The Pragmatic Programmers

Второе издание

прикладные структуры данных и алгоритмы

прокачиваем навыки



Джей Венгроу

под редакцией Брайана Макдоналда

КРОК

прикладные структуры данных и алгоритмы

прокачиваем навыки

Джей Венгроу

ПИТЕР®

Санкт-Петербург · Москва · Минск

2024

Выпущено при поддержке:

КРОК

Джей Венгроу

Прикладные структуры данных и алгоритмы. Прокачиваем навыки

Перевел с английского С. Черников

Научный редактор Анна Белых, старший инженер-разработчик компании КРОК

ББК 32.973.2-018
УДК 004.422.63+004.421

Венгроу Джей

В29 Прикладные структуры данных и алгоритмы. Прокачиваем навыки. — СПб.: Питер, 2024. — 512 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2068-0

Структуры данных и алгоритмы — это не абстрактные концепции, а турбина, способная превратить ваш софт в болид «Формулы-1». Научитесь использовать нотацию «О большое», выбирайте наиболее подходящие структуры данных, такие как хеш-таблицы, деревья и графы, чтобы повысить эффективность и быстродействие кода, что критически важно для современных мобильных и веб-приложений.

Книга полна реальных прикладных примеров на популярных языках программирования (Python, JavaScript и Ruby), которые помогут освоить структуры данных и алгоритмы и начать применять их в повседневной работе. Вы даже найдете слово, которое может существенно ускорить ваш код. Практикуйте новые навыки, выполняя упражнения и изучая подробные решения, которые приводятся в книге.

Начните использовать эти методы уже сейчас, чтобы сделать свой код более производительным и масштабируемым.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1680507225 англ.
ISBN 978-5-4461-2068-0

© 2020 The Pragmatic Programmers, LLC.
© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Библиотека программиста», 2023

Права на издание получены по соглашению с The Pragmatic Programmers, LLC. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 05.07.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1500 экз. Заказ Ф-1575. Отпечатано в типографии ООО «Экопейпер», 420044, Россия, г. Казань, пр. Ямашева, д. 36Б.

Краткое содержание

https://t.me/it_boooks/2

Предисловие 19

Глава 1. О важности структур данных 27

Глава 2. О важности алгоритмов 46

Глава 3. О да! Нотация «О большое» 61

Глава 4. Оптимизация кода с помощью О-нотации 73

Глава 5. Оптимизация кода с О-нотацией и без нее 89

Глава 6. Повышение эффективности с учетом оптимистичных сценариев 106

Глава 7. О-нотация в работе программиста 123

Глава 8. Молниеносный поиск с помощью хеш-таблиц 142

Глава 9. Создание чистого кода с помощью стеков и очередей 162

Глава 10. Рекурсивно рекурсируем с помощью рекурсии 179

Глава 11. Учимся писать рекурсивный код 192

Глава 12. Динамическое программирование 215

Глава 13. Рекурсивные алгоритмы для ускорения выполнения кода 231

Глава 14. Структуры данных на основе узлов 259

Глава 15. Тотальное ускорение с помощью двоичных деревьев поиска **281**

Глава 16. Расстановка приоритетов с помощью куч **312**

Глава 17. Префиксные деревья **338**

Глава 18. Отражение связей между объектами с помощью графов **365**

Глава 19. Работа в условиях ограниченного пространства **422**

Глава 20. Оптимизация кода **433**

Приложение. Решения к упражнениям **475**

Оглавление

Отзывы о втором издании 17

Предисловие 19

Для кого эта книга 20
Новое во втором издании 20
Что вы найдете в этой книге 21
Как читать эту книгу 22
Примеры кода 24
Интернет-ресурсы 24
Благодарности 25
Обратная связь 26
От издательства 26

Глава 1. О важности структур данных 27

Структуры данных 28
Массив: базовая структура данных 29
Операции над структурами данных 30
Измерение скорости 30
Чтение 31
Поиск 34
Вставка 37
Удаление 40
Множества: как одно правило может повлиять на эффективность 41
Выводы 45
Упражнения 45

Глава 2. О важности алгоритмов 46
Упорядоченные массивы 47
Поиск в упорядоченном массиве 50
Бинарный поиск 51
Программная реализация 54
Сравнение алгоритмов бинарного и линейного поиска 56
Викторина 59
Выводы 59
Упражнения 60

Глава 3. О да! Нотация «О большое» 61
«О большое»: количество шагов при наличии N элементов 62
Суть О-нотации 63
Погружение в суть О-нотации 65
Один алгоритм, разные сценарии 66
Алгоритм третьего типа 66
Логарифмы 67
Значение выражения O(log N) 68
Практические примеры 69
Выводы 71
Упражнения 71

Глава 4. Оптимизация кода с помощью О-нотации 73
Пузырьковая сортировка 73
Пузырьковая сортировка в действии 75
Программная реализация 79
Эффективность пузырьковой сортировки 81
Квадратичная задача 83
Линейное решение 85
Выводы 87
Упражнения 87

Глава 5. Оптимизация кода с О-нотацией и без нее 89
Сортировка выбором 89
Сортировка выбором в действии 90
Программная реализация 96

Эффективность сортировки выбором ... 97
Игнорирование констант ... 98
Категории алгоритмической сложности в О-нотации ... 100
Практический пример ... 102
Значимые шаги ... 103
Выводы ... 103
Упражнения ... 104

Глава 6. Повышение эффективности с учетом оптимистичных сценариев ... 106

Сортировка вставками ... 106
Сортировка вставками в действии ... 108
Программная реализация ... 112
Эффективность сортировки вставками ... 114
Средний случай ... 116
Практический пример ... 118
Выводы ... 121
Упражнения ... 121

Глава 7. О-нотация в работе программиста ... 123

Среднее арифметическое четных чисел ... 124
Конструктор слов ... 125
Выборка из массива ... 127
Среднее значение температуры в градусах Цельсия ... 128
Бирки для одежды ... 129
Подсчет единиц ... 130
Поиск палиндрома ... 131
Вычисление произведений всех пар чисел ... 132
Работа с несколькими наборами данных ... 134
Взломщик паролей ... 135
Выводы ... 138
Упражнения ... 138

Глава 8. Молниеносный поиск с помощью хеш-таблиц ... 142

Хеш-таблицы ... 142
Хеширование ... 143
Создание тезауруса для удовольствия и прибыли, но в основном для прибыли ... 145

Поиск в хеш-таблице ... 147
Однонаправленный поиск ... 148
Разрешение коллизий ... 148
Создание эффективной хеш-таблицы ... 151
Великий компромисс ... 152
Хеш-таблицы для организации данных ... 153
Хеш-таблицы для ускорения выполнения кода ... 155
Подмножество массива ... 156
Выводы ... 160
Упражнения ... 160

Глава 9. Создание чистого кода с помощью стеков и очередей ... 162

Стеки ... 162
Абстрактные типы данных ... 165
Стек в действии ... 166
Программная реализация: линтер на базе стека ... 170
О важности ограниченных структур данных ... 173
Очереди ... 173
Реализация очереди ... 175
Очередь в действии ... 176
Выводы ... 177
Упражнения ... 178

Глава 10. Рекурсивно рекурсируем с помощью рекурсии ... 179

Рекурсия вместо цикла ... 179
Базовый случай ... 181
Чтение рекурсивного кода ... 182
Рекурсия глазами компьютера ... 184
Стек вызовов ... 185
Переполнение стека ... 187
Обход файловой системы ... 187
Выводы ... 190
Упражнения ... 190

Глава 11. Учимся писать рекурсивный код ... 192

Категория рекурсивных задач: многократное выполнение действия ... 192
Рекурсивный прием: передача дополнительных параметров ... 193

Категория рекурсивных задач: вычисления ... 197
Два подхода к вычислениям ... 198
Нисходящая рекурсия: новый способ мышления ... 199
Нисходящий способ мышления ... 200
Вычисление суммы элементов массива ... 200
Обращение строки ... 202
Подсчет символов «x» в строке ... 202
Задача с лестницей ... 204
Базовый случай для задачи с лестницей ... 207
Генерация анаграмм ... 209
Эффективность алгоритма генерации анаграмм ... 211
Выводы ... 213
Упражнения ... 213

Глава 12. Динамическое программирование ... 215

Бесполезные рекурсивные вызовы ... 215
Пошаговый разбор выполнения рекурсивной функции max ... 217
Маленькое исправление для большого «О» ... 219
Эффективность рекурсии ... 220
Перекрывающиеся подзадачи ... 221
Динамическое программирование с помощью мемоизации ... 223
Реализация мемоизации ... 225
Восходящее динамическое программирование ... 227
Восходящий подход для вычисления элементов последовательности Фибоначчи ... 227
Мемоизация и восходящий подход ... 228
Выводы ... 229
Упражнения ... 229

Глава 13. Рекурсивные алгоритмы для ускорения выполнения кода ... 231

Разбиение ... 232
Программная реализация ... 235
Алгоритм быстрой сортировки (Quicksort) ... 238
Программная реализация ... 243
Эффективность быстрой сортировки ... 244
Взгляд на быструю сортировку сверху ... 245
Вычисление эффективности быстрой сортировки с помощью О-нотации ... 247

Временная сложность быстрой сортировки в худшем сценарии 250
Быстрая сортировка и сортировка вставками 251
Алгоритм быстрого выбора 252
Эффективность алгоритма быстрого выбора 254
Программная реализация 254
Сортировка как основа для других алгоритмов 255
Выводы 257
Упражнения 257

Глава 14. Структуры данных на основе узлов 259

Связные списки 259
Реализация связного списка 261
Чтение 263
Программная реализация: чтение элементов связного списка 264
Поиск 265
Программная реализация: поиск по связному списку 265
Вставка 266
Программная реализация: вставка элемента в связный список 269
Удаление 271
Программная реализация: удаление элемента из связного списка 272
Эффективность операций над связными списками 273
Связные списки в действии 274
Двусвязные списки 275
Программная реализация: добавление элемента в двусвязный список 276
Движение вперед и назад 277
Очереди на основе двусвязных списков 277
Программная реализация 278
Выводы 279
Упражнения 280

Глава 15. Тотальное ускорение с помощью двоичных деревьев поиска 281

Деревья 282
Двоичные деревья поиска 284
Поиск 285
Эффективность поиска в двоичном дереве поиска 287
Log(N) уровней 288

Программная реализация: поиск значения в двоичном дереве ... 289
Вставка ... 290
Программная реализация: вставка значения в двоичное дерево поиска ... 292
Порядок вставки ... 293
Удаление ... 294
Удаление узла с двумя дочерними элементами ... 296
Поиск узла-преемника ... 297
Узел-преемник с правым дочерним элементом ... 299
Полный алгоритм удаления ... 300
Программная реализация: удаление значения из двоичного дерева поиска ... 300
Эффективность удаления значения из двоичного дерева поиска ... 305
Двоичные деревья поиска в действии ... 305
Обход двоичного дерева поиска ... 306
Выводы ... 310
Упражнения ... 310

Глава 16. Расстановка приоритетов с помощью куч ... 312

Приоритетные очереди ... 312
Кучи ... 314
Свойство кучи ... 315
Полные деревья ... 316
Особенности кучи ... 317
Вставка в кучу ... 318
Поиск последнего узла ... 321
Удаление из кучи ... 322
Кучи и упорядоченные массивы ... 326
Проблема последнего узла… снова ... 327
Массивы в качестве куч ... 330
Обход кучи на основе массива ... 331
Программная реализация: вставка значения в кучу ... 332
Программная реализация: удаление значения из кучи ... 334
Другие варианты реализации кучи ... 336
Кучи в качестве приоритетных очередей ... 336
Выводы ... 337
Упражнения ... 337

Глава 17. Префиксные деревья 338

Префиксные деревья 339

Узел префиксного дерева 339

Класс Trie 340

Хранение слов 340

Важность звездочки 342

Поиск в префиксном дереве 344

Программная реализация 346

Эффективность поиска в префиксном дереве 348

Вставка значения в префиксное дерево 349

Программная реализация 353

Создание функции автозаполнения 354

Собираем все слова 354

Пошаговый разбор рекурсивных вызовов 356

Завершение функции автозаполнения 360

Префиксные деревья с дополнительными значениями: улучшенная функция автозаполнения 361

Выводы 362

Упражнения 363

Глава 18. Отражение связей между объектами с помощью графов 365

Графы 366

Графы и деревья 366

Терминология графов 367

Простейшая реализация графа 367

Ориентированные графы 368

Объектно-ориентированная реализация графа 368

Поиск по графу 371

Поиск в глубину 373

Пошаговый разбор поиска в глубину 374

Программная реализация 381

Поиск в ширину 382

Пошаговый разбор поиска в ширину 383

Программная реализация 392

Поиск в глубину и поиск в ширину 393

Эффективность поиска по графу ... 395
Временная сложность O(V + E) ... 397
Взвешенные графы ... 398
Реализация взвешенного графа ... 400
Задача о кратчайшем пути ... 400
Алгоритм Дейкстры ... 401
Подготовка алгоритма Дейкстры ... 402
Этапы алгоритма Дейкстры ... 403
Пошаговый разбор алгоритма Дейкстры ... 404
Поиск кратчайшего пути ... 411
Программная реализация: алгоритм Дейкстры ... 412
Эффективность алгоритма Дейкстры ... 418
Выводы ... 419
Упражнения ... 419

Глава 19. Работа в условиях ограниченного пространства ... 422

Выражение пространственной сложности с помощью O-нотации ... 422
Компромисс между временем выполнения и занимаемой памятью ... 425
Скрытые издержки рекурсии ... 428
Выводы ... 430
Упражнения ... 431

Глава 20. Оптимизация кода ... 433

Предварительное условие: определение текущей эффективности ... 433
Определение лучшей эффективности из возможных ... 434
Развитие воображения ... 435
Волшебные поиски ... 436
Волшебный поиск авторов книг ... 436
Дополнительная структура данных ... 438
Проблема двух сумм ... 440
Выявление закономерностей ... 443
Игра с монетками ... 443
Генерация примеров ... 445
Перестановка чисел для уравнивания сумм (задача о sum swap) ... 446
Жадные алгоритмы ... 451
Максимальный элемент массива ... 451

Наибольшая сумма элементов подраздела массива ... 452
Жадные предсказания цен на акции ... 459
Замена структуры данных ... 464
Алгоритм для проверки анаграмм ... 464
Группировка элементов массива ... 467
Выводы ... 470
Заключительные мысли ... 470
Упражнения ... 471

Приложение. Решения к упражнениям ... 475

Глава 1 ... 475
Глава 2 ... 476
Глава 3 ... 477
Глава 4 ... 477
Глава 5 ... 478
Глава 6 ... 479
Глава 7 ... 480
Глава 8 ... 480
Глава 9 ... 482
Глава 10 ... 483
Глава 11 ... 484
Глава 12 ... 487
Глава 13 ... 488
Глава 14 ... 490
Глава 15 ... 493
Глава 16 ... 495
Глава 17 ... 496
Глава 18 ... 497
Глава 19 ... 501
Глава 20 ... 502

Во втором издании «Прикладных структур данных и алгоритмов» вы найдете много полезных дополнений к и без того превосходному ресурсу для изучения структур данных и алгоритмов. Объяснение сложных тем вроде динамического программирования простым языком и контрольные вопросы в конце каждой главы делают эту книгу бесценной для всех разработчиков ПО, вне зависимости от наличия у них профильного образования.

Джейсон Пайк,
старший инженер-программист, KEYSYS Consulting

Идеальное введение в тему алгоритмов и структур данных для начинающих. Настоятельно рекомендую к прочтению!

Брайан Шо,
ведущий программист, Schau Consulting

Предисловие

Структуры данных и алгоритмы — это не просто абстрактные концепции. Освоив их, вы сможете писать *эффективный* код, благодаря которому программное обеспечение (ПО) работает быстрее и потребляет меньше памяти. Это особенно важно для современных приложений, которые работают на все более мобильных платформах и обрабатывают все больше данных.

Но от большинства ресурсов, посвященных этим темам, мало толку. Как правило, они перегружены специальными терминами, из-за чего тем, кто далек от математики, бывает трудно понять суть. Даже книги, авторы которых обещают «легкое» знакомство с алгоритмами, предполагают наличие у читателя математического образования. Из-за этого многие избегают изучения этих концепций, считая себя недостаточно «умными» для их понимания.

По правде говоря, все, что касается структур данных и алгоритмов, опирается на здравый смысл. Математика — это просто особый язык, и все ее концепции можно объяснить в доступной для любого читателя форме. Чтобы вам было несложно и интересно изучать весь изложенный материал, я использовал только понятную терминологию (и много диаграмм!).

Изучив все приведенные здесь концепции, вы сможете писать качественный, эффективный и быстрый код. Взвесив все плюсы и минусы вариантов кода, вы примете обоснованное решение о том, какой из них лучше всего подходит для конкретной ситуации.

Чтобы сделать все эти концепции максимально доступными и практичными, я привожу в этом пособии идеи, которые вы можете применить уже *сегодня*. Конечно, по ходу дела вы изучите и кое-какую теорию. Но эта книга посвящена в первую очередь тому, как применять на первый взгляд абстрактные идеи на

практике. Если вы хотите писать более качественный код и создавать более быстрое ПО, то обратились по адресу.

Для кого эта книга

Эта книга подойдет для вас, если вы:

- студент-информатик, который нуждается в простом и понятном ресурсе для изучения структур данных и алгоритмов. Эта книга станет хорошим дополнением к любому классическому учебнику, которым вы пользуетесь;
- начинающий разработчик, который знаком с азами программирования, но хочет изучить основы компьютерных наук, чтобы писать более качественный код и расширить свои знания в этой области;
- программист-самоучка, который никогда не изучал информатику формально (или разработчик, который ее изучал, но уже все забыл!), но хочет использовать мощь структур данных и алгоритмов для написания более качественного и масштабируемого кода.

Кем бы вы ни были, я постарался написать книгу так, чтобы она была доступна для понимания людям с любым уровнем подготовки.

Новое во втором издании

Почему второе? С момента публикации первого издания прошли годы, за которые я успел поделиться этим материалом с разными аудиториями. Я немного видоизменил способ подачи информации и нашел дополнительные темы, которые посчитал важными и интересными. Помимо этого, я осознал важность практических упражнений для применения изученных концепций.

Итак, ниже перечислены несколько основных отличий второго издания от первого.

1. *Отредактированный материал.* Чтобы изложенную в главах информацию было проще понять, я сильно видоизменил текст. В первом издании все термины тоже изложены вполне доступным языком, но я нашел возможность еще сильнее упростить подачу материала.

   Я полностью переписал многие разделы исходных глав и добавил ряд совершенно новых. По-моему, этого уже достаточно для публикации нового издания книги.

2. *Новые главы и темы.* Во второе издание добавлены шесть новых глав, посвященных особенно интересным, на мой взгляд, темам.

   В первом издании теория тоже сочеталась с практикой, но в обновленное я добавил еще больше материала, который вы можете сразу же применить. Главы 7 и 20 посвящены только отчетам об ошибках, повседневному кодированию и тому, как знание структур данных и алгоритмов может помочь в написании более эффективного ПО.

   В этой книге я постарался максимально доходчиво объяснить тему рекурсии. Я уже посвящал ей одну из глав в прошлом издании, но в это добавил новую — 11-ю, где изложил процесс написания рекурсивного кода, так как знаю, что это может вызвать много сложностей. Я не видел, чтобы этот вопрос рассматривался где-то еще, и считаю такое дополнение уникальным и ценным. Кроме того, я добавил главу 12, где рассказал о динамическом программировании, довольно популярной теме, важной для повышения эффективности рекурсивного кода.

   Существующих структур данных довольно много, и бывает трудно выбрать, какие из них добавить в книгу, а какие — нет. Но в последнее время я наблюдаю рост интереса к кучам и префиксным деревьям, которые мне тоже кажутся весьма занимательными. Об этих структурах данных мы поговорим в главах 16 и 17.

3. *Упражнения и решения.* Теперь в каждой главе вы найдете ряд упражнений для практического закрепления изученных концепций (все подробные решения вы найдете в приложении в конце книги). Благодаря этому улучшению книга превратилась в более полное учебное пособие.

Что вы найдете в этой книге

Как вы уже могли догадаться, бóльшая часть этой книги посвящена структурам данных и алгоритмам. Пособие состоит из следующих глав.

В главах 1 и 2 мы поговорим о том, что такое структуры данных и алгоритмы, и рассмотрим концепцию временной сложности, которая используется для определения скорости работы алгоритма. По ходу дела мы обсудим массивы, множества и двоичный (бинарный) поиск.

Глава 3 очень важна. В ней я постараюсь максимально доступно объяснить суть нотации «О большое», которая упоминается на протяжении всей книги.

В главах 4, 5 и 6 мы углубимся в эту тему и используем О-нотацию, чтобы заставить код работать быстрее. Попутно я расскажу о разных алгоритмах сортировки, в том числе о сортировке пузырьком, выбором и вставками.

В главе 7 вы примените все полученные ранее знания и оцените эффективность реального фрагмента кода.

В главах 8 и 9 мы обсудим несколько дополнительных структур данных, включая хеш-таблицы, стеки и очереди. Я покажу, как они влияют на скорость работы и простоту кода и как их использовать для решения реальных задач.

Глава 10 посвящена рекурсии — важнейшей концепции в мире информатики. Мы подробно разберем ее и посмотрим, когда она может пригодиться. Глава 11 поможет вам овладеть непростым навыком написания рекурсивного кода.

В главе 12 я покажу, как оптимизировать рекурсивный код и не допустить того, чтобы он вышел из-под контроля. В главе 13 вы узнаете, как использовать рекурсию в качестве основы для сверхбыстрых алгоритмов сортировки и выбора, и усовершенствуете свои навыки разработки алгоритмов.

В главах 14, 15, 16, 17 и 18 мы исследуем структуры данных на основе узлов, включая связный список, двоичное и префиксное дерево, кучу, граф, и узнаем, когда может пригодиться каждая из них.

В главе 19 мы поговорим о пространственной сложности алгоритма, которая имеет большое значение при создании программ для устройств с небольшим объемом дискового пространства или при работе с объемными данными.

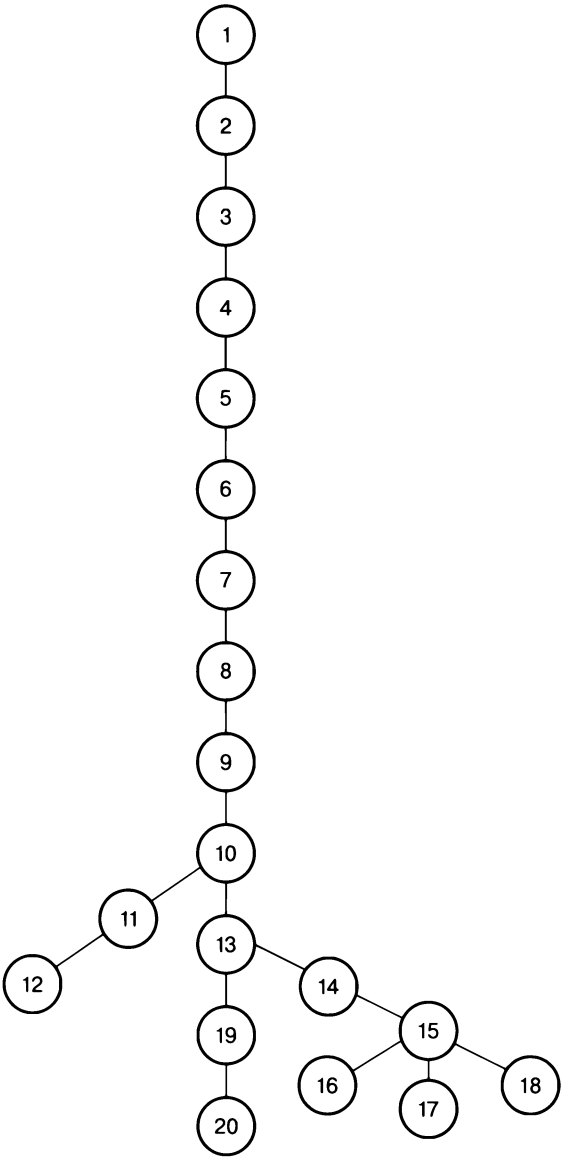
В последней главе рассматриваются разные практические приемы оптимизации эффективности кода и предлагаются новые идеи по улучшению кода, который вы пишете каждый день.

Как читать эту книгу

Главы нужно читать по порядку. Есть книги, где некоторые разделы можно пропускать, но *эта не относится к их числу*: здесь каждая последующая глава основана на информации из предыдущей. Структура книги выстроена так, чтобы вы могли углубляться в материал по мере чтения.

Но во второй половине книги есть главы, которые не вполне отвечают этому принципу. На диаграмме ниже показано, какие из глав взаимосвязаны и должны читаться по порядку.

Например, при желании после чтения главы 10 вы можете перейти сразу к главе 13 (кстати, эта диаграмма основана на структуре данных «дерево», с которой вы познакомитесь в главе 15).



Еще одно важное замечание: чтобы облегчить понимание материала, в книге я не всегда даю всю информацию о каком-то понятии при его первом упоминании. Иногда лучший способ раскрыть суть сложной концепции — сначала объяснить небольшую ее часть и лишь после ее усвоения переходить к следующей.

Не воспринимайте первое данное мной определение как общепринятое, пока не закончите чтение раздела по этой теме.

Я пошел на компромисс: чтобы книгу было легче читать, вначале я максимально упрощаю некоторые понятия, углубляясь в них по мере продвижения, вместо того чтобы стремиться к академической точности каждого предложения. Но не беспокойтесь, ближе к концу у вас сформируется четкое представление о пройденном материале.

Примеры кода

Концепции в этой книге не относятся к одному конкретному языку программирования. Поэтому примеры кода, которые вы здесь найдете, написаны на *разных языках*, в частности Ruby, Python и JavaScript, знакомство с которыми будет весьма кстати.

Разрабатывая примеры, я старался, чтобы все они были понятны даже тому, кто никогда не сталкивался с языком, на котором они написаны. По этой же причине я не использую идиомы, если чувствую, что они могут сбить с толку незнакомого с ними читателя.

Несмотря на то что частое переключение с одного языка программирования на другой может потребовать от читателя некоторых усилий, я считаю, что это позволило сделать книгу независимой от конкретного языка. Опять же, я постарался сделать примеры на *всех* языках простыми и понятными.

В разделе «Программная реализация» вы найдете несколько длинных фрагментов кода. Я рекомендую вам изучить их, но вам не обязательно понимать каждую строку, чтобы перейти к следующему разделу книги. Если вас утомляет изучение этих объемных фрагментов, просто пропустите их.

Наконец, важно отметить, что не каждый фрагмент кода здесь можно использовать на практике. В первую очередь я старался объяснить конкретные концепции, и, хотя я пытался сделать код полностью рабочим, я вполне мог не учесть некоторые пограничные случаи. У вас есть уйма возможностей для дальнейшей оптимизации приведенного здесь кода — не стесняйтесь!

Интернет-ресурсы

У этой книги есть своя веб-страница (https://pragprog.com/titles/jwdsal2), где вы найдете всю дополнительную информацию. Вы также можете помочь нам, сообщив об ошибках и опечатках, или поделиться предложениями относительно содержания пособия.

Благодарности

Может показаться, что готовая книга — это плод труда одного человека. Но любое издание, в том числе и это, не смогло бы выйти в свет без участия *множества* людей. Все они поддерживали меня на протяжении процесса ее написания. И я хотел бы лично поблагодарить *каждого*.

Я хочу сказать спасибо своей замечательной жене Рене за уделенное время и эмоциональную поддержку. За то, что заботилась обо всем, пока я писал, запершись в кабинете, как затворник. Спасибо моим очаровательным детям Туви, Лии, Шайе и Рами за терпение, которое они проявляли, пока я работал над книгой об «алгоритмах». И да — наконец-то все позади.

Спасибо моим родителям, Говарду и Дебби Венгроу, за то, что пробудили во мне интерес к программированию и помогли встать на этот путь. Приглашая репетитора для обучения девятилетнего меня, они и не подозревали, что закладывают фундамент моей будущей карьеры, а теперь еще и этой книги.

Хочу поблагодарить родителей моей жены, Пола и Крейндел Пинкус, за постоянную поддержку, за мудрость и теплоту, которые очень много для меня значат.

Когда я впервые отправил свою рукопись в Pragmatic Bookshelf, я думал, что она и так уже довольно хороша. Но благодаря опыту, конструктивной критике и предложениям замечательных сотрудников издательства книга стала намного лучше. Я хочу поблагодарить своего редактора Брайана Макдональда за то, что он познакомил меня с реальным процессом работы над книгой и поделился идеями, которые сделали лучше каждую главу этого пособия. Я выражаю признательность главному редактору Сюзанне Пфальцер и исполнительному редактору Дэйву Рэнкину. Вы приняли мою рукопись и показали, какой может быть эта книга, превратив ее в ресурс, доступный любому программисту. Спасибо издателям Энди Ханту и Дэйву Томасу за то, что поверили в этот проект и сделали Pragmatic Bookshelf замечательным издательством, с которым хочется сотрудничать.

Хочу поблагодарить чрезвычайно талантливого разработчика ПО и художника Коллин Макгакин за преобразование моих каракулей в красивые цифровые изображения. Это пособие не было бы таким потрясающим без иллюстраций, которые вы создали с присущим вам мастерством и вниманием к деталям.

Мне повезло, что так много экспертов оценили эту книгу. Благодаря их отзывам я смог максимально проработать весь текст, поэтому хотел бы поблагодарить их всех.

Рецензенты первого издания: Алессандро Бахгат, Иво Бальберт, Альберто Боскетти, Хавьер Кольядо, Мохамед Фуад, Дерек Грэхем, Нил Хэйнер, Питер

Хэмптон, Род Хилтон, Джефф Холланд, Джессика Янюк, Аарон Калэир, Стефан Кампер, Арун С. Кумар, Шон Линдсей, Найджел Лоури, Джой Маккэффри, Дейвид Морган, Джасдип Наранг, Стивен Орр, Кеннет Парех, Джейсон Пайк, Сэм Роуз, Фрэнк Руис, Брайан Шо, Тибор Симич, Маттео Ваккари, Стивен Вольф и Питер В. А. Вуд.

Рецензенты второго издания: Ринальдо Бонаццо, Майк Браун, Крейг Кастелаз, Джейкоб Че, Зульфикар Дхармаван, Ашиш Диксит, Дэн Дайбас, Эмили Экдал, Дерек Грэхем, Род Хилтон, Джефф Холланд, Грант Казан, Шон Линдсей, Найджел Лоури, Дэри Меркенс, Кевин Митчелл, Нуран Махмуд, Дэвид Морган, Брент Моррис, Эмануэль Ориджи, Джейсон Пайк, Айон Рой, Брайан Шо, Митчелл Фолк и Питер В. А. Вуд.

Помимо рецензентов, я хотел бы поблагодарить и всех читателей бета-версии книги, которые предоставляли отзывы в процессе написания и редактирования ее текста. Ваши предложения, комментарии и вопросы бесценны.

Я хотел бы выразить признательность за оказанную поддержку всем сотрудникам, студентам и выпускникам учебного лагеря по кодированию Actualize, проектом которого изначально и была эта книга. Особую благодарность выражаю Люку Эвансу за то, что он подал мне идею написать ее.

Спасибо вам всем за ваш вклад в реализацию этого проекта.

Обратная связь

Я очень люблю получать обратную связь, поэтому предлагаю вам найти меня в LinkedIn: https://www.linkedin.com/in/jaywengrow. Я с радостью приму ваш запрос на подключение — просто напишите, что вы читатель этой книги. С нетерпением жду ваших сообщений!

Джей Венгроу
jay@actualize.co
Май 2020 года

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

О важности структур данных

https://t.me/it_boooks/2

Когда человек только учится программировать, его основная цель — обеспечение правильной работы кода, которая оценивается с помощью одного простого критерия — фактической работоспособности.

Но с опытом к разработчикам ПО приходит понимание дополнительных нюансов, влияющих на *качество* кода. Они узнают, что два разных фрагмента кода могут решать одну задачу, но при этом один из них может быть *лучше* другого.

Есть много показателей качества кода, но один из важнейших — его сопровождаемость, которая охватывает такие аспекты, как читабельность, структурированность и модульность.

Еще одна отличительная черта качественного кода — его *эффективность*. Например, у вас может быть два фрагмента кода, решающих одну задачу, но один из них *может работать быстрее, чем другой*.

Взгляните на следующие две функции, каждая из которых выводит на экран все четные числа от 2 до 100:

```
def print_numbers_version_one():
  number = 2

  while number <= 100:
    # Если число четное, вывести его на экран:
    if number % 2 == 0:
      print(number)

    number += 1

def print_numbers_version_two():
  number = 2

  while number <= 100:
    print(number)
```

```
# Увеличить число на 2, чтобы получить следующее четное число:
number += 2
```

Как вы думаете, какая из функций работает быстрее?

Если вы выбрали версию 2, то вы правы. Дело в том, что в первой версии цикл выполняется 100 раз, а во второй — только 50. Получается, что версия 1 требует вдвое больше шагов, чем 2.

В этой книге мы будем говорить о написании *эффективного* кода. Умение писать код, который работает быстро, — один из важнейших навыков хорошего разработчика ПО.

Чтобы развить это умение, сначала нужно разобраться в том, что такое структуры данных и как они влияют на скорость работы создаваемого кода. Итак, начнем.

Структуры данных

Поговорим о данных.

Данные — это обширный термин, охватывающий все типы информации, вплоть до простейших чисел и строк. В классической программе «Hello World!» строка `"Hello World!"` будет фрагментом данных. По сути, даже самые сложные фрагменты данных обычно состоят из чисел и строк.

Структура данных — это то, как они *организованы*. Позже вы узнаете, что одни и те же данные могут быть организованы по-разному.

Рассмотрим следующий фрагмент кода:

```
x = "Hello! "
y = "How are you "
z = "today?"

print x + y + z
```

Эта простая программа работает с тремя фрагментами данных и выводит три строки для составления одного связного сообщения. Если бы нам потребовалось описать структуру данных в этой программе, мы бы сказали, что у нас есть три независимые строки, каждая из которых содержится в одной переменной.

Но эти же данные могут храниться и в массиве:

```
array = ["Hello! ", "How are you ", "today?"]

print array[0] + array[1] + array[2]
```

В этой книге вы узнаете, что организация данных важна не просто сама по себе, — она может значительно повлиять на *скорость выполнения вашего кода*. В зависимости от выбранного способа организации данных ваша программа может работать быстрее или медленнее, причем разница может быть в несколько порядков. А если вы создаете программу, которой предстоит обрабатывать большие объемы данных, или веб-приложение, используемое одновременно тысячами людей, то от выбранных структур данных может зависеть, будет ваше ПО нормально работать или сбоить из-за чрезмерной нагрузки.

Когда вы разберетесь с влиянием структур данных на производительность программного обеспечения, у вас появятся ключи к написанию быстрого и качественного кода, а вы сами перейдете на совершенно новый уровень как специалист.

В этой главе мы проанализируем две структуры данных: массивы и множества. Несмотря на их похожесть, они по-разному влияют на производительность программы. И проанализировать это влияние нам помогут описанные далее инструменты.

Массив: базовая структура данных

Массив — это одна из простейших структур данных в информатике. Надеюсь, вы уже работали с массивами и знаете, что это — списки элементов. Массив — универсальный инструмент, который может быть полезен во многих ситуациях. Рассмотрим небольшой пример.

В исходном коде приложения, позволяющего пользователям создавать и использовать списки покупок, вам может встретиться такой фрагмент:

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

Этот массив включает пять строк, каждая из которых содержит наименование продукта, который я могу купить в супермаркете (вы *просто обязаны* попробовать ягоды бузины (elderberries)).

При работе с массивами применяется особый технический жаргон.

Под *размером* массива понимается количество содержащихся в нем элементов. В нашем примере размер массива равен 5, так как он содержит пять значений.

Индекс массива — это число, определяющее позицию элемента в массиве.

В большинстве языков программирования нумерация индексов начинается с 0. Итак, в нашем примере индекс элемента `"apples"` — 0, а `"elderberries"` — 4:



Операции над структурами данных

Чтобы оценить производительность структуры данных, например массива, нужно проанализировать способы взаимодействия кода с ней.

Есть четыре основных способа использования структур данных, которые мы называем *операциями:*

- *Чтение* — получение элемента из определенного места структуры данных. В случае с массивом это означает поиск значения с определенным индексом. Например, поиск названия продукта с индексом 2 — это *чтение* массива.
- *Поиск* — нахождение определенного значения в структуре данных. В случае с массивом это означает выяснение того, есть ли конкретное значение в массиве, и если да, то какой у него индекс. Пример: *поиск* элемента `"dates"` в нашем списке продуктов.
- *Вставка* — добавление нового значения в структуру данных, в нашем случае — в массив. Если бы мы решили добавить инжир (`"figs"`) в список покупок, это было бы примером *вставки* нового значения в массив.
- *Удаление* — исключение значения из структуры данных. В случае с массивом это означает удаление из него одного из элементов. Например, если мы решим исключить бананы (`"bananas"`) из списка покупок, это значение будет *удалено* из массива.

В этой главе мы попробуем разобраться в том, насколько быстро выполняется каждая из этих операций.

Измерение скорости

Итак, как же измерить скорость выполнения операции?

Если вы вынесете из этой книги только один урок, пусть он будет таким: при измерении скорости выполнения операции мы учитываем не количество *времени*, а число *шагов*, которое для этого требуется.

Мы уже сталкивались с этим в примере функции, выводящей на экран четные числа от 2 до 100. Вторая версия работала быстрее, потому что требовала вдвое меньше шагов, чем первая.

Почему мы измеряем скорость выполнения кода числом шагов?

Дело в том, что мы никогда не можем однозначно сказать, что какая-то операция занимает, скажем, пять секунд. Выполнение одного и того же фрагмента кода может занять пять секунд на более новом компьютере и гораздо больше времени на старом оборудовании. А, допустим, на суперкомпьютерах будущего этот же код сможет работать еще быстрее. Измерять скорость операции в секундах или минутах ненадежно, поскольку длительность зависит от оборудования, на котором она выполняется.

Но мы *можем* измерить скорость операции в количестве вычислительных *шагов*, необходимых для ее выполнения. Если операция А требует 5 шагов, а Б — 500, мы можем предположить, что первая всегда будет выполняться быстрее, чем вторая, на любых компьютерах. Поэтому определение числа шагов — ключевой этап анализа скорости выполнения операции.

Измерение скорости выполнения операции по-другому еще называют измерением ее *временной сложности*. В книге я буду использовать термины *скорость*, *временная сложность*, *эффективность*, *производительность* и *время выполнения* как синонимы. Все они относятся к количеству шагов, необходимых для выполнения операции.

Теперь рассмотрим четыре операции над массивом и определим, сколько шагов нужно для выполнения каждой из них.

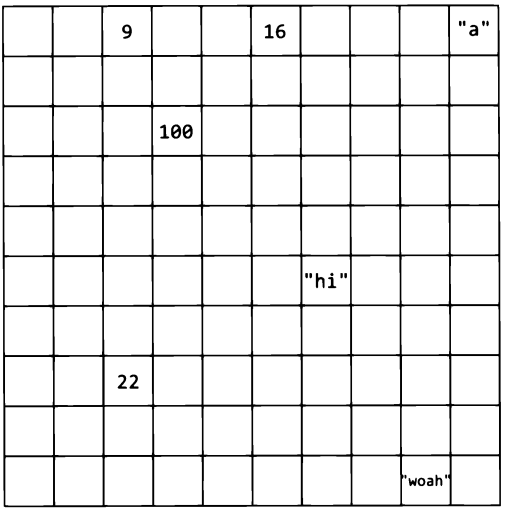
Чтение

Начнем с *чтения* — с определения значения элемента массива с конкретным индексом.

Компьютер может выполнить эту операцию всего за один шаг, потому что может обратиться к любому элементу массива с необходимым индексом и считать его значение. Если бы мы решили найти элемент с индексом 2 в массиве `["apples", "bananas", "cucumbers", "dates", "elderberries"]`, то компьютер перешел бы прямо к нему и сообщил, что в этой позиции находится значение `"cucumbers"`.

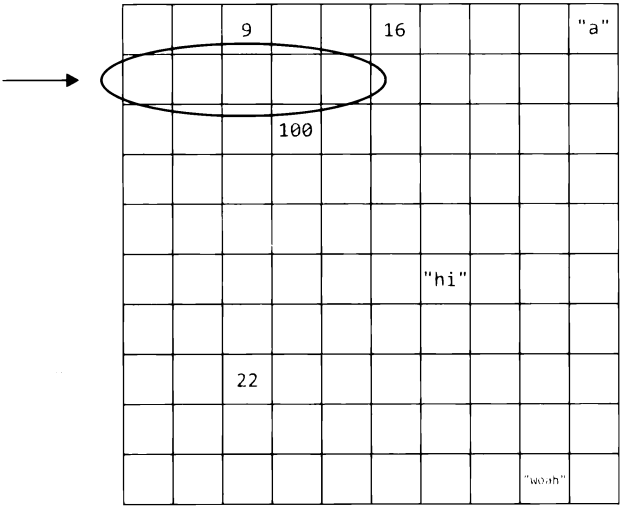
Как компьютеру удается отыскать нужный элемент всего за шаг? Давайте разберемся.

Память компьютера похожа на гигантский набор ячеек. Ниже показана сетка, где некоторые ячейки пустые, а некоторые содержат биты данных.



Хотя это очень упрощенная модель устройства памяти компьютера, она хорошо передает основную идею.

При объявлении массива программа выделяет для него непрерывный блок памяти из пустых ячеек. Так, если вы создаете массив для хранения пяти элементов, ваш компьютер находит группу из пяти пустых ячеек подряд и выделяет ее для использования в качестве массива:



У каждой ячейки есть определенный адрес. От почтового, где содержатся названия улиц и городов, его отличает лишь то, что он представлен числом. Значение адреса каждой последующей ячейки памяти на одну единицу превышает значение предыдущей:

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

На следующей схеме показан массив из нашего примера со списком покупок с индексами и адресами ячеек памяти.

	"apples"	"bananas"	"cucumbers"	"dates"	"elderberries"
Адрес ячейки памяти:	1010	1011	1012	1013	1014
Индекс:	0	1	2	3	4

Когда компьютер считывает значение по определенному индексу массива, он может сразу обратиться к нужному элементу в силу следующих факторов.

1. Компьютер может перейти к любому *адресу памяти* за один шаг. Например, если вы попросите его проверить значение в ячейке памяти с адресом 1063, он сможет получить к нему доступ без выполнения поиска. Точно так же, если я попрошу вас поднять мизинец правой руки, вам не придется перебирать все пальцы, чтобы определить нужный.

2. Выделяя блок памяти под массив, компьютер отмечает, с какого адреса он *начинается*. Так, если мы попросим компьютер найти первый элемент массива, он мгновенно обратится к соответствующему адресу памяти.

Теперь мы знаем, как именно компьютер может найти первое значение массива за один шаг. Впрочем, он может найти значение с *любым* индексом, выполнив простое сложение. Если бы мы попросили компьютер найти значение массива с индексом 3, он просто взял бы адрес памяти, соответствующий индексу 0, и прибавил к нему 3 (в конце концов, у адресов памяти последовательная нумерация).

Применим это к нашему массиву списка продуктов. Он начинается с адреса памяти 1010. Итак, если бы мы велели компьютеру считать значение элемента с индексом 3, он выполнил бы следующую логическую операцию.

1. Массив начинается с индекса 0, который соответствует адресу памяти 1010.
2. Индекс 3 находится ровно через три ячейки после индекса 0.
3. Логично предположить, что элемент с индексом 3 находится в ячейке памяти с адресом 1013, поскольку 1010 + 3 = 1013.

Как только компьютер выяснит, что индекс 3 соответствует адресу памяти 1013, он обратится к соответствующей ячейке и узнает, что в ней содержится значение `"dates"`.

Чтение из массива — довольно эффективная операция, ведь компьютер может обратиться к любому адресу памяти за один шаг. Хотя я разбил мыслительный процесс компьютера на три этапа, сейчас нас интересует его обращение к адресу памяти (в следующих главах мы подробнее поговорим о том, на каких шагах стоит сосредоточиться).

Конечно же, операция, которая выполняется всего за шаг, самая быстрая. Выходит, что массив — это не только основная, но и очень мощная структура данных, потому что мы можем считывать значения в нем с огромной скоростью.

А что, если бы вместо выяснения значения в позиции с индексом 3 мы попросили бы компьютер узнать индекс элемента `"dates"`? В этом случае речь пойдет об операции поиска, которую мы рассмотрим далее.

Поиск

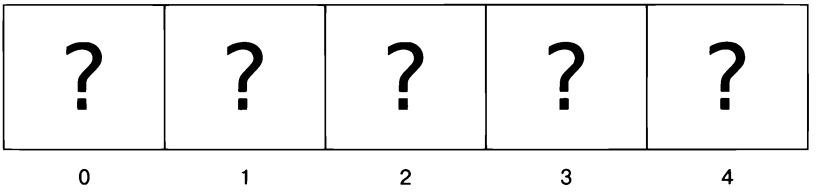
Как я уже говорил, при *поиске* в массиве компьютер проверяет, есть ли в нем конкретное значение, и если да, то в позиции с каким индексом.

В каком-то смысле эта операция обратна чтению, при котором мы предоставляем компьютеру *индекс* и просим возвратить соответствующее ему значение. При поиске же мы даем компьютеру *значение* и просим возвратить его индекс.

Эти две операции кажутся похожими, но в плане эффективности они сильно разнятся. Чтение выполняется быстро, так как компьютер может сразу обратиться к любому индексу и выяснить значение соответствующего элемента. Поиск требует времени, ведь у компьютера нет возможности сразу перейти к нужному значению.

Важный факт: компьютер может мгновенно получить доступ к любому адресу памяти, но не имеет ни малейшего представления о том, какие *значения* содержатся в каждой из ячеек.

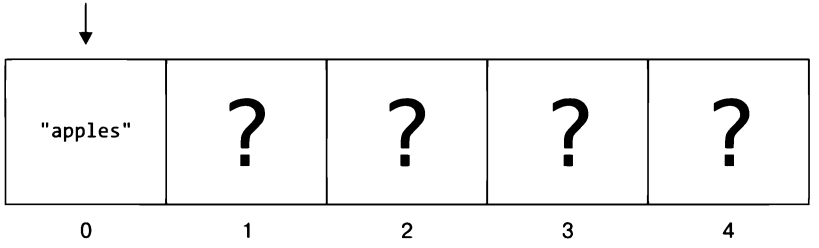
Вернемся к нашему примеру со списком продуктов. Компьютер не может сразу узнать содержимое каждой ячейки памяти. Для него массив выглядит примерно так:



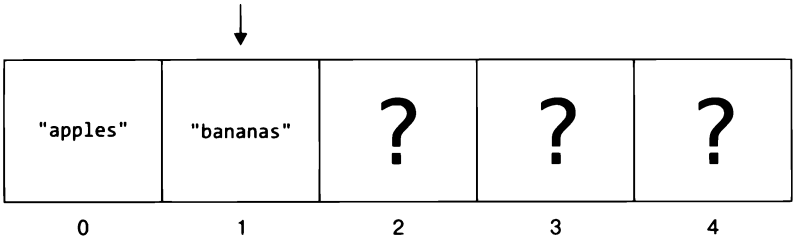
Чтобы найти нужное значение в массиве, он вынужден проверять все ячейки по очереди.

На следующих изображениях показан процесс, который компьютер будет использовать для поиска значения `"dates"` в нашем массиве.

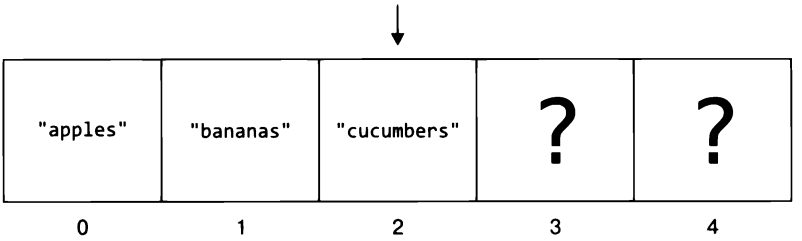
Сначала компьютер проверяет элемент с индексом 0:



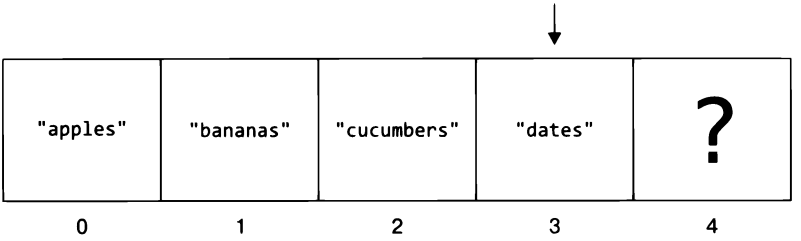
Поскольку его значение `"apples"`, а не `"dates"`, компьютер переходит к следующему индексу:



В позиции с индексом 1 тоже нет искомого значения `"dates"`, поэтому компьютер переходит к индексу 2:



И снова нам не повезло, и компьютер переходит к следующей ячейке:



Ура! Мы нашли нужное значение и теперь знаем, что его индекс — 3. Компьютеру больше не нужно переходить к следующей ячейке массива, так как он уже нашел то, что мы просили.

В этом примере поиск был выполнен за четыре шага, поскольку компьютеру пришлось проверить четыре ячейки, чтобы обнаружить значение `"dates"`.

В главе 2 вы узнаете о другом способе поиска в массиве, но эта базовая операция поиска, при которой компьютер проверяет каждую ячейку по одной, называется *линейным поиском*.

Какое же *максимальное* количество шагов может выполнить компьютер, чтобы провести линейный поиск в массиве?

Если искомое значение окажется в последней ячейке (например, `"elderberries"`), компьютеру придется перебрать *все* элементы массива, чтобы его отыскать.

Кроме того, если этого значения и вовсе нет в массиве, все равно придется проверить каждую ячейку, чтобы в этом убедиться.

Итак, получается, что при линейном поиске в массиве из пяти элементов максимальное число шагов равно пяти, а в массиве из 500 элементов — 500.

Проще говоря, линейный поиск в массиве из N элементов потребует максимум N шагов. Здесь N — переменная, которую можно заменить любым числом.

В любом случае понятно, что поиск менее эффективен, чем чтение, поскольку может требовать множества шагов, тогда как чтение всегда предполагает только один, вне зависимости от размера массива.

Теперь разберем операцию вставки.

Вставка

Эффективность операции вставки нового фрагмента данных в массив зависит от того, *куда* именно вы его вставляете.

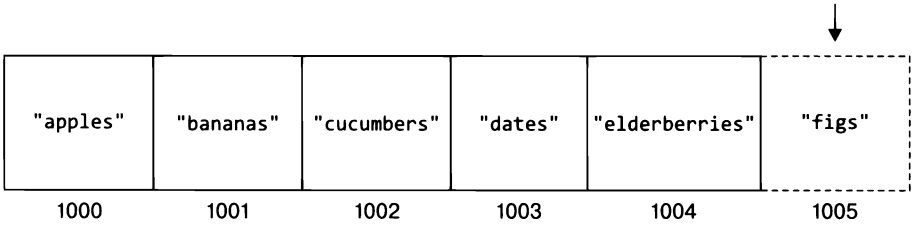
Допустим, мы хотим добавить значение `"figs"` в конец нашего списка покупок. На такую вставку у нас уйдет один шаг.

Это обусловлено еще одним фактором: при выделении блока памяти под массив компьютер всегда отслеживает его размер.

Если мы прибавим к этому тот факт, что компьютер знает, с какого адреса памяти массив начинается, то вычислить местонахождение его последнего элемента не составит труда: если массив начинается с адреса памяти 1010, а его размер 5, то его последний элемент хранится в ячейке 1014. Выходит, вставка элемента после него означала бы его добавление к *следующему* адресу памяти, равному 1015.

После вычисления адреса ячейки, в которую нужно поместить новое значение, компьютер сможет сделать это за один шаг.

Так выглядит вставка значения `"figs"` в конец массива:



Но есть одна проблема. Поскольку изначально компьютер выделил под массив только пять ячеек памяти, при добавлении шестого элемента ему придется найти дополнительные ячейки. Как правило, это делается автоматически, но каждый язык программирования обрабатывает этот процесс по-своему, поэтому я не буду вдаваться в детали.

Мы рассмотрели процесс вставки нового элемента в конец массива, но вставка в *начало* или в *середину* — это совсем другая история. В этих случаях нам приходится *сдвигать* фрагменты данных, чтобы освободить место для нового значения, а это требует дополнительных шагов.

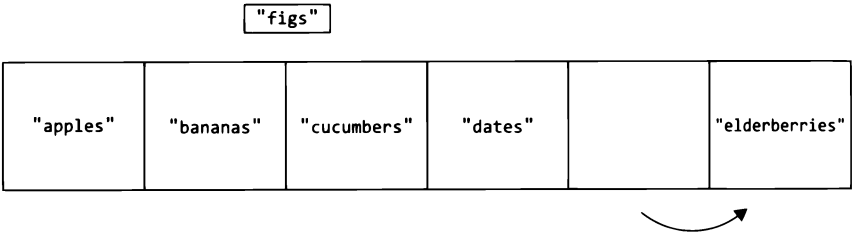
Допустим, мы хотим добавить значение `"figs"` в позицию массива с индексом 2:



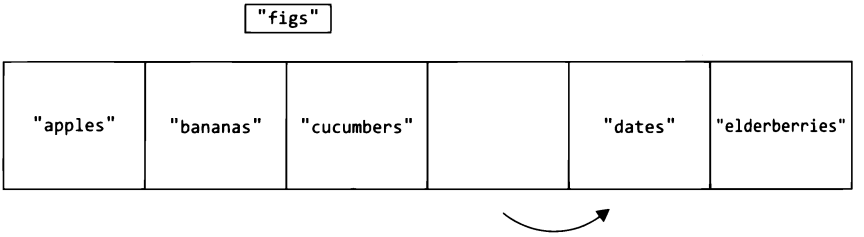
Чтобы освободить место для нового значения, нам нужно сдвинуть `"cucumbers"`, `"dates"` и `"elderberries"` вправо. На это у нас уйдет несколько шагов, так как нам нужно сначала сдвинуть значение `"elderberries"` на одну позицию вправо, чтобы освободить место для перемещения `"dates"`, а затем сдвинуть значение `"dates"`, чтобы освободить место для `"cucumbers"`.

Рассмотрим этот процесс подробнее.

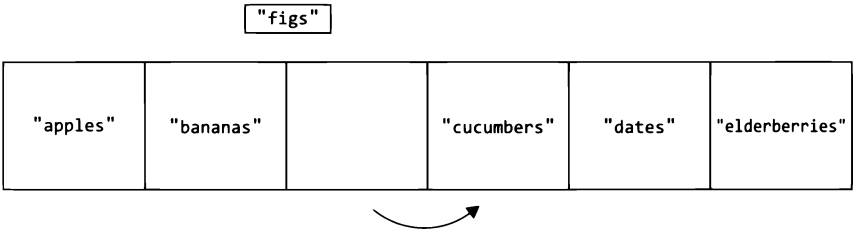
Шаг 1: сдвигаем вправо значение `"elderberries"`:



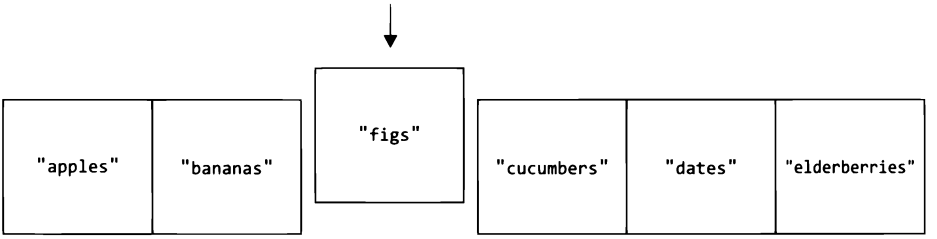
Шаг 2: сдвигаем вправо значение `"dates"`:



Шаг 3: сдвигаем вправо значение `"cucumbers"`:



Шаг 4: вставляем значение `"figs"` в позицию с индексом 2:



Обратите внимание, что в этом примере на операцию вставки ушло четыре шага: три из них заняло передвижение элементов вправо и только один ушел на фактическую вставку нового значения.

Худший сценарий при вставке в массив — тот, на который уйдет больше всего шагов, — вставка значения в *начало*, ведь нам придется сдвинуть вправо *все* остальные значения.

Проще говоря, в худшем случае вставка значения в массив из N элементов может потребовать $N + 1$ *шагов,* ведь перед вставкой нового значения нам нужно переместить все N элементов.

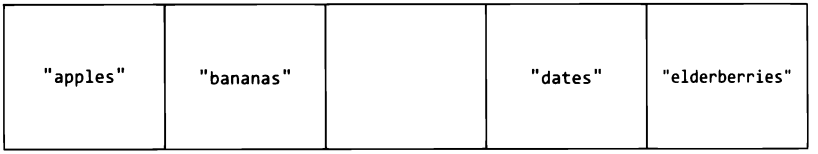
Теперь рассмотрим последнюю операцию над массивом — удаление элемента.

Удаление

Удаление элемента массива сводится к исключению значения в позиции с определенным индексом.

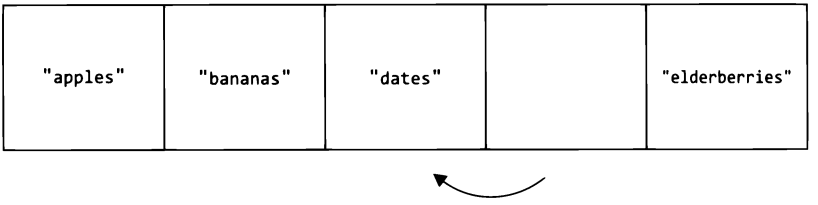
Давайте удалим из нашего массива значение с индексом 2 — `"cucumbers"`.

Шаг 1: удаляем значение `"cucumbers"` из массива:

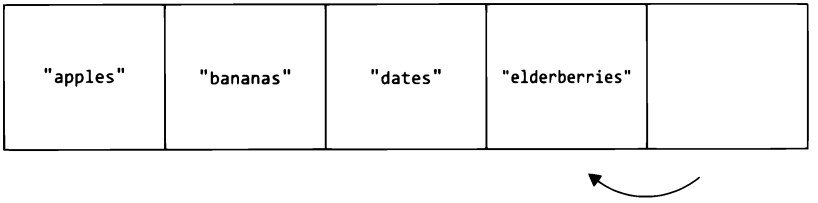


Хотя на удаление значения ушел всего шаг, у нас возникла пустая ячейка в середине массива. Сложно работать с массивом, когда в нем есть пробелы, поэтому для решения этой проблемы нам нужно сдвинуть влево значения `"dates"` и `"elderberries"`. Это означает, что процесс удаления требует дополнительных шагов.

Шаг 2: сдвигаем влево значение `"dates"`:



Шаг 3: сдвигаем влево значение `"elderberries"`:



Итак, на операцию удаления у нас ушло в общей сложности три шага: первый сводится к фактическому удалению значения, а два других — к сдвигу остальных значений для избавления от пробела.

Как и в случае вставки, худший сценарий при удалении — избавление от первого элемента массива, потому что для избавления от пробела в позиции с индексом 0 нам пришлось бы сдвинуть влево все оставшиеся элементы.

В случае с массивом из пяти элементов удаление первого потребовало бы одного шага, а сдвиг четырех оставшихся — четырех. В случае с массивом из 500 элементов удаление первого потребовало бы одного шага, а сдвиг оставшихся — 499. Следовательно, операция удаления значения из массива, состоящего из N элементов, потребует максимум N шагов.

Поздравляю! Вы проанализировали временну́ю сложность первой структуры данных. Теперь пришло время узнать о том, что у разных структур данных разная эффективность. Это важный аспект, так как выбор структуры данных для использования в коде может сильно повлиять на производительность вашего ПО.

Следующая структура данных — *множество* — на первый взгляд очень похожа на массив. Но позже вы увидите, что эффективность операций над массивами и множествами разная.

Множества: как одно правило может повлиять на эффективность

Теперь поговорим о *множестве*. Множество — это структура данных без повторяющихся значений.

Есть разные типы множеств, но мы будем говорить о *множестве на базе массива*. Это, как и массив, простой список значений. Единственная разница между ним и классическим массивом в том, что множество не допускает вставку повторяющихся значений.

Например, если вы попытаетесь добавить еще одно значение `"b"` в множество `["a", "b", "c"]`, компьютер просто не позволит этого сделать — ведь `"b"` уже есть в этом множестве.

Множества полезны, когда вам нужно гарантировать отсутствие дубликатов.

Например, если вы создаете телефонный онлайн-справочник, вам не нужно, чтобы один и тот же номер встречался в нем более одного раза. На самом деле, я сам столкнулся с такой проблемой: в местной телефонной книге наш домашний номер дублируется — по какой-то ошибке он указан не только рядом с моей фамилией, но и рядом с фамилией Зиркинд (да, это реальная история). Вы даже не представляете, как мне надоело получать телефонные звонки и сообщения

от людей, пытающихся дозвониться до Зиркиндов. Я уверен, что и Зиркинды недоумевают, почему им никто не звонит. А когда я звоню Зиркиндам, чтобы сообщить об ошибке, трубку берет моя жена, потому что я набираю свой номер (ладно, последнюю часть я придумал). Если бы только в этой программе использовалось множество...

В любом случае, множество на базе массива — это массив с одним дополнительным ограничением: запретом дубликатов. Хотя этот запрет полезен, он влияет на *эффективность* одной из четырех основных операций.

Рассмотрим проведение четырех основных операций на базе массива.

Чтение из множества выполняется точно так же, как и из массива — чтобы выяснить значение в позиции с определенным индексом, компьютеру нужен всего один шаг. Как вы уже знаете, это связано с тем, что компьютер может перейти к любому индексу множества, потому что способен с легкостью вычислить соответствующий адрес памяти и обратиться к нему.

Поиск в множестве тоже ничем не отличается от поиска в массиве и требует до N шагов. То же самое касается удаления элемента из множества: чтобы удалить значение и сдвинуть данные влево, нужно до N шагов.

Но, когда дело доходит до вставки, между массивами и множествами появляется различие. Сначала рассмотрим процесс вставки значения в *конец* множества (лучший сценарий для массива). Как мы уже убедились, компьютер может вставить значение в конец массива за один шаг.

В случае с множеством компьютеру сначала нужно убедиться, что вставляемого значения в нем еще нет, потому что множества используются именно для предотвращения вставки дубликатов.

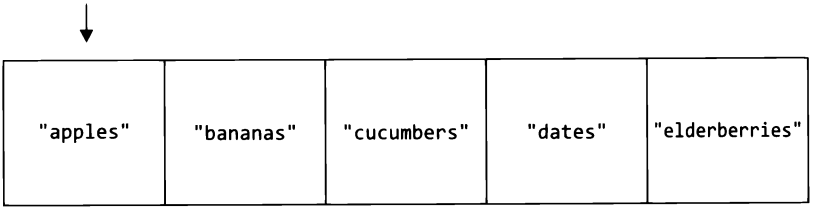
Как же удостовериться в том, что множество еще не содержит данные, которые мы пытаемся в него добавить? Как вы помните, компьютер не знает, какие значения содержатся в ячейках массива или множества, поэтому ему сначала нужно выполнить *поиск* и убедиться, что значения, которое мы хотим вставить, нет. Только тогда компьютер разрешит вставку.

Итак, в случае с множеством каждая операция вставки требует *предварительного выполнения поиска*.

Рассмотрим этот процесс на примере. Представьте, что наш прошлый список продуктов сохранен в виде множества. Это было бы верным решением, ведь мы не хотим дважды покупать один и тот же продукт. Если мы захотим вставить в множество `["apples", "bananas", "cucumbers", "dates", "elderberries"]`

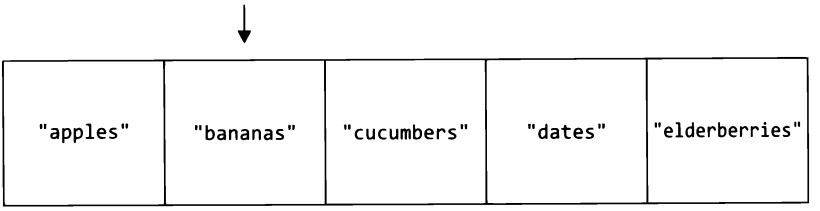
значение `"figs"`, компьютеру придется выполнить следующие шаги, начиная с поиска вставляемого значения.

Шаг 1: проверка позиции с индексом 0 на предмет наличия значения `"figs"`:

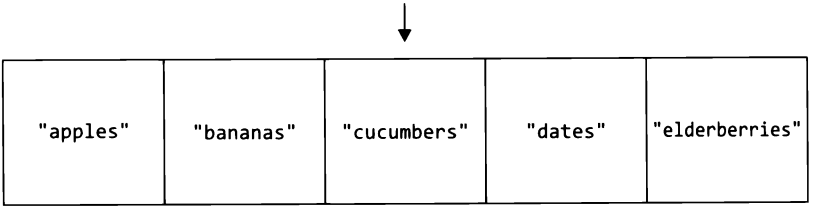


Его там нет, но оно может быть где-то еще. Нужно убедиться, что в этом множестве нет значения `"figs"`, прежде чем мы сможем его вставить.

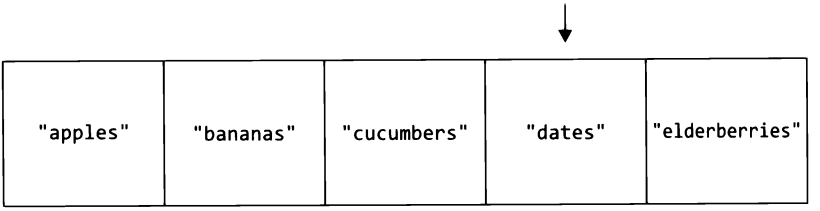
Шаг 2: проверка позиции с индексом 1:



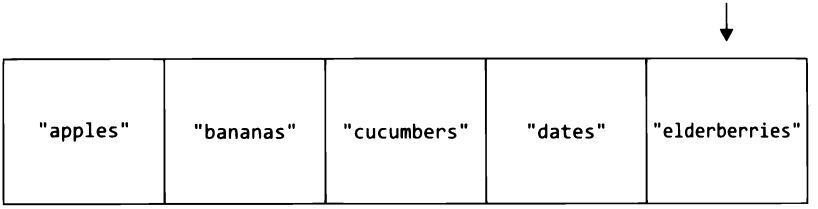
Шаг 3: проверка позиции с индексом 2:



Шаг 4: проверка позиции с индексом 3:

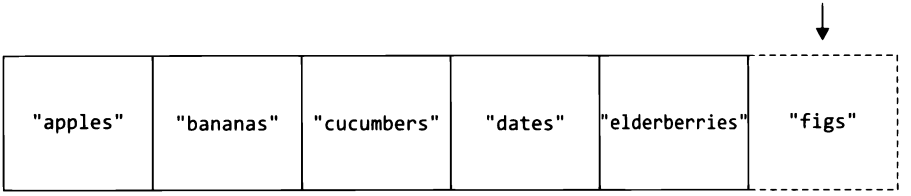


Шаг 5: проверка позиции с индексом 4:



После проверки всего множества мы можем с уверенностью сказать, что в нем нет значения `"figs"`. Теперь мы можем выполнить завершающий шаг операции вставки:

Шаг 6: вставка значения `"figs"` в конец множества:



Вставка значения в конец множества — это лучший сценарий. Но нам все же пришлось выполнить шесть шагов для множества, изначально содержащего пять элементов, проверив их все перед выполнением вставки.

Иначе говоря, вставка значения в конец множества из N элементов может потребовать до $N + 1$ шагов: N шагов для поиска, чтобы убедиться в отсутствии вставляемого значения, и один для фактической вставки. Сравните это с обычным массивом, где на вставку значения уходит один шаг.

В худшем случае при вставке значения в *начало* множества компьютеру придется проверить N ячеек, чтобы убедиться, что множество еще не содержит вставляемого значения, выполнить N шагов, чтобы сдвинуть все значения вправо, и еще один для вставки нового значения. Итого: $2N + 1$ шагов. Сравните это со вставкой значения в начало обычного массива, которая требует всего $N + 1$ шагов.

Означает ли это, что использования множеств нужно избегать лишь потому, что вставка значений в них выполняется медленнее, чем в обычные массивы? Ни в коем случае. Множества очень полезны, когда нужно обеспечить отсутствие дубликатов (надеюсь, однажды ошибка в моей телефонной книге будет исправлена). Но если такой необходимости нет, лучше выбрать массив, поскольку операция вставки для него будет гораздо эффективнее. Важно проанализировать потребности своего приложения и выбрать подходящую для него структуру данных.

Выводы

Подсчет количества шагов для выполнения операций лежит в основе анализа производительности структур данных. От выбора подходящей структуры зависит способность программы справляться с большой нагрузкой. В этой главе вы научились проводить такой анализ для сравнения массива и множества.

Теперь, когда вы познакомились с концепцией временной сложности структур данных, мы можем применить этот анализ для сравнения разных алгоритмов (даже в рамках *одной* структуры данных), чтобы обеспечить максимальную скорость и производительность кода. Об этом мы и поговорим в следующей главе.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 1».

1. Для массива из 100 элементов укажите число шагов для выполнения следующих операций:
   а) чтение;
   б) поиск значения, которого нет в массиве;
   в) вставка значения в начало массива;
   г) вставка значения в конец массива;
   д) удаление первого значения массива;
   е) удаление последнего значения массива.
2. Для множества на базе массива из 100 элементов укажите число шагов для выполнения следующих операций:
   а) чтение;
   б) поиск значения, которого нет в множестве;
   в) вставка нового значения в начало множества;
   г) вставка нового значения в конец множества;
   д) удаление первого значения множества;
   е) удаление последнего значения множества.
3. Обычно операция поиска в массиве направлена на нахождение первого экземпляра заданного значения, но иногда нужно найти *каждый* экземпляр этого значения. Допустим, мы хотим подсчитать, сколько раз в массиве встречается значение `"apple"`. Сколько шагов нам нужно, чтобы найти все его экземпляры? Дайте ответ с точки зрения *N*.

ГЛАВА 2

О важности алгоритмов

Мы уже рассмотрели две структуры данных и увидели, как выбор подходящей может повлиять на производительность кода. Даже такие похожие на первый взгляд структуры данных, как массив и множество, могут обладать разной эффективностью.

В этой главе вы узнаете, что помимо структуры данных на эффективность кода может повлиять и выбор *алгоритма*.

Вам может показаться, что здесь скрывается какая-то сложная концепция, но это вовсе не так. Алгоритм — это просто *набор инструкций для выполнения конкретной задачи*.

Даже простой процесс вроде приготовления завтрака — это уже алгоритм, поскольку он подразумевает выполнение определенного набора шагов для решения поставленной задачи. Алгоритм приготовления завтрака (по крайней мере, в моем случае) состоит из четырех шагов.

1. Возьмите тарелку.
2. Насыпьте в тарелку хлопья.
3. Налейте в тарелку молоко.
4. Опустите в тарелку ложку.

Если мы будем четко следовать этой инструкции, то сможем насладиться завтраком.

В случае с вычислениями под алгоритмом понимается набор инструкций, данных компьютеру для выполнения конкретной задачи. Так, когда мы пишем какой-либо код, мы создаем алгоритмы — наборы инструкций, которые должен выполнять компьютер.

Мы можем описывать алгоритмы простым языком, детально излагая инструкции, которые планируем дать компьютеру. Для описания принципа работы разных алгоритмов в книге я буду использовать как обычный язык, так и код.

Иногда для выполнения одной задачи можно использовать два разных алгоритма. Мы уже сталкивались с этим в начале первой главы, когда знакомились с двумя разными способами вывода четных чисел на экран. В том случае один алгоритм предполагал выполнение в два раза большего числа шагов, чем другой.

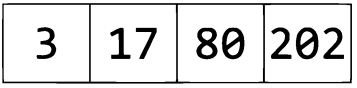
В этой главе мы познакомимся еще с двумя алгоритмами, решающими одну задачу. Только в этом случае один алгоритм будет *на несколько порядков* быстрее другого.

Но перед этим нам нужно рассмотреть новую структуру данных.

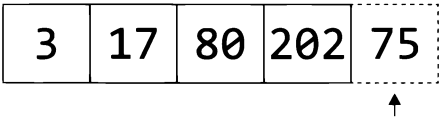
Упорядоченные массивы

Упорядоченный массив почти идентичен «классическому» из прошлой главы. Единственное отличие в том, что упорядоченные массивы требуют, чтобы значения всегда располагались — как вы уже догадались — *по порядку*. То есть новое значение нужно вставить так, чтобы в итоге все значения в массиве оставались отсортированными.

Для примера возьмем массив `[3, 17, 80, 202]`:

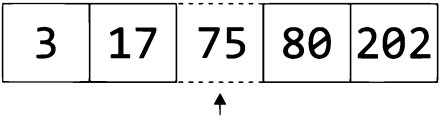


Допустим, мы хотим вставить в него число 75. Если бы этот массив был классическим, мы могли бы добавить это значение в конец:



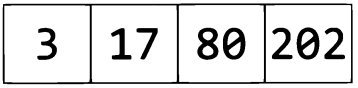
Как мы видели в главе 1, компьютер может сделать это за один шаг.

Но в случае с *упорядоченным массивом* нам придется вставить число 75 в определенную ячейку, чтобы значения располагались в порядке возрастания:

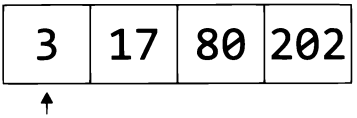


Но легче сказать, чем сделать. Компьютер не может просто вставить значение 75 в нужное место за один шаг, потому что сначала он должен *найти* подходящую позицию, а затем сдвинуть другие значения, чтобы освободить место. Рассмотрим этот процесс подробнее.

Начнем с того же упорядоченного массива.



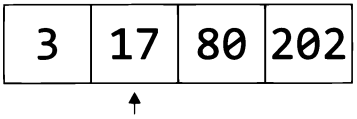
Шаг 1: проверяем значение с индексом 0, чтобы определить, где должно располагаться значение, которое мы хотим вставить (75): слева или справа от него:



Поскольку 75 больше 3, мы знаем, что новое значение должно быть вставлено где-то справа от него. Но мы еще не знаем, в какую именно ячейку, поэтому нам нужно проверить следующее значение.

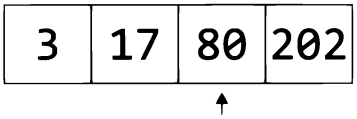
Мы назовем этот шаг *сравнением*, ведь мы сравниваем вставляемое значение с числом, присутствующим в упорядоченном массиве.

Шаг 2: проверяем значение в следующей ячейке:



Значение 75 больше 17, поэтому нам нужно двигаться дальше.

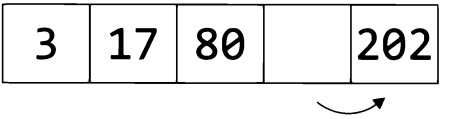
Шаг 3: проверяем значение в следующей ячейке:



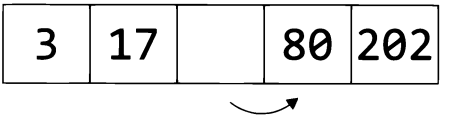
Значение 80 больше 75, поэтому мы приходим к выводу, что значение 75 нужно вставить слева от 80, чтобы сохранить порядок элементов массива. Чтобы

освободить место для вставки значения 75, нам нужно сдвинуть остальные данные.

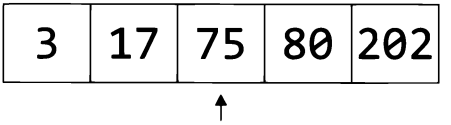
Шаг 4: сдвигаем последнее значение вправо:



Шаг 5: сдвигаем предпоследнее значение вправо:



Шаг 6: помещаем значение 75 в нужную ячейку:



Как видите, перед помещением значения в упорядоченный массив всегда нужно проводить поиск, чтобы определить правильное место вставки. Это одно из различий в производительности между классическим массивом и упорядоченным.

В этом примере мы видим, что изначально в массиве было четыре элемента, а на вставку ушло шесть шагов. Так, в случае с упорядоченным массивом из N элементов количество шагов для вставки значения равно $N + 2$.

Интересно, что это количество шагов остается одинаковым вне зависимости от того, в какую именно позицию массива помещается новое значение. При вставке значения в начало мы выполняем меньше сравнений и больше сдвигов, а ближе к концу — больше сравнений, но меньше сдвигов. Меньше всего шагов нужно для вставки нового значения в самый конец, так как эта операция не подразумевает никаких сдвигов. В этом случае мы выполняем N шагов для сравнения нового значения со всеми существующими и один для самой вставки, что в общей сложности дает $N + 1$ шагов.

Хотя упорядоченный массив проигрывает классическому в эффективности, если речь идет о вставке, он многократно превосходит его, когда дело доходит до поиска.

Поиск в упорядоченном массиве

В прошлой главе я описал процесс поиска определенного значения в классическом массиве: мы проверяем каждую ячейку по одной — слева направо, — пока не найдем искомое значение. Там же я отметил, что этот процесс называется линейным поиском. Посмотрим, чем отличается линейный поиск при работе с классическим и упорядоченным массивами.

Допустим, у нас есть обычный массив `[17, 3, 75, 202, 80]`. Если бы мы искали значение 22, которого здесь нет, нам пришлось бы проверить каждый элемент, потому что оно может быть где угодно. Учитывая это, мы можем остановить поиск до достижения конца массива, только если нам удастся найти нужное значение до этого момента.

Но в случае с упорядоченным массивом мы можем остановить поиск раньше, даже если искомого значения в массиве нет. Допустим, мы ищем число 22 в упорядоченном массиве `[3, 17, 75, 80, 202]`. Мы можем остановить поиск, как только дойдем до 75, поскольку 22 никак не может быть справа от него.

Вот так выглядит реализация линейного поиска в упорядоченном массиве на языке Ruby:

```
def linear_search(array, search_value)

  # Перебираем все элементы массива:
  array.each_with_index do |element, index|

    # Если находим искомое значение, возвращаем его индекс:
    if element == search_value
      return index

    # Если мы обнаружили элемент, значение которого превышает искомое,
    # можем выйти из цикла раньше:
    elsif element > search_value
      break
    end
  end

  # Если искомое значение в массиве не обнаружено, возвращаем nil:
  return nil

end
```

Этот метод принимает два аргумента: `array` — упорядоченный массив, где мы осуществляем поиск, и `search_value` — искомое значение.

Вот как можно использовать эту функцию для поиска значения `22` в массиве из нашего примера:

```
p linear_search([3, 17, 75, 80, 202], 22)
```

Как видите, метод `linear_search` перебирает все элементы массива в поисках значения `search_value`. Процесс останавливается, как только значение обрабатываемого элемента превышает `search_value`, поскольку мы знаем, что после этого нужного значения в массиве точно не будет.

Учитывая все это, мы можем прийти к выводу, что в определенных ситуациях линейный поиск в упорядоченном массиве может занять меньше шагов, чем в классическом. При этом, если искомое значение превышает значения остальных элементов массива или вообще отсутствует, нам все равно придется проверить каждую ячейку.

Итак, на первый взгляд, стандартные и упорядоченные массивы не сильно различаются в плане эффективности, по крайней мере, в худших сценариях. Для обоих типов, содержащих N элементов, на линейный поиск может уйти до N шагов.

Но далее мы познакомимся с мощным алгоритмом, многократно превосходящим линейный поиск.

До сих пор мы думали, что единственный способ нахождения значения в упорядоченном массиве — линейный поиск. Но это всего лишь *один из возможных алгоритмов* поиска значения, а не *единственный*.

Большое преимущество упорядоченного массива по сравнению с классическим в том, что первый позволяет использовать алгоритм *бинарного (двоичного) поиска*, который работает *гораздо* быстрее линейного.

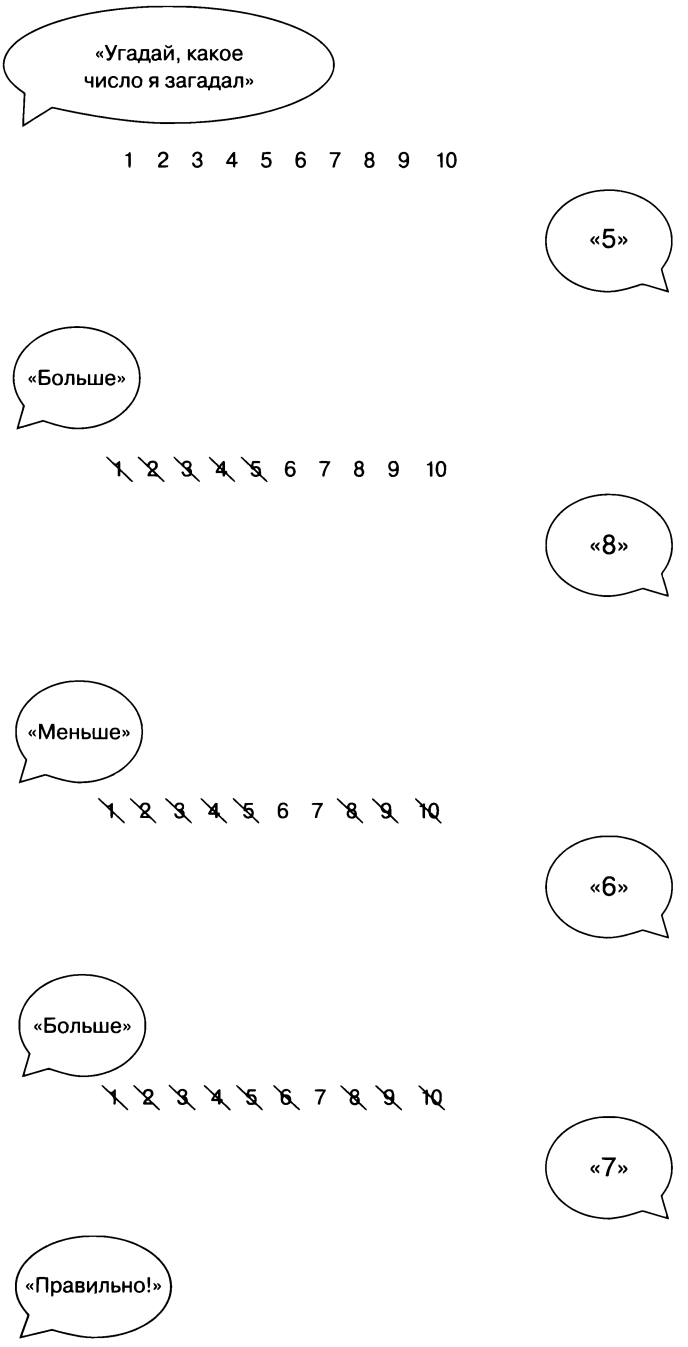
Бинарный поиск

Вы наверняка в детстве играли в такую игру: один загадывает число от 1 до 100, а другой пытается его угадать. Каждый раз, когда угадывающий называет число, ему сообщается, больше оно, чем загаданное, или нет.

Принцип этой игры понятен. Скорее всего, вы бы не начали с единицы, а вместо этого первым делом назвали бы число, которое находится ровно посередине. Почему? Потому что, выбрав 50, вне зависимости от полученной подсказки вы сразу исключаете половину возможных чисел!

Если вы называете 50, а вам говорят, что загаданное число больше, вы выбираете 75, исключая половину *оставшихся* чисел. Если затем вам скажут, что загаданное число меньше 75, вы выберете 62 или 63 и будете продолжать выбирать среднее значение, каждый раз исключая половину оставшихся чисел.

Представьте процесс угадывания числа от 1 до 10. Именно так выглядит принцип работы двоичного (или бинарного) поиска.

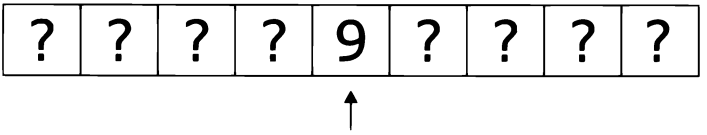

«Угадай, какое число я загадал»
1 2 3 4 5 6 7 8 9 10
«5»
«Больше»
1 2 3 4 5 6 7 8 9 10
«8»
«Меньше»
1 2 3 4 5 6 7 8 9 10
«6»
«Больше»
1 2 3 4 5 6 7 8 9 10
«7»
«Правильно!»

Посмотрим, как осуществляется бинарный поиск в упорядоченном массиве. Итак, у нас есть упорядоченный массив из, скажем, девяти элементов. Компьютер не знает, какое значение в каждой из ячеек, поэтому массив можно изобразить так:



Допустим, мы хотим найти значение 7. Вот как будет работать алгоритм двоичного поиска.

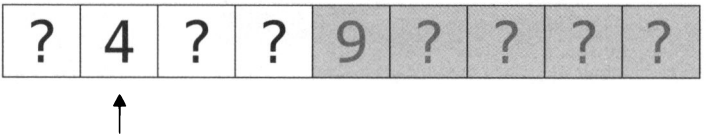
Шаг 1: начинаем поиск со средней ячейки. Мы можем обратиться к ней сразу, поскольку для вычисления ее индекса нам достаточно разделить длину массива на 2. Итак, проверяем значение в этой ячейке:



Обнаруженное значение равно 9, поэтому мы можем предположить, что число 7 находится где-то слева от него. Так мы успешно исключили половину ячеек массива — все ячейки справа от той, что содержит 9 (и ее саму):



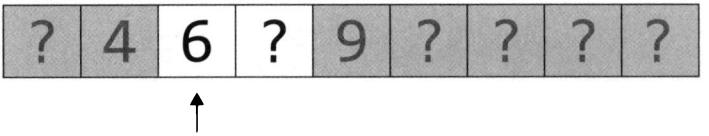
Шаг 2: проверяем среднюю из оставшихся ячеек. Таких здесь две, и мы произвольно выбираем левую:



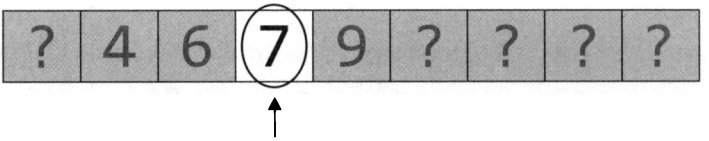
В ней мы видим число 4, поэтому искомое значение 7 должно находиться где-то справа от нее. Мы исключаем ячейку с числом 4 и ту, что слева от нее:



Шаг 3: у нас осталось две ячейки для проверки, и мы произвольно выбираем левую:



Шаг 4: проверяем последнюю ячейку (если это не 7, значит, такого значения в упорядоченном массиве вообще нет).



Мы нашли число 7 за четыре шага. Хотя в этом примере столько же шагов ушло бы и на линейный поиск, чуть позже вы сможете оценить истинную мощь двоичного.

Обратите внимание, что алгоритм двоичного поиска можно применить только к упорядоченному массиву. В классическом массиве значения могут располагаться в любом порядке, так что мы никогда не узнаем, где искать нужное значение — слева или справа от выбранной наугад ячейки. В этом и есть одно из преимуществ упорядоченных массивов: они позволяют осуществить бинарный поиск.

Программная реализация

Вот как выглядит реализация бинарного поиска на языке Ruby:

```
def binary_search(array, search_value)

  # Сначала определяем нижнюю и верхнюю границы диапазона, в котором
  # может находиться искомое значение. Изначально нижняя граница - это
  # первое значение массива, а верхняя - последнее:

  lower_bound = 0
  upper_bound = array.length - 1

  # Запускаем цикл, где последовательно проверяем средние значения диапазона:

  while lower_bound <= upper_bound do

    # Находим среднее значение между верхней и нижней границами:
```

```
    # (нам не нужно беспокоиться о том, что результат будет дробным,
    # так как в Ruby результат деления целых чисел всегда
    # округляется в меньшую сторону до ближайшего целого числа)

    midpoint = (upper_bound + lower_bound) / 2

    # Проверяем значение в средней ячейке:

    value_at_midpoint = array[midpoint]

    # Если это значение совпадает с искомым, поиск завершается.
    # Если нет, меняем нижнюю или верхнюю границу в зависимости от того,
    # превышает ли искомое значение то, которое мы обнаружили, или нет:

    if search_value == value_at_midpoint
      return midpoint
    elsif search_value < value_at_midpoint
      upper_bound = midpoint - 1
    elsif search_value > value_at_midpoint
      lower_bound = midpoint + 1
    end
  end

  # Если мы сузили границы до такой степени, что между ними не осталось
  # ячеек, значит, искомого значения в этом массиве нет:

  return nil
end
```

Разберем этот код. Как и метод `linear_search`, функция `binary_search` принимает в качестве аргументов массив `array` и искомое значение `search_value`.

Вот пример вызова этого метода:

```
p binary_search([3, 17, 75, 80, 202], 22)
```

Сначала он задает диапазон индексов ячеек, в которых может быть обнаружено значение `search_value`:

```
lower_bound = 0
upper_bound = array.length - 1
```

Поскольку в самом начале искомое значение `search_value` может быть обнаружено в любом месте массива, для нижней границы `lower_bound` мы задаем первый индекс, а для верхней `upper_bound` — последний.

Сам процесс поиска происходит внутри цикла `while`:

```
while lower_bound <= upper_bound do
```

Этот цикл выполняется, пока у нас остается диапазон ячеек, в которых может находиться искомое значение `search_value`. Как мы увидим далее, в процессе

поиска наш алгоритм будет последовательно сужать этот диапазон, пока между нижней и верхней границами не останется ячеек (`lower_bound <= upper_bound`). Если за это время искомое значение не будет найдено, мы придем к выводу, что `search_value` в массиве нет.

В цикле код проверяет значение `midpoint`, которое находится в середине диапазона:

```
midpoint = (upper_bound + lower_bound) / 2
value_at_midpoint = array[midpoint]
```

`value_at_midpoint` — это элемент в середине диапазона.

Если `value_at_midpoint` совпадает с искомым значением `search_value`, мы добились цели и можем возвратить индекс соответствующего элемента:

```
if search_value == value_at_midpoint
  return midpoint
```

Если `search_value` меньше, чем `value_at_midpoint`, значит, нужное значение следует искать левее. Так мы можем сузить диапазон поиска, выбрав в качестве `upper_bound` индекс слева от средней точки `midpoint`, поскольку `search_value` не может находиться справа от нее:

```
elsif search_value < value_at_midpoint
  upper_bound = midpoint - 1
```

И наоборот, если `search_value` больше, чем `value_at_midpoint`, значит, нужное значение следует искать справа от средней точки `midpoint`, поэтому мы увеличиваем значение `lower_bound`:

```
elsif search_value > value_at_midpoint
  lower_bound = midpoint + 1
```

Когда диапазон сужается до 0 элементов, мы возвращаем значение `nil`. В этом случае мы точно знаем, что `search_value` в массиве нет.

Сравнение алгоритмов бинарного и линейного поиска

При работе с небольшими упорядоченными массивами алгоритм двоичного поиска не сильно превосходит в эффективности алгоритм линейного поиска. Но посмотрим, что происходит с большими массивами.

Вот максимальное число шагов для выполнения каждого типа поиска в массиве со 100 значениями:

- линейный поиск — 100 шагов;
- двоичный поиск — 7 шагов.

При линейном поиске, если искомое значение находится в последней ячейке или превышает значение в ней, нам придется проверить каждый элемент. В случае с массивом в 100 значений на это уйдет 100 шагов.

Но при использовании двоичного поиска каждое предположение, которое мы делаем, исключает половину ячеек, которые нам иначе пришлось бы проверять. Первое предположение позволяет избавиться сразу от 50 ячеек.

Если мы посмотрим на это с другой стороны, то увидим закономерность.

В случае с массивом из трех элементов двоичный поиск потребовал бы максимум двух шагов.

Если мы удвоим количество ячеек в массиве (и для простоты добавим еще одну, чтобы их общее количество было нечетным), у нас получится семь ячеек. Двоичный поиск в таком массиве потребует максимум трех шагов.

Если мы снова удвоим число ячеек (и добавим еще одну), так чтобы упорядоченный массив содержал 15 элементов, то максимальное количество шагов для выполнения двоичного поиска будет равно четырем.

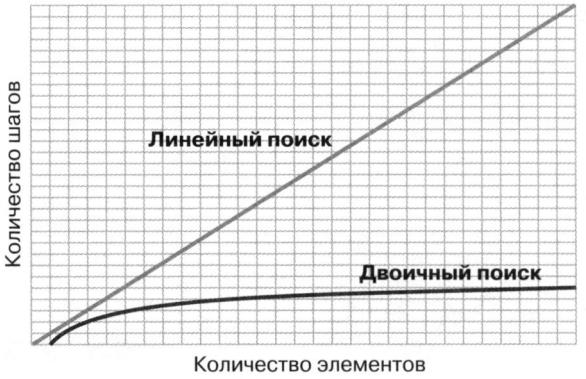
Закономерность следующая: каждый раз, когда мы удваиваем размер упорядоченного массива, количество шагов для выполнения двоичного поиска увеличивается на единицу. Это логично, так как каждый шаг поиска исключает половину элементов.

Все это говорит о необычайной эффективности такого алгоритма: каждый раз, когда мы удваиваем объем данных, в алгоритм двоичного поиска добавляется *всего один шаг.*

Сравним его с линейным поиском. Если бы в массиве было три элемента, нам потребовалось бы выполнить до трех шагов. В случае с массивом из семи элементов — до семи, а из 100 — до 100. Итак, при линейном поиске *число шагов равно количеству элементов.* Поэтому каждый раз, когда мы удваиваем размер массива, мы *удваиваем* и число шагов, которые нужно выполнить. В случае же с двоичным поиском удвоение размера массива добавляет *лишь один шаг.*

Посмотрим, как это работает на примере больших массивов. Линейный поиск в массиве из 10 000 элементов может потребовать до 10 000 шагов, а двоичный — всего 13 шагов. Линейный поиск в массиве из миллиона элементов потребует до одного миллиона шагов, а двоичный — *20 шагов*.

Разницу в производительности между алгоритмами линейного и двоичного поиска можно показать на графике:



Нам предстоит проанализировать еще много таких графиков, так что давайте потратим немного времени на то, чтобы разобраться в примере. Ось *X* отражает число элементов в массиве. То есть, двигаясь слева направо, мы имеем дело с возрастающим объемом данных.

Ось *Y* отражает число шагов алгоритма. То есть, двигаясь снизу вверх, мы имеем дело с возрастающим количеством шагов.

Если вы посмотрите на график производительности линейного поиска, то увидите, что количество шагов возрастает пропорционально росту числа элементов в массиве. По сути, для каждого дополнительного элемента в массиве алгоритм линейного поиска предусматривает один дополнительный шаг. В результате получается прямая диагональная линия.

С другой стороны, при бинарном поиске по мере увеличения объема данных количество шагов алгоритма увеличивается незначительно. Это вполне сочетается с тем, что мы уже знаем: удвоение числа элементов массива добавляет в алгоритм двоичного поиска всего один шаг.

Имейте в виду, что работать с упорядоченными массивами не всегда быстрее. Как вы уже видели, вставка значения в такой массив выполняется медленнее, чем в стандартный. Приходится идти на компромисс: при использовании упорядоченного массива мы готовы мириться с более медленной вставкой ради

более быстрого поиска. Опять же, всегда анализируйте потребности своего приложения, чтобы понять, какой из алгоритмов подходит вам больше. Будет ли ваше ПО осуществлять много вставок или, напротив, поиск для него важнее?

Викторина

Чтобы понять, разобрались ли вы с эффективностью двоичного поиска, попробуйте самостоятельно ответить на следующий вопрос (не подглядывайте в ответы).

Вопрос: на двоичный поиск в упорядоченном массиве из 100 элементов уходит семь шагов. Сколько шагов нужно, чтобы осуществить двоичный поиск в упорядоченном массиве из *200* элементов?

Ответ: 8 шагов.

Многие отвечают на этот вопрос интуитивно, говоря, что для этого нужно 14 шагов, но это неверно. Вся прелесть двоичного поиска в том, что каждая проверка значения позволяет исключить половину оставшихся элементов. Поэтому каждый раз, *удваивая* количество данных, мы добавляем в алгоритм только один шаг. В конце концов, это дублирование данных полностью исключается при первой же проверке!

Стоит отметить, что теперь, когда мы добавили в наш инструментарий бинарный поиск, вставка в упорядоченный массив тоже может стать быстрее. Как мы уже знаем, вставка требует предварительного поиска, но теперь вместо линейного мы можем использовать двоичный. Правда, вставка значения в упорядоченный массив по-прежнему будет медленнее, чем в обычный, поскольку во втором случае для вставки поиск вообще не нужен.

Выводы

Обычно есть несколько способов достижения конкретной вычислительной цели, и выбранный вами алгоритм может серьезно повлиять на скорость выполнения вашего кода.

Важно понимать, что ни одна структура данных и ни один алгоритм не могут идеально подходить для всех ситуаций. Например, то, что упорядоченные массивы позволяют осуществлять двоичный поиск, не означает, что всегда нужно использовать именно их. Если ваша программа в основном будет заниматься не поиском, а добавлением данных, лучше выбрать стандартные массивы, поскольку вставлять значения в них получится гораздо быстрее.

Мы уже изучили способ сравнения алгоритмов, который заключается в подсчете количества шагов, выполняемых каждым из них. В следующей главе мы рассмотрим формальный способ выражения временной сложности конкурирующих структур данных и алгоритмов. Освоив его, мы сможем получать более четкую информацию для принятия взвешенных решений при выборе подходящих алгоритмов.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 2».

1. Сколько шагов нужно для выполнения линейного поиска числа 8 в упорядоченном массиве `[2, 4, 6, 8, 10, 12, 13]`?
2. Сколько шагов нужно для выполнения двоичного поиска в примере выше?
3. Какое максимальное число шагов потребуется для выполнения двоичного поиска в массиве из 100 000 элементов?

ГЛАВА 3

О да! Нотация «О большое»

Ранее мы с вами говорили о том, что основной фактор, определяющий эффективность алгоритма, — это количество выполняемых им шагов.

Но мы не можем просто сказать, что один алгоритм состоит из 22 шагов, а другой — из 400, потому что количество выполняемых алгоритмом шагов не может быть сведено к конкретному числу. Возьмем, к примеру, линейный поиск. Количество шагов этого алгоритма зависит от числа элементов в массиве. Если массив содержит 22 элемента, алгоритм линейного поиска состоит из 22 шагов, а если 400 — то из 400 шагов.

Поэтому, чтобы количественно оценить эффективность этого алгоритма, мы можем сказать, что для выполнения линейного поиска в *массиве из N элементов* нужно *N шагов*. То есть если в массиве содержится *N* элементов, линейный поиск будет состоять из *N* шагов. Но такой способ выражения эффективности довольно громоздкий.

Чтобы облегчить обсуждение временной сложности, программисты позаимствовали из мира математики концепцию, благодаря которой можно лаконично и согласованно описывать эффективность структур данных и алгоритмов. Формальный способ выражения этих концепций называется *нотацией «О большое»* (Big O) или *О-нотацией* и позволяет легко оценивать эффективность любого алгоритма и сообщать о ней другим.

Разобравшись с О-нотацией, вы научитесь описывать алгоритмы согласованно и лаконично, как профессионалы.

Хотя О-нотация пришла из мира математики, я собираюсь обойтись без сложной терминологии и объяснить ее суть в контексте компьютерных наук. Мы начнем с малого: познакомимся с этой концепцией поверхностно, постепенно углуб-

ляясь в ее изучение на протяжении этой и следующих трех глав. Тема несложная, но еще проще ее будет усвоить по частям.

«О большое»: количество шагов при наличии N элементов

Согласованность О-нотации обусловлена особым способом подсчета количества шагов алгоритма. Сначала давайте попробуем оценить с ее помощью эффективность алгоритма линейного поиска.

В худшем случае количество шагов линейного поиска будет равно количеству элементов в массиве. Как мы уже говорили, линейный поиск в массиве из N элементов может потребовать до N шагов. С помощью О-нотации это можно выразить так: O(N).

Некоторые произносят это как «О большое от эн» или «сложность порядка N». Но я предпочитаю говорить просто: «О от эн».

Эта запись выражает ответ на ключевой вопрос: *сколько шагов будет выполнять алгоритм при наличии N элементов данных?* Выучите это предложение наизусть, потому что именно это определение О-нотации мы будем использовать на протяжении всей оставшейся части книги.

Ответ на ключевой вопрос в этом выражении заключен *в круглые скобки*. Запись O(N) говорит о том, что *алгоритм будет выполнять N шагов*.

Теперь давайте рассмотрим выражение временной сложности с помощью О-нотации на примере того же линейного поиска. Сначала мы задаем ключевой вопрос: если в массиве N элементов данных, сколько шагов нужно для выполнения линейного поиска? Мы знаем, что на выполнение такого поиска уйдет N шагов, поэтому выражаем ответ так: O(N). Для справки, O(N) еще называют алгоритмом с *линейной временной сложностью* или выполняемым за *линейное время*.

Сравним все это с выражением эффективности *чтения* из стандартного массива через О-нотацию. Как вы узнали из главы 1, на чтение из массива, вне зависимости от его размера, уходит всего шаг. Чтобы выразить сложность этого алгоритма в О-нотации, мы снова зададим ключевой вопрос: если в массиве N элементов данных, сколько шагов нужно для чтения из него? Ответ: один шаг, поэтому мы можем выразить сложность этого алгоритма так: O(1) (произносится как «О от единицы»).

Случай с O(1) довольно интересен: несмотря на то что ключевой вопрос основан на терминах N («Сколько шагов будет выполнять алгоритм при N элементах

данных?»), ответ никак не относится к N. В этом и суть: *сколько бы элементов* ни было в массиве, чтение из него *всегда* требует только одного шага.

И именно поэтому алгоритм со сложностью O(1) считается «самым быстрым». Даже по мере увеличения объема данных он не выполняет никаких дополнительных шагов, то есть всегда выполняет одно и то же их количество вне зависимости от значения N. Алгоритм O(1) еще называют алгоритмом с *постоянной временной сложностью* или выполняемым за *постоянное* (*константное*) *время*.

А где же математика?

Как я уже говорил, в этой книге я постарался объяснить тему О-нотации максимально просто. Но сделать это можно и другими способами. Если вы проходили формальный курс обучения, то наверняка знакомы с этой концепцией с математической точки зрения. О-нотация была позаимствована из мира математики, поэтому для ее описания часто используют математические термины. И тогда можно услышать такие выражения, как «"О" большое описывает верхнюю границу скорости роста функции» или «если функция $g(x)$ растет не быстрее функции $f(x)$, то g является элементом O(f)». Все эти фразы либо имеют смысл, либо нет — все зависит от вашего уровня подготовки. В своей книге я постарался объяснить эту тему так, чтобы ее могли понять даже те, кто далек от математики.

Если хотите углубиться в математические основы О-нотации, прочтите книгу *Introduction to Algorithms*[1] Томаса Х. Кормена, Чарльза И. Лейзерсона, Рональда Л. Ривеста и Клиффорда Штайна или статью Джастина Абрамса: https://justin.abrah.ms/computer-science/understanding-big-o-formal-definition.html.

Суть О-нотации

Теперь, когда мы познакомились с O(N) и O(1), мы начинаем понимать, что О-нотация не просто описывает количество шагов, которые выполняет алгоритм, например, с жестким числом, таким как 22 или 400. Это скорее ответ на поставленный ранее ключевой вопрос: сколько шагов будет выполнять алгоритм при наличии N элементов данных?

Но суть О-нотации не только в этом.

[1] *Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ.

Допустим, у нас есть алгоритм, который всегда выполняет три шага вне зависимости от количества данных. То есть при наличии N элементов алгоритм всегда состоит из трех шагов. Как это выразить с помощью О-нотации?

Исходя из всего, что вы узнали, вы можете ответить: O(3).

Но правильный ответ — O(1). Сейчас я объясню, почему.

Хотя с помощью О-нотации действительно можно выразить количество шагов алгоритма при наличии N элементов данных, это определение не затрагивает нечто более глубокое, «суть» этой нотации.

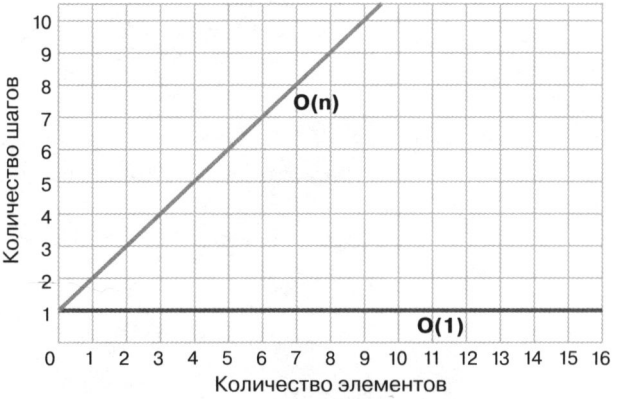
О-нотация может ответить на вопрос: *«Как изменяется производительность алгоритма по мере увеличения объема данных?»*

Она не просто говорит нам, сколько шагов выполняет алгоритм, но и показывает, как число шагов увеличивается по мере *изменения* объема данных.

В этом случае нам все равно, чему равна временная сложность алгоритма — O(1) или O(3), ведь она никак не зависит от объема данных, так как число шагов остается постоянным. По сути, оба алгоритма относятся к одному типу, ведь ни в первом, ни во втором алгоритме число шагов не меняется при изменении количества данных. Поэтому для нас они одинаковы.

В свою очередь, алгоритм со сложностью O(N) относится к другому типу, так как на его производительность *влияет* увеличение объема данных. Если конкретнее, то вместе с объемом данных увеличивается и количество шагов алгоритма. Именно об этом нам сообщает запись O(N). Она отражает пропорциональную зависимость между объемом данных и эффективностью алгоритма: описывает то, как растет количество шагов по мере увеличения объема данных.

Рассмотрим графики эффективности алгоритмов этих двух типов:



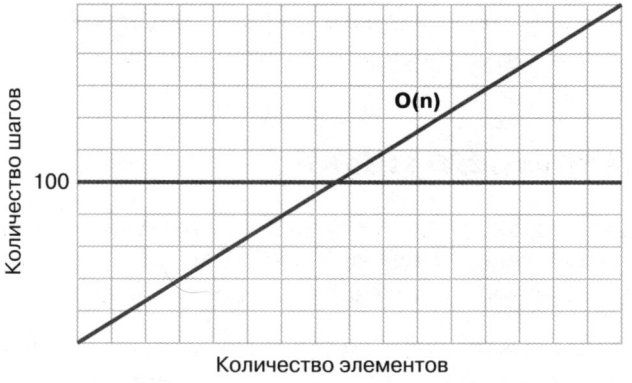
Обратите внимание, что график O(*N*) образует диагональную линию. Это связано с тем, что при каждом добавлении фрагмента данных в алгоритм добавляется один шаг: чем больше данных, тем больше шагов выполняет алгоритм.

Теперь посмотрим на график O(1), который образует горизонтальную линию. Так происходит потому, что число шагов остается неизменным вне зависимости от количества данных.

Погружение в суть О-нотации

Чтобы осознать истинную суть О-нотации, давайте немного углубимся в эту тему. Допустим, у нас есть алгоритм с постоянной сложностью, который всегда выполняет 100 шагов вне зависимости от количества данных. Насколько он эффективен по сравнению с алгоритмом O(*N*)?

Взгляните на следующий график:



Мы видим, что при работе с набором данных, где меньше 100 элементов, алгоритм O(*N*) выполняет меньше шагов, чем 100-шаговый алгоритм O(1). В точке, соответствующей 100 элементам, графики пересекаются, потому что при таком объеме данных оба алгоритма выполняют одинаковое количество шагов — 100. Но суть в другом: *при работе с массивами, где больше 100 элементов*, алгоритм O(*N*) выполняет больше шагов, чем O(1).

Поскольку всегда есть некая точка перелома, по достижении которой алгоритм O(*N*) начинает выполнять больше шагов, вплоть до бесконечности, в целом он считается менее эффективным, чем O(1), вне зависимости от того, сколько шагов выполняет последний.

То же верно и для алгоритма O(1), который всегда выполняет миллион шагов. В процессе роста объема данных неизбежно будет достигнута точка, после которой O(*N*) станет менее эффективным, чем O(1).

Один алгоритм, разные сценарии

Как вы уже знаете, сложность алгоритма линейного поиска не *всегда* O(N). Если искомый элемент находится в последней ячейке массива, то для его нахождения действительно нужно N шагов. Но когда элемент в *первой* ячейке, алгоритм линейного поиска найдет его всего за шаг. Этот случай линейного поиска будет выглядеть так: O(1). Если бы мы захотели максимально полно описать эффективность этого алгоритма, то сказали бы, что сложность линейного поиска равна O(1) *в лучшем случае* и O(N) — в *худшем*.

Хотя O-нотация эффективно описывает как лучший, так и худший сценарии для этого алгоритма, приводимые оценки обычно относятся к последнему, если не указано иное. Вот почему линейный поиск часто описывается как алгоритм со сложностью O(N), несмотря на то что в лучшем случае она *может* быть равна O(1).

Все из-за того, что такой «пессимистический» подход довольно полезен, поскольку точное знание того, насколько неэффективным может оказаться алгоритм, готовит к худшему и влияет на наше итоговое решение.

Алгоритм третьего типа

В прошлой главе вы узнали, что двоичный поиск в упорядоченном массиве выполняется намного быстрее, чем линейный. Теперь посмотрим, как можно описать сложность двоичного поиска с помощью O-нотации.

Мы не можем использовать запись O(1), так как число шагов алгоритма увеличивается по мере роста объема данных. Но этот алгоритм не вписывается и в категорию O(N), ведь число его шагов намного меньше, чем N элементов данных. Как мы видели, для выполнения двоичного поиска в массиве из 100 элементов нужно всего семь шагов.

Поэтому сложность алгоритма двоичного поиска находится где-то *между* O(1) и O(N). Так какова же она?

Временная сложность двоичного поиска выражается с помощью O-нотации так: O(log N).

Я произношу это как «О от логарифма эн». Алгоритмы такого типа обладают *логарифмической временной сложностью*, то есть выполняются за *логарифмическое время*.

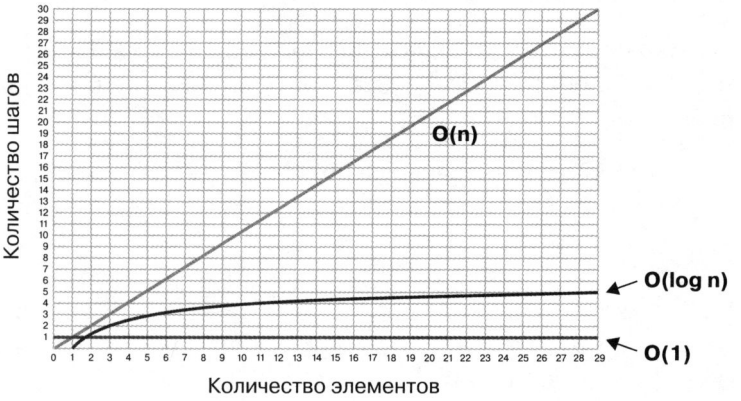
Проще говоря, O(log N) описывает алгоритм, *число шагов в котором увеличивается на единицу при каждом удвоении объема данных*. Как было сказано ранее,

в случае с двоичным поиском дело обстоит именно так. Чуть позже вы поймете, *почему* мы используем именно выражение O(log *N*), но сначала закрепим пройденный материал.

