O'REILLY®

Алгоритмы

с примерами на Python

Джордж Хайнеман

Алгоритмы

с примерами на Python

Джордж Хайнеман

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.421+004.43
Х15

Хайнеман Джордж

Х15 Алгоритмы. С примерами на Python. — СПб.: Питер, 2023. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1963-9

Когда нужно, чтобы программа работала быстро и занимала поменьше памяти, профессионального программиста выручают знание алгоритмов и практика их применения. Эта книга — как раз про практику. Ее автор Джордж Хайнеман предлагает краткое, но четкое и последовательное описание основных алгоритмов, которые можно эффективно использовать в большинстве языков программирования. О том, какими методами решаются различные вычислительные задачи, стоит знать и разработчикам, и тестировщикам, и интеграторам.

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

ББК 32.973.2-018.1
УДК 004.421+004.43

Права на издание получены по соглашению с O'Reilly. 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.

ISBN 978-1492091066 англ.
Authorized Russian translation of the English edition Learning Algorithms ISBN 9781492091066 © 2021 George T. Heineman.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1963-9
© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

https://t.me/it_boooks/2

Предисловие 10

Введение 12

От издательства 17

Глава 1. Решение задач 18

Глава 2. Анализ алгоритмов 48

Глава 3. Хороший хеш — залог успеха 79

Глава 4. Могучая куча 125

Глава 5. Сортировка без магии 154

Глава 6. Двоичные деревья: бесконечность под рукой 189

Глава 7. Графы: всегда на связи! 232

Глава 8. Подведем итоги 288

Об авторе 300

Иллюстрация на обложке 301

Оглавление

Предисловие 10

Введение 12

Для кого эта книга 12

Исходные тексты 13

Условные обозначения 15

Благодарности 16

От издательства 17

Глава 1. Решение задач 18

Что такое алгоритм 18

Поиск наибольшего значения в произвольной последовательности 23

Подсчет действий 24

Как оценить эффективность алгоритма по схеме 26

Поиск двух наибольших значений в произвольном списке 32

Турнирное дерево 36

Сложность по времени и сложность по памяти 42

Заключение 44

Тренировочные задания 44

Глава 2. Анализ алгоритмов 48

Как оценить сложность с помощью эмпирической модели 49

Умножать быстрее, чем в столбик 52

Классы вычислительной сложности 55

Асимптотический анализ 57

Подсчет всех действий 60

Подсчет всех байтов 61

Одна дверь захлопнулась — другая откроется 63
Двоичный поиск в упорядоченном массиве 64
Немногим сложнее, чем π 65
Двух зайцев одним выстрелом 67
Как все это работает 71
Приближенная кривая или четкие границы? 74
Заключение 75
Тренировочные задания 75

Глава 3. Хороший хеш — залог успеха 79
Соответствие значений ключам 79
Хеш-функции и хеш-суммы 85
Хеш-таблица: хранение данных по ключу 87
Определение коллизий и их разрешение последовательным просмотром 89
Раздельное хранение цепочек в списках 96
Оценка 99
Расширяемые хеш-таблицы 104
Оценка производительности динамических хеш-таблиц 109
Динамические массивы 112
Идеальный хеш 114
Проход таблицы циклом 117
Заключение 120
Тренировочные задания 121

Глава 4. Могучая куча 125
Двоичная куча 133
Добавление пары в кучу 137
Снятие элемента с кучи 140
Хранение двоичной кучи в массиве 143
Как погружаться и всплывать 144
Заключение 149
Тренировочные задания 150

Глава 5. Сортировка без магии 154
Обмен элементов в сортировке 155
Сортировка выбором 156
Структура квадратичных алгоритмов сортировки 159
Оценка производительности сортировки выбором и сортировки вставками 161
Рекурсия: разделяй и властвуй! 163
Сортировка слиянием 170
Быстрая сортировка 175
Пирамидальная сортировка 178
Сравнение быстродействия алгоритмов со сложностью O(N log N) 182
Сортировка Тима 183
Заключение 186
Тренировочные задания 187

Глава 6. Двоичные деревья: бесконечность под рукой 189
Введение 189
Двоичные деревья поиска 194
Поиск значения в двоичном дереве 201
Удаление значения из двоичного дерева 202
Обход двоичного дерева 207
Исследование быстродействия двоичных деревьев поиска 210
Сбалансированные двоичные деревья 212
Производительность сбалансированных деревьев 221
Хранение пар (ключ, значение) в двоичном дереве 222
Двоичное дерево как приоритетная очередь 223
Заключение 227
Тренировочные задания 228

