (Strassen V.) 283, 310, 501
Шэфер (Schafer R. W.) 353

Элгот (Elgot C. C.) 56
Эренфойхт (Ehrenfeucht A.) 403

Ямада (Yamada H.) 254
Янгер (Younger D.H.) 92
Яновская С. А. 56
Яо (Yao F. F.) 126, 127

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Автомат (automaton)
конечный (finite) 165, 361, 365,
— — детерминированный (deterministic) 165, 361
— — недетерминированный (nondeterministic) 356
— линейно ограниченный (linear bounded) 449
— магазинный (pushdown) 373
— — двусторонний (two-way) 373, 400
— — — детерминированный (deterministic) 373—375
— — детерминированный (deterministic) 373, 400, 401
— — недетерминированный (nondeterministic) 400
— — односторонний (one-way) 400
Адрес (address) 16
— возврата (return) 73
— значения (value) 73
— символический (symbolic) 33
Адресация (addressing)
— косвенная (indirect) 16
Активный (active) 484, 490
Алгол (ALGOL)
— Упрощенный (Pidgin) 48
Алгоритм (algorithm)
— Дейкстры (Dijkstra's) 236
— Евклида (Euclid's) 336
— — расширенный (extended) 336, 337
— Крускала (Kruskal's) 199
— префиксный (on-line) 129
— свободный (off-line) 129
— с предварительной обработкой данных (preconditioned) 329
— четырех русских (four Russians') 275, 277
— Шёнхаге — Штрассена (Schönhage — Strassen) 304, 306
— Штрассена (Strassen's) 259
Алфавит (alphabet) 19, 355
— входной (input) 165, 356, 375
— магазинный (pushdown list) 374, 375
Аннулятор (anihilator) 224
Антисимметричность (antisymmetry) 94
Аргумент (argument) *см.* Параметр

База данных (data base) 132, 147, 188
Баланс узла (balance of a vertex) 194

Балансировка (balancing) 81
БЛОК (INBLOCK) 185
Блок (block) 51
БПФ (FFT) *см.* Преобразование Фурье, быстрое
Брат (brother) 174
Быстрсорт (quicksort) 111, 113
БЫСТРСОРТ (QUICKSORT) 113

ВЕРШИНА (TOP) 61
Вес дерева (weight of a tree) 142
ВЗИМОЗАМЕНА (INTERCHANGE) 52
ВНЕШ_ИМЯ (EXTERNAL_NAME) 147
ВНУТПОРЯДОК (INORDER) 71
ВНУТР_ИМЯ (INTERNAL_NAME) 147
В_ОЖИДАНИИ (INWAITING) 186
Восприниматься *см.* Допускаться
ВПИСАТЬ (ENQUEUE) 62, 194
ВРЕМ (TEMP) 380
ВСТАВИТЬ (INSERT) 59, 128
Вход (input) 34
— вычисления (of computation) 477
ВЫБОР (SELECT) 118, 122
Вызов (call)
— по значению (by value) 52
— — наименованию (by name) 52
— — ссылке (by reference) 52
ВЫПИСАТЬ (DEQUEUE) 62 , 194
Выполнимость (satisfiability) 419
Выражение (expression)
— регулярное (regular) 355
— — полурасширенное (semiextended) 457
— — расширенное (extended) 456
Высота (height)
— дерева (of a tree) 68
— узла (of a vertex) 68
ВЫТОЛКНУТЬ (POP) 61
Выход (output) 34
Выход (в графе) (output vertex) 497
Вычисление (computation)
— битовое (bitwise) 35
— двойственное (dual) 496
— линейное (linear) 497
— машины с данным измерителем, правильное (of a machine with a given yardstick, valid) 462
— относительно поля (with respect to a field) 477
Вычисление (значения) полинома (evaluation of a polynomial) 34, 286, 328, 476, 487, 490—491, 499—500

Глубина (depth)
— средняя (expected) 111
— узла (of a vertex) 68
Головка (head) 40
Гомоморфизм (homomorphism) 461
— сохраняющий длину (length-preserving) 461

Грамматика (grammar)
— бесконтекстная (context-free) 91, 401
— — в нормальной форме Хомского (in Chomski normal form) 91
— контекстная (context-sensitive) 449
Граф (graph) 64
— ациклический (acyclic) 67
— двусвязный (biconnected) 206
— дополнительный (complement) 431
— корневой (rooted) 239
— неориентированный (undirected) 64
— ориентированный (directed) 64
— переходов *см.* Диаграмма (переходов)
— раскрашиваемый (colorable) 421
— связный (connected) 70, 216, 253
— сильно связный (strongly connected) 216