Запишем рассмотренные типы алгоритмов в порядке убывания эффективности:

O(1);
O(log *N*);
O(*N*).

Теперь посмотрим, как это выражается графически:



Обратите внимание, что кривая O(log *N*) растет очень медленно. Это делает алгоритм менее эффективным, чем O(1), но более эффективным, чем O(*N*).

Чтобы понять, почему сложность этого алгоритма выражается как O(log *N*), нужно познакомиться с понятием *логарифма*. Если с этой темой вы знакомы, можете пропустить следующий раздел.

Логарифмы

Давайте выясним, почему сложность таких алгоритмов, как двоичный поиск, записывается как O(log *N*). Что вообще означает log?

Log — это сокращение от слова *logarithm* (логарифм). Первым делом важно усвоить то, что логарифмы никак не связаны с алгоритмами, несмотря на схожесть этих слов.

Нахождение логарифма — это действие, обратное *возведению в степень*.

Например:

$$2^3 \text{ равнозначно } 2 \times 2 \times 2, \text{ что равно } 8.$$

Вычисление $\log_2 8$ — обратное действие. Здесь нужно определить, сколько раз нужно умножить число 2 само на себя, чтобы получить 8.

Для этого 2 нужно умножить само на себя 3 раза, поэтому $\log_2 8 = 3$.

Вот еще один пример:

$$2^6 \text{ равнозначно } 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64.$$

Так как для получения 64 нам пришлось умножить число 2 само на себя шесть раз, $\log_2 64 = 6$.

Выше приведено классическое объяснение логарифмов, но я предпочитаю использовать другое, более простое для понимания, особенно в контексте О-нотации.

Вот как еще можно подойти к вычислению $\log_2 8$: если бы мы продолжали *делить* 8 на 2 вплоть до получения 1, сколько двоек оказалось бы в нашем выражении?

$$8 / 2 / 2 / 2 = 1.$$

Иначе говоря, сколько раз нам нужно разделить 8 пополам, чтобы получить 1? В данном примере — три. Поэтому $\log_2 8 = 3$.

Точно так же можно вычислить и $\log_2 64$: сколько раз нужно разделить 64 пополам, чтобы получить 1?

$$64 / 2 / 2 / 2 / 2 / 2 / 2 = 1.$$

Поскольку у нас получилось шесть двоек, $\log_2 64 = 6$.

Теперь, когда вы узнали, что такое логарифмы, запись $O(\log N)$ обретет для вас смысл.

Значение выражения O(log N)

Вернемся к О-нотации. В мире информатики под выражением $O(\log N)$ подразумевается выражение $O(\log_2 N)$. Мы просто опускаем эту маленькую двойку для удобства.

Напомню, что О-нотация помогает узнать, сколько шагов будет выполнять алгоритм при наличии N элементов данных.

Запись O(log *N*) означает, что при наличии *N* элементов данных алгоритм будет выполнять log_2 *N шагов*. Если элементов 8, то алгоритм будет состоять из трех шагов, так как $\log_2 8 = 3$.

Другими словами, если мы будем последовательно делить 8 элементов пополам, нам понадобится три шага, чтобы получить 1 элемент.

Именно это происходит с бинарным поиском. Чтобы найти конкретный элемент, мы последовательно делим количество ячеек массива пополам, пока не сузим диапазон до предела.

Проще говоря: O(log *N*) *означает, что алгоритм выполняет столько шагов, сколько нужно для того, чтобы остаться с одним элементом в результате последовательного деления объема данных пополам.*

В следующей таблице вы можете увидеть поразительную разницу в эффективности между алгоритмами O(*N*) и O(log *N*):

***N* элементов**	**O(*N*)**	**O(log *N*)**
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10

В случае с алгоритмом O(*N*) число шагов увеличивается на единицу при каждом добавлении элемента, а в случае с алгоритмом O(log *N*) — при каждом удвоении всего объема данных.

В следующих главах вы познакомитесь с алгоритмами, которые не попадают ни в одну из рассмотренных нами трех категорий. А пока применим полученные знания к нескольким примерам кода.

Практические примеры

Вот типичный код на Python для вывода на экран всех элементов списка:

```
things = ['apples', 'baboons', 'cribs', 'dulcimers']

for thing in things:
    print("Here's a thing: %s" % thing)
```

Как описать сложность этого алгоритма с помощью О-нотации?

Во-первых, нужно понять, что мы имеем дело с примером алгоритма. По сути, любой код, который делает хоть что-то, — это уже алгоритм, некий способ выполнения определенной задачи. В нашем случае задача — вывод на экран всех элементов списка. Для ее выполнения мы используем такой алгоритм, как цикл `for` с оператором `print`.

Теперь нужно проанализировать, сколько шагов выполняет этот алгоритм. В нашем примере основная часть алгоритма — цикл `for` — состоит из четырех шагов, потому что в списке содержится четыре элемента, и каждый из них выводится на экран один раз.

Но количество шагов может меняться. Если бы наш список содержал десять элементов, цикл `for` выполнял бы десять шагов. Так как этот цикл `for` выполняет столько шагов, сколько элементов в списке, мы можем сказать, что сложность этого алгоритма равна O(N).

Следующий пример — простой алгоритм на основе Python для определения того, является ли число простым:

```
def is_prime(number):
    for i in range(2, number):
        if number % i == 0:
            return False
    return True
```

Этот код принимает в качестве аргумента число `number` и запускает цикл `for`, в котором делит переданное значение на каждое целое число от 2 до `number` и проверяет результат на наличие остатка. Если остатка нет, значит, число не простое, и код возвращает `False`. Если `number` делится с остатком на все числа диапазона, значит, оно простое, и код возвращает `True`.

Цель в этом примере немного отличается от тех, что были ранее. Раньше нас интересовало количество шагов алгоритма при наличии в массиве N элементов данных. Здесь мы имеем дело не с массивом, а с числом, которое передаем функции, и от величины этого числа зависит то, сколько раз будет выполняться цикл.

Ключевой вопрос в этом случае будет следующим: «Сколько шагов будет выполнять алгоритм при передаче числа N?»

Если мы передадим функции `is_prime` число 7, то цикл `for` будет выполняться примерно семь раз (технически он выполняется пять раз: диапазон чисел, на которые производится деление, начинается с 2 и заканчивается перед заданным числом). В случае с числом 101 цикл выполняется примерно 101 раз. Поскольку

ку количество шагов алгоритма увеличивается на один, когда повышается на единицу величина переданного функции числа, мы имеем дело с классическим примером алгоритма со сложностью O(N).

Опять же, ключевой вопрос здесь относится к N другого типа, ведь нас интересует число, а не массив. В следующих главах мы подробнее поговорим об определении разных типов N.

Выводы

О-нотация дает нам согласованную систему для сравнения любых алгоритмов. С ее помощью мы можем исследовать сценарии из реальной жизни, структуры данных и алгоритмы, чтобы выбрать те, которые позволяют сделать наш код максимально быстрым и способным справляться с большими нагрузками.

В следующей главе мы разберем реальный пример ускорения выполнения кода благодаря использованию О-нотации.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 3».

1. Используйте О-нотацию для описания временной сложности следующей функции, определяющей, високосный ли заданный год:

```
function isLeapYear(year) {
  return (year % 100 === 0) ? (year % 400 === 0) : (year % 4 === 0);
}
```

2. Используйте О-нотацию для описания временной сложности следующей функции, которая суммирует все числа в заданном массиве:

```
function arraySum(array) {
  let sum = 0;

  for(let i = 0; i < array.length; i++) {
    sum += array[i];
  }

  return sum;
}
```

3. Функция ниже основана на аналогии, которая с давних времен используется для описания принципа сложных процентов.

Представьте, что у вас есть шахматная доска и вы кладете на одну клетку одно рисовое зернышко. На вторую клетку вы кладете 2 зернышка, на третью — 4, на четвертую — 8, на пятую — 16 и т. д.

Следующая функция вычисляет номер клетки, на которую вам нужно поместить заданное количество зерен. Например, для 16 зерен функция вернет значение 5, так как на пятую клетку нужно положить именно 16 зерен.

Используйте О-нотацию для описания временной сложности этой функции:

```
function chessboardSpace(numberOfGrains) {
  let chessboardSpaces = 1;
  let placedGrains = 1;

  while (placedGrains < numberOfGrains) {
    placedGrains *= 2;
    chessboardSpaces += 1;
  }

  return chessboardSpaces;
}
```

4. Эта функция принимает массив строк и возвращает новый, содержащий только те строки, которые начинаются с символа `"a"`. Используйте О-нотацию для описания временной сложности этой функции:

```
function selectAStrings(array) {
  let newArray = [];

  for(let i = 0; i < array.length; i++) {
    if (array[i].startsWith("a")) {
      newArray.push(array[i]);
    }
  }

  return newArray;
}
```

5. Следующая функция находит медиану в *упорядоченном* массиве. Используйте О-нотацию для описания ее временной сложности:

```
function median(array) {
  const middle = Math.floor(array.length / 2);

  // Если в массиве четное количество чисел:
  if (array.length % 2 === 0) {
    return (array[middle - 1] + array[middle]) / 2;
  } else { // Если в массиве нечетное количество чисел:
    return array[middle];
  }
}
```

ГЛАВА 4

Оптимизация кода с помощью О-нотации

О-нотация — это отличный инструмент для выражения эффективности алгоритма. Мы уже использовали его для количественной оценки разницы между двоичным и линейным поиском и пришли к выводу, что двоичный поиск с временной сложностью O(log *N*) работает гораздо быстрее, чем линейный, временная сложность которого O(*N*).

Кроме того, О-нотация позволяет сравнить свой алгоритм с *другими распространенными алгоритмами* и выяснить, проигрывает он им в скорости или наоборот.

Если О-нотация пометит ваш алгоритм как «медленный», вы сможете попытаться найти способ оптимизировать его и перевести в категорию более быстрых алгоритмов. Это не всегда возможно, но об этом, безусловно, стоит подумать.

В этой главе мы напишем код для решения практической задачи, а затем измерим временную сложность нашего алгоритма с помощью О-нотации. Затем мы выясним, можно ли как-то оптимизировать этот алгоритм (спойлер: можно).

Пузырьковая сортировка

Но, прежде чем перейти к практической задаче, давайте рассмотрим еще одну категорию алгоритмической эффективности. Чтобы наглядно ее показать, я воспользуюсь одним из классических алгоритмов программирования.

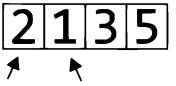
Алгоритмы сортировки многие годы являлись предметом обширных исследований в области информатики, и за это время были разработаны десятки их вариантов. Все они помогают найти ответ на следующий вопрос:

Как отсортировать значения в неотсортированном массиве, расположив их в порядке возрастания?

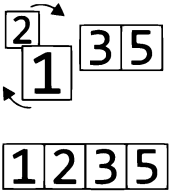
В этой и следующих главах мы рассмотрим целый ряд алгоритмов сортировки. Сначала поговорим об алгоритмах простой сортировки. В них легко разобраться, но они проигрывают в эффективности более быстрым и сложным алгоритмам.

Пузырьковая сортировка, или *сортировка пузырьком*, — это простейший алгоритм сортировки, который выполняет следующие шаги.

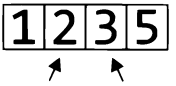
1. Указывает на два соседних значения в массиве, начиная с первых двух. Сравнивает первый элемент со вторым:



2. Если эти два элемента расположены не по порядку (например, левое значение больше правого), меняет их местами (если они в правильном порядке, то ничего не делает):



3. Сдвигает указатели на одну ячейку вправо:



4. Повторяет шаги с 1 по 3, пока не достигнет конца массива. Так выглядит первый *проход* по массиву — последовательное указание на все содержащиеся в нем значения.
5. Затем указатели возвращаются к первым двум значениям и выполняется еще один проход по массиву с повторением шагов с 1 по 4. Эти проходы

повторяются, пока не выяснится, что перестановки значений больше не нужны. Это говорит о том, что наш массив полностью отсортирован и задача выполнена.

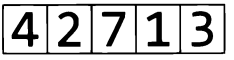
Пузырьковая сортировка в действии

Посмотрим, как работает пузырьковая сортировка на конкретном примере.

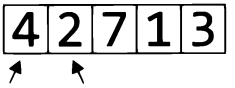
Допустим, мы хотим отсортировать массив `[4, 2, 7, 1, 3]`. Сейчас значения в нем расположены не по порядку, а мы хотим, чтобы числа стояли в порядке возрастания.

Начнем с первого прохода.

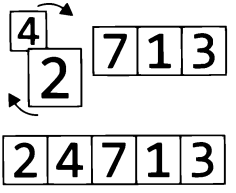
Это наш исходный массив:



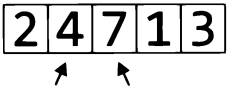
Шаг 1: сравниваем значения 4 и 2:



Шаг 2: они стоят не по порядку, поэтому мы меняем их местами:

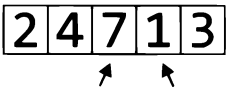


Шаг 3: сравниваем значения 4 и 7:

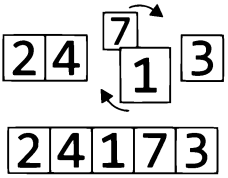


Порядок верный, поэтому нам не нужно менять их местами.

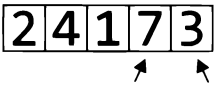
Шаг 4: сравниваем значения 7 и 1:



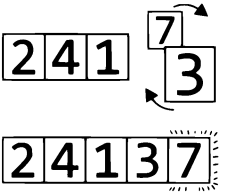
Шаг 5: они стоят не по порядку, поэтому мы меняем их местами:



Шаг 6: сравниваем значения 7 и 3:



Шаг 7: они стоят не по порядку, и мы меняем их местами:



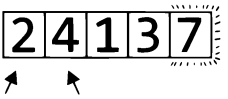
Теперь мы точно знаем, что значение 7 находится в правильной позиции, потому что перемещали его вправо, пока оно не оказалось в самом подходящем месте. На прошлой диаграмме я выделил это место с помощью штрихов, окружающих ячейку с числом 7.

Теперь стало ясно, почему алгоритм называется сортировкой *пузырьком*: при каждом проходе самое большое значение «всплывает» в нужное положение, как пузырек в воде.

Поскольку при выполнении этого прохода мы сделали по крайней мере одну перестановку значений, нужно выполнить еще один проход.

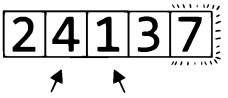
Начинаем второй проход.

Шаг 8: сравниваем значения 2 и 4:

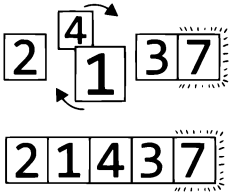


Порядок верный, так что мы можем двигаться дальше.

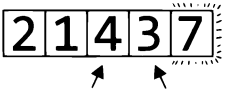
Шаг 9: сравниваем значения 4 и 1:



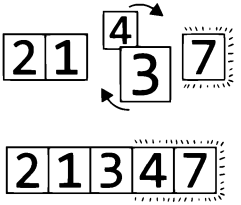
Шаг 10: они стоят не по порядку, поэтому мы меняем их местами:



Шаг 11: сравниваем значения 4 и 3:



Шаг 12: они стоят не по порядку, поэтому мы меняем их местами:



Нам не нужно сравнивать значения 4 и 7, ведь мы знаем, что 7 уже было помещено в правильную позицию во время прошлого прохода. Теперь нам известно, что значение 4 тоже стоит в правильном положении. На этом второй проход заканчивается.

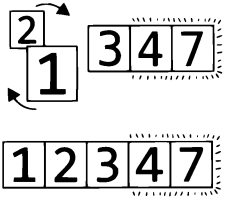
Поскольку при выполнении этого прохода мы сделали по крайней мере одну перестановку значений, нам необходимо выполнить еще один проход.

Начинаем третий проход.

Шаг 13: сравниваем значения 2 и 1:



Шаг 14: они стоят не по порядку, поэтому мы меняем их местами:



Шаг 15: сравниваем значения 2 и 3:



Порядок верный, поэтому в перестановке нет необходимости.

Теперь мы знаем, что значение 3 встало на свое место:



Поскольку при выполнении этого прохода мы сделали по крайней мере одну перестановку значений, нужно выполнить еще один проход.

Начинаем четвертый проход.

Шаг 16: сравниваем значения 1 и 2:



Порядок верный, поэтому нам не нужно менять их местами. Проход можно завершить, так как все оставшиеся значения уже отсортированы.

При выполнении этого прохода мы не сделали ни одной перестановки значений, поэтому знаем, что массив полностью отсортирован:



Программная реализация

Вот как выглядит пузырьковая сортировка на языке Python:

```
def bubble_sort(list):
    unsorted_until_index = len(list) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(unsorted_until_index):
            if list[i] > list[i+1]:
                list[i], list[i+1] = list[i+1], list[i]
                sorted = False
        unsorted_until_index -= 1

return list
```

Чтобы использовать эту функцию, можно передать ей неотсортированный массив:

```
print(bubble_sort([65, 55, 45, 35, 25, 15, 10]))
```

В результате она возвратит отсортированный массив.

Давайте разберем функцию построчно, чтобы увидеть, как она работает. Перед каждой строкой кода я буду приводить объяснение.

Первым делом создаем переменную `unsorted_until_index`. Она будет отслеживать крайний правый индекс массива, который еще *не был* отсортирован. При первом запуске алгоритма массив по определению не отсортирован, поэтому мы инициализируем эту переменную конечным индексом массива:

```
unsorted_until_index = len(list) - 1
```

Создаем переменную `sorted`, которая будет сообщать, отсортирован ли массив. Конечно, при первом запуске это не так, поэтому мы задаем для переменной значение `False`:

```
sorted = False
```

Мы запускаем цикл `while`, который продолжит работать, пока массив не будет полностью отсортирован. Каждое выполнение этого цикла соответствует проходу массива:

```
while not sorted:
```

Предварительно задаем для переменной `sorted` значение `True`:

```
sorted = True
```

Суть в том, что мы предполагаем, что массив отсортирован, до тех пор пока не сталкиваемся с необходимостью в перестановке значений. Тогда мы изменяем значение этой переменной на `False`. Если при проходе всего массива мы не сделаем ни одной перестановки, `sorted` присваивается значение `True` и мы будем знать, что массив полностью отсортирован.

В цикле `while` запускаем цикл `for`, где последовательно указываем на каждую пару значений в массиве. В качестве первого указателя используем переменную `i`, с помощью которой перебираем значения от начала массива до индекса, который еще не был отсортирован:

```
for i in range(unsorted_until_index):
```

В этом цикле мы сравниваем каждую пару соседних значений и меняем их местами, если они стоят не по порядку. При перестановке меняем значение `sorted` на `False`:

```
if list[i] > list[i+1]:
    list[i], list[i+1] = list[i+1], list[i]
    sorted = False
```

После каждого прохода мы точно знаем, что значение, которое мы передвинули ближе к правому краю массива, находится в правильном положении. Мы уменьшаем значение `unsorted_until_index` на 1, поскольку индекс, который отслеживала эта переменная, уже был отсортирован:

```
unsorted_until_index -= 1
```

Цикл `while` завершается, когда `sorted` приобретает значение `True`, показывая нам, что массив полностью отсортирован. После этого функция возвращает отсортированный массив:

```
return list
```

Эффективность пузырьковой сортировки

Алгоритм пузырьковой сортировки состоит из двух типов шагов.

- *Сравнение* — здесь сравниваются два числа и определяется, какое из них больше, а какое меньше.
- *Перестановка* (или *обмен* — *swaps*) — на этом шаге происходит перестановка чисел в порядке возрастания.

Начнем с определения количества *сравнений* в алгоритме пузырьковой сортировки.

В нашем примере массив состоит из пяти элементов. Как вы помните, при первом проходе нам пришлось произвести четыре сравнения пар чисел.

При втором проходе было выполнено только три сравнения. Мы не сравнивали последние два числа, поскольку точно знали, что после первого прохода последнее число оказалось в правильной позиции.

При третьем проходе мы произвели два сравнения, а при четвертом — только одно.

Итак, в общей сложности мы произвели 4 + 3 + 2 + 1 = 10 сравнений.

Мы можем сделать обобщение для массивов всех размеров и сказать, что при наличии N элементов нам необходимо произвести $(N-1) + (N-2) + (N-3) ... + 1$ сравнений.

Теперь, когда мы определили количество сравнений, выполняемых алгоритмом пузырьковой сортировки, определим количество *перестановок*.

В худшем случае, когда массив отсортирован в порядке убывания (полная противоположность тому, что мы хотим), нам действительно потребуется замена для каждого сравнения. В таком случае у нас будет 10 сравнений и 10 перестановок — всего 20 шагов.

Например, в случае с массивом из пяти значений, расположенных в порядке убывания, мы производим 4 + 3 + 2 + 1 = 10 сравнений и 10 перестановок. Итого — 20 шагов.

В случае с массивом из 10 значений мы выполняем 9 + 8 + 7 + 6 + 5 + 4 + 3 + + 2 + 1 = 45 сравнений и еще 45 перестановок — в общей сложности 90 шагов.

Если в массиве 20 значений, мы производим 19 + 18 + 17 + 16 + 15 + 14 + 13 + + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 190 сравнений и примерно 190 перестановок. Итого 380 шагов.

Мы можем сделать вывод, что этот алгоритм неэффективен, потому что по мере увеличения количества элементов число шагов растет *полиномиально*. Это хорошо видно в следующей таблице.

***N* элементов данных**	**Максимальное количество шагов**
5	20
10	90
20	380
40	1560
80	6320

Если вы проанализируете динамику роста числа шагов по мере увеличения N, то поймете, что каждый раз оно увеличивается примерно на N^2.

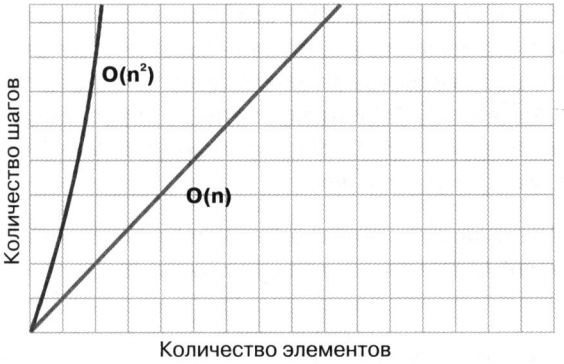
Взгляните на следующую таблицу:

***N* элементов данных**	**Количество шагов алгоритма**	***N*²**
5	20	25
10	90	100
20	380	400
40	1560	1600
80	6320	6400

Выразим временную сложность алгоритма пузырьковой сортировки с помощью О-нотации. Как вы помните, О-нотация позволяет выяснить, сколько шагов будет выполнять алгоритм при наличии N элементов данных.

Поскольку для пузырьковой сортировки N значений нужно N^2 шагов, сложность этого алгоритма будет равна $O(N^2)$.

Алгоритм со сложностью $O(N^2)$ считается относительно неэффективным, так как по мере роста объема данных число шагов увеличивается очень резко. Взгляните на следующий график, где $O(N^2)$ сравнивается с более быстрым алгоритмом $O(N)$:



Обратите внимание на то, как резко растет кривая, отражающая количество шагов алгоритма O(N^2), по сравнению с простой диагональной линией O(N).

Примечание: алгоритм O(N^2) еще называют алгоритмом с *квадратичной временной сложностью* или выполняемым за *квадратичное время*.

Квадратичная задача

Вот практический пример того, как можно заменить медленный алгоритм O(N^2) более быстрым O(N).

Допустим, вы создаете на языке JavaScript приложение, анализирующее оценки, которые люди выставляют продуктам по шкале от 1 до 10. Вы пишете функцию, которая проверяет, есть ли в массиве оценок повторяющиеся значения. Она будет использоваться при выполнении более сложных расчетов в других частях программы.

Например, в массиве `[1, 5, 3, 9, 1, 4]` есть два экземпляра значения `1`, поэтому мы возвращаем `true`, чтобы указать на присутствие в массиве повторяющихся чисел.

Один из первых подходов, который может прийти на ум, — использование вложенных циклов:

```
function hasDuplicateValue(array) {
    for(let i = 0; i < array.length; i++) {
        for(let j = 0; j < array.length; j++) {
            if(i !== j && array[i] === array[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Здесь мы перебираем все значения массива с помощью переменной `i`. Уделив внимание каждому из значений в `i`, мы запускаем *второй* цикл, который перебирает все значения в массиве с помощью переменной `j` и проверяет, совпадают ли значения в позициях `i` и `j`. Если да, значит, в массиве есть повторяющиеся значения и мы возвращаем `true`, если нет — возвращаем `false`, так как точно знаем, что в массиве дубликатов нет.

Конечно, это рабочий подход, но насколько он эффективен? Теперь, когда мы уже немного знаем об О-нотации, попробуем выразить сложность этой функции с помощью нее.

Как вы помните, О-нотация показывает, сколько шагов выполняет алгоритм при наличии N элементов данных. В этом примере нас волнует, сколько шагов будет выполнять наш алгоритм в худшем случае, при наличии N значений, в массиве, передаваемом функции `hasDuplicateValue`?

Чтобы это узнать, нужно выяснить, какие именно шаги выполняет наша функция, и проанализировать худший сценарий.

Прошлая функция выполняет шаги одного типа — *сравнения*. Она последовательно проверяет значения `array[i]` и `array[j]` на предмет их равенства, чтобы выяснить, есть ли дубликаты. В худшем случае, если в массиве нет повторяющихся значений, коду придется выполнить максимальное число итераций циклов и произвести все возможные сравнения, прежде чем возвратить значение `false`.

Исходя из этого, можно сделать вывод, что при N значений в массиве наша функция будет выполнять N^2 сравнений. Это связано с тем, что мы используем внешний цикл, который должен выполнить N итераций, чтобы обработать весь массив, а в *каждой итерации* выполняется *еще* N *итераций* внутреннего цикла. Итого мы имеем $N \times N$ шагов, то есть N^2 шагов, и сложность нашего алгоритма равна $O(N^2)$.

Мы можем доказать, что наша функция выполняет N^2 шагов, добавив в нее фрагмент кода для отслеживания числа шагов алгоритма:

```
function hasDuplicateValue(array) {
    let steps = 0; // количество шагов
    for(let i = 0; i < array.length; i++) {
        for(let j = 0; j < array.length; j++) {
            steps++; // увеличить количество шагов на 1
            if(i !== j && array[i] === array[j]) {
                return true;
            }
        }
    }
    console.log(steps); // вывести количество шагов на экран, если дубликаты
                        // не обнаружены
    return false;
}
```

Этот фрагмент выводит на экран число выполненных шагов при отсутствии в массиве дубликатов. Например, если мы запустим функцию `hasDuplicateValue([1, 4, 5, 2, 9])`, то на консоли JavaScript отобразится число 25, означающее, что при обработке массива из 5 элементов было произведено 25 сравнений. Передавая функции массивы с разным количеством элементов, мы всегда будем получать значение, равное квадрату размера массива. Это классический пример алгоритма $O(N^2)$.

Очень часто (но не всегда) алгоритм, вкладывающий один цикл в другой, имеет сложность $O(N^2)$. Поэтому, когда вы видите вложенный цикл, в вашей голове должен срабатывать сигнал тревоги $O(N^2)$.

То, что временная сложность нашей функции равна $O(N^2)$, должно насторожить нас, потому что алгоритм $O(N^2)$ считается относительно медленным. Когда вы сталкиваетесь с медленным алгоритмом, постарайтесь отыскать более быструю альтернативу. Лучших вариантов *может и не быть*, но в этом нужно убедиться.

Линейное решение

Ниже приведена еще одна реализация функции `hasDuplicateValue` без использования вложенных циклов. Сначала посмотрим, как она работает, а потом выясним, какая из двух реализаций эффективнее.

```
function hasDuplicateValue(array) {
    let existingNumbers = [];
    for(let i = 0; i < array.length; i++) {
        if(existingNumbers[array[i]] === 1) {
            return true;
        } else {
            existingNumbers[array[i]] = 1;
        }
    }
    return false;
}
```

Эта функция создает изначально пустой массив `existingNumbers`.

Далее мы используем цикл для проверки каждого числа в массиве. Встречая число, он помещает произвольное значение (мы выбрали `1`) в `existingNumbers` в позицию с *индексом*, соответствующим числу, которое ей встретилось.

Рассмотрим пример с массивом `[3, 5, 8]`. Сталкиваясь с числом 3, функция помещает в массив `existingNumbers` значение 1 в позицию с индексом 3. В результате массив выглядит примерно так:

```
[undefined, undefined, undefined, 1]
```

Теперь в позиции с индексом 3 находится значение 1. Так мы сможем запомнить, что уже встречали число 3 в нашем массиве.

Когда в ходе очередной итерации цикла мы встречаем в массиве число 5, значение 1 добавляется в позицию с индексом 5, после чего `existingNumbers` выглядит так:

```
[undefined, undefined, undefined, 1, undefined, 1]
```

Когда мы встретим число 8, массив `existingNumbers` будет выглядеть так:

```
[undefined, undefined, undefined, 1, undefined, 1, undefined, undefined, 1]
```

По сути, мы используем индексы массива `existingNumbers`, чтобы запомнить, какие из чисел в нем мы уже встречали.

Но вот в чем хитрость. Прежде чем сохранить 1 в позиции с соответствующим индексом, код *сначала проверяет, нет ли уже там единицы*. Если есть, значит, мы уже сталкивались с этим числом — то есть мы обнаружили дубликат. Тогда просто возвращаем значение `true` и завершаем работу функции. Если мы доходим до конца цикла, не возвратив `true`, значит, дубликатов в массиве нет, и мы возвращаем значение `false`.

Чтобы оценить эффективность нового алгоритма с помощью О-нотации, нам снова нужно определить число шагов, которые он выполняет в худшем сценарии.

Значимый шаг здесь — просмотр каждого числа и проверка того, не является ли значение в позиции с соответствующим индексом в массиве `existingNumbers` равным 1:

```
if(existingNumbers[array[i]] === 1)
```

Помимо сравнений, мы выполняем еще и *вставки* значений в массив `existingNumbers`, но в рамках этого анализа это довольно незначительный шаг. Подробнее об этом мы поговорим в следующей главе.

В худшем случае, когда в массиве не будет дубликатов, наша функция должна будет выполнить весь цикл.

Судя по всему, новый алгоритм выполняет N сравнений при наличии N элементов данных, потому что у нас есть только один цикл, который просто перебирает все числа в массиве. Давайте проверим это предположение, отследив количество выполняемых шагов в консоли JavaScript:

```
function hasDuplicateValue(array) {
    let steps = 0;
    let existingNumbers = [];
    for(let i = 0; i < array.length; i++) {
        steps++;
```

```
            if(existingNumbers[array[i]] === 1) {
                return true;
            } else {
                existingNumbers[array[i]] = 1;
            }
        }
        console.log(steps);
        return false;
    }
```

Теперь при запуске функции `hasDuplicateValue([1, 4, 5, 2, 9])` в консоли JavaScript отобразится значение 5, соответствующее размеру нашего массива. При дальнейшем тестировании мы обнаружим, что это подходит для массивов всех размеров, поэтому временная сложность алгоритма равна O(N).

Зная, что O(N) работает намного быстрее, чем O(N^2), мы выбираем второй подход и существенно оптимизируем функцию `hasDuplicateValue`, обеспечивая *огромный* прирост скорости.

На самом деле у новой реализации есть один недостаток: она потребляет больше памяти. Но пока не беспокойтесь об этом — мы вернемся к этой теме в главе 19.

Выводы

Четкое понимание сути O-нотации помогает выявлять медленный код и выбирать более быстрый из двух конкурирующих алгоритмов.

Но иногда O-нотация может заставить нас подумать, что скорость обоих алгоритмов одинаковая, тогда как один из них на самом деле более быстрый. В следующей главе вы научитесь оценивать эффективность разных алгоритмов, даже когда O-нотации не удается выявить разницу между ними.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 4».

1. Замените вопросительные знаки в таблице ниже количеством шагов алгоритма при заданном количестве элементов данных:

N элементов	O(N)	O(log N)	O(N^2)
100	100	?	?
2000	?	?	?

2. Если во время обработки массива с помощью алгоритма $O(N^2)$ мы выясняем, что при этом он выполняет 256 шагов, то каков размер массива?

3. С помощью О-нотации определите временную сложность следующей функции, которая находит наибольшее произведение любых пар чисел в заданном массиве:

```
def greatestProduct(array):
  greatestProductSoFar = array[0] * array[1]

  for i, iVal in enumerate(array):
    for j, jVal in enumerate(array):
      if i != j and iVal * jVal > greatestProductSoFar:
        greatestProductSoFar = iVal * jVal

  return greatestProductSoFar
```

4. Следующая функция находит наибольшее число в массиве, но имеет сложность $O(N^2)$. Перепишите эту функцию так, чтобы ее сложность была $O(N)$:

```
def greatestNumber(array):
  for i in array:
    # Допустим, значение i самое большое:
    isIValTheGreatest = True

    for j in array:
      # Если мы обнаруживаем большее, чем i, значение,
      # значит, i не самое большое:
      if j > i:
        isIValTheGreatest = False

    # Если после проверки остальных чисел i все еще наибольшее,
    # значит, i - самое большое число в массиве:
    if isIValTheGreatest:
      return i
```

ГЛАВА 5

Оптимизация кода с О-нотацией и без нее

Мы уже убедились, что О-нотация — отличный инструмент для сравнения алгоритмов и определения наиболее подходящего в определенной ситуации. Но, конечно же, это не *единственный* инструмент. Бывают случаи, когда два конкурирующих алгоритма описываются с помощью О-нотации одинаково, но на самом деле один из них работает быстрее, чем другой.

В этой главе вы научитесь различать алгоритмы, которые *кажутся* одинаково эффективными, и выбирать самый быстрый из них.

Сортировка выбором

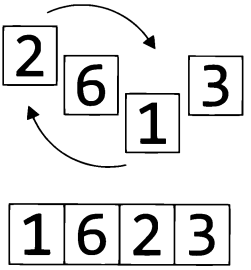
В прошлой главе мы рассмотрели алгоритм пузырьковой сортировки со сложностью O(N^2). Теперь мы познакомимся с сортировкой выбором и посмотрим, насколько эффективной она окажется.

Сортировка выбором состоит из следующих шагов.

1. Проверяем все ячейки массива слева направо в поисках наименьшего значения. Переходя от ячейки к ячейке, мы отслеживаем самое низкое значение, с которым столкнулись до сих пор (сохраняя соответствующий индекс в переменной). Если в какой-то из ячеек мы обнаруживаем значение меньше того, которое хранится в нашей переменной, мы сохраняем в переменной новый индекс. См. следующую схему:



2. Выяснив позицию с наименьшим значением, мы меняем его местами со значением, соответствующим индексу, с которого мы начали проход массива. При первом проходе это будет 0, при втором — 1 и т. д. Ниже показана перестановка при первом проходе.



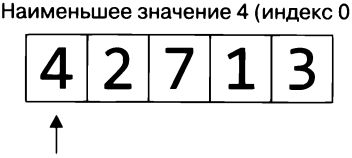
3. Каждый проход предполагает выполнение шагов 1 и 2. Мы повторяем проходы до тех пор, пока не достигнем того, который должен начинаться с последнего индекса массива. К этому моменту массив будет полностью отсортирован.

Сортировка выбором в действии

Для сортировки массива используем этот массив: `[4, 2, 7, 1, 3]`.

Начинаем первый проход.

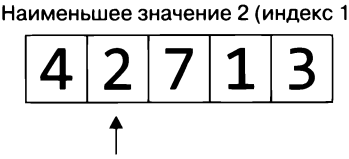
Сначала проверяем значение в позиции с индексом 0. По определению, оно будет наименьшим из тех, которые нам встречались до сих пор (кроме него, мы еще ни одного не видели). Итак, сохраняем его индекс в переменной:



Шаг 1: сравниваем значение 2 с наименьшим на данный момент значением (с 4):



Значение 2 меньше 4, следовательно, оно становится наименьшим:



Шаг 2: сравниваем следующее значение 7 с наименьшим на данный момент. 7 больше 2, поэтому 2 остается наименьшим:



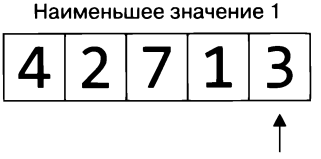
Шаг 3: сравниваем 1 с наименьшим значением:



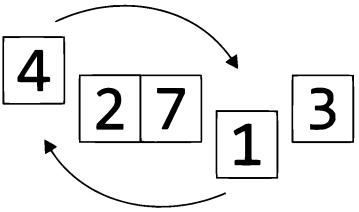
Значение 1 меньше 2, поэтому 1 становится новым наименьшим значением:



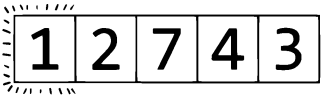
Шаг 4: сравниваем 3 с наименьшим значением, которое теперь равно 1. Мы достигли конца массива и выяснили, что его наименьшее значение — 1:



Шаг 5: поскольку 1 — наименьшее значение, мы меняем его местами с числом в позиции с индексом 0, с которого мы начали проход массива:



После перемещения в начало массива наименьшее значение оказалось в правильной позиции:



Теперь мы готовы начать второй проход.

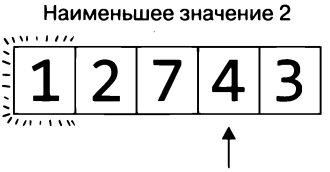
Первая ячейка с индексом 0 уже отсортирована, поэтому проход начинается со следующей ячейки с индексом 1. В ней содержится значение 2, наименьшее на данный момент:



Шаг 6: сравниваем 7 с наименьшим значением. 2 меньше 7, поэтому 2 остается наименьшим:



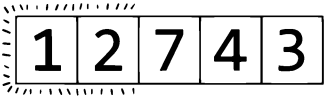
Шаг 7: сравниваем 4 с наименьшим значением. 2 меньше 4, поэтому все остается по-прежнему:



Шаг 8: сравниваем 3 с наименьшим значением. 2 меньше 3, поэтому 2 остается наименьшим значением:



Мы достигли конца массива. Наименьшее значение изначально находилось в правильной позиции, поэтому в перестановке нет необходимости. На этом заканчивается второй проход, в результате которого массив выглядит так:



Теперь выполняем третий проход.

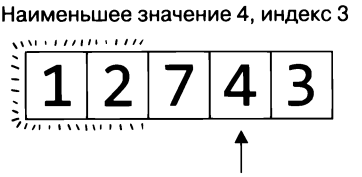
Начинаем с ячейки с индексом 2, которая содержит значение 7. Итак, 7 — изначально наименьшее значение:



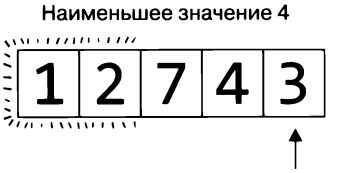
Шаг 9: сравниваем 4 с 7:



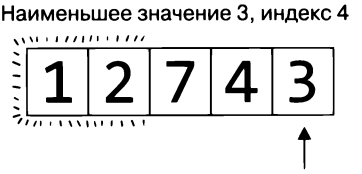
Теперь наименьшим значением становится 4:



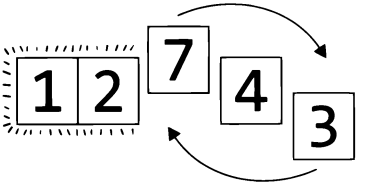
Шаг 10: встречаем значение 3, которое меньше 4:



Новым наименьшим значением становится 3:



Шаг 11: достигнув конца массива, меняем местами значения 3 и 7:

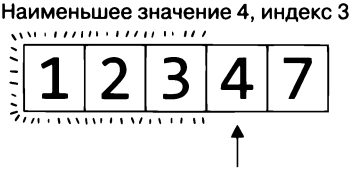


Теперь мы точно знаем, что 3 находится в правильной позиции:



Хотя мы видим, что массив уже отсортирован, *компьютер* еще не знает об этом, поэтому он должен выполнить четвертый проход.

Этот проход начинается с индекса 3. Наименьшее значение на данный момент — 4:



Шаг 12: сравниваем 7 и 4:



При выполнении этого прохода значение 4 остается наименьшим и нам не нужно переставлять его, ведь оно изначально находится в правильной позиции.

Поскольку все ячейки, кроме последней, уже отсортированы, она тоже должна находиться в правильной позиции, а значит, весь массив отсортирован:



Программная реализация

Так выглядит реализация сортировки выбором на языке JavaScript:

```
function selectionSort(array) {
  for(let i = 0; i < array.length - 1; i++) {
    let lowestNumberIndex = i;
    for(let j = i + 1; j < array.length; j++) {
      if(array[j] < array[lowestNumberIndex]) {
        lowestNumberIndex = j;
      }
    }

    if(lowestNumberIndex != i) {
      let temp = array[i];
      array[i] = array[lowestNumberIndex];
      array[lowestNumberIndex] = temp;
    }
  }
  return array;
}
```

Разберем этот код построчно.

Запускаем цикл, который отвечает за проходы массива. С помощью переменной `i` в качестве указателя он перебирает все значения массива до предпоследнего включительно:

```
for(let i = 0; i < array.length - 1; i++) {
```

Нам не придется запускать этот цикл для обработки последнего значения, так как к этому моменту массив уже будет полностью отсортирован.

Начинаем отслеживать *индекс* позиции, в которой находится наименьшее на данный момент значение:

```
let lowestNumberIndex = i;
```

В начале первого прохода индекс `lowestNumberIndex` будет равен 0, в начале второго — 1 и т. д.

Мы специально отслеживаем этот индекс, потому что далее в коде нам нужно будет получить доступ к наименьшему значению и к его индексу, а с помощью этого индекса мы можем ссылаться и на то, и на другое (для проверки наименьшего значения достаточно вызвать `array[lowestNumberIndex]`).

При выполнении каждого прохода проверяем все оставшиеся значения массива на предмет наличия среди них наименьшего по сравнению с текущим наименьшим значением:

```
for(let j = i + 1; j < array.length; j++) {
```

Если такое значение обнаружено, сохраняем его индекс в переменной `lowestNumberIndex`:

```
if(array[j] < array[lowestNumberIndex]) {
  lowestNumberIndex = j;
}
```

К моменту окончания работы внутреннего цикла мы будем знать индекс наименьшего значения для данного прохода.

Если наименьшее значение уже стоит в правильной позиции (что может быть, если это то значение, с которого начинается выполнение прохода), нам не нужно ничего делать. Но если оно находится *не на месте*, нам придется выполнить перестановку — поменять местами наименьшее значение и значение в позиции с индексом `i`, с которого началось выполнение прохода:

```
if(lowestNumberIndex != i) {
  let temp = array[i];
  array[i] = array[lowestNumberIndex];
  array[lowestNumberIndex] = temp;
}
```

Наконец, возвращаем отсортированный массив:

```
return array;
```

Эффективность сортировки выбором

Алгоритм сортировки выбором предусматривает два типа шагов: сравнения и перестановки. Во время каждого прохода мы сравниваем все значения с наименьшим на данный момент и при необходимости переставляем наименьшее значение в правильную позицию.

В примере с массивом из пяти элементов нам пришлось выполнить в общей сложности десять сравнений (см. следующую таблицу):

Проход №	Количество сравнений
1	4
2	3
3	2
4	1

В итоге получается 4 + 3 + 2 + 1 = 10 сравнений.

Сделаем обобщение для массивов всех размеров, указав, что при наличии N элементов нужно произвести:

$$(N - 1) + (N - 2) + (N - 3) \ldots + 1 \text{ сравнений.}$$

Что касается *перестановок*, то за один проход нужно выполнить максимум одну, потому что при каждом проходе мы меняем значения местами либо один раз, либо ни разу, в зависимости от расположения наименьшего. Сравните этот процесс с пузырьковой сортировкой, где в худшем случае (при обработке массива со значениями, которые стоят в порядке убывания) нам пришлось бы выполнять перестановку после *каждого* сравнения.

Вот наглядное сравнение этих двух алгоритмов сортировки:

N элементов	Максимальное число шагов при пузырьковой сортировке	Максимальное количество шагов при сортировке выбором
5	20	14 (10 сравнений + 4 перестановки)
10	90	54 (45 сравнений + 9 перестановок)
20	380	199 (180 сравнений + 19 перестановок)
40	1560	819 (780 сравнений + 39 перестановок)
80	6320	3239 (3160 сравнений + 79 перестановок)

Очевидно, что в сортировке выбором примерно вдвое меньше шагов, чем в пузырьковой, а значит, первый алгоритм выполняется в два раза быстрее.

Игнорирование констант

Но вот что интересно: оба алгоритма описываются с помощью О-нотации *совершенно одинаково*.

Как вы помните, О-нотация помогает определить, сколько шагов будет выполнять алгоритм при наличии N элементов данных. Поскольку при сортировке выбором количество шагов примерно в два раза меньше, чем N^2, было бы логично выразить эффективность этого алгоритма как $O(N^2/2)$. То есть при наличии N элементов данных этот алгоритм выполняет $N^2/2$ шагов. Следующая таблица подтверждает это:

N элементов	$N^2/2$	Максимальное количество шагов при сортировке выбором
5	$5^2/2 = 12{,}5$	14
10	$10^2/2 = 50$	54
20	$20^2/2 = 200$	199
40	$40^2/2 = 800$	819
80	$80^2/2 = 3200$	3239

Но реальность такова, что временная сложность сортировки выбором с помощью O-нотации обозначается так: O(N^2), как и временная сложность пузырьковой сортировки. Это объясняется главным правилом, о котором я еще не упоминал: *O-нотация игнорирует константы.*

Это значит, что O-нотация не учитывает никакие числа, кроме показателей степени, — все остальные просто исключаются из выражения.

Поэтому в нашем случае, несмотря на то что алгоритм выполняет $N^2/2$ шагов, мы отбрасываем часть «/ 2» и выражаем эффективность алгоритма как O(N^2).

Вот еще несколько примеров.

Если бы алгоритм выполнял $N/2$ шагов, мы бы сказали, что его сложность равна O(N).

Сложность алгоритма, выполняющего $N^2 + 10$ шагов, была бы выражена в виде O(N^2), так как число 10 было бы отброшено.

Если бы алгоритм выполнял $2N$ шагов (то есть $N \times 2$), мы бы снова исключили константу и сказали бы, что сложность алгоритма равна O(N).

В соответствии с этим правилом, даже сложность алгоритма O(100N), который работает *в 100 раз медленнее, чем O(N)*, будет считаться равной O(N).

На первый взгляд может показаться, что это правило делает O-нотацию совершенно бесполезной, поскольку один из двух одинаково описанных алгоритмов может оказаться *в 100 раз быстрее* другого. Именно так и произошло в примере с сортировкой выбором и пузырьковой: оба описываются как O(N^2), но первая выполняется в два раза быстрее.

Что же нам это дает?

Категории алгоритмической сложности в О-нотации

Все это приводит нас к важному выводу: О-нотация имеет дело только с *общими категориями* алгоритмической сложности.

Приведу в пример здания. Есть много разных типов зданий: одноэтажные, двухэтажные и трехэтажные дома на одну семью, высотные многоквартирные дома с разной этажностью, небоскребы разной высоты и формы.

Если бы мы сравнивали обычный частный дом с небоскребом, то говорить о количестве этажей было бы бессмысленно. Эти две постройки так сильно различаются по размерам и функциям, что нам не пришлось бы говорить: «Это двухэтажный дом, а это — стоэтажный небоскреб». Мы могли бы просто назвать одно здание домом, а другое — небоскребом. То есть для обозначения огромных различий между ними достаточно указать общие категории, к которым они относятся.

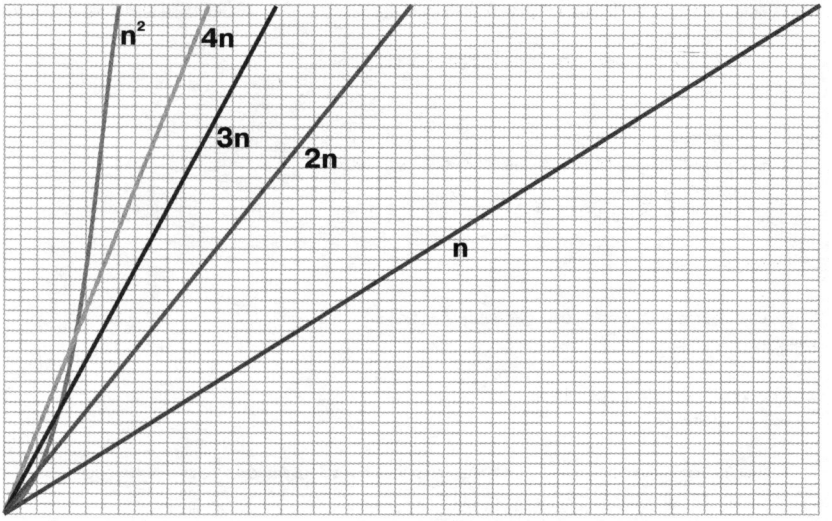
Это же относится и к эффективности алгоритмов. Например, если мы сравниваем $O(N)$ с $O(N^2)$, то нам в принципе не важно, какая у первого алгоритма фактическая сложность: $O(2N)$, $O(N/2)$ или даже $O(100N)$, потому что на фоне огромной разницы между общими категориями эти различия несущественны.

Теперь разберемся, почему $O(N)$ и $O(N^2)$ считаются двумя отдельными категориями, а $O(N)$ и $O(100N)$ — нет.

Как я упоминал в разделе «Суть О-нотации», О-нотация описывает не просто количество шагов алгоритма, а то, как оно изменяется по мере увеличения объема данных. $O(N)$ означает, что число шагов растет пропорционально объему данных. И это верно даже для алгоритмов, выполняющих $100N$ шагов. А вот $O(N^2)$ говорит об полиномиальном росте.

Алгоритмы $O(N^2)$ и $O(N)$ относятся к совершенно разным категориям. Это становится очевидно, если учесть, что по мере роста объема данных в какой-то момент алгоритм $O(N^2)$ начинает работать медленнее, чем $O(N)$, *вне зависимости* от коэффициента перед N.

Следующий график отлично иллюстрирует все вышесказанное:



Поэтому при сравнении показателей эффективности двух алгоритмов, относящихся к двум разным категориям сложности, достаточно определить эти общие категории. Сравнивать алгоритм O(2*N*) с O(N^2) — это все равно, что сравнивать двухэтажный дом с небоскребом. Мы можем просто сказать, что O(2*N*) относится к категории O(*N*).

Все типы алгоритмов, с которыми мы уже познакомились, будь то O(1), O(log *N*), O(*N*), O(N^2), и которые вы увидите далее, представляют собой основные категории алгоритмической сложности, которые очень сильно отличаются друг от друга. Умножение или деление количества шагов алгоритма на любую константу не позволяет перевести его из одной категории в другую.

Но если два алгоритма попадают *в одну* категорию, это не значит, что они работают с одинаковой скоростью. В конце концов, пузырьковая сортировка в два раза медленнее, чем сортировка выбором, хотя обе относятся к категории O(N^2). Поэтому, несмотря на то что О-нотация идеально подходит для сравнения алгоритмов из разных категорий, когда два алгоритма попадают *в одну* категорию, для определения самого быстрого из них придется провести дополнительный анализ.

Практический пример

Рассмотрим самый первый пример кода из главы 1 с небольшими изменениями:

```
def print_numbers_version_one(upperLimit):
  number = 2

  while number <= upperLimit:
    # Если число четное, вывести его на экран:
    if number % 2 == 0:
        print(number)

    number += 1

def print_numbers_version_two(upperLimit):
  number = 2

  while number <= upperLimit:
    print(number)

    # Увеличить число на 2, чтобы получить следующее четное число
    number += 2
```

Здесь мы используем два алгоритма для решения одной задачи — вывода на экран всех четных чисел, начиная с 2 и заканчивая некоторым значением `upperLimit` (в главе 1 этим верхним пределом было число 100, а здесь пользователь может задать его самостоятельно).

В главе 1 я отметил, что первая версия кода требует вдвое больше шагов по сравнению со второй. Теперь попробуем выразить это с помощью О-нотации.

Как уже не раз говорилось, О-нотация помогает определить, сколько шагов будет выполнять алгоритм при наличии N элементов данных. Но здесь N — это не размер массива, а просто число, которое мы передаем в функцию в качестве верхнего предела.

Первая версия кода выполняет примерно N шагов. То есть если значение `upperLimit` равно 100, функция выполняет около 100 шагов (на самом деле она выполняет 99, ведь отсчет начинается с 2). Поэтому мы можем с уверенностью сказать, что временная сложность первого алгоритма равна $O(N)$.

Вторая версия кода выполняет $N/2$ шагов. Получается, что при значении `upperLimit`, равном 100, функция выполняет всего 50 шагов. Как бы нам ни хотелось обозначить сложность этого алгоритма как $O(N/2)$, мы отбрасываем константу и сводим выражение к $O(N)$.

Конечно, вторая версия кода работает в два раза быстрее первой, и лучше выбрать ее. Это еще один пример, показывающий, что при сравнении двух алгоритмов с одинаковой временной сложностью, выраженной с помощью

О-нотации, придется провести дополнительный анализ для определения скорости каждого из них.

Значимые шаги

Вернемся к прошлому примеру и проанализируем еще кое-что. Как мы уже сказали, первая версия кода, `print_numbers_version_one`, выполняет N шагов. Это связано с тем, что цикл выполняется N раз, где N — значение верхнего предела.

Но действительно ли эта функция выполняет N шагов?

Если мы начнем разбираться, то увидим, что в рамках каждой итерации цикла выполняется *несколько* шагов.

Во-первых, у нас есть шаг сравнения (`if number % 2 == 0`), с помощью которого мы проверяем, делится ли число `number` на 2 без остатка. Этот шаг выполняется в рамках каждой итерации цикла.

Во-вторых, есть шаг вывода на экран (`print(number)`), который выполняется только с четными числами, то есть в рамках *каждой второй* итерации цикла.

И в-третьих, у нас есть шаг `number += 1`, который выполняется в рамках каждой итерации цикла.

Я уже как-то говорил, что нам еще предстоит разобраться в определении значимых шагов — таких, которые нужно учитывать при оценке алгоритма с помощью О-нотации. Итак, какие из шагов можно считать значимыми в нашем случае? Что нас интересует больше всего — сравнение, вывод значения на экран или инкрементирование числа?

Для нас важны *все* шаги. Просто при выражении их количества с помощью О-нотации мы отбрасываем константы, упрощая тем самым выражение.

Посмотрим, как это происходит. Если мы подсчитаем все шаги кода из нашего примера, то получим N сравнений, N инкрементаций и $N/2$ выводов на экран. Итого — $2{,}5N$ шагов. Но, отбросив коэффициент 2,5, мы получим выражение $O(N)$. Так какой же шаг был самым значимым? Все. Просто, отбрасывая константу, мы фокусируемся на количестве итераций цикла, а не на подробностях того, что происходит внутри него.

Выводы

Теперь у нас в арсенале есть несколько мощных аналитических инструментов. Мы можем использовать О-нотацию для определения общей эффективности

алгоритма и сравнения двух алгоритмов, относящихся к одной категории сложности.

Но при сравнении эффективности двух алгоритмов нужно учитывать еще один важный фактор. До сих пор мы фокусировались на скорости работы алгоритма в худшем сценарии. Но самое плохое не происходит постоянно. Большинство случаев — что-то среднее между худшим и лучшим сценариями. В следующей главе мы поговорим о том, как можно учесть все варианты.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 5».

1. С помощью О-нотации опишите временную сложность алгоритма, который выполняет $4N + 16$ шагов.
2. С помощью О-нотации опишите временную сложность алгоритма, который выполняет $2N^2$ шагов.
3. С помощью О-нотации опишите временную сложность функции, которая возвращает сумму всех чисел в массиве после их удвоения:

```
def double_then_sum(array)
  doubled_array = []

  array.each do |number|
    doubled_array << number *= 2
  end

  sum = 0

  doubled_array.each do |number|
    sum += number
  end

  return sum
end
```

4. С помощью О-нотации опишите временную сложность функции, которая принимает массив строк и выводит на экран каждую в разных регистрах:

```
def multiple_cases(array)
  array.each do |string|
    puts string.upcase
    puts string.downcase
    puts string.capitalize
  end
end
```

5. Следующая функция перебирает массив чисел и выводит на экран результат сложения каждого с четным *индексом* со всеми числами в массиве. Какова эффективность этой функции с точки зрения O-нотации?

```
def every_other(array)
  array.each_with_index do |number, index|
    if index.even?
      array.each do |other_number|
        puts number + other_number
      end
    end
  end
end
```

ГЛАВА 6

Повышение эффективности с учетом оптимистичных сценариев

До сих пор мы концентрировались лишь на том, сколько шагов будет выполнять алгоритм при самых плохих обстоятельствах. Причина довольно проста: хочешь лучшего — готовься к худшему.

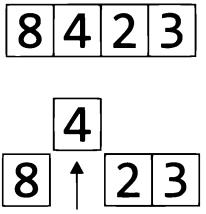
Но в этой главе вы увидите, что *не только* худший сценарий достоин внимания. Способность учитывать *все* варианты развития событий — это важный навык, благодаря которому вам будет легче выбрать подходящий алгоритм для конкретной ситуации.

Сортировка вставками

Мы уже рассмотрели два алгоритма сортировки: пузырьковую и сортировку выбором. Оба с временной сложностью $O(N^2)$, но второй работает в два раза быстрее. Пришло время познакомиться с третьим алгоритмом — сортировкой вставками — и узнать, что важно анализировать все сценарии, а не только худший.

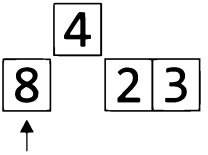
Алгоритм сортировки вставками включает следующие шаги.

1. При первом проходе мы удаляем значение с индексом 1 (вторая ячейка) и сохраняем его во временной переменной, образуя в этой позиции пробел:

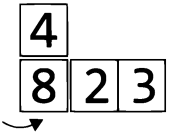


Во время следующих подходов мы удаляем значения в последующих ячейках.

2. Начинается фаза сдвига, в ходе которой мы сравниваем каждое значение слева от образовавшегося пробела со значением во временной переменной:

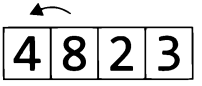


Если значение слева от пробела больше того, которое хранится во временной переменной, мы сдвигаем его вправо:



При перемещении значения вправо пробел «смещается» влево. Когда нам попадается значение меньше того, которое хранится во временной переменной, или мы достигаем левого конца массива, фаза сдвига завершается.

3. Помещаем временно удаленное значение в текущий пробел:

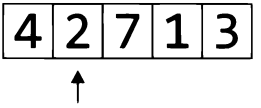


4. Один проход содержит в себе шаги с 1 по 3. Мы повторяем проходы до тех пор, пока не достигнем того, который будет начинаться с конечного индекса массива. К этому моменту массив будет полностью отсортирован.

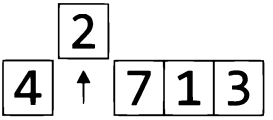
Сортировка вставками в действии

Теперь применим этот алгоритм сортировки к массиву `[4, 2, 7, 1, 3]`.

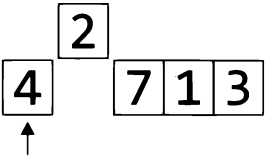
Начинаем первый проход с проверки значения с индексом 1. В нашем случае это число 2.



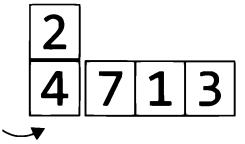
Шаг 1: временно удаляем 2 и сохраняем это значение внутри переменной `temp_value`. На диаграмме эта операция выглядит как сдвиг значения вверх относительно остальной части массива:



Шаг 2: сравниваем 4 со значением `temp_value`, которое равно 2:

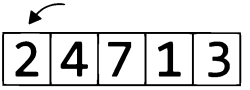


Шаг 3: поскольку 4 больше 2, сдвигаем 4 вправо:



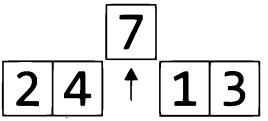
Больше сдвигать нечего, так как пробел достиг левого конца массива.

Шаг 4: заполняем пробел значением `temp_value`, завершая первый проход массива:

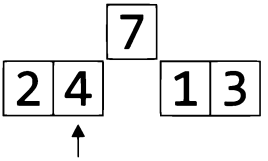


Начинаем второй проход.

Шаг 5: временно удаляем значение с индексом 2, сохраняя его в переменной `temp_value`. Здесь оно равно 7:

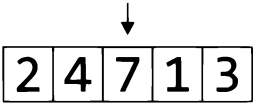


Шаг 6: сравниваем 4 со значением `temp_value`:



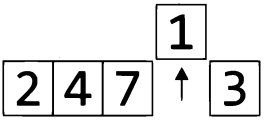
Значение 4 меньше 7, поэтому мы его не сдвигаем. Мы достигли значения, которое оказалось меньше, чем `temp_value`. Эта фаза сдвига завершена.

Шаг 7: помещаем значение `temp_value` обратно в пробел, заканчивая второй проход:

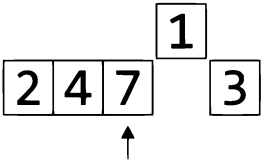


Начинаем третий проход.

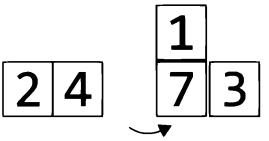
Шаг 8: временно удаляем значение 1 и сохраняем его в `temp_value`:



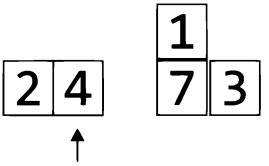
Шаг 9: сравниваем 7 со значением `temp_value`:



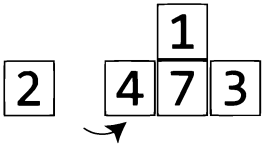
Шаг 10: 7 больше 1, поэтому сдвигаем 7 вправо:



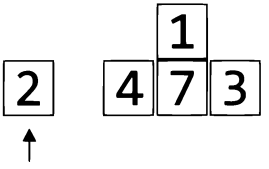
Шаг 11: сравниваем 4 со значением `temp_value`:



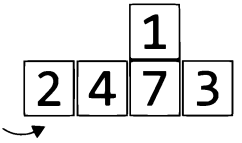
Шаг 12: 4 больше 1, поэтому сдвигаем 4 вправо:



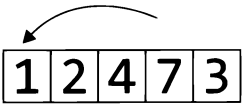
Шаг 13: сравниваем 2 со значением `temp_value`:



Шаг 14: 2 больше 1, поэтому сдвигаем это значение вправо:

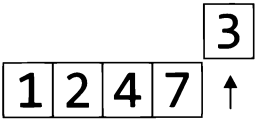


Шаг 15: пробел достиг левого конца массива. Заполняем его значением `temp_value`, завершая третий проход.

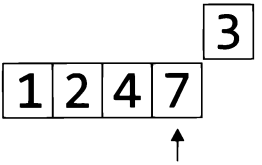


Начинаем четвертый проход.

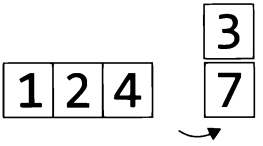
Шаг 16: временно удаляем значение с индексом 4, сохраняя его в `temp_value`. В данном случае это значение 3:



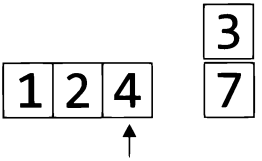
Шаг 17: сравниваем 7 со значением `temp_value`:



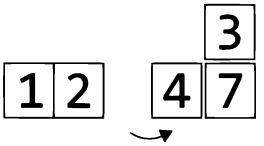
Шаг 18: 7 больше 3, поэтому мы сдвигаем 7 вправо:



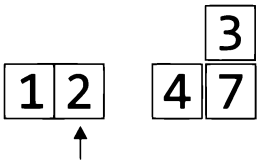
Шаг 19: сравниваем 4 со значением `temp_value`:



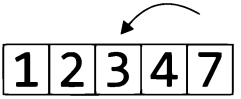
Шаг 20: 4 больше 3, поэтому сдвигаем 4 вправо:



Шаг 21: сравниваем 2 со значением `temp_value`. 2 меньше 3, поэтому фаза сдвига завершается.



Шаг 22: заполняем пробел значением `temp_value`.



Теперь наш массив полностью отсортирован:



Программная реализация

Так выглядит реализация алгоритма сортировки вставками на языке Python:

```
def insertion_sort(array):
  for index in range(1, len(array)):

    temp_value = array[index]
    position = index - 1

    while position >= 0:
      if array[position] > temp_value:
        array[position + 1] = array[position]
        position = position - 1
      else:
        break
```

```
        array[position + 1] = temp_value
    return array
```

Разберем код пошагово.

Начиная с индекса 1, мы запускаем цикл, который обрабатывает весь массив. Каждая итерация — это проход по массиву:

```
for index in range(1, len(array)):
```

В рамках каждого прохода мы сохраняем «удаленное» значение в переменной `temp_value`:

```
temp_value = array[index]
```

Создаем переменную `position`, которая начинается сразу слева от индекса `temp_value`. С помощью нее мы будем обрабатывать значения, сравнивая каждое из них со значением `temp_value`:

```
position = index - 1
```

Выполняя проход, мы будем перебирать эти позиции справа налево, сравнивая значения в них с `temp_value`.

Теперь запускаем внутренний цикл `while`, который будет выполняться, пока значение `position` больше или равно 0:

```
while position >= 0:
```

Выполняем сравнение — проверяем, превышает ли значение `position` значение `temp_value`:

```
if array[position] > temp_value:
```

Если да, то сдвигаем соответствующее левое значение вправо:

```
array[position + 1] = array[position]
```

Уменьшаем значение `position` на 1, чтобы сравнить значение левее с `temp_value` в рамках следующей итерации цикла `while`:

```
position = position - 1
```

Если в какой-то момент значение `position` будет меньше или равно `temp_value`, мы должны быть готовы завершить проход и заполнить пробел значением `temp_value`:

```
else:
  break
```

Последний шаг каждого прохода — заполнение пробела значением `temp_value`:

```
array[position + 1] = temp_value
```

После выполнения всех проходов мы возвращаем отсортированный массив:

```
return array
```

Эффективность сортировки вставками

В сортировке вставками выполняется четыре типа шагов: удаление, сравнение, сдвиг и вставка. Чтобы узнать, насколько алгоритм эффективен, нужно подсчитать количество этих шагов.

Начнем со сравнения. Оно происходит каждый раз, когда мы сопоставляем значение слева от пробела с `temp_value`. В худшем случае, когда массив отсортирован в обратном порядке, мы должны сравнивать каждое число слева от `temp_value` с этой же переменной при каждом проходе. Это связано с тем, что все значения слева от `temp_value` будут превышать ее значение и проход завершится, только когда пробел достигнет левого конца массива.

Во время первого прохода, когда в `temp_value` хранится значение с индексом 1, выполняется максимум одно сравнение, так как слева от переменной стоит только одно значение. При втором проходе выполняется два сравнения и т. д. При последнем проходе нам нужно сравнить `temp_value` с каждым значением в массиве, кроме содержимого исключенной ячейки. Проще говоря, если в массиве N элементов, при его последнем проходе выполняется максимум $N - 1$ сравнений.

Поэтому общее количество сравнений составляет:

$$1 + 2 + 3 + ... + (N - 1).$$

В нашем примере с массивом из пяти элементов максимальное количество сравнений составляет:

$$1 + 2 + 3 + 4 = 10.$$

Для массива из десяти элементов:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45.$$

Для массива из 20 элементов — 190 сравнений и т. д.

Наблюдая эту закономерность, мы можем сделать вывод, что максимальное количество сравнений для массива из N элементов — примерно $N^2/2$ ($10^2/2$ равно 50, а $20^2/2$ — 200. Подробнее об этом мы поговорим в следующей главе).

Теперь проанализируем другие типы шагов. Сдвиг происходит всякий раз, когда мы перемещаем значение на одну ячейку вправо. Когда массив отсортирован в обратном порядке, число сдвигов будет совпадать с количеством сравнений, так как нам придется перемещать значение вправо после каждого.

Сложив количество сравнений и сдвигов в худшем сценарии, мы получим:

$N^2/2$ сравнений

$+ N^2/2$ сдвигов.

Итого: N^2 шагов.

Удаление и вставка `temp_value` происходит один раз за проход. Поскольку число проходов всегда равно $N - 1$, количество удалений и вставок составляет $N - 1$.

Итак, мы имеем:

N^2 сравнений и сдвигов, вместе взятых;

$N - 1$ удалений;

$+ N - 1$ вставок.

Итого: $N^2 + 2N - 2$ шагов.

Как вы помните, O-нотация игнорирует константы. Опираясь на это правило, мы могли бы упростить полученное выражение до $O(N^2 + N)$.

Но у O-нотации есть еще одно важное правило:

Она учитывает только слагаемое самого высокого порядка.

То есть если алгоритм выполняет $N^4 + N^3 + N^2 + N$ шагов, мы считаем значимым только N^4 и заключаем, что временная сложность алгоритма равна $O(N^4)$. Почему?

Взгляните на следующую таблицу:

N	N^2	N^3	N^4
2	4	8	16
5	25	125	625
10	100	1000	10 000
100	10 000	1 000 000	100 000 000

По мере роста N значение N^4 становится настолько более значимым, чем остальные порядки N, что ими можно пренебречь. Например, если мы сложим

$N^4 + N^3 + N^2 + N$, подставив значения из нижней строки таблицы, то в сумме получим 101 010 100. Но мы вполне можем округлить это число до 100 000 000, проигнорировав более низкие порядки N.

Такой же подход мы можем применить и к сортировке вставками. Несмотря на то что мы уже упростили ее до $N^2 + N$, мы можем отбросить слагаемое более низкого порядка и сократить выражение до $O(N^2)$.

Итак, в худшем случае временная сложность сортировки вставками такая же, как и у пузырьковой и сортировки выбором — $O(N^2)$.

Ранее я сказал, что, несмотря на одинаковую оценку сложности, сортировка выбором выполняется быстрее, чем пузырьковая, поскольку первая требует $N^2/2$ шагов, а вторая — N^2. Может показаться, что сортировка вставками работает так же медленно, как и пузырьком, поскольку на ее выполнение тоже нужно примерно N^2 шагов.

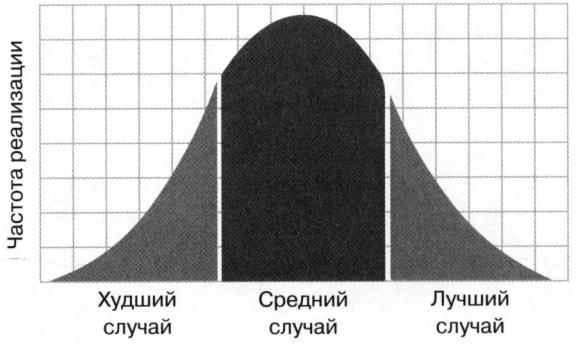
Если вы прекратите читать на этом месте, то будете считать сортировку выбором самой эффективной из трех рассмотренных, поскольку она выполняется в два раза быстрее. На самом деле все не так просто.

Средний случай

Действительно, в худшем случае сортировка выбором выполняется *быстрее*, чем вставками. Но при оценке алгоритма важно учитывать так называемый *средний случай*.

Почему?

Как правило, чаще всего вы будете сталкиваться со средними сценариями. Взгляните на эту кривую нормального распределения:



Лучшие и худшие сценарии реализуются редко. Большинство же случаев — средние.

Возьмем, к примеру, массив, отсортированный произвольно. Какова вероятность того, что значения в нем будут расположены в идеальном порядке возрастания или убывания? Скорее всего, значения будут стоять хаотично.

Поэтому давайте рассмотрим сортировку вставками в контексте всех сценариев.

Мы уже знаем, как этот алгоритм работает в худшем сценарии, когда значения в массиве отсортированы в порядке убывания: нам придется сравнивать и сдвигать каждое значение, с которым мы сталкиваемся (в общей сложности нужно выполнить N^2 сравнений и сдвигов).

В лучшем случае, когда данные уже отсортированы в порядке возрастания, нам придется выполнить только одно сравнение за проход и ни одного сдвига, так как все значения уже находятся на своих местах.

Но когда данные отсортированы произвольно, при выполнении одних проходов мы будем сравнивать и сдвигать все значения, при выполнении других — только их часть, а при выполнении третьих нам вообще не придется ничего трогать. Если вы вернетесь к разделу «Сортировка вставками в действии», то заметите, что при первом и третьем проходах мы сравниваем и сдвигаем все встречающиеся значения, при четвертом — только некоторые, а при втором выполняем только одно сравнение и ни одного сдвига.

Разница в том, что одни проходы требуют сравнения всех значений слева от `temp_value`, а другие завершаются раньше из-за обнаружения значения, меньшего, чем `temp_value`.

Итак, в худшем случае мы сравниваем и сдвигаем *все* данные, а в лучшем — *ничего* не сдвигаем (выполняя только одно сравнение за проход). Можно предположить, что в среднем случае придется сдвинуть около *половины* данных. Так, если в худшем случае сортировка вставками требует выполнения N^2 шагов, то в среднем — около $N^2/2$ (но с точки зрения O-нотации сложность обоих сценариев будет $O(N^2)$).

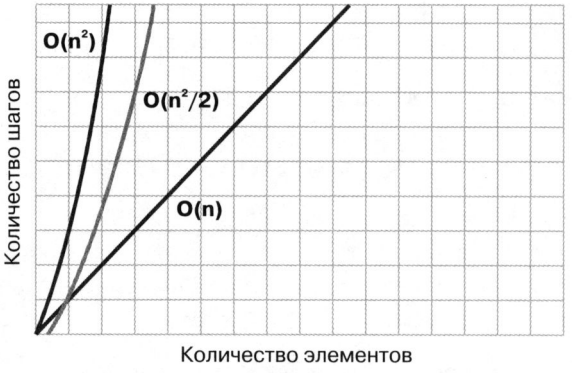
Рассмотрим конкретные примеры.

Массив `[1, 2, 3, 4]` уже отсортирован — это лучший сценарий. Худший сценарий для тех же данных — `[4, 3, 2, 1]`, а средний, допустим, `[1, 3, 4, 2]`.

Худший случай требует выполнения шести сравнений и сдвигов — всего 12 шагов, средний — четыре сравнения и два сдвига, всего 6 шагов. А в лучшем случае нам нужно выполнить три сравнения и ни одного сдвига.

Теперь мы видим, что производительность алгоритма сортировки вставками *сильно отличается* в зависимости от сценария. В худшем случае сортировка вставками требует выполнения N^2 шагов, в среднем — $N^2/2$, а в лучшем — около N шагов.

Эту разницу в производительности графически можно представить так:



Сравните эти показатели с эффективностью сортировки выбором, которая требует выполнения $N^2/2$ шагов во *всех* случаях. Это связано с тем, что у сортировки выбором нет механизма для досрочного завершения прохода. Каждый проход требует сравнения всех значений справа от выбранного индекса.

В этой таблице сортировка выбором сравнивается с сортировкой вставками.

	Лучший сценарий	**Средний сценарий**	**Худший сценарий**
Сортировка выбором	$N^2/2$	$N^2/2$	$N^2/2$
Сортировка вставками	N	$N^2/2$	N^2

Итак, какой же из алгоритмов лучше? На этот вопрос нет однозначного ответа. В среднем случае, когда массив отсортирован произвольно, они работают одинаково эффективно. Если вы предполагаете, что будете иметь дело с более-менее отсортированными данными, то сортировка вставками будет лучшим выбором. Если ожидаете, что данные будут отсортированы в обратном порядке, то отдайте предпочтение сортировке выбором. В среднем случае, когда вы не знаете, какими будут данные, оба алгоритма будут одинаково эффективны.

Практический пример

Допустим, вы пишете приложение на JavaScript и где-то в коде вам нужно найти пересечение двух массивов — получить список значений, которые есть в них

обоих. Например, для массивов `[3, 1, 4, 2]` и `[4, 5, 3, 6]` пересечением будет массив `[3, 4]`, так как эти значения есть и в том, и в другом.

Вот одна из возможных реализаций этого кода:

```
function intersection(firstArray, secondArray){
    let result = [];

    for (let i = 0; i < firstArray.length; i++) {
        for (let j = 0; j < secondArray.length; j++) {
            if (firstArray[i] == secondArray[j]) {
                result.push(firstArray[i]);
            }
        }
    }
    return result;
}
```

Здесь мы запускаем вложенные циклы. Во внешнем мы последовательно перебираем все значения из первого массива, запуская тем самым внутренний цикл, который сравнивает его с каждым из значений второго массива в поисках совпадений.

Алгоритм выполняет два типа шагов: сравнение и вставку. Мы сравниваем все значения двух массивов и помещаем совпадающие в массив `result`. Начнем с подсчета сравнений.

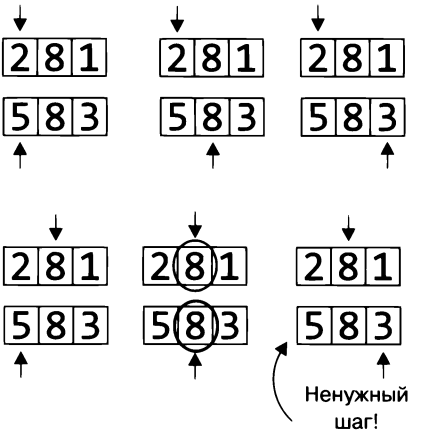
Если у двух массивов одинаковый размер N, то число выполняемых сравнений — N^2, так как мы сравниваем каждый элемент первого массива с каждым элементом второго. Если у нас есть два массива, у каждого из которых по пять элементов, в итоге мы выполним 25 сравнений. Итак, эффективность этого алгоритма поиска пересечения — $O(N^2)$.

Вставки потребовали бы максимум N шагов (если бы два массива оказались идентичными). Это слагаемое более низкого порядка по сравнению с N^2, поэтому мы по-прежнему считаем сложность алгоритма равной $O(N^2)$. Если бы массивы были разного размера, скажем N и M, то эффективность этой функции была бы равна $O(N \times M)$ (подробнее об этом мы поговорим в главе 7).

Можно ли как-то улучшить этот алгоритм?

Чтобы ответить на этот вопрос, рассмотрим остальные сценарии. В текущей реализации функции `intersection` мы выполняем N^2 сравнений во *всех* сценариях вне зависимости от того, идентичны массивы или, наоборот, полностью разные.

Но когда у двух массивов есть общие значения, вовсе не обязательно сверять *каждое* с каждым. Давайте выясним почему:



В этом примере бессмысленно продолжать выполнение второго цикла после нахождения общего значения (8). Нам незачем проверять остальные элементы второго массива — мы уже выяснили, что он, как и первый, содержит значение 8, и можем сразу добавить обнаруженное совпадение в массив `result`. Получается, в этой реализации мы выполняем ненужный шаг.

Чтобы это исправить, достаточно добавить в реализацию всего одно слово:

```
function intersection(firstArray, secondArray){
    let result = [];

    for (let i = 0; i < firstArray.length; i++) {
        for (let j = 0; j < secondArray.length; j++) {
            if (firstArray[i] == secondArray[j]) {
                result.push(firstArray[i]);
                break;
            }
        }
    }
    return result;
}
```

Добавление `break` позволяет досрочно завершить выполнение внутреннего цикла и сократить число шагов (а значит, сэкономить время).

При худшем варианте развития событий, когда у массивов нет ни одного общего значения, нам все равно придется выполнить N^2 сравнений. Но теперь в лучшем сценарии, когда два массива идентичны, нам будет достаточно выполнить всего N сравнений. В среднем, когда массивы разные, но содержат *несколько* одинаковых значений, производительность этого алгоритма будет где-то между N и N^2.

Эта оптимизация функции `intersection` довольно существенная, поскольку наша первая реализация предполагала выполнение N^2 сравнений во всех случаях.

Выводы

Способность различать лучшие, средние и худшие сценарии — важный навык, благодаря которому можно выбрать подходящий алгоритм для определенной ситуации и оптимизировать существующие. Помните, что, несмотря на важность подготовки к худшему, в большинстве случаев реализуются средние сценарии.

Теперь давайте применим знания об О-нотации на практике. В следующей главе мы рассмотрим алгоритмы, которые встречаются в реальных кодовых базах, и определим временную сложность каждого из них с помощью О-нотации.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 6».

1. С помощью О-нотации определите эффективность алгоритма, который выполняет $3N^2 + 2N + 1$ шагов.
2. С помощью О-нотации определите эффективность алгоритма, который выполняет $N + \log N$ шагов.
3. Следующая функция проверяет, содержит ли массив пару чисел, сумма которых равна 10.

```
function twoSum(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      if (i !== j && array[i] + array[j] === 10) {
        return true;
      }
    }
  }
  return false;
}
```

   Проанализируйте лучший, средний и худший случаи, а затем выразите эффективность алгоритма в самом плохом сценарии с помощью О-нотации.

4. Следующая функция проверяет, содержится ли в строке заглавная буква «X».

```
function containsX(string) {

  foundX = false;

  for(let i = 0; i < string.length; i++) {
    if (string[i] === "X") {
      foundX = true;
    }
  }

  return foundX;
}
```

Выразите временную сложность этой функции с помощью O-нотации.

Измените код так, чтобы в лучших и средних случаях алгоритм работал более эффективно.

ГЛАВА 7

О-нотация в работе программиста

Вы уже научились использовать О-нотацию для выражения временной сложности алгоритмов и узнали о множестве аспектов, которые нужно учитывать в ходе проведения анализа. В этой главе вы примените все полученные знания для определения эффективности примеров кода из реальных кодовых баз.

Оценка эффективности кода — это первый этап его оптимизации. В конце концов, если скорость его работы неизвестна, откуда нам знать, как именно его улучшить?

Определив, к какой категории алгоритмической сложности относится наш код, мы сможем решить, нужно ли его вообще улучшать. Например, алгоритм $O(N^2)$ обычно считается «медленным». Если наш алгоритм попадает в эту категорию, придется поискать способы его оптимизации.

Конечно, иногда эффективность $O(N^2)$ может оказаться предельной. Но, зная, что наш алгоритм считается медленным, мы можем попытаться найти более быстрые альтернативы.

В следующих главах вы познакомитесь со множеством приемов оптимизации кода для увеличения скорости его работы. Но сначала вам нужно определить исходную скорость.

Итак, начнем.

Среднее арифметическое четных чисел

Следующий метод языка Ruby принимает массив чисел и возвращает среднее значение всех *четных* из этого массива. Как выразить его эффективность с помощью О-нотации?

```
def average_of_even_numbers(array)

  # Среднее значение четных чисел будет определяться как сумма четных чисел,
  # деленная на их количество, поэтому мы отслеживаем
  # как сумму, так и количество:

  sum = 0.0
  count_of_even_numbers = 0

  # Перебираем все числа в массиве и, когда нам встречается
  # четное, изменяем значение суммы и количества:

  array.each do |number|
    if number.even?
      sum += number
      count_of_even_numbers += 1
    end
  end

  # Возвращаем среднее значение:

  return sum / count_of_even_numbers
end
```

Чтобы определить эффективность кода, разберемся в том, что он делает.

Как вы помните, О-нотация помогает определить, сколько шагов будет выполнять алгоритм при наличии N элементов данных. Поэтому сначала нам нужно определить, что именно подразумевается под N.

В нашем случае алгоритм обрабатывает массив чисел, переданный функции, поэтому N здесь — это размер массива, количество содержащихся в нем значений.

Теперь нужно определить, сколько шагов требуется алгоритму для обработки этих N значений.

Мы видим, что самая важная часть алгоритма — цикл, перебирающий все числа внутри массива, поэтому сначала проанализируем его. Поскольку цикл перебирает все N элементов, мы знаем, что алгоритм выполняет не менее N шагов.

Но *внутри* цикла мы видим, что в каждой из его итераций выполняется разное количество шагов. Сначала мы проверяем все числа на четность, а затем, если

число четное, выполняем еще два шага: изменяем значения переменных `sum` и `count_of_even_numbers`. Получается, что при обнаружении четных чисел мы выполняем на два шага больше, чем при обнаружении нечетных.

Мы уже знаем, что О-нотация фокусируется в первую очередь на худших сценариях. Худший случай в нашем примере — это когда все числа в массиве четные и мы вынуждены выполнять по три шага в рамках каждой итерации цикла. Получается, что при наличии N элементов данных наш алгоритм выполняет $3N$ шагов — по три для каждого из N чисел.

Вне цикла наш метод выполняет еще несколько шагов: перед запуском цикла мы инициализируем две переменные нулем. Технически это два шага. По завершении цикла мы выполняем еще один шаг: делим значение `sum` на значение `count_of_even_numbers`. По сути, наш алгоритм выполняет еще три шага в дополнение к $3N$ шагам, поэтому общее количество шагов равно $3N + 3$.

Но, как мы помним, О-нотация игнорирует константы, поэтому с ее точки зрения сложность нашего алгоритма будет $O(N)$, а не $O(3N + 3)$.

Конструктор слов

Следующий пример — алгоритм, который составляет двухсимвольные строки из отдельных символов в массиве. Например, если мы передадим ему массив `["a", "b", "c", "d"]`, он вернет нам новый, содержащий следующие строки:

```
[
  'ab', 'ac', 'ad', 'ba', 'bc', 'bd',
  'ca', 'cb', 'cd', 'da', 'db', 'dc'
]
```

Ниже приведена реализация этого алгоритма на языке JavaScript. Попробуем оценить его эффективность с помощью О-нотации:

```
function wordBuilder(array) {
  let collection = [];

  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      if (i !== j) {
        collection.push(array[i] + array[j]);
      }
    }
  }

  return collection;
}
```

Здесь мы используем вложенные циклы. Внешний перебирает все символы в массиве, отслеживая их индексы с помощью переменной `i`. Для каждого индекса `i` выполняется внутренний цикл, который снова перебирает все символы в том же массиве, отслеживая их индексы с помощью переменной `j`. В этом внутреннем цикле выполняется конкатенация символов с индексами `i` и `j`, за исключением случаев, когда `i` и `j` указывают на один и тот же индекс.

Чтобы определить эффективность этого алгоритма, нужно выяснить, что понимается под *N*. В этом примере, как и в прошлом, *N* соответствует количеству элементов внутри массива, передаваемого в функцию.

Следующий этап — определение количества шагов, которые алгоритм выполняет при наличии *N* элементов данных. В нашем случае внешний цикл перебирает все *N* элементов, для каждого из которых запускается внутренний цикл, тоже перебирающий все *N* элементов, что в итоге составляет $N \times N$ шагов. Это классический пример алгоритма со сложностью $O(N^2)$, и именно в эту категорию обычно попадают алгоритмы с вложенными циклами.

Теперь посмотрим, что произойдет, если мы изменим этот алгоритм так, чтобы он создавал массив *трехсимвольных* строк: то есть при передаче массива `["a", "b", "c", "d"]` возвращал следующий:

```
[
  'abc', 'abd', 'acb',
  'acd', 'adb', 'adc',
  'bac', 'bad', 'bca',
  'bcd', 'bda', 'bdc',
  'cab', 'cad', 'cba',
  'cbd', 'cda', 'cdb',
  'dab', 'dac', 'dba',
  'dbc', 'dca', 'dcb'
]
```

Эта реализация использует три вложенных цикла. Какова ее временная сложность?

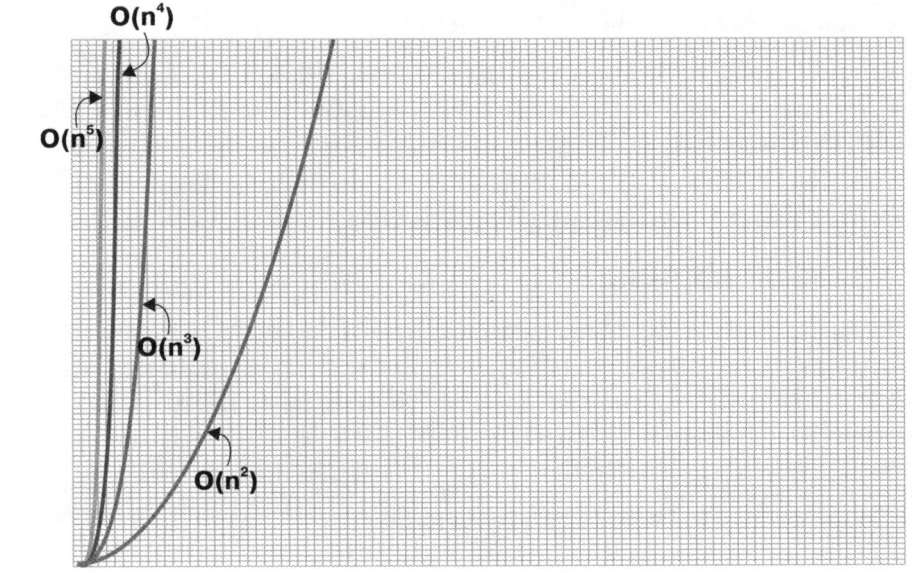
```
function wordBuilder(array) {
  let collection = [];

  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      for(let k = 0; k < array.length; k++) {
        if (i !== j && j !== k && i !== k) {
          collection.push(array[i] + array[j] + array[k]);
        }
      }
    }
  }

  return collection;
}
```

В этом алгоритме при наличии N элементов данных мы выполняем N шагов в цикле `i`, N в цикле `j` и N в цикле `k`, то есть всего $N \times N \times N$, или N^3 шагов. Получается, сложность этого алгоритма — $O(N^3)$.

При использовании четырех или пяти вложенных циклов сложность алгоритма была бы $O(N^4)$ и $O(N^5)$ соответственно. Посмотрим, как все это выглядит на графике.



Было бы здорово, если бы мы смогли оптимизировать код, чтобы перевести алгоритм из категории $O(N^3)$ в $O(N^2)$, поскольку в этом случае код стал бы на порядок быстрее.

Выборка из массива

В следующем примере мы создаем функцию, которая отбирает несколько элементов массива. Предположительно, нам придется работать с очень большими массивами, поэтому выборка будет включать только первое, среднее и последнее значения.

Вот реализация этой функции на Python. Попробуйте оценить ее эффективность через O-нотацию:

```
def sample(array):
  first = array[0]
  middle = array[int(len(array) / 2)]
```

```
    last = array[-1]

    return [first, middle, last]
```

Здесь N — это снова количество элементов в массиве, передаваемом в функцию.

Но эта функция выполняет одинаковое количество шагов вне зависимости от значения N. Чтение с начала, середины и последнего индекса массива занимает один шаг независимо от размера массива. Вычисление длины массива и ее деление пополам тоже выполняется за шаг.

Поскольку количество шагов постоянное, то есть остается неизменным вне зависимости от значения N, сложность этого алгоритма считается равной O(1).

Среднее значение температуры в градусах Цельсия

Рассмотрим еще один пример с вычислением среднего значения. Допустим, мы создаем ПО для прогнозирования погоды. Чтобы определить температуру в городе, мы собираем показания множества термометров по всему населенному пункту и находим среднюю температуру.

Мы хотим показывать данные в градусах Фаренгейта и Цельсия, но изначально показания получаем только в градусах Фаренгейта.

Наш алгоритм вычисляет среднюю температуру по Цельсию в два этапа: сначала переводит все показания в градусах Фаренгейта в градусы Цельсия, а затем вычисляет среднее значение.

Ниже приведен код на языке Ruby, который выполняет эти действия. Какова его эффективность с точки зрения О-нотации?

```
def average_celsius(fahrenheit_readings)

  # Собираем значения температуры в градусах Цельсия:
  celsius_numbers = []

  # Преобразуем каждое значение в градусы Цельсия и добавляем в массив:
  fahrenheit_readings.each do |fahrenheit_reading|
    celsius_conversion = (fahrenheit_reading - 32) / 1.8
    celsius_numbers.push(celsius_conversion)
  end

  # Вычисляем сумму всех значений температуры в градусах Цельсия:
  sum = 0.0

  celsius_numbers.each do |celsius_number|
```

```
    sum += celsius_number
  end

  # Возвращаем среднее значение:
  return sum / celsius_numbers.length
end
```

Во-первых, из примера видно, что под N подразумевается количество значений температуры в градусах Фаренгейта (`fahrenheit_readings`), переданных нашему методу.

Внутри метода мы запускаем два цикла: первый преобразует показания в градусы Цельсия, а второй суммирует все полученные в результате данные. Поскольку у нас есть два цикла, каждый из которых перебирает все N элементов, наш алгоритм выполняет $N + N$, то есть $2N$ шагов (и еще несколько шагов, число которых не зависит от N). Поскольку O-нотация игнорирует константы, сложность этого алгоритма будет $O(N)$.

Пусть вас не смущает, что в более раннем примере с конструктором слов использование двух циклов привело к эффективности $O(N^2)$. Там мы имели дело с *вложенными* циклами, поэтому при подсчете количества шагов мы *умножали* N на N. Но здесь мы используем два цикла, следующих друг за другом, поэтому просто *складываем* N и N шагов, получая $2N$, что соответствует алгоритмической сложности $O(N)$.

Бирки для одежды

Допустим, мы пишем программу для производителя одежды. Наш код принимает массив новых предметов одежды (в виде строк) и создает текст для каждой этикетки, которая может понадобиться.

Наши бирки должны содержать название предмета одежды и его размер от 1 до 5. Например, если у нас есть массив `["Purple Shirt", "Green Shirt"]`, мы создадим для этих рубашек следующие этикетки:

```
[
"Purple Shirt Size: 1",
"Purple Shirt Size: 2",
"Purple Shirt Size: 3",
"Purple Shirt Size: 4",
"Purple Shirt Size: 5",
"Green Shirt Size: 1",
"Green Shirt Size: 2",
"Green Shirt Size: 3",
"Green Shirt Size: 4",
"Green Shirt Size: 5"
]
```

Это код на Python, который создает такой текст для всего набора предметов одежды:

```
def mark_inventory(clothing_items):
  clothing_options = []

  for item in clothing_items:
    # Для размеров от 1 до 5 (в Python функция range
    # не включает второе число в диапазон):
    for size in range(1, 6):
      clothing_options.append(item + " Size: " + str(size))

  return clothing_options
```

Определим эффективность этого алгоритма. Поскольку обрабатываемые данные — предметы одежды (`clothing_items`), N будет отражать их количество.

Этот код содержит вложенные циклы, поэтому поначалу можно подумать, что сложность алгоритма будет $O(N^2)$, но это не так. Давайте проведем более тщательный анализ.

Сложность алгоритмов со вложенными циклами $O(N^2)$ только тогда, когда каждый из этих циклов выполняется N раз. Но в нашем случае N раз выполняется только внешний, тогда как внутренний выполняется постоянное количество раз — пять. То есть внутренний цикл всегда будет выполняться пять раз вне зависимости от значения N.

Итак, наш внешний цикл выполняется N раз, а внутренний — пять раз для каждой из N строк. Получается, что наш алгоритм выполняется за $5N$ шагов, но он относится к категории $O(N)$, потому что О-нотация игнорирует константы.

Подсчет единиц

Вот еще один алгоритм, сложность которого непросто оценить сразу. Эта функция принимает *массив массивов* со значениями `1` и `0`, а затем возвращает общее количество единиц в переданных массивах.

То есть если мы передадим этой функции:

```
[
  [0, 1, 1, 1, 0],
  [0, 1, 0, 1, 0, 1],
  [1, 0]
]
```

она возвратит значение 7, так как в переданных массивах в общей сложности семь единиц.

Вот реализация этой функции на Python:

```
def count_ones(outer_array):
  count = 0

  for inner_array in outer_array:
    for number in inner_array:
      if number == 1:
        count += 1

  return count
```

Какова временная сложность этого алгоритма?

Здесь мы опять можем заметить вложенные циклы и ошибочно предположить, что она составляет O(N^2). Но эти два цикла перебирают два совершенно разных набора значений.

Внешний перебирает внутренние массивы, а внутренний — фактические числа. В итоге, число итераций внутреннего цикла равно *общему количеству* чисел в переданных массивах.

Поскольку N — это общее количество чисел, а наш алгоритм просто обрабатывает каждое из них, временная сложность этой функции равна O(N).

Поиск палиндрома

Палиндром — это слово или фраза, которые одинаково читаются в обе стороны — слева направо и справа налево. Например, «ротатор», «потоп» и «заказ».

Вот функция на языке JavaScript, которая определяет, является ли строка палиндромом:

```
function isPalindrome(string) {

  // В качестве начального значения для leftIndex задаем 0:
  let leftIndex = 0;
  // В качестве начального значения для rightIndex задаем последний индекс
  // массива:
  let rightIndex = string.length - 1;

  // Повторяем, пока leftIndex не достигнет середины массива:
  while (leftIndex < string.length / 2) {

    // Если символ слева отличается от символа справа, значит,
    // эта строка не палиндром:
    if (string[leftIndex] !== string[rightIndex]) {
      return false;
    }
```

```
    // Сдвигаем leftIndex на единицу вправо:
    leftIndex++;
    // Сдвигаем rightIndex на единицу влево:
    rightIndex--;
  }

  // Если мы выполнили весь цикл, не обнаружив несоответствий, значит,
  // эта строка - палиндром:
  return true;
}
```

Определим временную сложность этого алгоритма.

Здесь N — это размер строки `string`, переданной функции.

Самое важное происходит внутри цикла `while`. Он интересен тем, что выполняется вплоть до достижения середины строки — то есть цикл выполняет $N/2$ шагов.

Но поскольку О-нотация игнорирует константы, мы отбрасываем «/ 2» и получаем оценку сложности алгоритма O(N).

Вычисление произведений всех пар чисел

Следующий пример — алгоритм, который принимает числовой массив и возвращает произведения всех возможных пар этих чисел.

Например, если мы передадим функции массив `[1, 2, 3, 4, 5]`, она вернет:

```
[2, 3, 4, 5, 6, 8, 10, 12, 15, 20]
```

Сначала 1 умножается на 2, 3, 4 и 5. Затем 2 — на 3, 4 и 5. Потом 3 — на 4 и 5. И, наконец, 4 умножается на 5.

Обратите внимание, что при умножении, скажем, двойки на другие числа мы используем только те, что справа от нее. Нам не нужно возвращаться назад и умножать 2 на 1 — это уже было сделано при умножении 1 на 2. Поэтому каждое число нужно умножать только на числа, которые остались справа от него.

Так выглядит реализация этого алгоритма на языке JavaScript:

```
function twoNumberProducts(array) {
  let products = [];

  // Внешний массив:
  for(let i = 0; i < array.length - 1; i++) {
```

```
    // Внутренний массив, где начальным значением j всегда будет индекс
    // справа от i:
    for(let j = i + 1; j < array.length; j++) {
      products.push(array[i] * array[j]);
    }
  }

  return products;
}
```

Теперь давайте разбираться. N — это количество элементов в массиве, переданном функции.

Мы выполняем внешний цикл N раз (на самом деле запускаем его $N - 1$ раз, но отбрасываем эту константу). Но внутренний цикл немного отличается. Поскольку начальным значением `j` всегда будет индекс справа от `i`, число шагов во внутреннем цикле уменьшается с каждым его запуском.

Посмотрим, сколько раз выполняется внутренний цикл в нашем примере с массивом из пяти элементов.

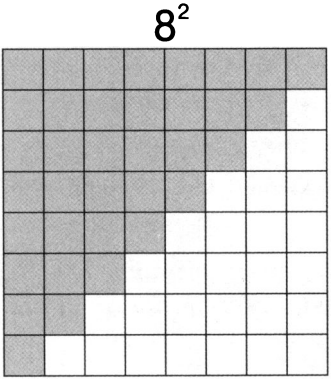
При `i`, равном 0, внутренний цикл выполняется для `j`, равного 1, 2, 3 и 4. При `i`, равном 1, — для `j`, равного 2, 3 и 4. При `i`, равном 2, — для `j`, равного 3 и 4. При `i`, равном 3, — для `j`, равного 4. В общей сложности внутренний цикл выполняется:

$$4 + 3 + 2 + 1 \text{ раз.}$$

То есть при наличии N элементов данных внутренний цикл выполняется приблизительно:

$$N + (N - 1) + (N - 2) + (N - 3) \ldots + 1 \text{ раз.}$$

Это соответствует примерно $N^2/2$ шагам. Изобразим это графически. Здесь N равно 8, поэтому на схеме изображено 8^2, или 64 квадрата.



Как видите, в верхнем ряду все N квадратов серые. В следующем ряду серого цвета $N - 1$ квадрат, а в последующем — $N - 2$. Эта закономерность прослеживается вплоть до нижнего ряда, где есть только один серый квадрат.

Все это указывает на то, что закономерность $N + (N - 1) + (N - 2) + (N - 3)... + 1$ равна $N^2/2$.

Итак, мы выяснили, что внутренний цикл выполняет $N^2/2$ шагов. Но поскольку О-нотация игнорирует константы, мы выражаем эту оценку в виде $O(N^2)$.

Работа с несколькими наборами данных

А что будет, если вместо вычисления произведений всех пар чисел в одном массиве мы решим перемножить каждое число из одного массива с каждым числом из *второго*?

Например, при наличии массивов `[1, 2, 3]` и `[10, 100, 1000]` результат был бы такой:

```
[10, 100, 1000, 20, 200, 2000, 30, 300, 3000]
```

Используем код из прошлого примера, но внесем в него небольшие изменения:

```
function twoNumberProducts(array1, array2) {
  let products = [];

  for(let i = 0; i < array1.length; i++) {
    for(let j = 0; j < array2.length; j++) {
      products.push(array1[i] * array2[j]);
    }
  }

  return products;
}
```

Проанализируем временную сложность этой функции.

Во-первых, что отражает N? Ответить на этот вопрос непросто, поскольку теперь у нас есть два набора данных — два массива.

Весьма заманчиво было бы объединить их и сказать, что N — это общее количество элементов в обоих массивах. Но это проблематично по следующей причине.

Рассмотрим два сценария. В первом используется два массива, в каждом из которых по 5 элементов. А во втором — один массив с 9 элементами и один с 1.

В обоих случаях мы бы сказали, что N равно 10, поскольку 5 + 5 = 10 и 9 + 1 = 10. Но эффективность алгоритмов в этих сценариях *сильно* различается.

В сценарии 1 код выполняет 25 (5 × 5) шагов. Поскольку $N = 10$, это эквивалентно $(N/2)^2$ шагам.

Но в сценарии 2 код выполняет 9 (9 × 1) шагов, что довольно близко к N шагам. А значит, этот алгоритм гораздо быстрее!

Поэтому мы не должны подразумевать под N общее количество чисел из обоих массивов, ведь тогда мы не сможем определить эффективность алгоритма с помощью O-нотации, так как она разная для разных сценариев.

Итак, мы оказались в затруднительном положении: у нас нет другого выбора, кроме как выразить временную сложность алгоритма в виде $O(N \times M)$, где N — размер одного массива, а M — размер другого.

Суть такого подхода в следующем: всякий раз, когда мы используем два разных набора данных, которые нужно перемножить, мы должны идентифицировать эти источники данных отдельно при описании эффективности алгоритма с помощью O-нотации.

Хотя это правильный способ оценки сложности алгоритма с помощью O-нотации, он менее полезен, чем другие, потому что сравнивать алгоритм $O(N \times M)$ с $O(N)$ — все равно что сравнивать яблоко с апельсином.

Но мы знаем, что $O(N \times M)$ находится в определенном диапазоне. Если значения N и M одинаковы, то сложность этого алгоритма равна $O(N^2)$. А если они не совпадают и мы произвольно назначаем M меньшим из двух чисел, то, чему бы оно ни было равно, в итоге мы получаем сложность $O(N)$. Так, в некотором смысле $O(N \times M)$ можно считать диапазоном между $O(N)$ и $O(N^2)$.

Хорошая работа? Нет, но это все, на что мы способны.

Взломщик паролей

Представьте, что вы хакер (этичный, разумеется) и вам нужно подобрать чей-то пароль. Вы выбираете метод грубой силы (brute-force) и пишете код, который создает все возможные строки заданной длины:

```
def every_password(n)
  (("a" * n)..("z" * n)).each do |str|
    puts str
  end
end
```

Он работает так: вы передаете функции число, которое служит значением переменной `n`.

Например, если `n` равно 3, то код `"a" * n` создает строку `"aaa"`, а затем запускает цикл, с помощью которого создает все возможные строки в диапазоне от `"aaa"` до `"zzz"`.

Результат выполнения этого кода будет выглядеть так:

```
aaa
aab
aac
aad
aae

...

zzx
zzy
zzz
```

Если `n` равно 4, этот код выведет на экран все возможные строки из 4 символов:

```
aaaa
aaab
aaac
aaad
aaae

...

zzzx
zzzy
zzzz
```

Если вы захотите запустить этот код при `n`, равном 5, вам придется подождать, пока работа этой функции завершится. Это медленный алгоритм! Но как мы можем проанализировать его с помощью О-нотации?

Давайте разберемся.

Для однократного вывода каждой из букв алфавита коду придется выполнить 26 шагов.

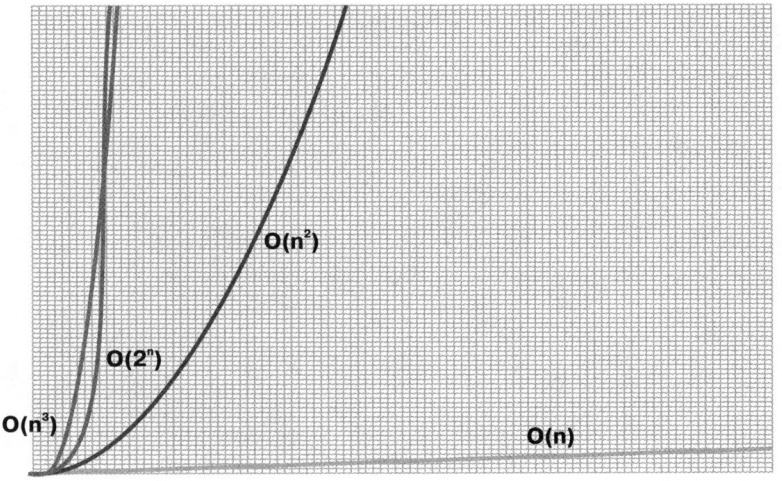
Для вывода всех двухсимвольных комбинаций — 26×26 шагов, а всех трехсимвольных — $26 \times 26 \times 26$.

Видите закономерность?

Длина строки	Комбинации
1	26
2	26^2
3	26^3
4	26^4

Если N — это длина каждой строки, то *количество комбинаций* — 26^N.

С помощью О-нотации сложность этого алгоритма выражается в виде $O(26^N)$. Он работает до ужаса медленно! Даже алгоритм, сложность которого равна «всего лишь» $O(2^N)$, крайне неэффективен. Сравним его с другими алгоритмами:



Как видите, в какой-то момент $O(2^N)$ начинает работать даже медленнее, чем $O(N^3)$.

В некотором смысле алгоритм $O(2^N)$ — противоположность $O(\log N)$. В $O(\log N)$ (например, при выполнении двоичного поиска) удвоение объема данных приводит к добавлению одного шага. В $O(2^N)$ добавление *одного* элемента данных приводит к *удвоению* количества шагов!

Получается, что каждый раз, когда мы увеличиваем значение N на единицу во взломщике паролей, количество шагов умножается на 26. Этот алгоритм работает невероятно медленно, именно поэтому метод грубой силы — далеко не самый эффективный способ взлома паролей.

Выводы

Поздравляю! Теперь вы настоящий знаток О-нотации. Вы можете анализировать любые алгоритмы и оценивать их временную сложность. Благодаря полученным знаниям вы сможете методично оптимизировать свой код.

В следующей главе вы познакомитесь с новой структурой данных — одним из самых полезных и часто используемых инструментов для ускорения алгоритмов.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 7».

1. С помощью О-нотации определите временную сложность следующей функции, которая возвращает `true`, если массив соответствует следующим критериям, и `false`, если нет:
   - первое и последнее значения массива в сумме дают 100;
   - второе и предпоследнее значения в сумме дают 100;
   - третье и предпредпоследнее значения в сумме дают 100 и т. д.

   Код этой функции выглядит так:

```
def one_hundred_sum?(array)
  left_index = 0
  right_index = array.length - 1

  while left_index < array.length / 2
    if array[left_index] + array[right_index] != 100
      return false
    end

    left_index += 1
    right_index -= 1
  end

  return true
end
```

2. С помощью О-нотации определите временную сложность следующей функции, которая объединяет два отсортированных массива, чтобы создать новый, содержащий все значения из обоих массивов:

```
def merge(array_1, array_2)
  new_array = []
  array_1_pointer = 0
  array_2_pointer = 0
```

```
  # Выполняем цикл, пока не достигнем конца обоих массивов:
  while array_1_pointer < array_1.length ||
      array_2_pointer < array_2.length

    # Если мы уже достигли конца первого массива,
    # добавляем элемент из второго:
    if !array_1[array_1_pointer]
      new_array << array_2[array_2_pointer]
      array_2_pointer += 1
    # Если мы уже достигли конца второго массива,
    # добавляем элемент из первого:
    elsif !array_2[array_2_pointer]
      new_array << array_1[array_1_pointer]
      array_1_pointer += 1
    # Если текущее число в первом массиве меньше, чем текущее
    # число во втором, добавляем из первого массива:
    elsif array_1[array_1_pointer] < array_2[array_2_pointer]
      new_array << array_1[array_1_pointer]
      array_1_pointer += 1
    # Если текущее число во втором массиве меньше текущего числа
    # в первом или равно ему, добавляем из второго массива:
    else
      new_array << array_2[array_2_pointer]
      array_2_pointer += 1
    end
  end

  return new_array
end
```

3. С помощью O-нотации определите временную сложность следующей функции, которая решает известную задачу: «Поиск иголки в стоге сена».

   Здесь стог и иголка представлены в виде строк. Например, если игла — это `"def"`, а стог сена — `"abcdefghi"`, то иглу можно найти в нем, поскольку `"def"` — подстрока строки `"abcdefghi"`. Но если игла — это `"dd"`, то в этом стоге ее найти нельзя.

   Эта функция возвращает значение `true` или `false` в зависимости от того, есть ли конкретная иголка в конкретном стоге:

```
def find_needle(needle, haystack)
  needle_index = 0
  haystack_index = 0

  while haystack_index < haystack.length
    if needle[needle_index] == haystack[haystack_index]
      found_needle = true

      while needle_index < needle.length
        if needle[needle_index] != haystack[haystack_index + needle_index]
          found_needle = false
```

```
        break
      end
      needle_index += 1
    end

    return true if found_needle
    needle_index = 0
  end

  haystack_index += 1
end

return false
end
```

4. С помощью О-нотации определите временную сложность следующей функции, которая находит наибольшее произведение трех чисел в заданном массиве:

```
def largest_product(array)
  largest_product_so_far = array[0] * array[1] * array[2]
  i = 0

  while i < array.length
    j = i + 1
    while j < array.length
      k = j + 1
      while k < array.length
        if array[i] * array[j] * array[k] > largest_product_so_far
          largest_product_so_far = array[i] * array[j] * array[k]
        end
        k += 1
      end
      j += 1
    end
    i += 1
  end

  return largest_product_so_far

end
```

5. Однажды я слышал шуточный совет для специалистов по кадрам: «Хотите мгновенно исключить самых неудачливых людей из списка кандидатов? Просто возьмите со стола половину всех резюме и выбросьте их».

   Если бы нам нужно было написать программу, сокращающую стопку резюме, пока не останется всего одно, мы могли бы использовать подход попеременного отбрасывания верхней и нижней половин стопки. То есть сначала мы можем выбросить верхнюю половину, а затем — нижнюю половину оставшейся стопки, продолжая попеременно исключать верхние и нижние

половины, пока у нас не останется одно резюме того счастливчика, которого мы и наймем!

Опишите эффективность этой функции с помощью О-нотации:

```
def pick_resume(resumes)
  eliminate = "top"

  while resumes.length > 1
    if eliminate == "top"
      resumes = resumes[resumes.length / 2, resumes.length - 1]
      eliminate = "bottom"
    elsif eliminate == "bottom"
      resumes = resumes[0, resumes.length / 2]
      eliminate = "top"
    end
  end

  return resumes[0]
end
```

ГЛАВА 8

Молниеносный поиск с помощью хеш-таблиц

Представьте, что пишете программу, позволяющую клиентам делать заказы в ресторане быстрого питания, и создаете меню с ценами. Вы могли бы использовать для этого следующий массив:

```
menu = [ ["french fries", 0.75], ["hamburger", 2.5], ["hot dog", 1.5],
         ["soda", 0.6] ]
```

Он содержит несколько подмассивов, в каждом из которых по два элемента: первый — это строка с названием блюда, а второй — его цена.

Как было сказано в главе 2, если бы этот массив был неупорядоченным, то нахождение цены заданного блюда заняло бы O(N) времени, поскольку компьютеру пришлось бы осуществить линейный поиск. Если бы массив был упорядоченным, компьютер мог бы выполнить двоичный поиск за время O(log N).

Хотя O(log N) — это уже довольно неплохо, мы можем добиться большего, *намного* большего. В этой главе вы познакомитесь с особой структурой данных — *хеш-таблицей*, позволяющей находить данные за постоянное время O(1). Понимая принцип работы хеш-таблиц и зная, в каких случаях их применять, вы сможете использовать их высокую скорость поиска во многих ситуациях.

Хеш-таблицы

Большинство языков программирования содержат структуру данных, называемую *хеш-таблицей*, которая обладает удивительной суперсилой — скорочтением. В зависимости от языка эти структуры еще могут называться хешами, картами, отображениями, хеш-картами, словарями и ассоциативными массивами.

Вот пример меню, реализованного с помощью хеш-таблицы на языке Ruby:

```
menu = { "french fries" => 0.75, "hamburger" => 2.5,
"hot dog" => 1.5, "soda" => 0.6 }
```

Хеш-таблица — это список парных значений. Первый элемент в каждой паре называется *ключом*, а второй — *значением*. Ключ и значение тесно связаны друг с другом. В нашем примере строка `"french fries"` — это ключ, а 0,75 — значение. Вместе они указывают на то, что картофель фри стоит 75 центов.

В Ruby вы можете найти значение ключа с помощью следующего синтаксиса:

```
menu["french fries"]
```

Этот код возвратит значение 0,75.

Средняя эффективность поиска значения в хеш-таблице — O(1), так как обычно на это уходит *только один шаг*. Разберемся почему.

Хеширование

Помните секретные коды, которые использовали в детстве для создания и расшифровки сообщений?

Например, вот простой способ сопоставить буквы с цифрами:

A = 1

B = 2

C = 3

D = 4

E = 5

и т. д.

С помощью этого кода слово

ACE преобразуется в 135,

CAB — в 312,

DAB — в 412,

BAD — в 214.

Процесс преобразования символов в числа называется *хешированием*, а код, который для этого используется, — *хеш-функцией*.

Есть еще много хеш-функций, помимо той, что я показал выше. Например, такая функция могла бы возвращать *сумму* всех чисел, соответствующих буквам хешируемой последовательности. При ее использовании результатом хеширования слова BAD было бы число 7, полученное после выполнения следующего двухэтапного процесса:

Шаг 1: преобразование слова BAD в значение 214.

Шаг 2: суммирование полученных чисел:

$$2 + 1 + 4 = 7$$

Другая хеш-функция могла бы не складывать, а перемножать соответствующие числа. Она преобразовала бы слово BAD в число 8:

Шаг 1: преобразование слова BAD в значение 214.

Шаг 2: перемножение полученных чисел:

$$2 \times 1 \times 4 = 8$$

В примерах из этой главы будет использоваться именно эта версия. Реальные хеш-функции намного сложнее, но эта позволит сделать наши примеры максимально ясными и простыми.

По правде говоря, чтобы быть валидной, хеш-функция должна соответствовать только одному критерию: она должна каждый раз преобразовывать одну и ту же строку *в одно и то же число*. Если функция возвращает противоречивые результаты при хешировании одной и той же строки, она невалидна.

В качестве примера невалидных хеш-функций можно привести те, которые используют в процессе вычислений случайные числа или текущее время. Они могут сначала преобразовать слово BAD в 12, а потом — в 106.

Но при использовании нашей хеш-функции «перемножения» слово BAD *всегда* будет преобразовываться в 8, потому что B всегда соответствует 2, A — 1, а D — 4. А $2 \times 1 \times 4$ *всегда* равно 8. С этим ничего нельзя поделать.

Обратите внимание, что с помощью этой функции слово DAB *тоже* преобразуется в 8, как и BAD. Это порождает некоторые сложности, о которых я расскажу чуть позже.

Теперь, когда мы знаем, что такое хеширование, мы можем понять, как работают хеш-таблицы.

Создание тезауруса для удовольствия и прибыли, но в основном для прибыли

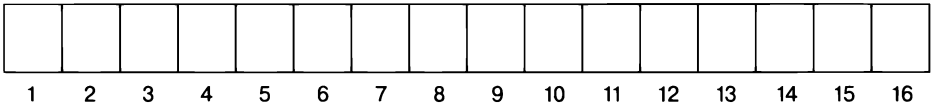
Представьте, что по ночам и в выходные вы в одиночку работаете над революционной программой Quickasaurus — приложением-тезаурусом, но не обычным, а сверхбыстрым. Вы знаете, что оно может захватить многомиллиардный рынок электронных словарей. Когда пользователь ищет слово в Quickasaurus, программа возвращает только один синоним, а не все возможные, как это делают старомодные приложения.

Поскольку у каждого слова есть синоним, эта ситуация отлично подходит для использования хеш-таблицы, которая представляет собой список парных элементов. Итак, приступим.

Мы можем представить наш тезаурус с помощью хеш-таблицы:

```
thesaurus = {}
```

Подобно массиву, хеш-таблица хранит данные в наборе ячеек, расположенных в один ряд. У каждой ячейки есть свой номер. Например:



(Мы опустили индекс 0, так как при использовании нашей хеш-функции «перемножения» там ничего не будет храниться.)

Добавим в хеш-таблицу первую запись:

```
thesaurus["bad"] = "evil"
```

Теперь ее код выглядит так:

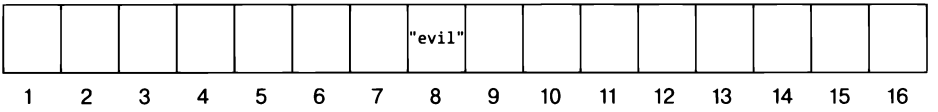
```
{"bad" => "evil"}
```

Давайте разберемся, как хеш-таблица хранит эти данные.

Сначала компьютер применяет хеш-функцию к ключу. Мы используем описанную ранее хеш-функцию «перемножения», поэтому это выглядит так:

$$BAD = 2 \times 1 \times 4 = 8.$$

Ключ `"bad"` хешируется в 8, поэтому компьютер помещает значение `"evil"` в ячейку 8:



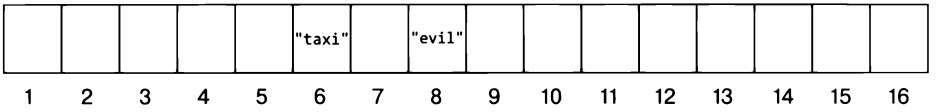
Теперь добавим еще одну пару «ключ — значение»:

```
thesaurus["cab"] = "taxi"
```

Компьютер хеширует ключ:

$$CAB = 3 \times 1 \times 2 = 6.$$

Итоговое значение равно 6, поэтому компьютер сохраняет значение `"taxi"` в ячейке 6.

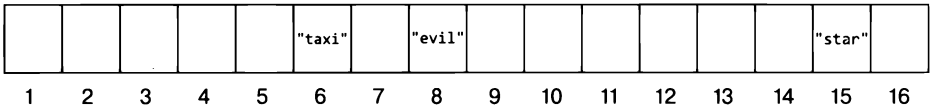


Добавим еще одну пару «ключ — значение»:

```
thesaurus["ace"] = "star"
```

Подытожим то, что здесь происходит: при добавлении каждой пары «ключ — значение» *значение* сохраняется в ячейке, *индекс* которой соответствует результату хеширования *ключа*.

Слово ACE хешируется в 15, так как ACE = 1 × 3 × 5 = 15, поэтому значение `"star"` сохраняется в ячейке с индексом 15:



Сейчас код нашей хеш-таблицы выглядит так:

```
{"bad" => "evil", "cab" => "taxi", "ace" => "star"}
```

Поиск в хеш-таблице

При поиске элементов в хеш-таблице мы используем ключ, чтобы найти связанное с ним значение. Посмотрим, как это работает на нашем примере с хеш-таблицей Quickasaurus.

Допустим, мы хотим найти значение, связанное с ключом `"bad"`. Для этого можно использовать следующий код:

```
thesaurus["bad"]
```

Компьютер выполняет два простых шага.

1. Хеширует ключ: BAD = 2 × 1 × 4 = 8.
2. Поскольку результат равен 8, заглядывает внутрь ячейки 8 и возвращает значение из нее. У нас это строка `"evil"`.

Давайте остановимся и посмотрим на картину в целом. Положение каждого значения в хеш-таблице определяется его ключом. То есть, хешируя сам ключ, мы вычисляем номер индекса, в который должно быть помещено связанное с этим ключом значение.

Зная, что ключ определяет местоположение значения, мы можем существенно упростить процесс поиска. Если у нас есть ключ и мы хотим найти связанное с ним значение, то он сам может сообщить, где оно. Точно так же, как мы хешировали ключ для сохранения значения в соответствующую ячейку, мы можем хешировать его, чтобы это значение найти.

Теперь ясно, почему на поиск значения в хеш-таблице обычно уходит одинаковое количество времени, то есть его сложность — O(1). Компьютер хеширует ключ, преобразуя его в число, и переходит к ячейке с соответствующим индексом, чтобы считать значение в ней.

Теперь понятно, почему хеш-таблица обеспечивает более быстрый поиск в меню, чем массив. Чтобы найти стоимость блюда в массиве, нам пришлось бы последовательно проверять все ячейки до обнаружения нужного значения. В случае с неупорядоченным массивом это заняло бы O(N) времени, а с упорядоченным — O(log N). Но в хеш-таблице пункты меню могут стать для нас ключами, чтобы мы смогли выполнять поиск за постоянное время O(1). *В этом* вся прелесть хеш-таблиц.

Однонаправленный поиск

Важно отметить, что поиск в один шаг в хеш-таблице возможен, только если мы знаем соответствующий ключ. Если бы мы попытались найти конкретное значение без его ключа, нам все равно пришлось бы проверить все пары «ключ — значение» в таблице и сложность такого поиска была бы O(*N*).

Поиск обладает сложностью O(1), только когда для нахождения *значения* мы используем *ключ*. Если мы решим использовать *значение*, чтобы найти связанный с ним *ключ*, быстрый поиск в хеш-таблице будет недоступен.

Это связано с базовым принципом работы хеш-таблиц: ключ определяет положение значения. Но это работает, только когда мы используем ключ, чтобы найти значение. Значение же не определяет местоположение ключа, поэтому мы можем найти его только методом перебора.

А где вообще хранятся эти ключи? На прошлых диаграммах мы видели только то, как в хеш-таблице хранятся значения.

В некоторых языках программирования ключи хранятся вместе со значениями. Это бывает полезно в случае возникновения коллизий, о которых мы поговорим в следующем разделе.

А пока рассмотрим еще один аспект однонаправленной природы хеш-таблиц, о котором важно знать. Хеш-таблица может содержать только один экземпляр ключа, но в ней может храниться несколько экземпляров значения.

Вернемся к примеру с меню из начала главы. Мы не можем указать гамбургер дважды (да нам это и не нужно, ведь у него только одна цена). Но мы *могли бы* включить в это меню несколько блюд по 2,50 доллара.

Во многих языках попытка сохранить пару «ключ — значение» при существующем ключе приведет к перезаписи старого значения.

Разрешение коллизий

Хеш-таблицы — отличный инструмент, но недостатки в нем тоже есть.

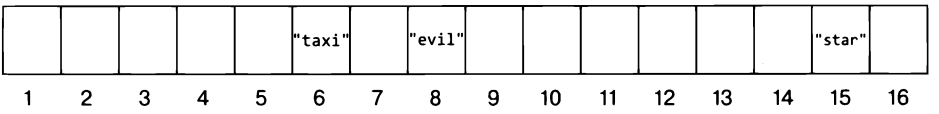
Вернемся к примеру с тезаурусом: что будет, если мы захотим добавить в него следующую запись?