Глава 7. Графы: всегда на связи! 232
В графе удобно хранить полезную информацию 232
Обход лабиринта в глубину 237
Другой способ обхода: в ширину 243
Ориентированные графы 251
Взвешенные графы 260

Алгоритм Дейкстры 262
Полный поиск кратчайших путей 275
Алгоритм Флойда — Уоршелла 278
Заключение 283
Тренировочные задания 284

Глава 8. Подведем итоги 288
Встроенные типы данных Python 291
Реализация стека в Python 294
Реализация очередей в Python 295
Реализация кучи и приоритетной очереди 296
Что изучать дальше? 297

Об авторе 300

Иллюстрация на обложке 301

Предисловие

Алгоритм — основа computer science и сущность нынешней информационной эпохи. Алгоритмы движут поисковыми системами, которые ежедневно отвечают на миллиарды запросов, и позволяют безопасно передавать данные по всему Интернету. Алгоритмы все чаще оказываются нужны пользователям самых различных областей — от целевой рекламы до онлайн-ценников. Новостные медиа полны обсуждений: что же такое алгоритмы и на что они способны.

Развитие науки, технологии, инженерии и математики (STEM[1]) дало мощный толчок к непрерывному становлению и модернизации мировой экономики. Однако специалистов, способных формулировать и применять алгоритмы на благо медицины, промышленности и даже государственного управления, просто не хватает. Необходимо, чтобы людей, которые умеют использовать алгоритмы в задачах из собственной области исследования и практики, стало больше.

Для того чтобы разбираться в алгоритмах, не нужно оканчивать четыре курса университета. Увы, большинство книг и материалов в Сети на эту тему рассчитаны именно на студентов: в них делается упор на математические доказательства и основные постулаты компьютерной науки. Справочники содержат пугающее количество самых разнообразных алгоритмов и бесчисленные их варианты для самых изощренных случаев. Слишком часто читатель сдается уже на первой главе такой книги. Это как читать полный английский словарь с целью подправить правописание: лучше бросить сразу. Куда полезнее взять специализированное пособие: в нем будет сто английских слов, в которых чаще всего ошибаются, а также правила, по которым слова составляются (и исключения из них). Вот так же и людям различных профессий и опыта, когда они хотят применять алгоритмы в своей работе, нужно пособие, нацеленное именно на их нужды.

В этой книге в доступной форме описаны алгоритмы, которые сразу же можно применять для улучшения эффективности программ. Все алгоритмы написаны на Python, одном из самых популярных и интуитивно понятном языке про-

[1] Science, Technology, Engineering and Mathematics — принятый в США подход к популяризации точных и естественных наук. — *Примеч. пер.*

граммирования, которым пользуются в самых различных сферах — в анализе данных, биоинформатике, промышленности.

Помимо аккуратных описаний самих алгоритмов, в помощь читателю в тексте приведено немало иллюстраций, поясняющих основные идеи. Исходные тексты программ распространяются под свободной лицензией и доступны в репозитории проекта.

Эта книга научит вас основным алгоритмам и структурам данных, принятым в computer science, — с ними ваши программы будут эффективнее. Может помочь при приеме на работу и, несомненно, послужит хорошим началом на пути изучения алгоритмов!

Цви Галиль,
почетный декан Технологического института Джорджии,
кафедра компьютерных наук им. Фредерика Дж. Стори,
Атланта, 2021

Введение

Для кого эта книга

Предполагается, что читатель уже имеет практический опыт программирования — например, на Python. Если вы никогда не писали программ, я бы посоветовал сначала выучить какой-нибудь язык программирования, а потом уже браться за чтение. В книге используется Python, потому что он подходит и программистам, и непрограммистам.

Алгоритмы предназначены для решения общих задач, которые постоянно возникают при разработке программного обеспечения. Преподавая на младших курсах, я стараюсь навести мосты между знаниями студентов и понятиями, которые я использую в своих алгоритмах. Большинство учебников содержат тщательные пояснения, но все равно их всегда недостаточно. Частенько студент не может самостоятельно изучить алгоритм, потому что ему не показали, как ориентироваться в материале.