ДАННЫЕ (DATA) 99
Дважды связанный (doubly linked) 61
Двойственное (вычисление) (dual) 496
Двусвязность (biconnectivity) 206
ДЕЛЕНИЕ (DIVIDE) 179, 180
Деление (division)
— полиномов (of polynomials) 320
— целых чисел (of integers) 313
Дерево (tree) 67
— 2-3 169
— *I* 386
— *S* 386
— АВЛ (AVL) 193
— бинарное *см.* Дерево двоичное
— вспомогательное (auxiliary) 390
— двоичное (binary) 68
— — полное (complete) 68
— двоичного поиска (binary search) 136
— доминаторное (dominator) 239
— корневое (rooted) 67
— — неориентированное (undirected) 70
— неориентированное (undirected) 70
— ориентированное (directed) 67
— остовное (spanning) 130
— — глубинное (depth-first) 203
— позиций *см.* Дерево позиционное
— позиционное (position) 387, 296
— — уплотненное (compact) 397
— помеченное (labeled) 103
— решений (decision) 37
— сбалансированное (balanced) 169, 194
— — ограниченное (bounded) 194
— сливаемое *см.* Сливаемое дерево
— сортирующее *см.* Сортирующее дерево
— упорядоченное (ordered) 68
Диаграмма (переходов) (diagram) 357
Диагонализировать (diagonalize) 453
Диагональ главная (main diagonal) 259

ДКА (DFA) 361
— скелетный (skeletal) 367
ДЛИНА (LENGTH) 99
Длина (length)
— внешних путей (external path) 194
— внутренних путей (internal path) 194
— пути (of a path) 64
— регулярного выражения (of a regular expression) 359
— цепочки (of a string) 355
1ДМА (1DPDA) 401
2ДМА (2DPDA) 373
— k-головчатый (k-head) 400
ДОБАВСЫНА (ADDSON) 173
Доминатор (dominator) 239
— непосредственный (immediate) 239
Допускаться (be accepted)
— конечным автоматом (by a finite automaton) 166, 357
— магазинным автоматом (by a pushdown store automaton) 376
— машиной Тьюринга (by a Turing machine) 41, 42, 406
— РАМ (by a RAM) 19

Зависимость линейная по модулю (linear dependence modulo) 480
Задача (problem)
— легко разрешимая (tractable) 404
— об упаковке (package placement) 447
— — устойчивом бракосочетании (stable marriage) 87
— о коммивояжере (travelling salesman) 447
— — кратчайшем пути (shortest path) 223
— — потоке (flow) 447—448
— — разбиении (partition) 447
— — ранце (knapsack) 446
— — расписании работ (scheduling) 448
— определения глубины (depth determination) 164
— пустоты дополнения (emptiness of complement) 457
— сортировки (sorting) 94
— трудно разрешимая (intractable) 404
ЗАДНИЙ (REAR) 62
Замыкание (closure) 225
— Клини (Kleene) 355
— рефлексивное и транзитивное (reflexive and transitive) 223
— транзитивное (transitive) 120, 223
ЗАТОЛКНУТЬ (PUSH) 61
Значение переменной (в вычислении) (valuation of a variable) 477
Значение операнда (the value of operand) 17

Идемпотентность (idempotence) 224
Идентификатор (identifier) 386
Идентифицировать (позицию) (identify) 386
Идентифицироваться (match) 399
Иерархия (hierarchy)
— временная *см.* Иерархия по времени
— емкостная *см.* Иерархия по емкости
— по времени (time) 471
— — емкости (space) 452
— — памяти *см.* Иерархия по емкости

ИЗВЛЕЧЬ_MIN (EXTRACT_MIN) 162, 191
Измеритель (yardstick)
Изоморфизм (isomorphism)
— деревьев (of trees) 102
— подграфу (subgraph) 446
ИМПЛАНТАЦИЯ (IMPLANT) 176, 177
ИМЯ (NAME) 58, 153
Имя переменной (в вычислении) (variable name) 477
Индекс (index) 50
Интерполяция (interpolation) 286
Исток (в графе) (source vertice) 497
Источник (source) 235
Итерация *см.* Замыкание Клини
— позитивная (positive) 355