```
thesaurus["dab"] = "pat"
```

Сначала компьютер хеширует ключ:

$$\text{DAB} = 4 \times 1 \times 2 = 8.$$

Затем он пытается добавить значение `"pat"` в ячейку 8 нашей хеш-таблицы:

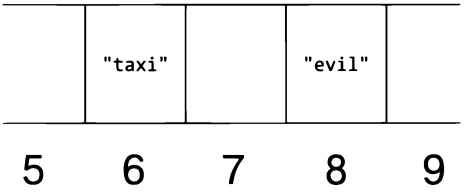


Но ячейка 8 уже содержит значение `"evil"`!

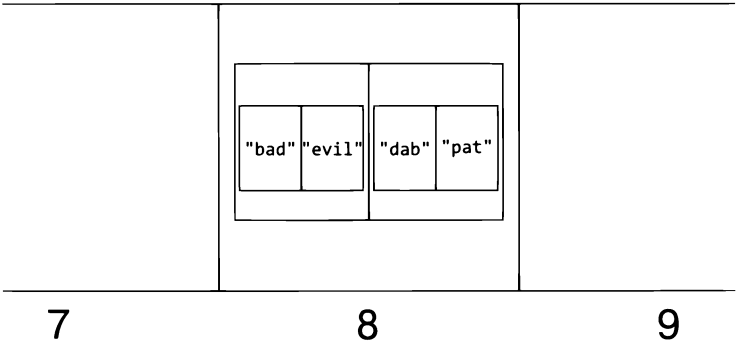
Попытка добавить данные в уже заполненную ячейку называется *коллизией*. К счастью, с ней можно справиться несколькими способами.

Один из классических подходов — *метод цепочек*. Когда возникает коллизия, вместо того чтобы поместить в ячейку какое-то *одно* значение, компьютер помещает в нее ссылку на массив.

Рассмотрим процесс сохранения данных в нашей хеш-таблице повнимательнее:



В нашем примере компьютер хочет добавить слово `"pat"` в ячейку 8, где уже есть значение `"evil"`. Для этого он заменяет содержимое ячейки 8 массивом:



В этом массиве есть подмассивы, в которых первое значение — это слово, а второе — его синоним.

Давайте посмотрим, как в этом случае работает поиск по хеш-таблице. Если мы ищем:

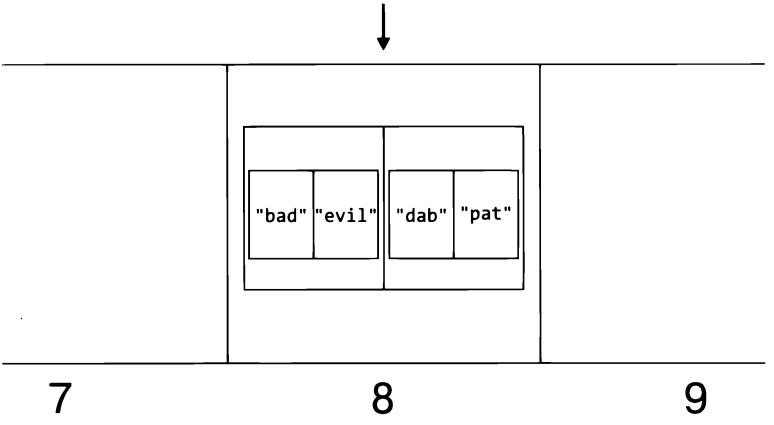
```
thesaurus["dab"]
```

компьютер выполняет следующие шаги.

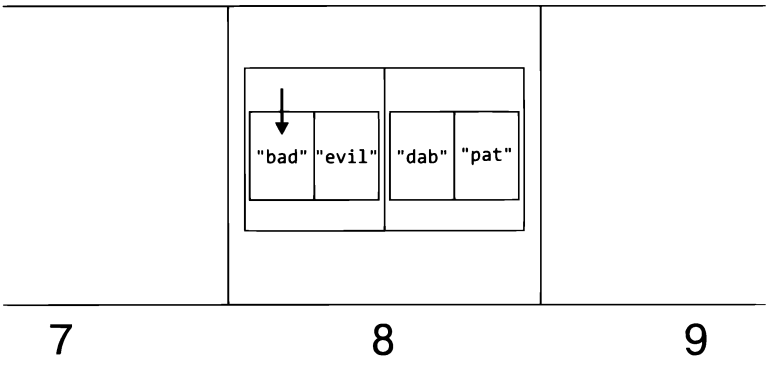
1. Хеширует ключ: DAB = 4 × 1 × 2 = 8.
2. Находит ячейку 8 и отмечает, что она содержит не одно значение, а массив массивов.
3. Осуществляет линейный поиск в массиве, проверяя индекс 0 каждого подмассива, пока не найдет наш ключ `"dab"`. Затем возвращает значение по индексу 1 соответствующего подмассива.

Представим этот процесс наглядно.

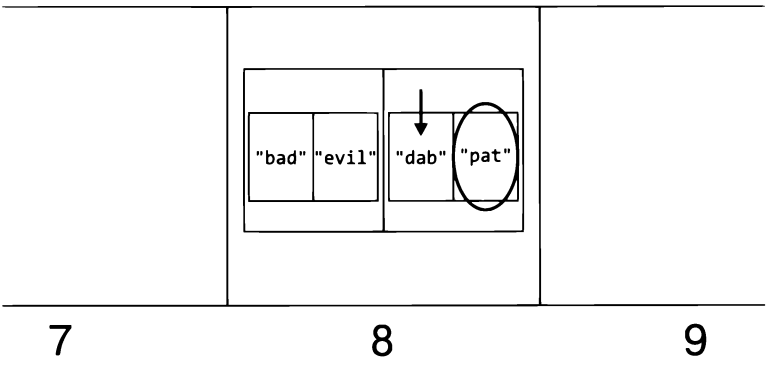
Результат хеширования слова DAB — 8, поэтому компьютер проверяет ячейку с этим индексом:



Ячейка 8 содержит массив массивов, поэтому компьютер выполняет линейный поиск в каждом подмассиве, начиная с индекса 0 первого:



Здесь нет искомого ключа "dab", поэтому компьютер переходит к индексу 0 следующего подмассива:



Мы обнаружили ключ "dab". Значение с индексом 1 этого подмассива ("pat") — именно то, которое мы ищем.

Когда компьютер находит ячейку со ссылкой на массив, на процесс поиска могут уйти дополнительные шаги — компьютеру придется провести линейный поиск в массиве с несколькими значениями. Если бы по какой-то причине все наши данные оказались в одной ячейке хеш-таблицы, то она была бы ничем не лучше массива. Выходит, что в худшем случае эффективность поиска в хеш-таблице равна O(N).

Поэтому очень важно проектировать хеш-таблицы так, чтобы в них возникало минимум коллизий, а поиск выполнялся за время O(1), а не O(N).

К счастью, большинство языков программирования заботятся обо всем этом за нас. Но, понимая, как именно работают хеш-таблицы, мы можем разобраться в том, как им удается достичь эффективности O(1).

Теперь обсудим, как сократить число коллизий.

Создание эффективной хеш-таблицы

На эффективность хеш-таблицы влияют три фактора:

- количество хранящихся данных;
- число доступных ячеек;
- используемая хеш-функция.

С первыми двумя факторами все понятно. Если у вас много данных и мало ячеек, то из-за большого числа коллизий хеш-таблица потеряет свою эффективность. А что насчет хеш-функции? Давайте разберемся.

Допустим, мы используем хеш-функцию, которая всегда возвращает значение от 1 до 9. Для этого она может преобразовать буквы в соответствующие числа и продолжать суммировать полученные значения, пока в итоге не получится одна цифра.

Например:

$$\text{PUT} = 16 + 21 + 20 = 57.$$

Поскольку 57 состоит более чем из одной цифры, хеш-функция раскладывает 57 на 5 и 7:

$$5 + 7 = 12.$$

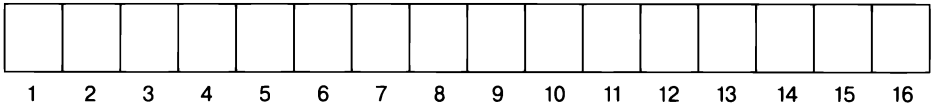
12 тоже состоит более чем из одной цифры, поэтому оно раскладывается на 1 и 2:

$$1 + 2 = 3.$$

Как итог: результат хеширования слова PUT — 3.

Эта функция *всегда* будет возвращать число от 1 до 9.

Вернемся к нашему примеру хеш-таблицы:



При использовании вышеописанной хеш-функции компьютер никогда не задействует ячейки с 10 по 16, даже если они есть: все данные будут помещены в ячейки 1–9.

Выходит, что хорошая хеш-функция должна распределять данные по всем доступным ячейкам. Чем лучше мы распределим данные, тем меньше возникнет коллизий.

Великий компромисс

Итак, мы выяснили, что эффективность хеш-таблицы зависит от числа коллизий. И, если подумать, то лучший способ их предотвратить — создать таблицу

с большим количеством ячеек. Представьте, что мы собираемся хранить в хеш-таблице всего пять элементов. Казалось бы, таблица с 1000 ячейками — отличный вариант, ведь в этом случае точно будет мало коллизий.

Но помимо предотвращения коллизий важно избегать и чрезмерного использования памяти.

Хотя создание хеш-таблицы на 1000 ячеек для хранения пяти элементов данных позволяет снизить число коллизий, память при этом будет использоваться максимально неэффективно.

Получается, хорошая хеш-таблица должна достичь *компромисса между предотвращением коллизий и потреблением памяти.*

Для этого программисты вывели эмпирическое правило: на каждые 7 элементов данных в хеш-таблице должно приходиться 10 ячеек (если вы планируете хранить 14 элементов, вам нужно 20 доступных ячеек и т. д.).

Такое соотношение данных и ячеек называется *коэффициентом заполнения* хеш-таблицы. Используя эту терминологию, мы бы сказали, что идеальный коэффициент заполнения — 0,7 (7 элементов / 10 ячеек).

Если вы изначально сохраните в хеш-таблице 7 фрагментов данных, то компьютер может выделить под нее 10 ячеек. Но по мере добавления других данных он будет расширять хеш-таблицу, создавая новые ячейки и изменяя хеш-функцию так, чтобы дополнительные данные равномерно распределялись по выделенным ячейкам.

Опять же, принцип работы хеш-таблицы во многом зависит от языка программирования. Он сам решает, каким должен быть ее размер, когда ее расширять и какую хеш-функцию использовать. Вы вполне можете рассчитывать на то, что в выбранном языке хеш-таблица уже реализована максимально эффективно.

Итак, мы разобрались в том, как работают хеш-таблицы, и поняли, что сложность поиска в них — O(1). Вскоре мы воспользуемся этим знанием, чтобы оптимизировать наш код.

Но сначала быстро рассмотрим несколько вариантов использования хеш-таблиц для организации данных.

Хеш-таблицы для организации данных

Хеш-таблицы хранят данные в виде пар значений, поэтому они бывают довольно полезны при организации информации.

Некоторые данные изначально идут в паре. Как, например, в случае с меню ресторана и тезаурусом: в меню название блюда идет в паре с его ценой, а в справочнике слова — в паре с их синонимами. В Python хеш-таблицы называются *словарями*, поскольку словарь — это распространенная форма парных данных: список слов с их определениями.

Другие примеры парных данных — результаты подсчета голосов, полученных разными кандидатами на выборах:

```
{"Candidate A" => 1402021, "Candidate B" => 2321443, "Candidate C" => 432}
```

Еще один пример — система управления запасами, которая отслеживает количество товаров в наличии:

```
{"Yellow Shirt" => 1203, "Blue Jeans" => 598, "Green Felt Hat" => 65}
```

Хеш-таблицы настолько хорошо подходят для хранения парных данных, что иногда мы даже можем использовать их для упрощения условной логики.

Возьмем, к примеру, функцию, которая возвращает значения распространенных кодов состояния HTTP:

```
def status_code_meaning(number)
  if number == 200
    return "OK"
  elsif number == 301
    return "Moved Permanently"
  elsif number == 401
    return "Unauthorized"
  elsif number == 404
    return "Not Found"
  elsif number == 500
    return "Internal Server Error"
  end
end
```

Если мы внимательно взглянем на этот код, то поймем, что условная логика вращается здесь вокруг парных данных: номеров кодов состояния и их соответствующих значений.

С помощью хеш-таблицы мы можем полностью исключить эту условную логику:

```
STATUS_CODES = {200 => "OK", 301 => "Moved Permanently",
                401 => "Unauthorized", 404 => "Not Found",
                500 => "Internal Server Error"}

def status_code_meaning(number)
  return STATUS_CODES[number]
end
```

Еще один распространенный способ использования хеш-таблиц — представление объектов с разными атрибутами. Например, так выглядит представление собаки:

```
{"Name" => "Fido", "Breed" => "Pug", "Age" => 3, "Gender" => "Male"}
```

Как видите, атрибуты — это своего рода парные данные, поскольку имя атрибута — это ключ, а сам он — значение. Поместив несколько хеш-таблиц в массив, мы можем создать целый список собак:

```
[
  {"Name" => "Fido", "Breed" => "Pug", "Age" => 3, "Gender" => "Male"},
  {"Name" => "Lady", "Breed" => "Poodle", "Age" => 6, "Gender" => "Female"},
  {"Name" => "Spot", "Breed" => "Dalmatian", "Age" => 2, "Gender" => "Male"}
]
```

Хеш-таблицы для ускорения выполнения кода

Хотя хеш-таблицы идеально подходят для хранения парных данных, их можно использовать и для ускорения выполнения кода, даже если эти данные не парные. Именно здесь начинается все самое интересное.

Вот простой массив:

```
array = [61, 30, 91, 11, 54, 38, 72]
```

Сколько шагов нужно выполнить, чтобы найти в нем определенное число?

Поскольку массив неупорядоченный, вам придется осуществить линейный поиск, который потребует выполнения N шагов. Мы уже говорили об этом в самом начале книги.

Но что будет, если мы используем некоторый код для преобразования этих чисел в вот такую хеш-таблицу:

```
hash_table = {61 => true, 30 => true, 91 => true, 11 => true, 54 => true,
38 => true, 72 => true}
```

Здесь мы сохранили каждое число в виде ключа вместе со связанным логическим значением `true`.

Как вы думаете, сколько теперь нужно шагов для нахождения определенного числа, которое хранится в этой хеш-таблице в качестве ключа?

С помощью следующего простого фрагмента кода:

```
hash_table[72]
```

я могу найти число 72 всего за один шаг.

То есть при выполнении поиска в хеш-таблице с числом 72 в качестве ключа я могу за один шаг определить, есть ли в этой таблице значение 72. Здесь все просто: если число 72 — это ключ, который есть в хеш-таблице, код возвратит значение `true`, связанное с этим ключом. Если же 72 не хранится в хеш-таблице в качестве ключа, код возвратит `nil` (разные языки возвращают разные значения при отсутствии заданного ключа. Ruby возвращает `nil`).

Так как поиск в хеш-таблице выполняется всего за шаг, я могу за один шаг найти в ней любое число (используемое в качестве ключа).

Это волшебство, не правда ли?

Преобразовав массив в хеш-таблицу, мы можем перевести алгоритм поиска из категории сложности O(*N*) в O(1).

Самое интересное здесь в том, что мы имеем дело *не* с парными данными, для организации которых лучше всего подходят хеш-таблицы, а со списком отдельных чисел.

Несмотря на то что мы присвоили значение каждому ключу, нам на самом деле не важно, что это за значение. Мы использовали для него слово `true`, но с тем же успехом могли бы выбрать любое произвольное значение.

Фокус в том, что, поместив каждое число в хеш-таблицу в качестве ключа, мы сможем найти любой из этих ключей всего за шаг. Если наш код возвращает какое-либо значение, мы точно знаем, что ключ есть в хеш-таблице, если же он возвращает `nil`, значит, искомого ключа нет.

Я называю этот способ *использованием хеш-таблицы в качестве индекса* (это мой термин). Благодаря оглавлению книги вы можете сразу узнать, описана ли в ней интересующая вас тема, а не слепо листать все издание. В вышеописанном примере мы создали хеш-таблицу, которая делает то же самое, что и оглавление, сообщая о том, есть ли конкретный элемент в исходном массиве или его нет.

Воспользуемся этим приемом для ускорения одного очень практичного алгоритма.

Подмножество массива

Допустим, нам нужно определить, служит ли один массив подмножеством другого. Для примера возьмем следующие два массива:

```
["a", "b", "c", "d", "e", "f"]
["b", "d", "f"]
```

Второй массив — подмножество первого, потому что каждое значение `["b", "d", "f"]` содержится в `["a", "b", "c", "d", "e", "f"]`.

Но в случае с массивами:

```
["a", "b", "c", "d", "e", "f"]
["b", "d", "f", "h"]
```

второй массив *не будет* подмножеством первого, так как содержит значение `"h"`, которого в первом массиве нет.

Как написать функцию, которая сравнивает два массива и сообщает о том, служит ли один из них подмножеством другого?

Один из способов сделать это — использовать вложенные циклы. Мы можем перебрать все элементы меньшего массива, запустив для каждого из них второй цикл, перебирающий все элементы большего. При обнаружении в меньшем массиве элемента, которого нет в большем, наша функция должна возвратить `false`. Если все циклы будут выполнены, значит, она не обнаружила таких элементов и код должен возвратить `true`.

Так выглядит реализация этого подхода на языке JavaScript:

```
function isSubset(array1, array2) {

  let largerArray;
  let smallerArray;

  // Определяем, какой массив меньше:
  if(array1.length > array2.length) {
    largerArray = array1;
    smallerArray = array2;
  } else {
    largerArray = array2;
    smallerArray = array1;
  }

  // Перебираем значения в меньшем массиве:
  for(let i = 0; i < smallerArray.length; i++) {

    // На время допускаем, что текущего значения
    // меньшего массива нет в большем
    let foundMatch = false;

    // Последовательно указывая на каждое значение меньшего массива,
    // перебираем все значения в большем:
    for(let j = 0; j < largerArray.length; j++) {

      // Если два значения равны, значит, текущее значение
      // меньшего массива есть в большем:
      if(smallerArray[i] === largerArray[j]) {
        foundMatch = true;
```

```
        break;
      }
    }

    // Если текущего значения меньшего массива нет
    // в большем, возвращаем false
    if(foundMatch === false) { return false; }
  }

  // Если все циклы выполнены, значит, все значения
  // меньшего массива есть в большем:
  return true;
}
```

Если мы проанализируем эффективность этого алгоритма, то выясним, что она равна O($N \times M$), поскольку число выполняемых им шагов равно произведению количества элементов в первом и во втором массивах.

Теперь воспользуемся мощью хеш-таблицы, чтобы оптимизировать наш алгоритм. Забудем о первоначальном подходе и начнем все сначала.

Новый подход таков: после определения большего и меньшего массивов мы запустим один цикл, в котором переберем все значения в большем массиве и сохраним их в хеш-таблице:

```
let hashTable = {};

for(const value of largerArray) {
  hashTable[value] = true;
}
```

Здесь мы создаем пустую хеш-таблицу внутри переменной `hashTable`, а затем перебираем все значения в большем массиве `largerArray`, добавляя их в хеш-таблицу в качестве ключа. Значением всех этих ключей будет `true`.

Например, массив `["a", "b", "c", "d", "e", "f"]` будет преобразован в следующую хеш-таблицу:

```
{"a": true, "b": true, "c": true, "d": true, "e": true, "f": true}
```

Она будет нашим «индексом», позволяющим в дальнейшем выполнять поиск конкретных элементов за один шаг.

А теперь самое интересное. Как только первый цикл завершится и у нас появится эта хеш-таблица, мы сможем запустить второй (не вложенный), перебирающий значения в *меньшем* массиве:

```
for(const value of smallerArray) {
  if(!hashTable[value]) { return false; }
}
```

Этот цикл просматривает каждый элемент меньшего массива `smallerArray` и проверяет, есть ли он в качестве ключа внутри хеш-таблицы `hashTable`. Как вы помните, эта таблица хранит в качестве ключей все элементы большего массива `largerArray`. И если мы обнаруживаем определенный элемент в `hashTable`, значит, он будет и в `largerArray`. А если мы не находим элемент в хеш-таблице, значит, его нет и в большем массиве.

Итак, мы проверяем, является ли каждый элемент меньшего массива `smallerArray` ключом в `hashTable`. Если нет, то его нет в большем массиве, а значит, меньший массив не служит подмножеством большего, и мы возвращаем `false` (но если этот цикл будет полностью выполнен, значит, меньший массив *является* подмножеством большего).

Объединим все это в одной функции:

```
function isSubset(array1, array2) {
  let largerArray;
  let smallerArray;
  let hashTable = {};

  // Определяем, какой массив меньше:
  if(array1.length > array2.length) {
    largerArray = array1;
    smallerArray = array2;
  } else {
    largerArray = array2;
    smallerArray = array1;
  }

  // Сохраняем все элементы большего массива largerArray внутри хеш-таблицы:
  for(const value of largerArray) {
    hashTable[value] = true;
  }

  // Перебираем все элементы меньшего массива smallerArray и возвращаем
  // false, если находим элемент, которого нет в hashTable:
  for(const value of smallerArray) {
    if(!hashTable[value]) { return false; }
  }

  // Если до этого момента код не возвратил значение false,
  // значит, все элементы меньшего массива есть в большем:
  return true;
}
```

Итак, сколько же шагов потребовал этот алгоритм? При создании хеш-таблицы мы перебрали все элементы *большего* массива один раз.

Мы перебрали и все элементы *меньшего* массива, выполняя всего один шаг при поиске каждого из них в хеш-таблице, ведь, как вы помните, поиск по таблице осуществляется за один шаг.

Если N — это общее количество элементов в обоих массивах, то сложность нашего алгоритма — $O(N)$, поскольку мы выполнили по одному шагу при обработке каждого элемента большего и меньшего массивов.

Этот алгоритм работает *намного* быстрее, чем первый, сложность которого была равна $O(N \times M)$.

Такой метод часто применяется в алгоритмах, требующих многократного поиска в массиве. То есть если ваш алгоритм будет искать значения внутри массива несколько раз, то каждый такой поиск будет требовать выполнения до N шагов. Но если у вашего массива будет хеш-таблица в качестве индекса, вы сможете свести каждый процесс поиска к выполнению одного шага.

Как я уже отмечал, этот метод особенно интересен тем, что при его использовании мы даже не имеем дело с парными данными: мы просто выясняем, есть ли тот или иной ключ в хеш-таблице. Если при использовании ключа для выполнения поиска в хеш-таблице мы получаем некоторое значение (любое), значит, искомый ключ есть в этой таблице.

Выводы

Хеш-таблицы — незаменимый инструмент для создания эффективного ПО. Эта структура данных позволяет производить чтение и вставку значений всего за шаг.

До сих пор мы анализировали разные структуры данных с точки зрения их эффективности и скорости. Но есть и такие, преимущества которых не ограничиваются этими показателями. В следующей главе мы рассмотрим две структуры данных, позволяющие улучшить чистоту кода и его сопровождаемость.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 8».

1. Напишите функцию, которая возвращает пересечение двух массивов — третий массив со всеми значениями из обоих исходных. Например, пересечение массивов `[1, 2, 3, 4, 5]` и `[0, 2, 4, 6, 8]` — это `[2, 4]`. Сложность вашей функции должна быть $O(N)$. Если ваш язык программирования предусматривает встроенный способ решения этой задачи, не используйте его. Создайте этот алгоритм самостоятельно.

2. Напишите функцию, которая принимает массив строк и возвращает первое повторяющееся значение. Например, в случае с массивом `["a", "b", "c", "d", "c", "e", "f"]` эта функция должна возвратить `"c"`, так как оно встречается в массиве более одного раза. Можете предположить, что массив содержит одну пару дубликатов. Убедитесь, что сложность этой функции — O(*N*).

3. Напишите функцию, которая принимает строку со всеми буквами алфавита, кроме одной, и возвращает эту недостающую букву. Например, строка `"the quick brown box jumps over a lazy dog"` содержит все буквы латинского алфавита, кроме `"f"`. Временная сложность вашей функции должна быть равна O(*N*).

4. Напишите функцию, которая возвращает первый *не*повторяющийся символ в строке. Например, в строке `"minimum"` есть два неповторяющихся символа — `"n"` и `"u"`, поэтому ваша функция должна возвратить `"n"`, так как он встречается первым. Временная сложность вашей функции должна быть равна O(*N*).

ГЛАВА 9

Создание чистого кода с помощью стеков и очередей

До сих пор при обсуждении структур данных мы сосредоточивались на том, как они влияют на *производительность* разных операций. Но умение работать с разными структурами данных позволяет вам создавать еще и более простой и удобочитаемый код.

В этой главе вы познакомитесь с двумя новыми структурами данных: стеками и очередями. Но, по правде говоря, это не будет для вас чем-то новым. Это просто массивы с некоторыми ограничениями. Но именно эти ограничения и делают их столь эффективными.

Стеки и очереди — отличные инструменты для качественной обработки временных данных. Это временные контейнеры, которые можно использовать, чтобы создавать отличные алгоритмы для чего угодно — от создания архитектуры операционной системы (ОС) до обработки заданий печати и обхода данных.

Временные данные как заказы еды в закусочной: запрос каждого клиента важен до тех пор, пока еда не будет приготовлена и подана — потом бланк заказа можно выбросить. Хранить эту информацию вовсе не обязательно. Временные данные — это информация, которая становится бесполезной после обработки, поэтому потом от нее можно смело избавиться.

Стеки и очереди обрабатывают именно такие данные, уделяя особое внимание *порядку* их обработки.

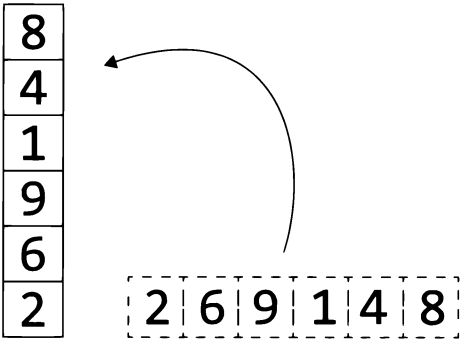
Стеки

Стек хранит данные так же, как и массив. Это просто список элементов. Но у стеков есть три ограничения:

- данные можно вставлять только в конец стека;
- удалить из стека можно только последний элемент;
- прочитать можно только последний элемент стека.

Стек похож на стопку тарелок: вы не можете увидеть лицевую сторону той, что находится под самой верхней. Точно так же добавить новую тарелку вы можете только на вершину стопки, а убрать — только самую верхнюю (пытаясь поступить иначе, вы просто разобьете посуду). В большинстве книг по информатике конец стека называется его *вершиной*, а начало — *дном* или *низом*.

Мы будем придерживаться этой терминологии и изображать стеки на диаграммах в виде вертикальных массивов:



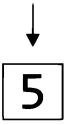
Как видите, первый элемент массива становится дном стека, в последний — его вершиной.

Хотя свойственные стеку ограничения могут поначалу показаться, скажем так, давящими, позже вы убедитесь в том, насколько они на самом деле полезны.

Начнем разбираться в принципе работы этой структуры данных с пустого стека.

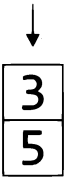
Вставка нового значения в стек называется *проталкиванием* (push). Это очень похоже на добавление тарелки на вершину стопки блюд.

Давайте втолкнем в стек значение 5:

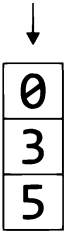


Пока не происходит ничего необычного. Мы просто вставляем элемент данных в конец массива.

Теперь втолкнем в стек значение 3:



Затем добавим в него 0:

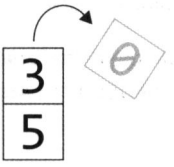


Обратите внимание, что мы всегда помещаем данные на вершину (то есть в конец) стека. Поместить 0 в самый низ или в середину стека мы бы не смогли, потому что при использовании стека данные можно добавлять только на его вершину.

Удаление элемента из стека называется *выталкиванием* (pop). Из-за свойственных стеку ограничений мы можем удалять данные только с его вершины.

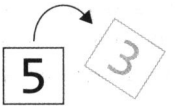
Вытолкнем несколько элементов из нашего стека.

Сначала выталкиваем 0:



Теперь в нашем стеке есть только два элемента: 5 и 3.

Затем выталкиваем 3:



В нашем стеке осталось только 5:



Для описания операций со стеком используется аббревиатура *LIFO*, которая расшифровывается как Last In, First Out («последним пришел — первым ушел»). Она означает, что *последний* элемент, *помещенный* в стек, всегда будет *первым*, *извлекаемым* из него. Здесь отлично подойдет пример с ленивыми студентами, которые всегда приходят на занятия последними, а уходят первыми.

Абстрактные типы данных

Стек не встроен в большинство языков программирования как тип данных или класс, поэтому вы должны реализовать его самостоятельно. Это резко отличает стек от массивов, которые доступны в большинстве языков.

Поэтому, чтобы создать стек, вам придется использовать одну из встроенных структур данных для фактического хранения значений. Вот один из способов реализации стека на языке Ruby, в котором используется массив:

```
class Stack
  def initialize
    @data = []
  end

  def push(element)
    @data << element
  end

  def pop
    @data.pop
  end

  def read
    @data.last
  end
end
```

Эта реализация стека хранит данные в массиве `@data`.

При каждой инициализации стека мы автоматически создаем пустой массив с помощью кода `@data = []`. Еще в нашем стеке есть методы, которые вталкивают (`push`) новые элементы в массив `@data`, выталкивают (`pop`) и считывают (`read`) элементы из него.

Но, окружив массив классом `Stack`, мы создали интерфейс, который ограничивает способы взаимодействия пользователя с этим массивом. Обычно мы можем считывать значения по любому индексу массива, но при взаимодействии с массивом через интерфейс стека мы можем прочитать только последний элемент. То же касается вставки и удаления элементов данных.

Стек отличается от массива. Массив встроен в большинство языков и напрямую взаимодействует с памятью компьютера, а стек — это набор правил и процессов, определяющих наш способ взаимодействия с массивом для достижения определенного результата.

По сути, стеку все равно, *какая* структура данных лежит в его основе. Его заботит лишь наличие списка элементов, работающих по принципу LIFO. По этой причине стек относится к так называемому *абстрактному типу данных*. Это своеобразная структура, которая представляет собой набор теоретических правил, применяющихся к какой-то другой встроенной структуре данных.

Множество, с которым вы познакомились в главе 1, — еще один пример абстрактного типа данных. В одних реализациях множеств могут использоваться массивы, а в других — хеш-таблицы. Но само множество — это просто теоретическая концепция, под которой понимается список неповторяющихся элементов данных.

Многие структуры, с которыми вы познакомитесь позже, относятся к абстрактным типам данных. По сути, это будут фрагменты кода, написанного поверх других встроенных структур данных.

Важно отметить, что к абстрактным типам может относиться даже встроенная структура данных. Если язык программирования реализует собственный класс `Stack`, это не отменяет того факта, что сам стек позволяет использовать разные структуры данных в качестве основы.

Стек в действии

Хотя стек обычно не используется для долговременного хранения данных, он может быть отличным инструментом обработки временных данных в рамках разных алгоритмов. Рассмотрим пример.

Давайте создадим начало линтера JavaScript — программы, которая проверяет код JavaScript программиста и синтаксическую корректность каждой строки. Это сложная задача, так как линтер должен проверять много разных аспектов синтаксиса.

Наш будет проверять только открывающие и закрывающие скобки, включая круглые, квадратные и фигурные. Именно они часто становятся причинами досадных синтаксических ошибок.

Чтобы решить эту проблему, сначала определим, какие синтаксические ошибки могут возникнуть при использовании скобок. Если мы проанализируем этот вопрос, то обнаружим три типа ошибок.

Одна из них возникает тогда, когда открывающая скобка не сопровождается соответствующей закрывающей:

```
(var x = 2;
```

Мы будем называть это синтаксической ошибкой первого типа.

Еще одна ошибка возникает, когда закрывающая скобка не сопровождается соответствующей открывающей:

```
var x = 2;)
```

Ее мы будем называть синтаксической ошибкой второго типа.

Синтаксическая ошибка третьего типа возникает, когда закрывающая скобка не относится к тому же *типу*, что и предшествующая ей открывающая:

```
(var x = [1, 2, 3)];
```

Здесь для каждой открывающей круглой и квадратной скобки есть соответствующая закрывающая, но закрывающая круглая скобка находится не на месте, поскольку ей предшествует открывающая квадратная.

Как же реализовать алгоритм, проверяющий строку кода JavaScript на предмет наличия синтаксических ошибок, связанных с использованием скобок? Именно здесь нам может пригодиться стек, позволяющий реализовать прекрасный алгоритм линтинга, который работает так.

Мы подготавливаем пустой стек и считываем каждый символ слева направо, придерживаясь следующих правил.

1. Если мы встречаем символ, но не скобку (круглую, квадратную или фигурную), то просто игнорируем его и двигаемся дальше.

2. Если мы встречаем *открывающую* скобку, то вталкиваем ее в стек. Если в стеке есть открывающая скобка, значит, мы ожидаем найти соответствующую ей закрывающую.
3. Если мы находим *закрывающую* скобку, то выталкиваем верхний элемент из стека и проверяем его:
   - если извлеченный элемент (всегда открывающая скобка) не соответствует текущей закрывающей скобке, значит, мы столкнулись с синтаксической ошибкой третьего типа;
   - если мы не смогли извлечь элемент, потому что стек был пуст, значит, перед текущей закрывающей скобкой нет соответствующей открывающей. Это синтаксическая ошибка второго типа;
   - если извлеченный элемент *соответствует* текущей закрывающей скобке, значит, открывающая была закрыта правильно и мы можем продолжить синтаксический анализ строки кода JavaScript.
4. Если после анализа всей строки в стеке что-то осталось, значит, в строке есть открывающая скобка без соответствующей закрывающей и мы столкнулись с синтаксической ошибкой первого типа.

Посмотрим, как это работает на конкретном примере:

```
(var x = {y: [1, 2, 3]})
```

После подготовки пустого стека мы начинаем считывать каждый символ слева направо.

Шаг 1: считываем первый символ — открывающую круглую скобку:

```
↓
(var x = {y: [1, 2, 3]})
```

Шаг 2: так как это открывающая скобка, мы помещаем ее в стек:

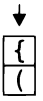


Игнорируем символы `var x =`, так как это не скобки.

Шаг 3: встречаем следующую открывающую скобку, на этот раз фигурную:

```
          ↓
(var x = {y: [1, 2, 3]})
```

Шаг 4: помещаем ее в стек:



Символы `y:` мы игнорируем.

Шаг 5: находим открывающую квадратную скобку:

```
          ↓
(var x = {y: [1, 2, 3]})
```

Шаг 6: добавляем ее в стек:



Символы `1, 2, 3` игнорируем.

Шаг 7: находим первую закрывающую скобку, квадратную:

```
                   ↓
(var x = {y: [1, 2, 3]})
```

Шаг 8: выталкиваем из стека верхний элемент, *открывающую* квадратную скобку:



Поскольку закрывающая квадратная скобка соответствует этому верхнему элементу стека, ошибки нет, и мы можем продолжить анализ.

Шаг 9: встречаем закрывающую фигурную скобку:

```
                   ↓
(var x = {y: [1, 2, 3]})
```

Шаг 10: выталкиваем из стека верхний элемент:



Им оказывается открывающая фигурная скобка, соответствующая текущей закрывающей.

Шаг 11: находим закрывающую круглую скобку:

```
                          ↓
(var x = {y: [1, 2, 3]})
```

Шаг 12: выталкиваем из стека последний элемент — соответствующую открывающую круглую скобку. Значит, ошибок пока не обнаружено.

Мы проанализировали всю строку кода, а стек пуст, поэтому линтер может прийти к выводу, что в строке нет синтаксических ошибок (связанных с использованием открывающих и закрывающих скобок).

Программная реализация: линтер на базе стека

Вот реализация описанного выше алгоритма на языке Ruby. Обратите внимание на то, что в ней используется приведенная ранее реализация класса `Stack`:

```
class Linter

  def initialize
    # Используем в качестве стека простой массив:
    @stack = Stack.new
  end

  def lint(text)
    # Запускаем цикл, который считывает каждый символ в нашем тексте:
    text.each_char do |char|

        # Если символ - открывающая скобка:
        if is_opening_brace?(char)

          # Вталкиваем его в стек:
          @stack.push(char)

        # Если символ - закрывающая скобка:
        elsif is_closing_brace?(char)

          # Выталкиваем верхний элемент из стека:
          popped_opening_brace = @stack.pop

          # Если при попытке извлечения элемента из стека мы получили
          # значение nil, значит, стек пуст, а в строке
          # нет открывающей скобки:
          if !popped_opening_brace
            return "#{char} doesn't have opening brace"
          end

          # Если тип извлеченной из стека открывающей скобки не соответствует
```

```
        # типу текущей закрывающей, выводим сообщение об ошибке:
        if is_not_a_match(popped_opening_brace, char)
          return "#{char} has mismatched opening brace"
        end
      end
    end

    # Если мы доходим до конца строки, а в стеке что-то остается:
    if @stack.read

      # Значит, в строке есть открывающая скобка без соответствующей ей
      # закрывающей, поэтому мы выводим сообщение об ошибке:
      return "#{@stack.read} does not have closing brace"
    end

    # Возвращаем значение true, если в строке нет ошибок:
    return true
  end

  private

  def is_opening_brace?(char)
    ["(", "[", "{"].include?(char)
  end

  def is_closing_brace?(char)
    [")", "]", "}"].include?(char)
  end

  def is_not_a_match(opening_brace, closing_brace)
    closing_brace != {"(" => ")", "[" => "]", "{" => "}"}[opening_brace]
  end
end
```

Метод `lint` принимает строку с кодом JavaScript и перебирает все ее символы с помощью фрагмента:

```
text.each_char do |char|
```

Если мы обнаруживаем открывающую скобку, то вталкиваем ее в стек:

```
if is_opening_brace?(char)
  @stack.push(char)
```

Обратите внимание, что мы используем вспомогательный метод `is_opening_brace?`, который проверяет, является ли символ открывающей скобкой. Он определен так:

```
["(", "[", "{"].include?(char)
```

Обнаружив закрывающую скобку, мы выталкиваем верхний элемент из стека и сохраняем его в переменной `popped_opening_brace`:

```
popped_opening_brace = @stack.pop
```

В нашем стеке хранятся только открывающие скобки, поэтому извлекаемый из него элемент всегда будет открывающей скобкой какого-то типа.

Иногда стек может быть пуст. Тогда при попытке извлечь из него элемент код вернет значение `nil`. Это будет означать, что мы столкнулись с синтаксической ошибкой второго типа:

```
if !popped_opening_brace
  return "#{char} doesn't have opening brace"
end
```

При обнаружении ошибки в процессе анализа мы возвращаем простую строку с сообщением о ней.

После удачного извлечения открывающей скобки из стека мы проверяем ее на предмет соответствия текущей закрывающей. Если их тип не совпадает, значит, мы столкнулись с синтаксической ошибкой третьего типа:

```
if is_not_a_match(popped_opening_brace, char)
  return "#{char} has mismatched opening brace"
end
```

(Вспомогательный метод `is_not_a_match` будет определен в коде позже.)

Наконец, по завершении анализа строки мы проверяем с помощью фрагмента `@stack.read`, не остались ли в стеке открывающие скобки. Если да, значит, в строке есть незакрытая скобка — и мы выдаем сообщение о синтаксической ошибке первого типа:

```
if @stack.read
  return "#{@stack.read} does not have closing brace"
end
```

Если в коде JavaScript нет ошибок, возвращаем `true`.

Мы можем использовать наш класс `Linter` так:

```
linter = Linter.new
puts linter.lint("( var x = { y: [1, 2, 3] } )")
```

Здесь код JavaScript верный, поэтому программа возвращает `true`.

Но если мы введем строку с ошибкой, например, без открывающей скобки:

```
"var x = { y: [1, 2, 3] })"
```

то получим сообщение об ошибке: `) doesn't have opening brace`.

В этом примере мы использовали стек для реализации линтера с очень изящным алгоритмом. Но если в основе стека лежит массив, зачем вообще его использовать? Разве мы не могли бы решить ту же задачу с помощью массива?

О важности ограниченных структур данных

Если стек — это просто ограниченная версия массива, значит, массив может делать все то же, что и стек. А раз так, то в чем же преимущество стека?

Ограниченные структуры данных, такие как стек (и очередь, о которой мы поговорим далее), важны по нескольким причинам.

Во-первых, используя ограниченные структуры данных, мы можем предотвратить возможные ошибки. Например, алгоритм линтинга работает корректно, только если мы извлекаем из стека верхние элементы. Если программист случайно напишет код, извлекающий элементы из середины массива, то алгоритм перестанет эффективно работать. Используя стек, мы можем извлекать только элементы, которые находятся на вершине.

Во-вторых, такие структуры данных предоставляют нам новую мысленную модель решения задач. К примеру, стек дает нам представление о принципе LIFO («последним пришел — первым ушел»), который мы потом можем применить для выполнения разных задач, вроде создания вышеописанного линтера.

А после знакомства с принципом работы стека мы начнем писать качественный и понятный другим разработчикам код. Увидев в алгоритме стек, они сразу поймут, что работа алгоритма основана на принципе LIFO.

Итак, стеки — идеальный выбор для обработки данных по принципу «последним пришел — первым ушел». Например, они подойдут для реализации функции отмены последнего действия в текстовом процессоре. Когда пользователь вводит текст, мы отслеживаем нажатия клавиш, помещая их в стек. Затем, когда нажимается кнопка отмены, мы извлекаем из стека данные о последней клавише и удаляем из документа соответствующий символ. После этого на вершине стека оказывается предпоследнее нажатие клавиши, которое при необходимости тоже можно отменить.

Очереди

Очередь — это еще одна структура для обработки временных данных. Она похожа на стек, только обрабатывает данные в другом порядке, но, как и стек, относится к абстрактным типам данных.

Очередь похожа на обычную толпу людей перед кассой в кинотеатре: первый человек у кассы выходит из очереди и входит в кинозал. Точно так же и первый элемент, добавленный в очередь, первым удаляется из нее. Вот почему программисты говорят, что очереди работают по принципу *FIFO* («First In, First Out», «первым пришел — первым ушел»).

Как и обычная вереница из людей, очередь изображается горизонтально. Начало очереди принято называть *головой*, а конец — *хвостом*.

Как и стеки, очереди — это массивы с тремя ограничениями:

- данные могут вставляться только в *конец* очереди (как и в случае со стеком);
- удалить из очереди можно только первый элемент (в отличие от стека);
- прочитать можно только первый элемент очереди (в отличие от стека).

Давайте разберемся, как работает очередь, на конкретном примере, начиная с пустой очереди.

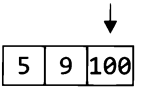
Сначала вставляем в очередь значение 5 (вставку нового значения обычно называют *постановкой в очередь* (enqueue), но мы будем использовать эти термины взаимозаменяемо):



Затем мы вставляем 9:

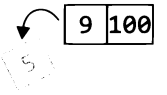


Потом — 100:

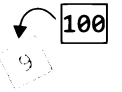


До сих пор очередь вела себя так же, как стек. Но *удаление элементов из очереди* (dequeue) происходит в обратном порядке — начиная с *головы*.

Чтобы удалить данные, мы должны начать с 5, так как это значение находится в начале очереди:



Затем мы удаляем 9:



Теперь в очереди остался только один элемент — значение 100.

Реализация очереди

Я уже говорил, что очередь относится к абстрактному типу данных. Как и многие из них, она может не быть реализована в том или ином языке программирования.

Так выглядит реализация очереди на языке Ruby:

```
class Queue
  def initialize
    @data = []
  end

  def enqueue(element)
    @data << element
  end

  def dequeue
    # Метод shift языка Ruby удаляет и
    # возвращает первый элемент массива:
    @data.shift
  end

  def read
    @data.first
  end
end
```

Опять же, с помощью класса `Queue` вокруг массива мы создали интерфейс, который ограничивает наше взаимодействие с данными, позволяя обрабатывать их только определенным образом. С помощью метода `enqueue` можно вставлять данные в конец массива, а с помощью `dequeue` — удалять из массива первый элемент. Метод `read` позволяет считывать значение первого элемента массива.

Очередь в действии

Очереди помогают выполнить множество задач — от обработки заданий печати и до организации фоновых рабочих процессов веб-приложений.

Допустим, мы разрабатываем на языке Ruby простой интерфейс для принтера, который должен принимать задания печати от разных компьютеров в сети. При этом мы хотим гарантировать, что документы будут печататься в порядке, в котором они были получены.

В следующем коде используется описанная выше реализация класса `Queue`:

```
class PrintManager

  def initialize
    @queue = Queue.new
  end

  def queue_print_job(document)
    @queue.enqueue(document)
  end

  def run
    # При каждом выполнении этого цикла мы считываем
    # документ в начале очереди:
    while @queue.read
      # Удаляем документ из очереди и печатаем его:
      print(@queue.dequeue)
    end
  end

  private

  def print(document)
    # Здесь находится код для запуска реального принтера.
    # В целях демонстрации мы будем выводить данные на экран
    puts document
  end

end
```

Мы можем использовать этот класс так:

```
print_manager = PrintManager.new
print_manager.queue_print_job("First Document")
print_manager.queue_print_job("Second Document")
print_manager.queue_print_job("Third Document")
print_manager.run
```

Каждый раз при вызове `queue_print_job` мы добавляем в очередь «документ» (здесь он представлен строкой):

```
def queue_print_job(document)
  @queue.enqueue(document)
end
```

При вызове функции `run` мы «печатаем» документы в порядке их получения:

```
def run
  while @queue.read
    print(@queue.dequeue)
  end
end
```

Обратите внимание, как мы удаляем из очереди каждый документ при его печати.

Если мы запустим код выше, программа выведет на экран три документа в порядке, в котором они были получены:

```
First Document
Second Document
Third Document
```

Это очень упрощенный пример, который не учитывает нюансов работы настоящей системы печати, он отражает базовый принцип использования очереди, которая вполне может применяться для построения такой системы.

Кроме того, очередь — идеальный инструмент для обработки асинхронных запросов: она гарантирует, что запросы будут обрабатываться в том порядке, в котором были получены. Часто она применяется и для моделирования реальных сценариев, где события должны происходить в определенном порядке. Например, как в случае с самолетами, ожидающими вылета, и пациентами, которые ждут врача.

Выводы

Как вы убедились, стеки и очереди позволяют создавать максимально качественные алгоритмы.

Теперь вы можете приступить к изучению рекурсии, которая требует использования стека и лежит в основе множества продвинутых и сверхэффективных алгоритмов, о которых я расскажу в следующих главах.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 9».

1. Если бы вы создавали ПО для колл-центра, которое ставит входящие звонки на удержание, а затем перенаправляет их следующему доступному оператору, вы бы использовали стек или очередь?
2. Представьте, что вы помещаете в стек числа в следующем порядке: 1, 2, 3, 4, 5, 6, а затем вытолкнете из него два элемента. Какое значение вы сможете прочесть из этого стека в итоге?
3. Представьте, что помещаете в очередь числа в следующем порядке: 1, 2, 3, 4, 5, 6, а затем извлекаете из нее два элемента. Какое значение вы сможете прочесть из очереди в итоге?
4. Напишите функцию, которая использует стек для обращения строки (то есть для преобразования `"abcde"` в `"edcba"`). Можете использовать нашу более раннюю реализацию класса `Stack`.

ГЛАВА 10

Рекурсивно рекурсируем с помощью рекурсии

Рекурсия — это ключевая концепция в информатике, открывающая путь к созданию более совершенных алгоритмов, с которыми вы познакомитесь далее. Если правильно ее использовать, то некоторые задачи будут выполняться максимально просто, как по волшебству.

Но, прежде чем мы погрузимся в эту тему, предлагаю вам ответить на вопрос.

Что произойдет при вызове функции `blah()`?

```
function blah() {
  blah();
}
```

Как вы, наверное, догадались, ее вызов будет повторяться бесконечно, так как функция `blah()` вызывает саму себя — функцию `blah()`, которая, в свою очередь, вызывает саму себя и т. д.

Рекурсия — это термин, обозначающий функцию, вызывающую саму себя. На самом деле бесконечная рекурсия, как в примере выше, совершенно бесполезна. Но при правильном использовании рекурсия может быть весьма мощным инструментом.

Рекурсия вместо цикла

Допустим, вы работаете в НАСА и вам нужно написать функцию, осуществляющую обратный отсчет перед запуском космического корабля. Она должна принимать число, например 10, и отображать числа от 10 до 0.

Прямо сейчас попробуйте реализовать эту функцию на любом языке и, как только закончите, продолжайте читать.

Скорее всего, вы написали простой цикл вроде этого фрагмента кода JavaScript:

```
function countdown(number) {
  for(let i = number; i >= 0; i--) {
    console.log(i);
  }
}

countdown(10);
```

Это неплохая реализация. Но вам не приходило в голову, что вы вообще могли бы обойтись без цикла?

Как?

Вместо этого можно использовать рекурсию.

Вот наша первая попытка применения рекурсии для реализации функции обратного отсчета:

```
function countdown(number) {
  console.log(number);

  countdown(number - 1);
}
```

Разберем работу этого кода по шагам.

Шаг 1: вызываем функцию `countdown(10)`. Значение аргумента `number` равно 10, поэтому отсчет начинается с него.

Шаг 2: значение `number` (которое сейчас равно 10) отображается в консоли.

Шаг 3: прежде чем завершить работу, функция `countdown` вызывает `countdown(9)`, так как `number` – 1 равно 9.

Шаг 4: в рамках вызова функции `countdown(9)` значение `number` (которое сейчас равно 9) отображается в консоли.

Шаг 5: перед завершением работы функция `countdown(9)` вызывает `countdown(8)`.

Шаг 6: в рамках вызова функции `countdown(8)` в консоли отображается значение 8.

Прежде чем мы продолжим анализировать работу кода, обратите внимание на то, как мы используем рекурсию. Не прибегая ни к каким циклам, а просто позволяя функции `countdown` многократно вызывать саму себя, мы можем вести обратный отсчет от 10 и отображать каждое число в консоли.

Рекурсию можно использовать почти везде, где можно использовать циклы. Но то, что вы *можете* использовать рекурсию, не значит, что вам *следует* это делать. Рекурсия — это инструмент, позволяющий писать изящный код. Рекурсивный подход из прошлого примера не превосходит классический цикл `for` в плане качества и эффективности. Но чуть позже мы рассмотрим примеры, где рекурсия раскрывается во всей красе. А пока продолжим изучать принцип ее работы.

Базовый случай

Вернемся к нашей функции `countdown`, пропустив для краткости несколько шагов...

Шаг 21: вызываем `countdown(0)`.

Шаг 22: отображаем в консоли значение `number` (то есть 0).

Шаг 23: вызываем `countdown(-1)`.

Шаг 24: отображаем в консоли значение `number` (−1).

Посмотрите-ка, что-то пошло не так: наша функция выводит на экран бесконечную последовательность отрицательных чисел.

Чтобы ее улучшить, мы должны найти способ завершить обратный отсчет на значении 0 и предотвратить бесконечные рекурсивные вызовы.

Решить проблему поможет добавление условного оператора, который гарантирует, что после обнуления `number` функция `countdown()` перестанет вызываться:

```
function countdown(number) {
  console.log(number);
  if(number === 0) {
    return;
  } else {
    countdown(number - 1);
  }
}
```

Теперь, когда `number` достигнет 0, наш код вместо очередного вызова `countdown()` просто завершит работу функции.

Случай, в котором функция *перестает* вызывать саму себя, называется *базовым*. Базовый случай для нашей функции `countdown()`— значение `number`, равное 0. Очень важно предусмотреть для каждой рекурсивной функции хотя бы один базовый случай, чтобы она не вызывала саму себя бесконечно.

Чтение рекурсивного кода

Чтобы разобраться в теме рекурсии, нужны время и практика, но в итоге вы все же научитесь *читать* и *писать* рекурсивный код. Читать его проще, чем писать, поэтому начнем с азов — займемся вычислением факториалов.

Показать, что такое *факториал,* проще всего на конкретных примерах.

Факториал числа 3:

$3 \times 2 \times 1 = 6$.

Факториал числа 5:

$5 \times 4 \times 3 \times 2 \times 1 = 120$.

И так далее. Вот рекурсивная реализация на языке Ruby, возвращающая факториал числа:

```
def factorial(number)
  if number == 1
    return 1
  else
    return number * factorial(number - 1)
  end
end
```

Этот код может показаться несколько запутанным. Чтобы понять принцип его работы, советую проанализировать его так.

1. Определите базовый случай.
2. Найдите значение функции для него.
3. Определите «предпоследний» случай, который был перед базовым.
4. Найдите значение функции для него.
5. Повторите этот процесс для случая, предшествующего тому, который вы проанализировали только что.

Применим этот подход к коду выше. Если мы внимательно на него посмотрим, то заметим, что у нас есть два пути:

```
if number == 1
  return 1
else
  return number * factorial(number - 1)
end
```

Мы видим, что здесь происходит рекурсия: функция `factorial` вызывает саму себя:

```
else
  return number * factorial(number - 1)
end
```

Поэтому следующий фрагмент кода должен относиться к базовому случаю, так как здесь функция себя *не* вызывает:

```
if number == 1
  return 1
```

Итак, мы можем сделать вывод, что базовый случай — значение `number`, равное 1.

Теперь определим, что возвращает функция в базовом случае `factorial(1)`. Код выглядит так:

```
if number == 1
  return 1
```

Здесь все довольно просто, так как в базовом случае никакой рекурсии не происходит. При вызове `factorial(1)` наш метод возвращает значение 1. Итак, отметим это в блокноте:

factorial(1) возвращает 1

Теперь перейдем к анализу следующего случая — `factorial(2)`, которому соответствует фрагмент кода:

```
else
  return number * factorial(number - 1)
end
```

При вызове `factorial(2)` код возвратит результат операции `2 * factorial(1)`. Чтобы его вычислить, нам нужно знать результат вызова `factorial(1)`. Взглянув на свои записи, вы увидите, что он равен 1. Получается, что результат операции `2 * factorial(1)` равен $2 \times 1 = 2$.

Запишите это в блокнот:

factorial(2) возвращает 2
factorial(1) возвращает 1

А что произойдет при вызове `factorial(3)`? За него отвечает тот же фрагмент кода:

```
else
return number * factorial(number - 1)
end
```

При вызове `factorial(3)` функция вернет результат `3 * factorial(2)`. Какое значение возвращается при вызове `factorial(2)`? Вам не нужно вычислять его снова — оно уже записано в блокноте! Результат этого вызова — 2.

Итак, при вызове `factorial(3)` функция возвратит 6 (потому что 3 × 2 = 6). Запишите и это:

factorial(3) возвращает 6
factorial(2) возвращает 2
factorial(1) возвращает 1

А теперь попытайтесь самостоятельно определить результат вызова `factorial(4)`.

Как видите, анализ с базового случая — отличный способ разобраться в работе рекурсивного кода.

Рекурсия глазами компьютера

Чтобы вникнуть в суть рекурсии, нужно увидеть, как компьютер обрабатывает рекурсивную функцию. Люди могут рассуждать о рекурсии, черкая в блокноте карандашом, но компьютеру приходится проделывать сложную работу при вызове функции из нее самой.

Итак, давайте разберемся, как компьютер реализует рекурсивную функцию.

Допустим, мы вызываем функцию `factorial(3)`. Поскольку значение 3 — это не базовый случай, компьютер достигает строки кода:

```
return number * factorial(number - 1)
```

которая запускает функцию `factorial(2)`.

И тут возникает загвоздка. Успевает ли компьютер завершить выполнение `factorial(3)`к моменту запуска `factorial(2)`?

В этом и заключается вся сложность рекурсии с точки зрения компьютера. Выполнение функции `factorial(3)` завершается только по достижении ключевого слова `end`. Возникает странная ситуация: компьютер еще не завершил вы-

полнение `factorial(3)`, но уже запускает `factorial(2)`, *продолжая при этом выполнять* функцию `factorial(3)`.

И на `factorial(2)` история не заканчивается, так как эта функция в свою очередь запускает `factorial(1)`. Получается какое-то безумие: в процессе выполнения `factorial(3)` компьютер вызывает функцию `factorial(2)`, выполняя которую, запускает `factorial(1)`. Выходит, что `factorial(1)` выполняется в рамках вызова *как* `factorial(2)`, *так и* `factorial(3)`.

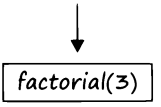
Как компьютер все это отслеживает? Ему нужно как-то запомнить, что по завершении выполнения функции `factorial(1)` ему необходимо вернуться и закончить выполнение `factorial(2)`, а затем не забыть завершить выполнение `factorial(3)`.

Стек вызовов

Помните стеки, о которых мы говорили в главе 9? Компьютер использует именно их, чтобы отслеживать функции, которые вызывает в данный момент. Такие стеки называются *стеками вызовов*.

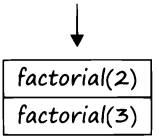
Вот как работает стек вызовов в нашем примере с функцией `factorial`.

Первым делом компьютер вызывает `factorial(3)`. Но до завершения выполнения этого метода вызывается `factorial(2)`. Чтобы не забыть, что работа функции `factorial(3)` еще не завершена, компьютер помещает информацию об этом в стек вызовов:



Это означает, что компьютер выполняет функцию `factorial(3)` (на самом деле он еще вынужден сохранять данные о выполняемой в данный момент строке кода и некоторые другие сведения, вроде значения переменных, но, чтобы не перегружать диаграммы, я их не указываю).

Затем компьютер приступает к выполнению функции `factorial(2)`, которая вызывает `factorial(1)`. Но, прежде чем приступить к выполнению `factorial(1)`, он должен запомнить, что работа функции `factorial(2)` еще не завершена, поэтому он помещает в стек вызовов соответствующую информацию:

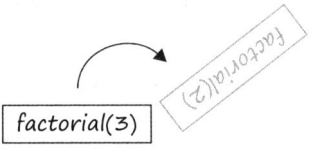


Теперь компьютер выполняет функцию `factorial(1)`. Поскольку значение 1 — это базовый случай, `factorial(1)` завершается без повторного вызова метода `factorial`.

По завершении выполнения `factorial(1)` компьютер проверяет стек вызовов, чтобы узнать о функциях, которые еще выполняются. Если в стеке вызовов что-то есть, значит, компьютеру еще нужно завершить выполнение запущенных ранее функций.

Как вы помните, из стека можно извлечь лишь верхний элемент. Для рекурсии это подходит идеально, так как верхний элемент будет *самой последней вызванной функцией*, которую компьютер должен обработать следующей. Это ситуация типа LIFO: функция, которая была вызвана последней, должна быть выполнена в первую очередь.

Следующий шаг компьютера: извлечение верхнего элемента из стека вызовов (сейчас это `factorial(2)`):



После этого компьютер завершает выполнение функции `factorial(2)`.

Пришло время извлекать из стека следующий элемент. К этому моменту остается только `factorial(3)` — компьютер выталкивает его и завершает выполнение соответствующей функции.

Теперь стек пуст, поэтому компьютер знает, что все рекурсивные вызовы завершены.

Итак, при вычислении факториала числа 3 компьютер выполняет следующие операции.

1. Сначала вызывается функция `factorial(3)`. Прежде чем ее выполнение будет завершено...
2. Вызывается `factorial(2)`. Пока ее выполнение не завершилось...
3. Вызывается `factorial(1)`.
4. Функция `factorial(1)` *завершается* первой.
5. Функция `factorial(2)` завершается с учетом результата `factorial(1)`.
6. `factorial(3)` завершается с учетом результата `factorial(2)`.

Функция факториала — это вычисление, основанное на рекурсии, которое выполняется так: `factorial(1)` передает свой результат (значение 1) функции `factorial(2)`, которая умножает 1 на 2 и передает результат (значение 2) функции `factorial(3)`, которая умножает его на 3, вычисляя итоговый результат — 6.

Некоторые описывают этот процесс как *передачу значения через стек вызовов*. То есть каждая рекурсивная функция возвращает вычисленное значение своей «родительской». В итоге функция, вызванная первой, вычисляет итоговое значение.

Переполнение стека

Вернемся к примеру с бесконечной рекурсией. Как вы думаете, что произойдет со стеком вызовов, если функция `blah()` будет вызывать себя бесконечно?

В случае бесконечной рекурсии компьютер снова и снова помещает значения в стек вызовов, который растет, пока в кратковременной памяти компьютера просто не останется места для хранения данных. В этой ситуации возникает ошибка *переполнения стека* (stack overflow). Компьютер просто прекращает осуществлять рекурсивные вызовы, как бы говоря: «Я отказываюсь снова вызывать функцию, потому что у меня заканчивается память!»

Обход файловой системы

Теперь, когда вы знаете, как работает рекурсия, рассмотрим задачи, которые можно решить только с ее помощью.

Рекурсия — отличный выбор, когда задача предполагает несколько уровней глубины, точное количество которых заранее неизвестно.

Возьмем, к примеру, обход файловой системы. Допустим, у вас есть сценарий, который что-то делает с содержимым каталога, например выводит имена всех подкаталогов. Но вы не хотите, чтобы он учитывал только те из них, которые находятся в текущем каталоге, — вас интересуют все подкаталоги внутри подкаталогов.

Создадим простой сценарий на Ruby, который выводит на экран имена всех подкаталогов в нашем каталоге:

```
def find_directories(directory)
  # Проверяем каждый файл в каталоге. Некоторые из этих "файлов" могут
  # оказаться подкаталогами
  Dir.foreach(directory) do |filename|
```

```
    # Если файл является подкаталогом:
    if File.directory?("#{directory}/#{filename}") &&
    filename != "." && filename != ".."

      # Выводим на экран полный путь к нему:
      puts "#{directory}/#{filename}"
    end
  end
end
```

Эту функцию можно вызвать, передав ей имя каталога. Чтобы вызвать ее для обработки текущего каталога, нужно написать следующее:

```
find_directories(".")
```

С помощью этого сценария мы просматриваем все файлы в заданном каталоге. Если файл будет подкаталогом (и не отражается в виде одинарной или двойной точки, которые означают текущий и родительский каталоги соответственно), мы выводим на экран его имя.

Это рабочий сценарий, но он выводит имена лишь подкаталогов текущего каталога, не затрагивая имена подкаталогов *внутри* них.

Обновим сценарий, чтобы углубить поиск на один уровень:

```
def find_directories(directory)
  # Обрабатываем с помощью цикла содержимое каталога первого уровня:
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
    filename != "." && filename != ".."
      puts "#{directory}/#{filename}"

      # Обрабатываем с помощью цикла содержимое подкаталога второго уровня:
      Dir.foreach("#{directory}/#{filename}") do |inner_filename|
        if File.directory?("#{directory}/#{filename}/#{inner_filename}") &&
        inner_filename != "." && inner_filename != ".."
          puts "#{directory}/#{filename}/#{inner_filename}"
        end
      end

    end
  end
end
```

Теперь при каждом столкновении с каталогом наш сценарий выполняет такой же цикл для обработки подкаталогов *этого* каталога и выводит их имена. Но у этого сценария тоже есть свои ограничения — в процессе поиска он углубляется только на два уровня. А что, если мы хотим проверить три, четыре или пять уровней? Тогда нам придется использовать пять уровней вложенных циклов.

А если мы хотим проверить все возможные подкаталоги? Вы, скорее всего, подумаете, что это невозможно, учитывая то, что нам даже не известно, сколько всего уровней.

Именно *здесь* проявляется настоящая мощь рекурсии. С ее помощью мы можем написать простой сценарий поиска на любой глубине!

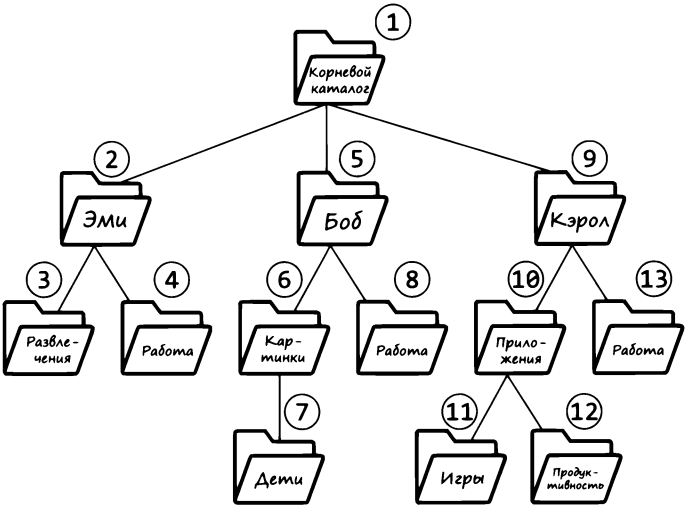
```
def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
    filename != "." && filename != ".."
      puts "#{directory}/#{filename}"

      # Рекурсивно вызываем эту функцию для обработки подкаталога:
      find_directories("#{directory}/#{filename}")
    end
  end
end
```

Когда сценарий находит файлы, которые являются подкаталогами, он вызывает метод `find_directories` для их обработки. Получается, что этот сценарий может искать каталоги на любых уровнях.

На следующей схеме наглядно представлен процесс и порядок обхода подкаталогов в примере с файловой системой.

Мы еще вернемся к этой теме в главе 18.



Выводы

Как вы видели на примере с файловой системой, рекурсия отлично подходит для алгоритмов, выполняющих задачи на любых уровнях.

Теперь вы знаете, как работает рекурсия и насколько полезной она может быть. Вы даже научились читать и анализировать рекурсивный код. Но на начальном этапе многим бывает сложно написать свою рекурсивную функцию. В следующей главе вы узнаете, как писать рекурсивный код, и попутно разберете еще целый ряд задач, требующих рекурсивного подхода.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 10».

1. Следующая функция выводит каждое второе число от младшего `low` до старшего `high`. Например, если `low` равно 0, а `high` — 10, то на экран будут выведены такие значения:

   ```
   0
   2
   4
   6
   8
   10
   ```

   Определите базовый случай для этой функции:

   ```
   def print_every_other(low, high)
     return if low > high
     puts low
     print_every_other(low + 2, high)
   end
   ```

2. Играя с моим компьютером, мой ребенок заменил в функции для вычисления факториала фрагмент `(n-1)` на `(n-2)`. Что произойдет, когда мы запустим следующую функцию `factorial(10)`?

   ```
   def factorial(n)
     return 1 if n == 1
     return n * factorial(n - 2)
   end
   ```

3. Ниже приведена функция, которая получает два числа, старшее `high` и младшее `low`, возвращая сумму всех чисел в диапазоне между ними. Например, если `low` равно 1, а `high` — 10, то функция возвратит сумму всех чисел от 1

до 10 — 55. Но в нашем коде нет базового случая, поэтому он будет выполняться бесконечно! Исправьте код, добавив в него правильный базовый случай:

```
def sum(low, high)
  return high + sum(low, high - 1)
end
```

4. У нас есть массив с числами и другими массивами, в которых тоже есть числа и массивы:

```
array = [ 1,
          2,
          3,
          [4, 5, 6],
          7,
          [8,
            [9, 10, 11,
              [12, 13, 14]
            ]
          ],
          [15, 16, 17, 18, 19,
            [20, 21, 22,
              [23, 24, 25,
                [26, 27, 29]
              ], 30, 31
            ], 32
          ], 33
        ]
```

Напишите рекурсивную функцию, которая выводит все числа (*только* числа).

ГЛАВА 11

Учимся писать рекурсивный код

В предыдущей главе вы узнали, что такое рекурсия и как она работает. Но поначалу мне было сложно писать свои рекурсивные функции, даже несмотря на то, что я понимал принцип их работы.

Постоянно практикуясь и изучая разные рекурсивные паттерны, я обнаружил несколько приемов, благодаря которым становится проще писать рекурсивный код. И я хочу поделиться ими с вами. Попутно мы затронем еще ряд ситуаций, где рекурсия проявляет себя особенно ярко.

Учтите, в этой главе мы не будем говорить об эффективности рекурсии. На самом деле, она может крайне негативно сказаться на временной сложности алгоритма, но об этом — в следующей главе. А здесь мы сосредоточимся на развитии рекурсивного образа мышления.

Категория рекурсивных задач: многократное выполнение действия

Выполняя разные рекурсивные задачи, я научился разделять их на «категории». Это сильно помогло мне: при обнаружении эффективной техники выполнения задачи из определенной категории я мог применить ее для решения других таких же задач.

К простейшим задачам относятся те, цель алгоритма которых — многократное выполнение какого-то действия.

Пример такой задачи — написание функции обратного отсчета перед запуском космического корабля из прошлой главы. Ее код выводит число, например 10,

потом 9, 8 и т. д. вплоть до 0. Хотя функция каждый раз выводит разные числа, ее суть сводится к многократному выполнению одинакового действия — к выводу числа на экран.

Так выглядела реализация этого алгоритма на языке JavaScript:

```
function countdown(number) {
  console.log(number);

  if(number === 0) { // значение number, равное 0, - это базовый случай
    return;
  } else {
    countdown(number - 1);
  }
}
```

Я обнаружил, что в таких задачах последняя строка кода отвечает за повторный вызов функции. В предыдущем фрагменте она выглядит так: `countdown(number - 1)`. Все, что делает эта строка, — осуществляет очередной рекурсивный вызов.

Еще один пример — алгоритм из прошлой главы, многократно выполняющий вывод имен каталогов.

Его код на языке Ruby выглядел так:

```
def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
    filename != "." && filename != ".."
      puts "#{directory}/#{filename}"

      # Рекурсивно вызываем эту функцию для обработки подкаталога:
      find_directories("#{directory}/#{filename}")
    end
  end
end
```

Здесь последняя строка кода `find_directories("#{directory}/#{filename}")` — это тоже простой рекурсивный вызов функции.

Рекурсивный прием: передача дополнительных параметров

Попробуем решить еще одну простейшую задачу — реализовать алгоритм, который принимает массив чисел и удваивает каждое из них. При этом мы не собираемся создавать новый массив: мы будем изменять исходный на месте.

Этот алгоритм тоже предполагает многократное выполнение действия (здесь это удвоение числа). Мы начинаем с удвоения первого числа, затем делаем то же самое со вторым и т. д.

Модификация на месте

На всякий случай стоит пояснить, что подразумевается под модификацией *на месте*.

Есть два основных способа управления данными. Рассмотрим их на примере удвоения чисел в массиве.

Чтобы удвоить элементы массива `[1, 2, 3, 4, 5]` для получения `[2, 4, 6, 8, 10]`, я мог бы сделать одно из двух.

Первый подход: создать новый массив с «удвоенными» значениями и оставить исходный в покое. Рассмотрим следующий код:

```
a = [1, 2, 3, 4, 5]
b = double_array(a)
```

Функция `double_array` создает и возвращает совершенно новый массив, поэтому при проверке значений `a` и `b` мы бы получили:

```
a # [1, 2, 3, 4, 5]
b # [2, 4, 6, 8, 10]
```

Исходный массив `a` не был изменен, а `b` — это совершенно новый массив.

Второй подход — модификация *на месте* — это фактическое изменение *исходного массива*, переданного функции.

При использовании этого подхода проверка значений `a` и `b` приведет к такому результату:

```
a # [2, 4, 6, 8, 10]
b # [2, 4, 6, 8, 10]
```

Здесь функция фактически изменяет массив `a`, а `b` просто указывает на тот же самый массив `a`.

Выбор между созданием нового массива и изменением исходного на месте зависит от контекста проекта. Подробнее об алгоритмах, использующих модификацию на месте, мы поговорим в главе 19.

Попробуем реализовать этот алгоритм, назовем его `double_array()`, на языке Python. Мы знаем, что его последняя строка будет содержать рекурсивный вызов, поэтому добавим ее в код:

```
def double_array(array):
  double_array(array)
```

Теперь нам нужно добавить фрагмент кода, отвечающий за фактическое удвоение числа. Но какое именно число мы будем удваивать? Давайте удвоим первое:

```
def double_array(array):
  array[0] *= 2
  double_array(array)
```

Итак, мы удвоили число с индексом 0, но как нам перейти к удвоению следующего?

Если бы вместо рекурсии мы использовали цикл, то отслеживали бы индекс с помощью переменной, увеличивая ее значение на 1:

```
def double_array(array):
  index = 0

  while (index < len(array)):
    array[index] *= 2
    index += 1
```

Но в рекурсивной версии алгоритма единственный аргумент функции — это массив. Так как же нам отслеживать и инкрементировать индекс?

Рассмотрим еще один прием...

Передачу функции дополнительных параметров!

Изменим начало нашей функции так, чтобы она принимала *два* аргумента: сам массив и индекс для отслеживания:

```
def double_array(array, index):
```

Теперь при вызове функции нам нужно передать массив и начальный индекс, равный 0:

```
double_array([1, 2, 3, 4, 5], 0)
```

Передача индекса функции в качестве аргумента позволяет отслеживать и инкрементировать его значение при каждом последующем рекурсивном вызове. Так это выглядит в виде кода:

```
def double_array(array, index):
  array[index] *= 2
  double_array(array, index + 1)
```

При каждом последующем вызове функции в качестве аргументов мы передаем тот же массив и индекс, увеличенный на единицу. Так мы можем отслеживать значение индекса, как в классическом цикле.

Но наш код все еще неидеален. По достижении конца массива эта функция выдаст ошибку при попытке удвоения несуществующего числа. Чтобы решить эту проблему, нужно предусмотреть базовый случай:

```
def double_array(array, index):
  # Базовый случай: индекс выходит за границу массива
  if index >= len(array):
    return

  array[index] *= 2
  double_array(array, index + 1)
```

Протестируем эту функцию с помощью следующего кода:

```
array = [1, 2, 3, 4]
double_array(array, 0)
print(array)
```

Теперь наша рекурсивная функция завершена. Но если ваш язык программирования поддерживает аргументы по умолчанию, мы можем сделать ее код еще проще.

Сейчас вызов функции выглядит так:

```
double_array([1, 2, 3, 4, 5], 0)
```

Мы признаем, что передача значения 0 в качестве второго параметра выглядит не очень. Это нужно нам для реализации трюка с отслеживанием индекса, которое *всегда* начинается с нуля.

Но использование параметров по умолчанию позволяет обойтись без этого и вызывать функцию так же, как раньше:

```
double_array([1, 2, 3, 4, 5])
```

Так выглядит наш обновленный код:

```
def double_array(array, index=0):
  # Базовый случай: индекс выходит за границу массива
  if (index >= len(array)):
    return

  array[index] *= 2
  double_array(array, index + 1)
```

Все, что мы сделали, — это задали аргумент по умолчанию `index = 0`. Так, при первом вызове функции нам не нужно передавать индекс в качестве параметра и при этом мы можем использовать его для всех последующих вызовов.

Прием с использованием дополнительных параметров функции очень удобен и широко применяется в написании рекурсивных функций.

Категория рекурсивных задач: вычисления

В прошлом разделе мы обсудили первую категорию рекурсивных функций, основанную на многократном выполнении одного действия. В оставшейся части главы мы подробно рассмотрим вторую общую категорию функций, выполняющих вычисления на основе подзадач.

Есть много функций для выполнения вычислений. Например, те, что возвращают сумму двух чисел или находят наибольшее значение в массиве. Они получают некоторые входные данные и возвращают результат вычислений, выполненных на основе этих данных.

В главе 10 мы выяснили, что рекурсия бывает особенно полезна для выполнения задач с произвольным количеством уровней. Вторая область, где рекурсия особенно хорошо себя проявляет, — *вычисления на основе результата выполнения подзадачи основной задачи.*

Прежде чем познакомиться с *подзадачами*, вернемся к примеру с вычислением факториала. Как вы уже знаете, факториал 6 равен:

$$6 \times 5 \times 4 \times 3 \times 2 \times 1.$$

Для написания функции, которая находит факториал числа, мы могли бы использовать классический цикл, который начинается с 1, то есть сначала умножает 1 на 2, затем полученный результат умножает на 3 и т. д., вплоть до 6.

На языке Ruby такая функция может быть реализована так:

```
def factorial(n)
  product = 1

  (1..n).each do |num|
    product *= num
  end

  return product
end
```

Но мы можем применить другой подход: вычислить факториал на основе результатов выполнения *подзадачи.*

Подзадача — это версия основной задачи меньшего размера. Применим этот подход к нашему случаю.

Если подумать, то `factorial(6)` будет равен числу 6, умноженному на результат выполнения функции `factorial(5)`.

Поскольку `factorial(6)` равен:

$$6 \times 5 \times 4 \times 3 \times 2 \times 1,$$

а `factorial(5)`:

$$5 \times 4 \times 3 \times 2 \times 1.$$

Результат выполнения функции `factorial(6)` равнозначен:

$$6 \times \texttt{factorial(5)}.$$

То есть, чтобы получить результат вызова `factorial(6)`, мы можем просто умножить на 6 полученный результат вызова `factorial(5)`.

`factorial(5)` — это менее масштабная задача, результат выполнения которой можно использовать для вычисления результата основной, поэтому она будет *подзадачей* `factorial(6)`.

Вот реализация этого подхода из предыдущей главы:

```
def factorial(number)
  if number == 1
    return 1
  else
    return number * factorial(number - 1)
  end
end
```

Опять же, ключевая строка здесь — `return number * factorial(number - 1)`, в которой мы вычисляем результат, умножая `number` на результат выполнения подзадачи `factorial(number - 1)`.

Два подхода к вычислениям

Итак, при написании функции для вычислений можно использовать два подхода: выполнять задачу «снизу вверх» или атаковать ее «сверху вниз», опираясь на результаты подзадач. Такие *восходящие* и *нисходящие* подходы часто описываются в литературе по информатике в разделах, посвященных рекурсии.

На самом деле оба подхода можно реализовать рекурсивно. Хотя ранее мы рассматривали восходящую реализацию подхода с использованием классического цикла, мы могли бы достичь той же цели и с помощью рекурсии.

Для этого применим вышеописанный трюк с передачей дополнительных параметров:

```
def factorial(n, i=1, product=1)
  return product if i > n
  return factorial(n, i + 1, product * i)
end
```

Здесь мы используем три параметра. Как и прежде, `n` — это число, факториал которого мы вычисляем. `i` — это простая переменная, которая начинается с 1 и увеличивается на единицу при каждом последующем вызове до достижения значения `n`. Наконец, `product` — это переменная, где хранится текущий результат умножения, который мы последовательно передаем функции при каждом вызове, отслеживая результат вычисления по мере выполнения рекурсивных вызовов.

Для такой реализации восходящего подхода мы вполне можем использовать рекурсию, но этот вариант не отличается особой простотой, и у него нет никаких преимуществ по сравнению с использованием классического цикла.

Двигаясь снизу вверх, мы используем для вычислений одну и ту же стратегию. Не важно, применяем мы цикл или рекурсию, сам подход к вычислениям один и тот же.

Но двигаться сверху вниз позволяет *только* рекурсия. И эта уникальная способность реализовать нисходящую стратегию — одна из ее сильнейших сторон.

Нисходящая рекурсия: новый способ мышления

Вот мы и подошли к основной теме этой главы: использование рекурсии при реализации нисходящего подхода. Этот подход *предоставляет нам новую мысленную стратегию выполнения задачи*, позволяя взглянуть на нее под другим углом.

Двигаясь сверху вниз, мы как бы откладываем задачу на потом, освобождая свой разум от мелких подробностей, о которых нам обычно приходится задумываться при реализации восходящего подхода.

Чтобы понять, о чем идет речь, еще раз проанализируем ключевую строку нашей нисходящей реализации функции `factorial`:

```
return number * factorial(number - 1)
```

Она производит вычисление на основе результата `factorial(number - 1)`. Когда мы пишем эту строку, обязательно ли нам понимать, как работает функция `factorial`, которую она вызывает? Нет. При написании кода, который вызывает другую функцию, мы предполагаем, что она возвратит правильное значение, и нам вовсе не обязательно понимать принцип ее работы.

При вычислении ответа на основе результата вызова `factorial` нам не обязательно понимать, как работает эта функция: мы просто ожидаем, что она вернет правильный результат. Но странность в том, что *мы сами и пишем функцию* `factorial`! Вышеупомянутая строка кода находится *внутри этой функции*. Но именно в этом вся прелесть нисходящего подхода: мы можем выполнить задачу, даже не зная, как это делается.

Используя рекурсию для реализации нисходящей стратегии, мы можем немного разгрузить свой мозг и даже проигнорировать некоторые подробности процесса вычисления, просто положившись на то, что результат решения подзадач будет правильным.

Нисходящий способ мышления

Если вы никогда не применяли нисходящую рекурсию, у вас уйдет некоторое время на то, чтобы научиться думать соответствующе. При использовании нисходящего подхода я рекомендую вам сделать следующее.

1. Представьте, что функция, которую вы пишете, уже реализована кем-то другим.
2. Определите подзадачу основной задачи.
3. Посмотрите, что происходит при вызове функции для решения подзадачи, и двигайтесь дальше.

Возможно, сейчас вы не совсем понимаете, что я хочу донести, но все прояснится, когда мы рассмотрим следующие примеры.

Вычисление суммы элементов массива

Допустим, нам нужно написать функцию `sum`, которая суммирует все числа в заданном массиве. Например, при передаче этой функции массива `[1, 2, 3, 4, 5]` она должна возвратить значение 15 — сумму чисел в нем.

Для начала представьте, что `sum` уже реализована. Это может быть нелегко, ведь мы знаем, что только пишем ее! Но давайте хотя бы попытаемся допустить, что функция `sum` уже существует и работает как положено.

Теперь определим подзадачу. Этот процесс — скорее, искусство, чем наука. Он требует практики. В нашем случае подзадача — это суммирование элементов массива `[2, 3, 4, 5]`, всех чисел из исходного массива, кроме первого.

Теперь посмотрим, что произойдет, если мы используем `sum` для выполнения нашей подзадачи. Если эта функция «уже работает как положено», а подзадача сводится к суммированию элементов массива `[2, 3, 4, 5]`, то результатом вызова `sum([2, 3, 4, 5])` будет значение 14 — сумма чисел 2, 3, 4 и 5.

Получается, что для нахождения суммы элементов массива `[1, 2, 3, 4, 5]` мы можем просто прибавить первое число, 1, к результату вызова `sum([2, 3, 4, 5])`.

В псевдокоде мы бы написали что-то вроде этого:

```
return array[0] + sum(the remainder of the array)
```

На языке Ruby мы можем реализовать это так:

```
return array[0] + sum(array[1, array.length - 1])
```

(Во многих языках синтаксис `array[x, y]` возвращает массив элементов с индексами от `x` до `y`.)

Хотите верьте, хотите нет, но мы закончили! Если не считать базового случая, к которому мы перейдем прямо сейчас, наша функция `sum` может быть реализована так:

```
def sum(array)
  return array[0] + sum(array[1, array.length - 1])
end
```

Обратите внимание, что мы не думали о том, как будем складывать все числа массива. Мы просто представили, что кто-то другой уже написал за нас функцию `sum`, и применили ее к подзадаче — отложили выполнение задачи на потом и тем самым выполнили ее.

И последнее: нам нужно обработать базовый случай. Если каждая подзадача будет рекурсивно вызывать свою подзадачу, то в итоге мы придем к подзадаче `sum([5])`. При ее выполнении функция попытается сложить 5 с остальными элементами массива, которых в нем *нет*.

Решить эту проблему можно, предусмотрев базовый случай:

```
def sum(array)
  # Базовый случай: массив содержит только один элемент:
  return array[0] if array.length == 1

  return array[0] + sum(array[1, array.length - 1])
end
```

Вот и все.

Обращение строки

Представьте, что нам нужно написать функцию `reverse`, которая обращает строку, то есть при передаче аргумента `"abcde"` возвращает `"edcba"`.

Сначала определим подзадачу. Опять же, для этого нужна практика, но обычно лучше всего проанализировать ближайшую меньшую версию задачи. Допустим, для строки `"abcde"` подзадачей будет строка `"bcde"`, то есть исходная строка без первого символа.

Теперь представим, что кто-то оказал нам большую услугу, реализовав функцию `reverse`. Как мило с его стороны!

Итак, если функция `reverse` уже есть, а наша подзадача сводится к обращению строки `"bcde"`, то мы можем вызвать `reverse("bcde")`, которая возвратит `"edcb"`.

После этого разобраться с символом `"a"` будет проще простого: достаточно поместить его в конец строки.

Итак, мы можем написать:

```
def reverse(string)
  return reverse(string[1, string.length - 1]) + string[0]
end
```

Наш процесс вычисления сводится к получению результата вызова функции `reverse` для выполнения подзадачи и добавлению первого символа в конец полученной строки.

Опять же, если не считать базового случая, можно сказать, что мы закончили. Я знаю, это похоже на волшебство.

Базовый случай — это строка из одного символа, поэтому для его обработки мы можем добавить в код следующий фрагмент:

```
def reverse(string)
  # Базовый случай: строка из одного символа
  return string[0] if string.length == 1

  return reverse(string[1, string.length - 1]) + string[0]
end
```

Дело сделано.

Подсчет символов «x» в строке

Пока мы в ударе, рассмотрим еще один пример и напишем функцию `count_x`, которая возвращает количество символов `"x"` в заданной строке. Получив стро-

ку "axbxcxd", она должна возвратить значение 3, так как в строке три экземпляра символа "x".

Сначала определим подзадачу. Как и в примере выше, подзадачей будет исходная строка без первого символа. Итак, подзадача для строки "axbxcxd" — "xbxcxd"».

Теперь представим, что функция count_x уже реализована. Если мы вызовем count_x для выполнения подзадачи с помощью кода count_x("xbxcxd"), то получим 3. К этому результату нам достаточно прибавить 1, если первый символ исходной строки тоже "x" (если *нет*, прибавлять ничего не нужно).

Итак, мы можем написать:

```
def count_x(string)
  if string[0] == "x"
    return 1 + count_x(string[1, string.length - 1])
  else
    return count_x(string[1, string.length - 1])
  end
end
```

Это довольно простое условное утверждение. Если первый символ строки — "x", мы прибавляем 1 к результату подзадачи, если нет — возвращаем результат этой подзадачи как есть.

Мы почти закончили. Осталось обработать базовый случай.

Можно сказать, что базовый случай в этом примере — строка из одного символа. Но здесь мы сталкиваемся с проблемой: у нас получается два базовых случая, так как этим одиночным символом может быть "x", а может и не быть:

```
def count_x(string)

  # Базовый случай:
  if string.length == 1
    if string[0] == "x"
      return 1
    else
      return 0
    end
  end

  if string[0] == "x"
    return 1 + count_x(string[1, string.length - 1])
  else
    return count_x(string[1, string.length - 1])
  end
end
```

К счастью, для решения этой проблемы мы можем применить еще один простой прием.

Во многих языках вызов `string[1, 0]` возвращает пустую строку. Благодаря этому мы можем упростить наш код:

```
def count_x(string)

  # Базовый случай: пустая строка
  return 0 if string.length == 0
  if string[0] == "x"
    return 1 + count_x(string[1, string.length - 1])
  else
    return count_x(string[1, string.length - 1])
  end
end
```

Базовый случай в этой версии — пустая строка (`string.length == 0`). Мы возвращаем 0, потому что в пустой строке никогда не будет `"x"`.

Когда в строке всего один символ, функция прибавляет 1 или 0 к результату следующего вызова. Этот следующий вызов — `count_x(string[1, 0])`, так как значение `string.length - 1` равно 0. Код `string[1, 0]` возвращает пустую строку, значит, этот последний вызов и есть базовый случай, в результате которого возвращается значение 0.

Вот теперь мы закончили.

Для справки: вызов `array[1, 0]` также возвращает пустой массив во многих языках программирования, поэтому вышеописанный трюк мы могли бы использовать и в предыдущих двух примерах.

Задача с лестницей

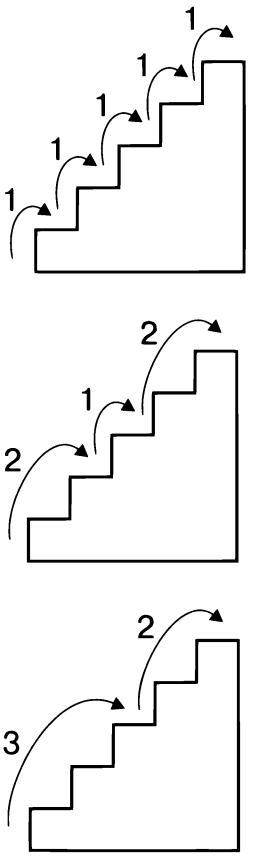
Освоив новую мысленную стратегию для выполнения некоторых вычислительных задач с помощью нисходящей рекурсии, вы все еще можете задаваться вопросом, зачем она вообще нужна. В конце концов, до сих пор вам удавалось выполнять подобные задачи с помощью циклов.

По сути, при выполнении относительно простых вычислений эта новая стратегия может и не понадобиться. Но когда дело доходит до более сложных функций, рекурсивный подход значительно упрощает написание кода. По крайней мере, для меня это так!

Приведу один из моих любимых примеров — задачу с лестницей.

Допустим, у нас есть лестница из *N* ступенек, а человек может преодолеть одну, две или три ступеньки за раз. Сколько есть возможных «способов» подъема по такой лестнице? Напишите функцию, которая вычисляет число таких способов для *N* ступенек.

На следующем изображении показаны три возможных способа подъема по пятиступенчатой лестнице.



Это всего лишь три варианта из множества.

Используем для выполнения этой задачи восходящий подход. Мы будем двигаться от простых случаев к более сложным.

Очевидно, что при наличии всего одной ступеньки возможный способ подъема только один.

На двухступенчатую лестницу можно подняться двумя способами. При этом человек может дважды подняться на одну ступеньку или преодолеть сразу две. Запишем это так:

```
1, 1
2
```

Для подъема по трехступенчатой лестнице есть четыре возможных способа:

```
1, 1, 1
1, 2
2, 1
3
```

При наличии четырех ступенек число способов подъема достигает семи:

```
1, 1, 1, 1
1, 1, 2
1, 2, 1
1, 3
2, 1, 1
2, 2
3, 1
```

Попробуйте составить остальные комбинации для пятиступенчатой лестницы. Будет непросто! А это всего пять ступенек. Представьте, сколько комбинаций будет для 11.

Теперь перейдем к главному вопросу: как *написать код* для подсчета всех возможных способов подъема?

Без рекурсивного мышления понять алгоритм выполнения этого вычисления очень трудно. Но при использовании соответствующего нисходящего подхода эта задача может оказаться на удивление легкой.

Допустим, первая подзадача для 11-ступенчатой лестницы — это 10-ступенчатая лестница. Как думаете, если бы мы знали число всех способов подъема по 10-ступенчатой лестнице, могли бы мы использовать его при подсчете возможных способов подъема по 11-ступенчатой?

Во-первых, мы точно знаем, что для подъема по 11-ступенчатой лестнице нужно *не меньше* шагов, чем для подъема по 10-ступенчатой. То есть у нас есть все способы достижения десятой ступеньки, с которой мы можем подняться еще на одну, чтобы добраться до вершины.

Но это будет неполное решение, так как мы знаем, что человек может добраться до вершины со ступенек 9 и 8.

Если задуматься, становится ясно, что при выборе любого способа подъема с 10-й ступеньки на 11-ю мы не учитываем ни один из способов, предполагающих прыжок с 9-й на 11-ю. И наоборот, если мы прыгаем с 9-й ступеньки на 11-ю, то исключаем все способы, предполагающие попадание на 10-ю.

Итак, мы точно знаем, что количество способов подъема будет включать как минимум число путей до 10-й и до 9-й ступенек.

А поскольку человек может перепрыгнуть и с 8-й ступеньки на 11-ю, то есть преодолеть три ступеньки за раз, мы должны учесть соответствующее число способов подъема.

Итак, мы выяснили, что число способов подъема на вершину равно, по крайней мере, сумме всех способов достижения ступенек 10, 9 и 8.

Но если подумать, становится очевидно, что других вариантов подъема, кроме этих, не существует: ведь человек не может перепрыгнуть с 7-й ступеньки на 11-ю. Поэтому количество способов подъема для *N* ступенек равно:

```
number_of_paths(n - 1) + number_of_paths(n - 2) + number_of_paths(n - 3)
```

Если не считать базового случая, можно сказать, что код функции уже готов!