Я опишу цель этой книги одним абзацем и одной иллюстрацией. Начнем мы с нескольких структур данных, которые показывают, как информация представляется примитивными атомарными типами, например 32-разрядными целыми числами или 64-разрядными вещественными. Некоторые алгоритмы, в частности **двоичный поиск**, работают с такими структурами напрямую. Более сложные алгоритмы, например алгоритмы на графах, используют важные абстрактные типы данных (*стек*, *приоритетная очередь* и т. п.), которые мы изучим по ходу изложения. Абстрактный тип подразумевает набор действий над ним, и вот эти действия будут эффективны, только если выбрать подходящую структуру данных для его реализации. Ближе к концу книги мы рассмотрим, как повысить быстродействие различных алгоритмов. Сами алгоритмы мы либо целиком напишем на Python, либо рассмотрим соответствующие сторонние пакеты Python, в которых они эффективно реализованы.

Если вы заглянете в сопутствующие книге исходные тексты программ, то найдете там в каждой главе файл `book.py` — это программа на Python, которая при запуске воспроизводит на вашем компьютере все схемы из книги (рис. В.1).

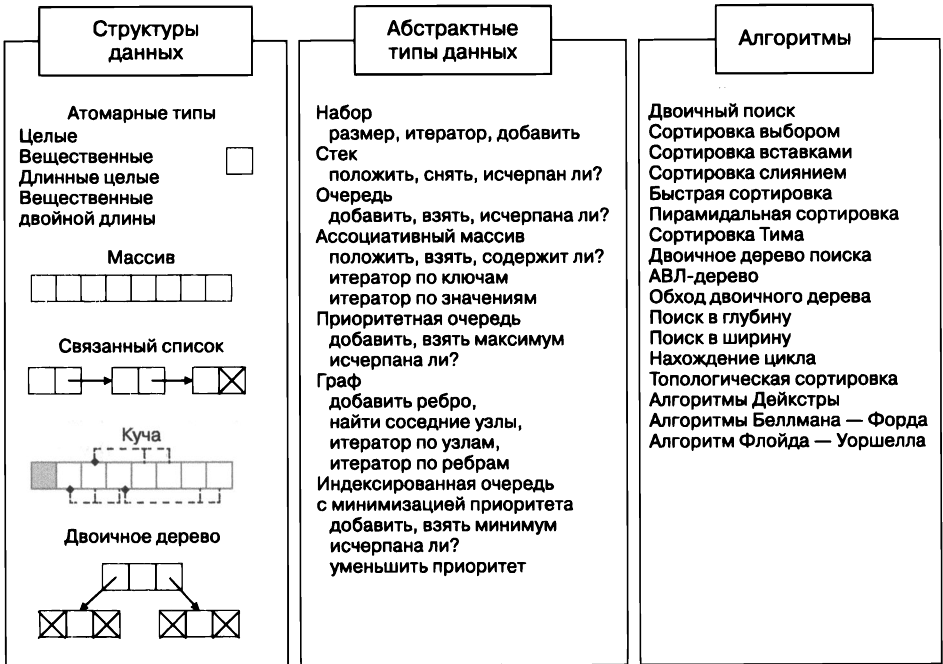


Рис. В.1. Сводка по тематическому содержанию книги

В конце каждой главы есть проверочные упражнения, на них вы можете потренироваться применять свежие знания. Очень советую сначала попробовать решить задачи самостоятельно, а потом сравнить с примерами решений в репозитории к книге.

Исходные тексты

Все программы из книги доступны в соответствующем репозитории GitHub: http://github.com/heineman/LearningAlgorithms. Они совместимы с Python 3.4 и выше. Везде, где это было разумно, я старался следовать рекомендациям Python и оформлять системные методы наподобие `__str__()` или `__len__()`. Примеры в книге отформатированы с отступом два пробела — так они лучше помещаются в ширину при печати; в программах репозитория используется стандартный отступ четыре пробела. Изредка в печатных примерах попадаются однострочники, например `if j == lo: break`.

В программах используются три свободно распространяемые библиотеки Python, которые необходимо скачать и установить самостоятельно[1]:

- NumPy (https://www.numpy.org) версии 1.19.5;
- SciPy (https://www.scipy.org) версии 1.6.0;
- NetworkX (https://networkx.org) версии 2.5.

NumPy и SciPy – одни из самых популярных свободных библиотек с огромным сообществом. Я их использую, чтобы измерить фактическую производительность алгоритмов. NetworkX – большой сборник эффективных алгоритмов для работы с графами, он нам понадобится в главе 7; там же есть удобная готовая структура данных, реализующая граф. Средства этих библиотек позволят не изобретать очередное колесо, если в этом нет нужды. Если они не установлены, не беда: примеры написаны так, что смогут работать и без них, нужные функции также есть в репозитории.