Китайская теорема об остатках (Chinese remaindering) 329
Класс эквивалентности (equivalence class) 206
Клетка (cell) 40
Клика (clique) 418
КНФ (CNF) 427
Код операции (operation code) 16
Кольцо (ring) 256
— коммутативное (commutative) 256
КОНЕЦ (HEAD) 66
Конец ребра (head of the edge) 64
Конец составного ребра (head of a composite edge) 242
Конец цепочки (suffix of a string) 355
Конкатенация (concatenation)
— множеств (of sets) 225
— списков (of lists) 50
— цепочек (of strings) 355
— языков (of languages) 355
Конфигурация (configuration) 378
— C' выводима из C (C derives C') 379
— поверхностная (surface) 378
— терминальная (terminal) 379
Команда (instruction)
— РАМ (of a RAM) 16—18
— РАСП (of a RASP) 26
КОМПОНЕНТА (ELEMENT) 58
Компонента (component)
— двусвязная (biconnected) 206
— связная (connected) 202
— сильно связная (strongly connected) 216
КОРЕНЬ (ROOT) 153
Корень (root)
— графа (of a graph) 67, 239
— дерева (of a tree) 67
— из единицы (of unity) 285
— — — примитивный (principal) 285
— сильно связной компоненты (of a strongly connected component) 216
КОРРЕКТИР (UPDATE) 380, 382
Критерий весовой (cost criterion)
— логарифмический (logarithmic) 23, 24
— равномерный (uniform) 23

Левое двойственное (множество выражений) (left dual) 496
ЛЕВЫЙСЫН (LEFTSON) 68
Легко разрешимый (tractable) 404
Лента (tape) 39
— входная (input) 15, 41, 165, 374
— выходная (output) 15, 44
Лес (forest) 67
— остовный (spanning) 130
— — глубинный (depth-first) 203
— — построенный поиском в глубину *см.* Лес остовный глубинный
Лист (leaf) 67
Литерал 16
Литерал (literal) 417, 427

Магазин (pushdown store) 61
Маркер (marker)
— дна (bottom) 375
— нижний *см.* Маркер дна
Массив (array) 58
Матрица (matrix)
— булева (Boolean) 274
— единичная (identity) 256
— невырожденная (nonsingular) 258
— нормированная (unit) 259
— обратная (inverse) 257
— перестановки (permutation) 259
— положительно определенная (positive definite) 282
— смежностей (adjacency matrix) 64
— тёплицева (Toeplitz) 282
— транспонированная (transpose) 259
— треугольная (triangular)
— — верхняя (upper) 258
— — нижняя (lower) 258
Машина (machine)
— адресная 31
— равнодоступная адресная *см.* Машина с произвольным доступом к памяти
— — — с хранимой программой *см.* Машина с произвольным доступом к памяти и хранимой программой
— с произвольным доступом к памяти (random access)
— — — — — — и хранимой программой (stored program) 26
— Тьюринга (Turing) 39
— — многоленточная (multitape) 39
— — недетерминированная (nondeterministic) 405, 406
Мгновенное описание (instantaneous description) 42, 356, 375
— — допускающее (accepting) 356
— — — для 2ДМА (of a 2DPDA) 376
— — 2ДМА (of a 2DPDA) 375
— — МТ (of a TM) 42, 406
— — НКА (of a NDFA) 356
— — НМТ (of a NDTM) 406
— — начальное (initial) 42, 356, 376, 406
— — — 2ДМА (of a 2DPDA) 376
— — — НКА (of a NDFA) 356
— — — НМТ (of a NDTM) 406
Метка пути (path label) 225
Метод расстановки (hashing) 132, 196

Минимизация конечного автомата (minimization of a finite automaton) 187
Множество (set)
— безопасное для разбиения (safe for a partition) 183
— пустое (empty) 355
— ребер, разрезающих циклы (feedback edge) 421
— регулярное (regular) 355
— узлов, разрезающих циклы (feedback vertex) 421
— универсальное (universal) *см.* База данных
МНОЖИТЕЛЬ (FACTOR) 264, 267
Моноид (monoid) 224
Мощность (cardinality) 64
Мультиграф (multigraph) 249
МО (ID) — *см.* Мгновенное описание

Наибольший общий делитель (greatest common divisor) 336, 339
Наименьшее общее кратное (least common multiple) 352
НАИМЕНЬШИЙ (SMALLEST) 175
НАЙТИ (FIND) 128
НАЙТИ_ГЛУБИНУ (FIND_DEPTH) 164
НАЙТИ_ПУТЬ (FIND_PATH) 249
Начало отсчета (origin) 164
Начало ребра (tail of the edge) 64
— — составного (composite) 242
Начало цепочки (prefix of a string) 355
НВП-разложение (LUP decomposition) 264
НВ-разложение (LU decomposition) 264
Независимость линейная по модулю (linear independence modulo) 480
НЕПУСТОЙ (NONEMPTY) 99
Ниже (отношение на поверхностных конфигурациях) (below) 380
НИЖНИЙ (LOW) 210
НИЖНЯЯСВЯЗЬ (LOWLINK) 217
НКА (NDFA) 356
НМА (NPDA) 400
1НМА (1NPDA) 401
НОВ (NEW) 380
НОД (GCD) 336, 344
НОК (LCM) 352
НОМЕР (NUMBER) 71
O (порядок величины — order of magnitude) 12
O_A (порядок величины для неветвящихся программ) 35
$O_Б$ (порядок величины при битовых вычислениях) 35
$O_{ДВ}$ (порядок величины при применении модели с двоичными векторами) 37
$O_{МТ}$ (порядок величины при использовании в качестве модели машины Тьюринга) 44
$O_С$ (порядок величины при использовании модели деревьев решений) 38
Область действия переменной (the scope of a variable) 48
Обозревать (scan) 40
Обработка предварительная (preconditioning) 490, 491
Образ (pattern) 363
ОБРАТНОЕ (RECIPROCAL) 314, 315
ОБРАТНЫЙ (RECIPROCAL) 321
ОБЪЕДИНИТЬ (UNION) 128, 148
Операнд (operand) 16
Оператор (statement)
— COMMENT 52
— FOR 49