```
def number_of_paths(n)
  number_of_paths(n - 1) + number_of_paths(n - 2) + number_of_paths(n - 3)
end
```

Это кажется невероятным, но перед нами почти весь код, который нам нужен. Осталось разобраться с базовым случаем.

Базовый случай для задачи с лестницей

Определить базовый случай для этой задачи проблематично, так как при достижении `n`, равного 3, 2 или 1, эта функция продолжает вызывать саму себя с помощью нулевого или отрицательного значения `n`. Например, `number_of_paths(2)` вызывает `number_of_paths(1)`, `number_of_paths(0)` и `number_of_paths(-1)`.

Один из способов решения этой проблемы — «жестко запрограммировать» все базовые случаи:

```
def number_of_paths(n)
  return 0 if n <= 0
  return 1 if n == 1
  return 2 if n == 2
  return 4 if n == 3
  return number_of_paths(n - 1) + number_of_paths(n - 2) +
    number_of_paths(n - 3)
end
```

Еще один способ обработки базовых случаев — использование довольно странного, но эффективного приема, который просто обеспечивает вычисление правильных чисел. Сейчас объясню.

Мы знаем, что результат `number_of_paths(1)` должен быть равен 1, поэтому начнем с базового случая:

```
return 1 if n == 1
```

Еще мы знаем, что вызов `number_of_paths(2)` должен возвращать значение 2, но нам *не нужно* прописывать этот базовый случай явно. Вместо этого мы можем воспользоваться тем фактом, что результат `number_of_paths(2)` будет вычисляться как `number_of_paths(1)` + `number_of_paths(0)` + `number_of_paths(-1)`. При вызове `number_of_paths(1)` функция возвращает 1, поэтому если мы сделаем так, чтобы вызов `number_of_paths(0)` тоже возвращал 1, а `number_of_paths(-1)` — 0, то в итоге получим нужную нам сумму — 2.

Итак, мы можем добавить следующие базовые случаи:

```
return 0 if n < 0
return 1 if n == 1 || n == 0
```

Теперь перейдем к вызову `number_of_paths(3)`, который должен возвратить сумму `number_of_paths(2)` + `number_of_paths(1)` + `number_of_paths(0)`. Мы знаем, что результат этого сложения — значение 4, поэтому давайте проверим, что у нас получается. Благодаря проделанной выше настройке базовых случаев вызов `number_of_paths(2)` возвращает 2, а `number_of_paths(1)` и `number_of_paths(0)` — 1, так что в итоге мы получаем 4. Это нам и нужно.

Полный код нашей функции может выглядеть и так:

```
def number_of_paths(n)
    return 0 if n < 0
    return 1 if n == 1 || n == 0
    return number_of_paths(n - 1) + number_of_paths(n - 2) +
        number_of_paths(n - 3)
end
```

Хотя эта реализация не такая понятная, как предыдущая, мы обрабатываем все базовые случаи с помощью всего двух строк кода.

Как видите, нисходящий рекурсивный подход сильно упрощает выполнение этой задачи.

Генерация анаграмм

В завершение нашего обсуждения выполним самую сложную рекурсивную задачу из тех, с которыми мы сталкивались. Для этого нам придется использовать все рекурсивные инструменты в своем арсенале.

Нам нужно написать функцию, которая возвращает массив всех анаграмм для заданной строки. Анаграмма — это результат переупорядочения всех символов в строке. Например, анаграммы строки `"abc"` следующие:

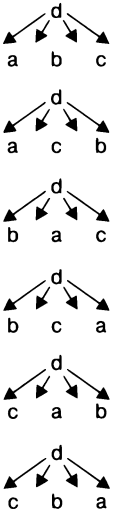
```
["abc",
"acb",
"bac",
"bca",
"cab",
"cba"]
```

Допустим, нам нужно составить все анаграммы для строки `"abcd"`. Для выполнения этой задачи применим нисходящий подход.

Предположим, что подзадача `"abcd"` — это `"abc"`. Тогда возникает вопрос: если бы у нас была работающая функция `anagrams`, которая возвращает все анаграммы строки `"abc"`, как их использовать для составления всех анаграмм строки `"abcd"`? Подумайте о возможных подходах.

Мне в голову пришел следующий (хотя есть и другие).

При наличии всех шести анаграмм строки `"abc"` мы могли бы получить все возможные перестановки строки `"abcd"`, помещая символ `"d"` во все возможные места внутри каждой анаграммы `"abc"`:



Вот реализация этого алгоритма на языке Ruby. Как видите, она гораздо сложнее всех предыдущих из этой главы:

```
def anagrams_of(string)
  # Базовый случай: если строка состоит из одного символа,
  # возвращаем массив, который содержит только эту односимвольную строку:
  return [string[0]] if string.length == 1

  # Создаем массив для хранения всех анаграмм:
  collection = []

  # Находим все анаграммы подстроки, от второго символа и до конца.
  # Например, подстрока для строки "abcd" - это "bcd",
  # поэтому мы находим все анаграммы "bcd":
  substring_anagrams = anagrams_of(string[1, string.length - 1])

  # Перебираем все подстроки
  substring_anagrams.each do |substring_anagram|

    # Перебираем все индексы подстроки от 0 до
    # первого индекса, выходящего за пределы строки:
    (0..substring_anagram.length).each do |index|

      # Создаем копию анаграммы подстроки:
      copy = String.new(substring_anagram)

      # Вставляем первый символ строки в
      # копию анаграммы подстроки. Место его вставки зависит
      # от индекса, который сейчас обрабатывается в цикле.
      # Теперь берем эту новую строку и добавляем в нашу коллекцию анаграмм:
      collection << copy.insert(index, string[0])
    end
  end

  # Возвращаем всю коллекцию анаграмм:
  return collection
end
```

Это непростой код, поэтому давайте подробно его разберем. Пока забудем о базовом случае.

Мы начинаем с создания пустого массива, где будем собирать всю коллекцию анаграмм:

```
collection = []
```

Именно его наша функция должна будет вернуть в конце.

Затем получаем массив всех анаграмм для подстроки исходной строки. Эта подстрока состоит из тех же символов, кроме первого, и представляет собой подзадачу. Например, подстрокой для строки `"hello"` будет `"ello"`:

```
substring_anagrams = anagrams_of(string[1, string.length - 1])
```

Обратите внимание, что мы используем нисходящий способ мышления, предполагая, что функция `anagrams_of` уже существует и обрабатывает подстроку как положено.

Затем мы перебираем все анаграммы подстроки:

```
substring_anagrams.each do |substring_anagram|
```

Прежде чем продолжить, отмечу, что мы используем здесь комбинацию из циклов и рекурсии. Использование рекурсии не означает *полное* исключение циклов из своего кода! Чтобы выполнить задачу, мы можем использовать любой инструмент.

При обработке каждой анаграммы подстроки мы перебираем все ее индексы, создаем ее копию и вставляем первый символ нашей строки (то есть единственный символ, которого нет в подстроке) по текущему индексу. Так мы создаем новую анаграмму, которую затем добавляем в коллекцию:

```
(0..substring_anagram.length).each do |index|
  copy = String.new(substring_anagram)
  collection << copy.insert(index, string[0])
end
```

В самом конце мы возвращаем коллекцию анаграмм `collection`.

Базовый случай здесь — это подстрока только с одним символом, и в этом случае сам символ представляет собой единственную анаграмму!

Эффективность алгоритма генерации анаграмм

Теперь остановимся на минутку и проанализируем эффективность нашего алгоритма. Мы обнаружим кое-что интересное, а именно — категорию алгоритмической сложности, с которой мы еще не сталкивались.

Если подумать о количестве анаграмм, создаваемых нашим алгоритмом, то можно заметить интересную закономерность.

Для строки из трех символов мы создаем перестановки, которые начинаются с каждого из этих трех символов. Их средний символ выбирается из двух оставшихся, а последним служит тот, который не был задействован вначале. Итого $3 \times 2 \times 1$ — шесть перестановок.

Для строк другой длины мы получаем:

4 символа: $4 \times 3 \times 2 \times 1$ анаграмм,

5 символов: $5 \times 4 \times 3 \times 2 \times 1$ анаграмм,

6 символов: $6 \times 5 \times 4 \times 3 \times 2 \times 1$ анаграмм.

Узнаете закономерность? Это факториал!

То есть если строка состоит из шести символов, количество анаграмм равно факториалу числа 6: $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$.

В математике факториал обозначается восклицательным знаком, поэтому факториал числа 6 выражается в виде 6!, а числа 10 — 10!.

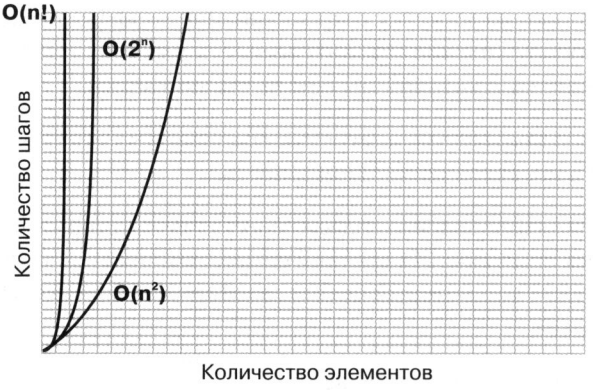
Как вы помните, О-нотация помогает определить, сколько шагов будет выполнять алгоритм при наличии N элементов данных. В нашем случае N — это длина строки.

Поэтому для строки длины N мы можем составить $N!$ анаграмм. С точки зрения О-нотации это выражается как $O(N!)$, а соответствующие алгоритмы обладают *факториальной временной сложностью*, то есть выполняются за *факториальное время*.

К категории $O(N!)$ относятся самые медленные алгоритмы из описанных в этой книге. На следующем графике показана их эффективность в сравнении с другими «медленными» алгоритмами.

Хотя алгоритм $O(N!)$ очень неторопливый, здесь лучшей альтернативы у нас нет, так как наша задача — составить *все* анаграммы, а для N-символьного слова их число равно $N!$.

В любом случае рекурсия сыграла ключевую роль в создании этого алгоритма, что служит наглядным примером использования рекурсивного подхода для решения сложной задачи.



Выводы

Чтобы освоить навык написания функций с использованием рекурсии, нужно практиковаться. Но теперь вы знакомы с приемами, позволяющими облегчить этот процесс.

Тем не менее, с рекурсией мы еще не закончили. Несмотря на то что это отличный инструмент для выполнения широкого спектра задач, ее использование может *сильно* замедлить работу вашего кода, если вы не будете осторожны. В следующей главе я расскажу, как применять рекурсию, сохраняя красоту и скорость выполнения кода.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 11».

1. С помощью рекурсии напишите функцию, которая принимает массив строк и возвращает общее количество символов во всех строках. Например, если входной массив — `["ab", "c", "def", "ghij"]`, то эта функция должна вернуть значение 10, так как в массиве всего 10 символов.

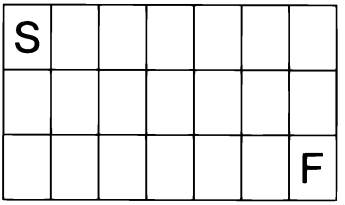
2. С помощью рекурсии напишите функцию, которая принимает массив чисел и возвращает новый, в котором будут только четные.

3. Есть последовательность так называемых треугольных чисел, которая начинается как 1, 3, 6, 10, 15, 21 и продолжается с *N*-го числа в шаблоне, равного *N* плюс предыдущее число. Например, седьмое число последовательности — 28, то есть 7 (номер числа *N*) плюс 21 (предыдущее число последовательности). Напишите функцию, которая принимает номер числа *N* и возвращает соответствующее число последовательности. Например, при передаче этой функции значения 7 она должна вернуть 28.

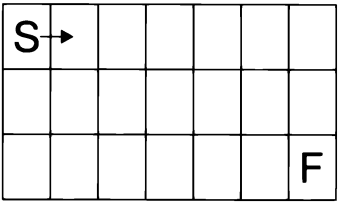
4. С помощью рекурсии напишите функцию, которая принимает строку и возвращает первый попавшийся индекс, соответствующий символу `"x"`. Например, строка `"abcdefghijklmnopqrstuvwxyz"` содержит `"x"` в позиции с индексом 23. Чтобы было проще, допустим, что в передаваемой функции строке есть как минимум один символ `"x"`.

5. Эта проблема известна как проблема «уникальных путей»: допустим, у вас есть сетка строк и столбцов. Напишите функцию, которая принимает число строк и столбцов и вычисляет количество возможных «кратчайших» путей от верхнего левого квадрата до нижнего правого.

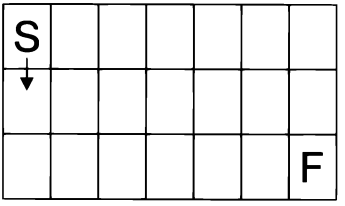
Например, так выглядит сетка с тремя строками и семью столбцами. Нам нужно перейти из квадрата «S» (Старт) в квадрат «F» (Финиш)



Под «кратчайшим» путем подразумевается то, что каждый раз вы передвигаетесь на один шаг вправо:



или на один шаг вниз:



Ваша функция должна вычислять *количество* кратчайших путей.

ГЛАВА 12

Динамическое программирование

В прошлой главе вы научились писать рекурсивный код и использовать рекурсию для решения некоторых сложных задач.

Но, несмотря на то что рекурсия может решить *некоторые* проблемы, при неправильном использовании она способна и породить *новые*. На самом деле именно рекурсия часто становится виновницей медленной работы некоторых алгоритмов — например, относящихся к категории сложности $O(2^N)$.

Но хорошая новость в том, что многих проблем вполне можно избежать. В этой главе вы узнаете, как выявить самые распространенные ловушки, замедляющие скорость работы рекурсивного кода, и научитесь описывать соответствующие алгоритмы с помощью О-нотации. Что еще важнее — вы узнаете, как решать все эти проблемы.

Еще одна хорошая новость: приемы из этой главы очень простые, но эффективные. И здесь мы узнаем, как с их помощью превратить рекурсивный кошмар в рекурсивное блаженство.

Бесполезные рекурсивные вызовы

Ниже представлена рекурсивная функция, которая находит наибольшее число в массиве:

```
def max(array)

  # Базовый случай: если массив содержит всего один элемент,
  # то этот элемент по определению будет наибольшим числом:

  return array[0] if array.length == 1
```

```
  # Сравниваем первый элемент с наибольшим числом из оставшихся в массиве.
  # Если он больше, возвращаем его в качестве наибольшего числа:
  if array[0] > max(array[1, array.length - 1])
    return array[0]

  # Если нет, возвращаем наибольшее число из оставшихся в массиве:
  else
    return max(array[1, array.length - 1])
  end
end
```

Каждый рекурсивный вызов сравнивает одно число (`array[0]`) с максимальным числом из оставшихся в массиве (чтобы найти максимальное число среди оставшихся, мы вызываем ту самую функцию `max`, внутри которой находимся. Именно это и делает ее рекурсивной).

Выполняем сравнение с помощью условного оператора, первая половина которого выглядит так:

```
if array[0] > max(array[1, array.length - 1])
  return array[0]
```

Исходя из этого фрагмента кода становится ясно, что если одиночное значение (`array[0]`) превышает текущее максимальное число из оставшихся в массиве (`max(array[1, array.length - 1])`), `array[0]` по определению будет наибольшим числом. Поэтому мы возвращаем именно его.

Вторая половина условного оператора выглядит так:

```
else
  return max(array[1, array.length - 1])
```

Глядя на второй фрагмент, мы понимаем, что если `array[0]` *не* превышает текущее максимальное число из оставшихся в массиве, то оно будет наибольшим во всем массиве, и мы возвращаем его.

Это рабочий код, но у него есть скрытая неэффективность. Если вы посмотрите внимательно, то заметите, что фрагмент `max(array[1, array.length - 1])` встречается в коде дважды — по одному разу в каждой половине условного оператора.

Проблема в том, что каждое упоминание `max(array[1, array.length - 1])` запускает лавину рекурсивных вызовов.

Разберем этот вопрос на примере массива `[1, 2, 3, 4]`.

Мы знаем, что сначала функция сравнит 1 с максимальным числом из оставшихся в массиве `[2, 3, 4]`. Это приведет к сравнению числа 2 с максимальным значением из оставшихся `[3, 4]`, а затем и к сравнению 3 с `[4]`. Еще один рекурсивный вызов будет выполнен для значения `[4]`, которое служит базовым случаем.

Но чтобы действительно увидеть, как работает наш код, мы начнем с анализа «нижнего» вызова и пройдем вверх по всей их цепочке.

Приступим.

Пошаговый разбор выполнения рекурсивной функции max

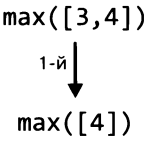
При вызове `max([4])` функция просто возвращает число 4, потому что массив с одним элементом — это базовый случай, как видно в следующей строке кода:

```
return array[0] if array.length == 1
```

Здесь все понятно — это просто единичный вызов функции:

```
max([4])
```

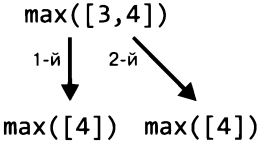
Пройдем вверх по цепочке вызовов и посмотрим, что произойдет при вызове `max([3, 4])`. В первой половине условного оператора (`if array[0] > max(array[1, array.length - 1])`) мы сравниваем 3 с `max([4])`. Но вызов `max([4])` сам по себе рекурсивен. На следующей схеме показано, как вызов функции `max([4])` выполняется в рамках вызова `max([3, 4])`:



Обратите внимание на метку «1-й» рядом со стрелкой, указывающей на то, что этот рекурсивный вызов был выполнен *первой* половиной условного оператора в рамках вызова `max([3, 4])`.

После выполнения этого шага код может сравнить 3 с результатом вызова `max([4])`, которым будет 4. Поскольку 3 меньше 4, выполняется вторая половина условного оператора (`return max(array[1, array.length - 1])`). В результате код возвращает `max([4])`.

Но это приводит к фактическому запуску второго вызова функции `max([4])`:

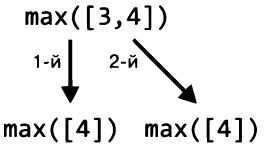


Как видите, `max([3, 4])` предусматривает два вызова `max([4])`. Конечно, мы хотели бы этого избежать. Если мы уже вычислили результат `max([4])`, зачем нам снова вызывать ту же функцию?

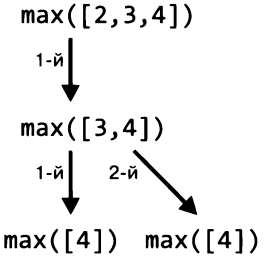
Но когда мы перемещаемся лишь на один уровень вверх по цепочке вызовов, ситуация ухудшается.

Вот что происходит при вызове `max([2, 3, 4])`.

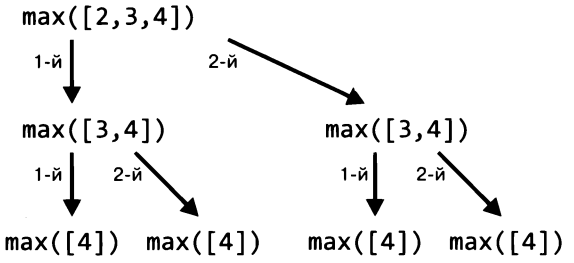
Первая половина условного оператора сравнивает число 2 с `max([3, 4])`. Процесс, как мы уже выяснили, выглядит так:



Получается, что вызов `max([3, 4])` в рамках вызова `max([2, 3, 4])` будет выглядеть так:

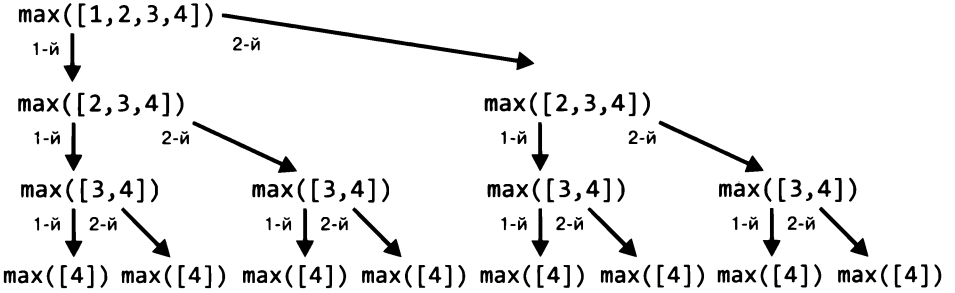


Но подвох в том, что мы разобрали только *первую* половину условного оператора, обрабатывающего вызов `max([2, 3, 4])`. Во второй мы *снова* вызываем `max([3, 4])`:



Ой!

Если мы осмелимся подняться на самую вершину цепочки и вызовем `max([1, 2, 3, 4])`, то после вызова функции `max` в обеих половинах условного оператора мы получим вот это:



Итак, при вызове `max([1, 2, 3, 4])` функция `max` запускается целых 15 раз.

Мы можем убедиться в этом, добавив в начало функции фрагмент кода `puts "RECURSION"`:

```
def max(array)
  puts "RECURSION"

  # оставшаяся часть кода опущена для краткости
```

Теперь при запуске кода в консоли будет 15 раз отображаться слово RECURSION.

Конечно, *некоторые* из этих вызовов очень важны. Например, нам нужно вычислить результат `max([4])`. Для этого вполне достаточно одного вызова такой функции, но в примере выше мы вызываем ее *восемь* раз.

Маленькое исправление для большого «О»

К счастью, есть простой способ избавиться от лишних рекурсивных вызовов: вызов функции `max` только один раз и *сохранение* полученного результата в переменной:

```
def max(array)

  return array[0] if array.length == 1

  # Вычисляем максимальное значение среди оставшихся в массиве
  # и сохраняем его в переменной:
```

```
    max_of_remainder = max(array[1, array.length - 1])

    # Сравниваем первое число со значением этой переменной:

    if array[0] > max_of_remainder
      return array[0]
    else
      return max_of_remainder
    end
  end
```

Эта простая модификация позволяет нам сократить количество вызовов функции `max` до четырех. Убедитесь в этом сами, добавив в код строку `puts "RECURSION"` и запустив его.

Трюк в том, что мы выполняем все необходимые вызовы функции только один раз и сохраняем их результат в переменной, что позволяет исключить повторы.

Разница в эффективности между исходной функцией и ее слегка измененной версией весьма существенная.

Эффективность рекурсии

Во второй, улучшенной версии функции `max` число рекурсивных вызовов соответствует количеству значений в массиве. Поэтому мы можем отнести ее к категории O(N).

Алгоритмы O(N), с которыми мы работали до сих пор, предполагали использование циклов, выполняемых N раз. Но мы можем оценить с помощью O-нотации и рекурсивные функции.

Как вы помните, O-нотация помогает узнать, сколько шагов будет выполнять алгоритм при наличии N элементов данных.

Поскольку улучшенная функция `max` выполняется N раз при N значений в массиве, ее временная сложность равна O(N). Даже если сама функция предусматривает несколько шагов, например пять, ее временная сложность будет равна O(5N), что опять же сводится к O(N).

Но в первой версии функция вызывала саму себя *дважды* при каждом запуске (кроме базового случая). Давайте посмотрим, как это работает для разных размеров массивов.

В следующей таблице показано, сколько раз вызывается функция `max` для массивов с разным количеством элементов:

N элементов	Количество вызовов
1	1
2	3
3	7
4	15
5	31

Видите закономерность? Увеличение числа элементов массива на единицу приводит почти к *удвоению* числа шагов алгоритма. Как было сказано в главе 7, такие алгоритмы относятся к категории $O(2^N)$ и работают очень медленно.

Но в улучшенной версии число рекурсивных вызовов функции `max` соответствует количеству элементов в массиве. Значит, она относится к категории $O(N)$.

Исходя из этого, можно сделать важный вывод: предотвращение ненужных рекурсивных вызовов — ключ к поддержанию высокой скорости работы рекурсивной функции. То, что казалось совсем незначительным изменением кода, простое сохранение результатов вычислений в переменной, в итоге позволило перевести функцию из категории $O(2^N)$ в $O(N)$.

Перекрывающиеся подзадачи

Последовательность Фибоначчи — это математическая числовая последовательность, которая начинается так:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

и продолжается до бесконечности.

Первые числа Фибоначчи — 0 и 1, а каждое последующее — сумма двух предыдущих. Например, 55 — это сумма двух предыдущих чисел — 21 и 34.

Следующая функция на языке Python возвращает *N*-е число последовательности Фибоначчи. Например, если мы передадим ей число 10, она возвратит 55, так как это десятый элемент этой последовательности (0 считается нулевым элементом).

```
def fib(n):
  # Первые два числа последовательности - базовые случаи:
  if n == 0 or n == 1:
    return n

  # Возвращаем сумму двух предыдущих чисел Фибоначчи:
return fib(n - 2) + fib(n - 1)
```

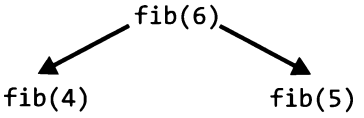
Ключевая строка этой функции:

```
return fib(n - 2) + fib(n - 1)
```

суммирует два предыдущих числа Фибоначчи. Это прекрасная рекурсивная функция.

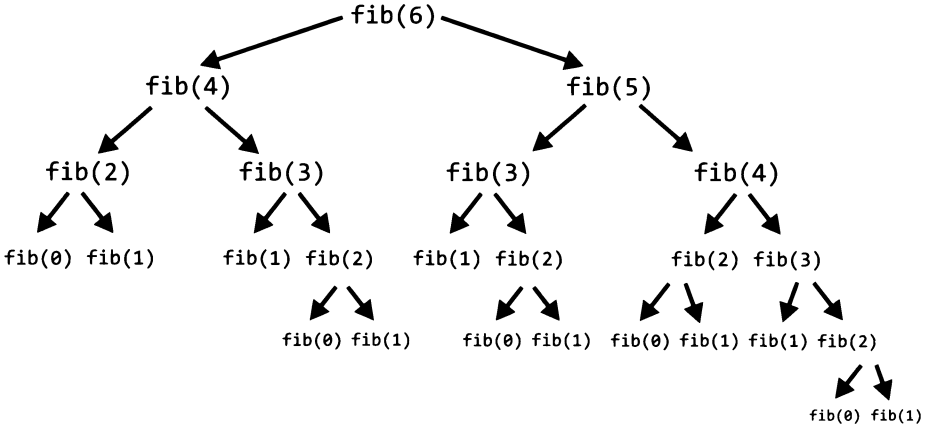
Но прямо сейчас вы должны насторожиться, ведь функция вызывает сама себя *дважды*.

Для примера рассмотрим вычисление шестого элемента последовательности Фибоначчи. Как видно на следующей схеме, функция `fib(6)` вызывает как `fib(4)`, так и `fib(5)`.



Мы уже знаем, что функция, которая дважды вызывает саму себя, относится к категории сложности $O(2^N)$.

Вот все рекурсивные вызовы, которые выполняются в рамках вызова `fib(6)`:



Согласитесь, что работа алгоритмов категории $O(2^N)$ выглядит довольно устрашающе.

Но оптимизировать функцию для вычисления чисел Фибоначчи не так легко, как это было в первом примере, где мы просто внесли в код одно простое изменение.

Мы не можем сохранить в переменной только один фрагмент данных. Нам *нужен* результат вычисления как `fib(n - 2)`, так и `fib(n - 1)` (так как каждое число Фибоначчи — сумма этих двух чисел), и, сохранив только один из них, мы не получим второй.

Для описания этого случая программисты используют термин *перекрывающиеся подзадачи*. Что он значит? Давайте разберемся.

Когда основная задача выполняется путем реализации ее уменьшенных версий, эти версии называются *подзадачами*. Эта концепция не должна быть для вас в новинку — мы часто сталкивались с ней при обсуждении рекурсии. В случае с последовательностью Фибоначчи мы определяем каждое число на основе предыдущих элементов последовательности, вычисление которых и будет подзадачей.

Перекрывающимися эти подзадачи делает то, что вызовы функций `fib(n - 2)` и `fib(n - 1)` приводят к многократному выполнению одних и тех же вычислений. То есть в рамках вызова `fib(n - 1)` производятся некоторые из действий, которые уже были выполнены в рамках вызова `fib(n - 2)`. Например, как видно на прошлой диаграмме, и `fib(4)`, и `fib(5)` вызывают функцию `fib(3)` (и осуществляют многие другие повторяющиеся вызовы).

Может показаться, что мы в тупике: наша функция для нахождения чисел Фибоначчи предполагает выполнение множества перекрывающихся вызовов, из-за чего алгоритм работает с черепашьей скоростью $O(2^N)$. И мы ничего не можем с этим поделать.

Или можем?

Динамическое программирование с помощью мемоизации

К счастью, нам на помощь приходит *динамическое программирование* (*ДП*) — процесс оптимизации рекурсивных функций, предполагающих выполнение перекрывающихся подзадач.

Не обращайте внимания на слово «динамическое». Никто точно не знает, как появился этот термин, и в методах, которые я вам покажу, тоже нет ничего особенно динамического.

Оптимизация алгоритма с помощью ДП может выполняться двумя способами.

Первый: *мемоизация*. Нет, это не опечатка. Мемоизация — это простая, но очень эффективная техника сокращения рекурсивных вызовов для решения перекрывающихся подзадач.

По сути, она снижает число рекурсивных вызовов за счет *запоминания* ранее вычисленных результатов (этим она действительно похожа на меморизацию).

В нашем примере с числами Фибоначчи при первом вызове `fib(3)` функция выполняет вычисления и возвращает 2. Но, прежде чем двигаться дальше, она сохраняет этот результат в хеш-таблице:

```
{3: 2}
```

Эта запись означает, что результат вызова функции `fib(3)` — значение 2.

Точно так же наш код будет запоминать результаты всех вновь выполненных вычислений. Например, после вызова функций `fib(4)`, `fib(5)` и `fib(6)` наша хеш-таблица будет выглядеть так:

```
{
  3: 2,
  4: 3,
  5: 5,
  6: 8
}
```

Благодаря этой хеш-таблице мы можем предотвратить будущие рекурсивные вызовы. Вот как это работает.

Без мемоизации функция `fib(4)` вызывает `fib(3)` и `fib(2)`, которые выполняют уже свои рекурсивные вызовы. При наличии хеш-таблицы мы можем подойти к делу иначе. Теперь вместо вызова `fib(3)` функция `fib(4)` сначала проверяет хеш-таблицу, чтобы узнать, есть ли уже в ней результат вычисления `fib(3)`. Функция вызывает `fib(3)`, только если в хеш-таблице *нет* ключа `3`.

Мемоизация позволяет решить главную проблему, связанную с перекрывающимися подзадачами: многократное выполнение одинаковых рекурсивных вызовов. Используя мемоизацию, мы сохраняем результаты каждого нового вычисления в хеш-таблице, чтобы использовать их в будущем. Так, мы выполняем то или иное вычисление, только если не выполняли его раньше.

Все это звучит хорошо, но есть один вопрос: «Как рекурсивные функции получают доступ к этой хеш-таблице?»

Ответ таков: мы передаем эту хеш-таблицу функции в качестве второго параметра.

Так как хеш-таблица — это особый объект в памяти, мы можем передавать ее от одного рекурсивного вызова к другому, даже если по ходу дела модифицируем ее. Это подходит даже для раскручивания стека вызовов. Несмотря на то что во время первого вызова хеш-таблица могла быть пустой, она вполне может заполниться данными к моменту завершения выполнения исходного вызова.

Реализация мемоизации

Для передачи хеш-таблицы мы изменяем функцию так, чтобы она принимала *два* аргумента, вторым из которых и будет наша хеш-таблица с именем `memo`:

```
def fib(n, memo):
```

При первом вызове функции мы передаем ей число и пустую хеш-таблицу:

```
fib(6, {})
```

Таблица передается дальше при каждом рекурсивном вызове функции `fib`, постепенно заполняясь данными.

Остальная часть функции выглядит так:

```
def fib(n, memo):

  if n == 0 or n == 1:
    return n

  # Проверяем хеш-таблицу (memo) на предмет наличия
  # результата вычисления fib(n):
  if not memo.get(n):

    # Если n нет в memo, вычисляем результат вызова fib(n) с помощью рекурсии
    # и сохраняем его в хеш-таблице:
    memo[n] = fib(n - 2, memo) + fib(n - 1, memo)

  # Сейчас результат вызова fib(n) уже точно находится в memo.
  # (Возможно, он был там и раньше, а может быть, мы сохранили его
  # в хеш-таблице при выполнении прошлой строки кода. Сейчас он там
  # точно есть.) Поэтому возвращаем его:
  return memo[n]
```

Проанализируем этот код построчно.

Итак, теперь наша функция принимает два параметра: `n` и хеш-таблицу `memo`:

```
def fib(n, memo):
```

Мы могли бы задать для `memo` значение по умолчанию, чтобы не нужно было явно передавать пустую хеш-таблицу при первом вызове функции:

```
def fib(n, memo={}):
```

Так или иначе базовые случаи 0 и 1 остаются прежними, и мемоизация на них никак не влияет.

Перед выполнением любых рекурсивных вызовов наш код сначала проверяет наличие результата вычисления `fib(n)` для заданного `n`:

```
if not memo.get(n):
```

Если он уже есть в хеш-таблице, мы просто возвращаем этот результат с помощью кода `return memo[n]`.

Если результата там нет, выполняем вычисление:

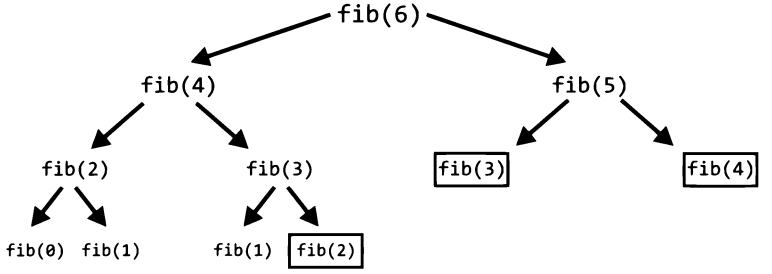
```
memo[n] = fib(n - 2, memo) + fib(n - 1, memo)
```

Сохраняем результат вычисления в хеш-таблице `memo`, чтобы не повторять этот процесс заново.

Обратите внимание на то, как мы передаем `memo` в качестве аргумента функции `fib` при каждом ее вызове, обеспечивая использование этой хеш-таблицы для всех вызовов функции `fib`.

Как видите, суть алгоритма осталась прежней. Мы все так же используем рекурсию для решения задачи, поскольку вычисление результата вызова `fib` все еще сводится к `fib(n - 2) + fib(n - 1)`. Но если этот результат новый, мы сохраняем его в хеш-таблице, а если он уже там есть — просто берем его оттуда, чтобы заново не вычислять.

Итак, рекурсивные вызовы нашей мемоизированной функции можно представить так:



На этой диаграмме прямоугольниками обведены вызовы, результат которых был извлечен из хеш-таблицы.

Какая же временная сложность у нашей функции теперь? Посмотрим, сколько рекурсивных вызовов мы совершаем при разных значениях *N*:

***N* элементов**	**Количество вызовов**
1	1
2	3
3	5
4	7
5	9
6	11

Итак, при наличии N элементов мы совершаем $2N - 1$ вызовов. О-нотация игнорирует константы, поэтому временная сложность этого алгоритма — $O(N)$.

Это в разы лучше, чем $O(2^N)$. Да здравствует мемоизация!

Восходящее динамическое программирование

Как я уже говорил, есть два способа динамического программирования. В прошлом разделе мы рассмотрели простую и эффективную технику под названием мемоизация.

Вторая техника, *восходящее динамическое программирование,* гораздо менее понятная, чем первая, и вообще не похожа на какой-то особый метод. Цель ее применения — отказ от рекурсии и использование для решения той же задачи другого инструмента (например, цикла).

Восходящий подход считается частью динамического программирования, так как суть динамического программирования сводится к тому, чтобы исключить из задачи, *которую можно реализовать рекурсивно,* дублирующиеся вызовы для перекрывающихся подзадач. Технически вместо рекурсии для этого можно использовать итерацию (то есть циклы).

Когда задачу можно выполнить с помощью рекурсии, такой подход начинает напоминать «метод». Например, нахождение чисел Фибоначчи с помощью итерации может потребовать больше усилий, так как итеративный подход не такой понятный (представьте, что вам нужно выполнить задачу с лестницей из прошлой главы с помощью цикла).

Посмотрим, как применить восходящий подход для создания функции вычисления элементов последовательности Фибоначчи.

Восходящий подход для вычисления элементов последовательности Фибоначчи

Здесь мы начинаем с первых двух чисел Фибоначчи: 0 и 1, и используем старую добрую итерацию для построения последовательности:

```
def fib(n):

  if n == 0:
    return 0

  # Начальные значения a и b - это первые два числа
  # последовательности соответственно:
```

```
    a = 0
    b = 1

    # Выполняем цикл, перебирая значения от 1 до n:
    for i in range(1, n):

      # a и b меняются, становясь следующими числами последовательности.
      # При этом новым значением b становится b + a, а новым значением a -
      # прежнее значение b. Чтобы внести эти изменения, используем
      # временную переменную:
      temp = a
      a = b
      b = temp + a

    return b
```

Начальные значения переменных `a` и `b` здесь — 0 и 1 соответственно, первые два числа Фибоначчи.

Чтобы вычислить каждое число последовательности вплоть до `n`, запускаем цикл:

```
for i in range(1, n):
```

Для определения каждого следующего числа последовательности складываем предпоследнее и последнее числа. Присваиваем переменной `temp` значение предпоследнего числа, а переменной `a` — последнего:

```
temp = a
a = b
```

Новое число последовательности, которое теперь будет храниться в переменной `b`, — это сумма двух предыдущих:

```
b = temp + a
```

Наш код — это простой цикл, перебирающий значения от 1 до N, поэтому алгоритм выполняет N шагов и обладает временной сложностью $O(N)$, как и тот, где использовалась мемоизация.

Мемоизация и восходящий подход

Итак, вы познакомились с двумя основными методами динамического программирования: мемоизацией и восходящим подходом. Какой из них лучше?

Обычно это зависит от конкретной задачи и причины, по которой вы используете рекурсию. Если рекурсия позволит вам выполнить задачу просто и качественно, можете придерживаться ее, применяя мемоизацию для решения про-

блемы перекрывающихся подзадач. Но если итеративный подход тоже понятен для вас, можете использовать и его.

Помните, что даже с применением мемоизации издержки при рекурсивном подходе больше, чем при итеративном. Во время рекурсии компьютер вынужден отслеживать все вызовы с помощью стека, который потребляет память, а мемоизация использует хеш-таблицы, которые тоже занимают место на компьютере (подробнее об этом — в главе 19).

В целом, восходящий подход обычно более предпочтителен. Но если вы считаете, что для реализации определенной задачи рекурсия подойдет вам больше, можете использовать ее, ускорив процесс с помощью мемоизации.

Выводы

Теперь, когда вы научились писать эффективный рекурсивный код, считайте, что вы обрели суперсилу. Дальше вы познакомитесь с очень эффективными и сложными алгоритмами, многие из которых опираются на принципы рекурсии.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 12».

1. Следующая функция принимает массив чисел и возвращает сумму, которая вычисляется, пока не превысит значение 100 при прибавлении какого-то числа. В таком случае это число игнорируется. Но функция выполняет лишние рекурсивные вызовы. Исправьте код, чтобы устранить ненужную рекурсию:

```
def add_until_100(array)
  return 0 if array.length == 0
  if array[0] + add_until_100(array[1, array.length - 1]) > 100
    return add_until_100(array[1, array.length - 1])
  else
    return array[0] + add_until_100(array[1, array.length - 1])
  end
end
```

2. Следующая функция использует рекурсию для определения N-го числа из последовательности Голомба. Но это ужасно неэффективно! Оптимизируйте функцию с помощью мемоизации (чтобы выполнить это упражнение,

вам вовсе не обязательно понимать, как вычисляется эта последовательность).

```
def golomb(n)
  return 1 if n == 1
  return 1 + golomb(n - golomb(golomb(n - 1)));
end
```

3. Это реализация задачи, связанной с поиском «уникальных путей», из прошлой главы. Используйте мемоизацию для повышения ее эффективности:

```
def unique_paths(rows, columns)
  return 1 if rows == 1 || columns == 1
  return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
end
```

ГЛАВА 13

Рекурсивные алгоритмы для ускорения выполнения кода

Как мы уже знаем, понимание рекурсии открывает возможности для разработки разных алгоритмов, например для обхода файловой системы или составления анаграмм. В этой главе вы узнаете, что рекурсия помогает создавать алгоритмы, способные ускорить выполнение кода.

В прошлых главах мы познакомились с рядом алгоритмов сортировки: пузырьком, выбором и вставками. Но в реальной жизни ни один из них не используется для сортировки массивов. Большинство языков программирования предусматривают встроенные функции сортировки массивов, которые позволяют не тратить время и силы на их реализацию. В основе многих из этих функций лежит так называемый алгоритм *быстрой сортировки* (Quicksort).

Мы собираемся подробно изучить механизм быстрой сортировки (несмотря на то, что он уже реализован во многих языках), потому что понимание принципа его работы позволяет использовать рекурсию для ускорения реальных алгоритмов.

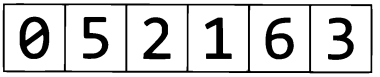
Quicksort — очень быстрый алгоритм сортировки, который особенно эффективен в средних случаях. Несмотря на то что в худших сценариях (когда элементы массивов отсортированы в обратном порядке) его скорость такая же, как у сортировки вставками и выбором, он работает намного быстрее в средних сценариях, которые реализуются чаще всего.

В основе алгоритма быстрой сортировки лежит концепция *разбиения,* поэтому сначала мы познакомимся именно с ней.

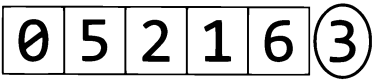
Разбиение

Разбиение (partitioning) массива — это выбор случайного значения, которое называется *опорным элементом* (pivot), и перераспределение остальных элементов так, чтобы те, которые меньше опорного, находились слева от него, а те, что больше, — справа. Рассмотрим алгоритм разбиения на примере.

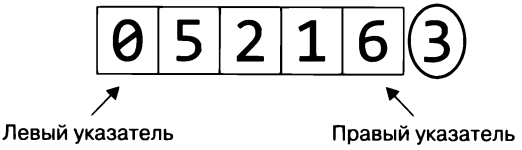
Допустим, у нас есть следующий массив:



Для согласованности мы всегда будем выбирать в качестве опорного элемента правое крайнее значение (но вообще можно выбрать любое другое). Здесь опорный элемент — число 3:



Теперь устанавливаем «указатели», один из которых направлен на крайнее левое значение массива, а другой — на крайнее правое, исключая опорный элемент:



Теперь мы готовы приступить к разбиению, которое состоит из следующих шагов (если вам что-то не понятно, не переживайте — все прояснится, когда мы обратимся к конкретному примеру).

1. Левый указатель последовательно смещается на одну ячейку вправо, вплоть до достижения значения, которое больше опорного или равно ему, после чего останавливается.
2. Правый указатель последовательно смещается на одну ячейку влево, пока не достигнет значения, которое меньше опорного или равно ему, после чего останавливается. Правый указатель останавливается и по достижении начала массива.
3. Остановка правого указателя означает, что мы на развилке. Если левый указатель достиг правого (или оказался справа от него), мы переходим

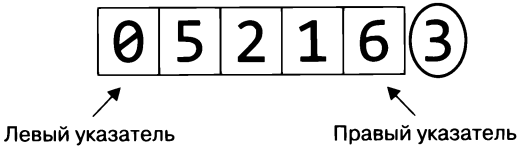
к выполнению шага 4. Если нет — меняем местами значения, на которые направлены левый и правый указатели, а затем повторяем шаги 1, 2 и 3.

4. Наконец меняем местами опорный элемент и значение, на которое сейчас направлен левый указатель.

После разбиения мы можем быть уверены, что все значения меньше опорного будут слева от него, а больше — справа. Это значит, что сам опорный элемент теперь в правильном положении в массиве, несмотря на то что остальные значения еще не полностью отсортированы.

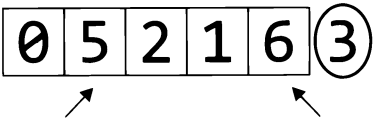
Рассмотрим этот процесс на примере.

Шаг 1: сравниваем значение, на которое направлен левый указатель (0), с опорным элементом (3):



Поскольку 0 меньше опорного значения, левый указатель смещается на одну ячейку.

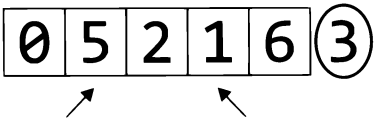
Шаг 2: перемещаем левый указатель:



Сравниваем значение, на которое указывает левый указатель (5), с опорным. Значение 5 не меньше опорного, поэтому левый указатель останавливается, и на следующем шаге мы активируем правый.

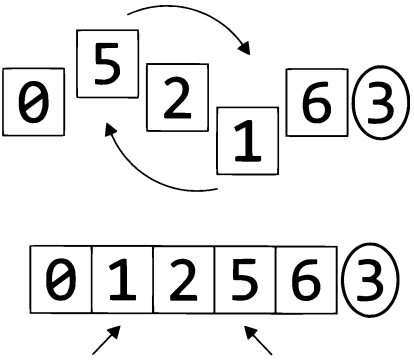
Шаг 3: сравниваем значение под правым указателем (6) с опорным. Оно превышает опорный элемент, поэтому указатель движется дальше.

Шаг 4: правый указатель смещается на одну ячейку:



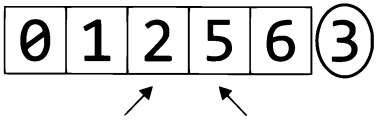
Сравниваем значение под правым указателем (1) с опорным. Оно меньше опорного элемента, поэтому правый указатель останавливается.

Шаг 5: оба указателя остановились, поэтому меняем местами значения, на которые они направлены:



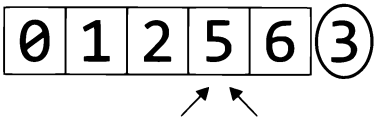
На следующем шаге мы снова активируем левый указатель.

Шаг 6: левый указатель смещается на одну ячейку:



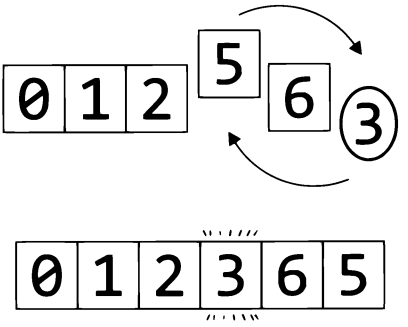
Сравниваем значение, на которое указывает левый маркер (2), с опорным. Значение 2 меньше опорного элемента, поэтому левый указатель движется дальше.

Шаг 7: левый указатель смещается на одну ячейку. Обратите внимание, что сейчас и левый, и правый указатели смотрят на одно значение:



Сравниваем значение под левым указателем с опорным элементом. Оно превышает опорное, и левый указатель останавливается. Так как левый указатель достиг правого, процесс перемещения указателей завершен.

Шаг 8: меняем местами опорный элемент и значение, на которое направлен левый указатель:



Хотя наш массив еще не полностью отсортирован, мы успешно выполнили разбиение. Теперь все числа меньше опорного элемента (3) находятся слева от него, а больше — справа. Это значит, что само значение 3 *теперь на месте*.

Программная реализация

Ниже приведена реализация класса `SortableArray` на языке Ruby, которая предусматривает метод `partition!`, разбивающий массив способом, который мы разобрали:

```
class SortableArray

  attr_reader :array

  def initialize(array)
    @array = array
  end

  def partition!(left_pointer, right_pointer)

    # В качестве опорного мы всегда выбираем крайний правый элемент.
    # Сохраняем индекс опорного элемента для дальнейшего использования:
    pivot_index = right_pointer

    # Считываем значение опорного элемента:
    pivot = @array[pivot_index]

    # Помещаем правый указатель слева от опорного элемента
    right_pointer -= 1

    while true

      # Перемещаем левый указатель вправо, пока он
      # не остановится на значении меньше опорного:
```

```
      while @array[left_pointer] < pivot do
        left_pointer += 1
      end

      # Перемещаем правый указатель влево до тех пор, пока он
      # не остановится на значении больше опорного:
      while @array[right_pointer] > pivot do
        right_pointer -= 1
      end

      # Левый и правый указатели остановились.

      # Проверяем положение левого указателя. Если он достиг
      # правого (или оказался справа от него), выходим из цикла,
      # чтобы далее в коде переместить опорный элемент:
      if left_pointer >= right_pointer
        break

      # Если левый указатель все еще слева от правого,
      # меняем местами значения, на которые они направлены:
      else
        @array[left_pointer], @array[right_pointer] =
          @array[right_pointer], @array[left_pointer]

        # Перемещаем левый указатель вправо,
        # готовясь к следующему раунду движения указателей
        left_pointer += 1
      end

    end

    # На последнем этапе разбиения меняем местами значение, на которое
    # направлен левый указатель, и опорный элемент:
    @array[left_pointer], @array[pivot_index] =
      @array[pivot_index], @array[left_pointer]

    # Возвращаем значение left_pointer для использования в методе
    # быстрой сортировки, который будет добавлен позднее
    return left_pointer
  end

end
```

Разберем этот код.

Метод `partition!` принимает в качестве параметров исходные позиции левого и правого указателей:

```
def partition!(left_pointer, right_pointer)
```

При его первом вызове эти указатели будут направлены на левый и правый концы массива соответственно. Но, как мы увидим далее, алгоритм быстрой сортировки предусматривает применение этого метода и к подразделам массива.

Получается, что левый и правый маркеры не всегда указывают на крайние значения массива, поэтому их позиции должны передаваться методу в качестве аргументов. Эта часть станет понятнее, когда мы познакомимся с полным алгоритмом быстрой сортировки.

Теперь выбираем опорный элемент, которым всегда служит крайнее правое значение обрабатываемого диапазона:

```
pivot_index = right_pointer
pivot = @array[pivot_index]
```

После этого наводим правый указатель `right_pointer` на элемент слева от опорного:

```
right_pointer -= 1
```

Запускаем цикл, который будет выполняться, пока левый `left_pointer` и правый `right_pointer` указатели не встретятся. Внутри этого цикла мы запускаем другой, чтобы продолжать перемещать левый указатель вправо до достижения элемента, который больше опорного или равен ему:

```
while @array[left_pointer] < pivot do
  left_pointer += 1
end
```

Точно так же перемещаем `right_pointer` влево, пока он не укажет на элемент меньше опорного или равный ему:

```
while @array[right_pointer] > pivot do
  right_pointer -= 1
end
```

Как только указатели остановятся, проверяем, не встретились ли они:

```
if left_pointer >= right_pointer
  break
```

Если встретились, выходим из цикла и готовимся к перемещению опорного элемента (этот процесс рассмотрим чуть позже). Если же нет, меняем местами значения, на которые они указывают:

```
@array[left_pointer], @array[right_pointer] =
  @array[right_pointer], @array[left_pointer]
```

Как только два указателя встретятся, мы меняем местами опорный элемент и значение, на которое указывает `left_pointer`:

```
@array[left_pointer], @array[pivot_index] =
  @array[pivot_index], @array[left_pointer]
```

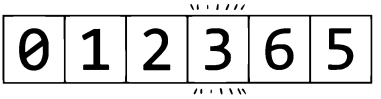
Функция возвращает значение `left_pointer`, которое потребуется для выполнения алгоритма быстрой сортировки (о котором я расскажу далее).

Алгоритм быстрой сортировки (Quicksort)

Алгоритм быстрой сортировки сочетает в себе разбиение и рекурсию и выполняется в несколько этапов.

1. Разбиваем массив, чтобы опорный элемент оказался на своем месте.
2. Рекурсивно повторяем шаги 1 и 2 для подмассивов слева и справа от опорного элемента. При этом мы последовательно разбиваем каждый подмассив на еще более мелкие слева и справа от соответствующего опорного элемента.
3. Когда остается пустой или одноэлементный подмассив, мы достигаем базового случая и ничего не делаем.

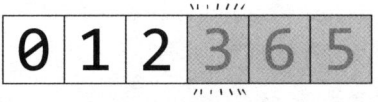
Вернемся к примеру. Мы начали с массива `[0, 5, 2, 1, 6, 3]` и выполнили одно разбиение. Поскольку алгоритм быстрой сортировки предусматривает такое разбиение, значит, мы уже выполнили один из его этапов и остановились на:



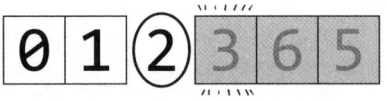
Исходным опорным элементом было число 3. Теперь, когда оно на своем месте, нам нужно отсортировать все значения слева и справа от него. Обратите внимание, что в нашем примере числа слева от опорного элемента уже отсортированы, но компьютер об этом еще не знает.

Следующий шаг после разбиения исходного массива — разбиение подмассива слева от опорного элемента.

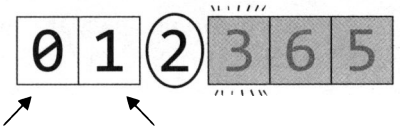
Чтобы не отвлекаться, на время скроем остальную часть массива:



Теперь из подмассива `[0, 1, 2]` выбираем опорным крайний правый элемент. Им будет число 2:



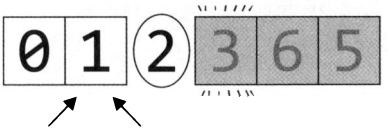
Устанавливаем левый и правый указатели в нужные положения:



Теперь мы готовы к разбиению этого подмассива. Продолжим с шага 8, на котором остановились ранее.

Шаг 9: сравниваем значение, на которое смотрит левый указатель (0), с опорным (2). Поскольку 0 меньше 2, продолжаем перемещение левого указателя.

Шаг 10: смещаем левый указатель на одну ячейку вправо — теперь он направлен на то же значение, что и правый:



Сравниваем значение под левым указателем с опорным. Так как 1 меньше опорного, двигаемся дальше.

Шаг 11: смещаем левый указатель на одну ячейку вправо — теперь он смотрит на опорный элемент:



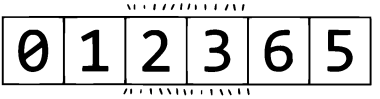
Левый маркер указывает на значение, равное опорному (ведь это значение и *есть* опорное!), поэтому останавливается.

Обратите внимание, как левый указатель сумел проскользнуть мимо правого. Но все в порядке. Алгоритм будет работать даже в таком случае.

Шаг 12: активируем правый указатель. Но, поскольку значение под ним (1) меньше опорного, он никуда не двигается.

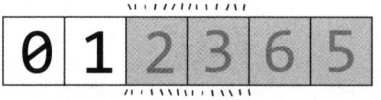
Так как левый указатель стоит справа от правого, мы прекращаем их перемещение на этом этапе разбиения.

Шаг 13: меняем местами опорный элемент и значение под левым указателем. Так получилось, что левый указатель смотрит на опорный элемент, который «меняется местами» с самим собой, что не приводит ни к каким изменениям. На этом процесс разбиения завершен и опорный элемент (2) теперь в правильной позиции:

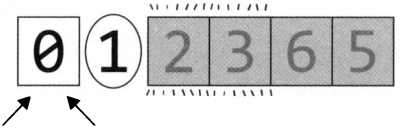


Теперь у нас есть подмассив `[0, 1]` слева от опорного элемента (2), а справа от него подмассива нет. Следующий этап — рекурсивное разбиение подмассива `[0, 1]` слева от опорного элемента. Нам не нужно проделывать то же самое с подмассивом справа, так как его не существует.

Поскольку на следующем шаге нас будет интересовать только подмассив `[0, 1]`, скроем остальную часть исходного массива:

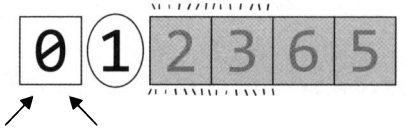


Чтобы разбить подмассив `[0, 1]`, сделаем крайний правый элемент (1) опорным. Куда же мы поместим левый и правый указатели? Они оба будут смотреть на 0, так как мы всегда располагаем правый указатель слева от опорного элемента. Дела у нас обстоят так:



Теперь мы готовы начать процесс разбиения.

Шаг 14: сравниваем значение под левым указателем (0) с опорным (1):



Оно меньше опорного, поэтому двигаемся дальше.

Шаг 15: сдвигаем левый указатель на одну ячейку вправо — теперь он направлен на опорный элемент:

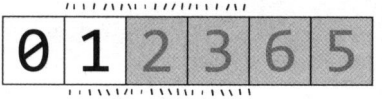


Поскольку значение под левым указателем (1) не меньше опорного (ведь оно и *есть* опорное), левый указатель останавливается.

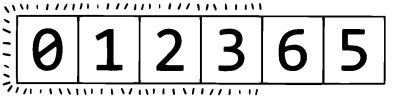
Шаг 16: сравниваем значение под правым указателем с опорным элементом. Он указывает на значение, которое меньше опорного, поэтому мы больше его не перемещаем, а левый указатель находится справа от правого, поэтому мы завершаем перемещение указателей на этом этапе разбиения.

Шаг 17: меняем местами значение под левым указателем с опорным элементом. Поскольку левый указатель сейчас смотрит на опорный элемент, это ничего нам не дает. Теперь опорный элемент находится на своем месте, и процесс разбиения завершен.

Его результат выглядит так:

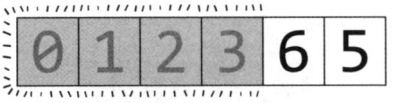


Теперь нам нужно разбить подмассив слева от последнего опорного элемента. У нас это одноэлементный подмассив `[0]`. Это базовый случай, поэтому мы ничего не делаем, предполагая, что элемент уже на своем месте. Итак, вот что у нас есть:

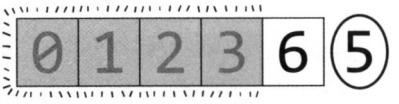


Мы начали с выбора значения 3 в качестве опорного и рекурсивно разбили подмассив слева от него (`[0, 1, 2]`). Теперь нам нужно сделать то же самое с подмассивом справа (`[6, 5]`).

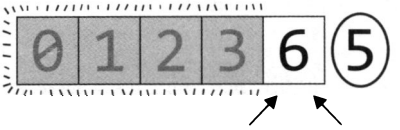
Скроем часть `[0, 1, 2, 3]`, так как уже отсортировали ее, и сосредоточимся только на подмассиве `[6, 5]`:



При его разбиении выбираем крайний правый элемент (5) в качестве опорного:



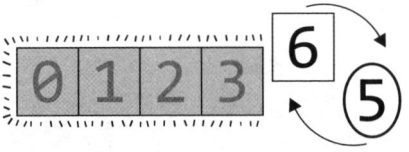
Левый и правый указатели смотрят на значение 6:



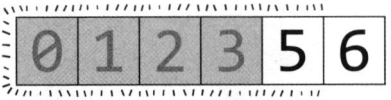
Шаг 18: сравниваем значение под левым указателем (6) с опорным (5). Поскольку 6 больше 5, левый указатель никуда не движется.

Шаг 19: правый указатель тоже направлен на 6, поэтому мы должны были бы перейти к следующей ячейке слева. Но слева ячеек нет, поэтому правый указатель тоже остается на месте. Левый указатель достиг правого, поэтому завершаем процесс перемещения указателей на этом этапе разбиения. Теперь мы готовы к выполнению последнего этапа.

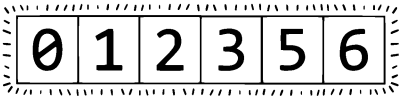
Шаг 20: меняем местами опорный элемент и значение под левым указателем:



Теперь наш опорный элемент (5) находится на месте:



Далее нам нужно рекурсивно разбить подмассивы слева и справа от опорного элемента (5). Слева от него подмассива нет, поэтому остается разбить только подмассив справа, но он содержит только один элемент `[6]`. Мы достигли базового случая и ничего не делаем, считая что значение 6 уже на своем месте:



Сортировка завершена!

Программная реализация

Ниже приведен код метода `quicksort!`, который мы можем объединить с предыдущим классом `SortableArray` для завершения алгоритма быстрой сортировки:

```
def quicksort!(left_index, right_index)
  # Базовый случай: пустой или одноэлементный подмассив:
  if right_index - left_index <= 0
    return
  end

  # Разбиваем диапазон элементов и получаем индекс опорного элемента:
  pivot_index = partition!(left_index, right_index)

  # Рекурсивно применяем метод quicksort! к тому, что
  # слева от опорного элемента:
  quicksort!(left_index, pivot_index - 1)

  # Рекурсивно применяем метод quicksort! к тому, что
  # справа от опорного элемента:
  quicksort!(pivot_index + 1, right_index)
end
```

Код здесь очень простой, но рассмотрим каждую его строку. Базовый случай пока пропустим.

Начнем с разбиения диапазона элементов между `left_index` и `right_index`:

```
pivot_index = partition!(left_index, right_index)
```

При первом запуске `quicksort!` мы разбиваем весь массив. Но в последующих вызовах эта строка кода разделяет любой диапазон элементов между `left_index` и `right_index`, который будет подразделом исходного массива.

Обратите внимание, что мы присваиваем значение, возвращаемое методом `partition!`, переменной `pivot_index`. Если вы помните, это будет `left_pointer` — позиция левого указателя, который к моменту завершения работы метода `partition!` достигает опорного элемента.

Теперь рекурсивно применяем метод `quicksort!` к подмассивам слева и справа от опорного элемента:

```
quicksort!(left_index, pivot_index - 1)
quicksort!(pivot_index + 1, right_index)
```

Рекурсия завершается, когда мы достигаем базового случая — пустого или одноэлементного подмассива:

```
if right_index - left_index <= 0
  return
end
```

Можем протестировать эту реализацию алгоритма быстрой сортировки с помощью следующего кода:

```
array = [0, 5, 2, 1, 6, 3]
sortable_array = SortableArray.new(array)
sortable_array.quicksort!(0, array.length - 1)
p sortable_array.array
```

Эффективность быстрой сортировки

Чтобы оценить эффективность алгоритма быстрой сортировки, сначала определим, насколько эффективен *один* этап разбиения.

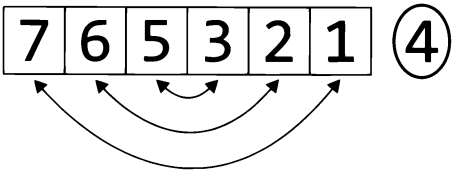
Сам процесс разбиения состоит из двух этапов:

- *сравнения* — сравниваем каждое из значений с опорным;
- *перестановки* — при необходимости меняем местами значения, на которые указывают левый и правый маркеры.

Каждый этап разбиения предусматривает как минимум N сравнений, так как мы сравниваем каждый элемент массива с опорным. При этом левый и правый указатели поочередно сдвигаются на ячейку, пока не достигнут одной точки.

Но количество перестановок будет зависеть от способа сортировки данных. Один этап разбиения предусматривает не более $N/2$ перестановок, ведь даже

если мы будем менять значения местами при каждой возможности, одна перестановка затрагивает два значения. На следующей схеме видно, что для разбиения массива из шести элементов нам нужно всего три перестановки:



Но в большинстве случаев менять значения местами на каждом шагу вовсе не обязательно. При работе с данными, отсортированными *произвольно*, мы переставляем примерно половину значений. Получается, в среднем мы выполняем около $N/4$ перестановок.

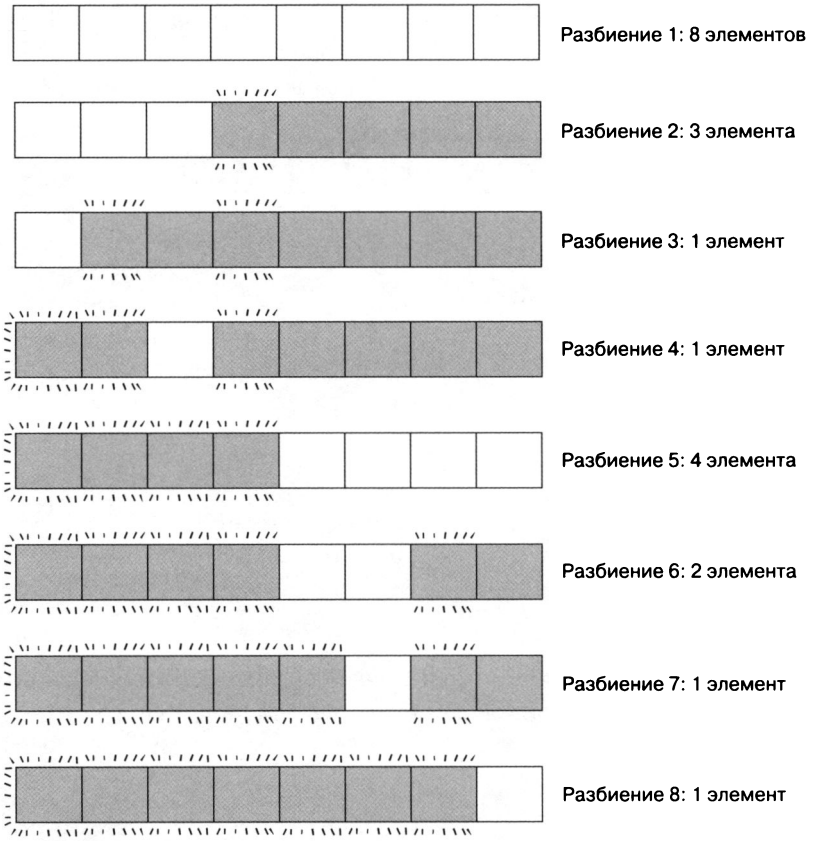
Итак, в среднем мы выполняем примерно N сравнений и $N/4$ перестановок. Можно сказать, что при наличии N элементов данных алгоритм выполняет около $1{,}25N$ шага. О-нотация игнорирует константы, поэтому разбиение выполняется за $O(N)$ времени.

Мы говорим об эффективности только *одного* этапа разбиения. Но алгоритм быстрой сортировки предусматривает *много* таких этапов, поэтому, чтобы определить эффективность алгоритма в целом, нужно провести дополнительный анализ.

Взгляд на быструю сортировку сверху

Чтобы упростить анализ, давайте рассмотрим диаграмму ниже, отражающую типичный процесс быстрой сортировки массива из восьми элементов. Здесь показано, сколько элементов затрагивает каждый этап разбиения. Я не стал указывать конкретные числа — сейчас это не важно. Обратите внимание, что активный подмассив на диаграмме — группа ячеек, не закрашенная серым цветом (см. рис. на с. 246).

Итак, перед нами восемь этапов разбиения, каждый из которых затрагивает подмассивы разных размеров. Мы выполняем разбиение исходного массива из восьми элементов, подмассивов с 4, 3 и 2 элементами и четырех одноэлементных массивов.



Суть быстрой сортировки как раз состоит в серии таких разбиений, на каждое из которых нужно около N шагов для N элементов каждого подмассива. Поэтому при сложении размеров всех подмассивов мы получим общее количество шагов алгоритма быстрой сортировки:

8 элементов

3 элементов

1 элемент

1 элемент

4 элемента

2 элемента

1 элемент

+ 1 элемент

Итого = примерно 21 шаг

Как мы видим, при сортировке исходного массива из восьми элементов алгоритм выполняет примерно 21 шаг (в лучшем или среднем сценарии, когда после каждого разбиения опорный элемент оказывается примерно в середине подмассива).

В случае с массивом из 16 элементов алгоритм быстрой сортировки выполняет около 64 шагов, а с массивом из 32 элементов — около 160. Взгляните на эту таблицу:

N	Количество шагов алгоритма быстрой сортировки (приблизительное)
4	8
8	24
16	64
32	160

Хотя в примере выше число шагов алгоритма при сортировке массива из 8 элементов было равно 21, здесь я указал 24. Точное число шагов может варьироваться. Такое значение я выбрал, потому что это вполне приемлемая аппроксимация, которая делает приведенное далее объяснение понятнее.

Вычисление эффективности быстрой сортировки с помощью О-нотации

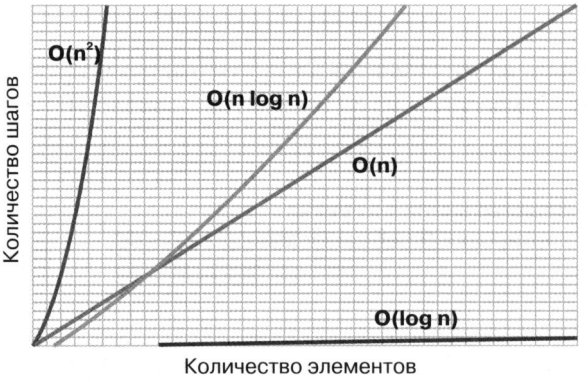
К какой категории алгоритмической сложности относится быстрая сортировка?

Если мы еще раз взглянем на приведенную выше закономерность, то заметим, что число шагов алгоритма быстрой сортировки при наличии *N* элементов в массиве примерно равно *N, умноженному на log N:*

N	log *N*	*N* × log *N*	Количество шагов алгоритма быстрой сортировки (приблизительное)
4	2	8	8
8	3	24	24
16	4	64	64
32	5	160	160

По сути, эффективность алгоритма быстрой сортировки выражается как O(*N* log *N*). Итак, мы обнаружили новую категорию алгоритмической сложности!

На следующем графике наглядно показана разница между O(N log N) и алгоритмами из других категорий.



То, что число шагов алгоритма быстрой сортировки согласуется с $N \times \log N$, это вовсе не совпадение. Если мы углубимся в этот процесс, то сможем понять, *почему* это именно так.

Когда мы разбиваем массив, то делим его на два подмассива. Если мы допустим, что в результате разбиения опорный элемент оказывается примерно в середине массива, как бывает в среднем случае, то два итоговых подмассива будут примерно одинакового размера.

Сколько раз мы должны разбить массив пополам, чтобы получить одноэлементные подмассивы? Для массива из N элементов нужно выполнить $\log N$ шагов. Взгляните на следующую схему (см. рис. на с. 249, сверху).

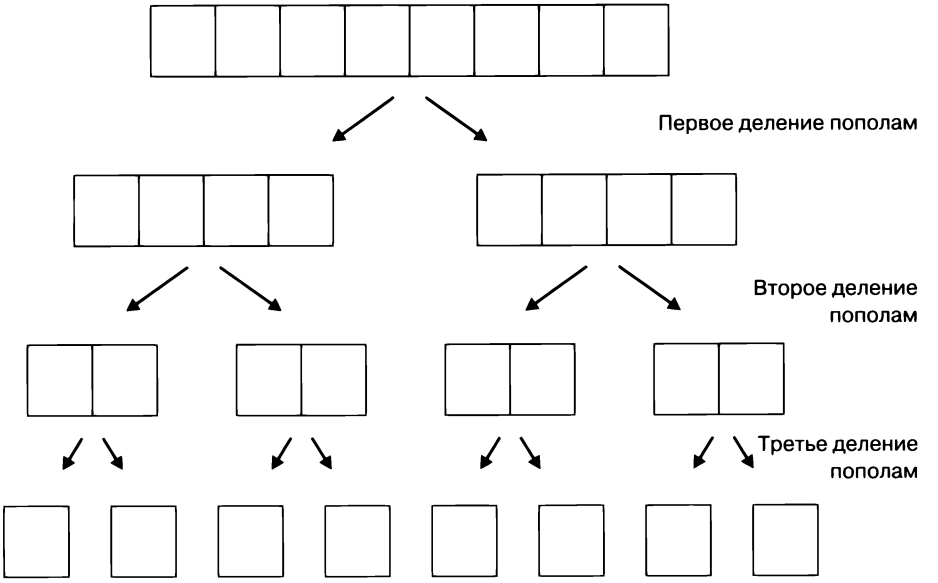
Как видите, чтобы разбить массив из восьми элементов на восемь самостоятельных массивов, нужно произвести три операции деления пополам — это и будет $\log N$, которое мы определяем как количество операций деления числа пополам для получения 1.

Вот почему алгоритм быстрой сортировки выполняет $N \times \log N$ шагов. Он предусматривает $\log N$ операций деления пополам, на каждую из которых приходится разбиение всех подмассивов, сумма элементов которых равна N (так как все подмассивы — части исходного массива из N элементов).

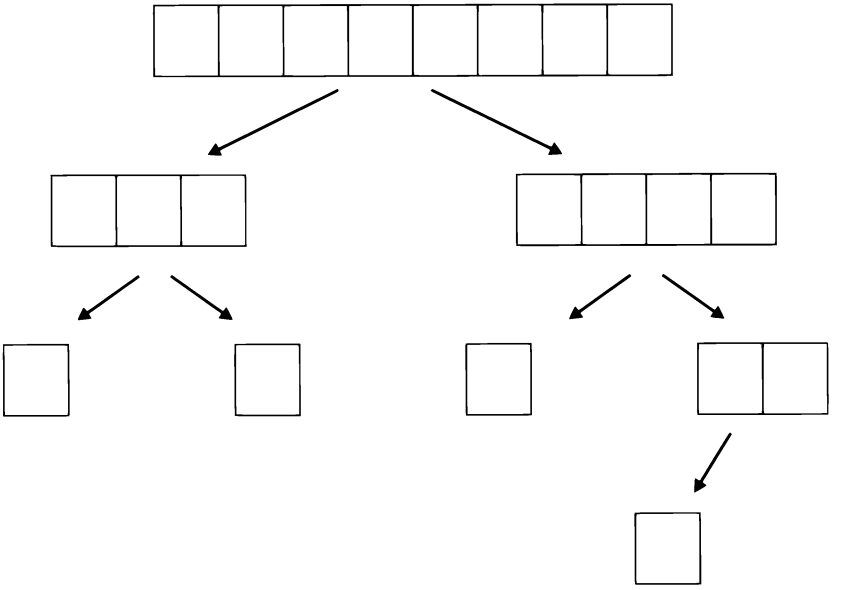
Это можно увидеть на диаграмме выше. В самой верхней части мы разбиваем исходный массив из восьми элементов на два подмассива из 4. Затем разбиваем оба эти подмассива, и этот процесс снова затрагивает все восемь элементов.

Но имейте в виду, что O($N \times \log N$) — это всего лишь аппроксимация. На самом деле мы сначала выполняем дополнительное разбиение исходного массива за

O(N) шагов. Кроме того, массив не разбивается на две равные половины, так как опорный элемент не участвует в этом процессе.



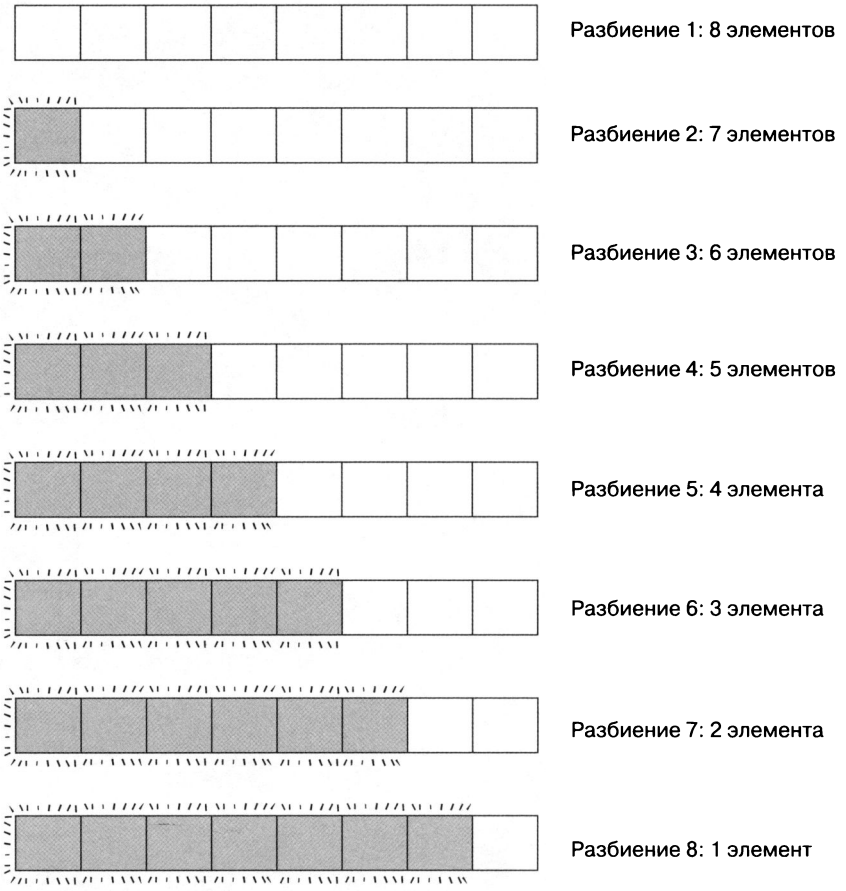
Вот как выглядит более реалистичный пример, где после каждого разбиения мы игнорируем опорный элемент:



Временная сложность быстрой сортировки в худшем сценарии

Для многих алгоритмов, с которыми мы сталкивались, лучшим был сценарий, когда массив уже был отсортирован. Но для быстрой сортировки лучший сценарий — тот, где после разбиения опорный элемент всегда оказывается точно в середине подмассива. Забавно, что обычно это происходит, когда значения в массиве перемешаны.

Худший сценарий для этого алгоритма — тот, где опорный элемент всегда оказывается не в середине, а в конце подмассива. Так происходит, если элементы массива упорядочены по возрастанию или убыванию. Вот как это выглядит:



Здесь видно, что опорный элемент всегда оказывается у левого края подмассива.

Хотя в этом случае каждое разбиение по-прежнему предполагает только одну перестановку, мы проигрываем из-за увеличения числа сравнений. В первом примере, где опорный элемент всегда оказывался примерно в середине, все разбиения после первого предполагали деление небольших подмассивов (самый крупный из которых содержал 4 элемента). Но здесь первые пять операций разбиения выполняются над подмассивами из 4 или более элементов и при каждом разбиении реализуется столько сравнений, сколько элементов в подмассиве.

Получается, что в этом худшем сценарии мы разбиваем массив на 8 + 7 + 6 + + 5 + 4 + 3 + 2 + 1 элементов, в общей сложности производя 36 сравнений.

Если говорить математическим языком, то при наличии N элементов алгоритм выполняет $N + (N - 1) + (N - 2) + (N - 3) ... + 1$ шагов. В главе 7 мы видели, что это количество шагов соответствует $N^2/2$ — с точки зрения O-нотации это будет $O(N^2)$.

Итак, в худшем случае алгоритм быстрой сортировки выполняется за $O(N^2)$ шагов.

Быстрая сортировка и сортировка вставками

Теперь, когда мы разобрались с принципом работы алгоритма быстрой сортировки, сравним его с одним из простейших алгоритмов сортировки — с сортировкой вставками:

	Лучший случай	**Средний случай**	**Худший случай**
Сортировка вставками	$O(N)$	$O(N^2)$	$O(N^2)$
Быстрая сортировка	$O(N \log N)$	$O(N \log N)$	$O(N^2)$

Как видите, в худшем случае сложность у всех одинаковая, а в лучшем сортировка вставками работает даже быстрее. Но в целом алгоритм быстрой сортировки превосходит сортировку вставками, потому что средний сценарий реализуется чаще всего. И в этих случаях сортировка вставками требует $O(N^2)$ шагов, тогда как быстрая сортировка — $O(N \log N)$.

Из-за превосходства алгоритма быстрой сортировки в средних случаях именно он лежит в основе функций сортировки во многих языках программирования. Поэтому вам вряд ли придется реализовывать его самостоятельно. Но вам может пригодиться еще один похожий алгоритм — алгоритм быстрого выбора.

Алгоритм быстрого выбора

Допустим, у нас есть массив произвольно отсортированных значений, который не нужно сортировать, но мы хотим найти в нем десятое наименьшее или пятое наибольшее значение. Это может пригодиться, например, если у нас много оценок за тест и мы хотим рассчитать для них 25-й процентиль или найти медианную оценку.

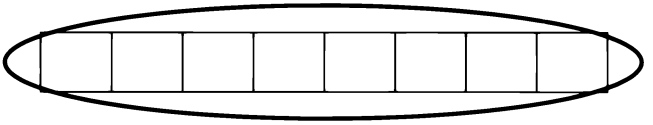
Один из способов реализации этой задачи — отсортировать весь массив, а затем перейти к соответствующему индексу.

Но даже при использовании алгоритма быстрой сортировки в среднем случае это заняло бы $O(N \log N)$ времени. И хотя это совсем неплохо, алгоритм *Quickselect* позволяет выполнить эту задачу еще быстрее. Как и алгоритм быстрой сортировки, он опирается на разбиения и может считаться своеобразным гибридом быстрой сортировки и двоичного поиска.

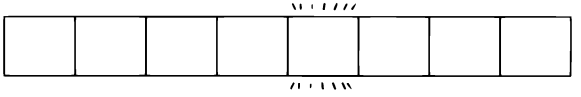
Как вы видели ранее в этой главе, в результате разбиения опорный элемент занимает свое место в массиве. Алгоритм быстрого выбора использует это так:

Допустим, у нас есть массив из восьми значений и мы хотим найти в нем второе наименьшее.

Сначала разбиваем исходный массив:



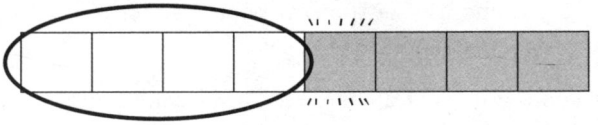
Мы надеемся, что после разбиения опорный элемент будет примерно в середине массива:



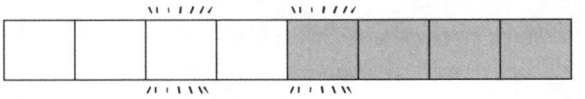
Теперь опорный элемент находится на месте, а поскольку он в пятой ячейке, мы знаем, чему равно пятое наименьшее значение в массиве.

И хотя мы ищем второе, а не пятое наименьшее значение, теперь мы знаем, что оно *где-то слева* от опорного элемента, и можем проигнорировать все, что справа от него, сосредоточившись на левом подмассиве. Этим алгоритм быстрого вызова напоминает двоичный поиск: мы последовательно делим массив пополам и фокусируемся на той половине, где находится искомое значение.

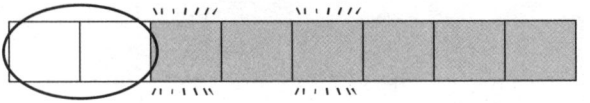
Теперь разбиваем подмассив слева от опорного элемента:



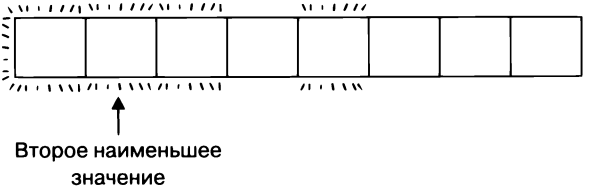
Допустим, в результате разбиения этого подмассива его опорный элемент оказывается в третьей ячейке:



Теперь мы знаем, что значение в третьей ячейке находится на своем месте, а значит, нам известно третье наименьшее значение. Так, по определению, второе наименьшее будет где-то слева от него. Теперь мы можем разбить подмассив слева от третьей ячейки:



После следующего разбиения наименьшее и второе наименьшее значения в массиве окажутся на своих местах:



Теперь мы можем получить значение из второй ячейки, точно зная, что это второе самое низкое значение во всем массиве. Одна из потрясающих особенностей алгоритма быстрого выбора в том, что мы можем найти нужное значение, *не сортируя весь массив.*

При выполнении быстрой сортировки каждое деление массива пополам сопровождалось разбиением каждого отдельного подмассива, из-за чего временная сложность была $O(N \log N)$. При использовании алгоритма быстрого выбора во время каждого деления массива пополам мы разбивали только интересующую нас половину — ту, где мы ожидали обнаружить искомое значение.

Эффективность алгоритма быстрого выбора

При анализе эффективности быстрого выбора становится ясно, что она составляет O(N) для средних сценариев. Почему?

В примере с массивом из восьми элементов мы выполнили три операции разбиения: одну — над исходным массивом из восьми элементов, одну — над подмассивом из четырех и одну — над подмассивом из двух.

Как вы помните, на каждое разбиение нужно около N шагов, где N — количество элементов подмассива. Получается, что общее количество шагов для выполнения трех разбиений равно 8 + 4 + 2 = 14. Итак, на разбиение массива из восьми элементов уходит примерно 14 шагов.

В случае с массивом из 64 элементов нужно около 64 + 32 + 16 + 8 + 4 + 2 = 126 шагов. Для 128 элементов — около 254, а для 256 элементов — 510.

Выходит, что для N элементов в массиве требуется выполнить около $2N$ шагов.

Мы можем сформулировать это иначе: при наличии N элементов нужно выполнить $N + (N/2) + (N/4) + (N/8) + ...$ 2 шага. В итоге всегда получается примерно $2N$ шагов.

Поскольку O-нотация игнорирует константы, мы отбрасываем 2 из $2N$ и в итоге сложность алгоритма быстрого выбора равна O(N).

Программная реализация

Ниже приведена реализация метода `quickselect!`, которую можно включить в описанный ранее класс `SortableArray`. Как видите, она очень похожа на код метода `quicksort!`:

```
def quickselect!(kth_lowest_value, left_index, right_index)
  # Если мы достигли базового случая - подмассива из одной ячейки,
  # мы точно знаем, что нашли искомое значение:
  if right_index - left_index <= 0
    return @array[left_index]
  end

  # Разбиваем массив и получаем индекс опорного элемента:
  pivot_index = partition!(left_index, right_index)

  # Если искомое значение слева от опорного элемента:
  if kth_lowest_value < pivot_index
    # Рекурсивно применяем алгоритм быстрого выбора к подмассиву
    # слева от опорного элемента:
    quickselect!(kth_lowest_value, left_index, pivot_index - 1)
  # Если искомое значение справа от опорного элемента:
  elsif kth_lowest_value > pivot_index
```

```
    # Рекурсивно применяем алгоритм быстрого выбора к подмассиву
    # справа от опорного элемента:
    quickselect!(kth_lowest_value, pivot_index + 1, right_index)
  else # если kth_lowest_value == pivot_index
    # Если после разбиения опорный элемент находится там же,
    # где и k-е наименьшее значение, значит, мы нашли то, что искали
    return @array[pivot_index]
  end
end
```

Чтобы найти второе наименьшее значение в неотсортированном массиве, запустите следующий код:

```
array = [0, 50, 20, 10, 60, 30]
sortable_array = SortableArray.new(array)
p sortable_array.quickselect!(1, 0, array.length - 1)
```

В качестве первого аргумента метод `quickselect!` принимает позицию искомого значения, начиная с индекса 0. Мы передали 1, чтобы указать на второе наименьшее значение в массиве. Второй и третий аргумент — крайние левый и правый индексы массива соответственно.

Сортировка как основа для других алгоритмов

Сложность самых быстрых из известных на момент написания этой книги алгоритмов сортировки — порядка $O(N \log N)$. Хотя алгоритм быстрой сортировки — один из самых популярных, есть и другие. Сортировка слиянием — это еще один известный рекурсивный алгоритм, относящийся к категории $O(N \log N)$, с которым я рекомендую ознакомиться.

То, что временная сложность самых быстрых алгоритмов сортировки — порядка $O(N \log N)$, очень важно, ведь сортировка может быть одной из составляющих других алгоритмов.

Например, вспомните задачу из главы 4, где нужно было проверить массив на предмет наличия повторяющихся значений.

Первый подход предполагал использование вложенных циклов и обладал временной сложностью $O(N^2)$. И хотя мы нашли решение, сложность которого $O(N)$, я намекнул на недостаток этого метода, связанный с дополнительным потреблением памяти (подробнее об этом — в главе 19). Итак, допустим, подход $O(N)$ мы не рассматриваем. Есть ли другие способы оптимизации решения с временной сложностью $O(N^2)$? Подсказка: это как-то связано с сортировкой!

На самом деле, если предварительно отсортировать массив, можно создать весьма эффективный и простой алгоритм.

Допустим, у нас есть исходный массив `[5, 9, 3, 2, 4, 5, 6]`. В нем два экземпляра значения 5, поэтому здесь мы сталкиваемся с наличием дубликатов.

Если мы заранее отсортируем этот массив, он примет вид `[2, 3, 4, 5, 5, 6, 9]`.

Далее мы можем использовать один цикл для перебора всех чисел. При обработке каждого числа мы проверяем, не идентично ли оно *следующему*. Если да, значит, мы нашли дубликат. Если мы дойдем до конца цикла, не обнаружив дубликатов, значит, в массиве их нет.

Хитрость здесь в том, что благодаря предварительной сортировке мы объединили повторяющиеся значения.

Мы начинаем с первого числа (`2`) и проверяем, не идентично ли оно следующему. Следующее число — `3`, поэтому пока дубликатов не обнаружено.

Теперь сравниваем `3` со следующим числом — `4` и двигаемся дальше. Сравниваем `4` с `5` и снова двигаемся дальше.

На этом этапе нам встречается первый экземпляр значения `5`, который мы сравниваем со следующим числом — вторым экземпляром того же значения. Ага! Мы нашли пару повторяющихся чисел, поэтому возвращаем `true`.

Так выглядит реализация этого алгоритма на языке JavaScript:

```
function hasDuplicateValue(array) {
  // Предварительная сортировка массива:
  // (В JavaScript следующий способ использования функции sort
  // нужен, чтобы отсортировать массив в числовом,
  // а не в алфавитном порядке)
  array.sort((a, b) => (a < b) ? -1 : 1);

  // Перебираем все значения в массиве до последнего:
  for(let i = 0; i < array.length - 1; i++) {

    // Если значение идентично следующему за ним,
    // значит, мы нашли дубликат:
    if(array[i] == array[i + 1]) {
      return true;
    }
  }

  // Если мы дошли до конца массива, не возвратив true,
  // значит, дубликатов в нем нет:
  return false;
}
```

Итак, в этом алгоритме сортировка используется в качестве одного из компонентов. Какова же его временная сложность?

Начнем с сортировки массива. Мы можем предположить, что сложность функции JavaScript `sort()` — $O(N \log N)$. Далее мы выполняем до N шагов при переборе элементов массива. Выходит, наш алгоритм выполняет $(N \log N) + N$ шагов.

Как вы помните, O-нотация учитывает только слагаемое самого высокого порядка, отбрасывая слагаемые более низкого. Здесь слагаемое N отбрасывается, поэтому мы сокращаем выражение до $O(N \log N)$.

Итак, мы использовали сортировку для реализации алгоритма с временной сложностью $O(N \log N)$. Это серьезная оптимизация по сравнению с исходным алгоритмом $O(N^2)$.

Сортировка используется во множестве разных алгоритмов. Теперь, делая так же, мы будем знать, что получаем алгоритм, который выполняется *не менее чем* за $O(N \log N)$. Конечно, он может работать медленнее по каким-то другим причинам, но мы точно знаем, что временная сложность $O(N \log N)$ всегда будет точкой отсчета.

Выводы

Рекурсивные алгоритмы быстрой сортировки и быстрого выбора позволяют находить простые и эффективные решения сложных задач. Это отличный пример того, как с помощью неочевидного, но хорошо продуманного алгоритма можно повысить производительность кода.

Теперь, когда мы разобрались с этими сложными алгоритмами, мы можем приступить к исследованию целого ряда дополнительных структур данных. Некоторые из них предусматривают операции, использующие рекурсию, но теперь вам не составит особого труда в них разобраться. Вы убедитесь, что каждая из этих структур не только интересна сама по себе, но и может значительно оптимизировать ваши приложения.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 13».

1. Вам дан массив положительных чисел. Напишите функцию, которая возвращает наибольшее произведение любых трех чисел. Подход с использованием трех вложенных циклов будет работать со скоростью $O(N^3)$, что очень медленно. Используйте сортировку для реализации алгоритма, так

чтобы он выполнялся за $O(N \log N)$ (есть и более быстрые реализации, но здесь нас интересует сортировка для ускорения выполнения кода).

2. Следующая функция находит в массиве целых чисел «недостающее». То есть ожидается, что в массиве будут все целые числа от 0 до n (где n — длина массива), кроме одного. Например, в массиве `[5, 2, 4, 1, 0]` нет числа `3`, а в `[9, 3, 2, 5, 6, 7, 1, 0, 4]` — 8.

   Вот реализация с временной сложностью $O(N^2)$ (работа одного только метода `include` занимает $O(N)$ шагов, поскольку компьютеру нужно произвести поиск по всему массиву, чтобы найти `n`):

   ```
   function findMissingNumber(array) {
     for(let i = 0; i < array.length; i++) {
       if(!array.includes(i)) {
         return i;
       }
     }

     // Если в массиве есть все числа:
     return null;
   }
   ```

   С помощью алгоритма сортировки напишите новую реализацию этой функции, временная сложность которой — $O(N \log N)$ (есть и более быстрые реализации, но здесь нас интересует сортировка как метод ускорения выполнения кода).

3. Напишите три разные реализации функции, которая находит наибольшее число в массиве. При этом сложность одной должна быть $O(N^2)$, второй — $O(N \log N)$, а третьей — $O(N)$.

ГЛАВА 14

Структуры данных на основе узлов

В следующих нескольких главах мы рассмотрим разные структуры, состоящие из *узлов* (node) — фрагментов данных, которые могут быть рассредоточены по всей памяти компьютера. Структуры данных на основе узлов предполагают новые способы организации данных и доступа к ним, позволяющие повысить производительность кода.

Здесь мы рассмотрим связный список — простейшую структуру данных на основе узлов, знакомство с которой поможет вам усвоить материал следующих глав. Как вы позже увидите, связные списки похожи на массивы, но обладают рядом особенностей, благодаря которым вы сможете повысить производительность кода.

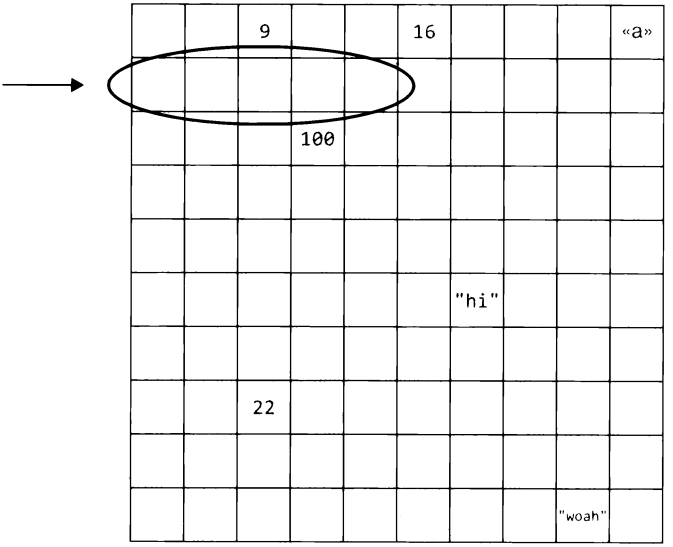
Связные списки

Связный список — это структура данных, которая, как и массив, представлена в виде списка элементов. Хотя на первый взгляд массивы и связные списки кажутся очень похожими, между ними есть ряд различий.

Как уже говорилось в главе 1, память компьютера похожа на гигантский набор ячеек, в которых хранятся биты данных. При создании массива компьютер выделяет в памяти непрерывную группу пустых ячеек для хранения данных приложения, как показано ниже.

Как вы помните, компьютер может получить доступ к любому адресу памяти за один шаг, а значит, мгновенно считать любое значение в массиве по его индексу. Если бы с помощью кода вы дали компьютеру команду «найти значение

по индексу 4», то он смог бы обратиться к соответствующей ячейке за один шаг. Все из-за того, что компьютер знает, с какого адреса памяти начинается массив: если с адреса 1000, то для нахождения значения по индексу 4 компьютеру достаточно обратиться к адресу памяти 1004.

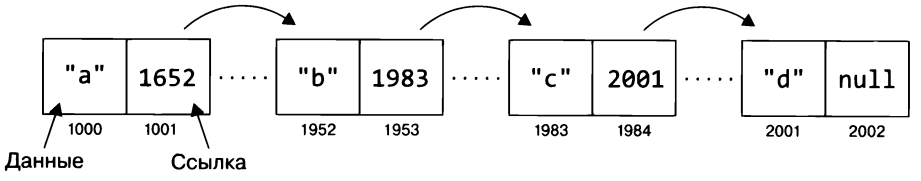


Связные списки работают иначе. Вместо того чтобы занимать непрерывный блок ячеек в памяти компьютера, данные в связных списках могут быть распределены по разным ячейкам памяти.

Связанные фрагменты данных, рассредоточенные по всей памяти, называются *узлами*. Каждый элемент связного списка представлен узлом. Возникает вопрос: если узлы находятся не рядом друг с другом в памяти, как компьютер узнает, какие входят в тот или иной список?

Дело в том, что каждый узел связного списка содержит дополнительный фрагмент информации: адрес памяти *следующего* узла этого списка — *ссылку*.

Вот так выглядит визуальный список:



Здесь представлен связный список из четырех элементов данных: `"a"`, `"b"`, `"c"` и `"d"`. Но для хранения этих данных используется *восемь* ячеек памяти, так как каждый узел состоит из двух ячеек. Первая содержит фактические данные, а вторая служит ссылкой на ячейку памяти, с которой начинается следующий узел. Ссылка конечного узла обозначается как `null`, так как этот узел последний.

Первый узел связного списка иногда называется его *головой*, а последний — *хвостом*.

Если компьютер знает, с какого адреса памяти начинается связный список, у него есть все необходимое для работы с ним! Поскольку каждый узел содержит ссылку на следующий, компьютеру нужно просто перейти по всем ссылкам, чтобы связать весь список воедино.

Возможность распределения данных по всей памяти компьютера — одно из преимуществ связного списка перед массивом. Для хранения своих данных массиву нужен целый блок смежных ячеек, найти которые становится все труднее по мере его роста. Конечно, вам не нужно об этом беспокоиться, ведь все это уже заложено в язык программирования. Но чуть позже я расскажу о более ощутимых различиях между списком и массивом, которые мы можем использовать в своих целях.

Реализация связного списка

В некоторых языках программирования, таких как Java, есть встроенные связные списки, в некоторых их нет, но реализовать их совсем несложно.

Создадим свой связный список на языке Ruby. Для его реализации мы будем использовать два класса: `Node` и `LinkedList`. Сначала создадим `Node`:

```
class Node

  attr_accessor :data, :next_node

  def initialize(data)
    @data = data
  end

end
```

У `Node` два атрибута: `data` содержит данные узла (например, строку `"a"`), а `next_node` — ссылку на следующий узел списка. Используем этот класс так:

```
node_1 = Node.new("once")
node_2 = Node.new("upon")
```

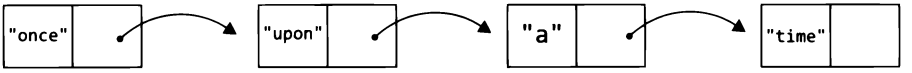
```
node_3 = Node.new("a")
node_4 = Node.new("time")

node_1.next_node = node_2
node_2.next_node = node_3
node_3.next_node = node_4
```

С помощью этого кода мы создали список из четырех узлов со строками `"once"`, `"upon"`, `"a"` и `"time"`.

Обратите внимание, что в нашей реализации `next_node` ссылается на другой *экземпляр* `Node`, а не на фактический адрес памяти. Но эффект тот же — узлы, скорее всего, рассредоточены по всей памяти компьютера, но мы можем использовать их ссылки для объединения списка.

Поэтому в дальнейшем под ссылкой мы будем подразумевать указатель на другой объект (узел), а не на конкретный адрес памяти. Для изображения связных списков мы будем использовать упрощенные схемы вроде этой:



Каждый узел здесь состоит из двух ячеек. Первая содержит данные узла, а вторая — указатель на следующий узел.

Это отражает структуру нашего класса `Node` на языке Ruby. В этой реализации метод `data` возвращает данные узла, а `next_node` — следующий узел в списке. *В этом контексте метод* `next_node` *служит ссылкой.*

Хотя нам удалось создать этот связный список с помощью одного лишь класса `Node`, нам нужно как-то указать программе на его начало. Для этого в дополнение к предыдущему классу `Node` мы создадим класс `LinkedList`. Так выглядит его простейшая форма:

```
class LinkedList

  attr_accessor :first_node

  def initialize(first_node)
    @first_node = first_node
  end

end
```

Все, что сейчас делает экземпляр `LinkedList`, — это отслеживает первый узел списка.

Ранее мы создали цепочку узлов, содержащую `node_1`, `node_2`, `node_3` и `node_4`. Теперь мы можем использовать наш класс `LinkedList`, чтобы сослаться на этот список:

```
list = LinkedList.new(node_1)
```

Сейчас эта переменная `list` действует как дескриптор связного списка, поскольку служит экземпляром `LinkedList`, у которого есть доступ к первому узлу списка.

Выходит, *при работе со связным списком мы можем получить мгновенный доступ только к его первому узлу*. Как мы скоро убедимся, это влечет за собой серьезные последствия.

Но на первый взгляд связные списки и массивы очень похожи — оба представлены просто в виде списка элементов. Но если начать анализировать две эти структуры данных, можно увидеть существенные различия в их производительности! Начнем с рассмотрения четырех классических операций — чтения, поиска, вставки и удаления.

Чтение

Как вы помните, компьютер может считать значение из массива за время O(1). Теперь определим эффективность чтения из связного списка.

Компьютер не сможет считать, скажем, значение третьего элемента связного списка за один шаг, потому что он заранее не знает, в какой ячейке памяти его нужно искать. В конце концов, узлы связного списка могут быть где угодно! Все, что изначально известно нашей программе, — адрес памяти, соответствующий *первому* узлу связного списка. Но она не знает, где находятся другие узлы.

Итак, чтобы считать значение третьего узла, компьютер должен выполнить несколько шагов: обратиться к первому узлу, перейти по ссылке ко второму и к третьему.

Получается, чтобы добраться до нужного узла, мы всегда будем начинать с первого (единственного узла, к которому у нас есть доступ) и следовать по цепочке вплоть до достижения искомого.

Иначе говоря, для чтения значения последнего узла списка из N узлов нам нужно выполнить N шагов. Связные списки, чтение которых в худшем случае выполняется за O(N) шагов, значительно уступают массивам, позволяющим считать любой элемент за O(1). Но не волнуйтесь — у связных списков еще будет время проявить себя, как мы скоро увидим.

Программная реализация: чтение элементов связного списка

Добавим в наш класс `LinkedList` метод `read`:

```
def read(index)
  # Начинаем с первого узла списка:
  current_node = first_node
  current_index = 0
  while current_index < index do
    # Последовательно переходим по ссылкам, пока не доберемся до
    # индекса искомого узла:
    current_node = current_node.next_node
    current_index += 1

    # Если мы вышли за пределы списка, значит, нужного
    # индекса в нем нет, поэтому возвращаем nil:
    return nil unless current_node
  end

  return current_node.data
end
```

Чтобы прочитать значение, к примеру, четвертого узла списка, вызываем наш метод, передавая ему индекс нужного узла:

```
list.read(3)
```

Проанализируем процесс работы этого метода.

Сначала мы создаем переменную `current_node`. Она ссылается на узел, к которому мы сейчас обращаемся. Для доступа к первому узлу мы используем код:

```
current_node = first_node
```

Как вы помните, `first_node` — это атрибут класса `LinkedList`.

Мы отслеживаем и индекс текущего узла `current_node`, чтобы в нужный момент понять, что достигли искомого индекса. Это делается с помощью кода:

```
current_index = 0
```

поскольку индекс первого узла — 0.

Затем мы запускаем цикл, который выполняется, пока значение `current_index` остается меньше искомого индекса:

```
while current_index < index do
```

В рамках каждой итерации цикла мы обращаемся к каждому следующему узлу списка, делая его новым текущим узлом `current_node`:

```
current_node = current_node.next_node
```

Увеличиваем значение `current_index` на 1:

```
current_index += 1
```

По окончании каждого прохода проверяем, не достигли ли мы конца списка, и возвращаем `nil`, если индекса, к которому мы обращаемся, в списке нет:

```
return nil unless current_node
```

Это работает, потому что в случае последнего узла значение `next_node` — `nil`, так как последнему узлу никогда не присваивалась ссылка на следующий. Получается, когда мы вызываем `current_node = current_node.next_node` для последнего узла, значение `current_node` становится равным `nil`.

Наконец, мы выходим из цикла только по достижении нужного индекса, а затем возвращаем значение соответствующего узла с помощью кода:

```
return current_node.data
```

Поиск

Как вы уже знаете, поиск — это нахождение значения в списке и возвращение его индекса. Мы уже убедились, что линейный поиск в массиве занимает O(N) времени, ведь компьютеру нужно последовательно проверить все значения.

В случае со связными списками временная сложность поиска тоже равна O(N), потому что для поиска нам нужно пройти тот же процесс, что и при чтении: мы начинаем с первого узла и переходим по ссылкам, попутно проверяя значение каждого, пока не найдем то, что ищем.

Программная реализация: поиск по связному списку

Так мы можем выполнить операцию поиска на языке Ruby. Назовем этот метод `index_of` и передадим ему искомое значение:

```
def index_of(value)
  # Начинаем с первого узла списка:
  current_node = first_node
  current_index = 0

  begin
    # Обнаружив искомые данные, возвращаем их:
    if current_node.data == value
      return current_index
    end

    # В противном случае переходим к следующему узлу:
```

```
    current_node = current_node.next_node
    current_index += 1
  end while current_node

  # Если мы проходим весь список,
  # не обнаружив искомых данных, возвращаем nil:
  return nil
end
```

Теперь можно искать любое значение в списке с помощью кода:

```
list.index_of("time")
```

Как видите, поиск очень похож на чтение. Разница в том, что здесь цикл не останавливается на определенном индексе, а выполняется, пока мы либо не найдем нужное значение, либо не достигнем конца списка.

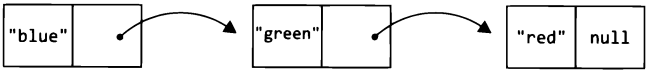
Вставка

Признаю, до сих пор связные списки не демонстрировали чудеса производительности. Они ничем не лучше массивов в плане поиска и сильно уступают им в плане чтения. Но не беспокойтесь, эта структура данных еще проявит себя. И этот момент как раз настал.

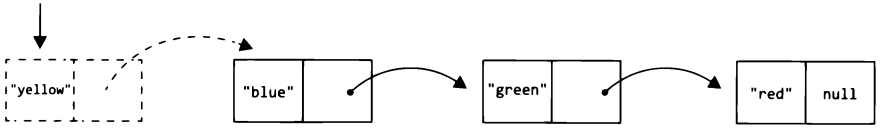
Вставка — это одна из операций, где *в определенных ситуациях* у связных списков явное преимущество перед массивами.

Напомним, что худший сценарий для вставки значения в массив — это когда программа сохраняет данные в позицию с индексом 0, потому что сначала ей приходится сдвинуть остальные данные на ячейку вправо, что требует времени O(N). Но в случае связных списков вставка в начало выполняется всего за шаг — за время O(1). Выясним почему.

Допустим, у нас есть следующий связный список:



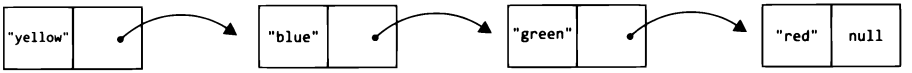
Чтобы добавить значение `"yellow"` в начало списка, достаточно создать новый узел со ссылкой, указывающей на узел со значением `"blue"`:



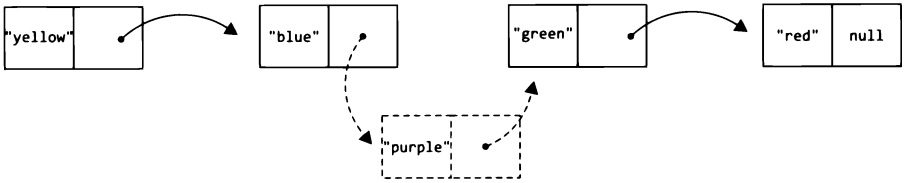
Нам еще придется обновить экземпляр `LinkedList` в коде, чтобы его атрибут `first_node` указывал на узел со значением `"yellow"`.

В отличие от массива, связный список позволяет вставлять данные в начало без перемещения остальных данных. Здорово, правда?

Теоретически вставка данных *в любое место* связного списка выполняется всего за шаг, но есть нюанс. Вернемся к нашему примеру. Вот как сейчас выглядит связный список:

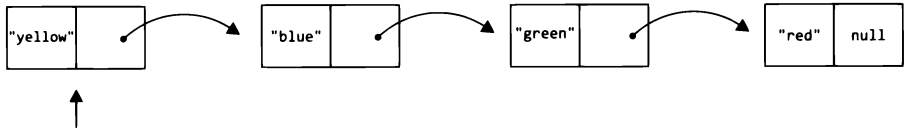


Допустим, теперь мы хотим вставить значение `"purple"` по индексу 2 (то есть между `"blue"` и `"green"`). Фактическая вставка выполняется за один шаг. Для этого мы можем создать новый узел со значением `"purple"` и просто изменить ссылку узла со значением `"blue"` так, чтобы она указывала на этот новый узел:

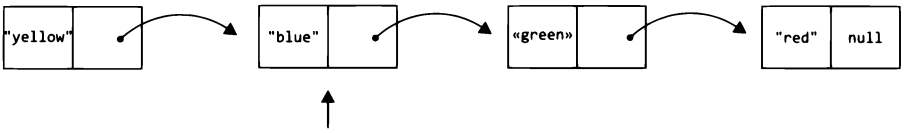


Но чтобы это сделать, компьютеру сначала нужно *добраться* до узла с индексом 1 (`"blue"`) и получить возможность изменить его ссылку, чтобы она указывала на созданный узел. А как мы уже знаем, чтение связного списка, то есть обращение к элементу по заданному индексу, выполняется за время $O(N)$. Рассмотрим этот процесс подробнее.

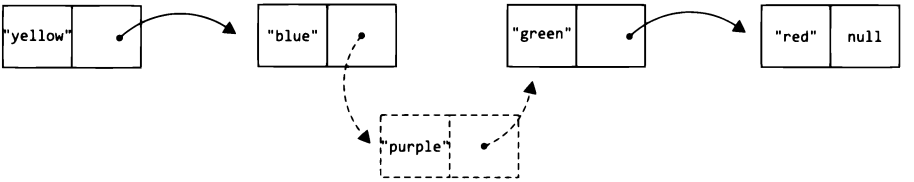
Итак, мы хотим добавить новый узел после узла с индексом 1. Первым делом компьютеру нужно добраться до этого индекса 1. Для этого придется начать с самого начала списка:



Затем, переходя по первой ссылке, мы получаем доступ к следующему узлу:



Обнаружив индекс 1, мы наконец можем добавить новый узел:



Здесь на вставку значения `"purple"` у нас ушло три шага. Если бы мы хотели добавить его в конец списка, шагов понадобилось бы пять: четыре для получения доступа к индексу 3 и один — для вставки нового узла.

Выходит, на практике временная сложность вставки значения в связный список — O(N), так как в худшем случае вставка в конец требует выполнения $N + 1$ шагов.

Но, как мы убедились, в лучшем случае вставка в *начало* выполняется за время O(1).

Интересно, что лучший и худший сценарии для массивов и связных списков противоположны друг другу:

Сценарий	Массив	Связный список
Вставка в начало	Худший случай	Лучший случай
Вставка в середину	Средний случай	Средний случай
Вставка в конец	Лучший случай	Худший случай

Как видите, массивы выигрывают при вставке значений в конец, а связные списки — при вставке в начало.

Итак, мы выяснили, что одно из преимуществ связных списков — быстрая вставка элементов в начало. Чуть позже в этой главе мы увидим отличный практический пример того, как можно этим воспользоваться.

Программная реализация: вставка элемента в связный список

Добавим в наш класс `LinkedList` метод вставки `insert_at_index`:

```
def insert_at_index(index, value)
    # Создаем новый узел с указанным значением:
    new_node = Node.new(value)

    # Если мы вставляем значение в начало списка:
    if index == 0
      # Делаем так, чтобы наш новый узел ссылался на тот, который был первым
      # ранее:
      new_node.next_node = first_node
      # Устанавливаем новый узел в качестве первого узла списка:
      self.first_node = new_node
      return
    end

    # Если мы вставляем значение не в начало списка:

    current_node = first_node
    current_index = 0

    # Сначала обращаемся к узлу, который находится
    # перед местом вставки нового узла:
    while current_index < (index - 1) do
      current_node = current_node.next_node
      current_index += 1
    end

    # Делаем так, чтобы ссылка нового узла указывала на следующий узел:
    new_node.next_node = current_node.next_node

    # Изменяем ссылку предыдущего узла, чтобы она указывала
    # на новый узел:
    current_node.next_node = new_node
  end
```

Чтобы использовать метод, мы передаем как новое значение (`value`), так и индекс места, куда мы хотим его вставить (`index`).

Например, чтобы вставить значение `"purple"` по индексу 3, используем следующий код:

```
list.insert_at_index(3, "purple")
```

Давайте разберем код метода `insert_at_index` подробней.

Сначала мы создаем новый экземпляр `Node` со значением, переданным нашему методу:

```
new_node = Node.new(value)
```

Далее обрабатываем случай вставки значения по индексу 0 (в начало списка). Этот процесс отличается от вставки значения в другие места списка, поэтому мы обрабатываем его отдельно.

Чтобы вставить значение в начало списка, нужно сделать так, чтобы ссылка new_node указывала на первый узел списка, и объявить new_node первым узлом:

```
if index == 0
  new_node.next_node = first_node
  self.first_node = new_node
  return
end
```

Ключевое слово return завершает работу метода, так как ему больше нечего делать.

Остальная часть кода обрабатывает случай вставки значения в любое место списка, кроме начала.

Как и в случае чтения и поиска, мы начинаем с получения доступа к первому узлу списка:

```
current_node = first_node
current_index = 0
```

Затем используем цикл while для получения доступа к узлу, который находится *перед* местом вставки new_node:

```
while current_index < (index - 1) do
  current_node = current_node.next_node
  current_index += 1
end
```

Сейчас текущий узел current_node — тот, который будет находиться перед новым узлом new_node.

Теперь задаем для нового узла ссылку на тот, который находится после текущего.