Время работы примеров в книге замеряется с помощью модуля `timeit`, который многократно запускает указанный фрагмент программы. Часто для большей аккуратности само измерение делается тоже несколько раз. Из всех замеров выбирается *быстрейший*, а не средний. Обычно считается, что так лучше измерять производительность: если запустить пример несколько раз, а потом взять среднее время работы, на это среднее повлияет худшее время выполнения, которое может быть большим из-за того, что именно тогда операционная система запускала какие-то еще задачи[2].

Если скорость работы алгоритма сильно зависит от свойств обрабатываемых данных (как в примере **сортировки вставками** из главы 5), о том, что в этом случае измеряется среднее время, будет сказано явно.

В репозитории больше 10 000 строк кода на Python, сценарии для запуска всех тестов и вычисления всех данных для таблиц в книге; можно воспроизвести также большую часть схем и графиков. Исходный текст снабжен, как это принято в Python, строками документации и тестами, причем тестовое покрытие, согласно https://coverage.readthedocs.io, составляет 95 %.

---

[1] В репозитории есть программы, использующие также Matplotlib (https://matplotlib.org) и некоторые другие библиотеки. – *Примеч. пер.*

[2] Строго говоря, timeit высчитывает *потребленное* время процесса, в которое не входят интервалы, когда работал не он, а другие задачи. Но активность других задач может повлиять на актуальность кэшей процессора и памяти: если постоянно выполнять один и тот же код в памяти на одних и тех же данных, кэши «разогреваются» и производительность растет, а если контекст задач меняется часто – теряют актуальность. Вопрос о том, какой способ адекватнее покажет производительность *алгоритма*, открыт. – *Примеч. пер.*

Если у вас возникнут технические вопросы или затруднения в работе примеров, свяжитесь с нами по адресу bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Условные обозначения

В книге используются следующие типографические обозначения:

Курсив

Выделяет новые термины, а также все, что автор хотел подчеркнуть.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, клавиш и их сочетаний, элементов интерфейса.

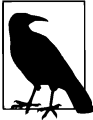
`Моноширинный шрифт`

Используется для программ в иллюстрациях и для ссылки на фрагменты программ в тексте — например, для переменных, имен функций, типов данных, операторов и ключевых слов.

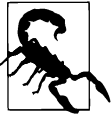
Полужирный шрифт

В русском переводе обозначает собственные имена алгоритмов и структур данных.

Кошачий лемур сопровождает совет или подсказку. Из всех приматов, включая человека, у кошачьего лемура самое большое суммарное поле зрения — до 280°. В тексте советы позволяют буквально расширить кругозор: в них изложена познавательная информация или описана интересная возможность Python.



Ворона сопровождает теоретические замечания. Большинство зоопсихологов сходятся на том, что вороны — очень умные птицы: они умеют решать сложные задачи и даже пользоваться предметами. В тексте примечание содержит определение термина или разъяснения важного понятия. Прежде чем читать дальше, их надо внимательно изучить.



Скорпион сопровождает предупреждение. Встретив скорпиона в природе, мы осторожно останавливаемся. В книге скорпион предупреждает о том, что надо остановиться и тщательно рассмотреть сложные аспекты, с которыми можно столкнуться при реализации алгоритмов.

Благодарности

Я считаю, что лучшее в computer science — это изучение алгоритмов. Спасибо читателям за то, что дали мне возможность поделиться своими наработками. Спасибо моей жене Дженнифер за то, что поддержала меня в затее с очередной книгой. Спасибо моим сыновьям, Николасу и Александру, за то, что они уже выросли и могут изучать программирование.

Эта книга стала лучше благодаря редакторам O'Reilly — Мелиссе Даффилд, Саре Грей, Бет Келли и Вирджинии Вильсон. Они помогали мне систематизировать общее содержание книги и выстраивать пояснения к нему. Технические рецензенты — Лаура Хелливелл, Чарли Лаверинг, Хелен Скотт, Стенли Силков и Аура Веларде — нашли множество неточностей, улучшили реализацию алгоритмов и дали пояснения к ним. Если остались какие-то недочеты — все они целиком на моей совести.