— GOTO 51
— IF 49
— READ 52
— REPEAT 49
— WHILE 49
— WRITE 52
— определения процедур (procedure-difinition) 51
— помеченный (labeled) 50
— присваивания (assignment) 49
Операция (operation)
— активная мультипликативная (active multiplicative) 490
— битовая (bit) 35
— ассоциативная (associative) 224
— дистрибутивная (distributive) 224
— коммутативная (commutative) 224
— с двоичными векторами (bit vector) 37
— элементарная (scalar) 230
Определитель (determinant) 258
Описание мгновенное *см.* Мгновенное описание
ОТЕЦ (FATHER) 153
Отец (father) 67
Отображение памяти (memory map) 16, 26
ОЧЕРЕДЬ (QUEUE) 96
Очередь (queue) 62
— сцепляемая (concatenable) 170
— с приоритетами (priority) 170

Палиндром (palindrome) 42
Пара лежит ниже (pair is below) 380
Параметр (parameter)
— фактический (actual) 51
— формальный (formal) 51
Паросочетание (pairing) 87
— устойчивое (stable) 87
ПЕРЕДНИЙ (FRONT) 62
Переменная (variable) *см.* Адрес символический
Переменная (вычисления) (variable) 477
— входная (input) 33
— выходная (output) 33
— глобальная (global) 52
— локальная (local) 52, 53
Переменная формальная (indeterminate) 476
ПЕРЕСЕЧЕНИЕ (INTERSECTION) 186
Перестановка (permutation)
— нечетная (odd) 258
— четная (even) 258
ПЕРЕСЫПКА (HEAPIFY) 108
ПНОД (HGCD) 339, 340
Подграф полный (complete subgraph) *см.* Клика
Поддерево (subtree) 67
— левое (left) 68
— правое (right) 68
Подматрица (submatrix) 259
— главная (principal) 259
Подпоследовательность (subsequence) 402
Подцепочка (substring) 355

ПОЗИЦИЯ (POSITION) 59
Позиция (в цепочке) (position) 386
ПОИСК (SEARCH) 135, 138, 173, 203
Поиск (search)
— в глубину (depth-first) 202
— двоичный (binary) 135
ПОИСКБ (SEARCHB) 212
ПОИСКВ (SEARCHC) 220
Покрытие (cover)
— множествами (set) 421—422
— точное (exact) 422
— узельное (vertex) 421
Поле (field) 256, 475
Полином (polynomial)
— плотный (dense) 348
— разреженный (sparse) 348
Полиномиально (polynomially)
— связанные (related) 39
— трансформируемый (transformable) 416
— эквивалентные *см.* Полиномиально связанные
Полнота (completeness)
— NP (NP-completeness) 416
— для $_{\mathcal{O}}\mathcal{NP}$-TIME (for $_{\mathcal{O}}\mathcal{NP}$-TIME) 416
— — $\mathcal{P}$-SPACE (for $\mathcal{P}$-SPACE) 440
Полукольцо замкнутое (closed semiring) 223
Полустепень исхода (out-degree) 64
Порядок (order)
— внутренний (in-) 68
— лексикографический (lexicographic) 95
— линейный (linear) 94
— обратный (post-) 68
— полный (total) *см.* Порядок линейный
— прямой (pre-) 68
— частичный (partial) 94
Последовательность остатков (remainder sequence) 336
Постоянная (вычисления) (constant) 477
ПОСТРДЕРЕВА (BUILDTREE) 144
Построение сортирующего дерева (construction of a heap) 108
ПОСТРСОРТДЕРЕВА (BUILDHEAP) 109
Потомок (descendant) 67
— подлинный (proper) 67
Правило Горнера (Horner's rule) 34
ПРАВЫЙСЫН (RIGHTSON) 68
ПРИНАДЛЕЖАТЬ (MEMBER) 128
ПРЕД (PRED) 163, 380
Предок (ancestor) 67
— подлинный (proper) 67
Предшественница (predecessor) 380
— непосредственная (immediate) 380
ПРЕДЫДУЩАЯ (PREVIOUS) 61
Преобразование Фурье (Fourier transform)
— быстрое (fast) 294
— дискретное (discrete) 285
— обратное (inverse) 286
Префикс (prefix) 355
Префиксный (режим, алгоритм) (on-line) 129
ПРОВЕРКА (TEST) 412