```
new_node.next_node = current_node.next_node
```

Наконец, меняем ссылку узла current_node (который находится перед новым), чтобы она указывала на новый узел new_node:

```
current_node.next_node = new_node
```

Готово!

Удаление

Связные списки весьма эффективны и при удалении элементов, особенно из начала списка.

Чтобы удалить первый узел связного списка, достаточно изменить значение `first_node`, чтобы оно соответствовало второму узлу связного списка.

Вернемся к нашему примеру со списком, содержащим значения `"once"`, `"upon"`, `"a"` и `"time"`. Чтобы удалить значение `"once"`, просто задаем в качестве начального узла списка значение `"upon"`:

```
list.first_node = node_2
```

Сравните это с массивом, где удаление первого элемента означает сдвиг всех оставшихся данных на одну ячейку влево и занимает O(N) времени.

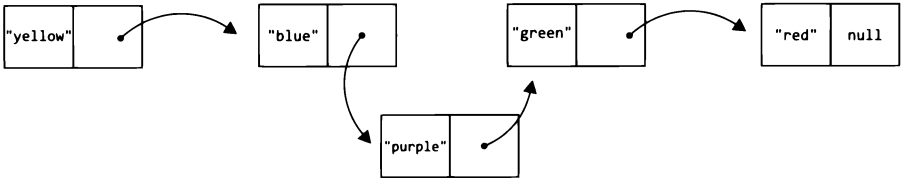
Удаление *последнего* узла связанного списка занимает один шаг — мы просто берем предпоследний узел и меняем его ссылку на `null`. Но для получения доступа к нему уйдет N шагов, так как нам нужно проследовать по всем ссылкам, с начала списка.

В следующей таблице сравниваются разные сценарии удаления элементов из массивов и связных списков. Обратите внимание на совпадение с процессом вставки:

Сценарий	Массив	Связный список
Удаление из начала	Худший случай	Лучший случай
Удаление из середины	Средний случай	Средний случай
Удаление с конца	Лучший случай	Худший случай

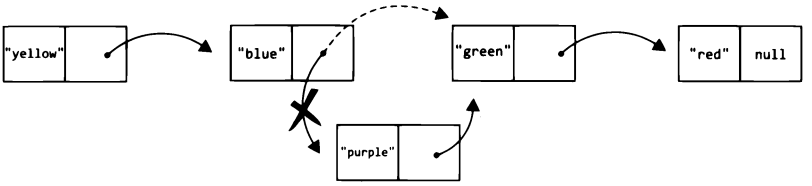
В то время как удалить узел из начала или конца связного списка довольно просто, удалить его из любого другого места будет уже несколько сложнее.

Допустим, мы хотим удалить значение по индексу 2 (`"purple"`) из нашего связного списка:



Для этого нам нужно сначала получить доступ к узлу, *предшествующему* тому, который мы удаляем (`"blue"`). А после поменять его ссылку, чтобы она указывала на узел, который находится *после* удаляемого узла (`"green"`).

На следующей диаграмме показано изменение ссылки узла `"blue"` с `"purple"` на `"green"`:



Интересно то, что всякий раз, когда мы удаляем узел из связного списка, он не исчезает из памяти компьютера. Мы просто исключаем его из своего списка, не позволяя другим узлам ссылаться на него. Это приводит к *эффекту* удаления узла из списка, даже если он до сих пор существует в памяти.

Разные языки программирования обрабатывают эти удаленные узлы по-разному. Некоторые автоматически обнаруживают неиспользуемые узлы и обрабатывают их с помощью «сборщика мусора» для освобождения памяти.

Программная реализация: удаление элемента из связного списка

Добавим в класс `LinkedList` метод удаления `delete_at_index`, принимающий в качестве аргумента индекс удаляемого узла:

```
def delete_at_index(index)
  # Если мы удаляем первый узел:
  if index == 0
    # Просто устанавливаем в качестве первого узла тот,
    # который сейчас второй:
    self.first_node = first_node.next_node
    return
  end

  current_node = first_node
  current_index = 0

  # Сначала находим узел, предшествующий тому, который мы
  # хотим удалить, и называем его текущим current_node:
  while current_index < (index - 1) do
    current_node = current_node.next_node
    current_index += 1
  end

  # Находим узел, следующий за удаляемым:
  node_after_deleted_node = current_node.next_node.next_node
```

```
    # Меняем ссылку current_node, чтобы она указывала на узел,
    # следующий за удаляемым, исключая удаляемый узел из списка:
    current_node.next_node = node_after_deleted_node
  end
```

Этот метод похож на `insert_at_index`, описанный ранее. Но давайте рассмотрим некоторые новые моменты.

Сначала он обрабатывает случай с индексом, равным 0, — ситуацию удаления первого узла списка. Код, решающий эту задачу, до смешного прост:

```
self.first_node = first_node.next_node
```

Достаточно указать в качестве начального узла (`first_node`) нашего списка второй!

Остальной код метода обрабатывает случай удаления узла в любом другом месте списка. Для этого мы используем цикл `while`, чтобы получить доступ к узлу, предшествующему тому, который мы хотим удалить. Он становится текущим (`current_node`).

Теперь берем узел, который находится *после* удаляемого, и сохраняем в переменной `node_after_deleted_node`:

```
node_after_deleted_node = current_node.next_node.next_node
```

Обратите внимание на наш маленький трюк в доступе к этому узлу: это просто узел, который идет через два после `current_node`!

Изменяем ссылку текущего узла (`current_node`), чтобы она указывала на тот, что следует за удаляемым (`node_after_deleted_node`):

```
current_node.next_node = node_after_deleted_node
```

Эффективность операций над связными списками

В следующей таблице указана временная сложность основных операций над связными списками и массивами:

Операция	**Массив**	**Связный список**
Чтение	O(1)	O(*N*)
Поиск	O(*N*)	O(*N*)
Вставка	O(*N*) (O(1) в конце)	O(*N*) (O(1) в начале)
Удаление	O(*N*) (O(1) в конце)	O(*N*) (O(1) в начале)

На первый взгляд, связные списки менее эффективны, чем массивы. Они работают так же быстро при поиске, вставке и удалении значений и намного медленнее при их чтении. Если так, то зачем вообще их использовать?

Настоящий потенциал связного списка в том, что при его использовании *фактическая вставка и удаление* выполняются за время O(1).

Но разве это не актуально только при вставке или удалении в начале списка? Мы же видели, что для вставки и удаления в другом месте нужно выполнить до N шагов только для получения доступа к нужному индексу!

Но иногда доступ к нужному узлу уже может быть получен с какой-то другой целью. Рассмотрим следующий пример.

Связные списки в действии

Одна из ситуаций, где можно увидеть всю мощь связных списков, — проверка списка и удаление из него множества элементов. Допустим, мы создаем приложение, которое просматривает существующие списки адресов электронной почты и удаляет из них все с недопустимым форматом.

Не важно, массив это или связный список — нам нужно последовательно проверить все адреса в нем. На это уходит N шагов. Но что же происходит при удалении этих адресов?

В случае с массивом при удалении каждого адреса электронной почты мы вынуждены тратить дополнительное время O(N) на то, чтобы сдвинуть оставшиеся данные влево для заполнения пробела. И все это нужно сделать до того, как мы сможем проверить следующий адрес.

Допустим, 1 из каждых 10 адресов электронной почты — с неверным форматом. В списке из 1000 адресов таких было бы 100. Тогда на чтение всех 1000 адресов у нашего алгоритма ушло бы 1000 шагов, а на удаление — до 100 000 дополнительных шагов, так как при удалении каждого из 100 адресов мы вынуждены смещать до 1000 остальных элементов.

Но при использовании связного списка удаление каждого элемента выполняется всего за шаг, поскольку для этого достаточно изменить ссылку узла, чтобы она указывала на нужный, после чего мы можем двигаться дальше. Получается, что для обработки списка из 1000 адресов нашему алгоритму пришлось бы выполнить 1100 шагов — 1000 для чтения и 100 для удаления.

Выходит, связные списки — это отличная структура данных, позволяющая эффективно вставлять и удалять значения, не сдвигая при этом остальные данные.

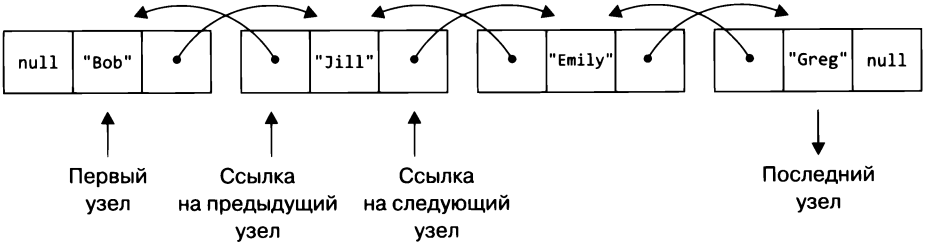
Двусвязные списки

На самом деле связные списки бывают разные. Тот, который мы обсуждали до сих пор, — «классический». Но можно внести небольшие изменения и наделить его дополнительными сверхспособностями.

Одна из разновидностей связного списка — *двусвязный*, или *двунаправленный*, *список*.

Он похож на обычный за исключением того, что у каждого узла в нем две ссылки: одна указывает на следующий узел, а другая — на *предыдущий*. А еще двусвязный список всегда отслеживает и первый, и последний узлы, а не только первый.

Двусвязный список выглядит так:



Реализовать двусвязный список на языке Ruby можно так:

```
class Node

  attr_accessor :data, :next_node, :previous_node

  def initialize(data)
    @data = data
  end

end

class DoublyLinkedList

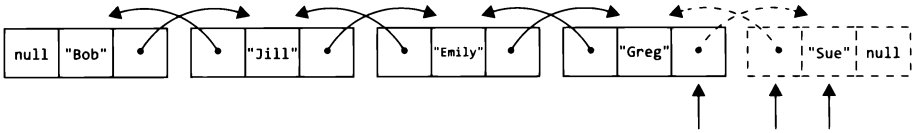
  attr_accessor :first_node, :last_node

  def initialize(first_node=nil, last_node=nil)
    @first_node = first_node
    @last_node = last_node
  end

end
```

Двусвязный список всегда знает, где находятся его первый и последний узлы, поэтому мы можем получить доступ к каждому из них за один шаг. Получается, мы можем считывать, вставлять и удалять значения за время O(1) как из начала списка, так и с его конца.

Так выглядит процесс вставки узла в конец двусвязного списка:



Как видите, мы создаем новый узел (`"Sue"`), одна из ссылок которого указывает на предыдущий, который раньше был последним (`"Greg"`) в этом списке. Затем мы меняем у узла `"Greg"` ссылку на следующий узел, чтобы она указывала на новый (`"Sue"`). И затем объявляем новый `"Sue"` последним узлом двусвязного списка.

Программная реализация: добавление элемента в двусвязный список

Это реализация нового метода `insert_at_end`, который мы можем добавить в наш класс `DoublyLinkedList`:

```
def insert_at_end(value)
  new_node = Node.new(value)

  # Если в двусвязном списке еще нет элементов:
  if !first_node
    @first_node = new_node
    @last_node = new_node
  else # Если в двусвязном списке уже есть хотя бы один узел:
    new_node.previous_node = @last_node
    @last_node.next_node = new_node
    @last_node = new_node
  end
end
```

Рассмотрим самые важные фрагменты этого метода.

Сначала мы создаем новый узел:

```
new_node = Node.new(value)
```

Затем устанавливаем для него ссылку на предыдущий (`previous_node`), чтобы она указывала на узел, который до этого был последним:

```
new_node.previous_node = @last_node
```

Теперь меняем ссылку последнего узла (значение которой до этого было `nil`), чтобы она указывала на новый узел (`new_node`):

```
@last_node.next_node = new_node
```

Наконец, сообщаем экземпляру `DoublyLinkedList`, что теперь его последний узел — новый узел:

```
@last_node = new_node
```

Движение вперед и назад

В «классическом» связном списке можно двигаться только *вперед* — мы можем получить доступ к первому узлу и, следуя по ссылкам, найти остальные. Но мы не можем двигаться назад, так как ни один узел ничего не знает о предыдущем.

Двусвязный список обеспечивает бóльшую гибкость, позволяя перемещаться по списку *и* вперед, *и* назад: мы можем начать с последнего узла и двигаться в обратном направлении, к первому.

Очереди на основе двусвязных списков

Двусвязные списки предусматривают мгновенный доступ как к первому, так и к последнему узлам, поэтому позволяют вставлять и удалять данные с обоих концов списка за время O(1). Именно поэтому они — *идеальная базовая структура данных для создания очереди.*

Как вы помните из главы 9, очереди — это списки элементов, допускающие вставку значений только в конец, а удаление — только из начала. Вы уже знаете, что очереди служат примером абстрактного типа данных и что их можно реализовать на основе массива.

Но массив — не самая эффективная структура данных для создания очереди. Несмотря на то что он позволяет выполнять вставку в конец за время O(1), удаление элементов в начале занимает O(N).

Двусвязный же список позволяет вставлять значения в конец *и* удалять их из начала за время O(1). Именно поэтому он отлично подходит для создания очереди.

Программная реализация

Вот пример очереди на основе двусвязного списка:

```
class Node

  attr_accessor :data, :next_node, :previous_node

  def initialize(data)
    @data = data
  end

end

class DoublyLinkedList

  attr_accessor :first_node, :last_node

  def initialize(first_node=nil, last_node=nil)
    @first_node = first_node
    @last_node = last_node
  end

  def insert_at_end(value)
    new_node = Node.new(value)

    # Если в двусвязном списке еще нет элементов:
    if !first_node
      @first_node = new_node
      @last_node = new_node
    else # Если в двусвязном списке уже есть хотя бы один узел:
      new_node.previous_node = @last_node
      @last_node.next_node = new_node
      @last_node = new_node
    end
  end

  def remove_from_front
    removed_node = @first_node
    @first_node = @first_node.next_node
    return removed_node
  end

end

class Queue
  attr_accessor :queue

  def initialize
    @data = DoublyLinkedList.new
  end

  def enqueue(element)
    @data.insert_at_end(element)
  end
```

```
    def dequeue
      removed_node = @data.remove_from_front
      return removed_node.data
    end

    def read
      return nil unless @data.first_node
      return @data.first_node.data
    end
  end
```

Чтобы все это работало, мы добавили метод `remove_from_front` в класс `DoublyLinkedList`, который выглядит так:

```
def remove_from_front
  removed_node = @first_node
  @first_node = @first_node.next_node
  return removed_node
end
```

Как видите, мы удаляем первый узел, задавая в качестве нового первого узла списка (`@first_node`) тот, который сейчас второй, а затем возвращаем удаленный узел.

Класс `Queue` реализует свои методы поверх нашего класса `DoublyLinkedList`. Метод постановки в очередь (`enqueue`) опирается на метод `insert_at_end` класса `DoublyLinkedList`:

```
def enqueue(element)
  @data.insert_at_end(element)
end
```

Аналогично, метод удаления из очереди (`dequeue`) использует возможность связного списка удалять элементы из начала:

```
def dequeue
  removed_node = @data.remove_from_front
  return removed_node.data
end
```

Реализация очереди на основе двусвязного списка позволяет вставлять и удалять элементы за время O(1), что здорово вдвойне!

Выводы

Как вы убедились, небольшие различия между массивами и связными списками открывают новые возможности для ускорения выполнения кода.

В процессе изучения связных списков вы познакомились и с концепцией узлов. Но связный список — это лишь простейшая из структур. В следующих главах

вы узнаете о более сложных и интересных структурах данных на основе узлов и раскроете для себя весь их потенциал.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 14».

1. Добавьте в классический `LinkedList` метод, который выводит все элементы списка.
2. Добавьте в `DoublyLinkedList` метод, который выводит все элементы списка *в обратном порядке*.
3. Добавьте в классический `LinkedList` метод, который возвращает последний элемент списка, исходя из того, что общее число элементов в списке вам неизвестно.
4. Это непростая задача. Добавьте в классический `LinkedList` метод, который разворачивает список. То есть если исходный список — это A → B → C, то в итоге все ссылки должны измениться так, чтобы получился список C → B → A.
5. А вот хорошая головоломка. Допустим, у вас есть доступ к узлу, который находится примерно в середине классического связного списка, но не к самому связному списку. То есть у вас есть переменная, указывающая на экземпляр `Node`, но нет доступа к экземпляру `LinkedList`. Переходя по ссылкам, вы сможете найти все элементы этого узла до конца списка, но у вас нет никакой возможности найти предшествующие узлы.

   Напишите код, который удаляет из списка лишь этот средний узел, не затрагивая остальные элементы.

ГЛАВА 15

Тотальное ускорение с помощью двоичных деревьев поиска

Бывают случаи, когда нужно расположить данные в определенном порядке. Например, нам может понадобиться алфавитный список каких-то названий или список продуктов, упорядоченных по цене по возрастанию.

Для упорядочивания данных подойдет алгоритм сортировки вроде Quicksort, но за это придется платить, ведь даже самый быстрый из них будет выполняться за время $O(N \log N)$. Получается, если нам приходится *часто* сортировать данные, то для экономии времени разумнее будет хранить их уже в отсортированном виде.

Упорядоченный массив — простой, но эффективный инструмент для хранения таких данных. Он позволяет быстро выполнять определенные операции, например чтение за время $O(1)$, а поиск — за $O(\log N)$ (с применением алгоритма двоичного поиска).

Но у упорядоченных массивов есть недостаток.

Когда дело доходит до вставки и удаления элементов, упорядоченные массивы работают медленно. Перед вставкой значения мы сначала должны сдвинуть все превышающие его числа на одну ячейку вправо, а при его удалении — влево. В худшем случае (при вставке или удалении значения из первой ячейки) на это уйдет N шагов, а в среднем — $N/2$. В любом случае, эта операция выполняется за время $O(N)$, что довольно медленно для простой вставки или удаления значения.

Если говорить о структуре данных с отличной скоростью при выполнении всех операций, то это будет хеш-таблица. Она позволяет выполнять поиск, вставку и удаление значений за время $O(1)$. Но упорядочить данные в этих таблицах

нельзя, а значит, они не подходят нашему приложению для работы с алфавитным списком.

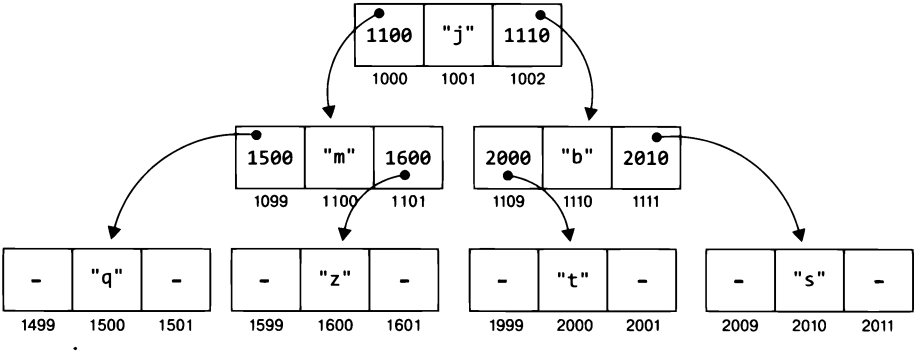
Что же делать, если нам нужна структура данных, допускающая упорядочивание значений и обеспечивающая их быстрый поиск, вставку и удаление?

Встречайте двоичное дерево поиска.

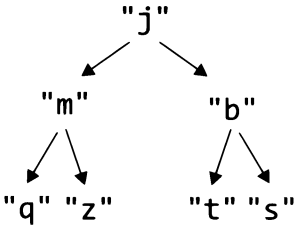
Деревья

Ранее вы познакомились со структурами данных на основе узлов, к которым относится связный список. В классическом связном списке каждый узел содержит ссылку, которая связывает его с каким-то одним узлом. *Дерево* — тоже структура данных на основе узлов, но у каждого узла в ней могут быть ссылки на *несколько* других узлов.

Примерно так будет выглядеть простое дерево:

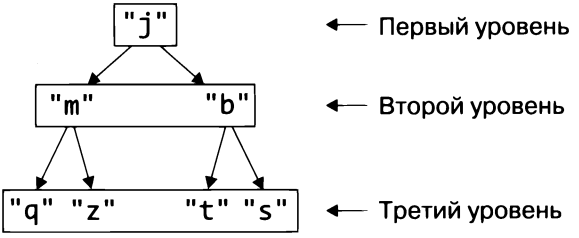


Здесь ссылки каждого узла указывают на два других узла. Давайте визуально упростим это дерево, не показывая все адреса памяти:



У деревьев есть ряд своих уникальных терминов:

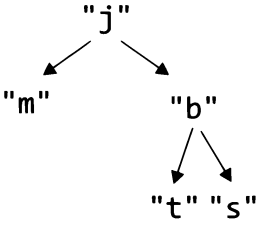
- самый верхний узел (у нас это `"j"`) называется *корнем*. Да, на рисунке корень на *вершине* дерева, но так обычно и изображают эту структуру данных;
- узел `"j"` — это *родитель* для узлов `"m"` и `"b"`, а узлы `"m"` и `"b"` — *дочерние элементы* для `"j"`. Точно так же `"m"` — родительский узел для `"q"` и `"z"`, а `"q"` и `"z"` — дочерние элементы для `"m"`;
- как и в генеалогическом древе, у узла могут быть *потомки* и *предки*. Потомки узла — это *все* следующие за ним узлы, а предки — *все* предшествующие ему. В примере `"j"` — предок всех остальных узлов в дереве, а они, в свою очередь, — потомки `"j"`;
- у деревьев есть *уровни*, где располагаются узлы. У дерева из нашего примера три уровня:



- одно из основных свойств дерева — его *сбалансированность: одинаковое количество узлов в поддеревьях*.

Например, дерево выше идеально сбалансировано. Если вы посмотрите на каждый узел, то увидите, что два его поддерева содержат одинаковое число узлов. У корневого узла (`"j"`) два поддерева, в каждом из которых по три узла. То же можно сказать о каждом узле этого дерева. Например, у `"m"` тоже два поддерева, в каждом из которых по одному узлу.

Дерево ниже *несбалансированное:*



Как видите, в правом поддереве корня больше узлов, чем в левом, что и обусловливает дисбаланс.

Двоичные деревья поиска

Есть много разных типов древовидных структур данных, но здесь мы сосредоточимся на *двоичном*, или *бинарном*, *дереве поиска*.

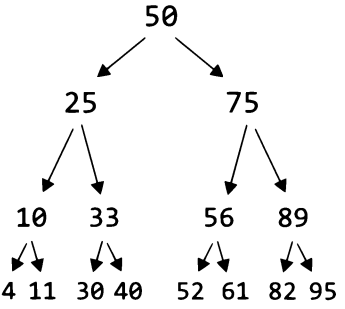
Обратите внимание на слова «двоичный» и «поиск».

В каждом узле *двоичного* дерева может быть не более двух дочерних элементов — только 0, 1 или 2 элемента.

Двоичное дерево *поиска* подчиняется следующим правилам:

- в каждом узле не может быть больше одного «левого» и одного «правого» дочернего элемента;
- «левые» потомки узла могут содержать только меньшие, чем сам узел, значения. Точно так же «правые» могут содержать только значения, превышающие значение самого узла.

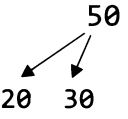
Вот пример двоичного дерева поиска с числовыми значениями:



Обратите внимание, что левый дочерний элемент каждого узла содержит меньшее значение, чем он сам, а правый — большее.

Еще заметьте, что значения всех левых потомков узла 50 меньше него, а все правые — больше. Эта закономерность прослеживается во всех узлах.

Ниже представлен пример двоичного дерева, которое не является двоичным деревом *поиска:*



Это дерево двоичное, потому что в каждом его узле 0, 1 или 2 дочерних элемента. Но это не двоичное дерево *поиска*, потому что у корневого узла два «левых» дочерних элемента — два дочерних элемента, значения которых меньше значения самого корневого узла. Чтобы бинарное дерево поиска было корректным, у него должно быть не более одного левого (меньшего) и одного правого (большего) дочернего элемента.

Реализация узла дерева на языке Python выглядит примерно так:

```
class TreeNode:
    def __init__(self,val,left=None,right=None):
        self.value = val
        self.leftChild = left
        self.rightChild = right
```

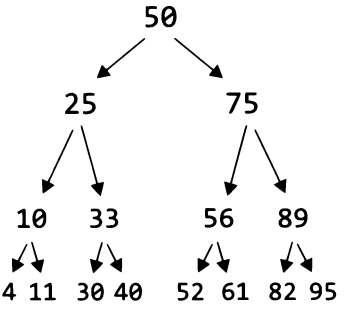
Теперь мы можем построить простое дерево:

```
node1 = TreeNode(25)
node2 = TreeNode(75)
root = TreeNode(50, node1, node2)
```

Как будет показано далее, благодаря уникальной структуре двоичного дерева поиска мы можем очень быстро найти в нем любое значение.

Поиск

Итак, у нас есть двоичное дерево поиска:



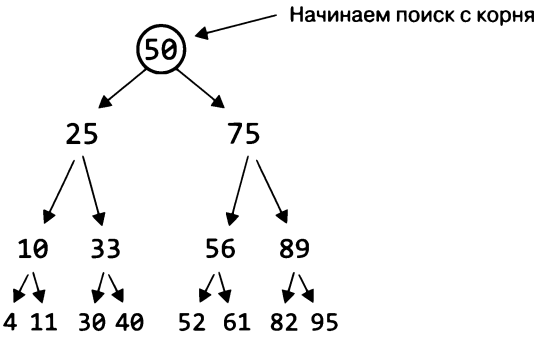
Алгоритм поиска в двоичном дереве состоит из следующих этапов.

1. Назначаем узел «текущим» (первый «текущий узел» — корень дерева).
2. Проверяем значение в текущем узле.

3. Если мы нашли то, что искали, отлично!
4. Если искомое значение меньше значения текущего узла, ищем его в левом поддереве.
5. Если больше — в правом.
6. Повторяем шаги с 1 по 5, пока не найдем искомое значение или не достигнем последнего узла дерева, — тогда делаем вывод, что в дереве искомого значения нет.

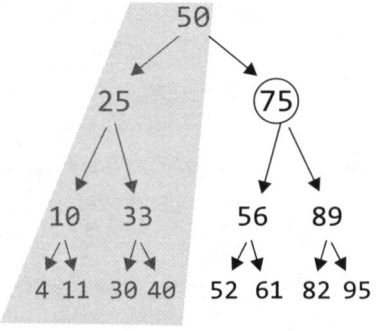
Допустим, мы хотим найти значение 61. Посчитаем, сколько шагов для этого нужно.

Поиск по дереву всегда начинается с корня:

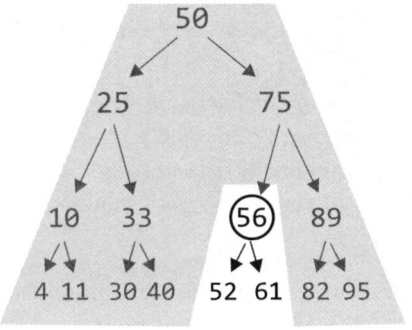


Затем проверяем, превышает ли искомое число (61) значение этого узла или наоборот. Если нет, ищем его в левом дочернем узле, если да — в правом.

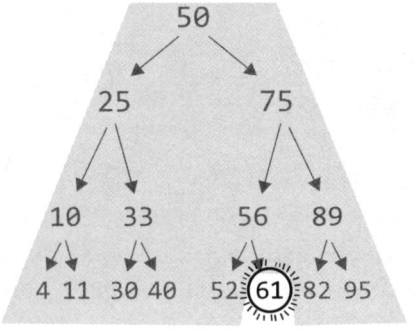
Поскольку 61 больше 50, мы знаем, что оно где-то справа, поэтому проверяем правый дочерний элемент. На следующей схеме серым цветом выделены все узлы, которые мы исключили из процесса поиска:



Поскольку 75 — не то число, которое мы ищем (61), переходим на следующий уровень. А так как 61 меньше 75, мы проверяем левый дочерний элемент, ведь 61 может быть только в левом поддереве.



Число 56 тоже не то, что мы ищем (61), поэтому продолжаем поиск и проверяем значение правого дочернего элемента:



Мы нашли то, что искали! На обнаружение искомого значения у нас ушло 4 шага.

Эффективность поиска в двоичном дереве поиска

Если вы еще раз просмотрите шаги выше, то заметите, что каждый исключает из процесса поиска половину оставшихся узлов. Например, мы начинаем поиск с корневого узла, зная, что искомое значение может быть в любом из его потомков. Но, продолжив поиск, скажем, с правого дочернего элемента, мы исключаем из поиска *всех левых потомков*.

Выходит, поиск в двоичном дереве выполняется за время $O(\log N)$, что подходит любому алгоритму, исключающему половину оставшихся значений на каждом шаге. Но скоро мы увидим, что это работает только с идеально сбалансированным двоичным деревом поиска — с лучшим сценарием.

Log(N) уровней

Вот еще одна причина того, почему временная сложность поиска в двоичном дереве поиска — O(log *N*). Дело здесь еще в одном свойстве двоичных деревьев: *если в сбалансированном двоичном дереве N узлов, то в нем будет около log N уровней* (*рядов*).

Давайте разберемся. Допустим, каждый уровень дерева полностью заполнен узлами, то есть в рядах узлов нет пробелов. Если подумать, то каждый раз, когда мы заполняем очередной уровень, мы примерно удваиваем число узлов, которые уже есть в дереве (если точнее, то мы удваиваем число узлов и добавляем еще один).

Например, двоичное дерево с четырьмя заполненными уровнями содержит 15 узлов (можете посчитать). При заполнении пятого уровня мы добавляем по два дочерних элемента каждому из восьми узлов четвертого. Получается, мы добавляем 16 новых узлов, увеличивая размер дерева примерно в два раза.

Выходит, что добавление каждого нового уровня приводит к удвоению размера дерева. Соответственно, у *дерева с N узлов должно быть log N уровней*.

При обсуждении бинарного поиска мы отметили, что модель log(*N*) такова, что на каждом шагу мы можем исключить половину оставшихся значений. Количество уровней двоичного дерева тоже подпадает под эту закономерность.

Возьмем, к примеру, двоичное дерево, которое должно содержать 31 узел. Пятый уровень вмещает 16 таких узлов. Итак, мы позаботилось примерно о половине данных, но нам еще нужно найти место для оставшихся узлов. На четвертом уровне мы можем разместить восемь из них, в результате у нас останется семь. На третьем мы размещаем четыре и т. д.

Действительно, log 31 равен (приблизительно) 5. Выходит, что у сбалансированного дерева из *N* узлов будет log *N* уровней.

Теперь понятно, почему поиск в двоичном дереве требует log *N* шагов: на каждом шаге мы спускаемся на уровень, выполняя столько шагов, сколько уровней в дереве.

Получается, поиск в двоичном дереве поиска занимает O(log *N*) времени.

Временная сложность O(log *N*) свойственна и двоичному поиску в упорядоченном массиве, при котором выбор каждого следующего числа исключает из поиска половину оставшихся значений. В этом отношении поиск в двоичном дереве так же эффективен, как и двоичный поиск в упорядоченном массиве. Но двоичные деревья поиска существенно превосходят упорядоченные массивы при вставке значения, о которой мы поговорим чуть позже.

Программная реализация: поиск значения в двоичном дереве

Для всех операций над двоичным деревом поиска мы будем активно использовать рекурсию. В главе 10 вы узнали, что она незаменима при работе со структурами данных разной глубины. Дерево — именно такая структура, ведь в нем может быть бесконечное количество уровней.

Вот как можно использовать рекурсию для поиска на языке Python:

```
def search(searchValue, node):
    # Базовый случай: узла нет
    # или мы нашли искомое значение:
    if node is None or node.value == searchValue:
        return node

    # Если искомое значение меньше значения текущего узла,
    # проверяем левый дочерний элемент:
    elif searchValue < node.value:
        return search(searchValue, node.leftChild)

    # Если искомое значение больше значения текущего узла,
    # проверяем правый дочерний элемент:
    else: # searchValue > node.value
        return search(searchValue, node.rightChild)
```

Функция поиска `search` принимает искомое значение `searchValue` и узел `node`, с которого мы начнем поиск. При первом вызове `search` этим узлом будет корень дерева. Но в рамках последующих рекурсивных вызовов узлом `node` может стать любой другой узел в дереве.

Наша функция обрабатывает четыре возможных случая, два из которых базовые:

```
if node is None or node.value == searchValue:
    return node
```

Первый базовый случай: узел содержит искомое значение `searchValue`. Здесь мы можем вернуть узел и не выполнять никаких рекурсивных вызовов.

Второй: значение `node` — `None`. Эта ситуация станет понятнее после изучения других случаев, так что вернемся к ней чуть позже.

Третий: искомое значение `searchValue` меньше значения текущего узла:

```
elif searchValue < node.value:
    return search(searchValue, node.leftChild)
```

При этом мы знаем, что если в дереве есть значение `searchValue`, то его можно найти только среди левых потомков узла. Поэтому мы рекурсивно вызываем `search` для левого дочернего элемента этого узла.

Следующий случай — обратный, когда `searchValue` превышает значение текущего узла:

```
else: # searchValue > node.value
    return search(searchValue, node.rightChild)
```

Здесь мы рекурсивно вызываем `search` для правого дочернего элемента узла.

Обратите внимание, что перед выполнением рекурсивных вызовов для дочерних элементов текущего узла мы не проверяем их наличие. Вот где нам пригодится первый базовый случай:

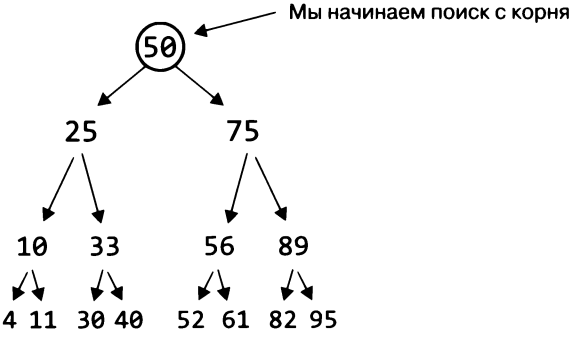
```
if node is None
```

То есть при вызове функции `search` для несуществующего дочернего узла мы возвращаем `None` (так как в узле ничего нет). Так происходит, если искомого значения `searchValue` в дереве нет, поскольку при попытке получить доступ к узлу, который *должен* содержать это значение, мы оказываемся в тупике. Тогда вполне уместно возвратить `None`, указывающее на то, что `searchValue` в дереве нет.

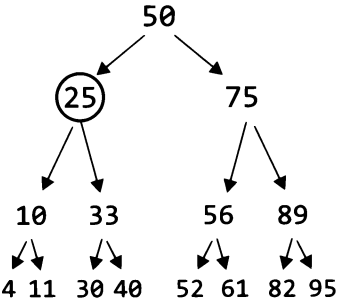
Вставка

Как я уже сказал, настоящая мощь двоичных деревьев поиска проявляется, когда речь заходит о вставке значений. Сейчас мы увидим почему.

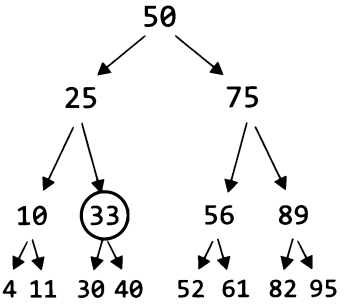
Допустим, мы хотим вставить число 45 в дерево из примера выше. Сначала нам нужно найти подходящий узел. Начинаем поиск с корня:



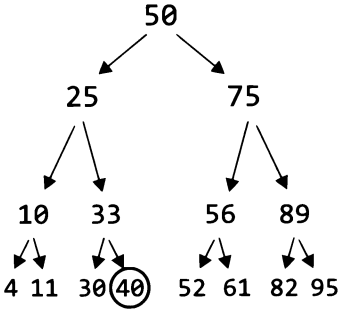
Значение 45 меньше 50, поэтому переходим к левому дочернему элементу:



Число 45 больше 25, поэтому проверяем правый дочерний элемент:

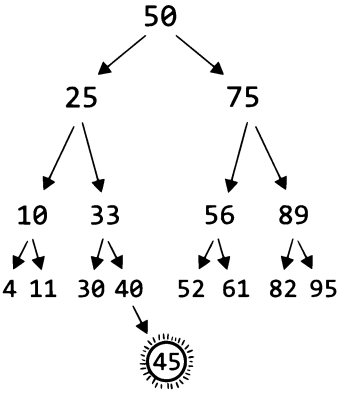


Значение 45 больше 33, поэтому проверяем правый дочерний элемент узла 33:



Мы достигли узла без дочерних элементов, поэтому нам больше некуда идти. Это значит, что мы готовы выполнить вставку значения.

Так как 45 больше 40, вставляем это значение в качестве правого дочернего элемента узла 40:



На вставку в этом примере у нас ушло пять шагов: четыре для поиска подходящего места и один — для вставки значения. На вставку всегда нужен один дополнительный шаг после поиска, что в общей сложности дает (log *N*) + 1 шагов. Как мы помним, O-нотация игнорирует константы, поэтому временная сложность этой операции — O(log *N*).

С другой стороны, вставка значения в упорядоченный массив занимает O(*N*) времени, потому что помимо поиска мы должны сдвинуть много данных вправо, чтобы освободить место для вставляемого значения.

Именно это делает двоичные деревья поиска такими эффективными. В то время как временная сложность поиска в упорядоченных массивах — O(log *N*), а вставки — O(*N*), временная сложность поиска *и* вставки в двоичных деревьях поиска — O(log *N*). Это очень важно для приложения, где часто приходится менять данные.

Программная реализация: вставка значения в двоичное дерево поиска

Вот реализация вставки нового значения в двоичное дерево поиска на языке Python. Эта функция, как и поиск, рекурсивная:

```
def insert(value, node):
    if value < node.value:

        # Если левого дочернего элемента нет,
        # вставляем значение в качестве левого дочернего элемента:
        if node.leftChild is None:
```

```
            node.leftChild = TreeNode(value)
        else:
            insert(value, node.leftChild)
    elif value > node.value:

        # Если правого дочернего элемента нет,
        # вставляем значение в качестве правого дочернего элемента:
        if node.rightChild is None:
            node.rightChild = TreeNode(value)
        else:
            insert(value, node.rightChild)
```

Функция вставки `insert` принимает значение (`value`), которое мы хотим вставить, и узел (`node`), который будет предком для вставляемого узла.

Сначала сравниваем вставляемое значение со значением текущего узла:

```
if value < node.value:
```

Если оно меньше значения узла, то мы понимаем, что оно должно быть вставлено в его левое поддерево.

Теперь проверяем, есть ли у текущего узла левый дочерний элемент. Если нет, делаем вставляемое значение левым дочерним элементом:

```
if node.leftChild is None:
    node.leftChild = TreeNode(value)
```

Это базовый случай, так как нам не нужно выполнять никаких рекурсивных вызовов.

Но если у узла уже есть левый дочерний элемент, мы не можем поместить на его место вставляемое значение. Поэтому мы рекурсивно вызываем функцию `insert` для левого дочернего элемента, чтобы продолжить поиск подходящего для вставки значения места:

```
else:
    insert(value, node.leftChild)
```

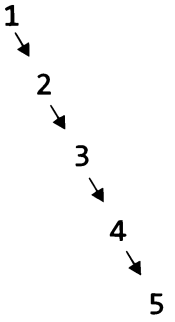
Наконец, мы обнаруживаем дочерний узел без дочернего элемента и делаем вставляемое значение его дочерним узлом.

Остальная часть кода функции обрабатывает случаи, когда вставляемое значение превышает значение текущего узла.

Порядок вставки

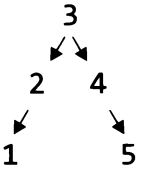
Важно отметить, что деревья хорошо сбалансированы, только когда создаются из данных, перемешанных произвольно. Но если мы вставим в дерево *отсор-*

тированные данные, оно может стать несбалансированным и менее эффективным. Например, если бы мы вставили следующие данные в таком порядке: 1, 2, 3, 4, 5, то получили бы вот такое дерево:



Оно линейное, поэтому поиск значения 5 в нем занимает время O(N).

Но если мы вставим те же данные в следующем порядке: 3, 2, 4, 1, 5, то дерево будет равномерно сбалансированным:



Только в сбалансированном дереве временная сложность поиска будет O(log N).

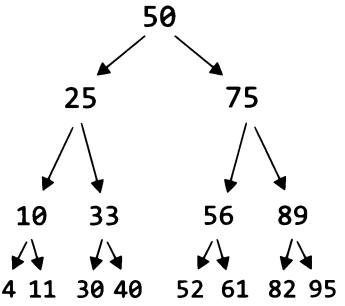
Поэтому если вы когда-нибудь решите преобразовать упорядоченный массив в двоичное дерево поиска, сначала перемешайте данные в случайном порядке.

Получается, что в худшем случае, когда дерево абсолютно *не сбалансировано*, поиск занимает время O(N), а в лучшем, когда дерево идеально сбалансировано, — O(log N). В типичном сценарии, когда данные вставляются в случайном порядке, дерево оказывается вполне сбалансированным, поэтому временная сложность поиска в нем — примерно O(log N).

Удаление

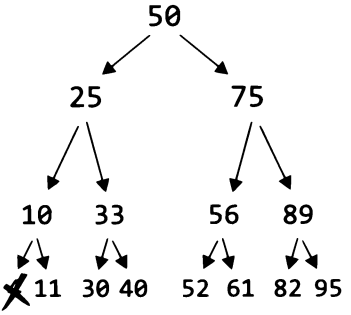
Удаление — это самая непростая операция над двоичным деревом поиска, требующая осторожного маневрирования.

Допустим, мы хотим удалить значение 4 из следующего двоичного дерева поиска:

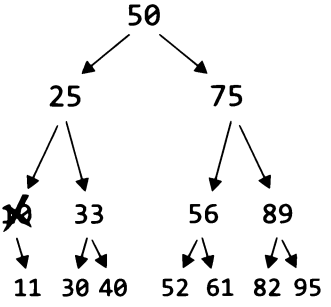


Сначала мы находим значение 4. Пропустим процесс поиска — он уже был описан выше.

Как только мы найдем нужное число (4), то сможем удалить его за один шаг:

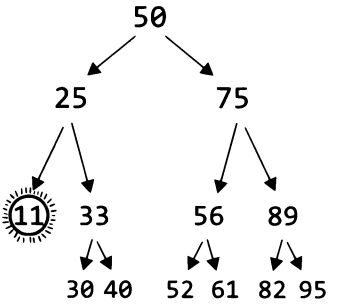


Это было несложно. Но давайте посмотрим, что произойдет при попытке удалить число 10:



В итоге число 11 потеряет связь с деревом. А мы не можем этого допустить, поскольку не хотим навсегда потерять это значение.

Чтобы решить эту проблему, нужно вставить значение 11 на место значения 10:

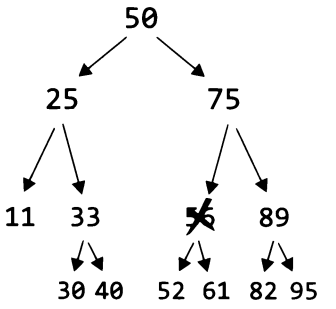


На этом этапе наш алгоритм удаления придерживается следующих правил:

- если у удаляемого узла нет дочерних элементов, просто удаляем его;
- если есть один дочерний узел, удаляем его и помещаем дочерний элемент на место удаленного узла.

Удаление узла с двумя дочерними элементами

Удаление узла с двумя дочерними элементами — самый сложный случай. Допустим, мы хотим удалить 56 из следующего дерева:

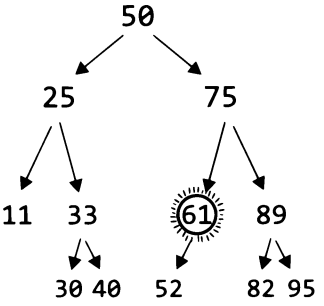


Что будем делать с его дочерними элементами 52 и 61? Мы не можем переместить *оба* элемента на прежнее место значения 56. И здесь в игру вступает следующее правило: при удалении узла с двумя дочерними элементами мы ставим на его место *узел-преемник* — дочерний узел с *наименьшим значением из всех, превышающих значение удаленного узла.*

Это сложно понять, но суть вот в чем: если мы выстроим удаленный узел и всех его потомков в порядке возрастания, то преемником будет тот, что следует сразу за удаляемым узлом.

В нашем примере легко понять, какой узел будет преемником, так как у удаляемого узла только два дочерних элемента. Если мы расположим числа 52-56-61 в порядке возрастания, то следующим числом после 56 будет 61.

Обнаружив преемника, мы помещаем его на место удаленного узла. Итак, мы меняем значение 56 на 61:

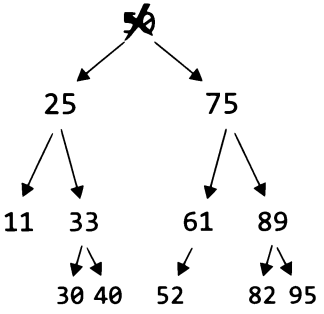


Поиск узла-преемника

Чем ближе к корню дерева удаляемый узел, тем сложнее будет найти узел-преемник.

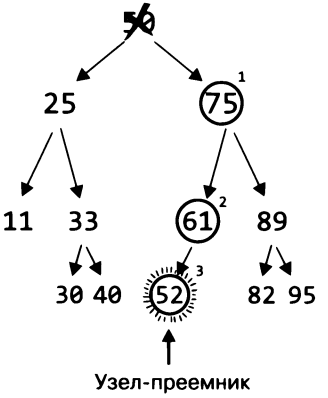
Так выглядит алгоритм поиска узла-преемника с точки зрения компьютера: компьютер посещает правый дочерний элемент удаляемого узла, а затем последовательно все левые дочерние элементы вплоть до нахождения последнего левого потомка, который и становится узлом-преемником.

Рассмотрим этот процесс на более сложном примере с удалением корневого узла:



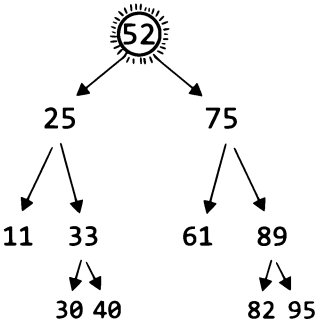
Нам нужно поместить узел-преемник на место удаленного значения 50, сделав его новым корневым узлом. Итак, найдем преемника.

Для этого сначала посещаем *правый* дочерний элемент удаляемого узла, а затем продолжаем спускаться по *левому* поддереву, пока не достигнем узла без левого дочернего элемента:



Итак, наш узел-преемник — значение 52.

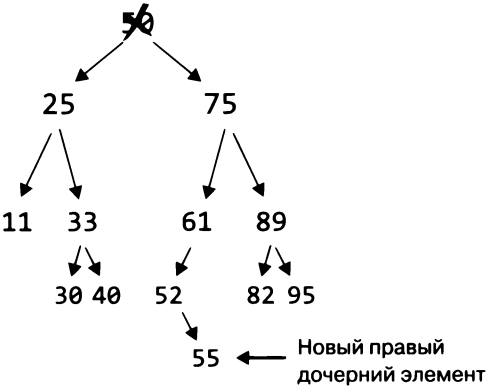
Помещаем его на место удаленного узла:



Мы закончили!

Узел-преемник с правым дочерним элементом

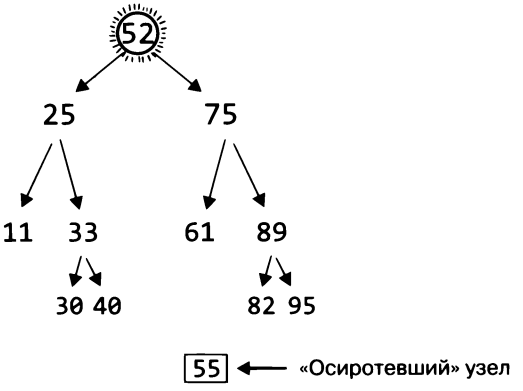
Но есть еще один неучтенный случай: когда у узла-преемника есть правый дочерний элемент. Добавим в наше дерево правый дочерний элемент для узла 52:



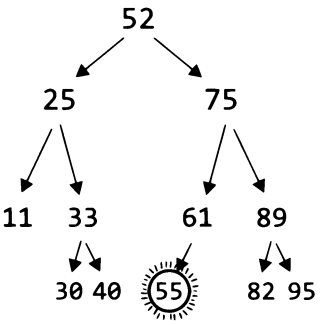
Сейчас мы не можем просто превратить узел-преемник (52) в корень дерева, ведь при этом его дочерний узел (55) останется сам по себе. Для таких случаев в алгоритме удаления есть еще одно правило: если у узла-преемника есть правый дочерний узел, то после перемещения преемника на место удаленного узла нужно переместить бывший правый элемент узла-преемника и сделать его *левым дочерним элементом бывшего родителя узла-преемника*.

Это тоже понять непросто, поэтому рассмотрим весь процесс пошагово.

Сначала мы помещаем узел-преемник (52) на место корня — в результате узел 55 остается без родителя:



Теперь делаем узел 55 левым дочерним элементом бывшего родителя узла-преемника. У нас это был узел 61, поэтому делаем узел 55 его левым дочерним элементом:



Вот теперь мы *действительно* закончили.

Полный алгоритм удаления

После объединения всех шагов алгоритм удаления значения из двоичного дерева поиска выглядит так:

- если у удаляемого узла нет дочерних элементов, просто удаляем его;
- если есть один дочерний узел, удаляем его и помещаем дочерний элемент на место удаленного узла;
- при удалении узла с двумя дочерними элементами мы ставим на его место *узел-преемник* — дочерний узел с *наименьшим значением из всех, превышающих значение удаленного узла;*
- чтобы найти узел-преемник, посещаем правый дочерний элемент удаляемого значения, а затем последовательно все левые дочерние элементы вплоть до нахождения последнего левого потомка, который и становится узлом-преемником;
- если у узла-преемника есть правый дочерний узел, то после перемещения преемника на место удаленного узла нужно переместить бывший правый элемент узла-преемника и сделать его *левым дочерним элементом бывшего родителя узла-преемника.*

Программная реализация: удаление значения из двоичного дерева поиска

Ниже приведена рекурсивная реализация удаления значения из двоичного дерева поиска на языке Python:

```
def delete(valueToDelete, node):

    # Базовый случай: мы достигли последнего узла
    # дерева без дочерних элементов:
    if node is None:
        return None

    # Если удаляемое значение меньше или больше значения текущего узла,
    # задаем левый или правый дочерний элемент соответственно
    # в качестве возвращаемого значения рекурсивного вызова
    # этого метода для левого или правого поддерева текущего узла
    elif valueToDelete < node.value:
        node.leftChild = delete(valueToDelete, node.leftChild)

        # Возвращаем текущий узел (и его поддерево, если оно есть)
        # для использования в качестве нового значения левого или правого
        # дочернего элемента его родителя:
        return node
    elif valueToDelete > node.value:
        node.rightChild = delete(valueToDelete, node.rightChild)
        return node

    # Если текущий узел - тот, который мы хотим удалить:
    elif valueToDelete == node.value:

        # Если у текущего узла нет левого дочернего элемента, удаляем его,
        # возвращая его правый дочерний элемент (и его поддерево, если оно
        # есть) в качестве нового поддерева его родителя:
        if node.leftChild is None:
            return node.rightChild

            # (Если у текущего узла нет левого ИЛИ правого дочернего
            # элемента, возвращаемым значением будет None, в соответствии
            # с первой строкой кода этой функции)

        elif node.rightChild is None:
            return node.leftChild

        # Если у текущего узла есть два дочерних элемента, удаляем его
        # с помощью функции lift (см. ниже), которая изменяет
        # значение текущего узла на значение узла-преемника:
        else:
            node.rightChild = lift(node.rightChild, node)
            return node

def lift(node, nodeToDelete):

    # Если у текущего узла этой функции есть левый дочерний элемент,
    # рекурсивно вызываем эту функцию для спуска
    # по левому поддереву в поисках узла-преемника
    if node.leftChild:
        node.leftChild = lift(node.leftChild, nodeToDelete)
        return node
```

```
        # Если у текущего узла нет левого дочернего элемента, значит, текущий
        # узел этой функции - узел-преемник, и мы помещаем его значение
        # на место удаленного узла:
        else:
            nodeToDelete.value = node.value
            # Возвращаем значение правого потомка узла-преемника, который теперь
            # используется как левый дочерний элемент его родителя:
            return node.rightChild
```

Это сложный код, поэтому разберем его подробно.

Функция удаления принимает два аргумента:

```
def delete(valueToDelete, node):
```

`valueToDelete` — это значение, которое мы удаляем из дерева, а `node` — его корень. При первом вызове этой функции `node` будет фактическим корневым узлом, но, поскольку функция вызывает себя рекурсивно, он может быть ниже в дереве, будучи просто корнем меньшего поддерева. В любом случае `node` — это корень либо всего дерева, либо одного из его поддеревьев.

Базовый случай появляется, когда узла не существует:

```
if node is None:
    return None
```

Так бывает, когда в рамках рекурсивного вызова функции совершается попытка получить доступ к несуществующему дочернему узлу. Тогда мы возвращаем `None`.

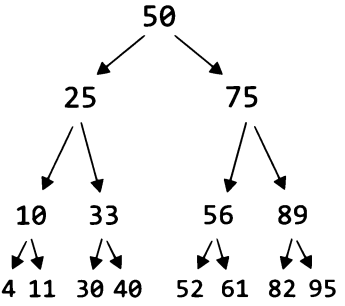
Теперь сравниваем удаляемое значение `valueToDelete` со значением текущего узла `node`:

```
elif valueToDelete < node.value:
    node.leftChild = delete(valueToDelete, node.leftChild)
    return node
elif valueToDelete > node.value:
    node.rightChild = delete(valueToDelete, node.rightChild)
    return node
```

Этот фрагмент кода может показаться непонятным, но работает он так. Если удаляемое значение меньше значения текущего узла, то мы знаем, что данное значение можно найти только среди левых потомков этого узла.

А теперь самая хитрая часть: мы *перезаписываем* значение левого дочернего элемента текущего узла `node`, превращая его в результат рекурсивного вызова функции `delete` для левого дочернего элемента текущего узла. В итоге `delete` возвращает узел, поэтому мы берем результат и делаем его левым дочерним элементом текущего узла.

Но обычно эта «перезапись» не приводит к фактическому изменению левого дочернего элемента, потому что вызов `delete` для него может вернуть его же. Чтобы понять, о чем речь, представьте, что мы пытаемся удалить значение 4 из следующего дерева:



Изначально `node` — корневой узел со значением 50. Поскольку удаляемое значение 4 (`valueToDelete`) меньше 50, левый дочерний элемент узла 50 должен быть результатом вызова функции `delete` для текущего левого дочернего элемента узла 50, то есть для узла 25.

Итак, каким же будет левый дочерний элемент узла 50? Давайте посмотрим.

При рекурсивном вызове функции `delete` для узла 25 мы снова определяем, что удаляемое значение 4 меньше 25 (текущий узел `node`), поэтому рекурсивно вызываем `delete` для левого дочернего элемента узла 25 — для узла 10. Но каким бы ни было значение левого дочернего элемента узла 25, `return node` свидетельствует о том, что в конце текущего вызова функции мы *возвращаем узел 25*.

Я уже говорил, что левый дочерний элемент узла 50 должен быть результатом вызова функции `delete` для узла 25. Но в итоге мы получили сам узел 25, так что «удаленный» элемент фактически не изменился.

Но левый или правый дочерний элемент текущего узла `node` *изменится*, если результатом очередного рекурсивного вызова станет фактическое удаление значения.

Рассмотрим следующий фрагмент:

```
elif valueToDelete == node.value:
```

Это значит, что `node` — тот узел, который мы хотим удалить. Чтобы правильно это сделать, сначала определим, есть ли у него дочерние элементы, так как это повлияет на алгоритм удаления.

Сначала проверяем, есть ли у удаляемого узла левые дочерние элементы:

```
if node.leftChild is None:
    return node.rightChild
```

Если их нет, можем вернуть его *правый* дочерний элемент в качестве результата вызова этой функции. Запомните, *какой бы узел мы ни возвратили, он станет либо левым, либо правым дочерним элементом предыдущего узла в стеке вызовов.* Итак, представим, что у текущего узла есть правый дочерний элемент. Тогда, возвращая правый дочерний узел, мы делаем его дочерним узлом *предыдущего* узла в стеке вызовов, удаляя текущий узел из дерева.

Если у текущего узла нет правого дочернего элемента, ничего страшного. В этом случае в качестве результата вызова функции мы передадим `None`, что тоже приведет к удалению из дерева текущего узла.

Далее, если у текущего узла есть левый дочерний элемент, но нет правого, мы все равно можем легко удалить текущий узел:

```
elif node.rightChild is None:
    return node.leftChild
```

Здесь мы удаляем текущий узел, возвращая его левый дочерний элемент, который становится дочерним узлом предыдущего узла в стеке вызовов.

Наконец, мы сталкиваемся с самым сложным случаем: *два* дочерних элемента у удаляемого узла:

```
else:
    node.rightChild = lift(node.rightChild, node)
    return node
```

Вызываем функцию `lift` и делаем возвращенный ею узел правым дочерним элементом текущего узла.

Что делает `lift`?

При вызове мы передаем ей правый дочерний элемент текущего узла в дополнение к самому узлу. Функция `lift` выполняет четыре задачи.

1. Находит преемника.
2. Перезаписывает значение удаляемого узла `nodeToDelete`, заменяя его значением узла-преемника (так узел-преемник оказывается в нужном месте). Обратите внимание, что мы не перемещаем фактический *объект* узла-преемника, а просто копируем его значение в узел, который «удаляем».
3. Чтобы ликвидировать исходный объект узла-преемника, функция превращает его правый дочерний элемент в левый дочерний элемент его родителя.

4. После завершения всех рекурсивных вызовов функция возвращает либо исходный правый дочерний элемент `rightChild`, переданный ей в самом начале, либо `None`, если исходный элемент `rightChild` стал узлом-преемником (что могло случиться при отсутствии у него левых дочерних элементов).

Теперь делаем значение, возвращенное функцией `lift`, правым дочерним элементом текущего узла. В результате правый дочерний элемент либо остается неизменным, либо меняется на `None`, если он был использован в качестве узла-преемника.

Не переживайте, если чего-то не поняли: функция `delete` — один из сложнейших фрагментов кода в этой книге. Даже после этого пошагового разбора вам может потребоваться тщательное изучение кода, чтобы вникнуть во все этапы процесса.

Эффективность удаления значения из двоичного дерева поиска

Подобно поиску и вставке, удаление значения из дерева тоже происходит за $O(\log N)$ времени. Это связано с тем, что данная операция предполагает выполнение поиска и нескольких дополнительных шагов для обработки «осиротевших» узлов. Сравните этот процесс с удалением значения из упорядоченного массива, которое занимает время $O(N)$ из-за необходимости сдвигать элементы влево, чтобы закрыть пробел из-за удаления значения.

Двоичные деревья поиска в действии

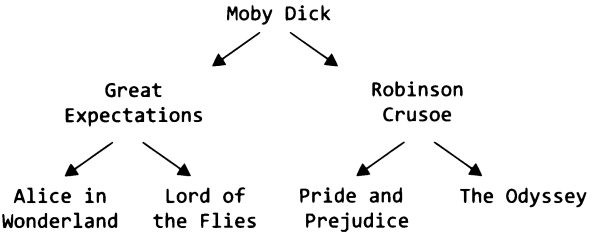
Как мы знаем, двоичные деревья поиска позволяют выполнять поиск, вставку и удаление значений за время $O(\log N)$, что делает их эффективной структурой данных для хранения упорядоченных фрагментов информации и управления ими. Эта структура будет особенно полезна, если мы собираемся часто изменять эти данные, ведь, несмотря на то что упорядоченные массивы не уступают двоичным деревьям при поиске, они сильно проигрывают им, когда дело доходит до вставки и удаления значений.

Допустим, мы создаем приложение, которое ведет список названий книг, и хотели бы, чтобы оно предусматривало следующие функции:

- вывод на экран списка названий книг в алфавитном порядке;
- возможность внесения изменений в список;
- возможность поиска названия книги в списке.

Если бы мы не собирались часто вносить изменения в свой список книг, то вполне могли бы выбрать упорядоченный массив. Но мы создаем приложение, в котором данные меняются часто и в режиме реального времени. Если бы в списке были миллионы названий книг, то двоичное дерево поиска было бы более приемлемым вариантом.

Дерево могло бы выглядеть так:



Здесь названия книг расположены в алфавитном порядке: то, которое начинается с буквы, встречающейся раньше в алфавите, считается «меньшим» значением, а с буквы, встречающейся позже, — «бóльшим».

Обход двоичного дерева поиска

Итак, вы уже знаете, как искать, вставлять и удалять данные из двоичного дерева поиска. Но мы хотим иметь возможность выводить на экран весь список названий книг в алфавитном порядке. Как же это сделать?

Во-первых, мы должны иметь возможность *посещать* каждый узел дерева. Под *посещением* подразумевается получение доступа к узлам. Процесс посещения всех узлов структуры данных называется ее *обходом*.

Во-вторых, нам нужно убедиться, что мы обходим дерево в порядке возрастания значений, чтобы вывести список названий книг в алфавитном порядке. Есть несколько способов обхода дерева, но для вывода значений книг по алфавиту в нашем приложении будет использоваться *центрированный* (*симметричный*) *обход*.

Рекурсия — отличный инструмент для выполнения обхода структуры данных. Мы создадим рекурсивную функцию `traverse`, которую можно будет вызывать для конкретного узла, после чего она будет делать следующее.

1. Рекурсивно вызывать саму себя для левого дочернего элемента узла вплоть до обнаружения узла без левого дочернего элемента.

2. «Посещать» узел (в нашем приложении этот шаг сопровождается выводом значения узла (названия книги)).
3. Рекурсивно вызывать саму себя для правого дочернего элемента узла вплоть до обнаружения узла без правого дочернего элемента.

Базовый случай для этого алгоритма — вызов функции `traverse` для несуществующего дочернего элемента. Здесь мы просто завершаем вызов функции, ничего не делая.

Ниже представлена функция `traverse_and_print` на Python, которая обрабатывает список названий книг. Обратите внимание на лаконичность ее кода:

```
def traverse_and_print(node):
    if node is None:
        return
    traverse_and_print(node.leftChild)
    print(node.value)
    traverse_and_print(node.rightChild)
```

Рассмотрим процесс центрированного обхода поэтапно.

Сначала мы вызываем функцию `traverse_and_print` для узла *Moby Dick*, что приводит к вызову функции `traverse_and_print` для его левого дочернего элемента *Great Expectations:*

```
traverse_and_print(node.leftChild)
```

Но прежде чем перейти к нему, мы добавляем в стек вызовов информацию о том, что находимся внутри вызова функции для узла *Moby Dick* и посещаем его левый дочерний элемент:



После этого вызывается `traverse_and_print` для левого дочернего элемента узла *Great Expectations — Alice in Wonderland.*

Прежде чем двигаться дальше, добавляем соответствующую информацию в стек вызовов:



Далее функция `traverse_and_print` запускается для левого дочернего элемента узла *Alice in Wonderland*, которого *нет* (базовый случай), поэтому ничего не происходит. Следующая строка кода функции `traverse_and_print`:

```
print(node.value)
```

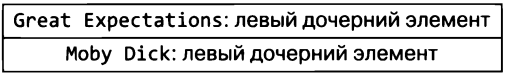
выводит на экран значение узла `"Alice in Wonderland"`.

Затем функция пытается посетить *правый* дочерний элемент узла *Alice in Wonderland:*

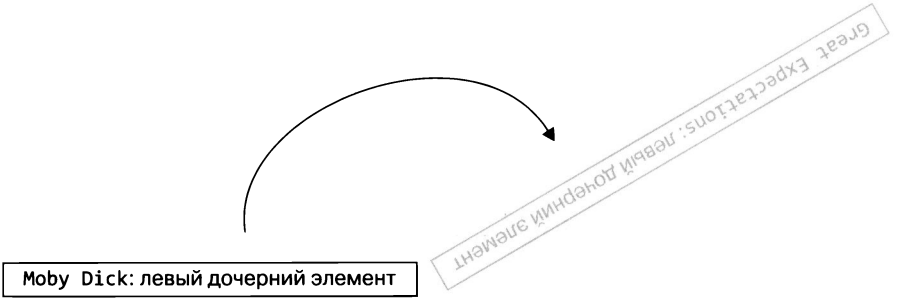
```
traverse_and_print(node.rightChild)
```

Но его тоже нет (базовый случай), поэтому функция просто завершает работу.

После завершения вызова `traverse_and_print("Alice in Wonderland")` мы проверяем стек вызовов, чтобы выяснить, на каком этапе находимся:



Все в порядке! Мы сейчас внутри вызова `traverse_and_print("Great Expectations")` и только что завершили вызов функции `traverse_and_print` для его левого дочернего элемента. Вытолкнем соответствующий фрагмент данных из стека вызовов:

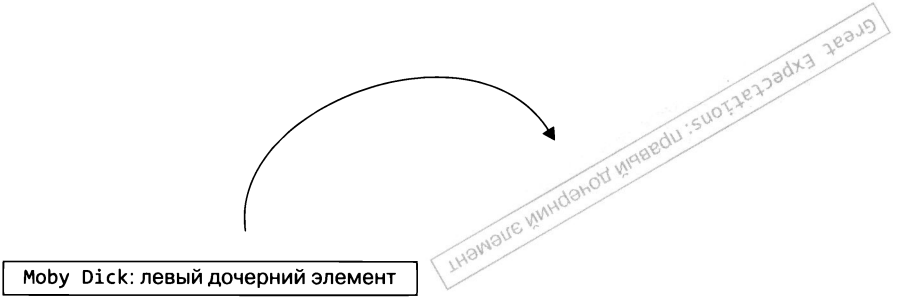


И продолжим. На этом этапе функция выводит на экран значение узла `"Great Expectations"` и вызывает саму себя для посещения его правого дочернего элемента — узла *Lord of the Flies*. Прежде чем перейти к нему, мы фиксируем важные сведения в стеке вызовов:

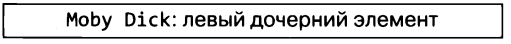


Теперь переходим к выполнению `traverse_and_print("Lord of the Flies")`. Сначала мы вызываем функцию `traverse_and_print` для левого дочернего элемента, которого нет, а затем выводим значение узла *Lord of the Flies*. Наконец, мы вызываем функцию `traverse_and_print` для его правого дочернего элемента, но его тоже не существует, поэтому функция завершает работу.

Проверяя стек вызовов, видим, что находимся внутри вызова функции `traverse_and_print` для правого дочернего элемента узла *Great Expectations*. Выталкиваем соответствующие данные из стека:



Итак, мы полностью разобрались с вызовом `traverse_and_print("Great Expectations")`, поэтому можем вернуться к стеку вызовов, чтобы определиться с дальнейшими действиями:



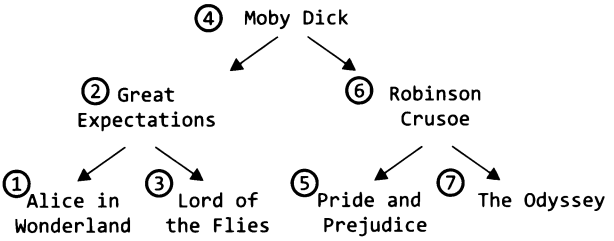
Мы видим, что находимся внутри вызова функции `traverse_and_print` для левого дочернего элемента узла *Moby Dick*. Мы можем вытолкнуть его из стека вызовов (оставив стек пустым на некоторое время) и перейти к следующему шагу вызова `traverse_and_print("Moby Dick")` — выводу значения узла *Moby Dick*.

Затем вызываем функцию `traverse_and_print` для правого дочернего элемента узла *Moby Dick* и добавляем соответствующую информацию в стек вызовов:



Для краткости (хотя, возможно, для этого уже слишком поздно) пройдемся по остальной части функции `traverse_and_print` отсюда.

К моменту завершения ее работы значения узлов будут выведены на экран в следующем порядке:



Именно так мы можем достичь цели и отобразить на экране названия книг, расположенных по алфавиту. Обратите внимание, что обход дерева занимает O(N) времени, поскольку по определению предполагает посещение всех N узлов дерева.

Выводы

Двоичное дерево поиска — это мощная структура данных на основе узлов, которая позволяет работать с упорядоченными данными и довольно быстро выполнять их поиск, вставку и удаление. Оно сложнее, чем связный список, но работает гораздо эффективнее.

Помимо двоичного дерева поиска есть много других типов деревьев, каждый из которых дает уникальные преимущества в разных ситуациях. В следующей главе вы познакомитесь с еще одной древовидной структурой данных, которая позволяет ускорить работу кода в одной весьма распространенной ситуации.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 15».

1. Представьте, что вам нужно вставить в пустое двоичное дерево поиска последовательность чисел в таком порядке: `[1, 5, 9, 2, 4, 10, 6, 3, 8]`.

   Нарисуйте диаграмму, изобразите на ней, как это дерево будет выглядеть в итоге. Помните, что числа должны вставляться в указанном порядке.

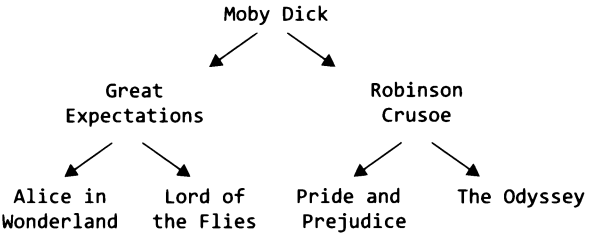
2. Если хорошо сбалансированное двоичное дерево поиска содержит 1000 значений, сколько максимум шагов нужно для поиска значения внутри него?

3. Напишите алгоритм, который находит наибольшее значение в двоичном дереве поиска.

4. В этой главе я показал способ *центрированного* обхода дерева для вывода списка названий книг. Есть и другой способ: *прямой обход*. Вот соответствующий код для нашего приложения:

```
def traverse_and_print(node):
    if node is None:
        return
    print(node.value)
    traverse_and_print(node.leftChild)
    traverse_and_print(node.rightChild)
```

На примере дерева из этой главы (см. следующую диаграмму) определите порядок вывода на экран названий книг при использовании прямого обхода.



5. Есть и еще один способ обхода дерева — *обратный обход*. Вот соответствующий код для нашего приложения:

```
def traverse_and_print(node):
    if node is None:
        return
    traverse_and_print(node.leftChild)
    traverse_and_print(node.rightChild)
    print(node.value)
```

На примере дерева из этой главы (которое используется и в предыдущем упражнении) определите порядок вывода на экран названий книг при использовании обратного обхода.

ГЛАВА 16

Расстановка приоритетов с помощью куч

Познакомившись с деревом, вы открыли для себя много новых структур данных. В прошлой главе мы рассмотрели только двоичное дерево поиска, но есть и много других типов деревьев. У них, как и у других структур данных, есть свои преимущества и недостатки. Главное — знать, какую использовать в конкретной ситуации.

В этой главе мы рассмотрим еще одну древовидную структуру данных — кучу. В некоторых ситуациях она бывает особенно полезна, например когда нужно отслеживать наибольший или наименьший элемент в наборе данных.

Чтобы понять, на что способна куча, сначала рассмотрим другую структуру данных — приоритетная очередь.

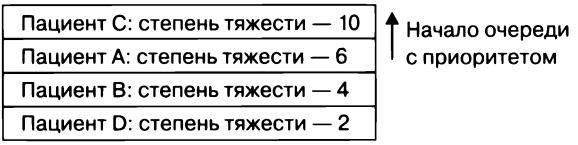
Приоритетные очереди

Как было сказано в главе 9, очередь — это список, где элементы обрабатываются по принципу FIFO. Это значит, что данные могут быть вставлены только в *конец* очереди, а считаны и удалены — только из ее *начала*. При использовании очереди мы получаем доступ к данным по порядку, в котором они были в нее вставлены.

Приоритетная очередь — это список, данные из которого удаляются и считываются так же, как из классической очереди, а вставляются так же, как в упорядоченный массив. То есть мы удаляем и получаем доступ к данным только из *начала* приоритетной очереди, но при их вставке всегда следим за тем, чтобы они оставались отсортированными в определенном порядке.

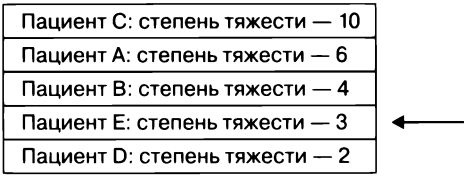
Один из классических примеров использования такой очереди — приложение для сортировки пациентов в отделении неотложной помощи. В больнице людей принимают не в порядке обращения, а в зависимости от степени тяжести их состояния. Если в отделение неотложной помощи будет доставлен пациент с опасной для жизни травмой, он будет помещен в начало очереди, даже если человек с гриппом прибыл на несколько часов раньше.

Допустим, наша система сортировки ранжирует степень тяжести состояния пациентов по шкале от 1 до 10, где 10 — это критическое состояние. В этом случае приоритетная очередь будет выглядеть так:



Мы всегда будем выбирать пациента из начала приоритетной очереди, поскольку он нуждается в оказании помощи больше всего. Сейчас наш приоритетный пациент — Пациент C.

Если теперь в больницу поступит новый пациент, Пациент E, степень тяжести состояния которого оценивается на 3, мы поместим его в соответствующее место в очереди:



Приоритетная очередь — пример абстрактного типа данных. Ее можно реализовать с помощью других, более базовых структур данных. Один из простейших способов реализации такой очереди — использование упорядоченного массива. Для этого мы применяем к массиву следующие ограничения:

- при вставке данных должен поддерживаться определенный порядок;
- данные могут быть удалены только с конца массива (который будет соответствовать началу приоритетной очереди).

Проанализируем эффективность этого простого подхода.

Приоритетная очередь предусматривает две основные операции: удаление и вставку данных.

Как вы помните из главы 1, удаление значения из начала массива выполняется за O(N) времени, поскольку нам приходится сдвигать все данные, чтобы заполнить пробел в позиции с индексом 0. Но в нашей реализации *конец* массива считается *началом* приоритетной очереди, что позволяет удалять значения с конца массива за время O(1).

Итак, мы можем удалять данные за один шаг. Это довольно неплохо, но что делать со вставкой значений?

Как вы помните, вставка значений в упорядоченный массив занимает O(N) времени, так как нам нужно проверить до N элементов массива, чтобы определить подходящее место для новых данных (и даже если отыскать правильное место получится быстро, придется сдвинуть все оставшиеся данные вправо).

Получается, наша приоритетная очередь на основе массива позволяет удалять значения за время O(1), а вставлять — за O(N). Если в нашей очереди будет много элементов, то вставка значений, выполняемая за время O(N), может замедлить работу приложения.

По этой причине программисты разработали другую структуру данных, которая служит более эффективной основой для приоритетной очереди, — кучу.

Кучи

Есть несколько типов куч, но мы сосредоточимся на рассмотрении *двоичной*.

Двоичная куча — это особый вид двоичного дерева. Как вы помните, двоичное дерево — это дерево, где у каждого узла не более двух дочерних элементов (одно из таких деревьев — двоичное дерево *поиска* из прошлой главы).

Двоичные кучи бывают двух видов: max-куча и min-куча. Мы будем рассматривать вторую, но позже вы увидите, что особой разницы между ними нет.

Далее я буду называть двоичную max-кучу просто кучей.

Куча (*heap*) — это двоичное дерево, которое предполагает следующие условия:

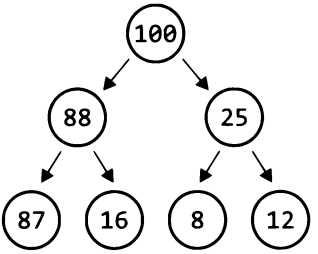
- значение каждого узла должно превышать значение каждого из его потомков. Это правило определяет основное *свойство кучи*;
- дерево должно быть *полным* (что это значит, я объясню чуть позже).

Разберем оба пункта, начиная со свойства кучи.

Свойство кучи

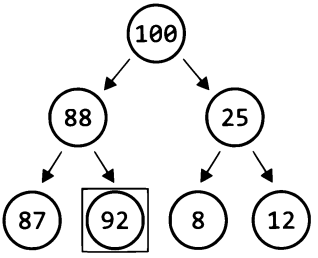
Свойство кучи — это условие, которое гласит, что значение каждого узла должно превышать значение каждого из его потомков.

Например, дерево на следующей схеме удовлетворяет этому условию, так как значение каждого его узла превышает значение любого из его потомков.



Здесь значение корневого узла 100 превышает значения всех его потомков. Точно так же значение узла 88 превышает значения обоих его дочерних элементов. То же верно и для узла 25.

Следующее дерево — неправильная куча, так как оно не удовлетворяет ее основному свойству:



Значение узла 92 превышает значение его родительского элемента 88, что нарушает главное условие.

Обратите внимание, что структуры кучи и двоичного дерева поиска сильно различаются. Во втором случае правый дочерний элемент каждого узла превышает значение данного узла. Но в куче у узла *не может* быть потомка, значение которого превышает его собственное. Так что не стоит валить эти структуры данных «в одну кучу».

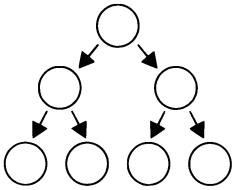
Мы можем создать кучу с противоположным свойством, сделав так, чтобы каждый узел содержал *меньшее* значение, чем любой из его потомков. Это и есть min-куча, о которой я уже говорил. Но мы сосредоточимся на max-куче, где значение каждого узла *превышает* значение всех его потомков. По сути, разница между этими кучами незначительная, так как в остальном они идентичны.

Полные деревья

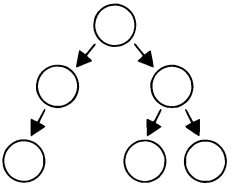
Теперь перейдем ко второму условию: дерево должно быть полным.

Полное дерево — это дерево, в котором все уровни заполнены узлами. При этом в последнем уровне *могут* быть пустые позиции, если справа от них нет никаких узлов. Лучше всего показать это на примере.

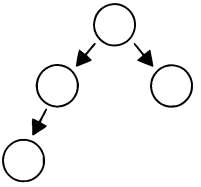
Следующее дерево полное, так как каждый уровень (ряд) дерева заполнен узлами:



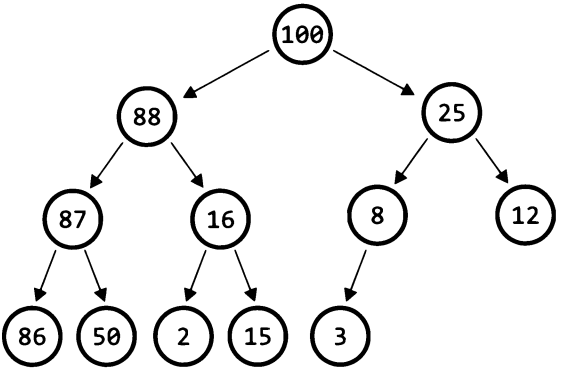
Следующее дерево *не* полное, так как на его третьем уровне нет узла:



Это дерево считается полным, так как пустые позиции есть только на нижнем уровне, а справа от них нет ни одного узла:



Получается, куча — это дерево, которое удовлетворяет свойству кучи *и* является полным. Вот еще один пример:



Это правильная куча, так как значение каждого узла превышает значения любого из его потомков, а дерево является полным. Хотя в нижнем ряду есть пустые позиции, справа от них нет узлов.

Особенности кучи

Теперь, когда мы поняли, что такое куча, рассмотрим некоторые ее характеристики.

Хотя куча должна быть упорядочена определенным образом, это условие бесполезно, когда дело доходит до поиска значения в ней.

Допустим, мы хотим найти значение 3 в куче выше. Если мы начнем с корневого узла 100, то в каком поддереве нам искать? В случае с двоичным деревом поиска мы бы знали, что значение 3 может находиться только среди левых потомков узла 100. Но здесь мы знаем лишь то, что искомое значение 3 должно быть потомком узла 100 и не может быть его предком. Но мы понятия не имеем, с какого дочернего элемента нам продолжать поиск. В этом примере значение 3 находится среди правых потомков узла 100, но в принципе оно могло оказаться и среди левых.

Из-за этого кучи считаются *слабо упорядоченными* в сравнении с двоичными деревьями поиска. Конечно, *некоторый* порядок в кучах есть: значения потомков не могут превышать значения предков, но этого недостаточно для того, чтобы сделать поиск в них целесообразным.

У кучи есть еще одна особенность, о которой вы, наверное, уже догадались: ее корневой узел всегда содержит *наибольшее* значение (в min-куче — наименьшее).

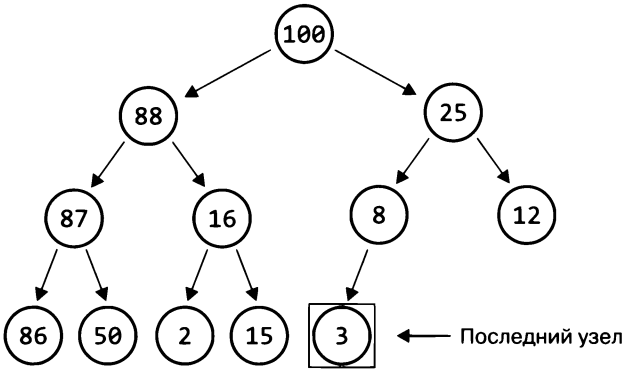
Именно это делает кучу отличным инструментом для реализации приоритетных очередей. Мы используем такие очереди для получения доступа к значению с наивысшим приоритетом, и в случае с кучей всегда знаем, что можем обнаружить его в корневом узле. Получается, что корневой узел кучи содержит элемент с наивысшим приоритетом.

Куча поддерживает две основные операции: вставку и удаление значения. Для поиска значения в куче нужно проверить каждый узел, поэтому такую операцию здесь выполняют нечасто (она может предусматривать необязательную операцию «чтения» — получение доступа к значению корневого узла).

Прежде чем мы перейдем к обсуждению основных операций над кучей, вам нужно узнать еще один термин, который будет активно использоваться в последующих описаниях алгоритмов.

В куче есть так называемый *последний узел* — крайний правый элемент нижнего уровня.

Взгляните на это изображение.



Здесь элемент со значением 3 — это последний узел, крайний правый элемент в нижнем ряду.

Теперь давайте разберемся с основными операциями над кучей.

Вставка в кучу

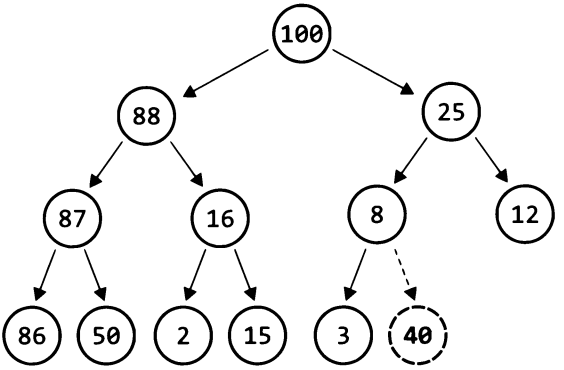
Чтобы вставить в кучу новое значение, мы выполняем следующие действия.

1. Создаем узел с новым значением и вставляем его в первую свободную позицию в нижнем уровне. Так это значение становится последним узлом кучи.

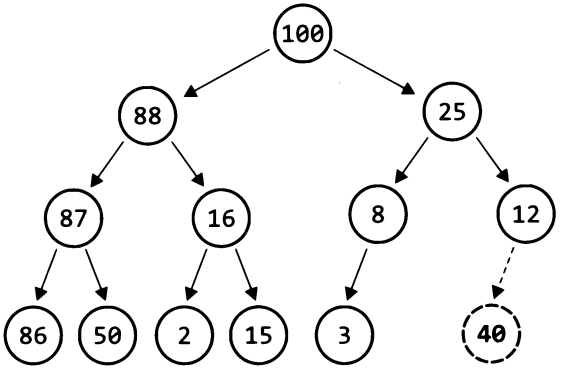
2. Сравниваем значение нового узла со значением его родительского элемента.
3. Если значение нового узла превышает значение родительского, меняем их местами.
4. Повторяем шаг 3, перемещая новый узел вверх по куче, пока у него не появится родитель, значение которого превышает его собственное.

Разберем этот алгоритм на примере. Допустим, нам нужно вставить в кучу значение 40:

Шаг 1: добавляем 40 в качестве последнего узла кучи:

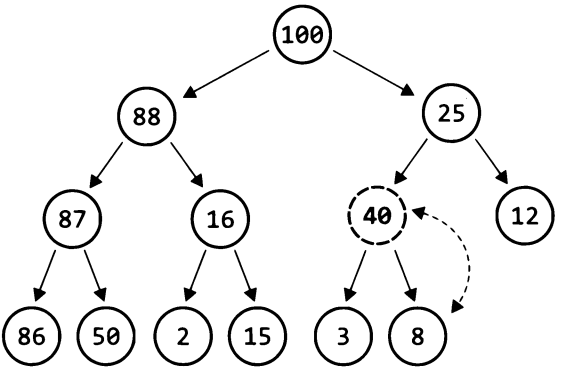


Обратите внимание: такое расположение нового узла было бы неверным:

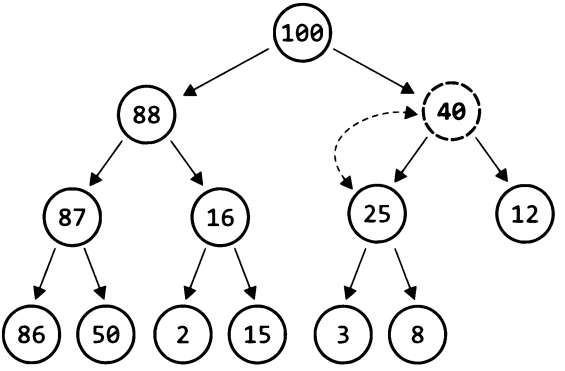


Добавляя узел 40 как дочерний элемент узла 12, мы делаем дерево *неполным*, так как новый узел находится справа от пустой позиции. Чтобы куча оставалась кучей, она должна быть полной.

Шаг 2: сравниваем значение 40 со значением родительского узла, которым оказывается 8. 40 больше 8, поэтому меняем эти узлы местами:



Шаг 3: сравниваем значение 40 со значением его нового родительского узла 25. 40 больше 25, и мы снова меняем узлы местами:



Шаг 4: сравниваем значение 40 со значением родительского узла 100. 40 меньше 100 — процесс вставки завершен!

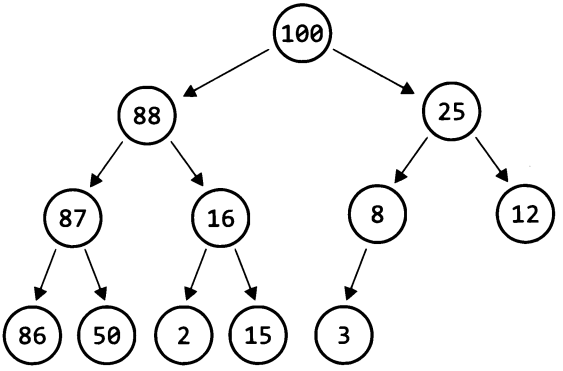
Перемещение нового узла вверх по куче называется *просачиванием*. При этом иногда узел движется вверх вправо, а иногда вверх влево, но всегда вверх, пока не займет правильное положение.

Временная сложность вставки значения в кучу — $O(\log N)$. Как было сказано в прошлой главе, любое двоичное дерево из N узлов содержит примерно $\log(N)$ уровней. Поскольку в худшем случае новому значению придется просачиваться до самого верха, на этот процесс уйдет не более $\log(N)$ шагов.

Поиск последнего узла

Алгоритм вставки кажется довольно простым, но в нем есть одна загвоздка. На первом этапе мы помещаем новое значение в качестве последнего узла кучи. Но возникает вопрос: как найти подходящее для него место?

Еще раз посмотрим, как выглядит куча до вставки значения 40:

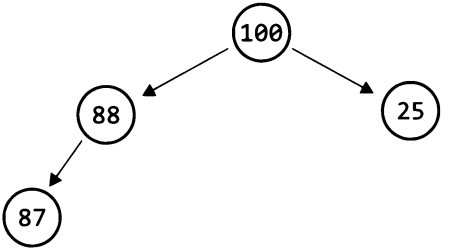


Становится очевидно, что для добавления значения 40 в качестве последнего узла мы должны сделать его правым дочерним элементом узла 8, ведь именно это следующее доступное место в нижнем ряду.

Но у компьютера нет глаз, и он не воспринимает кучу как набор рядов. Он видит только корневой узел и может переходить по ссылкам к его дочерним элементам. Как же нам реализовать алгоритм, позволяющий компьютеру найти подходящее место для вставки нового значения?

Вернемся к куче из примера. Начиная с корневого узла 100, говорим ли мы компьютеру искать место для вставки нового последнего узла среди правых потомков корня?

В нашем примере это место действительно в правом поддереве корневого узла 100, но взгляните на следующую кучу:



Здесь мы должны были бы добавить новый узел в качестве правого дочернего элемента узла 88, который находится в *левом* поддереве корня 100.

Получается, как и в случае с поиском значения в куче, для нахождения ее последнего узла (или подходящего для него места) нужно проверить все элементы.

Так как же нам найти место для вставки последнего узла? Я расскажу об этом чуть позже. Обещаю, мы еще вернемся к этой проблеме.

А пока рассмотрим еще одну операцию над кучей: удаление значения из нее.

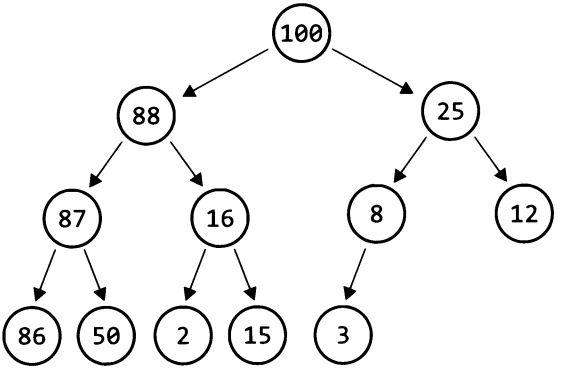
Удаление из кучи

Главное, что нужно знать об удалении значения из кучи — то, что *мы всегда удаляем только корневой узел*. Это соответствует принципу работы приоритетной очереди, так как мы получаем доступ к самому приоритетному элементу и удаляем только его.

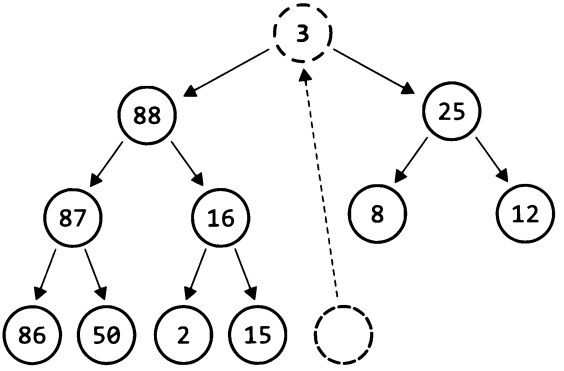
Алгоритм удаления корневого узла из кучи следующий.

1. Перемещение *последнего узла* на место корневого, которое приводит к фактическому удалению исходного корня.
2. Постановка (просачивание) корневого узла вверх на подходящее для него место. Позже я объясню, как это работает.

Допустим, мы собираемся удалить корневой узел из следующей кучи.



Корневой узел здесь — 100. Чтобы удалить его, мы перезаписываем значение корня, заменяя его значением последнего узла (у нас это значение 3). Перемещаем узел 3 на место исходного корня 100:



Нельзя оставлять кучу в таком виде, так как значения большинства потомков нового корня выше его собственного, что противоречит главному свойству кучи. Чтобы этого не допустить, нужно поместить узел 3 вниз на соответствующее ему место.

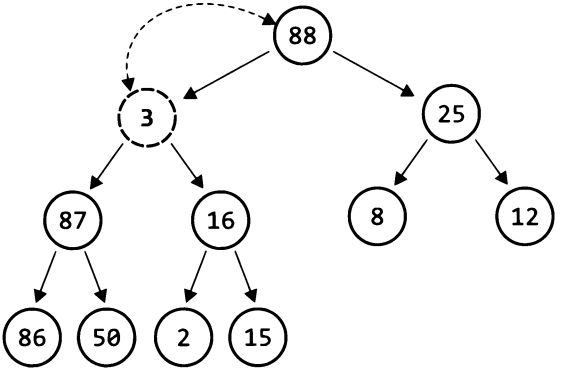
Просачивание вниз сложнее просачивания вверх, так как предполагает наличие двух возможных направлений: мы можем поменять узел местами с левым или правым дочерним элементом. Но при просачивании вверх такой проблемы нет, ведь у каждого узла только один родитель, с которым его можно поменять местами.

Так выглядит алгоритм просачивания узла *вниз*. Для ясности будем называть этот узел просачивающимся (согласен, звучит ужасно).

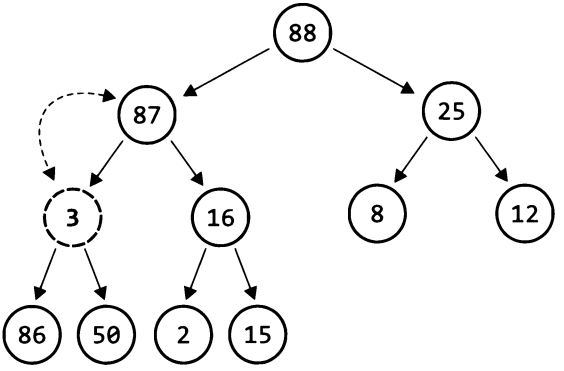
1. Проверяем оба дочерних элемента просачивающегося узла и определяем наибольший.
2. Если значение просачивающегося узла меньше значения наибольшего из двух его дочерних узлов, меняем их местами.
3. Повторяем шаги 1 и 2, пока у просачивающегося узла не останется ни одного дочернего элемента, значение которого превышает его собственное.

Рассмотрим работу этого алгоритма на примере.

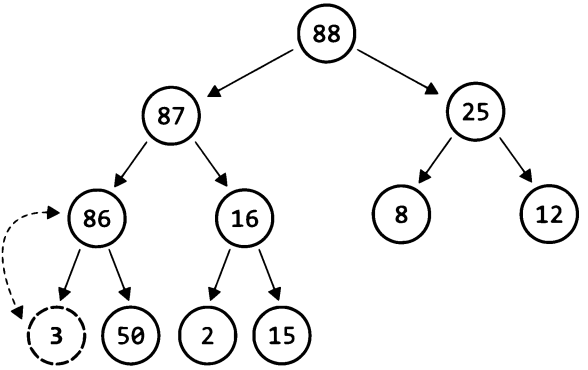
Шаг 1: сейчас у просачивающегося узла 3 два дочерних узла — 88 и 25. Значение 88 — большее из двух, а так как 3 меньше 88, мы меняем их местами:



Шаг 2: теперь дочерние элементы просачивающегося узла — 87 и 16. Значение наибольшего из них (87) превышает значение просачивающегося узла (3), поэтому меняем их местами:



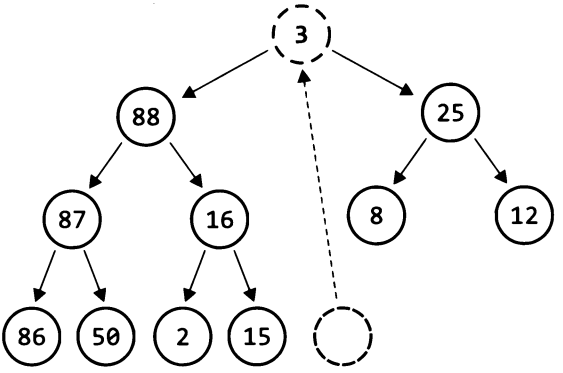
Шаг 3: теперь дочерние элементы просачивающегося узла — 86 и 50. Значение наибольшего из них (86) превышает значение просачивающегося узла (3), поэтому меняем их местами:



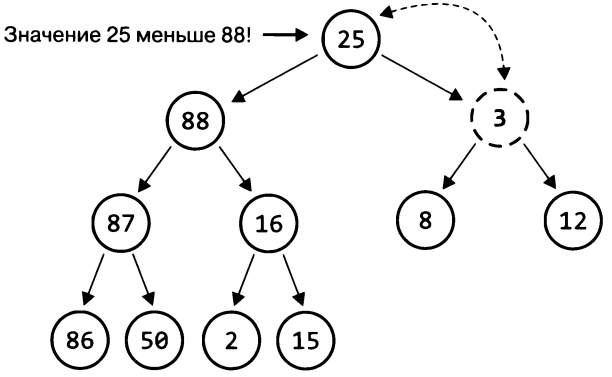
Теперь у просачивающегося узла нет дочерних элементов, значения которых больше его собственного (у него их вообще нет). Получается, свойство кучи восстановлено, а значит, процесс просачивания завершен.

Мы всегда заменяем просачивающийся узел *наибольшим* из двух его дочерних, потому что в противном случае структура данных перестанет удовлетворять свойству кучи. Посмотрите, что происходит при попытке поменять местами просачивающийся узел с наименьшим из его дочерних элементов.

Пусть значением корня снова будет 3:



Поменяем местами корень 3 и узел 25 — наименьший из его дочерних элементов:



Теперь узел 25 — родитель узла 88. Так как 88 превышает значение своего родительского узла, свойство кучи нарушено.

Как и в случае вставки, временная сложность удаления значения из кучи равна O(log *N*), так как при выполнении этой операции корневой узел должен просачиваться вниз через все log(*N*) уровней кучи.

Кучи и упорядоченные массивы

Теперь, когда мы разобрались с эффективностью кучи, посмотрим, почему эта структура данных отлично подходит для реализации приоритетных очередей.

Взгляните на следующую таблицу, где кучи сравниваются с упорядоченными массивами:

	Упорядоченный массив	**Куча**
Вставка	O(*N*)	O(log *N*)
Удаление	O(1)	O(log *N*)

Кажется, что у обеих структур данных примерно одинаковая эффективность. Упорядоченный массив работает медленнее кучи при вставке значения, но быстрее — при удалении.

И все же лучше выбрать кучу, и вот почему.

O(1) — это чрезвычайно быстро, O(log *N*) — *очень* быстро, а O(*N*) — относительно медленно. Зная это, таблицу выше можно переписать так:

	Упорядоченный массив	**Куча**
Вставка	Медленно	Очень быстро
Удаление	Чрезвычайно быстро	Очень быстро

Теперь понятно, почему лучше выбирать кучу. Если сравнивать структуру данных, которая всегда работает быстро, с той, которая иногда работает очень быстро, а иногда — медленно, конечно, мы отдадим предпочтение первой.

Важно отметить, что приоритетные очереди обычно предусматривают примерно одинаковое число операций вставки и удаления. В примере с отделением неотложной помощи мы стремились к тому, чтобы операции вставки и удаления выполнялись одинаково быстро. Если какая-то из них будет медленной, наша приоритетная очередь станет неэффективной.

Выходит, использование кучи гарантирует, что основные операции над приоритетной очередью — вставка и удаление — будут выполняться очень быстро.

Проблема последнего узла... снова

Хотя алгоритм удаления значения из кучи кажется довольно простым, он снова поднимает проблему последнего узла.

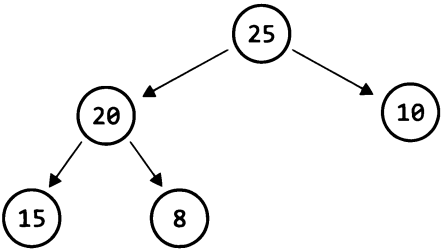
Как я уже говорил, на первом этапе удаления мы должны переместить последний узел на место корня. Но как отыскать этот узел?

Прежде чем перейти к решению этой проблемы, выясним, почему операции вставки и удаления так сильно зависят от нахождения последнего узла. Почему мы не можем вставить новые значения куда-нибудь еще? И почему при удалении нельзя заменить корень другим узлом вместо последнего?

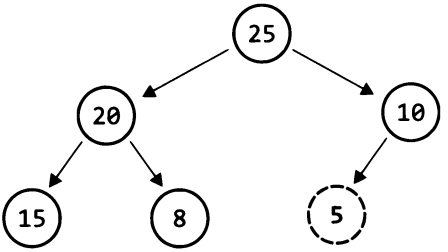
Если задуматься, то при использовании других узлов куча стала бы неполной. Но тогда возникает следующий вопрос: почему для кучи так *важна* полнота?

Потому что мы хотим, чтобы наша куча оставалась *сбалансированной*.

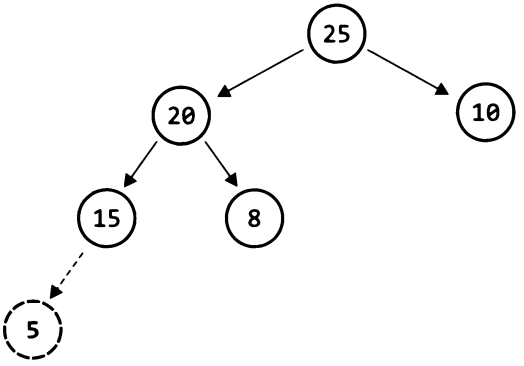
Чтобы убедиться в этом, еще раз рассмотрим процесс вставки. Допустим, у нас есть следующая куча:



Если мы хотим вставить в нее значение 5, то единственный способ сохранить кучу хорошо сбалансированной — сделать 5 последним узлом, здесь — дочерним элементом узла 10:

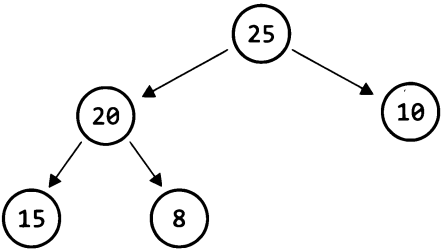


Любая альтернатива приведет к дисбалансу. Допустим, в параллельной вселенной алгоритм вставлял бы новый узел в качестве самого нижнего левого дочернего, который мы могли бы легко отыскать, обойдя все левые дочерние элементы. В этом случае узел 5 стал бы дочерним элементом узла 15:



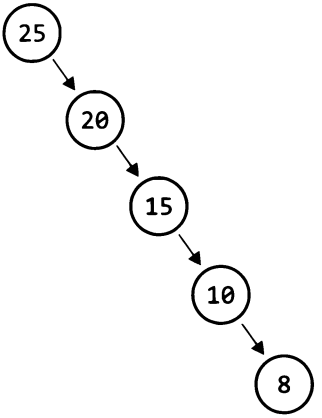
Сейчас наша куча уже не сбалансирована, и этот дисбаланс лишь усиливался бы по мере добавления новых узлов в соответствии с этим алгоритмом.

Точно так же при удалении значения из кучи мы всегда превращаем последний узел в корень, ведь иначе куча может стать несбалансированной. Вернемся к примеру:



Если бы в альтернативной вселенной мы всегда перемещали на место корня нижний правый узел, то 10 стал бы корневым — и мы получили бы несбалансированную кучу с множеством левых потомков и без единого правого.

Важность сбалансированности в том, что именно это свойство позволяет выполнять операции за время O(log *N*). Обход же сильно несбалансированного дерева, например этого, мог бы занять O(*N*) времени:



Но это возвращает нас к проблеме последнего узла. Какой алгоритм позволил бы находить последний узел любой кучи? (Без обхода всех *N* узлов.)

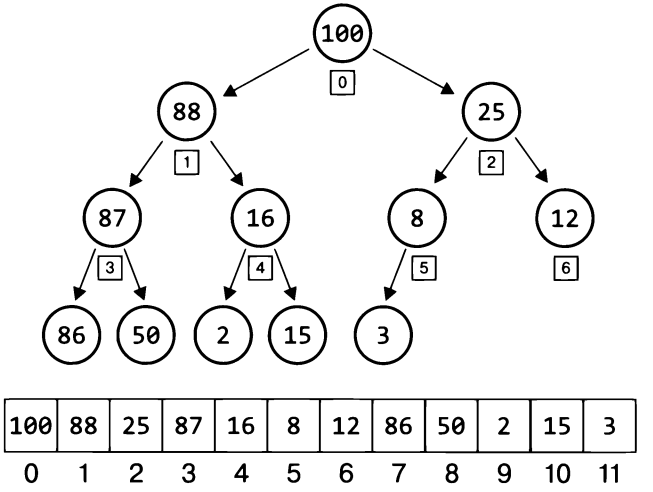
И тут наш сюжет делает неожиданный поворот.

Массивы в качестве куч

Для проведения операций над кучей нужно найти последний узел, поэтому нужно сделать так, чтобы поиск был максимально эффективным. Именно поэтому кучи *обычно реализуются на основе массивов.*

Хотя до сих пор мы думали, что каждое дерево состоит из независимых узлов, соединенных друг с другом ссылками (как в связном списке), вскоре вы поймете, что для реализации кучи подойдет и массив. То есть сама куча может быть абстрактным типом данных, в основе которого лежит массив.

Ниже показано, как массив используется для хранения значений кучи.



Здесь мы присваиваем каждому узлу определенный индекс массива. На прошлой диаграмме индекс каждого узла указан в квадрате под ним. Если вы посмотрите внимательно, то заметите, что мы присваиваем индексы узлам кучи по определенному шаблону.

Корневой узел всегда хранится в позиции с индексом 0. Мы опускаемся на уровень ниже и двигаемся слева направо, присваивая каждому узлу индекс следующей доступной ячейки в массиве. Итак, на втором уровне левому узлу (88) присваивается индекс 1, а правому (25) — 2. По достижении конца уровня переходим на следующий и продолжаем.

Мы используем массив для реализации кучи, потому что он позволяет решить проблему последнего узла. Как?

Когда мы реализуем кучу так, *последний узел всегда будет конечным элементом массива.* Присваивая узлам индексы массива, мы двигаемся сверху вниз и слева

направо, поэтому последний узел всегда будет конечным значением. Например, последний узел 3 из прошлого примера — это последнее значение массива.

Поскольку последний узел всегда в конце массива, найти его несложно: просто получить доступ к последнему элементу массива. Кроме того, при вставке нового узла в кучу мы помещаем его в конец массива, делая его последним узлом.

Прежде чем продолжить разбираться в принципе работы кучи на основе массива, создадим ее базовую структуру. Вот начало реализации кучи на языке Ruby:

```
class Heap
  def initialize
    @data = []
  end

  def root_node
    return @data.first
  end

  def last_node
    return @data.last
  end
end
```

Мы инициализируем кучу пустым массивом. У нас есть метод `root_node`, который возвращает первый элемент массива, и `last_node`, который возвращает его последнее значение.

Обход кучи на основе массива

Итак, алгоритмы вставки в кучу и удаления из нее предполагают просачивание узла, что означает обход кучи — получение доступа к родительскому или дочернему элементу узла. Но как мы можем переходить от узла к узлу, если все их значения просто хранятся в массиве? Было бы просто обойти кучу, будь у нас возможность следовать по ссылкам каждого узла. Но теперь, когда в основе кучи лежит массив, как узнать, какие именно узлы связаны друг с другом?

Ответ на этот вопрос довольно интересный. Оказывается, присвоив индексы узлам кучи по описанному ранее шаблону, мы можем:

- найти левый дочерний элемент узла по формуле `(index * 2) + 1`;
- найти правый дочерний элемент узла по формуле `(index * 2) + 2`.

Взгляните на узел 16 из прошлой диаграммы, который находится в ячейке с индексом 4. Чтобы найти его левый дочерний элемент, мы умножаем его индекс (4) на 2 и прибавляем 1, получая 9. Значит, узел с индексом 9 — это левый дочерний элемент узла с индексом 4.

Аналогично, чтобы найти правый дочерний элемент узла с индексом 4, мы умножаем 4 на 2 и прибавляем 2, получая 10. Значит, узел с индексом 10 — это правый дочерний элемент узла с индексом 4.

Эти формулы работают всегда, поэтому мы можем обращаться с массивом как с деревом.

Добавим эти два метода в класс `Heap`:

```
def left_child_index(index)
  return (index * 2) + 1
end

def right_child_index(index)
  return (index * 2) + 2
end
```

Каждый из них принимает индекс в массиве, возвращая индекс левого или правого дочернего элемента соответственно.

Вот еще одно важное свойство кучи на основе массива: возможность нахождения родителя узла по формуле `(index - 1) / 2`.

Обратите внимание, что в этой формуле используется целочисленное деление, при котором мы отбрасываем все числа после запятой. Например, результатом операции `3/2` считается `1`, а не `1,5`.

Вернемся к нашему примеру и сосредоточимся на индексе 4. Если мы вычтем из него 1, а затем разделим результат на 2, то получим 1. И, как вы можете видеть на диаграмме, родитель узла с индексом 4 находится по индексу 1.

Теперь мы можем добавить в наш класс `Heap` еще один метод:

```
def parent_index(index)
  return (index - 1) / 2
end
```

Он принимает индекс узла и вычисляет индекс его родительского элемента.

Программная реализация: вставка значения в кучу

Теперь, когда у нас есть все основные элементы кучи, реализуем алгоритм вставки:

```
def insert(value)
  # Делаем значение последним узлом, вставляя его в конец массива:
  @data << value

  # Отслеживаем индекс только что вставленного узла:
```

```
    new_node_index = @data.length - 1

    # Следующий цикл выполняет алгоритм просачивания узла вверх.

    # Если новый узел не находится на месте корня,
    # а его значение превышает значение его родительского узла:
    while new_node_index > 0 &&
    @data[new_node_index] > @data[parent_index(new_node_index)]

      # Меняем местами новый узел и его родительский элемент:
      @data[parent_index(new_node_index)], @data[new_node_index] =
      @data[new_node_index], @data[parent_index(new_node_index)]

      # Обновляем индекс нового узла:
      new_node_index = parent_index(new_node_index)
    end
  end
```

Как обычно, разберем код по частям.

Метод `insert` принимает значение, которое мы вставляем в кучу. Сначала делаем это новое значение последним узлом, добавляя его в конец массива:

```
@data << value
```

Отслеживаем индекс нового узла — он понадобится нам позднее. На этом этапе этот индекс — последний в массиве:

```
new_node_index = @data.length - 1
```

С помощью цикла `while` перемещаем новый узел вверх до нужного места:

```
while new_node_index > 0 &&
@data[new_node_index] > @data[parent_index(new_node_index)]
```

Цикл работает, пока выполняются два условия. Первое: значение нового узла должно превышать значение его родительского элемента. Второе: индекс нового узла должен быть больше 0, так как попытка сравнить корневой узел с его несуществующим родителем может привести к неоднозначным результатам.

При каждой итерации этого цикла мы меняем местами новый узел и его родительский элемент, так как значение нового узла сейчас превышает значение его родителя:

```
@data[parent_index(new_node_index)], @data[new_node_index] =
@data[new_node_index], @data[parent_index(new_node_index)]
```

Затем обновляем индекс нового узла:

```
new_node_index = parent_index(new_node_index)
```

Этот цикл выполняется, пока значение нового узла превышает значение его родителя, поэтому его работа завершается, когда новый узел оказывается на своем месте.

Программная реализация: удаление значения из кучи

Это реализация для удаления элемента из кучи на языке Ruby. Основной метод здесь — `delete`, но для упрощения кода мы создали два вспомогательных: `has_greater_child` и `calculate_larger_child_index`:

```
def delete
  # Из кучи всегда удаляется только корневой узел, поэтому
  # извлекаем из массива последний узел и делаем его корневым:
  @data[0] = @data.pop
  # Отслеживаем текущий индекс "просачивающегося узла":
  trickle_node_index = 0

  # Следующий цикл выполняет алгоритм просачивания узла вниз:

  # Продолжаем выполнять этот цикл, пока у просачивающегося узла есть
  # дочерний, значение которого превышает его собственное:
  while has_greater_child(trickle_node_index)
    # Сохраняем индекс большего дочернего элемента в переменной:
    larger_child_index = calculate_larger_child_index(trickle_node_index)

    # Меняем местами просачивающийся узел и его больший дочерний элемент:
    @data[trickle_node_index], @data[larger_child_index] =
    @data[larger_child_index], @data[trickle_node_index]

    # Обновляем индекс просачивающегося узла:
    trickle_node_index = larger_child_index
  end
end

def has_greater_child(index)
  # Проверяем, есть ли у узла с заданным индексом
  # левые и правые дочерние элементы и превышает ли значение любого
  # из них значение узла с заданным индексом:
   (@data[left_child_index(index)] &&
    @data[left_child_index(index)] > @data[index]) ||
   (@data[right_child_index(index)] &&
    @data[right_child_index(index)] > @data[index])
end

def calculate_larger_child_index(index)
  # При отсутствии правого дочернего элемента:
  if !@data[right_child_index(index)]
    # Возвращаем индекс левого дочернего элемента:
    return left_child_index(index)
  end

  # Если значение правого дочернего узла превышает значение левого:
  if @data[right_child_index(index)] > @data[left_child_index(index)]
```

```
    # Возвращаем индекс правого дочернего элемента:
    return right_child_index(index)
  else # Если значение левого дочернего узла больше или равно значению
       # правого,возвращаем индекс левого дочернего элемента:
    return left_child_index(index)
  end
```

Подробно разберем метод `delete`.

`delete` не принимает никаких аргументов, так как мы всегда удаляем только корневой узел. Вот как работает этот метод.

Сначала мы удаляем последнее значение из массива и делаем его первым:

```
@data[0] = @data.pop
```

Эта простая строка кода удаляет исходный корневой узел: мы перезаписываем значение корневого узла, заменяя его значением последнего.

Далее нужно переместить новый корневой узел в нужное место. Ранее мы назвали этот процесс «просачиванием», и наш код вполне отражает его суть.

Прежде чем это сделать, отслеживаем индекс просачивающегося узла — он понадобится нам позднее. Сейчас его индекс — 0:

```
trickle_node_index = 0
```

Теперь используем цикл `while` для просачивания узла вниз. Этот цикл продолжает выполняться, пока у просачивающегося узла есть дочерние элементы, значения которых превышают его собственное:

```
while has_greater_child(trickle_node_index)
```

Здесь используется метод `has_greater_child`, который проверяет наличие у этого узла дочерних элементов, значения которых превышают его собственное.

В этом цикле мы сначала находим индекс наибольшего из дочерних элементов просачивающегося узла:

```
larger_child_index = calculate_larger_child_index(trickle_node_index)
```

Мы используем метод `calculate_larger_child_index`, который возвращает индекс наибольшего дочернего элемента просачивающегося узла, и сохраняем этот индекс в переменной `larger_child_index`.

Теперь меняем местами просачивающийся узел и его наибольший дочерний элемент:

```
@data[trickle_node_index], @data[larger_child_index] =
@data[larger_child_index], @data[trickle_node_index]
```

Наконец, обновляем индекс просачивающегося узла, меняя его на индекс дочернего элемента, с которым он только что поменялся местами:

```
trickle_node_index = larger_child_index
```

Другие варианты реализации кучи

Итак, реализация кучи завершена. Мы использовали для этого массив, а *могли бы* реализовать кучу, например на основе связных списков (в этом случае для решения проблемы последнего узла использовались бы двоичные числа).

Но реализация на основе массива более распространена, поэтому я представил в книге именно ее. Еще очень интересно посмотреть, как использовать массив для реализации дерева.

На самом деле на основе массива можно реализовать *любое* двоичное дерево, в том числе и поиска из прошлой главы. Но куча — это первый случай двоичного дерева, где реализация на основе массива дает преимущество, помогая легко находить последний узел.

Кучи в качестве приоритетных очередей

Теперь, когда вы понимаете, как работают кучи, вернемся к приоритетным очередям.

Как вы помните, основная функция такой очереди — предоставление немедленного доступа к элементу с наивысшим приоритетом. В примере с отделением неотложной помощи мы хотели, чтобы приоритет отдавался пациенту в самом тяжелом состоянии.

Именно поэтому куча лучше всего подходит для реализации таких очередей, так как дает немедленный доступ к самому приоритетному элементу, который всегда находится в корневом узле. После того как мы разберемся с одним приоритетным элементом (и удалим его), на вершине кучи окажется следующий по порядку приоритетности. При этом такие операции над кучей, как вставка и удаление, выполняются за время $O(\log N)$.

Сравните это с упорядоченным массивом, вставка в который занимает $O(N)$ времени, так как компьютер должен гарантировать, что каждое новое значение в итоге окажется на своем месте.

Выходит, что слабая упорядоченность кучи — ее *преимущество*. То, что она не обязана быть идеально упорядоченной, позволяет нам вставлять новые значения за время $O(\log N)$. При этом куча *достаточно упорядочена*, для того чтобы мы

всегда могли получить доступ к одному нужному нам элементу. В этом и заключается основная ценность этой структуры данных.

Выводы

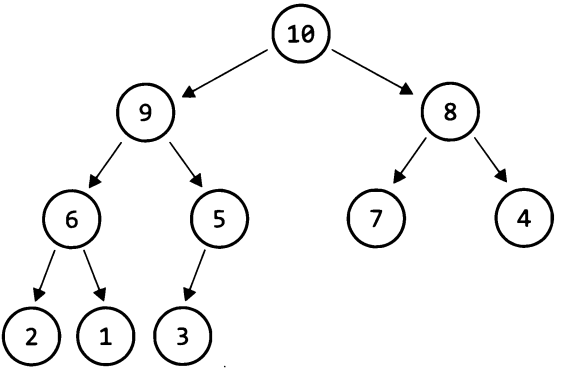
Мы поговорили о том, как разные типы деревьев позволяют оптимизировать процесс выполнения разных типов задач, и выяснили, что двоичные деревья поиска обеспечивают высокую скорость поиска и минимальные временные затраты при вставке значений, а кучи идеально подходят для построения приоритетных очередей.

В следующей главе мы рассмотрим другой тип дерева, который используется для выполнения некоторых распространенных операций над текстовыми данными.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 16».

1. Изобразите следующую кучу после вставки в нее значения 11:



2. Изобразите эту же кучу после удаления из нее корневого узла.
3. Представьте, что создали совершенно новую кучу, вставив в нее следующий порядок чисел: 55, 22, 34, 10, 2, 99, 68. Если затем вы извлечете эти числа из кучи по одному и вставите в новый массив, в каком порядке они будут располагаться?

ГЛАВА 17

Префиксные деревья

Вы когда-нибудь задумывались над тем, как работает функция автозаполнения в вашем смартфоне? Автозаполнение срабатывает, когда вы начинаете набирать какое-нибудь слово, например *catn*, а ваш телефон предлагает на выбор варианты *catnip* или *catnap* (да, я постоянно пишу друзьям о кошачьей мяте).

Чтобы эта функция работала, у вашего телефона должен быть доступ к словарю. Но какая именно структура данных используется для хранения всех этих слов?

Давайте представим, что все слова английского языка хранятся в массиве. Если бы он был неотсортированным, нам пришлось бы проверить *каждое слово* в этом словаре, чтобы найти те, которые начинаются с *catn*. Такая операция выполняется за время O(N), что очень медленно, учитывая, что N здесь — это количество слов в словаре.

Хеш-таблица здесь тоже не поможет — она хеширует *слово целиком*, чтобы определить, где именно в памяти должно храниться значение. Из-за отсутствия в хеш-таблице ключа *catn* мы не сможем легко найти в ней слова *catnip* или *catnap*.

Дела пойдут на лад, если мы будем хранить слова внутри упорядоченного массива. Если бы массив содержал все слова в алфавитном порядке, мы могли бы использовать двоичный поиск, чтобы найти то, что начинается с *catn*, за время O(log N). И хотя O(log N) — это уже неплохо, мы можем добиться еще большего. На самом деле при использовании специальной древовидной структуры данных скорость поиска нужных слов может достигать O(1).

В этой главе мы поговорим о *префиксных деревьях*, которые могут использоваться для создания приложений, ориентированных на работу с текстовыми данными, позволяя реализовать такие важные функции, как автозаполнение и автозамена. Эта структура данных может применяться и для решения других

задач, например для организации таких данных, как IP-адреса или телефонные номера.

Префиксные деревья

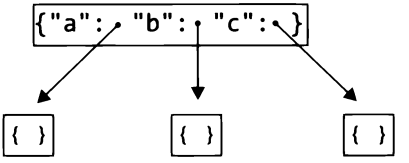
Префиксное дерево — это древовидная структура данных, которая идеально подходит для реализации функций вроде автозаполнения.

Прежде чем мы углубимся в детали, отмечу, что префиксные деревья не так хорошо документированы, как другие структуры данных в этой книге, и могут быть реализованы по-разному. Я выбрал реализацию, которую считаю наиболее простой и понятной, но вы можете найти и другие. В любом случае в основе большинства из них лежат одни и те же общие идеи.

Узел префиксного дерева

Как и большинство других деревьев, префиксное дерево состоит из узлов, указывающих на другие узлы. Но именно эта структура данных *не* двоична. У узла в двоичном дереве может быть не более двух дочерних элементов, тогда как у узла префиксного дерева — *сколько угодно*.

В нашей реализации каждый узел префиксного дерева содержит хеш-таблицу, где ключи — это буквы латинского алфавита, а значения — другие узлы этого дерева. Взгляните на следующую схему:



Здесь корневой узел содержит хеш-таблицу с ключами `"a"`, `"b"` и `"c"`, значения которых — это дочерние элементы узла; в этих элементах тоже есть хеш-таблицы, которые, в свою очередь, указывают на *их* дочерние элементы. Здесь мы оставили дочерние хеш-таблицы пустыми, но на дальнейших диаграммах в них будут данные.

Реализация узла префиксного дерева очень проста. Так выглядит наша версия класса `TrieNode` на языке Python:

```
class TrieNode:

  def __init__(self):
    self.children = {}
```

Как видите, здесь только хеш-таблица.

Если мы выведем (в консоль) данные из корневого узла дерева выше, то получим что-то вроде этого:

```
{'a': <__main__.TrieNode instance at 0x108635638>,
 'b': <__main__.TrieNode instance at 0x108635878>,
 'c': <__main__.TrieNode instance at 0x108635ab8>}
```

Опять же, ключи здесь — это односимвольные строки, а значения — экземпляры других узлов префиксного дерева.

Класс Trie

Для создания префиксного дерева нам понадобится отдельный класс `Trie`, который будет отслеживать корневой узел:

```
class Trie:

  def __init__(self):
    self.root = TrieNode()
```

Он отслеживает переменную `self.root`, указывающую на корень дерева. Здесь корень (`TrieNode`) созданного префиксного дерева (`Trie`) изначально пуст.

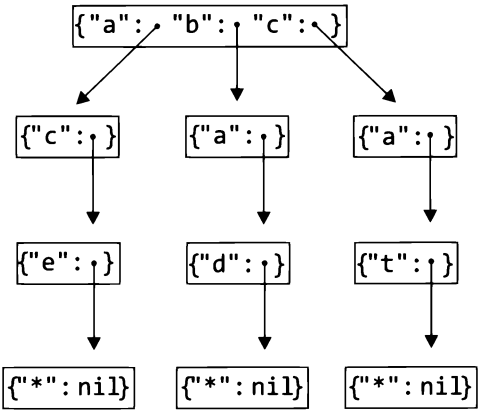
По мере продвижения по этой главе мы будем постепенно добавлять в класс `Trie` методы, позволяющие совершать разные операции над префиксным деревом.

Хранение слов

Итак, мы создаем префиксное дерево, чтобы хранить в нем слова. На следующей диаграмме показано, как эта структура данных хранит слова `"ace"`, `"bad"` и `"cat"`.

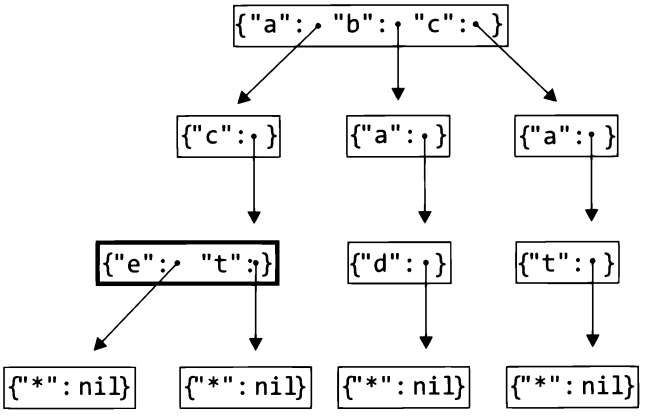
Префиксное дерево хранит эти слова, превращая каждый их символ в узел. Если вы начнете с корневого узла и проследите, например, за ключом `"a"`, то увидите, что он указывает на дочерний узел с ключом `"c"`. Ключ `"c"`, в свою очередь, указывает на узел с ключом `"e"`. Объединив эти три символа, мы получим слово `"ace"`.

Точно так же в дереве хранятся слова `"bad"` и `"cat"`.



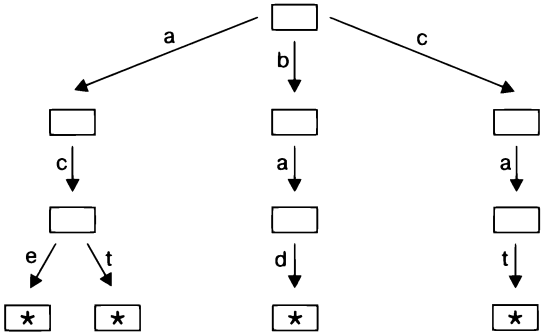
Как видите, у последних символов в этих словах есть свои дочерние узлы. Например, если вы взглянете на узел `"e"` слова `"ace"`, то увидите, что он указывает на дочерний узел, содержащий хеш-таблицу с ключом `"*"` (звездочка) (на самом деле неважно, какое здесь значение — оно может быть и нулевым). Он указывает на то, что мы достигли конца слова `"ace"`. О функции ключа `"*"` мы поговорим позже.

А сейчас перейдем к самому интересному. Допустим, мы хотим сохранить в префиксном дереве слово `"act"`. Для этого мы добавляем к уже существующим ключам `"a"` и `"c"` новый узел с ключом `"t"`:



Как видите, в хеш-таблице выделенного узла теперь *два* дочерних узла: `"e"` и `"t"`. Так мы указываем, что `"ace"` и `"act"` — допустимые слова.

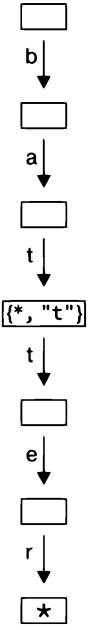
Позже мы будем использовать упрощенные схемы. Так, например, будет выглядеть дерево из прошлого примера:



Такой упрощенный стиль предполагает размещение каждого ключа хеш-таблицы рядом со стрелкой, указывающей на его дочерний узел.

Важность звездочки

Допустим, мы хотим сохранить в префиксном дереве слова `"bat"` и `"batter"`. Это интересный случай, поскольку в `"batter"` есть слово `"bat"`. Эту проблему можно решить так:

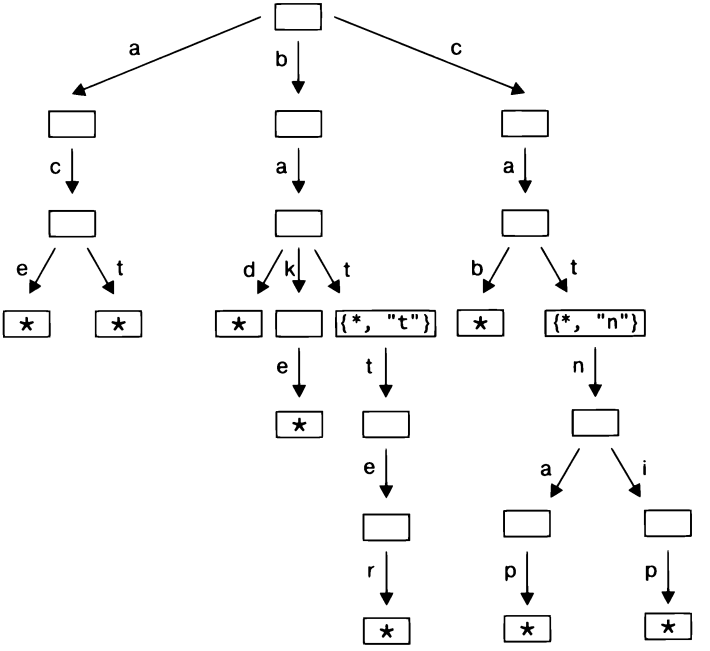


Первый ключ `"t"` указывает на узел с *двумя* ключами: `"*"` (с нулевым значением) и `"t"`, значение которого направлено на другой узел. Это говорит о том, что `"bat"` — это одновременно и отдельное слово, и префикс более длинного слова `"batter"`.

Обратите внимание, что в этой диаграмме вместо классического синтаксиса хеш-таблицы мы используем сокращенный. Это делается для экономии места. Чтобы указать, что узел содержит хеш-таблицу, мы использовали фигурные скобки. Но `{*, "t"}` — это не пара «ключ — значение», а просто два ключа. Значение ключа `"*"` нулевое, а значение ключа `"t"` — следующий узел.

Вот почему эти звездочки так важны: с их помощью можно указать, что те или иные части слов — это тоже отдельные слова.

Рассмотрим все это на более сложном примере. Вот префиксное дерево со словами `"ace"`, `"act"`, `"bad"`, `"bake"`, `"bat"`, `"batter"`, `"cab"`, `"cat"`, `"catnap"` и `"catnip"`:



Префиксные деревья в реальных приложениях могут содержать тысячи самых распространенных слов определенного языка.

Чтобы реализовать функцию автозаполнения, сначала проанализируем основные операции над префиксными деревьями.

Поиск в префиксном дереве

Классическая операция над префиксными деревьями — поиск: это определение наличия в дереве той или иной строки. При этом мы можем выяснить одно из двух: либо эта строка будет *полным* словом, либо его *префиксом* (началом слова). Эти две операции поиска похожи, но мы реализуем вторую — поиск префикса. Она будет находить и полные слова, которые принципиально ничем не отличаются от префиксов.

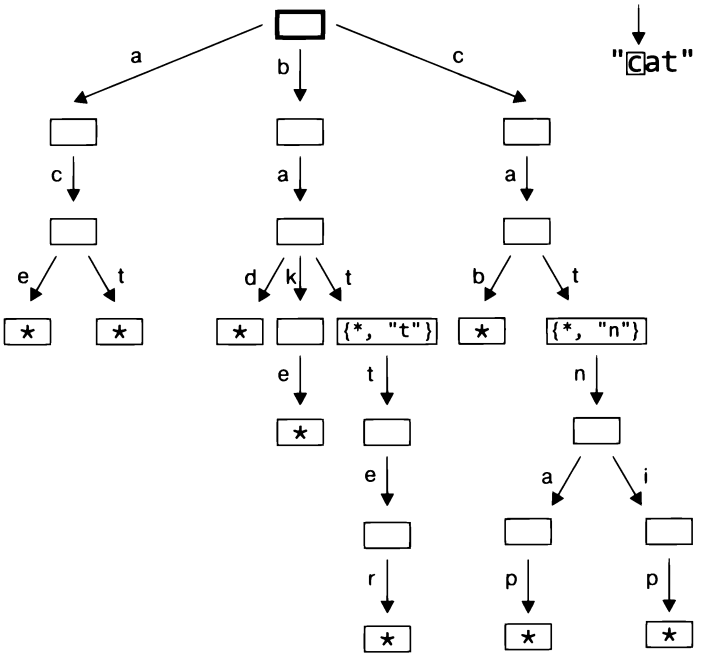
Алгоритм поиска префикса следующий (этапы станут понятнее при рассмотрении следующего примера).

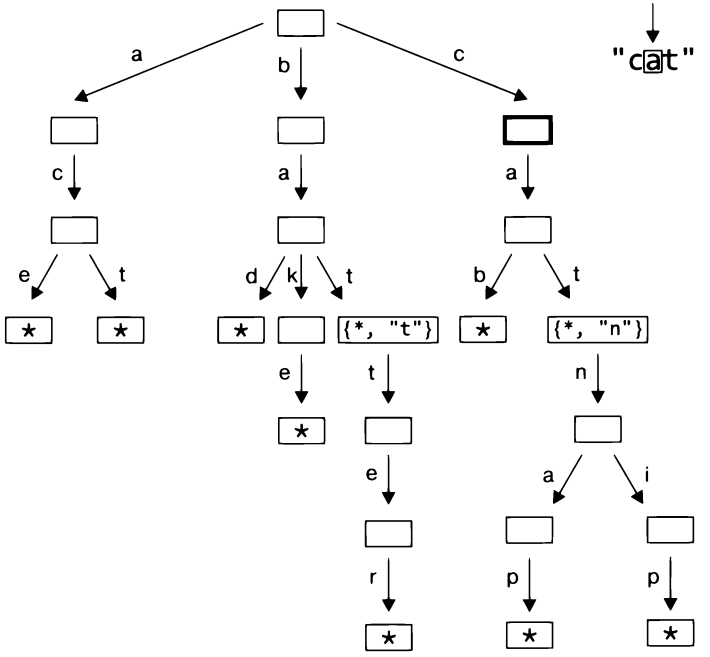
1. Мы создаем переменную `currentNode`, которая в начале поиска указывает на корневой узел.
2. Перебираем все символы искомой строки.
3. Последовательно указывая на каждый символ искомой строки, проверяем наличие у текущего узла `currentNode` дочернего элемента с соответствующим символом в качестве ключа.
4. Если его нет, возвращаем `None`, так как это означает, что в дереве искомой строки нет.
5. Если у `currentNode` *есть* дочерний элемент с текущим символом в качестве ключа, обновляем значение `currentNode`, делая его этим дочерним элементом. Затем возвращаемся к шагу 2 и продолжаем перебирать символы искомой строки.
6. Если мы дошли до конца строки, значит, нашли то, что искали.

Посмотрим на этот алгоритм в действии, выполнив поиск строки `"cat"` в дереве выше.

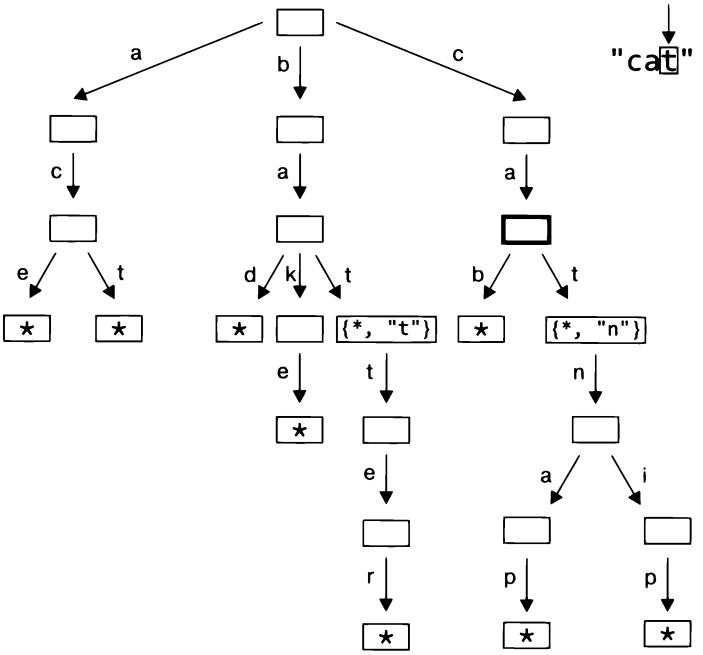
Настройка: задаем корень дерева в качестве текущего узла `currentNode` (на следующих диаграммах он выделен жирным). Мы указываем на первый символ искомой строки (`"c"`), как в верхнем правом углу на следующей диаграмме.

Шаг 1: дочерний ключ корневого узла — `"c"`, поэтому обновляем `currentNode`, делая его значением этого ключа. Продолжая перебор символов искомой строки, указываем на следующий символ (`"a"`) (см. вторую схему).


"cat"
a
b
c
c
a
a
e
t
d
k
t
b
t
*
*
*
*
{*, "t"}
*
{*, "n"}
e
t
n
e
a
i
r
p
p
*
*
*


"cat"
a
b
c
c
a
a
e
t
d
k
t
b
t
*
*
*
*
{*, "t"}
*
{*, "n"}
e
t
n
e
a
i
r
p
p
*
*
*

Шаг 2: проверяем `currentNode` на наличие дочернего элемента с ключом `"a"`. У него такой есть, поэтому делаем этот дочерний элемент новым текущим узлом `currentNode`. Затем приступаем к поиску следующего символа искомой строки (`"t"`):



Шаг 3: теперь мы указываем на символ `"t"` нашей искомой строки. У `currentNode` есть дочерний элемент `"t"`, поэтому переходим к нему (см. рис. на с. 347).

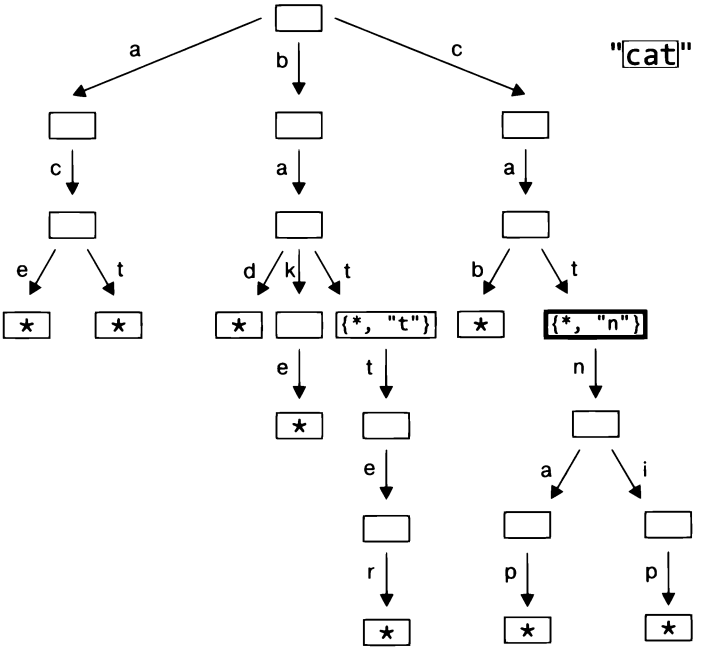
Мы достигли конца искомой строки, значит, нашли ее в префиксном дереве.

Программная реализация

Реализуем поиск в префиксном дереве, добавив метод `search` в ранее созданный класс `Trie`:

```
def search(self, word):
  currentNode = self.root

  for char in word:
    # Если у текущего узла есть дочерний ключ, соответствующий
    # текущему символу:
  if currentNode.children.get(char):
```




```
      # Переходим к этому дочернему узлу:
      currentNode = currentNode.children[char]
    else:
      # Если текущий символ не найден среди дочерних элементов
      # текущего узла, значит, искомого слова в дереве нет:
      return None

  return currentNode
```

Метод `search` принимает в качестве аргумента строку, отображающую искомое слово (или префикс).

Сначала устанавливаем корневой узел в качестве текущего `currentNode`:

```
currentNode = self.root
```

Затем перебираем все символы искомого слова:

```
for char in word:
```

В рамках каждой итерации цикла проверяем, есть ли у текущего узла дочерние элементы с текущим символом в качестве ключа. Если да, обновляем значение текущего узла, делая его этим дочерним узлом:

```
if currentNode.children.get(char):
  currentNode = currentNode.children[char]
```

Если нет, возвращаем `None`, обозначая, что мы зашли в тупик, а искомого слова в дереве нет.

Если мы дошли до конца цикла, значит, нашли в дереве нужное слово. В этом случае мы возвращаем `currentNode`. Возвращение текущего узла вместо `True` поможет при реализации автозаполнения — об этом чуть позже.

Эффективность поиска в префиксном дереве

Поиск в префиксном дереве невероятно эффективен.

Подсчитаем, сколько шагов выполняет наш алгоритм.

Итак, в процессе поиска мы последовательно фокусируемся на каждом символе искомой строки. При этом мы используем хеш-таблицу каждого узла, чтобы найти соответствующий дочерний узел за один шаг. Как вы знаете, поиск в хеш-таблице выполняется за время O(1). Получается, наш алгоритм выполняет столько шагов, сколько символов в искомой строке.

Такой поиск может быть намного быстрее, чем двоичный в упорядоченном массиве. Временная сложность двоичного поиска $O(\log N)$, где N — это число слов в словаре. При поиске же в префиксном дереве число шагов соответствует числу символов в искомой строке. Для слова вроде «cat» это всего три шага.

Выразить эффективность поиска в префиксном дереве с помощью O-нотации довольно сложно. Мы не можем сказать, что она равна O(1), так как число шагов непостоянно и зависит от длины искомой строки. Оценка $O(N)$ тоже может ввести в заблуждение, ведь под N обычно понимается количество элементов в структуре данных, а в нашем случае это число узлов в префиксном дереве, которое значительно превышает число символов в искомой строке.

Чаще всего временная сложность этого алгоритма выражается как $O(K)$, где K — число символов в искомой строке. Здесь для обозначения можно было использовать любую букву, кроме N, и была выбрана именно K.

Несмотря на то что временная сложность $O(K)$ непостоянна, так как размер искомой строки может меняться, в одном важном отношении категория $O(K)$ напоминает O(1). Производительность большинства алгоритмов с *не*посто-

янной временной сложностью зависит от числа имеющихся данных — по мере роста их объема (N) работа алгоритма замедляется. Но в случае с O(K) это не так. Наше дерево может значительно вырасти, но это никак не повлияет на скорость поиска в нем. При поиске строки из трех символов алгоритм O(K) всегда будет выполнять три шага вне зависимости от размера дерева. Единственное, что может повлиять на скорость его работы — размер входной строки, а не объем доступных данных, что делает алгоритм O(K) очень эффективным.

Хотя поиск — самая распространенная операция, выполняемая над префиксными деревьями, его трудно протестировать без заполнения дерева данными, поэтому давайте займемся вставкой.

Вставка значения в префиксное дерево

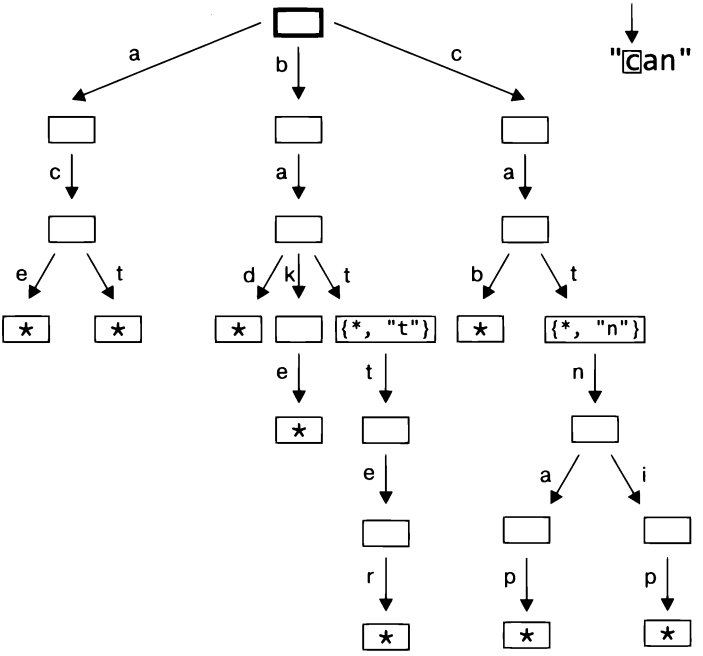
Операция вставки нового слова в префиксное дерево напоминает его поиск.

Алгоритм следующий.

1. Создаем переменную `currentNode`, которая изначально указывает на корневой узел.
2. Перебираем все символы вставляемой строки.
3. Последовательно указывая на каждый символ вставляемой строки, проверяем наличие у текущего узла `currentNode` дочернего элемента с соответствующим символом в качестве ключа.
4. Если он есть, обновляем значение `currentNode`, делая его этим дочерним элементом, а затем возвращаемся к шагу 2 и продолжаем перебирать символы вставляемой строки.
5. Если у текущего узла `currentNode` *нет* дочернего элемента с текущим символом в качестве ключа, создаем такой дочерний узел и обновляем значение `currentNode`, делая его этим дочерним узлом. Затем возвращаемся к шагу 2 и продолжаем перебирать символы вставляемой строки.
6. После вставки последнего символа нового слова добавляем последнему узлу дочерний элемент с ключом `"*"`, обозначающим конец слова.

Посмотрим на этот алгоритм в действии, вставив строку `"can"` в префиксное дерево из примера выше.

Настройка: задаем корень дерева в качестве текущего узла `currentNode` и указываем на первый символ вставляемой строки (`"c"`), как в верхнем правом углу на следующей схеме.

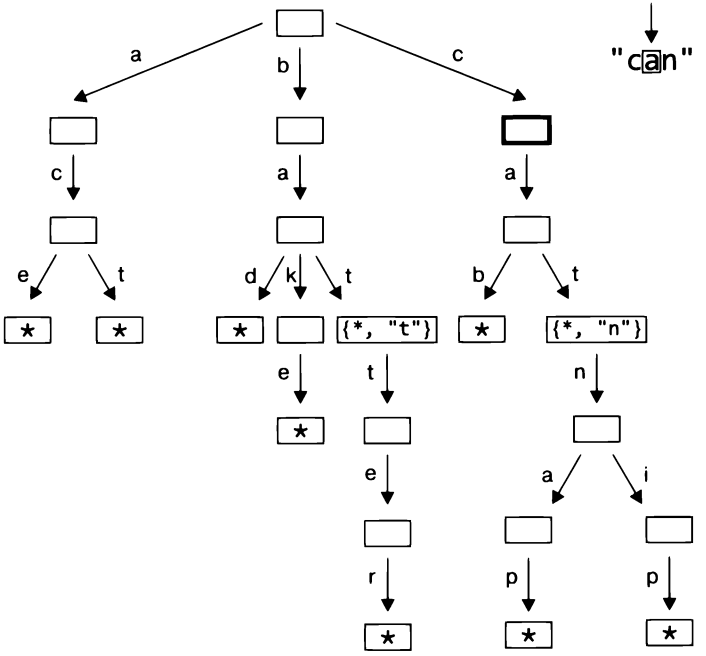


Шаг 1: так как дочерний ключ корневого узла — `"c"`, делаем `currentNode` значением этого ключа. Указываем на следующий символ (`"a"`) (с. 351, сверху).

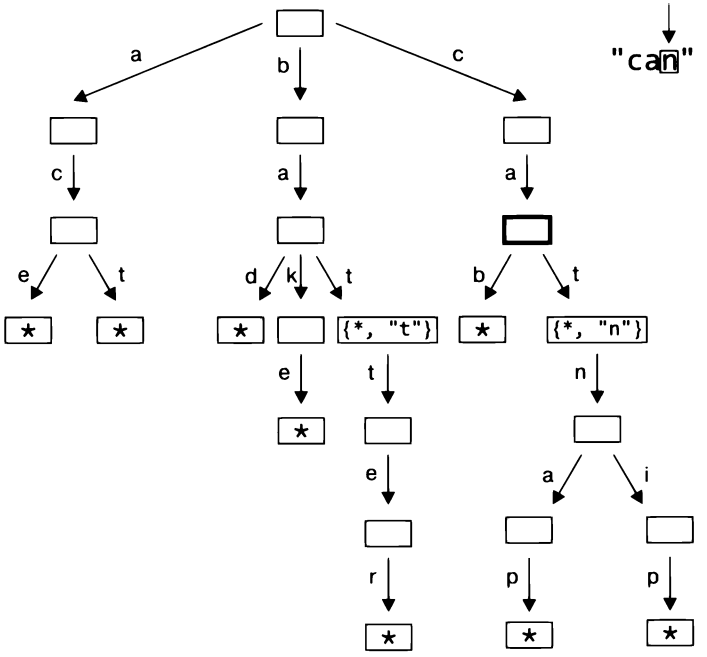
Шаг 2: проверяем `currentNode` на наличие дочернего элемента с ключом `"a"`. У него такой есть, поэтому делаем этот дочерний элемент новым текущим узлом `currentNode` и указываем на следующий символ нашей строки (`"n"`) (с. 351, снизу).

Шаг 3: у текущего узла `currentNode` *нет* дочернего элемента с ключом `"n"`, поэтому нам нужно его создать (с. 352, сверху) .

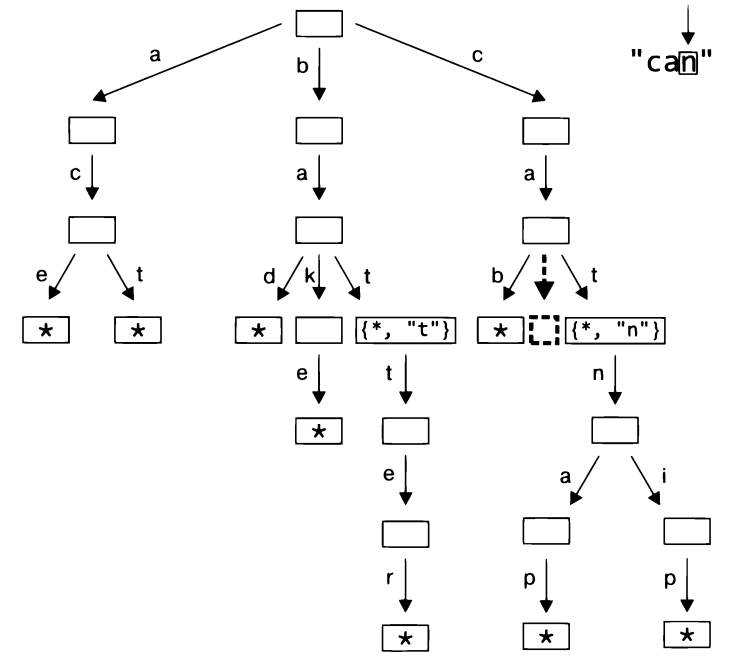
Шаг 4: мы завершили вставку строки `"can"` в префиксное дерево, поэтому добавляем ему дочерний элемент с ключом `"*"` (с. 352, снизу).


"can"
a
b
c
c
a
a
e
t
d
k
t
b
t
*
*
*
*
{*, "t"}
*
{*, "n"}
e
t
n
e
a
i
r
p
p
*
*
*

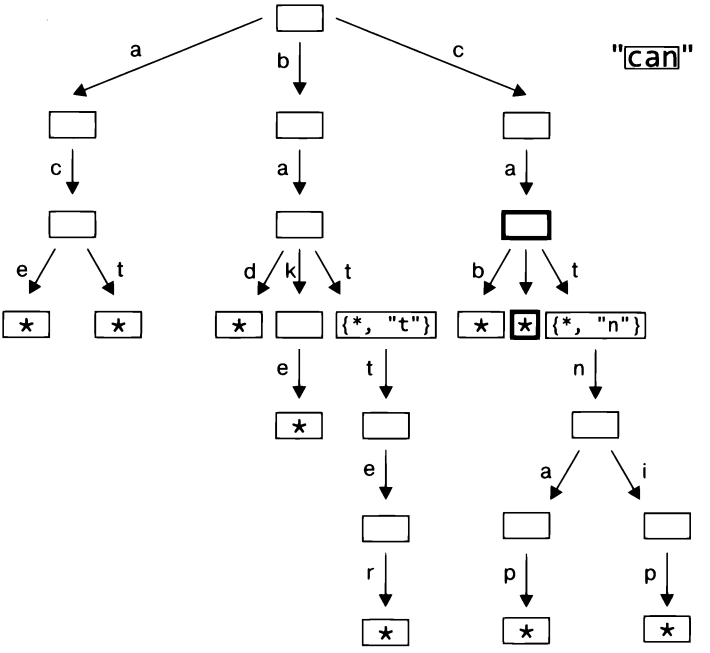
Шаг 1


"can"
a
b
c
c
a
a
e
t
d
k
t
b
t
*
*
*
*
{*, "t"}
*
{*, "n"}
e
t
n
e
a
i
r
p
p
*
*
*

Шаг 2



Шаг 3



Шаг 4

Готово!

Программная реализация

Ниже приведен код метода `insert` для нашего класса `Trie`. Как видите, он почти такой же, как код метода `search`:

```
def insert(self, word):
  currentNode = self.root

  for char in word:
    # Если у текущего узла есть дочерний ключ, соответствующий
    # текущему символу:
    if currentNode.children.get(char):
      # Переходим к этому дочернему узлу:
      currentNode = currentNode.children[char]
    else:
      # Если текущий символ не найден среди дочерних элементов
      # текущего узла, добавляем его в качестве нового дочернего узла:
      newNode = TrieNode()
      currentNode.children[char] = newNode

      # Переходим к этому узлу:
      currentNode = newNode

  # После вставки всего слова в префиксное дерево
  # добавляем в конце ключ *:
  currentNode.children["*"] = None
```

Первая часть кода этого метода схожа с кодом метода `search`. Разница проявляется, когда у текущего узла не оказывается дочернего элемента, соответствующего текущему символу. Тогда мы добавляем новую пару «ключ — значение» в хеш-таблицу текущего узла `currentNode`, где ключ — это текущий символ, а значение — новый узел `TrieNode`:

```
newNode = TrieNode()
currentNode.children[char] = newNode
```

Затем обновляем `currentNode`, делая его этим новым узлом:

```
currentNode = newNode
```

После этого выполняем итерации цикла, пока не завершим процесс вставки нового слова. В конце добавляем в хеш-таблицу конечного узла ключ `"*"` со значением `None`:

```
currentNode.children["*"] = None
```

Как и поиск, вставка в префиксное дерево занимает $O(K)$ времени. По сути, мы выполняем $K + 1$ шагов с учетом добавления ключа `"*"` в конце, но, так как мы отбрасываем константы, временная сложность алгоритма — $O(K)$.

Создание функции автозаполнения

Мы почти готовы реализовать полноценную функцию автозаполнения. Но, чтобы облегчить себе задачу, сначала создадим простой вспомогательный метод.

Собираем все слова

Следующий метод, который мы добавим в наш класс `Trie`, возвращает массив со *всеми* словами из префиксного дерева. Конечно, нам едва ли понадобится перечислять *все* слова из словаря, но мы сделаем так, чтобы этот метод принимал любой узел префиксного дерева в качестве аргумента и перечислял все слова, которые начинаются с этого узла.

Следующий метод, `collectAllWords`, составляет список из всех слов в префиксном дереве, начиная с указанного узла:

```
def collectAllWords(self, node=None, word="", words=[]):
  # Этот метод принимает три аргумента. Первый - это узел (node),
  # из значений потомков которого мы собираем слова.
  # Второй аргумент, word, - изначально пустая строка,
  # куда мы добавляем символы по мере продвижения по дереву.
  # Третий аргумент, words, - изначально пустой массив,
  # а по завершении работы функции будет содержать все слова
  # из префиксного дерева.
  # Текущий узел, current node, - это узел, переданный в качестве первого
  # параметра, или корневой узел, если конкретное значение не указано:
  currentNode = node or self.root

  # Перебираем все дочерние элементы текущего узла:
  for key, childNode in currentNode.children.items():
    # Если текущий ключ - *, значит, мы достигли конца
    # слова и можем добавить его в массив words:
    if key == "*":
      words.append(word)
    else: # Если мы все еще находимся в середине слова:
      # Рекурсивно вызываем эту функцию для дочернего узла
      self.collectAllWords(childNode, word + key, words)

  return words
```

Это рекурсивный метод, поэтому внимательно его разберем.

Метод принимает три основных аргумента: `node`, `word` и `words`. Первый позволяет указать узел префиксного дерева, с которого мы начнем сбор слов. Если мы не передадим этот аргумент, метод соберет все слова в дереве, начиная с корневого узла.

Аргументы `word` и `words` используются при рекурсивных вызовах метода, поэтому нам не нужно указывать их значения в начале. По умолчанию `words` — это

пустой массив. Мы добавляем в него строку после обнаружения полного слова в префиксном дереве, а по завершении работы функции этот массив будет возвращен.

Аргумент `word` — по умолчанию пустая строка, в которую мы добавляем новые символы по мере продвижения по древу. При обнаружении ключа `"*"` слово (`word`) считается завершенным и мы добавляем его в массив `words`.

Теперь разберем код построчно.

Сначала мы задаем значение текущего узла `currentNode`:

```
currentNode = node or self.root
```

Если мы не передадим методу конкретный параметр, то по умолчанию текущим узлом будет корень дерева. Давайте допустим, что `currentNode` действительно корневой узел.

Затем мы запускаем цикл, который перебирает все пары «ключ — значение» в дочерней хеш-таблице `currentNode`:

```
for key, childNode in currentNode.children.items():
```

В рамках каждой итерации цикла ключ (`key`) это всегда односимвольная строка, а значение (`childNode`) — экземпляр `TrieNode`.

Перейдем к условию `else`, где и происходит настоящее волшебство:

```
self.collectAllWords(childNode, word + key, words)
```

Эта строка кода рекурсивно вызывает функцию `collectAllWords` с дочерним узлом (`childNode`), передаваемым в качестве первого аргумента. Именно так мы обходим префиксное дерево и собираем слова в нем.

Второй аргумент, который мы передаем, `word + key`, позволяет прибавлять ключ к текущему слову, создавая итоговое слово по мере перехода от узла к узлу.

Третий аргумент — массив `words`. Передавая его при каждом рекурсивном вызове, мы можем помещать в него полные слова, создавая их список в процессе обхода префиксного дерева.

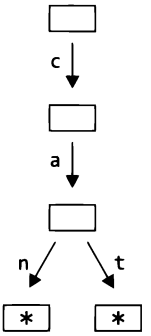
Базовый случай здесь — достижение ключа `"*"`, указывающего на завершение составления слова. На этом этапе мы можем добавить слово `word` в массив `words`:

```
if key == "*":
  words.append(word)
```

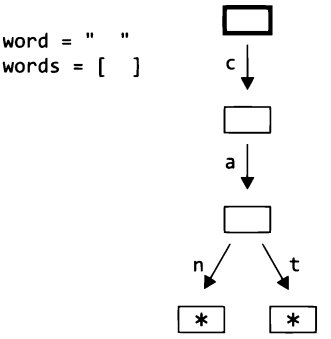
По завершении работы функции мы возвращаем этот массив. Если мы вызвали функцию, не указав конкретный узел в качестве аргумента, возвращенный массив будет содержать список всех слов в дереве.

Пошаговый разбор рекурсивных вызовов

Давайте посмотрим, как эта функция работает, на примере простого префиксного дерева с двумя словами: `"can"` и `"cat"`:

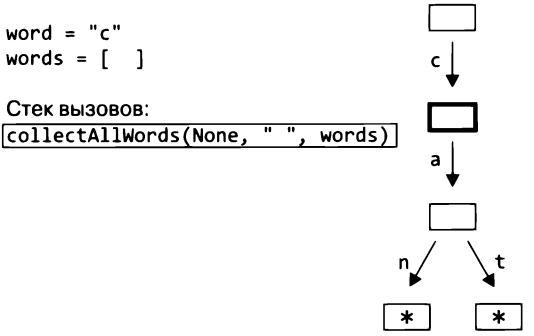


Вызов 1: при первом вызове функции `collectAllWords` в качестве текущего узла `currentNode` используется корень дерева, слово `word` — это пустая строка, и массив слов `words` тоже пуст:



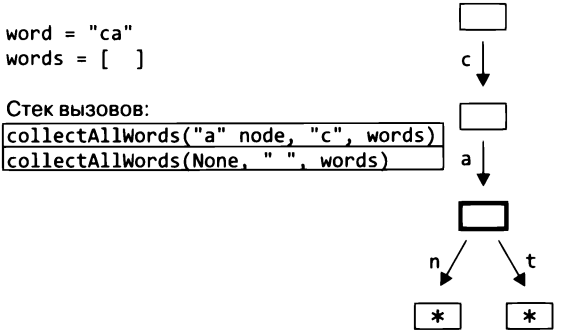
Перебираем дочерние элементы корневого узла. Здесь в нем только один дочерний ключ, `"c"`, который указывает на дочерний узел. Прежде чем рекурсивно вызвать `collectAllWords` для обработки этого узла, нужно добавить текущий вызов в стек вызовов.

Затем рекурсивно вызываем `collectAllWords` для дочернего узла `"c"`. В качестве аргумента `word` передаем `word + key` — строку `"c"` (результат прибавления ключа `"c"` к пустой строке `word`). Передаем все еще пустой массив `words`. Ниже показан результат этого рекурсивного вызова:



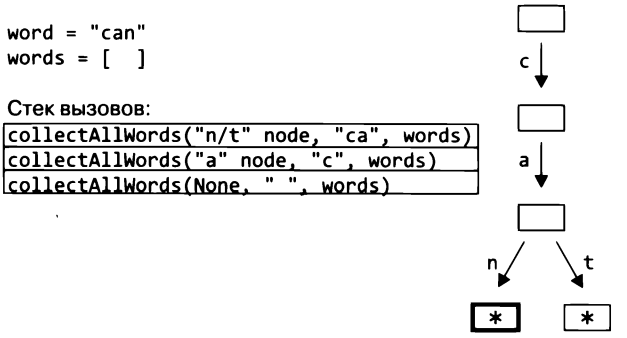
Вызов 2: перебираем дочерние элементы текущего узла. У него только один дочерний ключ, `"a"`. Прежде чем рекурсивно вызвать `collectAllWords` для этого узла, добавим текущий вызов в стек вызовов. На следующей схеме этот текущий узел обозначен как `"a"` — это значит, что узел `"a"` для него дочерний.

Теперь рекурсивно вызываем функцию `collectAllWords`, передав ей дочерний узел, строку `"ca"` (`word + key`) и по-прежнему пустой массив `words`:



Вызов 3: перебираем дочерние элементы текущего узла: здесь это `"n"` и `"t"`. Начнем с `"n"`, но перед выполнением любых рекурсивных вызовов добавляем текущий вызов в стек. На следующей диаграмме текущий узел обозначен как `"n/t"` — это значит, что узлы `"n"` и `"t"` для него дочерние.

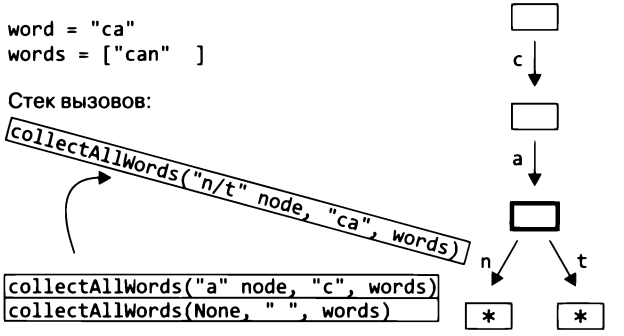
Затем вызываем `collectAllWords` для дочернего элемента `"n"`, передав `"can"` в качестве аргумента `word`, и пустой массив `words`:



Вызов 4: перебираем дочерние элементы текущего узла. У него только один дочерний элемент — `"*"`. Это базовый случай. Мы добавляем текущее значение `word`, строку `"can"`, в массив `words`:

```
words = ["can"]
```

Вызов 5: выталкиваем из стека вызовов верхний элемент, который соответствует рекурсивному вызову функции `collectAllWords` для узла с дочерними ключами `"n"` и `"t"` (где строка `word` была в значении `"ca"`). Значит, на этом этапе мы возвращаемся к этому вызову (как это обычно и бывает при извлечении элемента из стека):



Здесь нужно сделать одно важное замечание. В рамках текущего вызова значением строки `word` снова становится `"ca"`, так как именно оно было аргументом

word, переданным функции при вызове. Но теперь массив words содержит слово "can", несмотря на то что при первоначальном вызове он был пуст.

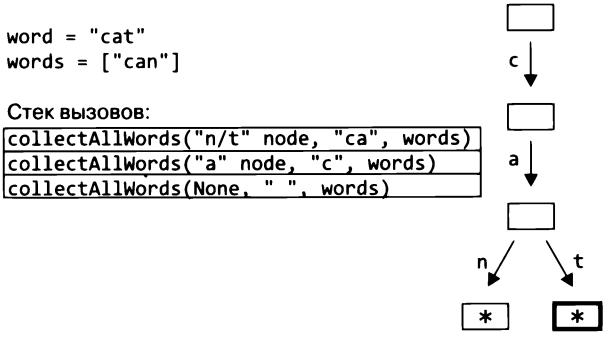
Так происходит потому, что многие языки программирования позволяют передавать массив вверх и вниз по стеку вызовов, так как он остается неизменным в памяти, даже после добавления новых значений (то же касается и хеш-таблиц, что и делало возможной их передачу с помощью мемоизации, которая обсуждалась в главе 12).

С другой стороны, при изменении строки компьютер фактически создает новую, вместо того чтобы изменять исходный строковый объект. Поэтому даже после обновления значения word с "ca" на "can" у прошлого вызова по-прежнему будет доступ только к исходной строке "ca" (в некоторых языках это может работать немного иначе, но для наших целей этого понимания вполне достаточно).

В любом случае мы находимся в рамках вызова, где массив words содержит слово "can", а значение строки word — это "ca".

Вызов 6: к этому моменту мы уже обработали в цикле ключ "n", поэтому теперь можем переходить к ключу "t". Перед выполнением рекурсивного вызова функции collectAllWords для дочернего узла "t" нужно снова добавить текущий вызов в стек. На самом деле это будет уже второй случай добавления этого вызова в стек. Мы уже выталкивали его, но теперь снова его туда помещаем.

При вызове функции collectAllWords для обработки дочернего элемента "t" мы передаем в качестве значения аргумента word строку "cat" (результат операции word + key) и массив words:



Вызов 7: перебираем дочерние элементы текущего узла. Его единственный дочерний элемент — `"*"`, поэтому добавляем текущее слово `"cat"` в массив `words`:

```
words = ["can", "cat"]
```

На этом этапе мы можем раскрутить стек вызовов, последовательно вытолкнув из него все вызовы и завершив их выполнение. Завершение последнего рекурсивного вызова, с которого и начался весь процесс, сопровождается возвращением массива `words`. Так как в нем есть строки `"can"` и `"cat"`, мы можем считать, что успешно собрали все слова из префиксного дерева.

Завершение функции автозаполнения

Итак, мы готовы реализовать функцию автозаполнения. На самом деле мы уже проделали бóльшую часть работы. Осталось только объединить все фрагменты.

Это базовый метод автозаполнения `autocomplete`, который мы добавляем в класс `Trie`:

```
def autocomplete(self, prefix):
  currentNode = self.search(prefix)
  if not currentNode:
    return None
  return self.collectAllWords(currentNode)
```

Вот и все. С помощью метода поиска (`search`) и сбора слов (`collectAllWords`) мы можем автоматически дополнить любой префикс. Вот как это работает.

Метод `autocomplete` принимает параметр `prefix` — строку символов, введенную пользователем.

Сначала ищем этот префикс в дереве. Если метод поиска не находит его, он возвращает `None`. Если находит — возвращает *узел префиксного дерева, соответствующий последнему символу этого префикса.*

Я уже говорил, что при обнаружении слова метод `search` мог бы просто возвращать `True`. Но мы сделали так, чтобы он возвращал последний узел, именно для того, чтобы в дальнейшем использовать его в реализации функции автозаполнения.

Далее метод `autocomplete` вызывает `collectAllWords` для обработки узла, возвращенного методом `search`. Это позволяет найти и собрать все слова, происходящие от конечного узла, которые могут быть добавлены к исходному префиксу.

Наконец, наш метод возвращает массив всех возможных окончаний для введенного префикса, которые мы затем можем предоставить пользователю.

Префиксные деревья с дополнительными значениями: улучшенная функция автозаполнения

Если подумать, хорошей функции автозаполнения вовсе не обязательно выводить *все* возможные варианты слов. Отображая, скажем, шестнадцать вариантов, мы рискуем перегрузить пользователя, поэтому правильнее было бы предоставить ему лишь самые популярные слова из списка.

Например, если пользователь вводит строку `"bal"`, то, скорее всего, хочет написать `"ball"`, `"bald"` или `"balance"`. Конечно, есть вероятность того, что он хочет ввести слово `"balter"` (которое означает импровизационный танец, если хотите знать), но она крайне мала, учитывая, что это *не* общеупотребительное слово.

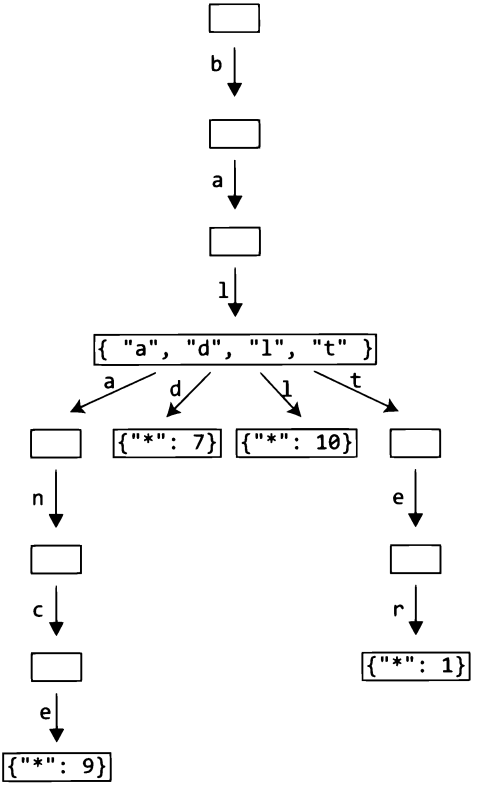
Чтобы отобразить самые популярные варианты слов, нам как-то нужно сохранить в префиксном дереве данные об их частотности. К счастью, для этого достаточно внести в дерево совсем небольшие изменения.

В текущей реализации префиксного дерева при каждой установке ключа `"*"` мы делали его значение нулевым, так как до этого нас интересовал только сам ключ, а не его значение.

Но мы можем использовать значения ключа для хранения дополнительных данных о словах, например об их популярности. Чтобы не усложнять, используем шкалу оценок от 1 до 10, где 1 — самое редкое слово, а 10 — самое распространенное.

Допустим, слово `"ball"` очень популярное, поэтому мы ставим ему 10. `"balance"` менее популярно, и мы ставим ему оценку 9, `"bald"` используется еще реже, поэтому получает 7, а малоизвестное слово `"balter"` — 1.

Мы можем сохранить эти оценки в префиксном дереве:



Используя число в качестве значения каждого ключа "*", мы можем хранить в префиксном дереве данные о популярности каждого слова. Теперь при сборе слов мы можем собрать и эти оценки и отсортировать слова в порядке уменьшения популярности, отобразив лишь несколько самых распространенных из них.

Выводы

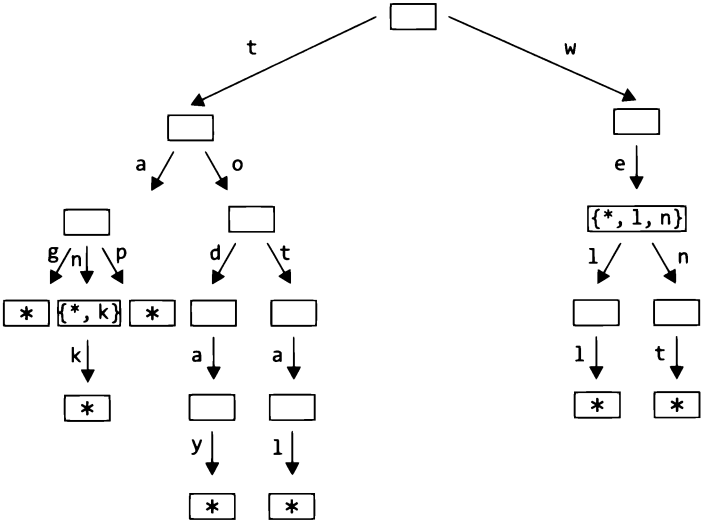
Итак, мы рассмотрели три типа деревьев: двоичные деревья поиска, кучи и префиксные деревья, но есть и *много* других: АВЛ-, красно-черные деревья, деревья 2-3-4 и т. д. У каждой из этих структур уникальные черты и поведение, которые будут особенно полезны в определенных ситуациях. С этими структурами данных вы можете познакомиться самостоятельно. В любом случае, вы уже понимаете, как разные деревья позволяют решать разные задачи.

Пришло время познакомиться с последней структурой данных в этой книге. Все, что вы узнали о деревьях, поможет вам разобраться с графами. Они бывают полезны в самых разных ситуациях, что и объясняет их популярность. Итак, приступим.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 17».

1. Перечислите все слова в следующем префиксном дереве:



2. Нарисуйте префиксное дерево, где хранятся следующие слова: `"get"`, `"go"`, `"got"`, `"gotten"`, `"hall"`, `"ham"`, `"hammer"`, `"hill"` и `"zebra"`.

3. Напишите функцию, которая обходит все узлы префиксного дерева и выводит на экран все ключи, в том числе и `"*"`.

4. Напишите функцию *автозамены*, которая помогает пользователю исправить опечатку, предложив правильное слово. Она должна принимать строку с текстом пользователя. Если пользовательской строки *нет* в префиксном дереве, функция должна вернуть альтернативное слово с самым длинным общим префиксом со строкой пользователя.

   Допустим, в нашем префиксном дереве есть слова `"cat"`, `"catnap"`, и `"catnip"`. Если пользователь случайно введет слово `"catnar"`, функция должна вернуть

"catnap", так как у этого слова самый длинный общий префикс со словом "catnar" — "catna", состоящий из пяти символов. Слово "catnip" подходит не так хорошо, так как у него более короткий четырехсимвольный общий префикс с "catnar" — "catn".

Вот еще один пример: если пользователь введет "caxasfdij", то функция может вернуть любое из слов: "cat", "catnap" или "catnip", поскольку у них всех один общий префикс с пользовательской строкой — "ca".

Если строка пользователя будет обнаружена в префиксном дереве, функция должна просто вернуть соответствующее слово. Это должно срабатывать, даже если пользовательская строка содержит неполное слово, так как мы пытаемся лишь исправить опечатку, а не предложить окончания для введенного префикса.

ГЛАВА 18

Отражение связей между объектами с помощью графов

Допустим, мы создаем социальную сеть, в которой люди могли бы заводить друзей. Дружеские отношения взаимные, поэтому если Алиса дружит с Бобом, то Боб тоже дружит с Алисой.

Как лучше организовать эти данные?

Один из простейших подходов — использование двумерного массива:

```
friendships = [
  ["Alice", "Bob"],
  ["Bob", "Cynthia"],
  ["Alice", "Diana"],
  ["Bob", "Diana"],
  ["Elise", "Fred"],
  ["Diana", "Fred"],
  ["Fred", "Alice"]
]
```

Каждый подмассив с парой имен здесь отражает «дружбу» между двумя людьми.

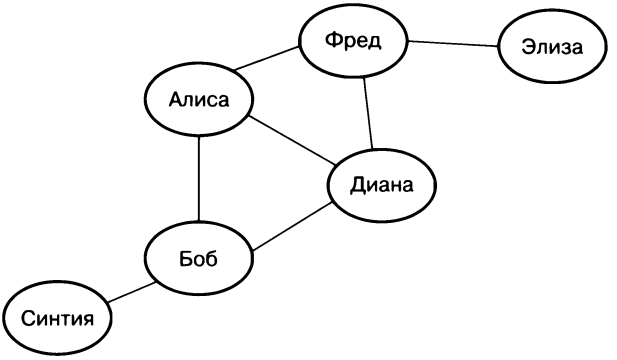
К сожалению, применение этого подхода не поможет нам быстро найти всех друзей Алисы. Если мы посмотрим внимательно, то увидим, что Алиса дружит с Бобом, Дианой и Фредом. Но, чтобы это определить, компьютеру пришлось бы проверить все подмассивы, так как имя Алисы может быть в любом из них. Это занимает время O(N), что очень медленно.

К счастью, мы можем сделать этот поиск *гораздо* эффективнее. С помощью *графа* мы можем найти всех друзей Алисы *за время* $O(1)$.

Графы

Граф — это структура данных, которая отражает связи между узлами, поэтому она отлично подходит для описания отношений.

Наша соцсеть в виде графа будет выглядеть так:



Каждый человек здесь представлен узлом, а каждая линия указывает на его дружбу с другим человеком. Взгляните на Алису, и вы увидите, что она дружит с Бобом, Дианой и Фредом, поскольку ее узел соединяется линиями с их узлами.

Графы и деревья

Как видите, графы похожи на деревья, о которых мы говорили последние несколько глав. Действительно, любое *дерево — это разновидность графа*. Обе эти структуры данных состоят из взаимосвязанных узлов.

Итак, в чем же разница между графом и деревом? Дело в том, что все деревья — это графы, но не все графы — деревья.

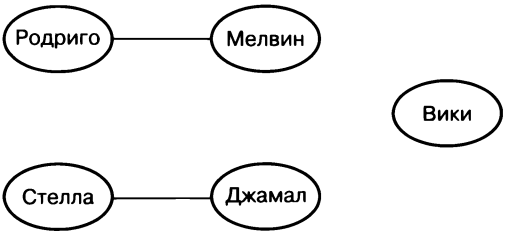
Чтобы граф считался деревом, в нем не должно быть *циклов* и все узлы должны быть *связаны*. Разберемся, что это значит.

В графе могут быть узлы, образующие так называемый *цикл* — узлы, циклически ссылающиеся друг на друга. В прошлом примере Алиса дружит с Дианой, Диана — с Бобом, а Боб — с Алисой. Эти три узла образуют цикл.

С другой стороны, деревьям «нельзя» содержать в себе циклы. Получается, если в графе есть цикл, то он не будет деревом.

Еще одна характерная черта деревьев в том, что каждый их узел как-то связан с каждым другим узлом, пусть даже косвенно. Но граф может не быть полностью связным.

Взгляните, к примеру, на этот:



В этой социальной сети две пары друзей, но они не дружат между собой. Кроме того, мы видим, что у Вики еще нет друзей: скорее всего, она зарегистрировалась совсем недавно. Но в деревьях не бывает узлов, не связанных с остальными.

Терминология графов

При описании графов используются особые термины. Мы привыкли называть каждый фрагмент данных *узлом*, но в случае с графом узел называется *вершиной*, а линии между узлами, то есть вершинами, — *ребрами*. Вершины, соединенные ребром, называются *смежными*, или *соседями*.

Итак, в нашем первом графе вершины «Алиса» и «Боб» смежные, так как у них общее ребро.

Как я уже говорил, граф может содержать вершину, не связанную с другими. Но граф, *все* вершины которого как-то связаны, называется *связным*.

Простейшая реализация графа

Ради лучшей организации кода мы будем использовать для представления наших графов объектно-ориентированные классы, но отмечу, что элементарный граф можно реализовать и с помощью хеш-таблицы (см. главу 8). Вот простейшая реализация нашей социальной сети с помощью хеш-таблицы на языке Ruby:

```
friends = {
  "Alice" => ["Bob", "Diana", "Fred"],
  "Bob" => ["Alice", "Cynthia", "Diana"],
  "Cynthia" => ["Bob"],
  "Diana" => ["Alice", "Bob", "Fred"],
  "Elise" => ["Fred"],
  "Fred" => ["Alice", "Diana", "Elise"]
}
```

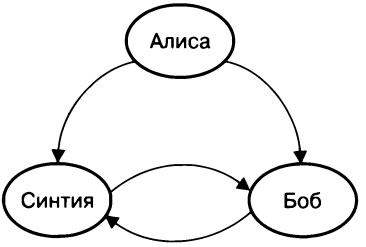
Найти друзей Алисы в таком графе мы можем за O(1) времени, так как чтение значения любого ключа из хеш-таблицы выполняется за один шаг:

```
friends["Alice"]
```

Этот фрагмент кода немедленно возвращает массив с именами всех друзей Алисы.

Ориентированные графы

В некоторых соцсетях отношения участников могут быть *не*взаимны. Например, социальная сеть может позволить Алисе подписаться на Боба, но Бобу не обязательно при этом подписываться на Алису. Чтобы отразить этот тип связи, построим новый граф:



Такой граф называется *ориентированным*. Здесь стрелки указывают *направление* связи: Алиса подписана на Боба и Синтию, но никто не подписан на Алису. Еще мы видим, что Боб и Синтия подписаны друг на друга.

Для хранения этих данных мы все еще можем использовать простую реализацию на основе хеш-таблицы:

```
followees = {
  "Alice" => ["Bob", "Cynthia"],
  "Bob" => ["Cynthia"],
  "Cynthia" => ["Bob"]
}
```

Единственная разница лишь в том, что мы используем массивы для указания имен людей, на которых подписан тот или иной человек.

Объектно-ориентированная реализация графа

Выше я показал вариант реализации графа на основе хеш-таблицы, но в дальнейшем мы будем использовать для этого объектно-ориентированный подход.

Вот начало объектно-ориентированной реализации графа на языке Ruby:

```
class Vertex
  attr_accessor :value, :adjacent_vertices

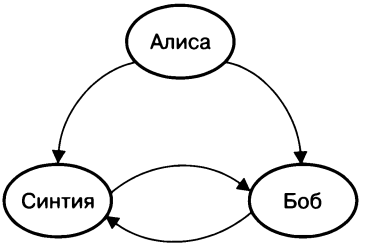
  def initialize(value)
    @value = value
    @adjacent_vertices = []
  end

  def add_adjacent_vertex(vertex)
    @adjacent_vertices << vertex
  end
end
```

У класса `Vertex` два основных атрибута: значение (`value`) и массив смежных вершин (`adjacent_vertices`). В нашем примере каждая вершина — это человек, а значение — строка с его именем. В более сложном приложении внутри вершины хранилось бы несколько элементов данных, например дополнительная информация из профиля пользователя.

Массив `adjacent_vertices` содержит все вершины, с которыми соединена выбранная. Мы можем добавить к ней новую смежную вершину с помощью метода `add_adjacent_vertex`.

Так можно использовать этот класс для построения ориентированного графа, который отражает, кто из участников на кого подписан:




```
alice = Vertex.new("alice")
bob = Vertex.new("bob")
cynthia = Vertex.new("cynthia")

alice.add_adjacent_vertex(bob)
alice.add_adjacent_vertex(cynthia)
bob.add_adjacent_vertex(cynthia)
cynthia.add_adjacent_vertex(bob)
```

Если бы мы создавали *неориентированный* граф для соцсети (где все дружеские отношения взаимны), то при добавлении Боба в список друзей Алисы логично было бы автоматически добавить Алису в список друзей Боба.

Для этого можно изменить метод `add_adjacent_vertex` так:

```
def add_adjacent_vertex(vertex)
  return if adjacent_vertices.include?(vertex)
  @adjacent_vertices << vertex
  vertex.add_adjacent_vertex(self)
end
```

Допустим, мы применяем этот метод к вершине Алисы и добавляем Боба в список ее друзей. Как и в прошлой версии, с помощью кода `@adjacent_vertices << vertex` мы добавляем Боба в список смежных вершин Алисы. Но затем мы применяем этот же метод к вершине Боба с помощью фрагмента кода `vertex.add_adjacent_vertex(self)`, добавляя так Алису в список друзей Боба.

Чтобы этот цикл не выполнялся вечно, мы добавили строку кода `return if adjacent_vertices.include?(vertex)`, которая прекращает работу метода, если, например, Боб уже есть в списке друзей Алисы.

Список смежности и матрица смежности

Для хранения соседей каждой вершины в нашей реализации графа используется простой список (в виде массива), который называется *списком смежности*.

Но есть и другой способ реализации, где вместо списков используются двумерные массивы. Этот подход, основанный на использовании *матриц смежности*, может пригодиться в определенных ситуациях.

Оба метода весьма популярны, но я решил использовать список смежности, потому что считаю его более понятным. Но все же рекомендую вам познакомиться и с матрицей — этот способ тоже очень полезен и довольно интересен для изучения.

Чтобы нам было проще работать, в дальнейшем мы будем использовать связные графы (вершины которых как-то связаны друг с другом). Так мы сможем использовать описанный выше класс `Vertex` для реализации всех алгоритмов. Суть в том, что при наличии доступа лишь к одной вершине мы можем найти остальные, так как все они связаны между собой.

Но помните, что, если мы имеем дело с несвязным графом, то обнаружить все вершины при наличии доступа только к одной из них будет невозможно. В этом

случае нужно хранить все вершины графа в какой-то дополнительной структуре данных, например в массиве, чтобы сохранить доступ к каждой из них (для хранения такого массива в реализациях графа часто используется отдельный класс `Graph`).

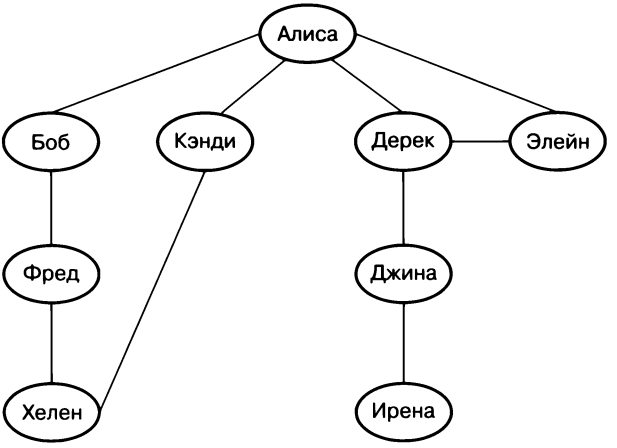
Поиск по графу

Одна из самых распространенных операций над графом — поиск определенной вершины.

Важно отметить, что при работе с графами у термина «поиск» может быть несколько значений. Самое простое — это нахождение конкретной вершины графа. Этот процесс напоминает поиск значения в массиве или пары «ключ — значение» в хеш-таблице.

Но применительно к графам у этого термина обычно более конкретное значение: *если у нас есть доступ к одной вершине графа, мы должны найти другую конкретную вершину, которая как-то связана с исходной.*

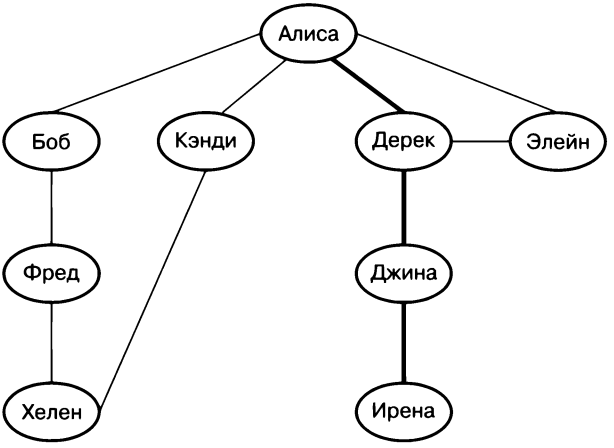
Рассмотрим пример социальной сети:



Допустим, у нас есть доступ к вершине Алисы. Если бы мы собрались искать Ирену, это означало бы, что мы попытаемся найти путь от Алисы к Ирене.

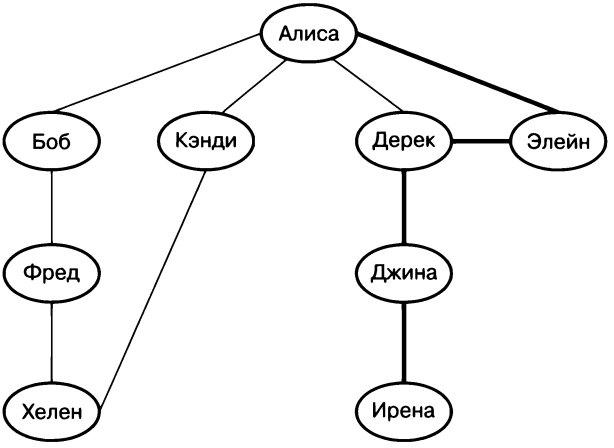
Как видите, *путей* от Алисы до Ирены два.

Более короткий довольно очевиден:



Мы можем добраться от Алисы до Ирены, переходя от вершины к вершине в такой последовательности: Алиса → Дерек → Джина → Ирена.

Но мы можем выбрать и более длинный путь:



Он выглядит так: Алиса → Элейн → Дерек → Джина → Ирена.

В теории графов «путь» — это конкретная последовательность ребер, позволяющая перейти от одной вершины к другой.

Поиск по графу (переход от одной вершины к другой) может пригодиться во многих случаях.

Самая очевидная цель выполнения такой операции — нахождение конкретной вершины связного графа. В этом случае поиск может использоваться для обнаружения *любой* вершины графа, даже если доступ есть только к одной из них.

Поиск по графу можно провести и для проверки наличия связи между двумя вершинами. Например, чтобы узнать, связаны ли как-то вершины Алисы и Ирены в этой сети, мы можем воспользоваться поиском.

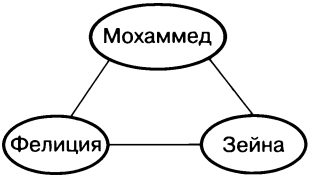
Поиск полезен не только для обнаружения какой-то конкретной вершины, но и для обхода графа, например чтобы выполнить операцию над каждой из его вершин. Чуть позже мы поговорим о том, как это работает.

Поиск в глубину

Есть два основных подхода к поиску по графу: *в глубину* и *в ширину*. Оба направлены на решение какой-то проблемы, но у каждого есть свои уникальные преимущества в определенных ситуациях. Начнем с рассмотрения поиска в глубину (DFS, depth first search), потому что этот процесс очень напоминает алгоритм обхода двоичного дерева (см. главу 15). По сути, этот же алгоритм был описан и в главе 10.

Как я уже говорил, поиск по графу можно выполнять для обнаружения конкретной вершины или просто для их обхода. Мы начнем с обсуждения алгоритма обхода графа, так как он немного проще.

Ключевая часть любого алгоритма поиска по графу — отслеживание уже посещенных вершин, без которого мы рискуем застрять в бесконечном цикле. Рассмотрим следующий граф:



Здесь Мохаммед дружит с Фелицией, которая дружит с Зейной, которая дружит с Мохаммедом. Если мы не будем отслеживать посещенные вершины, то наш код будет выполняться вечно.

Такой проблемы не возникало при обходе деревьев (или файловой системы), так как в деревьях нет циклов. А в графе циклы *могут* быть, поэтому лучше позаботиться об этой проблеме заранее.

Один из способов отслеживания посещенных вершин — использование хеш-таблицы. По мере обхода графа мы добавляем посещенную вершину (или ее значение) в хеш-таблицу в качестве ключа и присваиваем ему произвольное значение, например логическое `true`. Если вершина в хеш-таблице есть, значит, мы уже ее посещали.

Итак, алгоритм поиска в глубину предполагает выполнение следующих шагов.

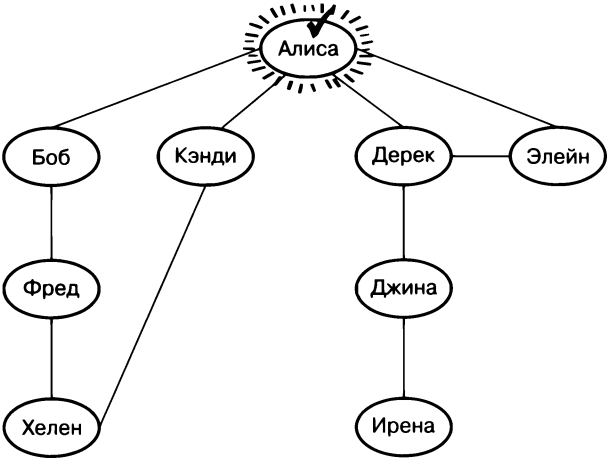
1. Начинаем со случайной вершины графа.
2. Добавляем текущую вершину в хеш-таблицу, чтобы отметить ее как посещенную.
3. Перебираем соседей текущей вершины.
4. Игнорируем смежные вершины, которые уже были посещены.
5. Если смежная вершина *не была* посещена, рекурсивно выполняем поиск в глубину, начиная с нее.

Пошаговый разбор поиска в глубину

Посмотрим на этот алгоритм в действии.

Начнем с вершины Алисы. На следующих схемах текущая вершина окружена штрихами, а галочка означает, что мы отметили вершину как посещенную (и добавили ее в хеш-таблицу).

Шаг 1: ставим галочку на вершине Алисы:



Далее с помощью цикла перебираем соседей Алисы: Боба, Кэнди, Дерека и Элейн.

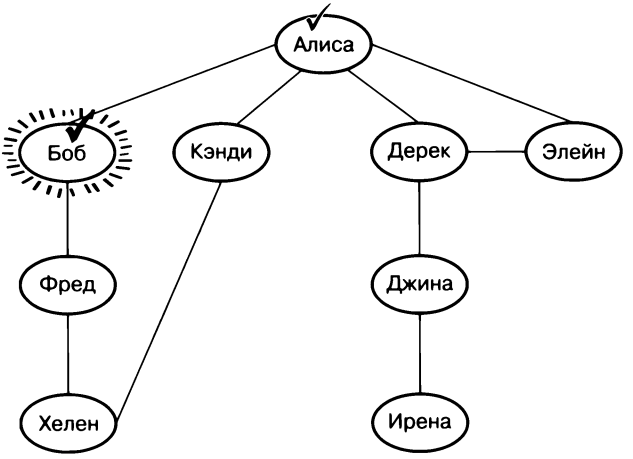
Порядок посещения соседей не важен, поэтому начнем с Боба. Он кажется довольно милым.

Шаг 2: выполняем поиск в глубину для Боба. Обратите внимание, что при этом мы совершаем рекурсивный вызов, поскольку уже находимся в процессе поиска в глубину для Алисы.

Как и при любой рекурсии, компьютеру нужно отслеживать все незавершенные вызовы функции, поэтому он сначала добавляет вершину Алисы в стек вызовов:



Теперь мы можем начать поиск в глубину для Боба, сделав его вершину текущей и отметив ее как посещенную:

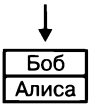


Затем перебираем соседей Боба: Алису и Фреда.

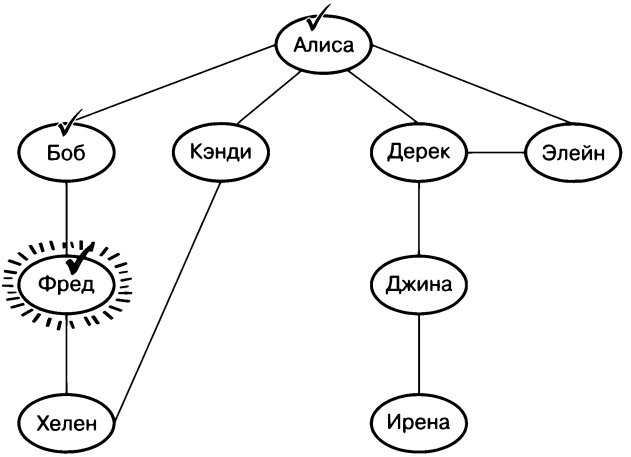
Шаг 3: вершину Алисы мы уже посетили, поэтому можем ее проигнорировать.

Шаг 4: выходит, единственный сосед Боба — это Фред, поэтому применяем функцию поиска в глубину к вершине Фреда. Сначала компьютер добавляет

вершину Боба в стек вызовов, чтобы не забыть о том, что он все еще находится в процессе поиска в глубину для Боба:



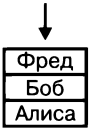
Теперь мы можем выполнить поиск в глубину для Фреда, сделав его вершину текущей и отметив ее как посещенную:



Перебираем соседей Фреда: Боба и Хелен.

Шаг 5: вершину Боба мы уже посетили, поэтому можем ее проигнорировать.

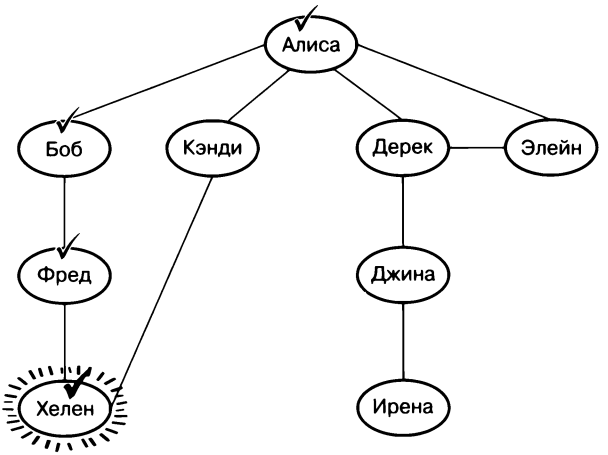
Шаг 6: единственный еще не посещенный сосед Фреда — это Хелен, поэтому мы применяем функцию поиска в глубину к вершине Хелен и компьютер добавляет вершину Фреда в стек вызовов:



Теперь мы выполняем поиск в глубину для Хелен, сделав ее вершину текущей и отметив ее как посещенную (верхний рисунок на с. 377).

Соседи Хелен — Фред и Кэнди.

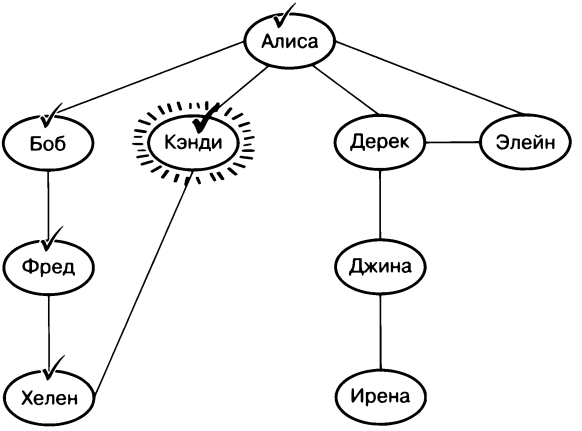
Шаг 7: вершину Фреда мы уже посетили, поэтому можем ее проигнорировать.



Шаг 8: вершина Кэнди еще *не* была посещена, поэтому рекурсивно применяем к ней функцию поиска в глубину и компьютер добавляет вершину Хелен в стек вызовов:



Теперь выполняем поиск в глубину для Кэнди, сделав ее вершину текущей и отметив ее как посещенную:



Соседи Кэнди — Алиса и Хелен.

Шаг 9: вершину Алисы мы уже посетили, поэтому можем ее проигнорировать.

Шаг 10: вершину Хелен мы тоже уже посетили, поэтому игнорируем и ее.

У вершины Кэнди нет других соседей, поэтому процесс поиска в глубину для нее завершен и компьютер начинает раскрутку стека вызовов.

Сначала он выталкивает из стека вершину Хелен. Мы уже перебрали всех ее соседей, так что поиск в глубину для нее тоже завершен.

Затем компьютер извлекает из стека вершину Фреда, соседей которого мы тоже перебрали.

Потом — вершину Боба, с которым мы тоже разобрались.

Теперь компьютер выталкивает из стека вызовов вершину Алисы. В процессе поиска для Алисы мы перебираем в цикле ее соседей. С Бобом (шаг 2) мы уже закончили, поэтому у нас остаются Кэнди, Дерек и Элейн.

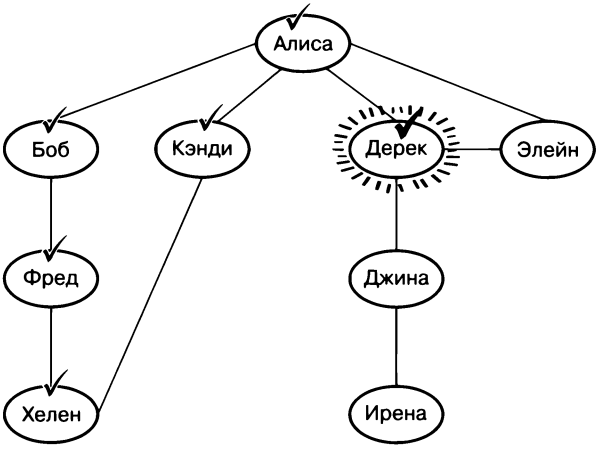
Шаг 11: вершину Кэнди мы уже посетили, поэтому нам не нужно выполнять поиск для нее.

Но мы еще не посетили вершины Дерека или Элейн.

Шаг 12: рекурсивно применяем функцию поиска в глубину к вершине Дерека. При этом компьютер снова добавляет вершину Алисы в стек вызовов:



Выполняем поиск в глубину относительно вершины Дерека, сделав ее текущей и отметив ее как посещенную:



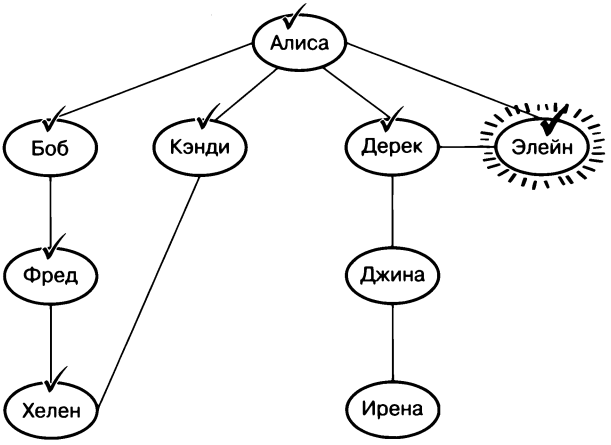
Соседи Дерека — Алиса, Элейн и Джина.

Шаг 13: вершину Алисы мы уже посетили, поэтому можем ее проигнорировать.

Шаг 14: посетим вершину Элейн, рекурсивно применив к ней функцию поиска в глубину. Перед этим компьютер добавляет вершину Дерека в стек вызовов:



Теперь выполняем поиск в глубину для Элейн, отметив ее вершину как посещенную:



Соседи Элейн — Алиса и Дерек.

Шаг 15: вершину Алисы мы уже посетили, поэтому можем ее проигнорировать.

Шаг 16: вершину Дерека мы тоже посетили.

Мы перебрали всех соседей Элейн, поэтому поиск для нее завершен. Компьютер извлекает вершину Дерека из стека вызовов и перебирает его еще не посещенных соседей. У нас единственная такая соседка — Джина.

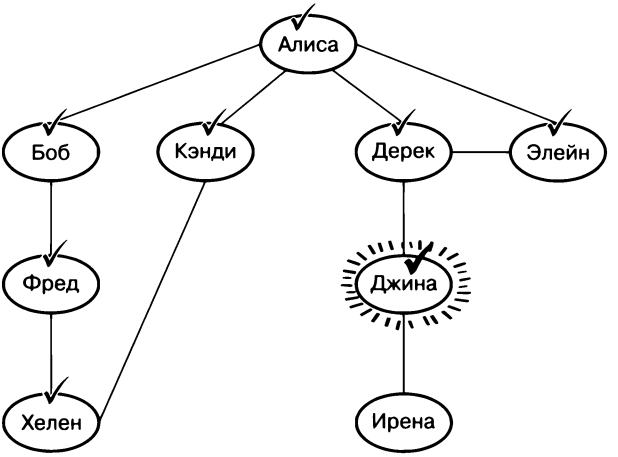
Шаг 17: мы еще не посещали эту вершину, поэтому рекурсивно применяем к ней функцию поиска в глубину. Перед этим компьютер снова добавляет вершину Дерека в стек вызовов:



Теперь выполняем поиск в глубину для Джины, отметив ее вершину как посещенную (см. следующее изображение).

Соседи Джины — Дерек и Ирена.

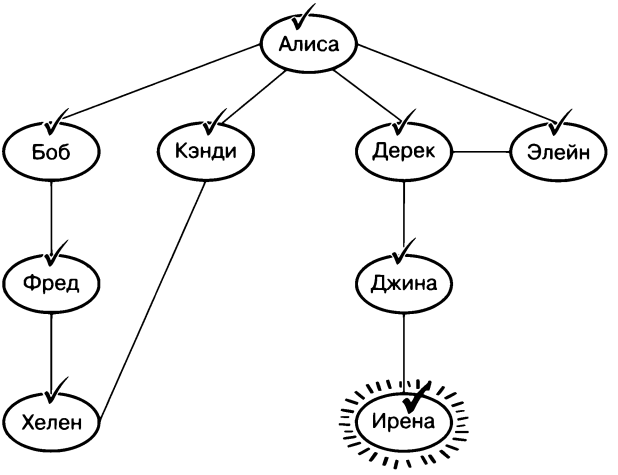
Шаг 18: Дерека мы уже посещали.



Шаг 19: у Джины осталась одна еще не посещенная соседка — Ирена. Компьютер добавляет в стек вызовов вершину Джины и рекурсивно применяет функцию поиска в глубину к вершине Ирены:



Выполняем поиск в глубину для Ирены, отметив ее вершину как посещенную:



Перебираем соседей Ирены. Ее единственная соседка — Джина.

Шаг 20: Джину мы уже посетили.

Теперь компьютер раскручивает стек вызовов, последовательно выталкивая вершины из него. Но все соседи этих вершин уже были посещены, поэтому компьютеру больше нечего делать с каждой из них.

Значит, процесс поиска завершен!

Программная реализация

Так выглядит реализация алгоритма поиска в глубину:

```
def dfs_traverse(vertex, visited_vertices={})
  # Отмечаем вершину как посещенную, добавляя ее в хеш-таблицу:
  visited_vertices[vertex.value] = true

  # Выводим значение вершины, чтобы убедиться
  # в работоспособности алгоритма:
  puts vertex.value

  # Перебираем соседей текущей вершины:
  vertex.adjacent_vertices.each do |adjacent_vertex|

    # Игнорируем соседнюю вершину, если мы ее уже посещали:
    next if visited_vertices[adjacent_vertex.value]

    # Рекурсивно вызываем этот метод для обработки соседней вершины:
    dfs_traverse(adjacent_vertex, visited_vertices)
  end
end
```

Метод `dfs_traverse` принимает одну вершину (`vertex`) и хеш-таблицу для хранения посещенных вершин (`visited_vertices`). При первом вызове этой функции хеш-таблица будет пустой, но по мере посещения вершин мы заполняем ее и передаем при каждом рекурсивном вызове.

Сначала отмечаем текущую вершину как посещенную, добавляя ее значение в хеш-таблицу:

```
visited_vertices[vertex.value] = true
```

Теперь выводим значение вершины, чтобы подтвердить факт ее посещения:

```
puts vertex.value
```

Далее мы перебираем всех соседей текущей вершины:

```
vertex.adjacent_vertices.each do |adjacent_vertex|
```

Если какая-то из соседних вершин уже была посещена — просто переходим к следующей итерации цикла:

```
next if visited_vertices[adjacent_vertex.value]
```

Если нет — рекурсивно вызываем функцию `dfs_traversal` для обработки соседней вершины:

```
dfs_traverse(adjacent_vertex, visited_vertices)
```

При этом мы передаем и хеш-таблицу `visited_vertices`, открывая следующему вызову к ней доступ. Для поиска конкретной вершины используем модифицированную версию прошлой функции:

```
def dfs(vertex, search_value, visited_vertices={})
  # Возвращаем исходную вершину,
  # если она - та, которую мы ищем:
  return vertex if vertex.value == search_value

  visited_vertices[vertex.value] = true

  vertex.adjacent_vertices.each do |adjacent_vertex|
    next if visited_vertices[adjacent_vertex.value]

    # Если соседняя вершина - та, которую мы ищем,
    # возвращаем ее:
    return adjacent_vertex if adjacent_vertex.value == search_value

    # Пытаемся найти искомую вершину, рекурсивно вызывая
    # метод для обработки соседней вершины:
    vertex_were_searching_for =
      dfs(adjacent_vertex, search_value, visited_vertices)

    # Если с помощью рекурсии выше мы смогли найти искомую вершину,
    # возвращаем ее:
    return vertex_were_searching_for if vertex_were_searching_for
  end

  # Если мы не нашли искомую вершину:
  return nil
end
```

Эту реализацию можно рекурсивно вызывать для обработки каждой вершины, но если она находит искомую (`vertex_were_searching_for`), то возвращает ее.

Поиск в ширину

Поиск в ширину (BFS, breadth-first search) — это еще один способ поиска по графу. В отличие от поиска в глубину, он *не* использует рекурсию, а полагается на уже известную нам очередь. Как вы помните, очередь — это структура данных, которая работает по принципу FIFO: элемент, добавленный в нее раньше всех, извлекается из нее первым.

Как и поиск в глубину, поиск в ширину предполагает *обход* графа — посещение каждой из его вершин.

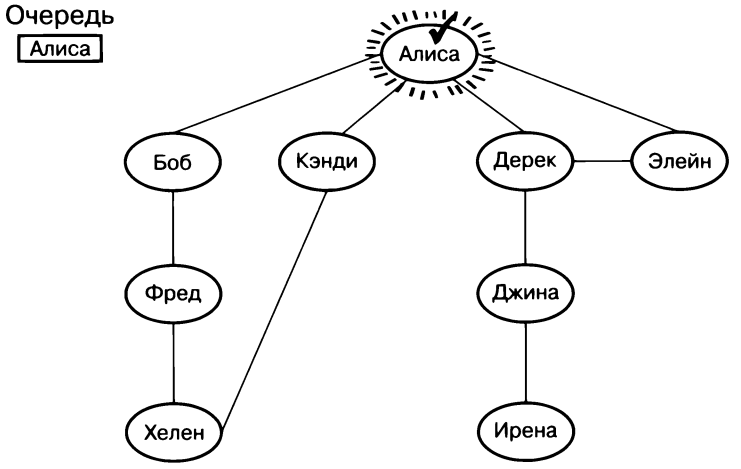
Этот алгоритм состоит из следующих шагов.

1. Выбираем любую вершину графа. Называем ее начальной.
2. Добавляем ее в хеш-таблицу, чтобы отметить как посещенную.
3. Добавляем начальную вершину в очередь.
4. Запускаем цикл, который выполняется, пока очередь не опустеет.
5. В рамках этого цикла удаляем из очереди первую вершину, которую назовем текущей.
6. Перебираем всех соседей текущей вершины.
7. Если соседняя вершина уже посещалась, игнорируем ее.
8. Если *нет* — отмечаем ее как посещенную, добавляя в хеш-таблицу, и ставим в очередь.
9. Повторяем эти шаги (начиная с 4), пока очередь не опустеет.

Пошаговый разбор поиска в ширину

Все не так сложно. Вы убедитесь в этом, как только мы пошагово разберем алгоритм выше.

Начнем с Алисы, выбрав ее вершину в качестве начальной. Первым делом отмечаем ее как посещенную и добавляем в очередь:

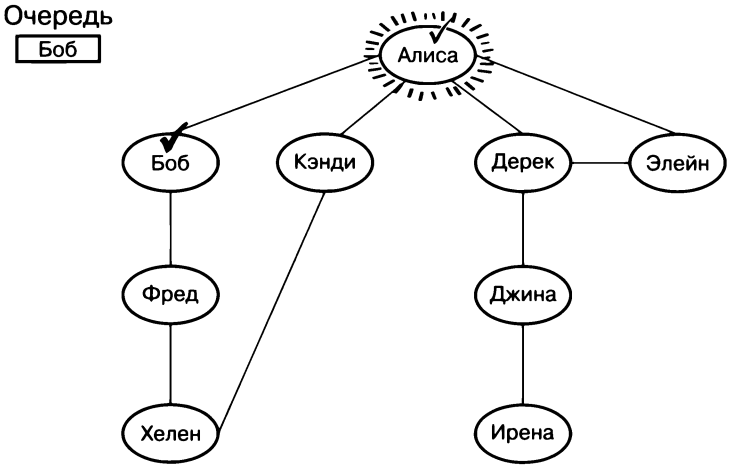


Теперь переходим к самому алгоритму.

Шаг 1: удаляем первую вершину из очереди, делая ее текущей. У нас это вершина Алисы, так как сейчас она — *единственный* элемент в очереди. Итак, наша очередь пуста.

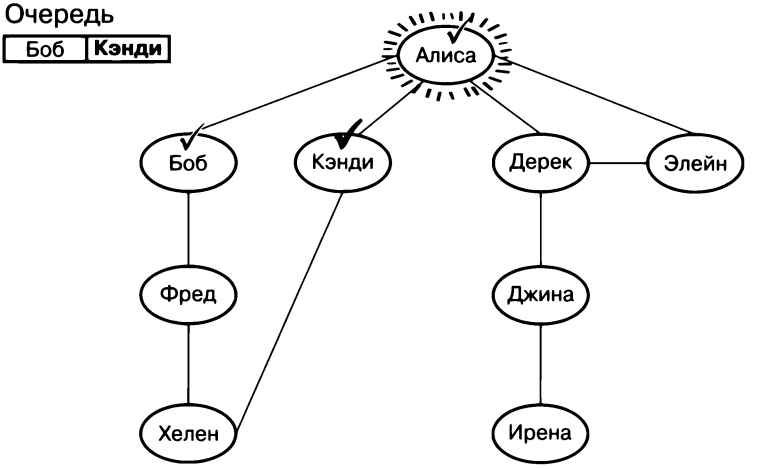
Поскольку вершина Алисы текущая, перебираем ее соседей.

Шаг 2: начинаем с Боба, отмечая его вершину как посещенную и добавляя в очередь.

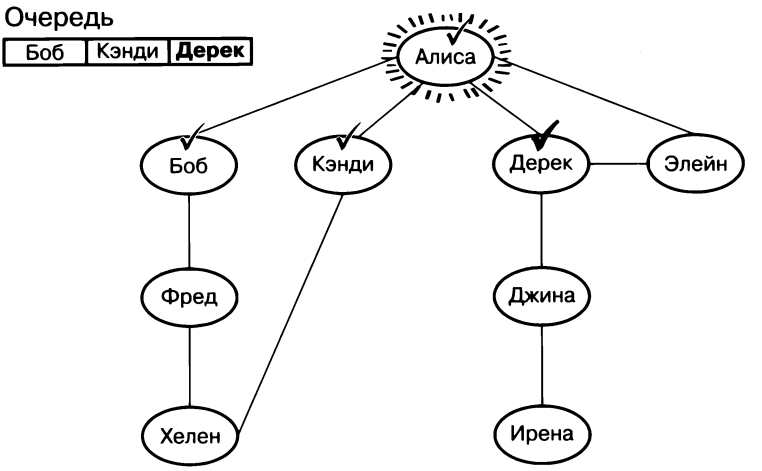


Обратите внимание, что вершина Алисы *все еще текущая* (на это указывают окружающие ее штрихи). Но мы все равно отметили вершину Боба как посещенную и добавили ее в очередь.

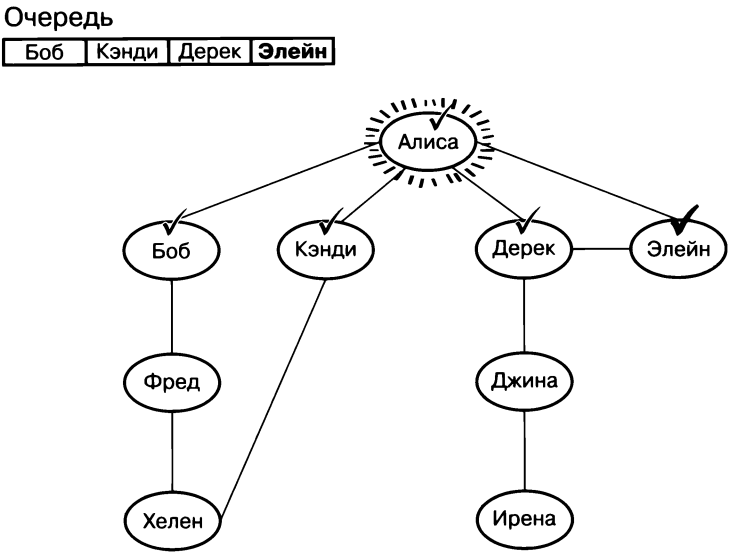
Шаг 3: выбираем другого соседа Алисы. Пусть это будет Кэнди. Отмечаем ее вершину как посещенную и добавляем ее в очередь.



Шаг 4: отмечаем вершину Дерека как посещенную и добавляем ее в очередь:



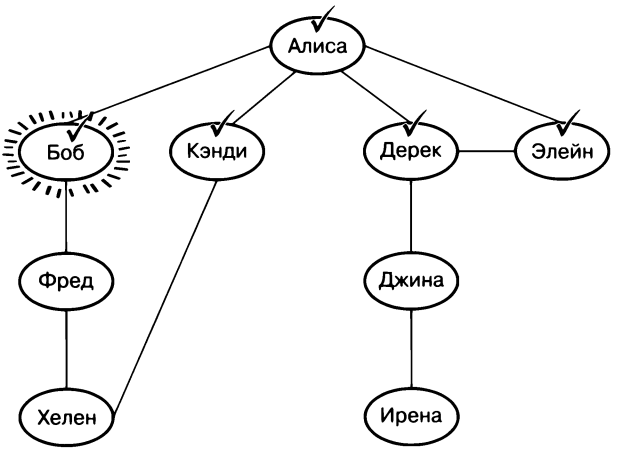
Шаг 5: делаем то же самое с вершиной Элейн:



Шаг 6: теперь, когда мы перебрали всех соседей текущей вершины (Алисы), удаляем первый элемент из очереди и делаем его текущей вершиной. У нас это вершина Боба, поэтому мы исключаем ее из очереди и делаем текущей:

Очередь

Кэнди	Дерек	Элейн



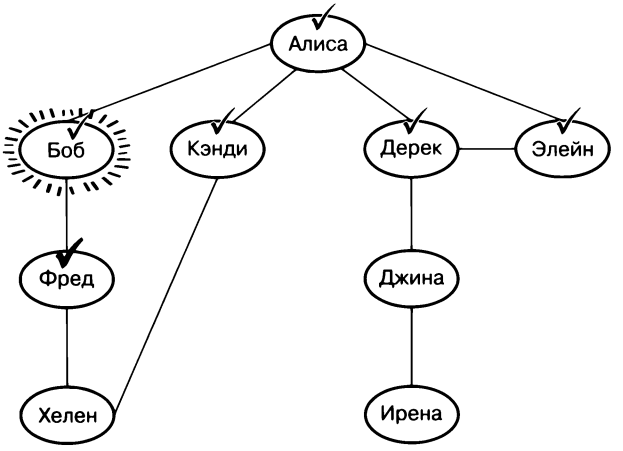
Так как вершина Боба текущая, перебираем всех его соседей.

Шаг 7: Алису мы уже посетили, поэтому можем ее проигнорировать.

Шаг 8: Фреда мы еще не посещали, поэтому отмечаем его вершину как посещенную и добавляем в очередь:

Очередь

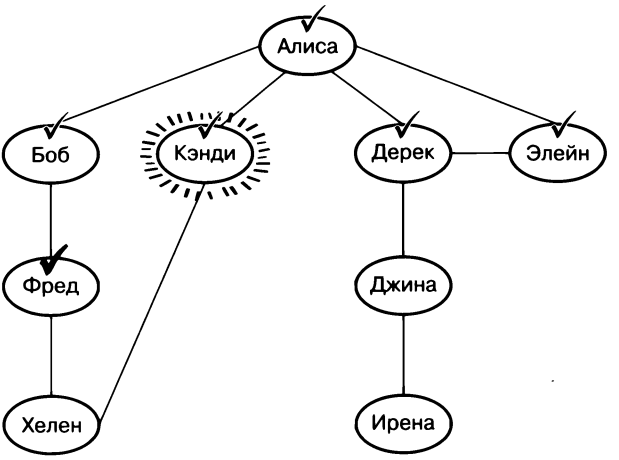
Кэнди	Дерек	Элейн	**Фред**



Шаг 9: других соседей у Боба нет, поэтому мы исключаем первый элемент из очереди и делаем его текущей вершиной. Сейчас это вершина Кэнди:

Очередь

Дерек	Элейн	Фред



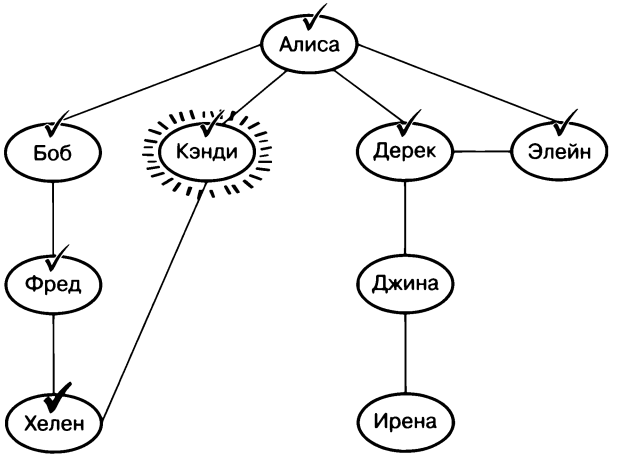
Перебираем соседей Кэнди.

Шаг 10: Алису мы уже посетили, поэтому снова можем ее проигнорировать.

Шаг 11: Хелен мы еще не посещали, поэтому отмечаем ее вершину как посещенную и добавляем в очередь:

Очередь

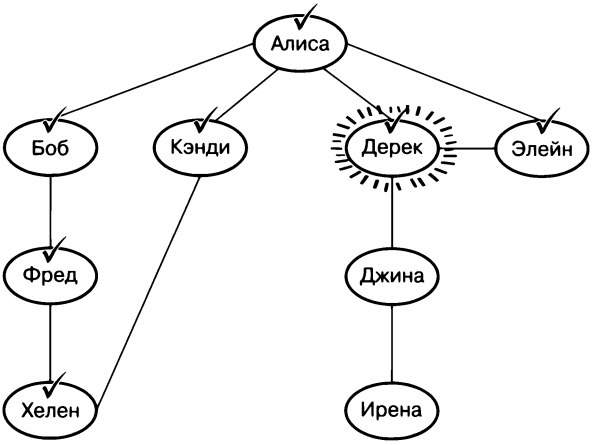
Дерек	Элейн	Фред	**Хелен**



Шаг 12: мы перебрали всех соседей текущей вершины (Кэнди), поэтому удаляем первый элемент из очереди (Дерек) и делаем его текущей вершиной:

Очередь

Элейн	Фред	Хелен



У вершины Дерека есть три смежные вершины, поэтому перебираем их.

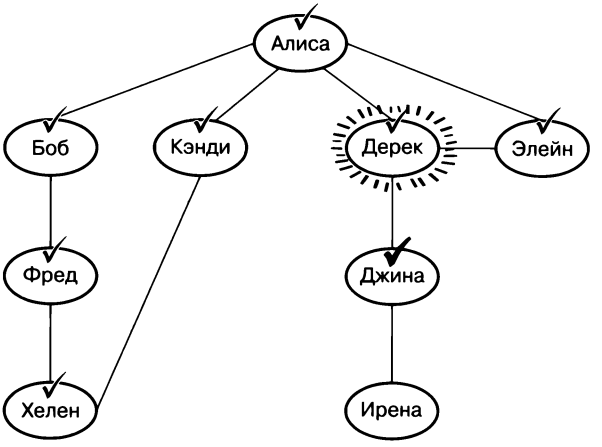
Шаг 13: Алису мы уже посетили, поэтому можем ее проигнорировать.

Шаг 14: то же касается и Элейн.

Шаг 15: остается вершина Джины, поэтому отмечаем ее как посещенную и добавляем в очередь:

Очередь

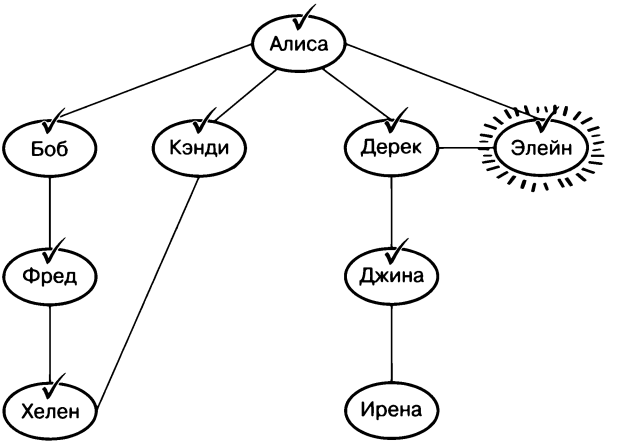
Элейн	Фред	Хелен	**Джина**



Шаг 16: мы перебрали всех соседей Дерека, поэтому удаляем из очереди вершину Элейн и делаем ее текущей:

Очередь

Фред	Хелен	Джина



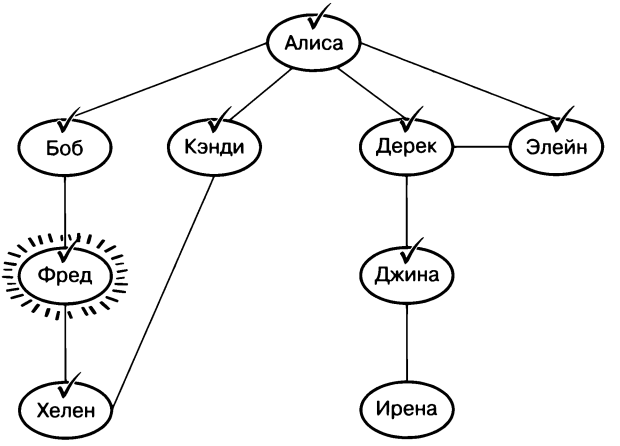
Шаг 17: перебираем соседей Элейн. Алису мы уже посещали.

Шаг 18: Дерека — тоже.

Шаг 19: удаляем из очереди вершину Фреда и делаем ее текущей:

Очередь

Хелен	Джина



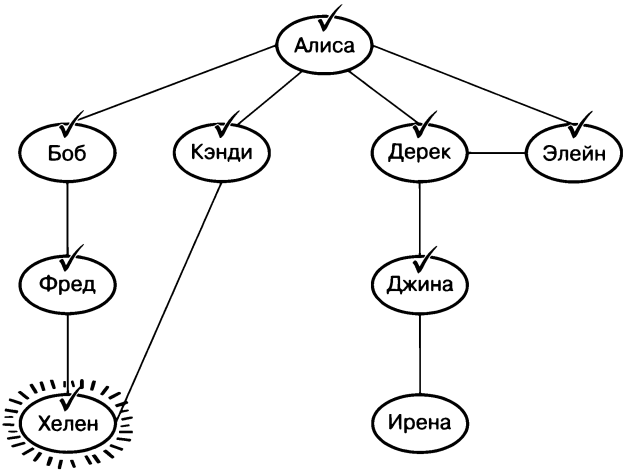
Шаг 20: перебираем соседей Фреда. Боба мы уже посещали.

Шаг 21: Хелен — тоже.

Шаг 22: поскольку вершина Хелен в самом начале очереди, мы исключаем ее и делаем текущей.

Очередь

Джина



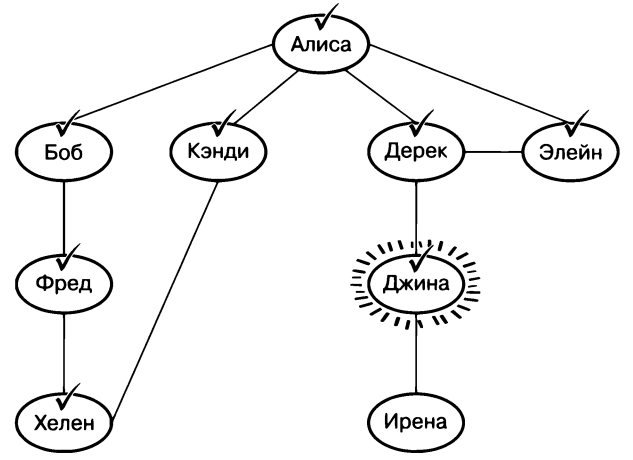
Шаг 23: у Хелен два соседа. Фреда мы уже посещали.

Шаг 24: Кэнди — тоже.

Шаг 25: исключаем вершину Джины из очереди и делаем ее текущей.

Очередь

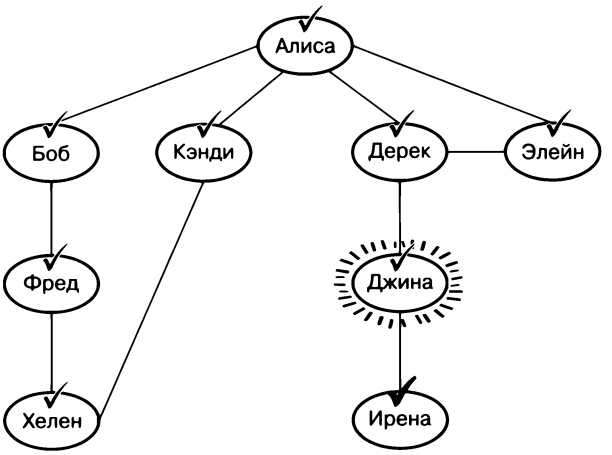
(пустая)



Шаг 26: перебираем соседей Джины. Дерека уже посещали.

Шаг 27: единственный сосед Джины, которого мы не посетили, — Ирена, поэтому посещаем ее вершину и добавляем ее в очередь:

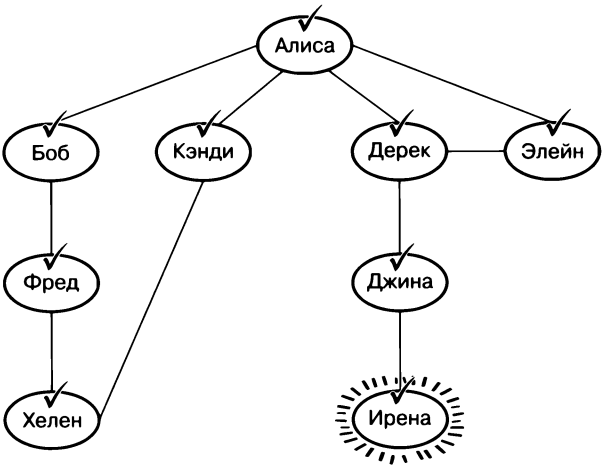
Очередь
Ирена



Итак, мы перебрали всех соседей Джины.

Шаг 28: удаляем из очереди первый (и единственный) элемент — вершину Ирены, которая становится текущей:

Очередь
(пустая)



Шаг 29: у Ирены только одна соседка — Джина, но мы ее уже посетили.

Теперь мы должны бы удалить следующий элемент из очереди, но она уже пуста! Значит, процесс обхода завершен.

Программная реализация

Так выглядит реализация алгоритма поиска в ширину:

```
def bfs_traverse(starting_vertex)
  queue = Queue.new

  visited_vertices = {}
  visited_vertices[starting_vertex.value] = true
  queue.enqueue(starting_vertex)

  # Пока очередь не опустеет:
  while queue.read

    # Удаляем первую вершину из очереди и делаем ее текущей:
    current_vertex = queue.dequeue

    # Выводим на экран значение текущей вершины:
    puts current_vertex.value

    # Перебираем соседей текущей вершины:
    current_vertex.adjacent_vertices.each do |adjacent_vertex|

      # Если мы еще не посетили эту соседнюю вершину:
      if !visited_vertices[adjacent_vertex.value]

        # Отмечаем соседнюю вершину как посещенную:
        visited_vertices[adjacent_vertex.value] = true

        # Добавляем соседнюю вершину в очередь:
        queue.enqueue(adjacent_vertex)
      end
    end
  end
end
```

Метод `bfs_traverse` принимает значение `starting_vertex` — вершину, с которой мы начинаем поиск.

Сначала создаем очередь, лежащую в основе алгоритма:

```
queue = Queue.new
```

Создаем хеш-таблицу `visited_vertices`, с помощью которой будем отслеживать посещенные вершины:

```
visited_vertices = {}
```

Отмечаем начальную вершину `starting_vertex` как посещенную и добавляем ее в очередь:

```
visited_vertices[starting_vertex.value] = true
queue.enqueue(starting_vertex)
```

Запускаем цикл, который выполняется, пока очередь не опустеет:

```
while queue.read
```

Удаляем первый элемент из очереди и делаем его текущей вершиной:

```
current_vertex = queue.dequeue
```

Теперь отображаем в консоли значение вершины, чтобы убедиться в том, что алгоритм работает правильно:

```
puts current_vertex.value
```

Перебираем всех соседей текущей вершины:

```
current_vertex.adjacent_vertices.each do |adjacent_vertex|
```

Добавляем каждую еще не посещенную вершину в хеш-таблицу, отмечая ее как посещенную, и ставим ее в очередь:

```
if !visited_vertices[adjacent_vertex.value]
  visited_vertices[adjacent_vertex.value] = true
  queue.enqueue(adjacent_vertex)
end
```

Вот и все.

Поиск в глубину и поиск в ширину

Как вы видели, при поиске в ширину мы сначала посещаем всех прямых соседей Алисы, а после постепенно удаляемся от нее. При поиске в глубину мы, наоборот, сразу же максимально удаляемся от Алисы, после чего возвращаемся к ней, чтобы продолжить поиск со следующего прямого соседа.

Итак, у нас есть два способа поиска по графу: в глубину и в ширину. Какой из них лучше?

Конечно, все зависит от конкретной ситуации. Иногда поиск в глубину может выполняться быстрее, чем в ширину, иногда — наоборот.

Обычно выбор подхода зависит от особенностей графа и от ваших целей. Как уже говорилось, ключевая разница между ними в том, что при поиске в ширину

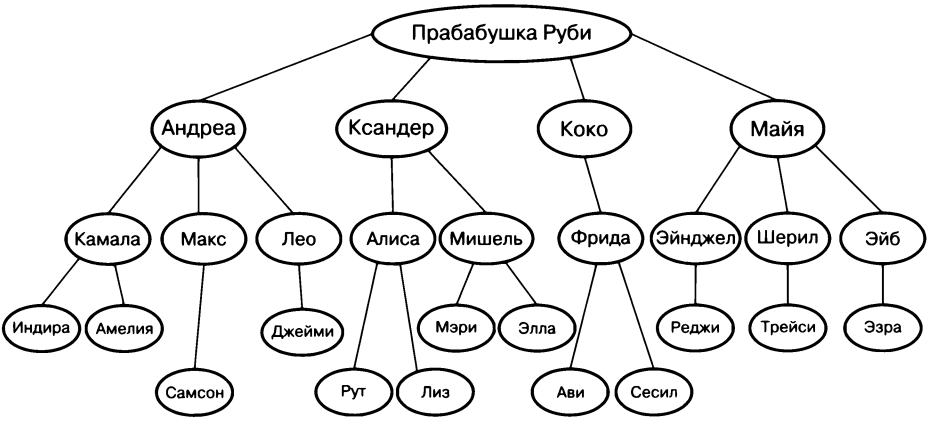
мы начинаем с обхода всех вершин по соседству с начальной, а при поиске в глубину, напротив, сразу же максимально удаляемся от начальной вершины и возвращаемся к ней, только когда заходим в тупик.

Допустим, мы хотим установить все *прямые* связи пользователя социальной сети. Например, нам нужно найти всех реальных друзей Алисы в графе из прошлого примера. При этом нас не интересуют друзья ее друзей — нам нужен только список ее контактов первого уровня.

Как вы помните, при поиске в ширину мы нашли всех друзей Алисы (Боба, Кэнди, Дерека и Элейн), прежде чем перешли к ее контактам второго уровня.

Но при обходе графа с помощью алгоритма поиска в глубину мы обнаружили Фреда и Хелен (они не друзья Алисы), прежде чем нашли остальных ее друзей. Если бы граф был больше, мы бы потратили на обход ненужных вершин еще больше времени.

Но рассмотрим другой сценарий: наш граф — это генеалогическое древо, которое выглядит примерно так:



Здесь перечислены все потомки прабабушки Руби, великой праматери замечательной семьи. Мы знаем, что Рут — правнучка Руби, и хотим найти ее в этом графе.

Используй мы для этого поиск в ширину, нам бы пришлось обойти всех детей и внуков Руби, прежде чем мы обнаружим первого правнука.

Поиск в глубину же позволяет сразу спуститься по графу и обнаружить первого правнука всего за несколько шагов. Возможно, нам придется пройти весь граф, чтобы добраться до Рут, но у нас хотя бы есть шанс быстро ее найти, тог-

да как при поиске в ширину нам придется обойти всех детей и внуков, прежде чем начать обход правнуков.

Итак, перед началом поиска важно понять: хотим ли мы оставаться близко к начальной вершине в процессе или, наоборот, стремимся максимально от нее отдалиться. В первом случае отлично подойдет поиск в ширину, а во втором — в глубину.

Эффективность поиска по графу

Отразим временную сложность поиска по графу с помощью О-нотации.

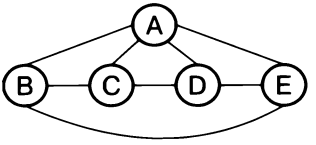
В худшем сценарии при использовании обоих способов поиска мы обходим все вершины. Так может произойти, если мы ищем в графе вершину, которой в нем нет, или она оказывается последней вершиной графа, которую мы проверяем.

Так или иначе мы обходим все вершины графа. Может показаться, что временная сложность этого алгоритма равна O(N), где N — количество вершин.

Но оба алгоритма поиска предполагают, что при обходе каждой вершины *мы перебираем и всех ее соседей*. Мы можем проигнорировать соседнюю вершину, если она уже посещалась, но нам все равно придется выполнить дополнительный шаг, чтобы проверить, посещали мы ее или нет.

Выходит, что при посещении каждой вершины мы совершаем дополнительные шаги для проверки всех ее соседей. Выразить это с помощью О-нотации уже сложнее, ведь у каждой вершины может быть разное количество смежных вершин.

Покажу это на примере простого графа:



Здесь у вершины A четыре соседа, а у вершин B, C, D и E по три соседа. Сколько шагов нужно для выполнения поиска по этому графу?

Как минимум мы должны посетить каждую из пяти вершин. Уже только на это у нас уйдет пять шагов.

Кроме того, мы должны перебрать всех соседей каждой вершины.

Для этого нужно выполнить следующие шаги:

A: 4 шага для проверки 4 соседей;

B: 3 шага для проверки 3 соседей;

C: 3 шага для проверки 3 соседей;

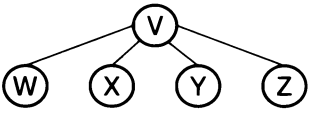
D: 3 шага для проверки 3 соседей;

E: 3 шага для проверки 3 соседей.

Итого: 16 итераций.

В общей сложности мы должны выполнить пять шагов для посещения пяти вершин плюс 16 итераций для перебора их соседей — всего 21 шаг.

Но вот еще один граф с пятью вершинами:



Как и у прошлого, у него пять вершин, но для перебора их соседей нужно следующее количество итераций:

V: 4 шага для проверки 4 соседей;

W: 1 шаг для проверки 1 соседа;

X: 1 шаг для проверки 1 соседа;

Y: 1 шаг для проверки 1 соседа;

Z: 1 шаг для проверки 1 соседа.

Итого: 8 итераций.

В общей сложности нужно выполнить пять шагов для посещения пяти вершин плюс восемь итераций для перебора их соседей — всего 13 шагов.

Итак, у нас есть два графа по пять вершин. Но на поиск в первом уйдет 21 шаг, а во втором — 13.

Получается, что при оценке временной сложности алгоритма мы не можем просто подсчитать число вершин в графе. *Нам нужно учесть и число соседей каждой вершины.*

Для определения временной сложности поиска по графу придется использовать *две* переменные, одна из которых будет отражать количество вершин в графе, а вторая — общее число соседей каждой вершины.

Временная сложность O(V + E)

Любопытно, что для определения вышеописанных параметров в O-нотации вместо N используются переменные V и E.

V — начальная буква в слове *vertex* (вершина) и обозначает число вершин в графе.

E — начальная буква в слове *edge* (ребро) и обозначает количество ребер в графе.

Программисты определяют временную сложность поиска по графу как O($V + E$). Это значит, что количество шагов алгоритма равно *общему количеству* вершин и ребер графа. Разберемся, почему так.

Если вы посмотрите на два прошлых графа, то заметите, что выражение $V + E$ не вполне точно их описывает.

В графе A-B-C-D-E пять вершин и восемь ребер. Всего должно было бы получиться 13 шагов. Но на фактический поиск ушел бы 21 шаг.

В графе V-W-X-Y-Z пять вершин и четыре ребра. Согласно выражению O($V + E$), поиск должен выполняться за девять шагов. Но мы выяснили, что на самом деле их 13.

Причина несоответствия в том, что выражение O($V + E$) учитывает количество ребер только один раз, тогда как фактический поиск по графу затрагивает каждое *ребро* более одного раза.

Например, в графе V-W-X-Y-Z всего четыре ребра. Но ребро между вершинами V и W задействуется дважды: когда текущая вершина — V и мы находим ее соседа W и когда текущая вершина — W и мы находим ее соседа V.

Учитывая это, для более точного подсчета шагов при поиске по графу V-W-X-Y-Z нужно прибавить к пяти вершинам следующее:

2 × ребро между V и W;

2 × ребро между V и X;

2 × ребро между V и Y;

2 × ребро между V и Z.

В итоге мы получаем $V + 2E$, так как $V = 5$, а каждое ребро используется дважды.

Но *O-нотация игнорирует константы*, поэтому временная сложность этого алгоритма будет O($V + E$), несмотря на то что фактическое число шагов — $V + 2E$.

Итак, оценка O($V + E$) приблизительная, но нас это вполне устраивает.

Очевидно, что увеличение числа ребер приводит к увеличению количества шагов. Например, в графах A-B-C-D-E и V-W-X-Y-Z по пять вершин, но, так как у первого больше ребер, на поиск по нему уйдет гораздо больше шагов.

В конце концов, в худшем случае, когда искомая вершина последняя из обнаруженных (или ее вообще нет в графе), поиск по графу занимает время $O(V + E)$. И это касается как поиска в ширину, так и поиска в глубину.

Но мы уже знаем, что в зависимости от формы графа и вероятного местоположения искомых данных мы можем оптимизировать процесс поиска, выбрав тот или иной его вариант. То есть подходящий метод поиска позволяет увеличить шансы на то, что худший сценарий не реализуется и для нахождения нужной вершины нам не придется обходить весь граф.

В следующем разделе вы узнаете о графе особого типа, с собственным набором методов поиска, подходящем для решения некоторых действительно сложных, но важных задач.

Графовые базы данных

Графы позволяют очень эффективно работать с данными, отражающими разные типы связей (например, между друзьями в социальной сети), и для их хранения в реальных приложениях часто используются специальные *графовые базы данных* (*БД*). Для оптимизации операций над данными в таких базах используются подходы, о которых вы узнаете в этой главе, и другие концепции теории графов. На самом деле графовые БД лежат в основе многих приложений, обеспечивающих работу социальных сетей.

Примеры таких баз данных:

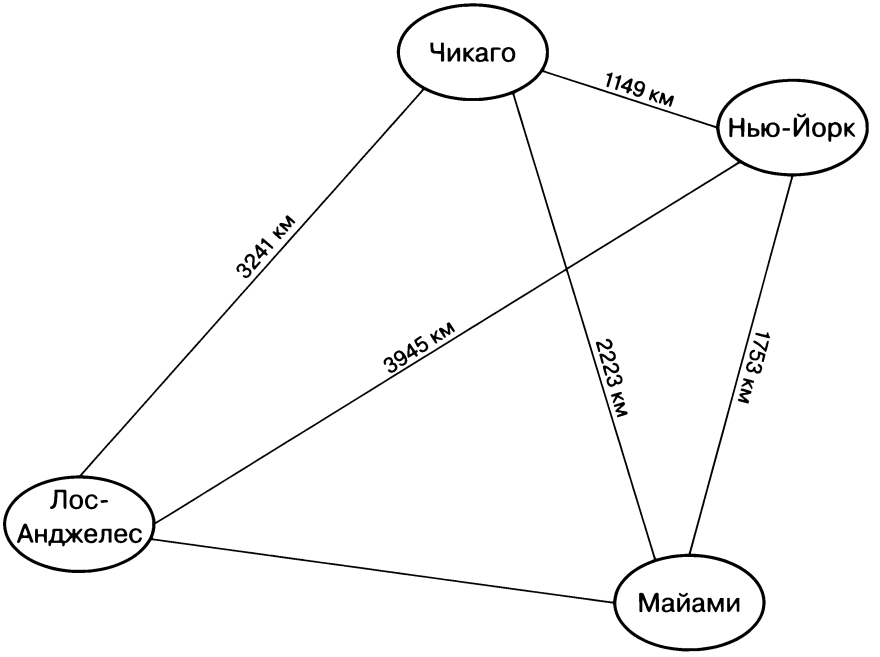
- Neo4j (http://neo4j.com/);
- ArangoDB (https://www.arangodb.com/);
- Apache Giraph (http://giraph.apache.org/).

Подробно ознакомиться с принципом работы графовых БД можно на соответствующих веб-сайтах.

Взвешенные графы

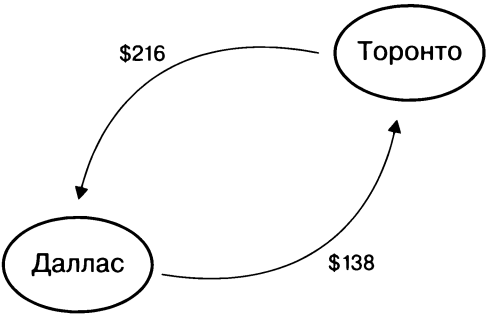
Мы уже выяснили, что есть много разных графов. Еще одна их полезная разновидность — *взвешенный* (*или размеченный*) *граф*, *ребра* которого несут дополнительную информацию.

Вот взвешенный граф простой карты с указанием нескольких крупных городов США:



Каждое ребро этого графа сопровождается числом, отражающим расстояние в милях (км) между городами, которые соединяет это ребро. Например, расстояние между Чикаго и Нью-Йорком — 1149 км.

Взвешенные графы могут быть ориентированными. Ниже можно увидеть, что стоимость перелета из Далласа в Торонто — 138 долларов, а обратный перелет из Торонто в Даллас стоит 216 долларов:



Реализация взвешенного графа

Чтобы присвоить вес ребрам нашего графа, нужно внести в код небольшие изменения. Например, использовать хеш-таблицу для представления смежных вершин вместо массива:

```
class WeightedGraphVertex
  attr_accessor :value, :adjacent_vertices

  def initialize(value)
    @value = value
    @adjacent_vertices = {}
  end

  def add_adjacent_vertex(vertex, weight)
    @adjacent_vertices[vertex] = weight
  end
end
```

Как видите, теперь `@adjacent_vertices` — это хеш-таблица, а не массив. В ней будут пары «ключ — значение», где ключ — это соседняя вершина, а значение — вес (ребра, соединяющего эту вершину с соседней).

Теперь, используя метод `add_adjacent_vertex` для добавления соседней вершины, мы передаем и ее, и вес.

Итак, для создания графа, отражающего цены на рейсы из Далласа в Торонто и обратно, запускаем следующий код:

```
dallas = City.new("Dallas")
toronto = City.new("Toronto")

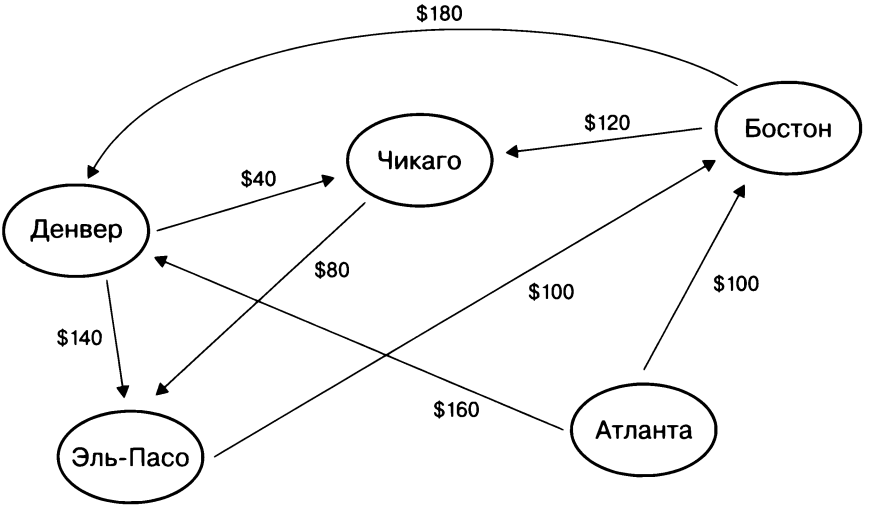
dallas.add_adjacent_vertex(toronto, 138)
toronto.add_adjacent_vertex(dallas, 216)
```

Задача о кратчайшем пути

Взвешенные графы очень полезны для моделирования разных наборов данных и предусматривают несколько мощных алгоритмов, помогающих использовать эти данные максимально эффективно.

Применим один из таких алгоритмов, чтобы сэкономить немного денег.

На следующем графе показана стоимость авиарейсов между пятью городами.



Допустим, я нахожусь в Атланте и хочу отправиться в Эль-Пасо. К сожалению, как видно на графе, прямого рейса из Атланты в Эль-Пасо сейчас нет, но я могу добраться до нужного города с пересадками. Например, полететь из Атланты в Денвер, а из Денвера — в Эль-Пасо. Но есть и другие маршруты, и стоимость у каждого из них разная. Если я выберу маршрут Атланта — Денвер — Эль-Пасо, то вся поездка обойдется мне в 300 долларов, а при выборе маршрута Атланта — Денвер — Чикаго — Эль-Пасо — всего 280 долларов.

Как реализовать алгоритм, который находит самый *дешевый* способ достижения пункта назначения? Предположим, нас не волнует число пересадок: мы просто хотим определить самый недорогой маршрут.

Этот тип головоломки известен как *задача о кратчайшем пути*. Он может принимать и другие формы. Например, если бы на графе были показаны расстояния между городами, мы могли бы попытаться найти самый короткий маршрут. Но сейчас речь идет о нахождении самого дешевого маршрута, так как вес отражает цену на авиабилеты.

Алгоритм Дейкстры

Есть много алгоритмов для решения задачи о кратчайшем пути. Один из самых известных разработан Эдсгером Дейкстрой в 1959 году и носит его имя.

В следующем разделе мы используем этот алгоритм для поиска самого дешевого маршрута на примере графа выше.

Подготовка алгоритма Дейкстры

Важно отметить, что у алгоритма Дейкстры есть бесплатный бонус: по завершении всего процесса нам будет известен самый дешевый маршрут из Атланты не только в Эль-Пасо, но и в *остальные* известные города. Далее вы увидите, что принцип работы этого алгоритма позволяет собрать все эти данные. Так что в итоге мы определим самый дешевый маршрут от Атланты до Чикаго, до Денвера и т. д.

Но перед использованием алгоритма нужно подумать о том, как мы будем хранить минимальные значения стоимости поездки из начального города во все известные пункты назначения. В коде для этого будет использоваться хеш-таблица, а при пошаговом разборе процесса работы алгоритма — следующая таблица:

Из Атланты в:	**Город № 1**	**Город № 2**	**Город № 3**	**И т. д.**
	?	?	?	?

Алгоритм начнет с вершины Атланты — это единственный из известных на этот момент городов. По мере обнаружения новых городов мы будем добавлять их в таблицу и фиксировать минимальную стоимость поездки из Атланты в каждый из них.

По завершении работы алгоритма таблица будет выглядеть так:

Самый дешевый маршрут из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
	$100	$200	$160	$280

В коде это будет в виде следующей хеш-таблицы:

```
{"Atlanta" => 0, "Boston" => 100, "Chicago" => 200, "Denver" => 160, "El
Paso" => 280}
```

Обратите внимание, что Атланта включена в эту таблицу со значением 0 — оно понадобится алгоритму для работы. И это вполне логично, учитывая, что перелет из Атланты в Атланту ничего не стоит, ведь вы уже и так на месте!

В дальнейшем обсуждении будем называть эту таблицу `cheapest_prices_table`, так как в ней будут храниться минимальные значения стоимости поездки из начального города до всех пунктов назначения.

Если бы мы хотели только определить самую дешевую поездку до города, то в итоге таблица `cheapest_prices_table` содержала бы все необходимые нам данные. Но нас интересует фактический маршрут. То есть, если мы хотим добраться из Атланты в Эль-Пасо, нам мало просто знать, что минимальная стоимость поездки — 280 долларов: мы хотим знать, что она соответствует определенному маршруту: Атланта — Денвер — Чикаго — Эль-Пасо.

Поэтому нам понадобится *еще одна* таблица — `cheapest_previous_stopover_city_table`. Ее назначение станет понятным при обсуждении работы алгоритма, поэтому пока опустим этот момент. Сейчас мне важно показать, как эта таблица будет выглядеть по окончании работы алгоритма.

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
	Атланта	Денвер	Атланта	Чикаго

Обратите внимание, что в коде она тоже будет реализована в виде хеш-таблицы.

Этапы алгоритма Дейкстры

Итак, у нас все готово. Теперь давайте рассмотрим конкретные этапы алгоритма Дейкстры. Для ясности я буду описывать алгоритм с точки зрения городов, но вы можете заменить слово «город» на «вершина», чтобы применить его для любого взвешенного графа. Имейте в виду, что эти шаги будут более понятны, когда мы рассмотрим их на конкретном примере. Итак, начнем.

1. Посещаем начальный город, делая его «текущим».
2. Проверяем цены на билеты из текущего города в каждый соседний.
3. Если стоимость поездки в соседний город из начального ниже текущего значения в таблице `cheapest_prices_table` (или если соседний город вообще не указан в ней):
   а) обновляем таблицу `cheapest_prices_table`, включая в нее обнаруженную минимальную стоимость поездки;
   б) обновляем таблицу `cheapest_previous_stopover_city_table`, указывая соседний город в качестве ключа, а текущий — в качестве значения.
4. Обращаемся к еще не посещенному городу, до которого дешевле всего добраться из начального, делая его текущим.
5. Повторяем шаги со 2 по 4, пока не посетим все известные города.

Опять же, все станет понятнее, когда мы рассмотрим конкретный пример.

Пошаговый разбор алгоритма Дейкстры

Разберем алгоритм Дейкстры шаг за шагом.

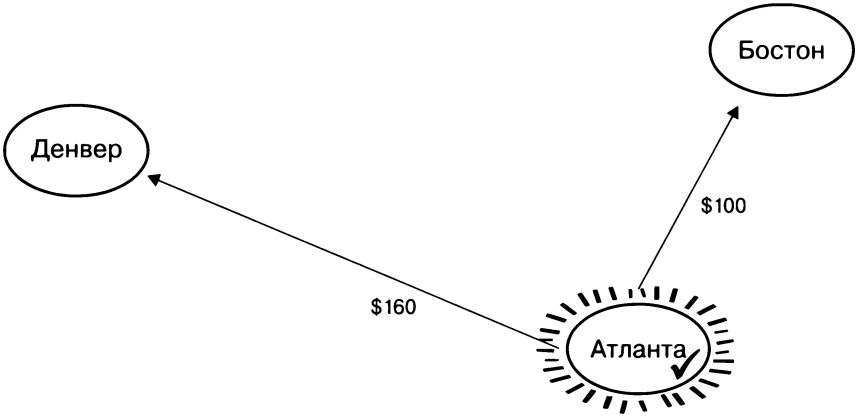
Изначально в таблице `cheapest_prices_table` есть только Атланта:

Из Атланты в:
$0

Сразу после запуска алгоритма Атланта — единственный город, к которому у нас есть доступ: другие мы еще не «обнаружили».

Шаг 1: официально посещаем Атланту и делаем ее текущим городом (`current_city`).

Чтобы отметить город как текущий, окружаем его штрихами, а чтобы указать на то, что мы его уже посетили, — ставим галочку:



Теперь проверяем соседние города по отношению к текущему: так мы «обнаруживаем» новые пункты назначения. Если у текущего города есть соседние, о которых мы раньше не знали, добавляем их на карту.

Шаг 2: один из соседей Атланты — Бостон. Стоимость поездки из Атланты в Бостон — 100 долларов. Мы проверяем таблицу `cheapest_prices_table`, чтобы сравнить эту стоимость с минимальной известной ценой на билет из Атланты в Бостон. Оказывается, что сравнивать ее пока не с чем. Значит, сейчас обнаруженная цена — минимальная *из известных*, поэтому добавляем ее в таблицу `cheapest_prices_table`:

Из Атланты в:	**Бостон**
$0	*$100*

Мы внесли изменения в таблицу с минимальными ценами, поэтому нужно обновить таблицу `cheapest_previous_stopover_city_table`, сделав соседний город (Бостон) ключом, а текущий — значением:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**
	Атланта

Добавление этих данных в таблицу означает, что для снижения стоимости поездки из Атланты в Бостон ($100) *перед Бостоном* нужно посетить именно Атланту. Сейчас это и так понятно, ведь Атланта — единственный известный нам город, из которого можно добраться до Бостона. Но в дальнейшем значимость второй таблицы станет более очевидной.

Шаг 3: мы уже проверили Бостон, но у Атланты есть еще один соседний город, Денвер. Проверяем, действительно ли маршрут поездки стоимостью $160 — самый дешевый из известных маршрутов Атланта — Денвер. Но Денвер еще не внесен в таблицу `cheapest_prices_table`, поэтому добавляем в нее обнаруженную стоимость в качестве минимальной на этом этапе:

Из Атланты в:	**Бостон**	**Денвер**
$0	$100	$160

Затем добавляем Денвер и Атланту в качестве пары «ключ — значение» в таблицу `cheapest_previous_stopover_city_table`:

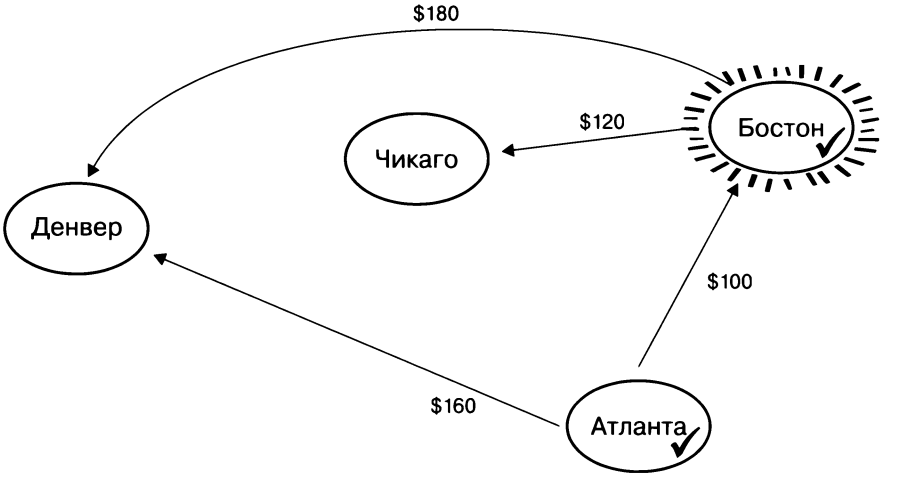
Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Денвер**
	Атланта	*Атланта*

Шаг 4: к этому моменту мы проверили всех соседей Атланты и готовы посетить следующий город. Но нам нужно как-то его выбрать.

Теперь, согласно шагам алгоритма выше, приступаем к посещению только тех городов, где еще не были. Среди еще не посещенных городов всегда *сначала* выбираем тот, путь до которого *из начального города* дешевле всего. Эти данные можно получить из таблицы `cheapest_prices_table`.

Известные, но еще не посещенные города в нашем примере — Бостон и Денвер. Судя по таблице `cheapest_prices_table`, добраться из Атланты в Бостон дешевле, чем из Атланты в Денвер, поэтому посетим Бостон.

Шаг 5: посещаем Бостон и обозначаем его как текущий город:



Теперь нужно проверить соседние с Бостоном города.

Шаг 6: у Бостона два соседа — Чикаго и Денвер (Атланта не в счет — нам не нужно лететь из Бостона в Атланту).

Какой город мы должны посетить первым — Чикаго или Денвер? Опять же, сначала нужно посетить тот, путь до которого *из Атланты* обойдется дешевле всего. Итак, давайте посчитаем.

Стоимость перелета из Бостона в Чикаго — 120 долларов. Согласно таблице `cheapest_prices_table`, минимальная цена на билет из Атланты до Бостона — 100 долларов. Выходит, стоимость самого дешевого маршрута от Атланты до Чикаго *с последней пересадкой в Бостоне* будет 220 долларов.

Поскольку сейчас этот маршрут между Атлантой и Чикаго — единственный из известных, добавим его стоимость в таблицу `cheapest_prices_table`:

Из Атланты в:	**Бостон**	**Чикаго**	**Денвер**
$0	$100	*$220*	$160

Опять же, поскольку мы внесли изменения в таблицу, нам нужно обновить таблицу `cheapest_previous_stopover_city_table`. При этом соседний город всегда становится ключом, а текущий — значением:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Чикаго**	**Денвер**
	Атланта	*Бостон*	Атланта

Итак, мы проверили Чикаго, теперь займемся Денвером.

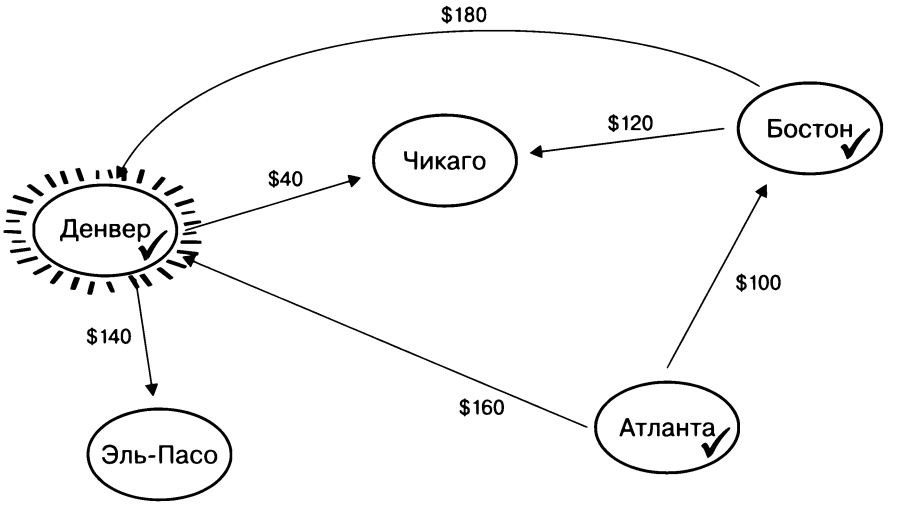
Шаг 7: если мы посмотрим на ребро графа, соединяющее Бостон с Денвером, то увидим, что стоимость перелета между ними — 180 долларов. Самый дешевый рейс из Атланты в Бостон стоит 100 долларов, поэтому минимальная стоимость маршрута от Атланты до Денвера *с последней пересадкой в Бостоне* будет составлять 280 долларов.

Интересно, что в нашей таблице `cheapest_prices_table` уже есть стоимость перелета из Атланты в Денвер — 160 долларов — это *гораздо дешевле* по сравнению со стоимостью маршрута Атланта — Бостон — Денвер. Поэтому *мы не обновляем* ни одну из таблиц, чтобы оставить значение $160 как минимальную известную стоимость поездки из Атланты в Денвер.

Итак, мы закончили с Бостоном и проверили все его соседние города. Теперь посетим следующий город.

Шаг 8: мы еще не посетили Чикаго и Денвер. Опять же, сначала обратимся к городу, путь до которого *из начального города* (Атланты) обойдется нам дешевле всего. Обратите внимание, что речь идет об исходной точке маршрута.

Согласно таблице `cheapest_prices_table`, добраться из Атланты в Денвер ($160) дешевле, чем из Атланты в Чикаго ($220), поэтому посетим Денвер:



Проверяем соседние к Денверу города.

Шаг 9: у Денвера два соседних города — Чикаго и Эль-Пасо. Чтобы решить, какой из них посетить первым, нужно проанализировать стоимость перелета в эти города. Начнем с Чикаго.

Добраться из Денвера в Чикаго можно всего за 40 долларов (отличное предложение!). Значит, стоимость самого дешевого маршрута от Атланты до Чикаго *с последней пересадкой в Денвере* — 200 долларов, поскольку добраться из Атланты в Денвер можно за 160.

Согласно таблице `cheapest_prices_table`, текущая минимальная стоимость поездки из Атланты в Чикаго — 220 долларов. Значит, новый маршрут через Денвер позволяет добраться из Атланты в Чикаго еще дешевле, поэтому мы обновляем таблицу `cheapest_prices_table`:

Из Атланты в:	**Бостон**	**Чикаго**	**Денвер**
$0	$100	*$200*	$160

Каждое обновление таблицы `cheapest_prices_table` должно сопровождаться обновлением таблицы `cheapest_previous_stopover_city_table`. Итак, мы указываем соседний город (Чикаго) в качестве ключа, а текущий (Денвер) — в качестве значения. Ключ «Чикаго» уже существует, поэтому при обновлении таблицы перезапишем его значение, заменив его с Бостона на Денвер:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Чикаго**	**Денвер**
	Атланта	*Денвер*	Атланта

Выходит, для минимизации стоимости поездки из Атланты в Чикаго нам нужно прилететь в Чикаго из Денвера: последняя пересадка должна быть именно в Денвере. Только тогда мы сэкономим максимальное количество денег.

Эта информация пригодится нам при определении самого дешевого маршрута из Атланты в пункт назначения. Потерпите, мы почти у цели!

Шаг 10: у Денвера есть еще один соседний город — Эль-Пасо. Цена билета из Денвера в Эль-Пасо — 140 долларов. Теперь мы можем подсчитать стоимость первого известного маршрута от Атланты до Эль-Пасо. Согласно таблице `cheapest_prices_table`, добраться из Атланты в Денвер можно минимум за 160 долларов. Значит, если затем мы поедем из Денвера в Эль-Пасо, то потратим еще 140 долларов, в результате чего общая стоимость поездки из Атланты в Эль-Пасо составит 300 долларов. Добавим эту сумму в таблицу `cheapest_prices_table`:

Из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
$0	$100	$200	$160	*$300*

Добавим пару «ключ — значение» «Эль-Пасо — Денвер» в нашу таблицу `cheapest_previous_stopover_city_table`:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	Бостон	Чикаго	Денвер	Эль-Пасо
	Атланта	Денвер	Атланта	*Денвер*

Значит, чтобы сэкономить больше денег при поездке из Атланты в Эль-Пасо, нужно сделать последнюю пересадку в Денвере.

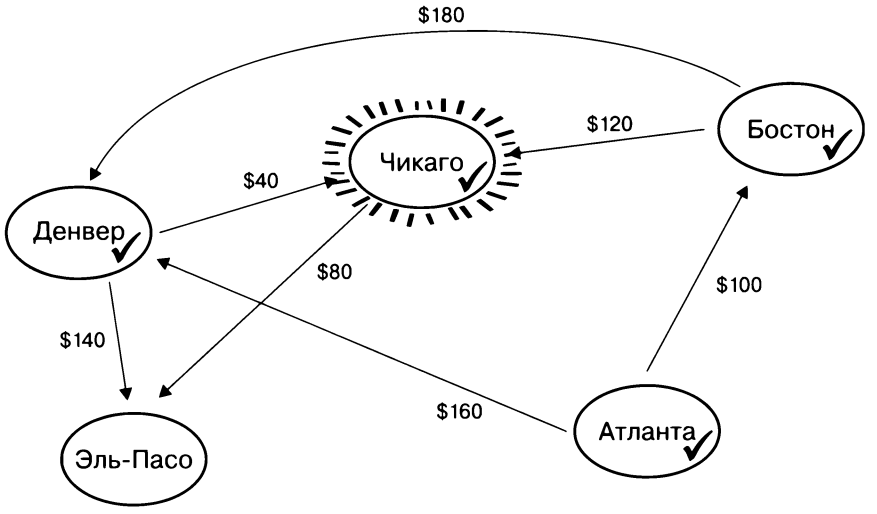
Итак, мы проверили всех соседей текущего города и теперь можем посетить следующий.

Шаг 11: у нас есть два известных, но еще не посещенных города — Чикаго и Эль-Пасо. Добраться из Атланты в Чикаго ($200) дешевле, чем в Эль-Пасо ($300), поэтому посещаем Чикаго.

Шаг 12: у Чикаго только один соседний город — Эль-Пасо. Цена билета из Чикаго до Эль-Пасо — 80 долларов (неплохо). С учетом этой информации мы можем определить минимальную стоимость поездки из Атланты в Эль-Пасо с последней пересадкой в Чикаго.

Согласно таблице `cheapest_prices_table`, минимальная стоимость поездки из Атланты в Чикаго — 200 долларов. Прибавим к этой сумме 80 долларов, и получим общую стоимость поездки из Атланты в Эль-Пасо *с последней пересадкой в Чикаго* — 280 долларов.

Кажется, мы нашли маршрут подешевле! Согласно таблице `cheapest_prices_table`, минимальная стоимость поездки из Атланты в Эль-Пасо составляет 300 долларов. Но если мы полетим через Чикаго, стоимость снизится до 280 долларов.



Соответственно, нам нужно обновить таблицу `cheapest_prices_table`, указав стоимость найденного маршрута до Эль-Пасо:

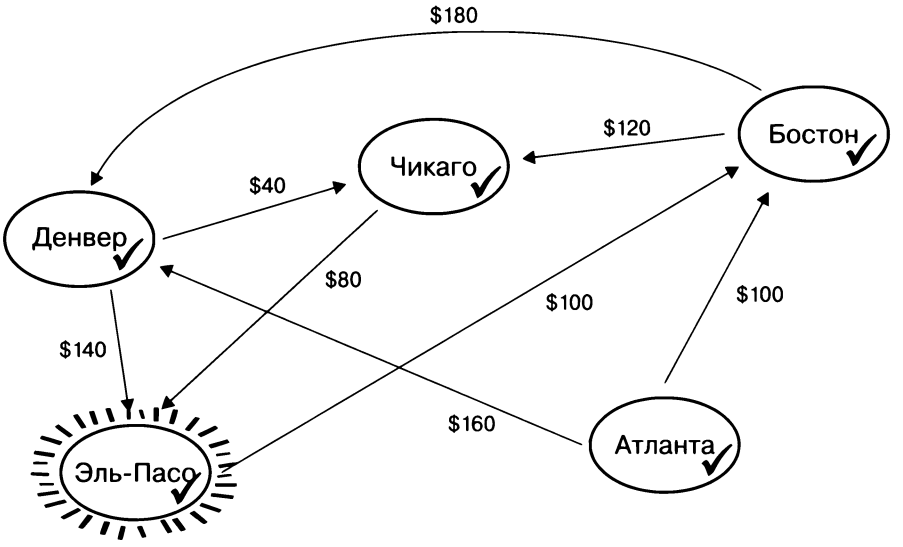
Из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
$0	$100	$200	$160	$280

Обновим таблицу `cheapest_previous_stopover_city_table`, указав Эль-Пасо в качестве ключа и Чикаго в качестве значения:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
	Атланта	Денвер	Атланта	*Чикаго*

У Чикаго больше нет соседей, так что теперь мы можем посетить следующий город.

Шаг 13: Эль-Пасо — это единственный из известных, но еще не посещенных городов, поэтому сделаем его текущим.



Шаг 14: из Эль-Пасо можно полететь только в Бостон. Такой перелет обойдется в 100 долларов. Согласно таблице `cheapest_prices_table`, минимальная стоимость поездки из Атланты в Эль-Пасо — 280 долларов. Итак, если мы полетим из Атланты в Бостон *с последней пересадкой в Эль-Пасо*, то общая стоимость

поездки составит 380 долларов. Это дороже текущей наименьшей стоимости перелета из Атланты в Бостон ($100), поэтому таблицы мы не обновляем.

Мы посетили все известные города, и у нас есть вся необходимая информация для нахождения самого недорогого маршрута из Атланты в Эль-Пасо.

Поиск кратчайшего пути

Чтобы узнать минимальную стоимость поездки из Атланты в Эль-Пасо, мы можем просто заглянуть в таблицу `cheapest_prices_table` и выяснить, что эта стоимость составляет 280 долларов. Но если нас интересует *конкретный маршрут*, нам нужно сделать еще кое-что.

Помните нашу таблицу `cheapest_previous_stopover_city_table`? Пришло время использовать данные из нее.

Сейчас она выглядит так:

Город последней пересадки, снижающей стоимость поездки из Атланты в:	**Бостон**	**Чикаго**	**Денвер**	**Эль-Пасо**
	Атланта	Денвер	Атланта	Чикаго

Мы можем использовать эту таблицу, чтобы найти кратчайший путь из Атланты в Эль-Пасо, двигаясь в обратном направлении.

Сначала посмотрим на Эль-Пасо: ему соответствует Чикаго. Значит, самый дешевый маршрут из Атланты в Эль-Пасо предполагает последнюю пересадку в Чикаго. Запишем это так: Чикаго → Эль-Пасо.

Если мы снова обратимся к таблице `cheapest_previous_stopover_city_table`, то увидим, что ключу Чикаго соответствует значение Денвер. Значит, самый дешевый маршрут из Атланты в *Чикаго* предполагает последнюю пересадку в Денвере. Добавим этот город в наш маршрут: Денвер → Чикаго → Эль-Пасо.

В той же таблице мы видим, что самый дешевый маршрут из Атланты в Денвер — прямой рейс между этими городами: Атланта → Денвер → Чикаго → Эль-Пасо.

Атланта — начальный город, поэтому полученный маршрут оказывается самым дешевым способом добраться из Атланты в Эль-Пасо.

Рассмотрим логику, лежащую в основе составления самого дешевого маршрута.

Как вы помните, в таблице `cheapest_previous_stopover_city_table` для каждого пункта назначения указан город последней пересадки, позволяющий снизить общую стоимость поездки из Атланты.

Исходя из этой таблицы, для минимизации стоимости поездки из Атланты в Эль-Пасо нам нужно лететь...

- ... в Эль-Пасо из Чикаго,
- ... в Чикаго — из Денвера,
- ...а в Денвер — из Атланты.

Значит, самый дешевый маршрут: Атланта → Денвер → Чикаго → Эль-Пасо.

Вот и все. Наконец-то!

Программная реализация: алгоритм Дейкстры

Прежде чем реализовать весь алгоритм на языке Ruby, реализуем класс `City`, который похож на `WeightedGraphVertex`, но использует такие понятия, как `routes` (маршруты) и `price` (цена). Это (немного) упростит наш будущий код:

```
class City
  attr_accessor :name, :routes

  def initialize(name)
    @name = name
    @routes = {}
  end

  def add_route(city, price)
    @routes[city] = price
  end
end
```

Чтобы использовать данные из примера выше, запустите следующий код:

```
atlanta = City.new("Atlanta")
boston = City.new("Boston")
chicago = City.new("Chicago")
denver = City.new("Denver")
el_paso = City.new("El Paso")

atlanta.add_route(boston, 100)
atlanta.add_route(denver, 160)
boston.add_route(chicago, 120)
boston.add_route(denver, 180)
chicago.add_route(el_paso, 80)
denver.add_route(chicago, 40)
denver.add_route(el_paso, 140)
```

Ниже приведен весь код алгоритма Дейкстры — пожалуй, самый сложный в этой книге. Но если вы готовы внимательно изучить его, читайте дальше.

В нашей реализации главный метод определяется не внутри класса `City`, а вне его. Этот метод принимает два экземпляра класса `City` и возвращает кратчайший путь между ними:

```
def dijkstra_shortest_path(starting_city, final_destination)
  cheapest_prices_table = {}
  cheapest_previous_stopover_city_table = {}

  # Для упрощения кода мы будем использовать простой массив для отслеживания
  # известных городов, которые еще не посетили:
  unvisited_cities = []

  # Для отслеживания посещенных городов мы используем хеш-таблицу.
  # Для этого можно использовать массив, но нам предстоит выполнять поиск,
  # поэтому лучше выбрать хеш-таблицу, так как она эффективнее:
  visited_cities = {}

  # Добавляем название начального города в качестве первого ключа в таблицу
  # cheapest_prices_table. Задаем для этого ключа значение 0,
  # так как ехать нам никуда не надо:
  cheapest_prices_table[starting_city.name] = 0

  current_city = starting_city

  # Этот цикл - ядро алгоритма. Он выполняется, пока у нас
  # не закончатся города для посещения:
  while current_city

    # Добавляем название текущего города (current_city) в хеш-таблицу
    # visited_cities, чтобы отметить его как посещенный, и удаляем его
    # из списка еще не посещенных городов:
    visited_cities[current_city.name] = true
    unvisited_cities.delete(current_city)

  # Перебираем все соседние к текущему города:
  current_city.routes.each do |adjacent_city, price|

    # При обнаружении нового города добавляем его название
    # в список еще не посещенных городов (unvisited_cities):
    unvisited_cities <<
      adjacent_city unless visited_cities[adjacent_city.name]

    # Рассчитываем стоимость поездки из НАЧАЛЬНОГО (starting) города
    # в СОСЕДНИЙ (adjacent), используя ТЕКУЩИЙ (current) в качестве последней
    # пересадки:
    price_through_current_city =
      cheapest_prices_table[current_city.name] + price

    # Если стоимость поездки из НАЧАЛЬНОГО города в СОСЕДНИЙ
    # минимальна на этом этапе...
    if !cheapest_prices_table[adjacent_city.name] ||
```

```
      price_through_current_city <
        cheapest_prices_table[adjacent_city.name]

      # ...обновляем две наши таблицы:
      cheapest_prices_table[adjacent_city.name] =
        price_through_current_city
      cheapest_previous_stopover_city_table[adjacent_city.name] =
        current_city.name
      end
    end

    # Переходим к следующему из еще не посещенных городов. Выбираем тот,
    # куда дешевле всего добраться из НАЧАЛЬНОГО:
    current_city = unvisited_cities.min do |city|
      cheapest_prices_table[city.name]
    end
  end

  # Основная часть алгоритма завершена. В данный момент таблица
  # cheapest_prices_table содержит все минимальные значения стоимости
  # поездки из начального города в остальные. Но, чтобы
  # проложить точный маршрут от начального города
  # до пункта назначения, нужно двигаться дальше.

  # Для построения кратчайшего пути используем простой массив:
  shortest_path = []

  # Чтобы построить кратчайший путь, движемся в обратном направлении,
  # начав с пункта назначения. Поэтому используем пункт
  # назначения в качестве названия текущего города (current_city_name):
  current_city_name = final_destination.name

  # Запускаем цикл, который будет выполняться до достижения начального
  # города:
  while current_city_name != starting_city.name

    # Добавляем каждое обнаруженное значение current_city_name в массив,
    # который используется для построения кратчайшего пути:
    shortest_path << current_city_name

  # Для каждого города находим название соответствующего города последней
  # пересадки в таблице cheapest_previous_stopover_city_table:
  current_city_name =
    cheapest_previous_stopover_city_table[current_city_name]
  end

  # В конце добавляем название начального города к кратчайшему пути:
  shortest_path << starting_city.name

  # Обращаем порядок следования элементов возвращаемого массива,
  # чтобы отобразить весь путь от начала до конца:
  return shortest_path.reverse
end
```

Разберем код подробно.

Функция `dijkstra_shortest_path` принимает две вершины: начальный город (`starting_city`) и пункт назначения (`final_destination`).

В конце эта функция возвратит массив строк с самым дешевым маршрутом. В нашем случае он будет выглядеть так:

```
["Atlanta", "Denver", "Chicago", "El Paso"]
```

Сначала наша функция настраивает две основные таблицы, используемые алгоритмом:

```
cheapest_prices_table = {}
cheapest_previous_stopover_city_table = {}
```

Затем настраиваем способ отслеживания посещенных и непосещенных городов:

```
unvisited_cities = []
visited_cities = {}
```

Может показаться странным, что для отслеживания непосещенных городов (`unvisited_cities`) мы используем массив, а для отслеживания посещенных (`visited_cities`) — хеш-таблицу. Мы выбрали хеш-таблицу, потому что в остальной части кода используем ее только для выполнения поиска, а поиск в ней выполняется гораздо быстрее, чем массиве.

С выбором оптимальной структуры данных для отслеживания непосещенных городов все не так просто. Далее в коде в качестве следующего посещаемого города всегда выбирается тот из непосещенных, до которого дешевле всего добраться из начального. Получается, в идеале нам всегда нужен немедленный доступ к этому самому дешевому варианту из еще не посещенных городов, обеспечить который проще с помощью массива, чем с помощью хеш-таблицы.

Вообще, для этого идеально подошла бы приоритетная очередь, суть которой и заключается в предоставлении быстрого доступа к наименьшему (или наибольшему) значению из набора элементов. Как вы помните из главы 16, для реализации приоритетных очередей лучше всего подходит куча.

Но вместо этого я решил использовать в реализации обычный массив, чтобы сделать код как можно проще и компактнее, так как алгоритм Дейкстры сам по себе достаточно сложный. Но советую вам попробовать заменить массив приоритетной очередью.

Затем мы добавляем в таблицу `cheapest_prices_table` первую пару «ключ — значение», где ключ — название начального города (`starting_city`), а значе-

ние — 0. Опять же, это логично, так как нам не нужно добираться до начального города — мы уже там:

```
cheapest_prices_table[starting_city.name] = 0
```

Для завершения настройки задаем начальный город (`starting_city`) в качестве текущего (`current_city`):

```
current_city = starting_city
```

Ядро алгоритма — цикл, который выполняется, пока мы можем получить доступ к текущему городу (`current_city`). В рамках этого цикла отмечаем текущий город как посещенный, добавляя его название в хеш-таблицу посещенных городов (`visited_cities`). Если такой город есть в списке непосещенных городов (`unvisited_cities`), удаляем его оттуда:

```
while current_city
  visited_cities[current_city.name] = true
  unvisited_cities.delete(current_city)
```

Затем запускаем внутри цикла `while` другой цикл для перебора всех соседних к текущему городу (`current_city`) городов:

```
current_city.routes.each do |adjacent_city, price|
```

Внутри этого вложенного цикла сначала добавляем каждый соседний город в массив `unvisited_cities`, если мы еще его не посещали:

```
unvisited_cities << adjacent_city unless visited_cities[adjacent_city.name]
```

В этой реализации название города может неоднократно попадать в массив непосещенных городов (`unvisited_cities`), и это нормально, так как мы удаляем все эти экземпляры с помощью строки кода `unvisited_cities.delete(current_city)`. Альтернативный подход: убедиться в отсутствии текущего города в массиве непосещенных перед его добавлением туда.

Затем вычисляем минимальную стоимость поездки из начального города в соседний с последней пересадкой в текущем. Для этого находим в таблице `cheapest_prices_table` минимальную стоимость поездки в текущий город и прибавляем к ней стоимость поездки из текущего города в соседний. Результат вычисления сохраняется в переменной `price_through_current_city`:

```
price_through_current_city = cheapest_prices_table[current_city.name] + price
```

Сравниваем значение переменной `price_through_current_city` с текущей минимальной стоимостью поездки из начального города в соседний, указанной в таблице `cheapest_prices_table`. Если соседнего города (`adjacent_city`) еще

нет в таблице, то значение переменной `price_through_current_city` по определению будет минимальной стоимостью соответствующей поездки:

```
if !cheapest_prices_table[adjacent_city.name] ||
  price_through_current_city < cheapest_prices_table[adjacent_city.name]
```

Если значение `price_through_current_city` *оказывается* новой минимальной стоимостью поездки из начального города в соседний, обновляем обе основные таблицы: сохраняем новую стоимость соответствующей поездки в таблице `cheapest_prices_table` и обновляем данные в таблице `cheapest_previous_stopover_city_table`, передав название соседнего города (`adjacent_city`) в качестве ключа и название текущего (`current_city`) в качестве значения:

```
cheapest_prices_table[adjacent_city.name] = price_through_current_city
cheapest_previous_stopover_city_table[adjacent_city.name] = current_city.name
```

После перебора всех соседних к текущему городов посещаем следующий из непосещенных городов. Выбираем тот, куда дешевле всего добраться из начального города, и делаем его новым текущим (`current_city`):

```
current_city = unvisited_cities.min do |city|
  cheapest_prices_table[city.name]
end
```

Если городов для посещения не осталось, значение `current_city` станет равным `nil` и выполнение цикла `while` завершится.

Сейчас обе таблицы содержат все необходимые данные. При желании на этом этапе можно просто вернуть таблицу `cheapest_prices_table` и просмотреть минимальные цены на поездки из начального города во все известные города.

Но нас интересует конкретный маршрут, позволяющий с минимальными затратами добраться из начального города до пункта назначения (`final_destination`).

Поэтому мы создаем массив `shortest_path`, который будет возвращен по окончании работы функции:

```
shortest_path = []
```

И создаем переменную `current_city_name`, начальным значением которой будет название конечного города (`final_destination`):

```
current_city_name = final_destination.name
```

Запускаем цикл `while`, который заполняет данными массив `shortest_path`, двигаясь в обратном направлении от конечного города к начальному:

```
while current_city_name != starting_city.name
```

В рамках этого цикла добавляем название текущего города (`current_city_name`) в массив `shortest_path` и находим в таблице `cheapest_previous_stopover_city_table` название города, где мы должны сделать пересадку перед посещением текущего. Этот город становится новым текущим городом (`current_city_name`):

```
shortest_path << current_city_name
current_city_name = cheapest_previous_stopover_city_table[current_city_name]
```

Чтобы код было проще читать, мы сделали так, чтобы работа цикла завершалась по достижении значения `starting_city`, поэтому теперь вручную добавляем название начального города в конец массива `shortest_path`:

```
shortest_path << starting_city.name
```

На этом этапе массив `shortest_path` содержит все города из кратчайшего маршрута от конечного города до начального. Поэтому обращаем порядок следования элементов возвращенного массива, чтобы отобразить маршрут от начального города до конечного:

```
return shortest_path.reverse
```

Хотя наша реализация имеет дело с городами и ценами, имена переменных можно изменить для нахождения кратчайшего пути в *любом* другом взвешенном графе.

Эффективность алгоритма Дейкстры

Алгоритм Дейкстры — это общий подход к нахождению кратчайшего пути во взвешенном графе, а не конкретная реализация. На самом деле реализовать этот алгоритм можно по-разному.

Например, в коде выше для хранения названий непосещенных городов (`unvisited_cities`) мы использовали простой массив, но, как я уже говорил, вместо него можно было использовать и приоритетную очередь.

Временная сложность этого алгоритма сильно зависит от особенностей его реализации, так что просто проанализируем *нашу*.

При использовании простого массива для отслеживания непосещенных городов (`unvisited_cities`) время выполнения алгоритма может достигать $O(V^2)$. Это связано с тем, что худший сценарий для алгоритма Дейкстры реализуется, когда каждая вершина графа связана со всеми остальными его вершинами. В этом случае при посещении каждой вершины мы проверяем веса ребер, связывающих эту вершину с остальными. В итоге получается V вершин $\times$ V вершин — $O(V^2)$.

Другие реализации, такие как использование приоритетной очереди вместо массива, приводят к более высокой скорости. Опять же, у алгоритма Дейкстры есть несколько вариантов реализации, и временную сложность каждой из них нужно анализировать отдельно.

Но, какую бы реализацию этого алгоритма вы ни выбрали, она позволит вам подойти к обходу графа осмысленно и отыскать кратчайший путь без необходимости находить *все* возможные, а затем выбирать из них самый короткий.

Выводы

В этой главе была представлена последняя структура данных из описанных в книге. Мы подобрались к концу нашего путешествия. Теперь вы знаете, что графы — чрезвычайно мощный инструмент для работы с данными, отражающими связи между разными объектами. Они не только ускоряют код, но и помогают выполнять довольно сложные задачи.

По правде говоря, тема графов достойна отдельной книги. Для работы с этой структурой данных есть много интересных и полезных алгоритмов, в частности алгоритм нахождения минимального остовного дерева, топологическая сортировка, двунаправленный поиск, алгоритмы Флойда — Уоршелла, Беллмана — Форда и раскраска графов. Опираясь на знания, которые вы получили в этой главе, вы сможете изучить эти дополнительные темы самостоятельно.

До сих пор мы уделяли основное внимание скорости работы нашего кода: оценивали его эффективность с точки зрения времени, выраженного через количество шагов, выполняемых алгоритмами.

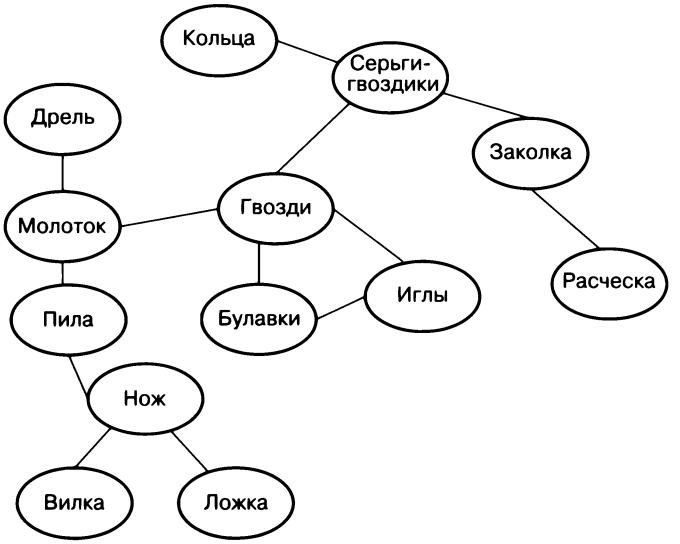
Но эффективность кода выражается не только через скорость, но и, например, через *память*, потребляемую структурой данных или алгоритмом. Поэтому в следующей главе вы научитесь анализировать *пространственную* сложность алгоритмов.

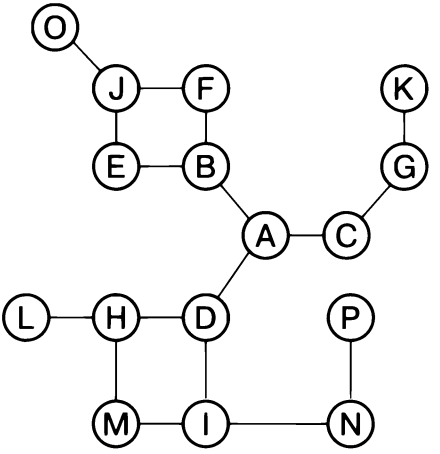
Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 18».

1. Первый из изображенных далее графов лежит в основе механизма рекомендаций интернет-магазина. Каждая вершина — это товар, доступный на сайте. Ребра соединяют каждый товар с другими «похожими», которые будут предложены пользователю при просмотре этого.

Если пользователь интересуется «гвоздями», какие еще товары ему высветятся?





2. Выполняя *поиск в глубину* по второму из графов ниже, начиная с вершины «A», в каком порядке мы обойдем все его вершины? Допустим, при наличии нескольких смежных вершин мы сначала посещаем вершину с той буквой, которая встречается в алфавите раньше всех.

3. Выполняя *поиск в ширину* по предыдущему графу, начиная с вершины «A», в каком порядке мы обойдем все его вершины? Допустим, при наличии

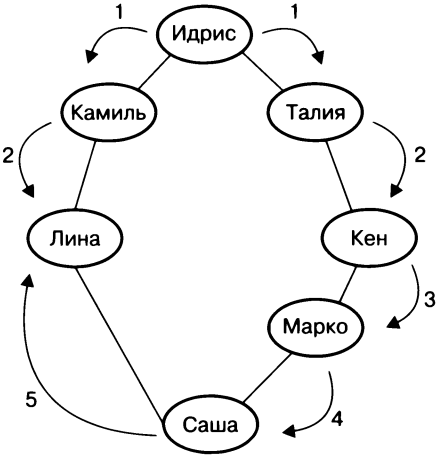
нескольких смежных вершин мы сначала посещаем вершину с той буквой, которая встречается в алфавите раньше всех.

4. В разделе «Поиск в ширину» был код, который выполняет простой *обход* графа в ширину, выводя значение каждой его вершины. Измените его так, чтобы он выполнял *поиск* значения конкретной вершины, переданной функции (как мы это делали при реализации кода для поиска в глубину). При обнаружении искомой вершины функция должна возвращать ее значение. Если же вершина не найдена — значение `null`.

5. В разделе «Алгоритм Дейкстры» мы видели, как этот алгоритм помогает находить кратчайший путь во взвешенном графе. Но понятие кратчайшего пути есть и в обычном графе.

   В случае классического (невзвешенного) графа под кратчайшим путем понимается наименьшее расстояние от одной вершины до другой.

   Это особенно полезно для социальных сетей. Рассмотрим такой пример:



   Вершины Идриса и Лины связаны двумя путями. Для Идриса Лина — контакт второго уровня, если идти к ней через вершину Камиля, и контакт пятого уровня, если следовать через вершину Талии. Итак, если нас интересует, *насколько тесно* Идрис связан с Линой, то из этих двух контактов важнее для нас будет контакт второго уровня.

   Напишите функцию, которая принимает две вершины графа и возвращает кратчайший путь между ними. Она должна возвращать массив с конкретным путем, например, `["Idris", "Kamil", "Lina"]`.

   Подсказка: этот алгоритм может содержать элементы поиска в ширину и алгоритма Дейкстры.

ГЛАВА 19

Работа в условиях ограниченного пространства

До сих пор мы анализировали эффективность разных алгоритмов, опираясь на их скорость — их временную сложность. Но помимо нее есть еще один важный показатель эффективности — *пространственная (или емкостная) сложность* — *объем* потребляемой алгоритмом *памяти*.

Пространственная сложность алгоритма важна, когда память ограничена. Если вы собираетесь работать с огромным объемом данных или пишете программу для небольшого устройства с ограниченной памятью, то пространственная сложность может играть очень важную роль в этом процессе.

В идеальном мире мы всегда использовали бы быстрые и эффективные с точки зрения памяти алгоритмы. Но реальность такова, что иногда мы вынуждены выбирать что-то одно. Чтобы решить, чему отдавать приоритет в конкретной ситуации — скорости или экономии памяти, — нужно провести тщательный анализ.

Выражение пространственной сложности с помощью О-нотации

Любопытно, что для отражения пространственной сложности программисты используют ту же самую О-нотацию.

В главе 3 я описал значение О-нотации как ответ на «ключевой вопрос». В случае с временной сложностью он был такой: *«Сколько шагов будет выполнять алгоритм при наличии N элементов данных?»*

Чтобы использовать О-нотацию для выражения пространственной сложности, нам достаточно переформулировать ключевой вопрос: *«Сколько единиц памяти будет потреблять алгоритм при наличии N элементов данных?»*

Рассмотрим простой пример.

Допустим, мы пишем функцию на языке JavaScript, которая принимает массив строк, а возвращает массив этих же строк в верхнем регистре. То есть при передаче массива `["tuvi", "leah", "shaya", "rami"]` она возвращает `["TUVI", "LEAH", "SHAYA", "RAMI"]`.

Вот один из способов реализации этой функции:

```
function makeUppercase(array) {
  let newArray = [];
  for(let i = 0; i < array.length; i++) {
    newArray[i] = array[i].toUpperCase();
  }
  return newArray;
}
```

Функция `makeUppercase()` принимает массив (`array`) и создает *новый* с именем `newArray`, заполняя его строками из исходного массива, используя символы верхнего регистра.

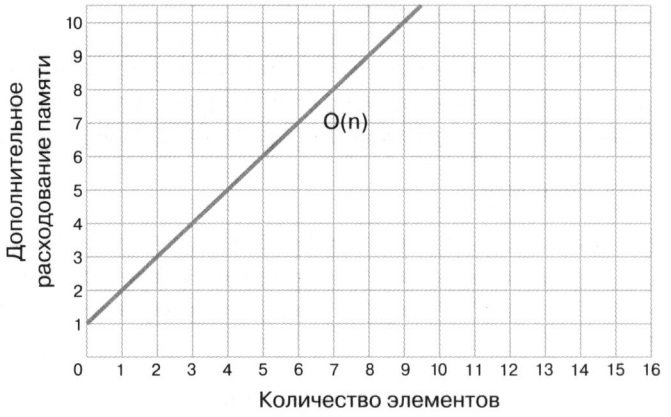
К моменту завершения работы этой функции в памяти компьютера будет два массива — исходный `["tuvi", "leah", "shaya", "rami"]` и новый `["TUVI", "LEAH", "SHAYA", "RAMI"]`.

Если мы проанализируем эту функцию с точки зрения пространственной сложности, то поймем, что она создает новый массив с N элементами *в дополнение* к исходному, в котором тоже N элементов.

Итак, вернемся к нашему ключевому вопросу: *«Сколько единиц памяти будет потреблять алгоритм при наличии N элементов данных?»*

Наша функция сгенерировала дополнительные N элементов данных (в виде нового массива `newArray`), поэтому *пространственная сложность* (*или эффективность*) *этой функции равна* $O(N)$.

Следующий график должен показаться вам знакомым:



Обратите внимание, что этот график такой же, как те, что вы видели в прошлых главах, с временной сложностью O(*N*). Отличие лишь в том, что вертикальная ось теперь отражает *потребляемую память*, а не время.

Теперь рассмотрим альтернативную версию функции `makeUppercase()`, которая использует память более эффективно:

```
function makeUppercase(array) {
  for(let i = 0; i < array.length; i++) {
    array[i] = array[i].toUpperCase();
  }
  return array;
}
```

Здесь мы не создаем новых массивов. Вместо этого мы изменяем исходный *на месте*, последовательно переводя каждую его строку в верхний регистр, после чего возвращаем измененный массив.

Это серьезное улучшение в плане потребления памяти, ведь новая функция вообще *не занимает дополнительную память*.

Как выразить это с помощью O-нотации?

Как вы помните, скорость алгоритма с временной сложностью O(1) остается постоянной вне зависимости от объема данных. Точно так же и пространственная сложность O(1) означает, что алгоритм потребляет одинаковый объем памяти вне зависимости от объема данных.

Наша пересмотренная функция `makeUppercase()` потребляет постоянный объем дополнительного пространства (равный нулю!) независимо от того, сколько

элементов в исходном массиве — четыре или сто. Поэтому пространственная сложность этой функции равна O(1).

Важно отметить, что при использовании O-нотации для описания пространственной сложности мы учитываем только *новые данные*, которые генерирует алгоритм. Например, вторая версия функции `makeUppercase()` тоже имеет дело с *N* элементами данных в виде переданного ей массива, но при оценке ее пространственной сложности с помощью O-нотации мы их не учитываем, так как исходный массив в любом случае существует, а нас интересует только дополнительная память, потребляемая алгоритмом (она еще называется *вспомогательной*).

Но в некоторых источниках могут встречаться и оценки пространственной сложности, учитывающие объем входной информации. Это нормально. Просто знайте, что в этой книге мы его не учитываем, а, увидев оценку пространственной сложности в каком-то другом месте, выясните, содержит ли она исходный массив.

Теперь сравним две версии функции `makeUppercase()` с точки зрения их временной и пространственной сложности:

Версия	Временная сложность	Пространственная сложность
Версия № 1	O(*N*)	O(*N*)
Версия № 2	O(*N*)	O(1)

Итак, временная сложность обеих функций — O(*N*), так как они предполагают выполнение *N* шагов при наличии *N* элементов данных. Но вторая версия более эффективна с точки зрения потребления памяти: ее пространственная сложность равна O(1), а пространственная сложность первой версии — O(*N*).

Итак, версия № 2 более эффективна по сравнению с первой, так как потребляет меньше памяти, не жертвуя при этом скоростью, что очень хорошо.

Компромисс между временем выполнения и занимаемой памятью

Рассмотрим функцию, которая принимает массив и возвращает значение `true`, если находит в нем повторяющиеся значения (мы обсуждали ее в главе 4).

```
function hasDuplicateValue(array) {
    for(let i = 0; i < array.length; i++) {
        for(let j = 0; j < array.length; j++) {
            if(i !== j && array[i] === array[j]) {
```

```
                return true;
            }
        }
    }
    return false;
}
```

Этот алгоритм использует вложенные циклы и выполняется за $O(N^2)$ времени. Назовем эту реализацию версией № 1.

А вот вторая реализация, версия № 2, где используется хеш-таблица и всего один цикл:

```
function hasDuplicateValue(array) {
    let existingValues = {};
    for(let i = 0; i < array.length; i++) {
        if(!existingValues[array[i]]) {
            existingValues[array[i]] = true;
        } else {
            return true;
        }
    }
    return false;
}
```

Здесь мы начинаем с пустой хеш-таблицы `existingValues` и, перебирая все элементы массива, сохраняем каждый новый в качестве ключа (присваиваем ему произвольное значение `true`). Но если мы встречаем элемент, который уже был ключом хеш-таблицы, мы возвращаем `true`, ведь это значит, что мы обнаружили дубликат.

Какой же из этих двух алгоритмов эффективнее? Зависит от того, что для вас важнее — время выполнения или занимаемое пространство. С точки зрения времени выполнения, версия № 2 намного эффективнее, так как выполняется за время $O(N)$, а версия № 1 — за $O(N^2)$.

Но если речь идет о занимаемом *пространстве,* то версия № 1 более эффективна, чем версия № 2. Пространственная сложность второй версии — $O(N)$, так как эта функция создает хеш-таблицу, где могут быть все N значений входного массива. Первая же версия не потребляет дополнительной памяти сверх той, что нужна для хранения исходного массива, поэтому ее пространственная сложность — $O(1)$.

Сравним эти две версии функции `hasDuplicateValue()`:

Версия	Временная сложность	Пространственная сложность
Версия № 1	$O(N^2)$	$O(1)$
Версия № 2	$O(N)$	$O(N)$

Мы видим, что версия № 1 более эффективна в плане потребления памяти, а № 2 — в плане скорости работы. Так какой же из этих алгоритмов следует выбрать?

Ответ зависит от конкретной ситуации. Если нам нужно, чтобы наше приложение работало с молниеносной скоростью, и у нас достаточно памяти, то версия № 2 — отличный вариант. Но когда мы вынуждены экономно расходовать память, а скорость у нас не в приоритете, выбор лучше сделать в пользу версии № 1. Принимая любые технологические решения, требующие компромисса, важно смотреть на картину в целом.

Сравним третью версию той же функции с первыми двумя:

```
function hasDuplicateValue(array) {
    array.sort((a, b) => (a < b) ? -1 : 1);

    for(let i = 0; i < array.length - 1; i++) {
        if (array[i] === array[i + 1]) {
            return true;
        }
    }
    return false;
}
```

Эта реализация, назовем ее версией № 3, начинается с сортировки массива. Затем функция перебирает все значения в массиве, сравнивая каждое из них со следующим значением. Если они равны, значит, мы обнаружили дубликат. Но если мы доходим до конца массива, так и не обнаружив двух одинаковых значений подряд, то понимаем, что дубликатов в нем точно нет.

Проанализируем временную и пространственную сложность версии № 3.

Временная сложность этого алгоритма равна $O(N \log N)$. Допустим, алгоритм сортировки, реализованный на языке JavaScript, выполняется за время $O(N \log N)$ (самые быстрые из известных алгоритмов сортировки работают именно с такой скоростью). Дополнительные N шагов для перебора элементов массива — это совсем немного по сравнению с числом шагов для выполнения сортировки, поэтому в конечном итоге временная сложность может оказаться $O(N \log N)$.

С пространственной сложностью все не так просто, так как разные алгоритмы сортировки потребляют разное количество памяти. Некоторые из тех, с которыми мы имели дело в начале книги, например сортировка пузырьком и выбором, не занимают дополнительное пространство, поскольку вся сортировка происходит на месте. Но более быстрые алгоритмы используют некоторое количество дополнительной памяти по причинам, о которых мы поговорим чуть

позже. Например, пространственная сложность большинства реализаций алгоритма Quicksort равна O(log *N*).

Итак, сравним версию № 3 с двумя предыдущими:

Версия	Временная сложность	Пространственная сложность
Версия № 1	$O(N^2)$	$O(1)$
Версия № 2	$O(N)$	$O(N)$
Версия № 3	$O(N \log N)$	$O(\log N)$

Версия № 3 позволяет достичь определенного баланса между временем выполнения и занимаемой памятью. С точки зрения времени, версия № 3 работает быстрее, чем № 1, но медленнее, чем № 2. В плане занимаемого места она эффективнее, чем № 2, но менее эффективна по сравнению с версией № 1.

Выходит, мы можем выбрать версию № 3 в случаях, когда время выполнения и занимаемое пространство *одинаково* важны для нас.

В любом случае, в каждой конкретной ситуации нам нужно определиться с минимально допустимой скоростью и предельным потреблением памяти. Только после этого мы сможем выбрать подходящий алгоритм.

До сих пор мы говорили о том, как алгоритмы могут занимать дополнительное пространство при создании новых фрагментов данных, например массивов или хеш-таблиц. Но алгоритм может занимать пространство, даже если не делает ничего такого. И это нельзя игнорировать.

Скрытые издержки рекурсии

Мы уже много говорили о рекурсивных алгоритмах. Но давайте рассмотрим еще одну простую рекурсивную функцию:

```
function recurse(n) {
  if (n < 0) { return; }

  console.log(n);
  recurse(n - 1);
}
```

Она принимает число `n` и ведет обратный отсчет от него до 0, выводя последовательно уменьшающиеся значения на экран.

На первый взгляд эта простая рекурсия кажется безобидной. Ее временная сложность равна O(*N*), так как число рекурсивных вызовов функции соответ-

ствует величине `n`. Она не создает новых структур данных, поэтому не занимает дополнительного места.

Или все-таки занимает?

О том, как работает рекурсия, мы говорили в главе 10. Там вы узнали, что каждый раз, когда функция рекурсивно вызывает саму себя, в стек добавляется элемент, позволяющий компьютеру вернуться к внешней функции по завершении работы внутренней.

Если мы передадим в нашу рекурсивную функцию число 100, она добавит вызов `recurse(100)` перед вызовом `recurse(99)`, а затем добавит `recurse(99)` в стек перед вызовом `recurse(98)`.

На момент вызова `recurse(-1)` в стеке будет 101 элемент, от `recurse(100)` до `recurse(0)`.

Несмотря на то что стек вызовов в итоге будет раскручен, для хранения этих 100 элементов в нем нам нужен определенный объем памяти. Получается, что *пространственная сложность нашей рекурсивной функции равна* $O(N)$. N здесь — это число, переданное функции. Если мы передаем 100, нам придется временно хранить 100 элементов в стеке вызовов.

Из этого вытекает важный принцип: *рекурсивная функция занимает единицу пространства при каждом рекурсивном вызове*. Таким хитрым способом рекурсия буквально пожирает память компьютера: даже если функция не создает новые данные, в процессе самой рекурсии данные добавляются в стек вызовов.

Чтобы правильно рассчитать объем памяти, потребляемой рекурсивной функцией, нужно определить максимальный размер стека вызовов.

В нашем случае размер стека будет примерно равен величине переданного функции числа `n`.

Это может показаться незначительным. В конце концов, современные компьютеры могут обработать несколько элементов стека вызовов, верно? Что ж, посмотрим.

Когда я передаю число 20 000 функции `recurse` на своем стильном современном ноутбуке, *он не может его обработать*. 20 000 не кажется большим числом. Но при запуске `recurse(20000)` происходит следующее.

Мой компьютер выводит на экран числа от `20000` до `5387`, а затем отображает сообщение об ошибке:

```
RangeError: Maximum call stack size exceeded
```

Поскольку рекурсия продолжалась с `20000` примерно до `5000` (я округляю `5387` в меньшую сторону), размер стека вызовов предположительно достиг значения 15 000, когда компьютер столкнулся с нехваткой памяти. Оказывается, он не способен обработать стек вызовов с более чем 15 000 элементов.

Это *сильно* ограничивает возможности применения рекурсии, так как я не могу передать своей замечательной функции `recurse` число, превышающее 15 000!

Сравним этот процесс с использованием простого цикла:

```
function loop(n) {
  while (n >= 0) {
    console.log(n);
    n--;
  }
}
```

Эта функция решает ту же задачу, применяя вместо рекурсии обычный цикл.

Функция не использует рекурсию и не занимает дополнительную память, поэтому может работать с огромными числами, не приводя к нехватке места. При передаче этой функции больших чисел на ее выполнение может уйти некоторое время. Но она справится со своей задачей и не сдастся преждевременно, как рекурсивная функция.

Теперь понятно, почему пространственная сложность алгоритма быстрой сортировки — $O(\log N)$. Он выполняет $O(\log N)$ рекурсивных вызовов, поэтому максимальный размер стека вызовов достигает значения $\log(N)$.

Получается, прежде чем реализовать функцию с помощью рекурсии, нужно проанализировать все плюсы и минусы этого подхода. Рекурсия позволяет использовать «волшебный» нисходящий способ мышления, описанный в главе 11, но нам нужно, чтобы функция выполнила свою задачу. А если мы обрабатываем много данных или просто имеем дело с большим числом вроде 20 000, рекурсия может не справиться.

Опять же, я не призываю вас вовсе отказываться от рекурсии. Я лишь говорю, что в каждой конкретной ситуации следует взвешивать все плюсы и минусы того или иного алгоритма.

Выводы

Итак, вы знаете, как оценивать эффективность алгоритмов с точки зрения времени выполнения *и* занимаемого пространства. Теперь у вас достаточно инфор-

мации, чтобы сравнивать алгоритмы между собой и принимать обоснованные решения о том, какой подход использовать при создании вашего приложения.

Поскольку вы уже можете принимать решения самостоятельно, пришло время перейти к последнему пункту нашего путешествия. В заключительной главе я поделюсь с вами несколькими советами по оптимизации кода и опишу ряд реалистичных сценариев, которые мы оптимизируем вместе.

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 19».

1. Ниже приведен алгоритм составителя слов из одноименного раздела главы 7. Опишите его пространственную сложность с помощью О-нотации:

```
function wordBuilder(array) {
  let collection = [];

  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      if (i !== j) {
        collection.push(array[i] + array[j]);
      }
    }
  }

  return collection;
}
```

2. Ниже приведена функция, которая обращает порядок следования элементов массива. Опишите ее пространственную сложность с помощью О-нотации:

```
function reverse(array) {
  let newArray = [];

  for (let i = array.length - 1; i >= 0; i--) {
    newArray.push(array[i]);
  }

  return newArray;
}
```

3. Создайте новую функцию для обращения порядка элементов массива, пространственная сложность которой равна O(1).

4. Ниже приведены три разные реализации функции, которая принимает массив чисел и возвращает массив с теми же числами, умноженными на 2. Например, при передаче массива `[5, 4, 3, 2, 1]` эта функция возвращает `[10, 8, 6, 4, 2]`.

```
function doubleArray1(array) {
  let newArray = [];

  for(let i = 0; i < array.length; i++) {
    newArray.push(array[i] * 2);
  }

  return newArray;
}

function doubleArray2(array) {
  for(let i = 0; i < array.length; i++) {
    array[i] *= 2;
  }

  return array;
}

function doubleArray3(array, index=0) {
  if (index >= array.length) { return; }

  array[index] *= 2;
  doubleArray3(array, index + 1);

  return array;
}
```

Заполните следующую таблицу, описав эффективность этих трех версий с точки зрения времени выполнения и занимаемого пространства:

Версия	Временная сложность	Пространственная сложность
Версия № 1	?	?
Версия № 2	?	?
Версия № 3	?	?

ГЛАВА 20

Оптимизация кода

Мы прошли долгий путь. Теперь у вас достаточно знаний для анализа временной и пространственной сложности алгоритмов, применяемых к разным структурам данных, благодаря чему вы сможете писать быстрый, эффективный и простой код.

В последней главе я поделюсь с вами дополнительными приемами оптимизации кода. Иногда бывает сложно понять, как улучшить алгоритм. За годы работы я сформулировал для себя определенные стратегии, которые помогают находить возможности для повышения эффективности кода. Надеюсь, они пригодятся и вам.

Предварительное условие: определение текущей эффективности

Прежде чем мы начнем обсуждать методы оптимизации кода, поговорим о том, что нужно сделать *до того*, как приступать к этому процессу.

Обязательное условие перед проведением оптимизации — *определение текущей эффективности кода*. Вы ведь не можете ускорить работу алгоритма, если не знаете, насколько быстро он работает сейчас.

К этому моменту у вас уже должно сложиться четкое представление об О-нотации и разных категориях алгоритмической сложности. Как только вы выясните, к какой из них принадлежит ваш алгоритм, можете приступать к оптимизации.

В оставшейся части этой главы я буду называть этап определения текущей эффективности алгоритма «выполнением предварительного условия».

Определение лучшей эффективности из возможных

Все методы из этой главы очень полезны, но вскоре вы увидите, что некоторые из них особенно эффективны в одних ситуациях, а некоторые — в других.

Тем не менее, метод, о котором я расскажу сейчас, применим ко *всем* алгоритмам и должен быть первым шагом в процессе их оптимизации.

Вот в чем его суть.

После определения эффективности алгоритма (после выполнения предварительного условия) представьте себе то, что я называю «лучшей мыслимой эффективностью» (некоторые называют это «лучшим мыслимым временем выполнения» или «лучшим воображаемым О-большим», говоря о скорости работы алгоритма).

По сути, лучшая мыслимая эффективность — это максимальное значение, выраженное с помощью О-нотации, к которому нужно стремиться при выполнении поставленной задачи: та степень эффективности, превзойти которую невозможно.

Например, для написания функции, которая выводит все элементы массива, лучшая мыслимая эффективность алгоритма будет O(N). Учитывая, что нам нужно вывести каждый из N элементов массива на экран, *у нас нет другого выбора,* кроме как обработать все N элементов. С этим ничего нельзя поделать, ведь придется «затронуть» каждый элемент, чтобы вывести его на экран. Поэтому O(N) — лучшая мыслимая эффективность в этом сценарии.

Получается, что при оптимизации алгоритма нужно определить *два* показателя с помощью О-нотации: *текущую* эффективность нашего алгоритма (выполнить предварительное условие) и его *максимально возможную* эффективность при решении поставленной задачи.

Если эти значения не совпадают, значит, нам есть что оптимизировать. Например, если мой текущий алгоритм реализуется за время O(N^2), а поставленную задачу в принципе можно выполнить за время O(N), то мне еще есть к чему стремиться. Разрыв между этими двумя показателями отражает выгоду, которую мы можем получить за счет оптимизации.

Итак, вам нужно сделать следующее.

1. Определить категорию сложности текущего алгоритма (это предварительное условие).

2. Определить лучшую мыслимую эффективность алгоритма, выполняющего поставленную задачу.
3. Если эти оценки не совпадают, попытаться оптимизировать код, чтобы максимизировать эффективность используемого алгоритма.

Важно подчеркнуть, что *не всегда можно достичь лучшей мыслимой эффективности*. Вы можете о чем-то мечтать, но это не значит, что ваши идеи обязательно воплотятся в реальность.

Бывает и так, что текущую реализацию вообще нельзя оптимизировать. Но лучшая мыслимая эффективность может стать для нас неким ориентиром в процессе оптимизации.

Обычно мне удается улучшить алгоритм до той степени, которая находится *между* текущим и максимально возможным показателями эффективности.

Например, если текущая версия моего алгоритма выполняется за время $O(N^2)$, а лучшая временная сложность из возможных — $O(\log N)$, я постараюсь оптимизировать алгоритм до $O(\log N)$. Если в результате мой код будет выполняться *хотя бы* за $O(N)$, это уже будет большим успехом, которого я смог достичь благодаря определению лучшей мыслимой эффективности.

Развитие воображения

Как вы уже видели, определение лучшей эффективности из возможных дает нам цель, к которой нужно стремиться при оптимизации. Чтобы извлечь из этого приема максимум пользы, задействуйте воображение и определите лучшую мыслимую эффективность, которую можно было бы назвать *поразительной*. Советую вам ориентироваться на максимальную эффективность из принципиально возможных.

Чтобы разжечь воображение, я использую следующий прием. Выбираю *по-настоящему* удивительный показатель временной сложности своего алгоритма и спрашиваю себя: «Если бы кто-нибудь сказал мне, что знает, как выполнить эту задачу с такой поразительной эффективностью, поверил бы я ему?» Если да, то я использую этот показатель как лучшую эффективность из возможных.

После определения текущего и целевого показателя алгоритмической сложности мы можем приступать к оптимизации алгоритма.

В оставшейся части главы мы обсудим дополнительные методы и стратегии, которые помогут повысить эффективность кода.

Волшебные поиски

Это один из моих любимых приемов оптимизации. Я просто спрашиваю себя: «Если бы я мог каким-то волшебным способом находить нужную информацию за время O(1), то смог бы я ускорить свой алгоритм?» Если да, я использую структуру данных (как правило, хеш-таблицу), чтобы воплотить это волшебство в жизнь. Я называю эту технику «волшебными поисками».

Посмотрим, как она работает, на конкретном примере.

Волшебный поиск авторов книг

Допустим, мы пишем программу для библиотек и у нас есть данные о книгах и их авторах в двух отдельных массивах.

Массив с именами авторов (`authors`) выглядит так:

```
authors = [
  {"author_id" => 1, "name" => "Virginia Woolf"},
  {"author_id" => 2, "name" => "Leo Tolstoy"},
  {"author_id" => 3, "name" => "Dr. Seuss"},
  {"author_id" => 4, "name" => "J. K. Rowling"},
  {"author_id" => 5, "name" => "Mark Twain"}
]
```

Это массив хеш-таблиц, каждая из которых содержит имя и идентификатор автора.

У нас есть еще один массив — в нем содержатся данные о книгах:

```
books = [
  {"author_id" => 3, "title" => "Hop on Pop"},
  {"author_id" => 1, "title" => "Mrs. Dalloway"},
  {"author_id" => 4, "title" => "Harry Potter and the Sorcerer's Stone"},
  {"author_id" => 1, "title" => "To the Lighthouse"},
  {"author_id" => 2, "title" => "Anna Karenina"},
  {"author_id" => 5, "title" => "The Adventures of Tom Sawyer"},
  {"author_id" => 3, "title" => "The Cat in the Hat"},
  {"author_id" => 2, "title" => "War and Peace"},
  {"author_id" => 3, "title" => "Green Eggs and Ham"},
  {"author_id" => 5, "title" => "The Adventures of Huckleberry Finn"}
]
```

Как и в `authors`, в массиве `books` есть несколько хеш-таблиц. Каждая из них содержит название книги и `author_id`, с помощью которого мы можем определить имя автора книги, используя данные из массива `authors`. Например, книге *Hop on Pop* соответствует значение `author_id`, равное 3. Значит, автор этой книги Dr. Seuss (доктор Сьюз), так как в массиве `authors` его имени присвоен идентификатор 3.

Теперь предположим, что мы хотим написать код, объединяющий эту информацию в одном массиве:

```
books_with_authors = [
  {"title" => "Hop on Pop", "author" => "Dr. Seuss"}
  {"title" => "Mrs. Dalloway", "author" => "Virginia Woolf"}
  {"title" => "Harry Potter and the Sorcerer's Stone", "author" => "J. K.
Rowling"}
  {"title" => "To the Lighthouse", "author" => "Virginia Woolf"}
  {"title" => "Anna Karenina", "author" => "Leo Tolstoy"}
  {"title" => "The Adventures of Tom Sawyer", "author" => "Mark Twain"}
  {"title" => "The Cat in the Hat", "author" => "Dr. Seuss"}
  {"title" => "War and Peace", "author" => "Leo Tolstoy"}
  {"title" => "Green Eggs and Ham", "author" => "Dr. Seuss"}
  {"title" => "The Adventures of Huckleberry Finn", "author" => "Mark Twain"}
]
```

Для этого нам нужно перебрать массив `books` и связать каждую книгу с соответствующим автором. Как это реализовать?

Один из возможных подходов — использование вложенных циклов. При этом внешний цикл будет перебирать книги, для каждой из которых будет запускаться внутренний, перебирающий авторов до обнаружения того, у кого будет соответствующий идентификатор. Так выглядит реализация этого подхода на языке Ruby:

```
def connect_books_with_authors(books, authors)
  books_with_authors = []

  books.each do |book|
    authors.each do |author|
      if book["author_id"] == author["author_id"]
        books_with_authors <<
          {title: book["title"], author: author["name"]}
      end
    end
  end

  return books_with_authors
end
```

Прежде чем оптимизировать этот код, выполним предварительное условие и определим текущую эффективность нашего алгоритма.

Временная сложность алгоритма равна $O(N \times M)$, ведь чтобы найти подходящего автора для каждой из N книг, нам нужно перебрать M авторов.

Теперь посмотрим, можем ли мы оптимизировать наш алгоритм.

Опять же, первое, что нужно сделать, — определить лучшую мыслимую эффективность. Здесь нам в любом случае придется перебрать все N книг, поэтому

превзойти показатель O(N), судя по всему, не удастся. Так как O(N) — самая высокая скорость из возможных, будем считать ее лучшей мыслимой эффективностью.

Теперь мы готовы применить технику «волшебных поисков». Задаем себе вопрос из начала раздела: «Если бы я мог каким-то волшебным способом находить нужную информацию за время O(1), то смог бы я ускорить свой алгоритм?»

Применим этот подход к нашему сценарию. Сейчас мы используем внешний цикл, перебирающий все книги, для каждой из которых мы запускаем внутренний, который пытается отыскать идентификатор автора этой книги (`author_id`) в массиве `authors`.

Но что, если бы нам удалось найти автора *за время* O(1)? Что, если бы вместо перебора *всех* авторов при обработке каждой книги мы могли бы найти нужного за один шаг? Это сильно ускорило бы работу нашего алгоритма за счет исключения внутреннего цикла и приблизило ее к показателю эффективности O(N).

Теперь, когда мы выяснили, что эта волшебная функция поиска может нам помочь, попробуем воплотить эту магию в жизнь.

Дополнительная структура данных

Один из самых простых способов реализовать эту волшебную функцию поиска — добавление в код дополнительной структуры данных, которая позволит хранить информацию особым образом, чтобы мы смогли быстро находить то, что нужно. Во многих случаях для этого идеально подходит хеш-таблица: поиск в ней выполняется за время O(1) (см. главу 8).

Пока хеш-таблицы с именами авторов хранятся в массиве, мы всегда будем выполнять M шагов (где M — количество авторов) для нахождения конкретного идентификатора `author_id` в этом массиве. Но если мы сохраним ту же информацию в отдельной хеш-таблице, то получим «волшебную» способность находить любого автора за время O(1).

Таблица будет выглядеть примерно так:

```
author_hash_table =
{1 => "Virginia Woolf", 2 => "Leo Tolstoy", 3 => "Dr. Seuss", 4 => "J. K.
Rowling", 5 => "Mark Twain"}
```

Ключ здесь — идентификатор автора, а значение — его имя.

Итак, оптимизируем алгоритм так, чтобы он сначала перемещал данные из массива `authors` в эту хеш-таблицу и только потом перебирал книги в цикле:

```
def connect_books_with_authors(books, authors)
  books_with_authors = []
  author_hash_table = {}

  # Преобразование массива с именами авторов в хеш-таблицу:
  authors.each do |author|
    author_hash_table[author["author_id"]] = author["name"]
  end

  books.each do |book|
    books_with_authors <<
      {"title" => book["title"], "author" => author_hash_table[book["author_
id"]]}
  end

  return books_with_authors
end
```

Сначала мы перебираем массив `authors` и используем данные из него для создания хеш-таблицы `author_hash_table`. На это уходит M шагов, где M — число авторов.

Затем перебираем список книг и используем хеш-таблицу `author_hash_table` для нахождения имени нужного за один шаг. Этот цикл выполняет N шагов, где N — количество книг.

Итак, временная сложность оптимизированного алгоритма равна $O(N + M)$, так как он выполняет один цикл для обработки N книг и еще один для обработки M авторов. Значит, он работает гораздо быстрее, чем наша исходная версия, которая выполнялась за время $O(N \times M)$.

Важно отметить, что при создании еще одной хеш-таблицы мы используем дополнительное пространство $O(M)$, тогда как наш первоначальный алгоритм вообще не занимал дополнительную память. Но эта оптимизация — отличный вариант, если мы готовы пожертвовать местом ради ускорения работы алгоритма.

Итак, мы получили волшебную способность, сначала вообразив себе поиск, выполняемый за время $O(1)$, а затем реализовав эту идею с помощью хеш-таблицы, позволяющей хранить данные в подходящем для поиска виде.

В том, что хеш-таблицы позволяют находить данные за время $O(1)$, нет ничего нового — мы обсуждали это еще в главе 8. Здесь я просто делюсь конкретным приемом, суть которого — *представить*, как именно выполнение поиска за время $O(1)$ повлияет на скорость работы кода. Представив потенциальные выгоды такого молниеносного поиска, вы можете попробовать использовать хеш-таблицу или другую структуру данных для воплощения этой мечты в реальность.

Проблема двух сумм

Рассмотрим еще один сценарий, который поможет извлечь выгоду из волшебного поиска. Кстати, это один из моих любимых примеров оптимизации.

Проблема двух сумм — известное упражнение по программированию. Задача в том, чтобы написать функцию, которая принимает массив чисел и возвращает значение `true` или `false` в зависимости от того, есть ли в массиве пара чисел, которые в сумме дают 10 (или другое заданное число). Чтобы было проще, предположим, что в массиве никогда не бывает повторяющихся значений.

Допустим, у нас есть массив:

```
[2, 0, 4, 1, 7, 9]
```

Наша функция вернет `true`, так как `1` и `9` в сумме дают 10.

При передаче массива:

```
[2, 0, 4, 5, 3, 9]
```

эта функция вернет `false`. Несмотря на то что три числа — `2`, `5` и `3` — в сумме дают 10, нам нужно найти *два*.

Первое решение, которое приходит на ум, сводится к использованию вложенных циклов для сравнения всех чисел друг с другом и выяснения того, дают они в сумме 10 или нет. Так выглядит реализация этой функции на языке JavaScript:

```
function twoSum(array) {
  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      if(i !== j && array[i] + array[j] === 10) {
        return true;
      }
    }
  }
  return false;
}
```

Прежде чем пытаться что-то оптимизировать, нужно выполнить предварительное условие и оценить текущую эффективность нашего кода.

Как и любой алгоритм, использующий вложенный цикл, эта функция выполняется за время $O(N^2)$.

Чтобы узнать, стоит ли оптимизировать алгоритм, нужно сравнить текущую эффективность с максимально возможной.

Судя по всему, здесь нам придется посетить каждое число в массиве хотя бы раз. Поэтому максимально возможная эффективность будет равна O(*N*). И если бы кто-то сказал мне, что эту задачу можно выполнить за время O(*N*), я бы поверил. Итак, давайте считать O(*N*) лучшей мыслимой эффективностью алгоритма.

Теперь зададим себе вопрос: «Если бы я мог каким-то волшебным способом находить нужную информацию за время O(1), то смог бы я ускорить свой алгоритм?»

Иногда полезно задаться этим вопросом в процессе анализа текущей реализации. Так мы и сделаем.

Представим работу внешнего цикла на примере массива `[2, 0, 4, 1, 7, 9]`. Сначала цикл обрабатывает число 2.

Что нас может интересовать при обработке этого числа? Согласно условиям задачи, мы хотим узнать, можно ли прибавить это число к другому в массиве, чтобы в сумме получить 10.

Получается, обращаясь к 2, мы хотели бы узнать, есть ли *среди элементов массива число 8*. Если бы у нас был волшебный способ узнать, есть ли в этом массиве 8, за время O(1), то мы немедленно вернули бы `true`. Назовем число 8 *дополняющим* к числу 2, так как в сумме они дают 10.

Точно так же при обработке числа 0 мы хотели бы найти в массиве дополняющее его значение 10 за время O(1) и т. д.

При таком подходе мы можем перебрать элементы массива только раз, попутно выполняя волшебный поиск дополняющих значений за время O(1). При обнаружении в массиве числа, дополняющего текущее, мы возвращаем `true`, а если доходим до конца массива, не обнаружив таких чисел, — `false`.

Теперь, когда мы оценили потенциальную выгоду от выполнения волшебных поисков за время O(1), попробуем провернуть наш трюк с добавлением новой структуры данных. Опять же, обычно по умолчанию применяется хеш-таблица (частота, с которой ее используют для ускорения работы алгоритмов, удивительна).

Поскольку мы хотим находить любое число в массиве за время O(1), будем хранить эти числа в качестве ключей в хеш-таблице, которая может выглядеть так:

```
{2: true, 0: true, 4: true, 1: true, 7: true, 9: true}
```

Значение ключей может быть произвольным, пусть это будет `true`.

Теперь мы можем найти любое число за время O(1). Как нам отыскать дополняющее число? Ранее мы заметили, что при обработке числа 2 дополняющим его значением должно быть 8, потому что 2 + 8 = 10.

По сути, мы можем вычислить значение, дополняющее любое число с помощью вычитания его из 10. Поскольку 10 – 2 = 8, 8 — дополняющее число к 2.

Теперь у нас есть все для создания по-настоящему быстрого алгоритма:

```
function twoSum(array) {
  let hashTable = {};

  for(let i = 0; i < array.length; i++) {
    // Проверяем, есть ли в хеш-таблице ключ, при добавлении которого
    // к текущему числу получается 10:
    if(hashTable[10 - array[i]]) {
      return true;
    }

    // Сохраняем каждое число в хеш-таблице в качестве ключа:
    hashTable[array[i]] = true;
  }

  // Возвращаем false, если доходим до конца массива,
  // не обнаружив ни одного числа, дополняющего другие до 10:
  return false;
}
```

Этот алгоритм перебирает каждое число в массиве по одному разу.

Последовательно фокусируясь на каждом из этих чисел, мы проверяем, есть ли в хеш-таблице ключ, значение которого дополняет текущее число до 10. Мы вычисляем это с помощью кода `10 - array[i]` (например, если значение `array[i]` — 3, то дополняющим его числом будет 7, так как 10 – 3 = 7).

При обнаружении дополняющего числа мы немедленно возвращаем `true`, что говорит об обнаружении двух чисел, сумма которых равна 10.

В процессе перебора чисел мы вставляем каждое из них в хеш-таблицу в качестве ключа. Так по мере обработки массива мы заполняем таблицу значениями.

Этот подход позволил увеличить скорость работы алгоритма до O(*N*). Мы добились этого, сохранив все элементы данных в хеш-таблице, чтобы иметь возможность выполнять поиск в цикле за время O(1).

Станьте волшебником в мире программирования, вооружившись своей волшебной палочкой — хеш-таблицей (ладно, хватит об этом).

Выявление закономерностей

Одна из самых полезных стратегий как для оптимизации кода, так и для разработки алгоритмов — выявление закономерностей в задаче. Часто это помогает разработать простой алгоритм для выполнения сложной задачи.

Игра с монетками

В качестве примера рассмотрим так называемую игру с монетками, где два игрока по очереди убирают из кучи одну или две монетки. *Проигрывает* тот, кто уберет последнюю. Весело, правда?

Оказывается, в этой игре есть свои закономерности, и при использовании правильной стратегии можно *заставить* противника взять последнюю монету. Разберем этот процесс на примере небольшой кучки монет.

Если в кучке осталась одна монета, проигрывает тот, чья очередь ходить, так как у него нет другого выбора, кроме как взять ее.

Если две, то игрок, который ходит, может выиграть, взяв только одну монету, тем самым заставив противника взять последнюю.

Когда в кучке лежит три монеты, тот, чья очередь ходить, может уберечь себя от проигрыша, убрав две монеты и заставив противника взять последнюю.

Если же в кучке осталось четыре монеты, то у игрока возникает проблема. Если он уберет одну, противнику достанется кучка из трех монет, что даст возможность выиграть, как было показано выше. Точно так же, если игрок уберет две монеты, у противника тоже останется две, а это тоже позволит ему добиться победы.

Допустим, нам нужно написать функцию, вычисляющую наши шансы на победу при заданном количестве монет в исходной куче. Как это сделать? Если задуматься, то для вычисления точного результата для любого заданного количества монет можно было бы использовать подзадачи. Так, естественным подходом для решения основной задачи была бы нисходящая рекурсия.

Вот реализация этого рекурсивного подхода на языке Ruby:

```
def game_winner(number_of_coins, current_player="you")
  if number_of_coins <= 0
    return current_player
  end

  if current_player == "you"
    next_player = "them"
  elsif current_player == "them"
    next_player = "you"
  end
```

```
    if game_winner(number_of_coins - 1, next_player) == current_player ||
      game_winner(number_of_coins - 2, next_player) == current_player
      return current_player
    else
      return next_player
    end
  end
```

Мы сообщаем функции `game_winner` исходное количество монет (`number_of_coins`) и указываем игрока, чья очередь делать ход (это либо вы (`"you"`), либо ваш противник (`"them"`)). Затем функция возвращает значение `"you"` или `"them"` в качестве вероятного победителя. Когда функция вызывается впервые, вы (`"you"`) будете тем, кто ходит (`current_player`).

Мы определяем базовый случай, когда `current_player` достается количество монет, меньшее или равное нулю. Значит, другой игрок взял последнюю монету, а тот, кто ходит, по умолчанию становится победителем.

Теперь определяем переменную `next_player`, которая отслеживает очередность ходов и показывает, кто из игроков будет ходить следующим.

Затем рекурсивно вызываем функцию `game_winner` для кучек монет, где на одну и две монеты меньше, чем в текущей, и смотрим, чем закончится игра для следующего игрока (`next_player`) в этих сценариях. Если соперник проиграет в обоих сценариях, значит, `current_player` выиграет.

Это был сложный алгоритм, но мы справились. Теперь разберемся с тем, как его улучшить.

Чтобы выполнить предварительное условие, сначала нужно оценить текущую эффективность алгоритма.

Вы, скорее всего, заметили, что эта функция делает несколько рекурсивных вызовов. Если вы уже немного напряглись, то не безосновательно: временная сложность этой функции — $O(2^N)$, то есть она работает чрезвычайно медленно.

Мы можем усовершенствовать ее с помощью мемоизации (см. главу 12). Это позволит довести скорость до $O(N)$, где N — исходное число монет. Улучшение будет существенным.

Но давайте посмотрим, можно ли еще как-то ускорить работу нашего алгоритма.

Чтобы оценить возможности дальнейшей оптимизации алгоритма, нужно определить его лучшую мыслимую эффективность.

Поскольку N — это всего лишь одно число, я могу представить, как такой алгоритм выполняется за время $O(1)$, ведь нам не нужно перебирать N элементов в массиве или что-то в этом роде. Если бы кто-то сказал мне, что ему удалось

создать алгоритм для игры с монетками, который выполняется за время O(1), я бы ему поверил. Итак, мы будем стремиться к показателю O(1).

Но как его достичь? В этом нам поможет выявление закономерностей.

Генерация примеров

Как правило, каждая задача выполняется по определенному шаблону, но я нашел шаблон, применимый к *любой* задаче. Его суть в том, чтобы *сгенерировать множество входных примеров,* вычислить на их основе выходные данные и поискать в результатах закономерности.

Применим этот подход к нашему случаю.

Если мы вычислим победителя игры для разного количества монет в кучке (от 1 до 10), то получим следующую таблицу:

Количество монет	Победитель
1	Противник
2	Вы
3	Вы
4	Противник
5	Вы
6	Вы
7	Противник
8	Вы
9	Вы
10	Противник

Здесь мы наблюдаем очевидную закономерность. По сути, каждое третье число, начиная с 1, обеспечивает победу противнику. В остальных случаях победителем оказываетесь вы.

Итак, если мы вычтем из общего количества монет 1, каждое число, которое без остатка делится на 3, будет соответствовать случаю, когда выигрывает ваш противник. Итак, мы можем определить победителя с помощью единственной операции деления:

```
def game_winner(number_of_coins)
  if (number_of_coins - 1) % 3 == 0
    return "them"
  else
    return "you"
  end
end
```

Здесь мы видим, что если после вычитания 1 количество монет (`number_of_coins`) делится на 3 без остатка, то победителем становится ваш противник (`"them"`). В остальных случаях побеждаете вы (`"you"`).

Этот алгоритм предусматривает выполнение единственной математической операции, поэтому его временная и пространственная сложность равна O(1). А еще эта версия гораздо проще предыдущей, так что это действительно беспроигрышный вариант.

Итак, после генерации множества примеров (входных данных) и выявления победителя (выходные данные) мы смогли найти в этой игре закономерность, с помощью которой добрались до сути задачи и превратили чрезвычайно медленный алгоритм в молниеносный.

Перестановка чисел для уравнивания сумм (задача о sum swap)

Теперь рассмотрим пример, где для оптимизации алгоритма можно *комбинировать* волшебный поиск с выявлением закономерностей.

Допустим, мы хотим написать функцию, которая принимает два массива целых чисел. Пусть они выглядят так:

```
array_1 = [5,  3,  2,  9,  1]  Сумма: 20
array_2 = [1,  12,  5]         Сумма: 18
```

Сейчас сумма чисел в первом массиве (`array_1`) равна 20, а во втором (`array_2`) — 18.

Наша функция должна найти по одному числу в каждом массиве, перестановка которых позволяет уравнять суммы элементов обоих.

Если мы поменяем местами 2 из первого массива и 1 из второго в нашем примере, то получим:

```
array_1 = [ 5, 3, (1) 9, 1]   Сумма: 19
array_2 = [(2) 12, 5]         Сумма: 19
```

В результате сумма чисел в обоих массивах станет равной 19.

Чтобы не усложнять, обойдемся без фактической перестановки и сделаем так, чтобы функция просто возвращала массив с двумя индексами, соответствующими числам, которые нужно поменять местами. Итак, мы должны поменять местами значение по индексу `2` массива `array_1` и значение по индексу `0` масси-

ва `array_2`, поэтому функция возвращает массив `[2, 0]`. Если уравнять суммы массивов невозможно, функция должна вернуть значение `nil`.

Один из способов реализации этого алгоритма — использование вложенных циклов. То есть внешний цикл должен перебирать числа массива `array_1`, для каждого из которых будет запускаться внутренний, перебирающий числа массива `array_2` и сравнивающий суммы элементов обоих, которые получаются при перестановке этих двух чисел.

Прежде чем приступить к оптимизации этого алгоритма, нужно выполнить предварительное условие и определить его текущую эффективность.

Этот алгоритм использует вложенные циклы, предполагающие перебор M элементов второго массива при обработке каждого из N элементов первого, поэтому его временная сложность — $O(N \times M)$ (я использую N и M, потому что массивы могут быть разных размеров).

Можно ли ускорить его работу? Чтобы это выяснить, определим его максимально возможную эффективность.

Судя по всему, нам придется хотя бы раз перебрать все элементы двух массивов, так как нам нужно выяснить их значения. Но вполне вероятно, что этим можно и *ограничиться*. Если это так, то лучшая мыслимая эффективность алгоритма — $O(N + M)$. К этому показателю мы и будем стремиться.

Теперь попробуем выявить в этой задаче скрытые закономерности. Опять же, лучший способ для этого — сгенерировать множество примеров.

Итак, рассмотрим ряд примеров, где перестановка двух чисел приводит к уравниванию сумм элементов двух массивов:



Здесь можно заметить ряд закономерностей. Некоторые из них могут показаться очевидными, но давайте все же их проанализируем.

Одна из закономерностей в том, что для уравнивания сумм нужно поменять местами большее число из большего массива и меньшее из меньшего.

Вторая закономерность: в результате одной перестановки сумма элементов обоих массивов меняется на одинаковую величину. Например, если поменять местами 7 и 4, сумма элементов одного массива *уменьшится* на 3, а другого — *увеличится* на 3.

Третья любопытная закономерность: перестановки всегда приводят к тому, что суммы элементов двух массивов оказываются *ровно посередине* между их исходными значениями.

Например, в первом случае исходная сумма элементов массива `array_1` была равна 18, а массива `array_2` — 12. При правильном обмене элементов их суммы стали равны 15, а это ровно посередине между 18 и 12.

Если подумать, то третья закономерность — логическое продолжение первых двух. Поскольку при перестановке значений суммы элементов массивов меняются на одинаковую величину, *единственная* точка, где эти суммы уравниваются, находится точно посередине между исходными значениями.

Выходит, если мы знаем суммы элементов двух массивов, то можем посмотреть любое число в одном из них и определить, с каким именно его нужно поменять местами.

Вернемся к нашему примеру:

```
array_1 = [5, 3, 3, 7]   Сумма: 18
array_2 = [4, 1, 1, 6]   Сумма: 12
```

Мы знаем, что для успешного обмена суммы двух массивов должны оказаться ровно посередине между значениями 18 и 12, то есть стать равными 15.

Посмотрим на разные числа из первого массива и определим, какое мы хотели бы поменять. Как и раньше, будем называть второе число дополняющим. Начнем с первого числа первого массива (`array_1`) — 5.

С каким числом нужно поменять местами это значение? Итак, мы знаем, что в результате перестановки правильных чисел сумма элементов массива `array_1` должна уменьшиться на 3, а массива `array_2` — увеличиться на 3, поэтому нам нужно поменять местами 5 и 2. Так получилось, что во втором массиве нет чис-

ла 2, поэтому мы не можем поменять местами 5 из первого массива с числом из второго, чтобы уравнять суммы их элементов.

Следующее число в первом массиве — 3. Его нужно поменять местами с 0 из второго массива, чтобы уравнять суммы их элементов. Увы, во втором массиве нет числа 0.

Последнее число в первом массиве — 7. Чтобы суммы элементов обоих массивов были равны 15, нужно поменять местами 7 и 4. К счастью, во втором массиве есть значение 4 и мы можем произвести успешную перестановку.

Как же выразить эту закономерность в коде?

Для начала определим, насколько должна измениться сумма элементов массива:

```
shift_amount = (sum_1 - sum_2) / 2
```

Здесь `sum_1` — это сумма элементов первого массива (`array_1`), а `sum_2` — второго (`array_2`). Если значение `sum_1` равно 18, а `sum_2` — 12, их разность равна 6. Поделив ее на 2, мы можем вычислить значение `shift_amount` — то, насколько должна измениться сумма элементов каждого массива.

Значение `shift_amount` равно 3. Значит, сумма элементов второго массива должна увеличиться на 3, чтобы достичь целевого значения (а сумма элементов первого должна *уменьшиться* на 3).

Итак, начнем построение алгоритма с вычисления сумм элементов двух массивов. После этого переберем в цикле все числа одного из массивов и найдем в другом дополняющее число.

Например, если бы мы перебирали числа второго массива, то знали бы, что текущее число нужно поменять местами со значением из первого массива, равным сумме текущего числа и значения `shift_amount`. Например, если текущее число равно 4, то для определения дополняющего мы добавляем к нему значение `shift_amount` (3) и получаем 7. Значит, для уравнивания сумм нужно поменять местами текущее число второго массива с числом 7 из первого.

Итак, мы выяснили, что можем определить дополняющее значение числа из одного массива в другом. Но зачем? Мы ведь все еще используем вложенные циклы, из-за которых алгоритм выполняется за время $O(N \times M)$, ведь при обработке каждого числа одного массива мы должны перебрать другой в поисках дополняющего значения.

Здесь мы можем вспомнить о волшебных поисках и спросить себя: «Если бы я мог каким-то волшебным способом находить нужную информацию за время $O(1)$, то смог бы я ускорить свой алгоритм?»

Действительно, если бы мы могли находить дополняющее число в другом массиве за время O(1), наш алгоритм работал бы намного быстрее. И мы можем реализовать этот мгновенный поиск с помощью старой доброй хеш-таблицы.

Если сначала мы сохраним числа из одного массива в хеш-таблице, то сможем находить любое из них за время O(1) при переборе чисел другого массива.

Так выглядит полная версия этого алгоритма:

```
def sum_swap(array_1, array_2)
  # Хеш-таблица для хранения значений первого массива:
  hash_table = {}
  sum_1 = 0
  sum_2 = 0

  # Вычисляем сумму элементов первого массива, сохраняя его значения
  # в хеш-таблице вместе с индексами
  array_1.each_with_index do |num, index|
    sum_1 += num
    hash_table[num] = index
  end

  # Вычисляем сумму элементов второго массива:
  array_2.each do |num|
    sum_2 += num
  end

  # Вычисляем, на сколько должна измениться сумма элементов второго массива:
  shift_amount = (sum_1 - sum_2) / 2

  # Перебираем все числа во втором массиве:
  array_2.each_with_index do |num, index|

    # Проверяем хеш-таблицу на предмет наличия дополняющего числа
    # в первом массиве, значение которого равно сумме текущего числа
    # и величины, на которую должна измениться сумма элементов массива:
    if hash_table[num + shift_amount]
      return [hash_table[num + shift_amount], index]
    end
  end

  return nil
end
```

Такой алгоритм работает намного быстрее, чем его исходная версия с временной сложностью $O(N \times M)$. Если N — это первый массив, а M — второй, то алгоритм выполняется за время $O(N + M)$. Хотя мы перебираем второй массив дважды и технически временная сложность равна $O(N + 2M)$, мы сводим это выражение к $O(N + M)$, так как O-нотация игнорирует константы.

Этот алгоритм занимает дополнительное место $O(N)$, так как мы копируем все N чисел из первого массива в хеш-таблицу. При этом мы жертвуем памятью, чтобы выиграть время, но все отлично, если скорость у нас в приоритете.

В любом случае, это еще один пример того, как выявление закономерностей позволяет разработать простое и быстрое решение сложной задачи.

Жадные алгоритмы

Следующая тактика позволяет ускорить некоторые из самых упрямых алгоритмов. Такой подход работает не всегда, но, когда он срабатывает, ситуация меняется кардинально.

Поговорим о написании жадных алгоритмов.

Но сначала разберемся, что означает этот странный термин. *Жадный алгоритм* на каждом шаге выбирает то, что представляется лучшим вариантом *в данный момент*. Чтобы понять смысл этой фразы, рассмотрим простой пример.

Максимальный элемент массива

Напишем алгоритм нахождения наибольшего числа в массиве. Один из способов это сделать — использовать вложенные циклы и сравнить каждое число с остальными элементами массива. Обнаружение значения, превышающего остальные, говорит о нахождении максимального элемента.

Учитывая использование вложенных циклов, временная сложность такого алгоритма будет равна $O(N^2)$.

Другой подход — сортировка элементов массива в порядке возрастания и возвращение его конечного значения. При использовании алгоритма быстрой сортировки, вроде Quicksort, это займет время $O(N \log N)$.

Есть и третий вариант — использование жадного алгоритма:

```
def max(array)
  greatest_number = array[0]

  array.each do |number|
    if number > greatest_number
      greatest_number = number
    end
  end

  return greatest_number
end
```

Первая строка функции предполагает, что первое число в массиве наибольшее (`greatest_number`). Это жадное допущение, которое объявляет первое число наибольшим, потому что это число максимальное из найденных до сих пор.

Конечно же, оно еще и *единственное* из увиденных до сих пор, но именно это и делает жадный алгоритм — выбирает лучший вариант на основе информации, которой он обладает сейчас.

Теперь перебираем все числа в массиве. Обнаружив значение, превышающее `greatest_number`, делаем его новым наибольшим числом. Это допущение тоже жадное, так как на каждом шаге алгоритм выбирает лучший вариант на основе имеющейся информации.

Этот алгоритм напоминает ребенка в кондитерском магазине, который хватает первую попавшуюся конфету, но, как только замечает конфету побольше, бросает первую и берет новую.

Но этот на первый взгляд наивный подход на самом деле работает. К моменту завершения выполнения функции у переменной `greatest_number` действительно будет значение максимального элемента массива.

Жадность нельзя назвать хорошим качеством в социальном плане, но она может творить чудеса с точки зрения скорости работы алгоритма. Алгоритм выше выполняется за время O(N), так как мы обрабатываем каждое число массива только один раз.

Наибольшая сумма элементов подраздела массива

Рассмотрим еще один пример жадного алгоритма.

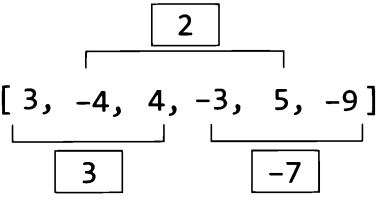
Нам нужно написать функцию, которая принимает массив чисел и возвращает наибольшую сумму элементов любого «подраздела» массива.

Например, рассмотрим следующий массив:

```
[3, -4, 4, -3, 5, -9]
```

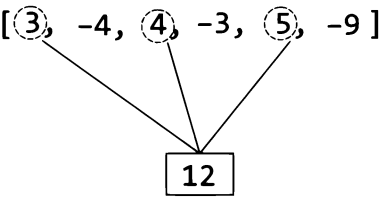
Если бы мы вычисляли сумму всех чисел в нем, то получили бы –4.

Но мы можем вычислить сумму элементов *подразделов* массива:

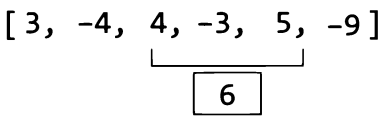


Подраздел — это *непрерывная последовательность элементов массива*: несколько чисел, идущих *подряд*.

Следующие числа *не* образуют подраздел, так как не идут подряд:



Наша задача — найти наибольшую сумму элементов *любого* подраздела массива. Здесь эта сумма — 6, и она получена из следующего подраздела:



Чтобы нам было проще, допустим, что массив содержит хотя бы одно положительное число.

Как же написать код для вычисления наибольшей суммы элементов подраздела массива?

Один из подходов — вычислить сумму всех подразделов массива и выбрать наибольшую из них. Но в массиве из N элементов может быть около $N^2/2$ подразделов, поэтому только на создание разных подразделов уйдет $O(N^2)$ времени.

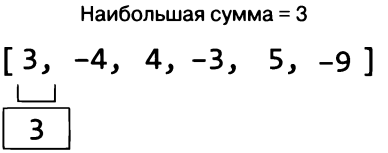
И снова начнем с определения лучшей мыслимой эффективности. Нам точно придется проверить каждое число хотя бы раз, поэтому мы никак не сможем превзойти показатель $O(N)$. Итак, сделаем $O(N)$ нашей целью.

На первый взгляд эта цель кажется недостижимой. Как вообще вычислить сумму элементов нескольких подразделов при однократном переборе массива?

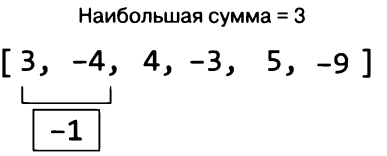
Посмотрим, что произойдет, если мы немного пожадничаем...

Жадный алгоритм здесь будет пытаться «схватить» наибольшую сумму на каждом этапе перебора массива. Так это может выглядеть на примере прошлого массива.

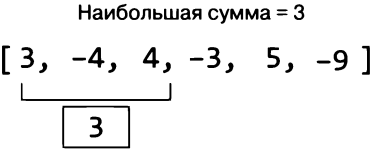
В самом начале мы сталкиваемся с числом 3 и делаем жадное допущение, что искомая наибольшая сумма равна 3:



Затем обнаруживаем число –4. Прибавляя его к предыдущему (3), получаем текущую сумму, равную –1. Итак, 3 все еще наибольшая сумма:

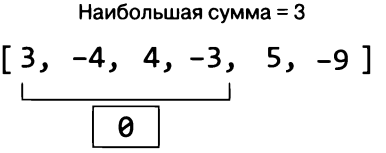


Мы нашли число 4. Прибавляя его к текущей сумме, получаем 3:



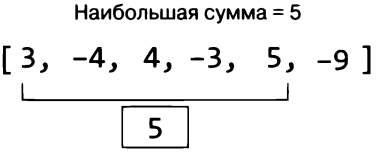
Сейчас 3 все еще наибольшая сумма.

Находим число –3, и наша текущая сумма теперь равна 0:

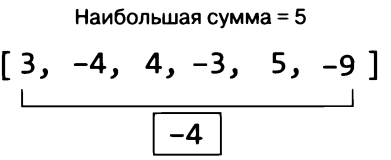


Опять же, несмотря на то что *текущая* сумма равна 0, *наибольшая* все еще равна 3.

Мы нашли число 5, и текущая сумма теперь равна 5. Охваченные жадностью, мы объявляем ее наибольшей на данный момент:

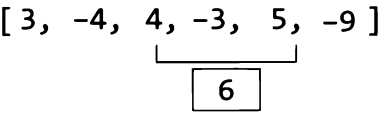


Мы достигаем последнего числа, равного –9, и наша текущая сумма уменьшается до –4:



По достижении конца массива наша наибольшая сумма будет равна 5. Может показаться, что при использовании жадного подхода наш алгоритм в итоге должен вернуть число 5.

Но на самом деле 5 *не* наибольшая сумма элементов подраздела массива. В нем есть подраздел, элементы которого в сумме дают 6:

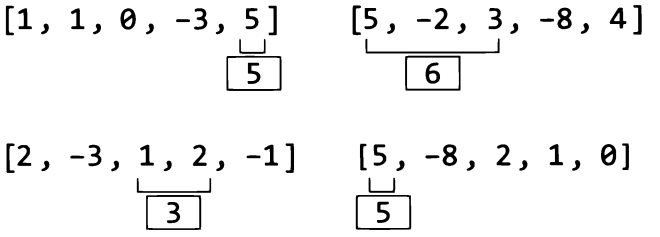


Проблема алгоритма в том, что он находит наибольшую сумму только для подразделов, которые всегда начинаются с первого элемента массива. Но есть и другие, которые могут начинаться с других элементов. А мы их не учли.

Выходит, жадный алгоритм не оправдал наших надежд.

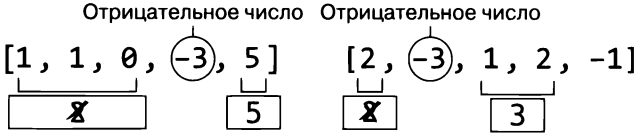
Но сдаваться рано! Чтобы жадный алгоритм работал как следует, нужно просто немного его подправить.

Посмотрим, поможет ли нам выявление в этом какой-нибудь закономерности (обычно помогает). Как уже было сказано, лучший способ найти закономерность — создать множество примеров. Итак, создадим несколько массивов с наибольшими суммами элементов подразделов и попробуем найти в них что-нибудь интересное:



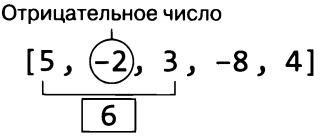
Возникает интересный вопрос: «Почему в одних случаях подраздел с наибольшей суммой находится в начале массива, а в других — нет?»

Как видите, подраздел с наибольшей суммой находится *не* в начале массива из-за отрицательного числа, которое прерывает последовательность:



То есть наибольший подраздел *мог бы* начаться с первого элемента массива, но отрицательное число прервало последовательность, и он начался с более позднего элемента.

Но подождите секунду. В некоторых случаях наибольший подраздел *включает* отрицательное число, которое не прерывает последовательность:



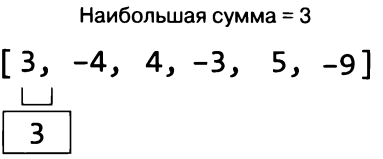
В чем же разница?

Закономерность такая: если отрицательное число приводит к тому, что сумма предыдущего подраздела отрицательная, последовательность прерывается. Но если отрицательное число просто уменьшает сумму текущего подраздела, но при этом ее значение остается положительным, последовательность не прерывается.

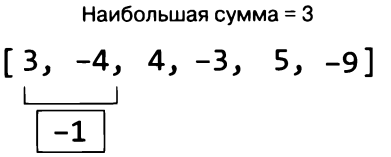
Это вполне логично. Когда при переборе массива сумма текущего подраздела становится меньше 0, *лучше просто обнулить текущую сумму*, иначе текущая отрицательная сумма уменьшит значение искомой наибольшей.

Итак, настроим наш жадный алгоритм с учетом этой закономерности.

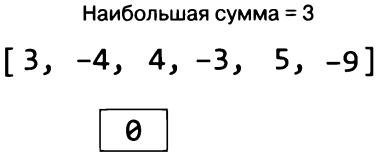
Снова начнем с числа 3. Текущая наибольшая сумма равна 3:



Мы обнаруживаем число –4, и текущая сумма становится равной –1:



Мы пытаемся найти подраздел с наибольшей суммой, а текущая сумма отрицательная, поэтому, прежде чем переходить к следующему числу, обнулим ее:

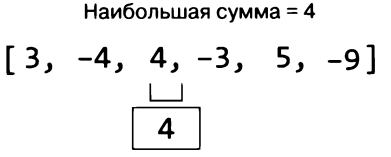


С этого следующего числа начнется новый подраздел.

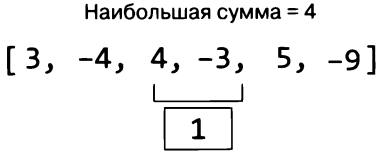
Опять же, идея в том, что, если следующее число положительное, мы можем просто начать новый подраздел с него, не позволяя текущему отрицательному числу понизить значение суммы его элементов. Поэтому будем обнулять текущую отрицательную сумму и считать следующее положительное число *началом* нового подраздела.

Продолжим.

Мы нашли число 4, которое становится началом нового подраздела. Получается, что текущая сумма равна 4, и сейчас она наибольшая:



Теперь мы находим число –3, и текущая сумма становится равной 1:



Затем мы сталкиваемся с числом 5. При этом текущая сумма увеличивается до 6 и становится наибольшей:



Наконец, мы нашли число –9, и текущая сумма становится равной –3. На этом этапе нужно бы обнулить ее, но мы достигли конца массива, поэтому можем заключить, что наибольшая сумма элементов его подраздела равна 6. И действительно, так оно и есть.

Так выглядит реализация этого алгоритма:

```
def max_sum(array)
  current_sum = 0
  greatest_sum = 0

  array.each do |num|
    # Если текущая сумма отрицательная, обнуляем ее:
    if current_sum + num < 0
      current_sum = 0
    else
      current_sum += num

      # Жадно допускаем, что текущая сумма наибольшая, если это
      # максимальная из тех, с которыми мы сталкивались до сих пор:
      greatest_sum = current_sum if current_sum > greatest_sum
    end
  end

  return greatest_sum
end
```

Жадный алгоритм позволяет решить эту довольно сложную задачу за время $O(N)$, так как предполагает однократный перебор элементов массива. Это серьезное улучшение по сравнению с исходной версией, которая выполнялась за $O(N^2)$ времени. Пространственная сложность этого алгоритма — $O(1)$, так как мы не генерируем никаких дополнительных данных.

Хотя выявление закономерности помогло нам найти правильное решение, благодаря «жадному» образу мышления мы изначально знали, какую именно закономерность ищем.

Жадные предсказания цен на акции

Рассмотрим еще один жадный алгоритм.

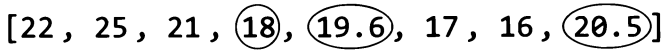
Допустим, мы пишем финансовое приложение, прогнозирующее цены акций. Алгоритм, над которым мы работаем, выявляет восходящий тренд в динамике цен на некую акцию.

Если конкретнее, то мы разрабатываем функцию, которая принимает массив цен на акцию и устанавливает факт наличия в нем любых *трех* цен, образующих восходящий тренд.

Рассмотрим следующий массив цен на некую акцию за определенный период времени:

```
[22, 25, 21, 18, 19.6, 17, 16, 20.5]
```

Несмотря на то что это трудно определить на глаз, в массиве есть три цены, которые формируют восходящий тренд:



Если мы будем идти слева направо, то обнаружим три цены, из которых «крайняя правая» цена больше «средней», а средняя больше «крайней левой».

В следующем массиве нет трех цен, образующих восходящий тренд:

```
[50, 51.25, 48.4, 49, 47.2, 48, 46.9]
```

Наша функция должна возвращать `true`, если массив содержит три цены, образующие восходящий тренд, и `false`, если не содержит.

Как же реализовать такую функцию?

Один из способов — использовать три вложенных цикла: пока один последовательно обрабатывает каждую цену, второй перебирает все последующие. При этом каждая итерация второго цикла сопровождается запуском третьего вложенного цикла, который проверяет все цены, следующие за второй. Обрабатывая каждую комбинацию из трех цен, мы проверяем, расположены ли они в порядке возрастания. Обнаружив такую комбинацию, мы возвращаем `true`. А если мы завершаем работу циклов, так и не выявив восходящего тренда, — `false`.

Временная сложность этого алгоритма равна $O(N^3)$. Он очень медленный! Можно ли его оптимизировать?

Сначала определимся с лучшей мыслимой эффективностью. Нам точно придется проверить каждую цену акции, чтобы выявить восходящий тренд, поэтому эффективность нашего алгоритма не может превышать O(*N*). Теперь попробуем приблизиться к этому показателю.

Пришло время снова пожадничать.

Итак, чтобы решить поставленную задачу, нам нужно как-то попытаться последовательно «захватывать» то, что сейчас кажется нам низшей, средней и высшей точками восходящего тренда.

Вот что мы для этого сделаем.

Допустим, первая цена в массиве — самая низкая точка восходящего тренда, образуемого тремя ценами.

Среднюю цену мы инициализируем числом, которое будет гарантированно превышать максимальное значение массива. Для этого мы сделаем ее равной бесконечности (многие языки программирования поддерживают концепцию бесконечности). Этот шаг может показаться странным, но чуть позже вы поймете, почему нужно сделать именно так.

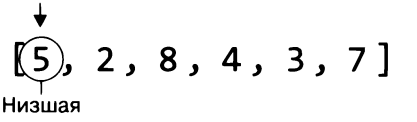
Затем мы выполним один проход по массиву, руководствуясь следующими правилами.

1. Если текущая цена ниже самой низкой известной, то она становится новой самой низкой ценой.
2. Если текущая цена выше самой низкой цены, но ниже средней, то она становится новой средней ценой.
3. Если текущая цена выше самой низкой и средней цен, значит, мы обнаружили восходящий тренд, образуемый тремя ценами!

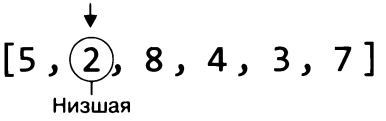
Рассмотрим все это на примере. Допустим, у нас есть следующий массив цен:

```
[5, 2, 8, 4, 3, 7]
```

Начинаем перебирать элементы массива со значения 5 и жадно допускаем, что 5 — низшая из трех цен, образующих восходящий тренд.



Затем переходим к числу 2. Оно меньше 5, поэтому мы предполагаем, что 2 — низшая из трех цен, образующих восходящий тренд:



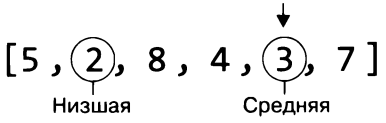
Находим в массиве число 8. Оно больше текущей низшей цены, поэтому мы оставляем ее равной 2. Но значение 8 меньше текущей средней цены, которая равна бесконечности, поэтому мы жадно допускаем, что 8 — *средняя* из трех цен, образующих восходящий тренд:



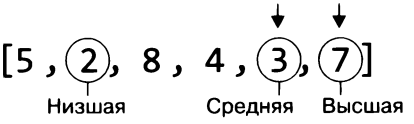
Теперь мы нашли число 4. Оно больше 2, поэтому мы все еще предполагаем, что 2 — самая низкая точка нашего тренда. Но 4 меньше 8, поэтому мы меняем значение средней точки с 8 на 4. Это тоже из-за жадности: снижая значение средней точки, мы увеличиваем свои шансы на обнаружение в массиве более высокой цены, формирующей искомую тенденцию. Итак, новое значение средней цены — 4:



Следующее число в массиве — 3. Мы оставляем низшую цену равной 2, так как значение 3 превышает ее. Но 3 меньше 4, поэтому делаем его нашей новой средней ценой:



Наконец, мы обнаруживаем число 7 — последнее значение в массиве. Поскольку 7 превышает нашу среднюю цену (3), массив содержит три цены, образующие восходящий тренд, и наша функция может вернуть значение `true`:



Обратите внимание, что в массиве есть две комбинации цен, образующих такой тренд: 2-3-7 и 2-4-7. Но это не важно, ведь нас интересует только факт наличия в массиве *любой* такой комбинации. Поэтому обнаружения одной из них уже достаточно для того, чтобы возвратить `true`.

Вот как выглядит реализация этого алгоритма:

```
def increasing_triplet?(array)
  lowest_price = array[0]
  middle_price = Float::INFINITY

  array.each do |price|
    if price <= lowest_price
      lowest_price = price

    # Если текущая цена выше самой низкой,
    # но ниже средней:
    elsif price <= middle_price
      middle_price = price

    # Если текущая цена выше средней:
    else
      return true
    end
  end

  return false
end
```

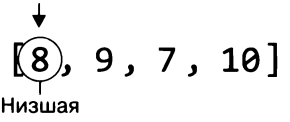
У этого алгоритма есть один нелогичный аспект, на который стоит обратить внимание: в некоторых сценариях вам может показаться, что он не работает, хотя это не так.

Рассмотрим один из таких:

```
[8, 9, 7, 10]
```

Что произойдет, если мы применим наш алгоритм к этому массиву?

Сначала 8 станет низшей точкой:



Затем 9 — средней точкой:



После мы находим значение 7, которое становится новой низшей точкой:



Потом мы достигаем значения 10:



Поскольку 10 превышает текущую среднюю точку (9), наша функция возвращает `true`. Это правильный ответ, так как наш массив действительно содержит тренд, образуемый значениями 8-9-10. Но к моменту завершения работы функции в переменной, соответствующей низшей точке, будет храниться значение 7. А оно не входит в восходящий тренд!

Несмотря на это функция вернула правильный ответ, потому что все, что от нее требовалось, — найти значение, превышающее среднюю точку. Средняя точка была найдена после обнаружения более низкой, поэтому дальнейшее нахождение числа, превышающего среднюю точку, говорит о том, что в массиве есть комбинация значений, образующих восходящий тренд. Это верно, даже несмотря на то, что в процессе прохода по массиву мы заменили значение низшей точки другим числом.

В любом случае, наш жадный подход окупился, поскольку при выполнении задачи мы сумели обойтись однократным перебором элементов массива. Это поразительное улучшение: нам удалось перевести алгоритм из категории $O(N^3)$ в $O(N)$.

Конечно, жадный подход работает *не всегда*. Но это еще один инструмент, который вы можете попробовать применить для оптимизации своих алгоритмов.

Замена структуры данных

Еще один прием оптимизации — представить, что произойдет, если мы используем для хранения информации другую структуру данных.

Например, мы можем работать над задачей, в которой данные предоставляются нам в виде массива. Но переосмысление тех же данных в виде хеш-таблицы, дерева или другой структуры иногда открывает неожиданные возможности для оптимизации.

Описанное ранее использование хеш-таблицы для реализации функции волшебного поиска — пример использования именно этого приема. Но далее вы увидите, что замена структуры данных может помочь и в других ситуациях.

Алгоритм для проверки анаграмм

Рассмотрим конкретный пример. Допустим, мы пишем функцию, которая определяет, являются ли две заданные строки анаграммами друг друга. Мы говорили об анаграммах в одном из разделов главы 11, но там имели дело с функцией, которая создавала все анаграммы заданной строки. Здесь мы просто будем сравнивать две строки и возвращать `true`, если они анаграммы друг друга, и `false`, если нет.

В принципе, для этой задачи можно использовать ту же функцию генерации анаграмм: создать все анаграммы первой строки и посмотреть, совпадает ли вторая с какой-либо из них. Но для N символов в строке количество анаграмм всегда будет равно $N!$, поэтому алгоритм будет выполняться за время $O(N!)$ — безумно медленно.

Вы уже знаете, что надо делать. Прежде чем мы приступим к оптимизации кода, определим его лучшую мыслимую эффективность.

Итак, нам обязательно придется хотя бы раз обработать каждый символ обеих строк. Эти строки могут быть разного размера, поэтому их однократная обработка займет $O(N + M)$ времени. Я не могу представить, как выполнить эту задачу быстрее, поэтому именно к такому показателю мы и будем стремиться.

Подумаем, как достичь этой цели.

Второй возможный подход предполагает использование вложенных циклов для сравнения двух строк. Например, при обработке каждого символа первой строки во внешнем цикле мы сравниваем текущий с каждым символом второй. Обнаружив совпадение, мы можем удалить символ из второй строки. Идея в том, что, если каждый символ из первой строки есть и во второй, то к моменту завершения работы внешнего цикла мы удалим все символы из второй строки.

Итак, если по завершении работы цикла во второй строке остались символы, значит, строки не анаграммы друг друга. Если мы все еще перебираем символы первого слова, но уже удалили всю вторую строку, это значит то же самое. Но если к моменту завершения работы цикла все символы второй строки будут удалены, то эти две строки действительно анаграммы друг друга.

Так выглядит реализация этого алгоритма на языке Python:

```
def areAnagrams(firstString, secondString):
  # Преобразуем вторую строку (secondString) в массив, чтобы удалять
  # из нее символы, так как в Python строки неизменяемы:
  secondStringArray = list(secondString)

  for i in range(0, len(firstString)):

    # Если мы еще не закончили перебор символов первой строки,
    # а массив символов второй уже пуст:
    if len(secondStringArray) == 0:
      return False

    for j in range(0, len(secondStringArray)):

      # Если мы находим в обеих строках один символ:
      if firstString[i] == secondStringArray[j]:

        # Удаляем этот символ из массива второй строки
        # и возвращаемся во внешний цикл:
        del secondStringArray[j]
        break

  # Две строки - анаграммы друг друга, только если к моменту
  # завершения процесса перебора символов первой строки
  # массив символов второй оказывается пустым:
  return len(secondStringArray) == 0
```

Важно отметить, что удаление элементов массива при его обработке в цикле может приводить к ошибкам: делать это неправильно — все равно, что рубить сук, на котором сидишь. Но даже если ошибок не будет, алгоритм все равно будет выполняться за время $O(N \times M)$. Это намного быстрее, чем $O(N!)$, но гораздо медленнее нашего целевого показателя $O(N + M)$.

Еще более быстрый подход — сортировка двух строк. Если в результате сортировки две строки абсолютно одинаковые, значит, они — анаграммы друг друга, если разные, то нет.

Этот подход требует $O(N \log N)$ времени для обработки каждой строки с использованием алгоритма быстрой сортировки вроде Quicksort. Поскольку две исходные строки могут быть разного размера, итоговая временная сложность алгоритма будет равна $O(N \log N + M \log M)$. Это быстрее, чем $O(N \times M)$, но давайте не будем останавливаться на достигнутом. Мы ведь стремимся к $O(N + M)$, помните?

В такой ситуации может пригодиться использование альтернативной структуры данных. Сейчас мы имеем дело со строками, но давайте представим, что бы произошло при сохранении этих строк в какой-нибудь другой структуре.

Мы *могли бы* сохранить строку в виде массива отдельных символов. Но это нам никак бы не помогло.

Теперь представим эту строку в виде хеш-таблицы. Как бы она выглядела?

Один из вариантов — хеш-таблица, где каждый символ сохраняется в качестве ключа, значение которого — количество экземпляров этого символа в слове. Например, строка `"balloon"` будет выглядеть так:

```
{"b" => 1, "a" => 1, "l" => 2, "o" => 2, "n" => 1}
```

Согласно этой таблице, в строке есть одна буква `"b"`, одна `"a"`, две буквы `"l"`, две `"o"` и одна буква `"n"`.

В этой хеш-таблице есть *не вся* информация. Например, по ней мы не можем определить *порядок* символов в строке. Так мы теряем часть данных.

Но *именно это* и позволяет определить, являются ли две строки анаграммами друг друга. То есть две строки — анаграммы друг друга, если содержат одинаковое количество вхождений каждого из символов, вне зависимости от порядка их следования.

Возьмем, к примеру, слова «rattles», «startle» и «starlet». Во всех есть две буквы `"t"`, одна `"a"`, одна `"l"`, одна `"e"` и одна `"s"`, что и делает их анаграммами друг друга.

Теперь мы можем реализовать алгоритм, который преобразует каждую строку в хеш-таблицу, содержащую число вхождений каждого из символов. После преобразования строк остается только сравнить две хеш-таблицы. Если они идентичны, значит, исходные строки — анаграммы друг друга.

Его реализация выглядит так:

```
def areAnagrams(firstString, secondString):
  firstWordHashTable = {}
  secondWordHashTable = {}

# Создаем хеш-таблицу из первой строки:
for char in firstString:
  if firstWordHashTable.get(char):
    firstWordHashTable[char] += 1
  else:
    firstWordHashTable[char] = 1

# Создаем хеш-таблицу из второй строки:
```

```
for char in secondString:
  if secondWordHashTable.get(char):
    secondWordHashTable[char] += 1
  else:
    secondWordHashTable[char] = 1

# Две строки - анаграммы, только если две хеш-таблицы идентичны:
return firstWordHashTable == secondWordHashTable
```

Здесь мы перебираем все символы в обеих строках только один раз, что требует выполнения $N + M$ шагов.

Проверка равенства значений в хеш-таблицах может потребовать дополнительных $N + M$ шагов. Это касается и языка JavaScript, который позволяет производить такое сравнение только путем перебора всех пар «ключ — значение» в этих хеш-таблицах. Но в итоге все равно получается $2(N + M)$ шагов, что сводится к $O(N + M)$, а это намного быстрее по сравнению с любым из наших предыдущих подходов.

Справедливости ради отмечу, что создание хеш-таблиц требует дополнительного места. Предыдущий вариант, сортировка и сравнение строк, не привел бы к трате дополнительной памяти, если бы сортировка выполнялась на месте. Но если наш приоритет — это скорость, вариант с использованием хеш-таблиц будет самым эффективным, так как предполагает *однократный* перебор символов каждой строки.

Итак, преобразовав строки в другую структуру данных (хеш-таблицы), мы смогли получить доступ к исходным данным так, что наш алгоритм стал молниеносно быстрым.

Выбор альтернативы для структуры данных не всегда очевиден, поэтому лучше всего представить, как данные будут выглядеть после преобразования во все другие форматы, и проанализировать открывающиеся возможности для оптимизации. Но чаще всего обычно подходят хеш-таблицы, так что начинайте именно с них.

Группировка элементов массива

Вот еще один пример того, как замена структуры данных позволяет оптимизировать код. Допустим, у нас есть массив, содержащий несколько разных значений, и мы хотим упорядочить их так, чтобы одинаковые были сгруппированы вместе. При этом нас не волнует порядок следования самих этих *групп*.

Например, у нас есть следующий массив:

```
["a", "c", "d", "b", "b", "c", "a", "d", "c", "b", "a", "d"]
```

Наша цель — отсортировать буквы, распределив их по группам:

```
["c", "c", "c", "a", "a", "a", "d", "d", "d", "b", "b", "b"]
```

Опять же, нас не волнует порядок следования групп, поэтому устроят и такие результаты:

```
["d", "d", "d", "c", "c", "c", "a", "a", "a", "b", "b", "b"]
["b", "b", "b", "c", "c", "c", "a", "a", "a", "d", "d", "d"]
```

В принципе, выполнить эту задачу можно с помощью любого классического алгоритма сортировки, который вернет следующий массив:

```
["a", "a", "a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
```

Как вы знаете, самые быстрые алгоритмы сортировки выполняются за время $O(N \log N)$. Но можем ли мы достичь большей скорости?

Начнем с определения лучшей мыслимой эффективности. Мы точно знаем, что предельная скорость алгоритмов сортировки — $O(N \log N)$, поэтому представить более быстрый способ сортировки трудно.

Но мы не собираемся сортировать элементы массива в полном смысле этого слова. Поэтому если бы кто-то сказал мне, что нашу задачу можно решить за время $O(N)$, я бы поверил. Конечно, мы не сможем превзойти показатель $O(N)$, так как нам придется обработать каждое значение хотя бы раз. Итак, мы будем стремиться к временной сложности $O(N)$.

Воспользуемся приемом выше и представим наши данные в другом виде.

Начнем с хеш-таблицы. Как бы выглядел наш массив строк в этом случае?

С помощью того же подхода, что и в случае с анаграммами, мы могли бы представить массив так:

```
{"a" => 3, "c" => 3, "d" => 3, "b" => 3}
```

Как и в прошлом примере, здесь мы теряем часть данных. У нас не вышло бы преобразовать эту хеш-таблицу в исходный массив, так как мы не знали бы исходного порядка строк.

Но для наших целей группировки эта потеря данных не важна. По сути, в хеш-таблице есть все данные, которые нужны для создания сгруппированного массива.

Например, мы можем перебрать все пары «ключ — значение» в хеш-таблице и использовать эти данные для заполнения массива нужным количеством экземпляров строк. Код будет выглядеть так:

```
def group_array(array)
  hash_table = {}
  new_array = []

  # Сохраняем количество экземпляров каждой строки в хеш-таблице:
  array.each do |value|
    if hash_table[value]
      hash_table[value] += 1
    else
      hash_table[value] = 1
    end
  end

  # Перебираем хеш-таблицу и заполняем новый массив
  # нужным количеством экземпляров каждой строки:
  hash_table.each do |key, count|
    count.times do
      new_array << key
    end
  end

  return new_array
end
```

Наша функция `group_array` принимает массив и первым делом создает пустую хеш-таблицу и пустой массив.

Сначала мы подсчитываем число экземпляров каждой строки и сохраняем соответствующие значения в хеш-таблице:

```
array.each do |value|
  if hash_table[value]
    hash_table[value] += 1
  else
    hash_table[value] = 1
  end
end
```

В результате получаем хеш-таблицу, которая выглядит так:

```
{"a" => 3, "c" => 3, "d" => 3, "b" => 3}
```

Затем перебираем все пары «ключ — значение» и используем эти данные для заполнения нового массива:

```
hash_table.each do |key, count|
  count.times do
    new_array << key
  end
end
```

То есть при обработке пары `"a" => 3` мы добавляем в новый массив три буквы `"a"`, пары `"c" => 3` — три буквы `"c"` и т. д. К моменту завершения работы функции новый массив будет содержать все строки, объединенные в группы.

Алгоритм выполняется всего за O(N) времени. Это серьезное улучшение по сравнению с временем O($N \log N$), которое ушло бы на сортировку. При этом мы используем дополнительное место O(N) для создания хеш-таблицы и нового массива, хотя для экономии могли бы перезаписать исходный массив. Что касается объема памяти, занимаемого хеш-таблицей, то он не превысит O(N) даже в худшем случае, когда каждая строка в массиве уникальна.

Опять же, если у нас в приоритете скорость, то этот вариант просто фантастический, так как позволяет добиться наибольшей эффективности из возможных.

Выводы

Методы из этой главы могут пригодиться при оптимизации кода. Начинать этот процесс всегда следует с определения текущей и лучшей мыслимой эффективности. После этого можете применять абсолютно любые методы оптимизации.

Вы наверняка обнаружите, что в каких-то ситуациях одни методы работают лучше, чем другие. Но, выбирая подходящий инструмент для решения задачи, не стоит отбрасывать ни один из этих подходов.

С опытом вы научитесь чувствовать, какие методы и приемы применять в той или иной ситуации, а возможно, даже разработаете и собственные!

Заключительные мысли

В ходе этого путешествия вы многому научились.

Теперь вы знаете, что выбор алгоритма и структуры данных сильно влияет на производительность кода.

Вы научились определять текущую эффективность кода.

Узнали о том, как оптимизировать код, сделав его быстрее, проще и эффективнее с точки зрения занимаемой памяти.

Эта книга дала вам все необходимое для принятия обоснованных технологических решений. Создание качественного ПО предполагает оценку преимуществ и недостатков доступных вариантов и поиск компромисса между ними. Теперь и вы способны опираться на плюсы и минусы каждого подхода при выборе оптимального для выполнения поставленной задачи. Вы даже можете обдумывать *новые* варианты, не такие очевидные.

Подходы к оптимизации всегда лучше *тестировать* с помощью инструментов сравнительного анализа. Применение таких техник, которые позволяют оценить

фактическое потребление памяти и скорость работы кода, помогает разобраться в том, стоит ли вносить определенное улучшение. Знания, полученные в ходе знакомства с этой книгой, укажут вам правильное направление, а инструменты сравнительного анализа позволят принять верное решение.

Я надеюсь, при чтении этой книги вы поняли, что темы, которые на первый взгляд кажутся сложными, на самом деле состоят из более простых и понятных концепций. Не стоит пугаться определенного подхода просто потому, что на каком-то из ресурсов его недостаточно понятно объяснили. Помните, что любую концепцию можно разложить на доступные для усвоения фрагменты.

Тема структур данных и алгоритмов очень обширная и глубокая. В этой книге мы лишь поверхностно прошлись по основным аспектам. Но, опираясь на полученную здесь базу, вы можете самостоятельно исследовать и осваивать новые области мира информатики. Искренне надеюсь, что вы продолжите развивать навыки в этом направлении.

Желаю удачи!

Упражнения

Выполните следующие упражнения, чтобы закрепить знания, полученные из этой главы. Решения вы найдете в приложении в разделе «Глава 20».

1. Вы работаете над программой для анализа данных о спортсменах. Ниже представлены два массива с данными об игроках, которые занимаются разными видами спорта:

```
basketball_players = [
  {first_name: "Jill", last_name: "Huang", team: "Gators"},
  {first_name: "Janko", last_name: "Barton", team: "Sharks"},
  {first_name: "Wanda", last_name: "Vakulskas", team: "Sharks"},
  {first_name: "Jill", last_name: "Moloney", team: "Gators"},
  {first_name: "Luuk", last_name: "Watkins", team: "Gators"}
]

football_players = [
  {first_name: "Hanzla", last_name: "Radosti", team: "32ers"},
  {first_name: "Tina", last_name: "Watkins", team: "Barleycorns"},
  {first_name: "Alex", last_name: "Patel", team: "32ers"},
  {first_name: "Jill", last_name: "Huang", team: "Barleycorns"},
  {first_name: "Wanda", last_name: "Vakulskas", team: "Barleycorns"}
]
```

   Если присмотреться, то можно увидеть, что некоторые спортсмены занимаются несколькими видами спорта. Например, Джилл Хуанг (Jill Huang) и Ванда Вакульскас (Wanda Vakulskas) играют и в баскетбол, и в футбол.

Напишите функцию, которая принимает два массива с данными о спортсменах и возвращает массив с именами тех, кто занимается *обоими* видами спорта. У нас это будет:

```
["Jill Huang", "Wanda Vakulskas"]
```

Несмотря на то что у некоторых игроков совпадают имена и фамилии, только у одного человека из списка будет конкретное *полное* имя (имя и фамилия).

Мы можем использовать алгоритм с вложенными циклами, сравнивая каждого игрока из одного массива с каждым из другого, но это потребует $O(N \times M)$ времени. Оптимизируйте его так, чтобы он выполнялся за время $O(N + M)$.

2. Вы пишете функцию, которая принимает массив целых чисел от 0, 1, 2, 3... до N, где нет одного целого числа. Ваша *функция должна возвратить это число.*

   Например, в следующем массиве есть все целые числа от 0 до 6, кроме 4:

   ```
   [2, 3, 0, 6, 1, 5]
   ```

   Поэтому функция должна вернуть значение 4.

   В следующем массиве есть все целые числа от 0 до 9, кроме 1:

   ```
   [8, 2, 3, 9, 4, 7, 5, 0, 6]
   ```

   Здесь функция должна вернуть значение 1.

   Использование подхода с вложенными циклами потребует до $O(N^2)$ времени. Оптимизируйте код так, чтобы он выполнялся за $O(N)$ времени.

3. Вы работаете над очередной программой для прогнозирования цен на акции. Функция, которую вы пишете, принимает массив цен на некую акцию, спрогнозированных на конкретный период времени.

   Например, следующий массив:

   ```
   [10, 7, 5, 8, 11, 2, 6]
   ```

   предсказывает, что в следующие семь дней цены на акцию будут соответствующие (в первый день цена закрытия составит 10 долларов, во второй — 7 и т. д.).

   Ваша функция должна рассчитать наибольшую прибыль, которую можно получить при совершении сделки — одной «покупки», за которой следует одна «продажа».

В прошлом примере больше всего денег можно было бы заработать при покупке акций по 5 долларов и продаже по 11. Мы получили бы прибыль в размере 6 долларов на акцию.

Обратите внимание, что мы могли бы заработать еще больше, совершив несколько сделок, но сейчас эта функция вычисляет максимальную прибыль, которую можно получить от *одной* покупки, за которой следует *одна* продажа.

Мы могли бы использовать вложенные циклы, чтобы найти прибыль от каждой возможной комбинации купли-продажи. Но на такой алгоритм ушло бы $O(N^2)$ времени, что очень медленно для нашей популярной торговой платформы. Ваша задача — оптимизировать код так, чтобы алгоритм выполнялся за время $O(N)$.

4. Вы пишете функцию, которая принимает массив чисел и вычисляет наибольшее произведение любых двух из него. Задача может показаться простой, так как мы можем просто найти два самых больших числа и перемножить их. Но в массиве могут быть и отрицательные числа:

```
[5, -10, -6, 9, 4]
```

Наибольшее произведение здесь — результат перемножения двух *самых меньших* чисел: –10 и –6.

Мы могли бы использовать вложенные циклы для умножения каждой возможной пары чисел, но это заняло бы $O(N^2)$ времени. Оптимизируйте функцию так, чтобы алгоритм выполнялся за время $O(N)$.

5. Вы создаете программу, которая анализирует данные о температуре тела сотен пациентов. Температура измерялась у здоровых людей, поэтому результаты измерений варьируются от 97,0 до 99,0 градусов по Фаренгейту. Важный момент: значения температуры в этом приложении содержат *не больше одного знака после запятой.*

Вот пример массива со значениями температуры:

```
[98.6, 98.0, 97.1, 99.0, 98.9, 97.8, 98.5, 98.2, 98.0, 97.1]
```

Напишите функцию, которая сортирует эти значения в порядке возрастания.

При использовании классического алгоритма сортировки вроде Quicksort этот процесс занял бы время $O(N \log N)$. *Но здесь можно реализовать более быстрый алгоритм сортировки.*

Да, все верно. Несмотря на то что самая быстрая сортировка занимает $O(N \log N)$ времени, здесь все немного иначе. Почему? Потому что в этом

примере *диапазон возможных значений ограничен* и мы можем отсортировать их за время O(N). Количество шагов при этом может быть равно N, умноженному на константу, но временная сложность алгоритма все равно будет O(N).

6. Вы пишете функцию, которая принимает массив неотсортированных целых чисел и возвращает размер *самой длинной возрастающей последовательности* из целых чисел, увеличивающихся на 1. Например, в массиве:

   ```
   [10, 5, 12, 3, 55, 30, 4, 11, 2]
   ```

   самая длинная возрастающая последовательность — 2-3-4-5, потому что каждое из этих целых чисел на единицу больше предыдущего. Хотя в массиве еще есть последовательность 10-11-12, она состоит лишь из трех целых чисел. Наша функция должна вернуть значение 4, так как оно соответствует размеру самой *длинной* возрастающей последовательности из чисел этого массива.

   Вот еще пример:

   ```
   [19, 13, 15, 12, 18, 14, 17, 11]
   ```

   Самая длинная последовательность здесь — 11-12-13-14-15, поэтому функция вернет значение 5.

   Если мы отсортируем массив, то сможем найти самую длинную возрастающую последовательность, пройдя по нему всего раз. Но процесс сортировки займет O($N \log N$) времени. Оптимизируйте функцию так, чтобы она выполнялась за время O(N).

ПРИЛОЖЕНИЕ

Решения к упражнениям

Здесь вы найдете готовые решения заданий из раздела «Упражнения» для каждой главы.

Глава 1

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 45.

1. Разберем каждый из случаев:
   а) чтение из массива всегда выполняется за один шаг;
   б) на поиск несуществующего элемента в массиве размером 100 уйдет 100 шагов: компьютеру нужно проверить каждый элемент, чтобы знать, что искомого точно нет;
   в) вставка значения в начало массива потребует 101 шага: 100 для смещения каждого элемента вправо и один — для вставки нового;
   г) вставка значения в конец массива всегда выполняется за один шаг;
   д) на удаление первого значения массива уходит 100 шагов: сначала компьютер удалит первый элемент, а затем сдвинет оставшиеся 99 на одну ячейку влево;
   е) удаление последнего значения массива всегда выполняется за один шаг.
2. Разберем каждый из случаев:
   а) как и в случае с массивом, чтение из множества на основе массива выполняется за один шаг;

б) как и в случае с массивом, поиск значения, которого нет в множестве на основе массива, потребует выполнения 100 шагов: компьютеру нужно проверить каждый элемент, чтобы знать, что искомого точно нет;

в) перед вставкой значения в множество сначала придется провести в нем поиск, чтобы убедиться, что вставляемого значения нет. На это уйдет 100 шагов. Затем нужно сдвинуть все 100 элементов вправо, чтобы освободить место для нового значения, и поместить новое значение в начало множества. В итоге на все у нас уйдет 201 шаг;

г) для вставки нового значения в начало множества нужно выполнить 101 шаг: 100 для поиска и один для вставки;

д) на удаление первого значения множества уйдет 100 шагов, как и в случае с классическим массивом;

е) удаление последнего значения множества выполняется за один шаг, как и в случае с классическим массивом.

3. Если в массиве N элементов, то на поиск всех экземпляров строки `"apple"` в нем нужно N шагов. Если мы хотим найти только один экземпляр этой строки, то можем остановить поиск сразу после его обнаружения. Но если нас интересуют все экземпляры, придется проверить все элементы массива.

Глава 2

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 60.

1. Линейный поиск здесь занимает четыре шага. Мы проверяем каждый элемент слева направо с начала массива. Поскольку 8 — четвертый элемент, мы обнаружим его за четыре шага.

2. Здесь на двоичный поиск уходит всего один шаг. Мы начинаем с проверки среднего элемента массива, которым и является значение 8!

3. Чтобы выполнить эту задачу, нужно посчитать, сколько раз придется делить 100 000 пополам, чтобы получить 1. Итак, при последовательном делении 100 000 на 2 нужно выполнить 16 операций деления, пока результат не станет равным примерно 1,53.

Значит, в худшем случае для выполнения двоичного поиска в массиве из 100 000 элементов нам потребуется выполнить 16 шагов.

Глава 3

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 71.

1. O(1). Здесь под годом *N* мы можем понимать значение, переданное в функцию. Но оно не влияет на количество выполняемых ею шагов.
2. O(N). Обработка *N* элементов массива в цикле потребует выполнения *N* шагов.
3. O(log *N*). *N* соответствует значению `numberOfGrains`, которое передается в функцию. Цикл выполняется, пока условие `placedGrains < numberOfGrains` остается истинным, но исходное значение `placedGrains` — 1, и при каждой итерации цикла оно *удваивается*. Например, для достижения `numberOfGrains`, равного 256, нам пришлось бы выполнить операцию удвоения значения `placedGrains` девять раз, а это значит, что наш цикл выполнялся бы девять раз при *N*, равном 256. Если бы `numberOfGrains` было равно 512, цикл выполнялся бы 10 раз, а при значении 1024 — 11. При удвоении значения *N* количество итераций цикла увеличивается на единицу, поэтому временная сложность алгоритма равна O(log *N*).
4. O(*N*). *N* — это число строк в массиве, и цикл предусматривает выполнение *N* шагов.
5. O(1). *N* — это размер массива, но алгоритм выполняет фиксированное количество шагов вне зависимости от величины *N*. Он учитывает четность числа *N*, но все равно выполняет одно и то же количество шагов.

Глава 4

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 87.

Представленные здесь алгоритмы написаны на языке Python, но их можно загрузить на JavaScript и Ruby со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Вот заполненная таблица:

***N* элементов**	**O(*N*)**	**O(log *N*)**	**O(N^2)**
100	100	Около 7	10 000
2000	2000	Около 11	4 000 000

2. В массиве 16 элементов, поскольку 16^2 равно 256 (другими словами, квадратный корень из 256 равен 16).
3. Временная сложность этого алгоритма — $O(N^2)$, где N — это размер массива. Внешний цикл обрабатывает все N элементов массива, для каждого из которых запускается внутренний цикл, который тоже обрабатывает все N элементов массива. Получается, общее количество шагов равно N^2.
4. Временная сложность следующей функции — $O(N)$, так как элементы в ней перебираются только один раз:

```
def greatestNumber(array):
  greatestNumberSoFar = array[0]

  for i in array:
    if i > greatestNumberSoFar:
      greatestNumberSoFar = i

  return greatestNumberSoFar
```

Глава 5

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 104.

1. Отбросив константы, сократим выражение до $O(N)$.
2. Отбросив константу, сократим выражение до $O(N^2)$.
3. Временная сложность этого алгоритма — $O(N)$, где N — размер массива. Хотя он использует два разных цикла, которые обрабатывают N элементов, общее число шагов будет $2N$, что после отбрасывания константы дает временную сложность $O(N)$.
4. Временная сложность этого алгоритма — $O(N)$, где N — размер массива. В рамках цикла мы выполняем три шага, а значит, в общей сложности наш алгоритм выполняет $3N$ шагов, что после отбрасывания константы дает временную сложность $O(N)$.
5. Временная сложность этого алгоритма — $O(N^2)$, где N — размер массива. Хотя мы запускаем внутренний цикл только в половине случаев, этот алгоритм выполняет $N^2/2$ шагов. Но, отбросив константу 2, мы можем выразить его временную сложность как $O(N^2)$.

Глава 6

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 121.

Представленные здесь алгоритмы написаны на языке JavaScript, но их можно загрузить на Python и Ruby со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. По правилам O-нотации выражение $2N^2 + 2N + 1$ сокращается до $O(N^2)$. Избавляясь от констант, мы получаем $N^2 + N$, но мы отбрасываем и N, как слагаемое более низкого порядка.

2. Отбросив $\log N$ как слагаемое более низкого порядка, сокращаем выражение до $O(N)$.

3. Важно отметить, что функция завершается, как только мы находим пару, сумма которой равна 10. Выходит, лучший сценарий реализуется, когда сумма первых двух чисел составляет 10, ведь так мы можем завершить выполнение функции еще до запуска циклов. Средний случай соответствует сценарию, где искомые два числа находятся примерно в середине массива. Худший предполагает отсутствие двух чисел, в сумме дающих 10. В этом случае мы должны дождаться окончания работы обоих циклов. Временная сложность при таком сценарии равна $O(N^2)$, где N — размер массива.

4. Временная сложность этого алгоритма — $O(N)$, так как размер массива равен N, а цикл перебирает все N элементов.

   Алгоритм продолжит выполнение цикла, даже если символ `"X"` будет найден до достижения конца массива. Мы можем оптимизировать код, если вернем значение `true` сразу после обнаружения `"X"`:

```
function containsX(string) {

  for(let i = 0; i < string.length; i++) {
    if (string[i] === "X") {
      return true;
    }
  }

  return false;
}
```

Глава 7

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 138.

1. N здесь — это размер массива. Наш цикл предусматривает $N/2$ итераций, так как обрабатывает по два значения в рамках каждой из них. Но временная сложность функции будет $O(N)$, так как мы отбрасываем константу.
2. Здесь определить N сложнее, так как мы имеем дело с двумя разными массивами. Но алгоритм обрабатывает каждое значение только раз, поэтому N может быть общим количеством значений в обоих массивах, что приводит к временной сложности $O(N)$. Еще можно обозначить число элементов одного массива буквой N, а другого — M, выразив временную сложность в виде $O(N + M)$. Но мы просто складываем N и M, поэтому проще использовать N для обозначения общего количества элементов данных в обоих массивах и выразить временную сложность в виде $O(N)$.
3. В худшем случае количество шагов алгоритма равно произведению числа символов в «строке-иголке» и «строке — стоге сена». Строки могут быть разной длины, поэтому временная сложность функции равна $O(N \times M)$.
4. Если N — это размер массива, а функция содержит три вложенных цикла, то временная сложность равна $O(N^3)$. Действительно, средний цикл выполняет $N/2$ шагов, а внутренний — $N/4$, так что общее число шагов равно $N \times (N/2) \times (N/4)$, итого $N^3/8$. Отбросив константу, мы получаем $O(N^3)$.
5. N — это количество резюме в массиве. При каждой итерации цикла мы удаляем половину из них, поэтому временная сложность алгоритма будет $O(\log N)$.

Глава 8

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 160.

Представленные здесь алгоритмы написаны на языке JavaScript, но их можно загрузить на Python и Ruby со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Следующая реализация сначала сохраняет значения из первого массива в хеш-таблице, а затем сверяет с ней каждое значение второго:

```
function getIntersection(array1, array2) {
  let intersection = [];
```

```
  let hashTable = {};

  for(let i = 0; i < array1.length; i++) {
    hashTable[array1[i]] = true;
  }

  for(let j = 0; j < array2.length; j++) {
    if(hashTable[array2[j]]) {
      intersection.push(array2[j]);
    }
  }

  return intersection;
}
```

Временная сложность этого алгоритма равна O(*N*).

2. Следующая реализация проверяет каждую строку в массиве. Если проверяемой строки в хеш-таблице нет, она добавляется в нее. Если строка *есть*, значит, она была добавлена туда ранее, что говорит о наличии дубликата! Временная сложность этого алгоритма — O(*N*):

```
function findDuplicate(array) {
  let hashTable = {};

  for(let i = 0; i < array.length; i++) {
    if(hashTable[array[i]]) {
      return array[i];
    } else {
      hashTable[array[i]] = true;
    }
  }
}
```

3. Эта реализация начинается с создания хеш-таблицы со всеми символами строки. Затем мы перебираем все символы алфавита и проверяем, есть ли в хеш-таблице каждый из них. Если буквы в таблице нет, значит, ее нет и в исходной строке, поэтому возвращаем ее:

```
function findMissingLetter(string) {
    // Сохраняем все обнаруженные в строке буквы в хеш-таблице:
    let hashTable = {};
    for(let i = 0; i < string.length; i++) {
        hashTable[string[i]] = true;
    }

    // Сообщаем о букве, которой нет в исходной строке:
    let alphabet = "abcdefghijklmnopqrstuvwxyz";
    for(let i = 0; i < alphabet.length; i++) {
        if(!hashTable[alphabet[i]]) {
            return alphabet[i];
        }
    }
}
```

4. Реализация начинается с перебора всех символов в строке. Если обрабатываемого символа нет в хеш-таблице, он добавляется в нее в качестве ключа со значением 1, которое указывает на то, что символ был обнаружен в строке один раз. Если символ есть в хеш-таблице, мы просто увеличиваем значение соответствующего ключа на 1. Получается, если значение ключа `"e"` — 3, то символ `"e"` встречается в строке три раза.

   Теперь снова перебираем символы строки и возвращаем первый из тех, которые встречаются в ней только раз. Временная сложность этого алгоритма — O(*N*):

```
function firstNonDuplicate(string) {
  let hashTable = {};

  for(let i = 0; i < string.length; i++) {
    if(hashTable[string[i]]) {
      hashTable[string[i]]++;
    } else {
      hashTable[string[i]] = 1;
    }
  }

  for(let j = 0; j < string.length; j++) {
    if(hashTable[string[j]] == 1) {
      return string[j];
    }
  }
}
```

Глава 9

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 178.

Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Мы хотим быть вежливыми со звонящими и отвечать на вызовы в порядке их получения, поэтому будем использовать очередь, которая обрабатывает данные по принципу FIFO («первым пришел — первым ушел»).

2. Мы сможем прочитать значение 4, которое после выталкивания значений 6 и 5 будет верхним элементом стека.

3. Мы сможем прочитать значение 3, которое после удаления значений 1 и 2 окажется в начале очереди.

4. Воспользуемся преимуществом стека, который позволяет выталкивать элементы в обратном порядке. Итак, сначала поместим каждый символ строки в стек, а затем будем выталкивать их по одному, добавляя в конец новой строки:

```
def reverse(string)
  stack = Stack.new

  string.each_char do |char|
    stack.push(char)
  end

  new_string = ""

  while stack.read
    new_string += stack.pop
  end

  return new_string
end
```

Глава 10

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 190.

Представленные здесь алгоритмы написаны на языке Python, но их можно загрузить на Ruby и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Базовый случай — `if low > high`, то есть нам нужно остановить рекурсию, как только значение `low` превысит `high`. Иначе функция будет выводить числа, превышающие заданное старшее, вплоть до бесконечности.

2. Здесь мы столкнемся с бесконечной рекурсией! Функция `factorial(10)` вызовет `factorial(8)`, которая вызовет `factorial(6)`, которая вызовет `factorial(4)`, которая вызовет `factorial(2)`, которая вызовет `factorial(0)`. При этом мы не достигнем базового случая `if n == 1` и рекурсия продолжится: функция `factorial(0)` вызовет `factorial(-2)` и т. д.

3. Допустим, значение `low` равно `1`, а `high` — `10`. При вызове `sum(1, 10)` функция возвращает `10 + sum(1, 9)` — результат прибавления 10 к сумме чисел от 1 до 9. Функция `sum(1, 9)` вызывает `sum(1, 8)`, которая вызывает `sum(1, 7)` и т. д.

   Нужно, чтобы последним вызовом был `sum(1, 1)`, в результате которого мы хотим вернуть число 1. Это и есть базовый случай:

```
def sum(low, high):
  # Базовый случай:
  if high == low:
    return low

  return high + sum(low, high - 1)
```

4. Здесь мы используем похожий подход, как в примере с файловой системой, описанном в тексте главы:

```
def print_all_items(array):
  for value in array:
    # Если текущий элемент это "список", то есть массив:
    if isinstance(value, list):
      print_all_items(value)
    else:
      print(value)
```

Перебираем все значения внешнего массива. Если какое-то из них само окажется массивом, рекурсивно вызываем функцию для обработки этого подмассива. Если нет, мы считаем, что достигли базового случая, и просто выводим значение.

Глава 11

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 213.

Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Назовем нашу функцию `character_count`. Первым делом допустим, что она уже реализована.

   Теперь определим подзадачу. Если основная задача — массив `["ab", "c", "def", "ghij"]`, то подзадача — тот же массив, где нет одной из исходных строк. Пусть подзадачей будет массив без *первой* строки — `["c", "def", "ghij"]`.

   Теперь посмотрим, что произойдет при применении «уже реализованной» функции к этой подзадаче. При вызове `character_count(["c", "def", "ghij"])` функция вернула бы `8`, так как общее число символов в строках равно восьми.

   Итак, чтобы выполнить исходную задачу, просто добавим длину первой строки (`"ab"`) к результату вызова функции `character_count` для решения подзадачи.

Так выглядит одна из возможных реализаций:

```
def character_count(array)
  # Альтернативный базовый случай:
  # return array[0].length if array.length == 1

  # Базовый случай: когда массив пуст:
  return 0 if array.length == 0

  return array[0].length + character_count(array[1, array.length - 1])
end
```

Обратите внимание, что наш базовый случай — это пустой массив: количество строк, равное нулю. В комментариях указан еще один вполне допустимый базовый случай, при котором в массиве есть только одна строка. Тогда мы возвращаем длину этой единственной строки.

2. Представим, что функция `select_even` уже есть, и определим подзадачу. В случае с массивом `[1, 2, 3, 4, 5]` подзадачей будут все числа, кроме первого. Итак, допустим, функция `select_even([2, 3, 4, 5])` уже работает и возвращает `[2, 4]`.

   Первое число в массиве равно 1, поэтому достаточно вернуть `[2, 4]`. Но если бы первым числом был 0, мы вернули бы массив `[2, 4]` с добавленным в него значением `0`.

   Базовый случай здесь — пустой массив.

   Вот одна из возможных реализаций этой функции:

```
def select_even(array)
  return [] if array.empty?

  if array[0].even?
    return [array[0]] + select_even(array[1, array.length - 1])
  else
    return select_even(array[1, array.length - 1])
  end
end
```

3. Каждое следующее треугольное число — это сумма `n` (номер числа) и предыдущего числа этой последовательности. Если наша функция называется `triangle`, то мы выражаем это в виде `n + triangle(n - 1)`. Базовый случай — `n`, равное 1.

```
def triangle(n)
  return 1 if n == 1
  return n + triangle(n - 1)
end
```

4. Допустим, наша функция `index_of_x` уже реализована. Пусть подзадачей будет исходная строка без первого символа. Например, для входной строки `"hex"` подзадачей будет `"ex"`.

   Результат вызова `index_of_x("ex")` — значение `1`. Для решения исходной задачи нужно прибавить к этому результату 1, так как дополнительный символ `"h"` в начале строки смещает `"x"` на одну ячейку вправо.

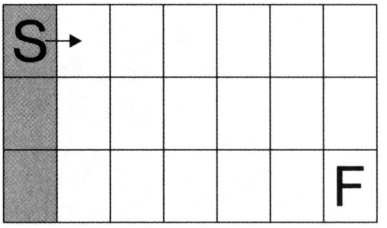
```
def index_of_x(string)
  return 0 if string[0] == 'x'
  return index_of_x(string[1, string.length - 1]) + 1
end
```

5. Эта задача похожа на «Задачу о лестнице». Давайте разберем ее.

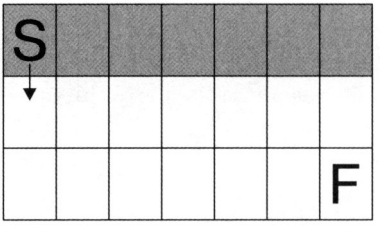
   Из исходного положения у нас есть только два варианта движения: переместиться на одну клетку вправо или вниз.

   Значит, общее число уникальных кратчайших путей равно сумме числа путей из клетки справа от S и числа путей из клетки под S.

   Количество путей из пространства справа от S равно количеству путей в сетке из шести столбцов и трех строк:



   Число путей из пространства под S равно количеству путей в сетке из семи столбцов и двух строк:



   С помощью рекурсии это можно выразить так:

```
return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
```

Остается лишь добавить базовый случай. Это могут быть ситуации, когда в нашей сетке только одна строка или один столбец: в этих случаях нам доступен только один путь.

Вот полная версия кода функции:

```
def unique_paths(rows, columns)
  return 1 if rows == 1 || columns == 1
  return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
end
```

Глава 12

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 229.

Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Проблема здесь в *двух* рекурсивных вызовах функции. Но мы можем легко сократить их до одного:

```
def add_until_100(array)
  return 0 if array.length == 0
  sum_of_remaining_numbers = add_until_100(array[1, array.length - 1])
  if array[0] + sum_of_remaining_numbers > 100
    return sum_of_remaining_numbers
  else
    return array[0] + sum_of_remaining_numbers
  end
end
```

2. Вот версия функции с мемоизацией:

```
def golomb(n, memo={})
  return 1 if n == 1

  if !memo[n]
    memo[n] = 1 + golomb(n - golomb(golomb(n - 1, memo), memo), memo)
  end

  return memo[n]
end
```

3. Чтобы применить мемоизацию здесь, превратим количество строк и столбцов в ключ, который может быть в форме простого массива `[rows, columns]`:

```
def unique_paths(rows, columns, memo={})
  return 1 if rows == 1 || columns == 1
```

```
  if !memo[[rows, columns]]
    memo[[rows, columns]] = unique_paths(rows - 1, columns, memo) +
    unique_paths(rows, columns - 1, memo)
  end
  return memo[[rows, columns]]
end
```

Глава 13

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 257.

Представленные здесь алгоритмы написаны на языке JavaScript, но их можно загрузить на Python и Ruby со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Мы знаем, что после сортировки самые большие числа будут в конце массива, и мы сможем просто перемножить их. Сортировка займет время $O(N \log N)$:

```
function greatestProductOf3(array) {
  array.sort((a, b) => (a < b) ? -1 : 1);

  return array[array.length - 1] * array[array.length - 2] *
    array[array.length - 3];
}
```

   Глядя на этот код, мы понимаем, что в массиве есть как минимум три значения. Вы можете добавить код для обработки случаев, когда это не так.

2. Если мы предварительно отсортируем массив, то, скорее всего, значение каждого числа будет соответствовать его индексу: 0 будет находиться по индексу 0, 1 — по индексу 1 и т. д. Затем мы сможем перебрать массив в поисках числа, значение которого не соответствует его индексу. Если мы его найдем, значит, прямо перед ним должно быть искомое значение:

```
function findMissingNumber(array) {
  array.sort((a, b) => (a < b) ? -1 : 1);

  for(let i = 0; i < array.length; i++) {
    if(array[i] !== i) {
      return i;
    }
  }

  return null;
}
```

Сортировка требует $N \log N$ шагов, а работа цикла — еще N шагов. Но мы можем сократить выражение $(N \log N) + N$ до $O(N \log N)$, отбросив N как слагаемое более низкого порядка.

3. Эта реализация предполагает использование вложенных циклов и выполняется за $O(N^2)$ времени:

```
function max(array) {
  for(let i = 0; i < array.length; i++) {
    iIsGreatestNumber = true;

    for(let j = 0; j < array.length; j++) {
      if(array[j] > array[i]) {
        iIsGreatestNumber = false;
      }
    }

    if(iIsGreatestNumber) {
      return array[i];
    }
  }
}
```

Эта реализация просто сортирует массив и возвращает последнее число. Сортировка занимает время $O(N \log N)$:

```
function max(array) {
  array.sort((a, b) => (a < b) ? -1 : 1);

  return array[array.length - 1];
}
```

Временная сложность этой реализации — $O(N)$, так как она предполагает только один проход по массиву:

```
function max(array) {
  let greatestNumberSoFar = array[0];

  for(let i = 0; i < array.length; i++) {
    if(array[i] > greatestNumberSoFar) {
      greatestNumberSoFar = array[i];
    }
  }

  return greatestNumberSoFar;
}
```

Глава 14

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 280.

Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Один из способов это сделать — использовать простой цикл `while`:

```
def print
  current_node = first_node

  while current_node
    puts current_node.data
    current_node = current_node.next_node
  end
end
```

2. При использовании двусвязного списка у нас есть мгновенный доступ к последним узлам, и благодаря их ссылкам на «предыдущий узел» мы можем получить доступ к предыдущим. Следующий код — это просто обратная версия прошлого фрагмента:

```
def print_in_reverse
  current_node = last_node

  while current_node
    puts current_node.data
    current_node = current_node.previous_node
  end
end
```

3. Здесь мы используем для посещения узлов цикл `while`. Но, прежде чем двигаться дальше, мы проверяем ссылку текущего узла, чтобы установить, есть ли следующий:

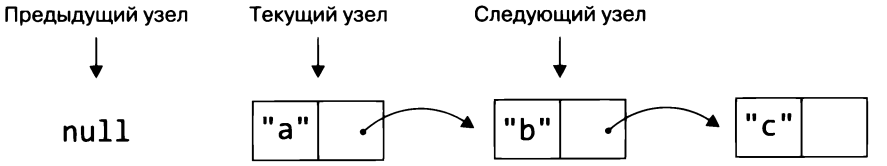
```
def last
  current_node = first_node

  while current_node.next_node
    current_node = current_node.next_node
  end

  return current_node.data
end
```

4. Один из способов обратить вспять классический связанный список — перебрать его элементы, отслеживая три переменные.

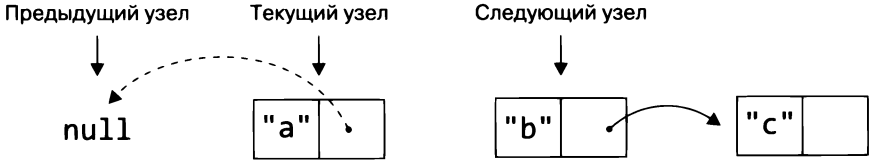
Первая переменная `current_node` соответствует текущему узлу, который обрабатывается в настоящий момент. Мы отслеживаем и узел `next_node`, который следует сразу после текущего, и `previous_node`, который идет перед текущим:



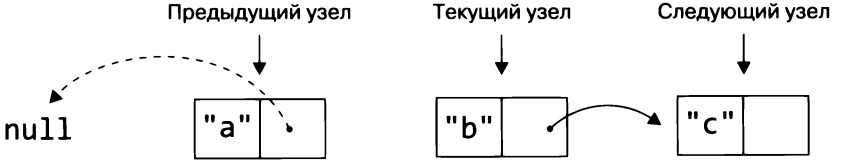
Обратите внимание, что в самом начале, когда текущий узел `current_node` самый первый, значением `previous_node` будет `null`, так как перед первым узлом нет других.

После настройки переменных запускаем цикл.

В рамках него мы сначала меняем ссылку текущего узла (`current_node`) так, чтобы она указывала на предыдущий (`previous_node`):



Теперь сдвигаем все переменные вправо:



Снова запускаем цикл и повторяем процесс изменения ссылки `current_node`, так чтобы она указывала на `previous_node`, вплоть до конца списка. К этому моменту порядок следования его элементов станет обратным исходному.

Реализация этого алгоритма выглядит так:

```
def reverse!
  previous_node = nil
  current_node = first_node
```

```
    while current_node
      next_node = current_node.next_node

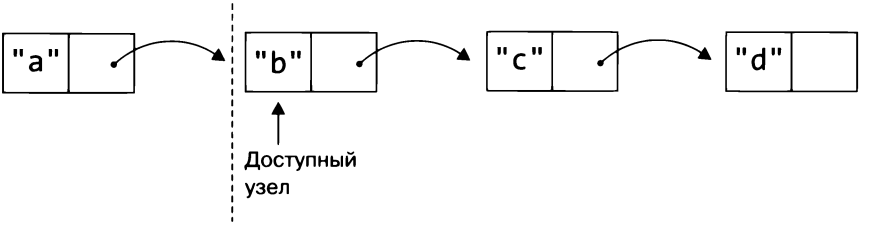
      current_node.next_node = previous_node

      previous_node = current_node
      current_node = next_node
    end

    self.first_node = previous_node
  end
```

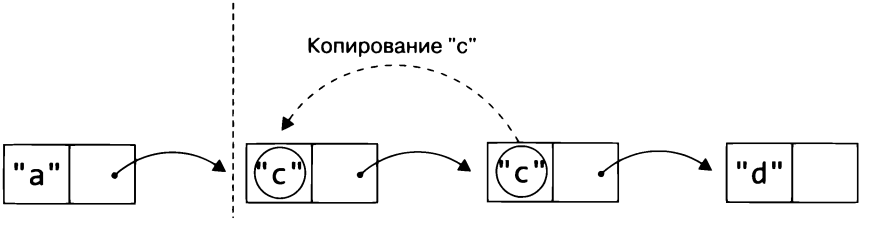
5. Хотите верьте, хотите нет, но мы можем удалить средний узел без доступа к какому-либо из предшествующих ему.

   Ниже изображены четыре узла, из которых нам доступен только `"b"`. Значит, у нас нет доступа к `"a"`, так как в классическом связном списке ссылки указывают только на *следующий* узел. Пунктирная линия на диаграмме указывает на то, что у нас нет доступа ни к одному из узлов слева:

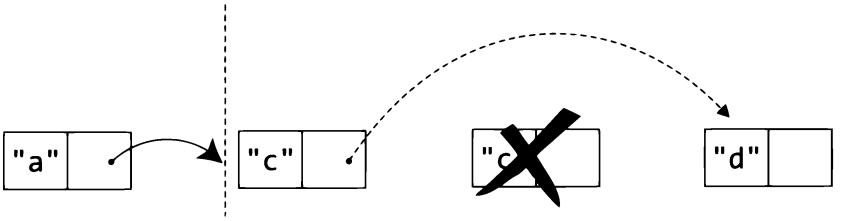


   Теперь посмотрим, как удалить узел `"b"` (даже без доступа к `"a"`). Для ясности будем называть его узлом доступа, так как он первый из доступных для нас.

   Сначала копируем данные из узла, следующего за узлом доступа, в этот же узел, перезаписывая данные в нем. В нашем случае это будет копирование строки `"c"` в узел доступа:



Теперь меняем ссылку узла доступа, чтобы она указывала на узел, который находится через один справа от него, что приводит к удалению исходного узла "c":



Код для выполнения этой задачи довольно лаконичный:

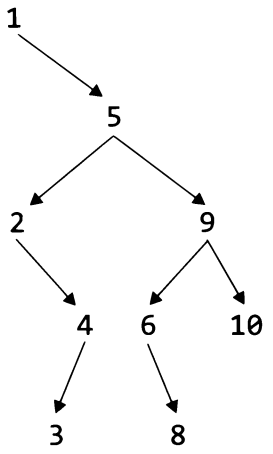
```
def delete_middle_node(node)
  node.data = node.next_node.data
  node.next_node = node.next_node.next_node
end
```

Глава 15

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 310.

Представленные здесь алгоритмы написаны на языке Python, но их можно загрузить на Ruby и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Дерево должно выглядеть так. Обратите внимание, что оно плохо сбалансировано, так как у корневого узла есть только правое поддерево, а левого нет:

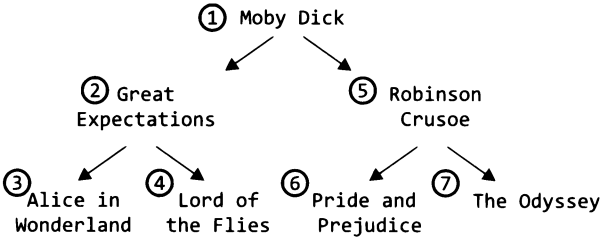


2. Поиск в сбалансированном двоичном дереве требует не более $\log(N)$ шагов. Выходит, что при N, равном 1000, поиск должен осуществляться максимум за 10 шагов.

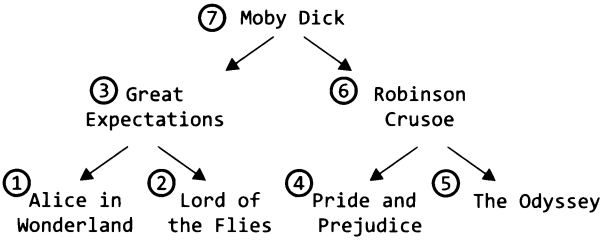
3. Наибольшее значение в двоичном дереве поиска всегда будет соответствовать правому нижнему узлу. Найти его можно, рекурсивно посещая правые дочерние элементы каждого узла вплоть до самого последнего:

```
def max(node):
    if node.rightChild:
        return max(node.rightChild)
    else:
        return node.value
```

4. На этой схеме указан порядок вывода названий книг на экран при использовании прямого обхода:



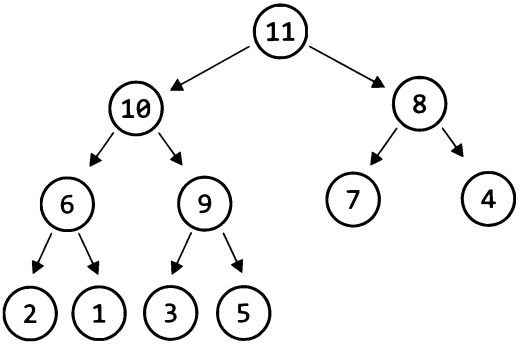
5. Ниже указан порядок вывода названий книг на экран при использовании обратного обхода:



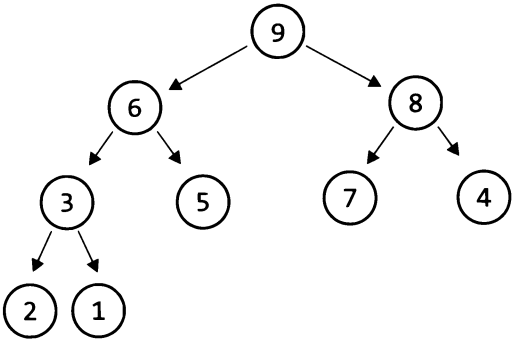
Глава 16

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 337.

1. После вставки значения 11 куча будет выглядеть так:



2. После удаления корневого узла куча будет выглядеть так:



3. Числа будут стоять в порядке убывания (в случае max-кучи, а случае min-кучи — в порядке *возрастания*).

   Понимаете, что это значит? Вы только что открыли еще один алгоритм сортировки!

   Сортировка кучей (*Heapsort*) — это алгоритм, который предполагает вставку всех значений в кучу и дальнейшее извлечение каждого из них. Как вы могли заметить, значения всегда оказываются отсортированными.

   Как и быстрая сортировка (Quicksort), сортировка кучей выполняется за время $O(N \log N)$, так как нам нужно вставить N значений в кучу, а на каждую операцию вставки уходит $\log N$ шагов.

Есть и более сложные версии алгоритма сортировки кучей для улучшения ее эффективности, но в их основе лежит та же идея.

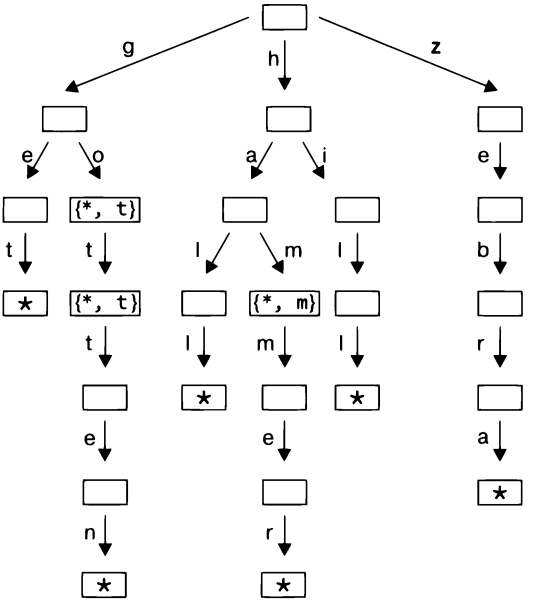
Глава 17

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 363.

Представленные здесь алгоритмы написаны на языке Python, но их можно загрузить на Ruby и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. В этом префиксном дереве хранятся слова: `"tag"`, `"tan"`, `"tank"`, `"tap"`, `"today"`, `"total"`, `"we"`, `"well"` и `"went"`.

2. Префиксное дерево, в котором хранятся слова: `"get"`, `"go"`, `"got"`, `"gotten"`, `"hall"`, `"ham"`, `"hammer"`, `"hill"` и `"zebra"`, выглядит так:



3. Следующая функция начинает с указанного узла префиксного дерева и перебирает все его дочерние элементы. При этом она выводит на экран ключ каждого из дочерних узлов и рекурсивно вызывает саму себя для его обработки:

```
def traverse(self, node=None):
  currentNode = node or self.root
```

```
    for key, childNode in currentNode.children.items():
      print(key)
      if key != "*":
        self.traverse(childNode)
```

4. Наша реализация автозамены сочетает в себе функции поиска (`search`) и сбора слов (`collectAllWords`):

```
def autocorrect(self, word):
  currentNode = self.root
  # Отслеживаем, какая часть слова пользователя уже обнаружена
  # в префиксном дереве. Нам предстоит соединить ее
  # с лучшим из найденных в том же дереве суффиксом
  wordFoundSoFar = ""

  for char in word:
    # Если у текущего узла есть дочерний ключ, соответствующий
    # текущему символу:
    if currentNode.children.get(char):
      wordFoundSoFar += char
      # Переходим к этому дочернему узлу:
      currentNode = currentNode.children.get(char)
    else:
      # Если текущий символ не найден среди дочерних элементов
      # текущего узла, собираем все суффиксы, происходящие
      # от текущего узла, и получаем самый первый из них.
      # Объединяем суффикс с найденным префиксом,
      # чтобы предложить пользователю вариант слова:
      return wordFoundSoFar + self.collectAllWords(currentNode)[0]

  # Если слово пользователя обнаружено в префиксном дереве:
  return word
```

Суть в том, что сначала мы находим в префиксном дереве самый длинный общий префикс со строкой пользователя. Оказавшись в тупике, вместо того чтобы просто вернуть `None` (как это делает функция `search`), мы вызываем функцию `collectAllWords` для обработки текущего узла, чтобы собрать все происходящие от него суффиксы. Затем мы берем первый суффикс массива и объединяем его с префиксом, чтобы предложить пользователю новый вариант слова.

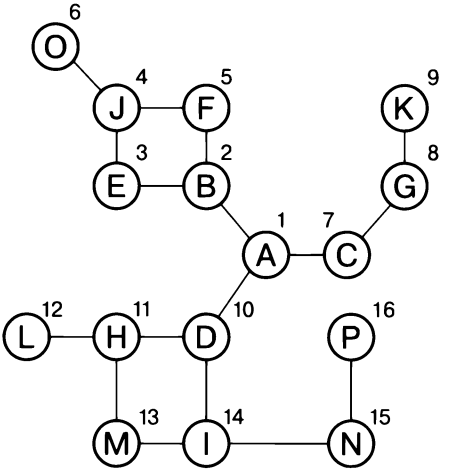
Глава 18

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 419.

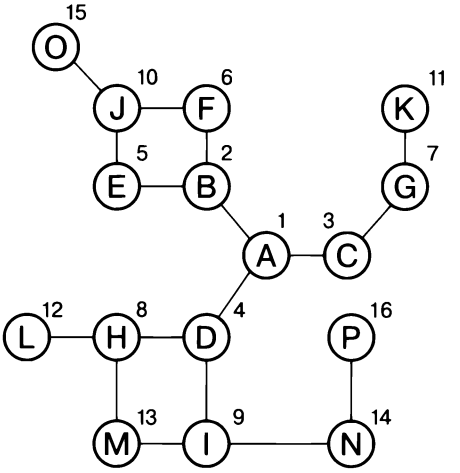
Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Если пользователь интересуется «гвоздями», веб-сайт порекомендует ему «серьги-гвоздики», «иглы», «булавки» и «молоток».

2. При поиске в глубину мы обойдем вершины графа в следующем порядке: A-B-E-J-F-O-C-G-K-D-H-L-M-I-N-P:



3. При поиске в ширину мы обойдем вершины графа в следующем порядке: A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P:



4. Ниже приведена реализация алгоритма поиска в ширину:

```
def bfs(starting_vertex, search_value, visited_vertices={})
  queue = Queue.new
```

```
    visited_vertices[starting_vertex.value] = true
    queue.enqueue(starting_vertex)

    while queue.read
      current_vertex = queue.dequeue

      return current_vertex if current_vertex.value == search_value

      current_vertex.adjacent_vertices.each do |adjacent_vertex|
        if !visited_vertices[adjacent_vertex.value]
          visited_vertices[adjacent_vertex.value] = true
          queue.enqueue(adjacent_vertex)
        end
      end
    end

    return nil
  end
```

Чтобы найти кратчайший путь в невзвешенном графе, мы используем алгоритм поиска в ширину. Его главная особенность в том, что сначала мы обходим все соседние вершины с начальной. Именно это и поможет нам найти кратчайший путь.

Применим этот подход к социальной сети из нашего примера. Так как поиск в ширину начинается с обхода вершин, которые находятся рядом с вершиной Идриса, мы обнаружим сначала кратчайший путь до вершины Лины, и только потом — более длинный путь до нее. На самом деле мы можем закончить поиск, как только обнаружим вершину Лины в первый раз (эта реализация функции не предусматривает остановки процесса поиска, но вы сами можете изменить код).

Итак, при первом посещении каждой вершины мы знаем, что *текущая всегда* будет частью кратчайшего пути от *начальной* вершины до *посещаемой* (помните, что при поиске в ширину текущая и посещаемая вершины не обязательно совпадают).

Например, когда мы впервые посещаем вершину Лины, текущей будет вершина Камиля, потому что при поиске в ширину мы сначала добираемся до Лины через Камиля и только потом через Сашу. Добравшись до Лины (через Камиля), мы можем сохранить в таблице данные о том, что кратчайший путь от Идриса до Лины проходит через вершину Камиля. Эта таблица похожа на `cheapest_previous_stopover_city_table` из пошагового разбора алгоритма Дейкстры.

На самом деле при посещении *любой* вершины кратчайший путь до нее от вершины Идриса будет проходить через текущую. Мы будем хранить все эти данные в таблице `previous_vertex_table`.

В конце мы сможем использовать обратную последовательность шагов от Лины до Идриса, чтобы выстроить кратчайший путь между этими двумя вершинами.

Так выглядит наша реализация:

```
def find_shortest_path(first_vertex, second_vertex, visited_vertices={})
  queue = Queue.new

  # Как и в случае с алгоритмом Дейкстры, отслеживаем в таблице каждую
  # вершину, предшествующую посещаемой
  previous_vertex_table = {}

  # Используем поиск в ширину:
  visited_vertices[first_vertex.value] = true
  queue.enqueue(first_vertex)

  while queue.read
    current_vertex = queue.dequeue
    current_vertex.adjacent_vertices.each do |adjacent_vertex|
      if !visited_vertices[adjacent_vertex.value]
        visited_vertices[adjacent_vertex.value] = true
        queue.enqueue(adjacent_vertex)

        # Сохраняем в таблице previous_vertex_table соседнюю вершину
        # (adjacent_vertex) в качестве ключа, а текущую
        # (current_vertex) - в качестве значения.
        # Это указывает на то, что текущая вершина
        # предшествует соседней
        previous_vertex_table[adjacent_vertex.value] =
          current_vertex.value
      end
    end
  end

  # Как и в случае с алгоритмом Дейкстры, обращаем порядок шагов,
  # опираясь на таблицу previous_vertex_table, чтобы построить
  # кратчайший путь между вершинами
  shortest_path = []
  current_vertex_value = second_vertex.value

  while current_vertex_value != first_vertex.value
    shortest_path << current_vertex_value
    current_vertex_value = previous_vertex_table[current_vertex_value]
  end
  shortest_path << first_vertex.value
  return shortest_path.reverse
end
```

Глава 19

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 431.

Представленные здесь алгоритмы написаны на языке JavaScript, но их можно загрузить на Python и Ruby со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Пространственная сложность этого алгоритма равна O(N^2), потому что он создает массив `collection`, где в итоге будет N^2 строк.

2. Пространственная сложность этой реализации — O(N), так как мы создаем новый массив `newArray`, в котором N элементов.

3. В следующей реализации используется алгоритм, который меняет местами первый и последний элементы. Потом мы меняем второй и предпоследний, затем третий и третий с конца и т. д. Поскольку все модификации происходят на месте и мы не создаем новых данных, пространственная сложность этого алгоритма — O(1).

```
function reverse(array) {
  for (let i = 0; i < array.length / 2; i++) {
    // Новый способ перестановки значений в массиве:
    [array[i], array[(array.length - 1) - i]] =
    [array[(array.length - 1) - i], array[i]];

    // Альтернативный, старый способ перестановки:

    // let temp = array[(array.length - 1) - i];
    // array[(array.length - 1)- i] = array[i];
    // array[i] = temp;
  }

  return array;
}
```

   Хотя для каждой перестановки можно создать временную переменную, у нас никогда не будет больше одного фрагмента данных в любой момент на протяжении всего алгоритма.

4. Вот заполненная таблица:

Версия	Временная сложность	Пространственная сложность
Версия № 1	O(N)	O(N)
Версия № 2	O(N)	O(1)
Версия № 3	O(N)	O(N)

Все три версии выполняют столько шагов, сколько чисел в массиве, поэтому их временная сложность одинаковая — O(N).

Версия № 1 создает новый массив для хранения удвоенных чисел. Его длина та же, что и у исходного, поэтому пространственная сложность этой версии — O(N).

Версия № 2 модифицирует исходный массив на месте, поэтому не использует дополнительную память, и ее пространственная сложность равна O(1).

Версия № 3 тоже модифицирует исходный массив на месте. Но функция рекурсивная, поэтому на пике стек будет содержать N вызовов, и пространственная сложность этой версии — O(N).

Глава 20

Это решения упражнений, которые можно найти в разделе «Упражнения» на с. 471.

Представленные здесь алгоритмы написаны на языке Ruby, но их можно загрузить на Python и JavaScript со страницы, посвященной этой книге (https://pragprog.com/titles/jwdsal2/source_code).

1. Чтобы оптимизировать этот алгоритм, спросите себя: «Если бы я мог каким-то волшебным способом находить нужную информацию за время O(1), то смог бы я ускорить свой алгоритм?»

   Итак, при переборе одного массива мы хотели бы «волшебным способом» находить соответствующее имя спортсмена и в другом за время O(1). Для этого сначала преобразуем один из массивов в хеш-таблицу. Мы будем использовать полное имя (имя *и* фамилию) в качестве ключа и `true` (или что-то другое) в качестве значения.

   После этого перебираем другой массив. Последовательно обрабатывая имя каждого спортсмена, мы выполняем поиск в хеш-таблице за время O(1), чтобы выяснить, занимается ли этот человек другим видом спорта. Если да, добавляем имя спортсмена в массив `multisport_athletes`, который функция возвращает по окончании своей работы.

   Так выглядит соответствующий код:

```
def find_multisport_athletes(array_1, array_2)
  hash_table = {}
  multisport_athletes = []

  array_1.each do |athlete|
```

```
    hash_table[athlete[:first_name] + " " + athlete[:last_name]] = true
  end

  array_2.each do |athlete|
    if hash_table[athlete[:first_name] + " " + athlete[:last_name]]
      multisport_athletes << athlete[:first_name] +
        " " + athlete[:last_name]
    end
  end

  return multisport_athletes
end
```

Временная сложность этого алгоритма равна O(N + M), так как мы перебираем каждый из массивов с именами игроков только один раз.

2. Здесь может пригодиться создание примеров для выявления закономерностей.

   Возьмем массив из шести целых чисел и посмотрим, что будет, если каждый раз удалять из него разные числа:

```
[1, 2, 3, 4, 5, 6] : нет 0: сумма = 21
[0, 2, 3, 4, 5, 6] : нет 1: сумма = 20
[0, 1, 3, 4, 5, 6] : нет 2: сумма = 19
[0, 1, 2, 4, 5, 6] : нет 3: сумма = 18
[0, 1, 2, 3, 5, 6] : нет 4: сумма = 17
[0, 1, 2, 3, 4, 6] : нет 5: сумма = 16
```

   Хм. Когда мы удаляем из массива 0, сумма его элементов равна 21. Когда удаляем 1 — 20, когда удаляем 2 — 19 и т. д. Здесь прослеживается определенная закономерность!

   Прежде чем двигаться дальше, назовем значение 21 полной суммой — суммой элементов массива при отсутствии в нем значения 0.

   Если мы проанализируем примеры выше, то увидим, что сумма любого массива меньше полной, а *разница равна значению отсутствующего числа*. Например, когда в массиве нет числа 4, сумма его элементов равна 17, что на четыре меньше, чем 21, а если не будет 1, сумма элементов массива станет равна 20, что на единицу меньше, чем 21.

   Итак, первым делом вычисляем полную сумму элементов массива. Затем можно вычесть фактическую сумму из полной и получить значение числа, которого в массиве нет.

   Так выглядит соответствующий код:

```
def find_missing_number(array)
  # Вычисляем полную сумму (сумму элементов массива при отсутствии
  # в нем 0):
```

```
  full_sum = 0
  (1..array.length).each do |n|
    full_sum += n
  end

  # Вычисляем ТЕКУЩУЮ сумму:
  current_sum = 0

  array.each do |n|
    current_sum += n
  end
  # Разница между двумя суммами и будет отсутствующим числом:
  return full_sum - current_sum
end
```

Временная сложность этого алгоритма равна O(N). Для вычисления полной суммы он должен выполнить N шагов, а для вычисления фактической — еще N шагов. Итого: $2N$ шагов, что сводится к O(N).

3. Мы можем заставить эту функцию работать намного быстрее с помощью жадного алгоритма (это вполне уместно, учитывая, что наша функция предназначена для увеличения прибыли от торговли акциями).

   Чтобы получить максимальный доход, нам нужно покупать акции как можно дешевле и продавать как можно дороже. Итак, сначала назначаем первую попавшуюся цену ценой покупки (`buy_price`). Затем перебираем все остальные цены и, обнаружив более низкую, обновляем значение `buy_price`.

   В процессе перебора цен проверяем, какую прибыль мы получили бы, если бы продали по конкретной цене. Прибыль вычисляется путем вычитания значения `buy_price` из текущей цены и сохраняется в переменной `greatest_profit`. Ее значение обновляется каждый раз, когда мы обнаруживаем цену, которая дает *бóльшую* прибыль.

   После перебора цен переменная `greatest_profit` будет содержать максимально возможную прибыль, которую мы можем получить в результате одной сделки купли-продажи акций.

   Вот как выглядит код нашего алгоритма:

```
def find_greatest_profit(array)
  buy_price = array[0]
  greatest_profit = 0

  array.each do |price|
    potential_profit = price - buy_price

    if price < buy_price
```

```
      buy_price = price
    elsif potential_profit > greatest_profit
      greatest_profit = potential_profit
    end
  end

  return greatest_profit
end
```

Поскольку мы перебираем N цен только один раз, функция выполняется за время $O(N)$. Итак, мы заработали кучу денег и сделали это быстро.

4. Это еще один алгоритм, ключом к оптимизации которого может стать создание примеров для выявления закономерности.

   Как было отмечено в описании упражнения, наибольшее произведение может быть результатом перемножения отрицательных чисел. Рассмотрим несколько примеров массивов и наибольших произведений, полученных путем перемножения двух чисел из этих массивов:

```
[-5, -4, -3, 0, 3, 4] → Наибольшее произведение: 20 (-5 × -4)
[-9, -2, -1, 2, 3, 7] → Наибольшее произведение: 21 (3 × 7)
[-7, -4, -3, 0, 4, 6] → Наибольшее произведение: 28 (-7 × -4)
[-6, -5, -1, 2, 3, 9] → Наибольшее произведение: 30 (-6 × -5)
[-9, -4, -3, 0, 6, 7] → Наибольшее произведение: 42 (6 × 7)
```

   Здесь видно, что наибольшее произведение может быть получено путем перемножения двух либо наибольших, либо *наименьших* (отрицательных) чисел.

   Зная об этом, реализуем наш алгоритм так, чтобы он отслеживал следующие четыре числа:

   - наибольшее;
   - второе по величине;
   - наименьшее;
   - второе наименьшее.

   Затем сравниваем произведение двух наибольших чисел с произведением двух наименьших. Большее из этих двух значений и будет искомым наибольшим произведением двух чисел массива.

   Как же найти два наибольших и наименьших числа? Можно отсортировать массив. Но сортировка выполняется за время $O(N \log N)$, а мы стремимся к показателю $O(N)$.

   На самом деле мы можем снова пожадничать и найти все четыре числа за *один* проход по массиву.

Вот код этого алгоритма, за которым следует его объяснение:

```
def greatest_product(array)
  greatest_number = -Float::INFINITY
  second_to_greatest_number = -Float::INFINITY

  lowest_number = Float::INFINITY
  second_to_lowest_number = Float::INFINITY
  array.each do |number|
    if number >= greatest_number
      second_to_greatest_number = greatest_number
      greatest_number = number
    elsif number > second_to_greatest_number
      second_to_greatest_number = number
    end

    if number <= lowest_number
      second_to_lowest_number = lowest_number
      lowest_number = number
    elsif
      number < second_to_lowest_number
      second_to_lowest_number = number
    end
  end

  greatest_product_from_two_highest =
    greatest_number * second_to_greatest_number

  greatest_product_from_two_lowest =
    lowest_number * second_to_lowest_number

  if greatest_product_from_two_highest > greatest_product_from_two_lowest
    return greatest_product_from_two_highest
  else
    return greatest_product_from_two_lowest
  end
end
```

Перед запуском цикла приравниваем значения `greatest_number` и `second_to_greatest_number` к *отрицательной* бесконечности — начальные значения переменных будут *ниже* остальных чисел в массиве.

Затем перебираем все числа. Если текущее число превышает значение `greatest_number`, присваиваем этой переменной его значение. Если мы уже нашли второе по величине число, то обновляем значение переменной `second_to_greatest_number`, присваивая ей то, которое было в переменной `greatest_number` до обнаружения текущего числа. Так значение `second_to_greatest_number` действительно будет вторым по величине числом.

Если текущее число меньше значения `greatest_number`, но больше `second_to_greatest_number`, присваиваем текущее значение переменной `second_to_greatest_number`.

Точно так же находим наименьшее (`lowest_number`) и второе наименьшее числа (`second_to_lowest_number`).

После обнаружения этих четырех чисел мы находим произведения двух наибольших и наименьших чисел и возвращаем наибольшее произведение.

5. Чтобы оптимизировать этот алгоритм, нам нужно отсортировать конечное число значений. Например, в этом массиве не может быть более 21 уникального показания температуры:

```
97.0, 97.1, 97.2, 97.3 ... 98.7, 98.8, 98.9, 99.0
```

Вернемся к массиву из описания упражнения:

```
[98.6, 98.0, 97.1, 99.0, 98.9, 97.8, 98.5, 98.2, 98.0, 97.1]
```

Мы можем представить его в виде *хеш-таблицы*, сохранив каждое показание в качестве ключа, а количество вхождений — в качестве значения. Эта таблица будет выглядеть примерно так:

```
{98.6 => 1, 98.0 => 2, 97.1 => 2, 99.0 => 1,
 98.9 => 1, 97.8 => 1, 98.5 => 1, 98.2 => 1}
```

Теперь запускаем цикл, который перебирает значения в диапазоне от 97,0 до 99,0 и *проверяет хеш-таблицу*, чтобы выяснить число вхождений соответствующего значения. Каждый такой поиск выполняется за время O(1).

Теперь используем это число вхождений для заполнения данными нового массива. Наш цикл настроен на перебор значений от 97,0 до 99,0, поэтому значения в итоговом массиве будут стоять в порядке возрастания.

Вот как выглядит код нашего алгоритма:

```
def sort_temperatures(array)
  hash_table = {}

  # Сохраняем в хеш-таблице показания температуры и количество
  # их вхождений:
  array.each do |temperature|
    if hash_table[temperature]
      hash_table[temperature] += 1
    else
      hash_table[temperature] = 1
    end
  end

  sorted_array = []

  # Сначала умножаем значение температуры на 10, чтобы во время
  # работы цикла увеличивать его на целое число, избегая ошибок,
  # связанных с использованием чисел с плавающей запятой:
```

```
  temperature = 970

  # Перебираем в цикле значения от 970 до 990
  while temperature <= 990

    # Если в хеш-таблице есть текущее значение температуры:
    if hash_table[temperature / 10.0]

      # Помещаем в массив sorted_array соответствующее число экземпляров
      # текущего значения температуры:
      hash_table[temperature / 10.0].times do
        sorted_array << temperature / 10.0
      end
    end

    temperature += 1
  end

  return sorted_array
end
```

В этом цикле мы обрабатываем значения температуры, умноженные на 10, чтобы избежать проблем, характерных для арифметических операций над числами с плавающей запятой. Например, увеличение значения переменной на 0,1 в цикле приводит к неоднозначным результатам и некорректной работе кода.

Теперь проанализируем эффективность алгоритма. Мы выполняем N шагов для создания хеш-таблицы и 21 итерацию цикла, перебирая все возможные значения температуры от 97,0 до 99,0.

В рамках каждой итерации этого цикла мы запускаем вложенный цикл для заполнения массива `sorted_array` значениями температуры. Но *максимальное количество итераций этого внутреннего цикла равно* N (число значений температур), так как он выполняется только один раз для каждого значения температуры в исходном массиве.

Выходит, мы выполняем N шагов для создания хеш-таблицы: 21 — во внешнем цикле и N — во внутреннем. Итого: $2N + 21$, что сводится к $O(N)$.

6. Эта оптимизация — самый блестящий способ использования приема волшебных поисков из всех, что я видел.

   Представьте, что мы перебираем массив чисел и обнаруживаем в нем значение 5. Зададим себе вопрос: «Если бы я мог каким-то волшебным способом находить нужную информацию за время $O(1)$, то смог бы я ускорить свой алгоритм?»

   Чтобы определить, будет ли число 5 частью самой длинной восходящей последовательности, нужно выяснить, есть ли в массиве числа 6, 7, 8 и т. д.

Мы можем выполнить каждый из этих поисков за время O(1), если сначала сохраним все числа из массива в хеш-таблице. В результате массив `[10, 5, 12, 3, 55, 30, 4, 11, 2]` выглядел бы так:

```
{10 => true, 5 => true, 12 => true, 3 => true, 55 => true,
 30 => true, 4 => true, 11 => true, 2 => true}
```

Здесь при обнаружении числа 2 мы можем запустить цикл, который проверяет наличие следующего числа в хеш-таблице. Если он его находит, мы увеличиваем длину текущей последовательности на единицу. Этот процесс продолжается, пока цикл не сталкивается с отсутствием в хеш-таблице следующего числа последовательности. Каждый из этих поисков выполняется всего за шаг.

Возможно, вы еще не поняли, как это поможет нам. Допустим, у нас есть массив `[6, 5, 4, 3, 2, 1]`. При обработке числа 6 мы выясняем, что с него восходящая последовательность не начинается. Дойдя до 5, мы обнаруживаем последовательность 5-6. Когда мы достигаем 4, то находим последовательность 4-5-6. Когда нам попадается 3, мы выстраиваем последовательность 3-4-5-6 и т. д. При этом для нахождения всех этих последовательностей мы все равно выполняем примерно $N^2/2$ шагов.

Но мы начнем строить последовательность, только если текущее число — это *наименьший ее элемент*. То есть мы не будем строить последовательность 4-5-6, если в массиве есть 3.

Но как узнать, действительно ли текущее число — наименьший элемент последовательности? С помощью волшебного поиска!

Перед запуском цикла для поиска последовательности мы выполним один шаг, чтобы проверить, есть ли в хеш-таблице число, *которое на 1 меньше текущего*. Итак, если текущее число — 4, сначала проверяем, есть ли в массиве 3. Если есть, мы не начинаем строить последовательность. Чтобы не совершать лишних шагов, начнем выстраивать последовательность только после обнаружения наименьшего ее элемента.

Вот как выглядит соответствующий код:

```
def longest_sequence_length(array)
  hash_table = {}
  greatest_sequence_length = 0

  # Сохраняем числа в хеш-таблице в качестве ключей:
  array.each do |number|
    hash_table[number] = true
  end

  # Перебираем все числа в массиве:
```

```
  array.each do |number|

    # Если текущее число - первый элемент последовательности,
    # (то есть при отсутствии числа, которое на единицу меньше
    # текущего):
    if !hash_table[number - 1]

      # Начинаем отсчет количества элементов последовательности
      # с текущего числа. Так как оно - первое в последовательности,
      # ее длина сейчас равна 1:
      current_sequence_length = 1

      # Подготавливаем текущее число (current_number)
      # к использованию в цикле while:
      current_number = number

      # Запускаем цикл while, который работает, пока не столкнется
      # с отсутствием в хеш-таблице очередного элемента
      # последовательности:
      while hash_table[current_number + 1]

        # Переходим к следующему элементу последовательности:
        current_number += 1

        # Увеличиваем длину последовательности на 1:
        current_sequence_length += 1

        # Жадно отслеживаем наибольшее значение длины последовательности:
        if current_sequence_length > greatest_sequence_length
          greatest_sequence_length = current_sequence_length
        end
      end
    end
  end

  return greatest_sequence_length
end
```

Этот алгоритм выполняет N шагов для построения хеш-таблицы, еще N — для перебора массива и еще примерно N шагов — для поиска элементов последовательности в хеш-таблице. В общей сложности мы выполняем около $3N$ шагов, а значит, временная сложность этого алгоритма равна $O(N)$.

Уважаемый читатель!

Вы прочитали интересную книгу о прикладных структурах данных и алгоритмах. Для закрепления информации самое время применить полученные знания на практике. Это можно сделать вместе с компанией КРОК, специалисты которой приняли решение улучшить качество переводной ИТ-литературы в русскоязычном сообществе и выполнили научное редактирование переведенного текста книги. Если вы представитель бизнеса и потенциальный заказчик ИТ-решений — профессионалы из КРОК смогут помочь вам внедрить решения, о которых вы прочитали в книге. Если вы студент — приходите на практику в КРОК и закрепляйте знания опытом. Если вы опытный специалист — присылайте резюме и добро пожаловать в дружную команду профессионалов.

Тимур Напреев



«Вам нужен полный набор структур данных и дополняющих их распространенных (и не очень) алгоритмов. Узнайте, как, когда и зачем оптимизировать код. Заставьте его работать, сделайте его быстрым, затем сделайте его элегантным — и научитесь использовать компромиссы на этом пути».

Скотт Хансельман, программист Microsoft, профессор, блогер

прикладные структуры данных и алгоритмы

Структуры данных и алгоритмы — это не абстрактные концепции, а турбина, способная превратить ваш софт в болид «Формулы-1». Научитесь использовать нотацию «О большое», выбирайте наиболее подходящие структуры данных, такие как хеш-таблицы, деревья и графы, чтобы повысить эффективность и быстродействие кода, что критически важно для современных мобильных и веб-приложений.

Книга полна реальных прикладных примеров на популярных языках программирования (Python, JavaScript и Ruby), которые помогут освоить структуры данных и алгоритмы и начать применять их в повседневной работе. Вы даже найдете слово, которое может существенно ускорить ваш код. Практикуйте новые навыки, выполняя упражнения и изучая подробные решения, которые приводятся в книге.

Начните использовать эти методы уже сейчас, чтобы сделать свой код более производительным и масштабируемым.

Джей Венгроу — опытный преподаватель, программист и создатель Actualize — популярного учебного курса по программированию. Он стремится сделать разработку программного обеспечения доступной для всех и создает программы обучения, которые позволяют разбить сложные темы на более простые и понятные части.

Выпущено при поддержке:

КРОК

Pragmatic Bookshelf

ПИТЕР

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

PiterBooks
PiterForPeople
ThePiterBooks
Company/piter

ISBN:978-5-4461-2068-0



Основы программирования