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

Решение задач

https://t.me/it_boooks/2

В этой главе

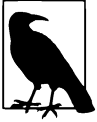
- Несколько алгоритмов решения одной и той же вводной задачи.
- Как исследовать производительность алгоритма на входных наборах данных размером *N*.
- Как посчитать количество основных действий, выполненных при обработке конкретного набора данных.
- Как определить уровень падения производительности при удвоении входных данных.
- Как предсказать *сложность алгоритма по времени* на основании подсчета действий, которые он выполнит на входных данных размером *N*.
- Как предсказать *сложность алгоритма по памяти* на основании оценки дополнительной памяти, которая потребуется ему для обработки входных данных размером *N*.

Что ж, начнем!

Что такое алгоритм

Объяснять алгоритм – это как рассказывать сказку. Каждый алгоритм – новый волшебный предмет или неожиданный поступок, которые помогают быстро и легко справиться с казавшейся трудной задачей. В этой главе мы рассмотрим несколько решений нетрудной задачи, а потом разберемся, какой алгоритм

эффективнее и почему. Кроме того, поймем, как изучать быстродействие алгоритма *независимо от его реализации* — хотя непосредственные данные о работе конкретных реализаций у нас тоже будут.



Алгоритм — это пошаговая инструкция решения некоторой задачи, реализованная в виде программы. Программа должна возвращать правильный ответ за предсказуемое время. Изучая алгоритм, мы проверяем и правильность ответа (в том числе на всех допустимых входных данных), и его вычислительную сложность (возможно, есть более эффективные решения задачи).

Давайте посмотрим, как процесс решения и анализа задачи проходит в жизни. Скажем, нам нужно найти наибольшее значение в несортированном списке. На рис. 1.1, *слева*, приведено три *набора данных* для нашей задачи[1] — каждый набор в виде списка Python. Данные обрабатываются алгоритмом (изображен в виде цилиндра), который должен выдавать правильный ответ; ответы перечислены в правой части. Как реализован алгоритм решения? Как он себя ведет на различных входных наборах? Можно ли предсказать время работы? Как быстро можно найти наибольшее из миллиона чисел?

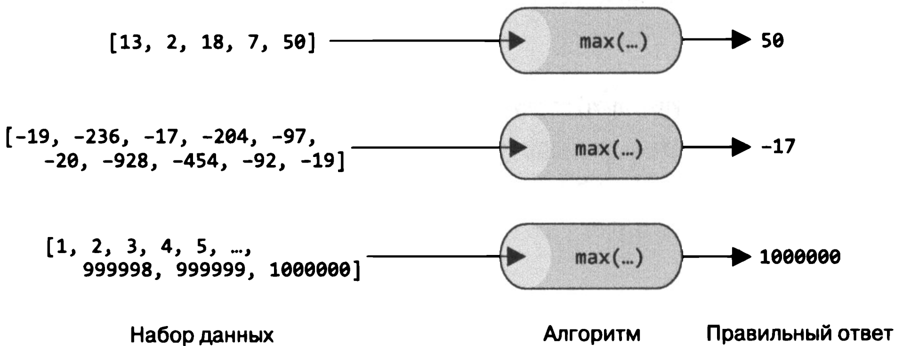


Рис. 1.1. Обработка алгоритмом различных наборов входных данных

[1] Автор использует термин problem instance — буквально: экземпляр задачи. Однако в тексте под этим всегда подразумеваются только входные данные, и никогда — конкретная задача целиком, то есть данные, условия применимости и требуемый результат. Например, частая формулировка problem instance of size N — это очевидно входные данные размером N. — *Примеч. пер.*

Алгоритм и реализующая его программа должны не только давать правильный ответ, но и завершаться за предсказуемое время. Наша задача уже решена — для нее в Python есть функция `max()`. Правда, аккуратно исследовать эффективность алгоритма, оперируя произвольными входными данными, нельзя. Имеет смысл тщательно подготовить подходящие наборы данных.

Таблица 1.1 показывает время работы функции `max()` на двух типах входных данных различных размеров: в одних наборах целые числа в списке идут по возрастанию, в других — по убыванию. На разных компьютерах результаты окажутся разными, но всегда можно проверить неизменность двух вещей.

- Время вычисления `max()` на *достаточно длинных* возрастающих последовательностях всегда больше времени вычисления на таких же, но убывающих.
- Если длина последовательности увеличивается вдесятеро, время вычисления `max()` на ней тоже увеличивается плюс-минус вдесятеро, как если бы мы каждую проверку делали вручную.

В нашей задаче вычисляется наибольшее значение, а исходная последовательность не меняется. В некоторых других случаях, например в алгоритмах сортировки элементов списка из главы 5, требуется не вычислить новое значение, а изменить сами введенные данные. В нашей книге *N* обозначает размер набора входных данных.