Программа (program)
— для РАМ (for RAM) 16
— на Упрощенном Алголе (Pidgin ALGOL) 48
— неветвящаяся (straight-line) 32
Программирование динамическое (dynamic programming) 83
Продукция (production) 91
Прохождение дерева (traversal of a tree) 68
— — во внутреннем порядке (inorder) 69
— — в обратном порядке (postorder) 69
— — в прямом порядке (preorder) 69
Процедура (procedure) 51
— рекурсивная (recursive) 70
Путь (path) 64
— внешний (external) 194
— внутренний (internal) 194
— простой (simple) 64

Разбиение (partitioning) 181
Разбиение грубейшее (coarsest partition) 181
Разветвление (branching) *см.* **goto**-оператор
Разделяй и властвуй (divide and conquer) 75
РАЗМЕР (SIZE) 147, 312
Разность циклическая (cyclic difference) 309
Разрез (cutset) 447
РАМ (RAM) 15
РАМ-программа (program) 16
— недетерминированная (nondeterministic) 415
Ранг (rank)
— матрицы (of a matrix) 259
— по столбцам (column) 481
— — строкам (row) 481
— узла (of a vertex) 155
РАСП (RASP) 26
РАСП-программа (program) 26
— недетерминированная (nondeterministic) 415
Расстановка (hashing) 132
Расширение поля формальными переменными (extension of a field by indeterminates) 476
РАСЩЕПИТЬ (SPLIT) 129
Ребро (edge) 64
— древесное (tree) 203, 215
— обратное (back) 203, 215
— поперечное (cross) 215
— прямое (forward) 215
— составное (composite) 242
Регистр (register) 15
Редукция транзитивная (transitive reduction) 249
Режим
— префиксный (on-line) 129
— свободный (off-line) 129
Рекурсия (recursion) 70

Свертка (convolution) 287
— отрицательно обернутая (negative wrapped) 289
— положительно обернутая (positive wrapped) 289

СВОБОДНАЯ (FREE) 59
Свободный (режим, алгоритм) (off-line) 129
Сводимый (язык) (reducible) 416
Свойство сортирующего дерева (heap property) 108
СВЯЗАТЬ (LINK) 164
Связность (графа) (connectedness) 253
СВЯЗЬ (LINK) 66
Сеть логическая (logic circuit, network) 35, 55, 498
— — комбинационная (combinational) 55
Сжатие путей (path compression) 152
Символ (symbol)
— входной (input) 40
— ленточный (tape) 40
— на ленте *см.* Символ ленточный
— несущественный (don't care) 399
— нетерминальный (nonterminal) 91
— пустой (the blank) 40
— терминальный (terminal) 91
СЛЕД (SUCC) 163
СЛЕДУЮЩАЯ (NEXT) 58
СЛЕДУЮЩИЙ (NEXT) 66, 147
Сливаемое дерево (mergeable heap) 170
СЛИТЬ (MERGE) 195
СЛИЯНИЕ (MERGE) 82
Словарь (dictionary) 129, 170
Слово (word) *см.* Цепочка
— машинное (computer) 14, 25. *См. также* Число, хранимое в регистре
Сложность (complexity)
— арифметическая (arithmetic) 476
— асимптотическая (asymptotic) *см.* Сложность временна̀я, Сложность емкостная
— временна̀я (time) 22, 27, 44
— — в среднем (expected) 22
— — — худшем случае *см.* Сложность временна̀я
— — НМТ (NDTM) 407
— в худшем случае (worst case) 22
— емкостная (space) 21, 44
— — логарифмическая (logarithmic) 24, 27
— — НМТ (NDTM) 409
— реализации булевой функции (realization of a Boolean function) 498, 499
— средняя (expected) 22
— усредненная *см.* Сложность средняя
Смежный (adjacent) 64
Смещение (displacement) 164
Содержимое регистра (the contents of a register) 16
СОРТ (SORT) 82
СОРТВЗБАЛТЫВАНИЕМ (BUBBLESORT) 123
Сортдеревом (heapsort) 106, 110
Сортировка (sorting) 93
— внешняя (external) 94
— внутренняя (internal) 99
— вставками (insertion) 126
— вычерпыванием (bucket) 95
— лексикографическая (lexicographic) 96, 98
— слиянием (merge) 82
— с помощью сравнений (by comparisons) 104
— топологическая (topological) 87
— цифровая (radix) 95