Таблица 1.1. Запуск функции max() на двух типах наборов входных данных размером N[1]

N	Восходящая последовательность	Нисходящая последовательность
100	0.001	0.001
1000	0.013	0.013
10 000	0.135	0.125
100 000	1.367	1.276
1 000 000	14.278	13.419

[1] В оригинале используются американские договоренности о записи целых и вещественных чисел: тысячи в целых числах разделяются запятыми, а дробная часть вещественного отделяется от целой точкой. В российском стандарте по-другому, но, поскольку в Python разделителем в вещественном числе также является точка, этот формат мы сохранили. — *Примеч. пер.*

Замечания о времени работы

- Невозможно *с уверенностью* предсказать время работы алгоритма на наборе, допустим, из 100 000 элементов (обозначим это время T(100 000)). Для разных языков программирования и на разных компьютерах оно будет разным.
- Однако, зная T(10 000), предсказать T(100 000) — время работы на десятикратно больших данных можно, хотя погрешность такого предсказания неизбежна.

Когда придумываешь алгоритм, важнее всего убедиться, что он работает правильно *на всех допустимых наборах входных данных*. Как сравнивать свойства двух алгоритмов, если они решают одну и ту же задачу? Этим мы по большей части займемся в главе 2. Изучение алгоритма тесно связано с интересными задачами из настоящей жизни, в которых он требуется. Математическая сторона алгоритма может быть непростой, но мы постараемся каждое отвлеченное понятие связать с таким вопросом жизненной практики.

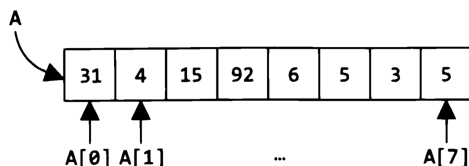
Принято думать, что оценить эффективность алгоритма — значит подсчитать, сколько ему потребовалось *вычислительных операций*. Но как раз это совсем не просто! Центральный процессор компьютера (CPU) исполняет *машинные инструкции* — арифметические операции (наподобие сложения и умножения), пересылку данных из памяти в регистры процессора, сравнения и т. п. Современные языки программирования бывают как компилируемые (C или C++), в которых текст программы перед запуском транслируется в машинные инструкции, так и интерпретируемые (Python или Java). Программа на интерпретируемом языке транслируется в промежуточное представление, называемое *байт-кодом*. Затем программа-интерпретатор, Python например (в свою очередь сам написанный на C и откомпилированный), разбирает и выполняет этот байт-код[1]. При этом некоторые функции, например `min()` и `max()`, встроены в Python — они написаны на C, скомпилированы в машинные инструкции и так выполняются.

---

[1] Главное отличие машинного кода от байт-кода в том, что машинный код содержит инструкции *процессора*, которые работают с тем, что на самом деле есть в процессоре, — атомарными ячейками памяти, их адресами и регистрами, а байт-код — инструкции программы-*интерпретатора*, которые, по сути, отличаются от языка программирования только представлением — в них используются имена, объектная модель, составные типы данных и все остальное. — *Примеч. пер.*

МАССИВ ВСЕМОГУЩИЙ

Массив — фиксированный набор однотипных значений, занимающий один непрерывный фрагмент оперативной памяти. Это одна из самых старых и самых надежных структур данных. Если значений больше одного — программисты хранят их в массиве. На схеме изображен целочисленный массив из восьми элементов:



В массиве A восемь ячеек. Каждая ячейка доступна по номеру, например, `A[0]` — это 31, а `A[7]` — 5. В массивах можно хранить и строки, и вообще любые объекты сколь угодно сложного типа.

Что следует знать о массивах.

- Начальный элемент массива A длиной *N* имеет индекс 0 и обозначается `A[0]`, конечный обозначается `A[N-1]` и имеет индекс *N* – 1.
- Зная номер `i` элемента в массиве, можно прочитать оттуда элемент `Ae[i]` или записать на место `A[i]` новое значение. При этом `i` — это индекс в диапазоне от 0 до *N* – 1.
- Длина массива всегда известна. В Python и Java ее можно задать в процессе работы программы, а в C — только заранее.
- Чтобы изменить длину массива, необходимо выделить память под новый массив нужной длины и скопировать туда все данные из старого. Просто так размер массива уменьшить или увеличить нельзя.

Несмотря на простоту, массивы — очень гибкий и удобный инструмент организации данных. В Python имеется тип данных `list`, с которым можно смело работать как с массивом, хотя его возможности куда шире: в `list` можно хранить объекты разных типов одновременно, а размер его может меняться в процессе работы программы.

Посчитать, сколько машинных инструкций выполнилось при работе алгоритма, практически невозможно – современные компьютеры выполняют их *миллиардами* за секунду! Давайте вместо этого считать, сколько *основных действий* выполнил алгоритм. Вопрос «Сколько действий?» может означать «Сколько раз сравнивались два элемента массива?» или, к примеру, «Сколько раз вызывалась

некоторая функция?». В случае max() нас будет интересовать, сколько раз вызывалась операция сравнения *«меньше»* (>). Подробнее о правилах подсчета действий мы поговорим в главе 2.

Ну что же, настала пора сорвать покров тайны с алгоритма функции max() и понять, что именно определяет ее поведение.

Поиск наибольшего значения в произвольной последовательности

Рассмотрим проблемную реализацию поиска наибольшего значения произвольной последовательности в примере 1.1. Каждый элемент A сравнивается с my_max, и, если найдено значение больше, my_max обновляется.

Пример 1.1. Проблемная реализация поиска наибольшего значения последовательности

```
def flawed(A):
  my_max = 0        ❶
  for v in A:       ❷
    if my_max < v:
      my_max = v    ❸
  return my_max
```

❶ Переменная my_max хранит текущее наибольшее значение. Здесь она инициализируется нулем.

❷ В цикле for определена переменная v, которая на каждом проходе цикла равна очередному элементу A. Оператор if внутри цикла выполняется для каждого такого v.

❸ Переменная my_max обновляется, если v оказалось больше.

Главное в нашем решении — операция сравнения двух выражений (<), которая определяет, меньше ли первое выражение второго. На рис. 1.2 показано, что по мере прохождения переменной v всех значений из A переменная my_max меняется трижды. Функция flawed() определяет наибольшее значение A из шести элементов, вызывая оператор < не более шести раз, по одному разу на каждый элемент. Если в наборе данных будет N элементов, flawed() вызовет < не более N раз.

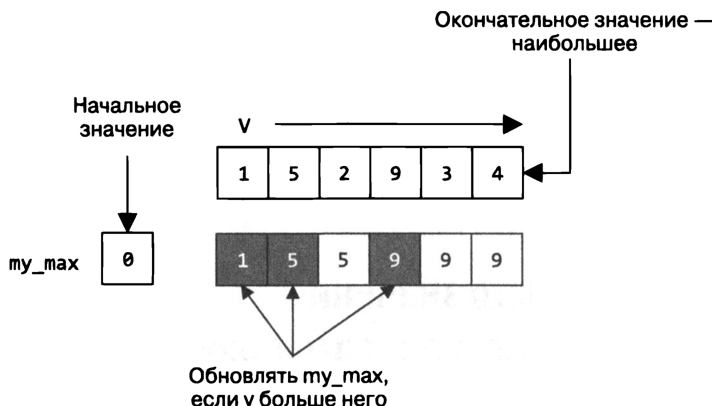
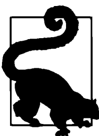


Рис. 1.2. Как работает flawed()

Эта реализация алгоритма содержит ошибку: предполагается, что в `A` есть хотя бы одно неотрицательное число. Вызов `flawed([-5,-3,-11])` вернет `0`, что неправильно. Часто вместо нуля пытаются использовать «наименьшее возможное число», примерно так: `my_max = float('-inf')`. Этот подход также небезупречен, потому что для пустого списка `A = []` он вернет `-inf`, которого там не было. Недочет надо исправить.



Python-выражение range(x,y) вычисляет последовательность целых чисел от x до y (не включая y). Если x больше, чем y, можно задать убывающую последовательность от x до y, не включая y, с помощью range(x,y,–1). Если сделать список из range(1,7), list(range(1,7)) даст [1,2,3,4,5,6]. Соответственно, list(range(5,0,–1)) даст [5,4,3,2,1], а если дополнительно задать шаг — приращение последовательности — list(range(1,10,2)) даст [1,3,5,7,9]: разность между соседними элементами будет равна 2.

Подсчет действий

Первоначальное значение `my_max` лучше выбрать из элементов `A` — тогда можно быть уверенными, что вычисленное наибольшее значение будет тоже из `A`. Работающая на всех входных наборах, кроме пустых, а значит, правильная функция `largest()` приведена в примере 1.2. Здесь мы выбираем в качестве начального значения `my_max` начальный элемент `A`, а затем сравниваем его со всеми остальными: вдруг там найдется побольше?