Сортирующее дерево (heap) 106
СОСТОЯНИЕ (STATUS) 53
Состояние (state) 40
— допускающее (accepting) 41, 166, 356, 375
— заключительное (final) 41. *См. также* Состояние допускающее
— — 2ДМА (of a 2DPDA) 375
— начальное (initial) 40
— начальное (start) 166, 356, 375
— управляющего устройства (of a finite control) 375
СОЧЕТ (COMB) 89
СПИСОК (LIST) 147
Список (list) 58
— дважды связанный (doubly linked) 61
— свободный (free) 60
— смежностей (adjacency) 66
Способность пропускная (capacity) 448, 497
Сравнение (comparison) 38
— ключевое (key) 121
СТ (DEG) 312
Стабильный (метод сортировки) (stable) 126
Статистики порядковые (order statistics) 93, 117
Стек (stack) 61
Степень (degree)
— полинома 312, 491
— полинома от нескольких переменных (of a multivariate polynomial) 491
— узла 64
Стоимость (cost) 223, 225
— пути (of a path) 223
Структура данных (data structure) 58. *См. также* Граф, Дерево, Массив, Очередь, Список, Стек
СУММАСТРОК (ROWSUM) 278
Сумматор (accumulator) 16
Суффикс (suffix) 355
Схема логическая (logic circuit) *см.* Сеть логическая
— сдвигающая (shifting network) 497
Сцепление *см.* Конкатенация
СЧЕТ (COUNT) 71, 153
Счетчик команд (location counter) 16, 26
Сын (son) 67
— *а* 386
— левый (left) 68
— правый (right) 68

Тавтология (tautology) 446
Теорема о свертке (convolution theorem) 288
ТЕРМ (TERM) 380
Терминатор (terminator) 379
Тип данных (data type) 48
Точка сочленения (articulation point) 206
Тройка допустимая (admissable triple) 309
Трудно разрешимый (intractable) 404

УДАЛИТЬ (DELETE) 128
УЗЕЛ (VERTEX) 71

Узел (vertex, node) 64
— смежный (adjacent)
Умножение (multiplication)
— активное (active) 484
— векторов (of vectors) 493
— и/или (and/or) 352, 399
— комплексных чисел (of complex numbers) 478, 479, 489
— матриц (of matrices) 259, 495
— матрицы на вектор (of a matrix by a vector) 486, 494
— полиномов (of polynomials) 480, 487, 492, 493
— целых чисел (of integers) 77—80, 304
Уровень узла (level of a vertex) 68
Устройство управляющее (finite control) 40, 165, 356

Форма (form)
— билинейная (bilinear) 495
— конъюнктивная нормальная (conjunctive normal) 427
— нормальная Хомского (Chomsky normal) 91
Формула булева (Boolean expression) 417
— выполнимая (satisfiable) 419
— Лагранжа интерполяционная (Lagrangian interpolation formula) 329
Фрагмент стека (stack frame) 73
Функция (function) 51
— булева (Boolean) 498
— логарифмическая (logarithmic) 23
— конструируемая по времени (time-constructable) 471
— — — емкости (space) 411
— — — памяти *см.* Функция, конструируемая по емкости
— отказов (failure) 368
— переходов (next-move) 41
— переходов (state transition) 356
— расстановки (hashing) 130
— стоимости (cost) 130
— экспоненциальная (exponential) 39
— элементарная (elementary) 466

Ханойские башни (towers of Hanoi) 88
Характеристический (вектор) (characteristic) 63

Цвет (color) 421
ЦЕПОЧКА (STRING) 99
Цепочка (string) 355
— допускаемая (автоматом, машиной) (accepted) 166, 357, 376
— пустая (empty) 165, 355
— с несущественными символами (with don't cares) 393
Цепочка-текст (text-string) 363
Цепь (chain) 397
ЦИКЛ (CYCLE) 457
Цикл (cycle) 64
— гамильтонов (Hamilton) 421
— — ориентированный (directed) 421
— эйлеров (Euler) 249
Циркулянт (circulant) 310

ЧАСТИЧНО НАЙТИ (partial FIND) 158
Черпак (bucket) 95
ЧИС (NUM) 278
Число (number)
— Каталана (Catalan) 91
— хранимое в регистре (stored in a register) 14, 25, 37
— хроматическое (chromatic) 421
ЧН (PF) 158

Шаг (вычисления) (step) 477
Шаг (работы автомата, машины) (move)
— 2ДМА (by a 2DPDA) 376
— НМТ (by a NDTM) 406

Эквивалентность (equivalence)
— автоматов (of automata) 166
— векторов по модулю (of vectors modulo) 480
— регулярных выражений (of regular expressions) 356
— отношение (relation) 206
— состояний (of states) 166
Элем (Item) 58
ЭЛЕМЕНТ (ELEMENT) 153
ЭЛЕМЕНТ (ITEM) 59
Элемент единичный (identity) 224
— k-й наименьший (the kth smallest element) 117
— обратный (inverse) 256
Элементы смежные матрицы (adjacent entries of a matrix) 252

Язык (language) 19,355
— бесконтекстный (context-free) 401
— допускаемый автоматом (accepted by an automaton) 166, 357
— — 2ДМА (by a 2DPDA) 376
— — МТ (by a TM) 42
— — НМТ (by a NDTM) 407
— — программой (by a program) 19
— контекстный (context-sensitive) 449
NP-полный (NP-complete) 416
— полный для недетерминированного полиномиального времени (for nondeterministic polynomial time) *см.* Язык NP-полный
— порождаемый грамматикой (generated by a grammar) 91
— регулярный (regular) 356

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ К РУССКОМУ ПЕРЕВОДУ 5

ПРЕДИСЛОВИЕ 7

1. МОДЕЛИ ВЫЧИСЛЕНИЙ 11

1.1. Алгоритмы и их сложности 11
1.2. Машины с произвольным доступом к памяти 15
1.3. Вычислительная сложность RAM-программ 22
1.4. Модель с хранимой программой 26
1.5. Модификация RAM 32
1.6. Простейшая модель вычислений: машина Тьюринга 39
1.7. Связь машин Тьюринга и RAM 45
1.8. Язык высокого уровня — Упрощенный Алгол 47
Упражнения 54
Замечания по литературе 56

2. РАЗРАБОТКА ЭФФЕКТИВНЫХ АЛГОРИТМОВ 57

2.1. Структуры данных: списки, очереди и стеки 58
2.2. Представления множеств 63
2.3. Графы 64
2.4. Деревья 67
2.5. Рекурсия 70
2.6. Разделяй и властвуй 75
2.7. Балансировка 81
2.8. Динамическое программирование 83
2.9. Эпилог 85
Упражнения 86
Замечания по литературе 92

3. СОРТИРОВКА И ПОРЯДКОВЫЕ СТАТИСТИКИ 93

3.1. Задача сортировки 94
3.2. Цифровая сортировка 95
3.3. Сортировка с помощью сравнений 104

3.4. Сортдеревом — упорядочение с помощью $O(n \log n)$ сравнений 106
3.5. Быстрсорт — упорядочение за среднее время $O(n \log n)$ 111
3.6. Порядковые статистики 117
3.7. Среднее время для порядковых статистик 119
Упражнения 122
Замечания по литературе 127

4. СТРУКТУРЫ ДАННЫХ ДЛЯ ЗАДАЧ, КАСАЮЩИХСЯ РАБОТЫ С МНОЖЕСТВАМИ 128

4.1. Основные операции над множествами 128
4.2. Метод расстановки 132
4.3. Двоичный поиск 135
4.4. Деревья двоичного поиска 136
4.5. Оптимальные деревья двоичного поиска 141
4.6. Простой алгоритм для нахождения объединения непересекающихся множеств 146
4.7. Древовидные структуры для задачи ОБЪЕДИНИТЬ—НАЙТИ 150
4.8. Приложения и обобщения алгоритма ОБЪЕДИНИТЬ—НАЙТИ 162
4.9. Схемы сбалансированных деревьев 168
4.10. Словари и очереди с приоритетами 171
4.11. Сливаемые деревья 175
4.12. Сцепляемые очереди 178
4.13. Разбиение 181
4.14. Резюме 188
Упражнения 188
Замечания по литературе 195

5. АЛГОРИТМЫ НА ГРАФАХ 197

5.1. Остовное дерево наименьшей стоимости 197
5.2. Метод поиска в глубину 202
5.3. Двусвязность 206
5.4. Поиск в глубину в ориентированном графе 214
5.5. Сильная связность 216
5.6. Задачи нахождения путей 223
5.7. Алгоритм транзитивного замыкания 227
5.8. Алгоритм нахождения кратчайшего пути 229
5.9. Задачи о путях и умножение матриц 230
5.10. Задачи с одним источником 235
5.11. Доминаторы в ориентированных ациклических графах: комбинирование понятий 238
Упражнения 247
Замечания по литературе 254

6. УМНОЖЕНИЕ МАТРИЦ И СВЯЗАННЫЕ С НИМ ОПЕРАЦИИ 255

6.1. Основные понятия 255
6.2. Алгоритм Штрассена для умножения матриц 259
6.3. Обращение матриц 262
6.4. НВП-разложение матрицы 263
6.5. Приложения НВП-разложения 272
6.6. Умножение булевых матриц 274
Упражнения 279
Замечания по литературе 283

7. БЫСТРОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ И ЕГО ПРИЛОЖЕНИЯ 284

7.1. Дискретное преобразование Фурье и обратное к нему 284
7.2. Алгоритм быстрого преобразования Фурье 290
7.3. БПФ при использовании битовых операций 298
7.4. Произведение полиномов 303
7.5. Алгоритм Шёнхаге — Штрассена для умножения целых чисел 304
Упражнения 308
Замечания по литературе 310

8. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ ЧИСЛАМИ И ПОЛИНОМАМИ 311

8.1. Аналогии между целыми числами и полиномами 312
8.2. Умножение и деление целых чисел 313
8.3. Умножение и деление полиномов 320
8.4. Модульная арифметика 323
8.5. Модульная арифметика полиномов и вычисление их значений 327
8.6. Применение китайской теоремы об остатках 329
8.7. Китайская теорема об остатках и интерполяция полиномов 333
8.8. Наибольшие общие делители и алгоритм Евклида 336
8.9. Асимптотически быстрый алгоритм нахождения НОД полиномов 339
8.10. НОД целых чисел 345
8.11. Еще раз о применении китайской теоремы об остатках 347
8.12. Разреженные полиномы 348
Упражнения 350
Замечания по литературе 353

9. АЛГОРИТМЫ ИДЕНТИФИКАЦИИ 354

9.1. Конечные автоматы и регулярные выражения 354
9.2. Распознавание образов, задаваемых регулярными выражениями 363
9.3. Распознавание подцепочек 367
9.4. Двусторонний детерминированный магазинный автомат 373
9.5. Позиционные деревья и идентификаторы позиций 385
Упражнения 398
Замечания по литературе 403

10. NP-ПОЛНЫЕ ЗАДАЧИ 404

10.1. Недетерминированные машины Тьюринга 405
10.2. Классы $\mathcal{P}$ и $\mathcal{NP}$ 414
10.3. Языки и задачи 417
10.4. NP-полнота задачи выполнимости 420
10.5. Еще несколько NP-полных задач 428
10.6. Задачи с полиномиально ограниченной памятью 440
Упражнения 446
Замечания по литературе 450

11. НЕКОТОРЫЕ ДОКАЗУЕМО ТРУДНО РАЗРЕШИМЫЕ ЗАДАЧИ 451

11.1. Иерархии по сложности 451
11.2. Иерархия по емкостной сложности для детерминированных машин Тьюринга 452
11.3. Задача, требующая экспоненциальных времени и памяти 456

11.4. Неэлементарная задача 466
Упражнения 471
Замечания по литературе 474

12. НИЖНИЕ ОЦЕНКИ ЧИСЛА АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ 475

12.1. Поля 475
12.2. Еще раз о неветвящихся программах 477
12.3. Матричная формулировка задач 479
12.4. Нижняя граница для числа умножений, связанная с рангом по строкам 480
12.5. Нижняя граница для числа умножений, связанная с рангом по столбцам 483
12.6. Граница для числа умножений, связанная с рассмотрением строк и столбцов 488
12.7. Предварительная обработка 490
Упражнения 493
Замечания по литературе 501

СПИСОК ЛИТЕРАТУРЫ 502

ГЛОССАРИЙ 514

ИМЕННОЙ УКАЗАТЕЛЬ 516

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 519

А. Ахо, Дж. Хопкрофт, Дж. Ульман

ПОСТРОЕНИЕ И АНАЛИЗ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ

Научный редактор Л. Штейнпресс Младший редактор Н. Полякова
Художник С. Бычков
Художественный редактор В. Шаповалов
Технический редактор Л. Бирюкова Корректор Н. Гиря

ИБ № 1079

Сдано в набор 5.06.78 Подписано к печати 23.11.78 Формат 60×90¹/₁₆ Бумага типографская № 2 Гарнитура литератур. Печать высокая. Объем 16,75 бум. л. Усл. печ. л. 33,50 Уч.-изд. л. 33,92 Тираж 26000 экз. Изд. № 1/9799 Заказ № 2768. Цена 2р. 60к.

Издательство «Мир» Москва, 1-й Рижский пер., 2

Ордена Октябрьской Революции и ордена Трудового Красного Знамени Первая Образцовая типография имени А. А. Жданова Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. Москва, М-54, Валовая, 28