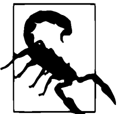
Пример 1.2. Правильная функция, которая находит наибольшее значение в списке

```
def largest(A):
  my_max = A[0]                  ❶
  for idx in range(1, len(A)):   ❷
    if my_max < A[idx]:
      my_max = A[idx]            ❸
  return my_max
```

❶ Сделаем `my_max` равным начальному элементу списка (он доступен по индексу `0`).

❷ Переменная `idx` принимает целочисленные значения от `1` до `len(A)-1` включительно, не достигая `len(A)`.

❸ Если в `A` по индексу `idx` стоит большее значение, обновить `my_max`.



Если передать largest() пустой список — largest([]), в первой же строчке возникнет исключение IndexError, потому что никакого элемента A[0] в списке нет. Встроенная функция max([]) для пустого списка порождает исключение ValueError с пояснением о том, что пустые последовательности не входят в область определения max(). Так программисту проще понять, в чем его ошибка.

В исправленном алгоритме можно уже начинать считать количество действий. Сколько раз вызывалась в нем операция сравнения <? Правильно, $N - 1$ раз. Значит, мы не только избежали ошибки, но и улучшили производительность алгоритма (по правде говоря, совсем чуть-чуть).

Почему важно считать именно операции сравнения? Это действие, описанное в алгоритме, — сравнить два значения. Другие операторы в программе (например, `for` или `while`) выбираются в соответствии с возможностями используемого языка программирования и вполне могут отличаться. Подробнее об этом мы поговорим в следующей главе, а пока продолжим считать сравнения[1].

[1] Если измерить производительность обеих функций методом, который автор предлагает в предисловии (с помощью timeit), окажется, что flawed() работает в полтора раза быстрее largest()! Дело в том, что исправленная функция задействует внутри цикла существенно больше операций, чем ошибочная: в ней используется индексирование (доступ к элементу массива по номеру), а в ошибочной этого не требуется. Таким образом, оценка основных действий алгоритма не всегда напрямую соотносится с затраченным временем и количеством выполненных операций для конкретной реализации этого алгоритма. В частности, индексирования в largest() можно было избежать и выиграть в производительности, но это потребовало бы большего знания о специфике работы цикла for в Python, и читать программу стало бы труднее. — *Примеч. пер.*

Как оценить эффективность алгоритма по схеме

Допустим, нам предлагают совсем иной алгоритм решения нашей задачи. На каком из них остановиться? Рассмотрим функцию `alternate()` из примера 1.3. В ней каждое значение из `A` сравнивается со всеми остальными, и, если выяснится, что оно не меньше, это и есть ответ. Выдаст ли этот алгоритм правильный ответ? И сколько раз в нем выполняется сравнение на входных данных размером N?

Пример 1.3. Другой способ найти наибольшее значение в списке A

```
def alternate(A):
    for v in A:           ❶
        for x in A:
            if v < x:     ❷
                break
        else:
            return v      ❸
    return None           ❹
```

❶ Для каждого `v` из `A` рассмотрим все `x` из `A` и сравним их.

❷ Если `v` меньше какого-то `x`, можно больше не сравнивать: это не максимум.

❸ Если мы просмотрели все `x`, так ни разу и не выполнив `break`, значит, `v` — это максимум и его можно уже возвращать.

❹ До этого места выполнение дойдет только при пустом `A`. В таком случае вернем специальный объект Python — `None`.

Функция `alternate()` пытается найти такое значение `v` из `A`, чтобы никакое другое значение `x` из `A` не оказалось больше него. На этот раз сложно предсказать, сколько потребуется операций сравнения, потому что внутренний цикл по `x` сразу останавливается, как только выясняется, что `x` больше `v`, а внешний — как только `v` оказывается максимумом. Работа `alternate()` показана на рис. 1.3.

В данном случае было выполнено 14 сравнений. Впрочем, очевидно, что общее количество действий зависит от того, какие конкретно значения находятся в списке. Что, если бы они шли в другом порядке? Например, так, чтобы для

ответа потребовалось как можно меньше действий? Такой набор данных называется *лучшим случаем* для `alternative()`. Например, если наибольший из N элементов последовательности стоит в ее начале, количество сравнений в точности равно N. Итак:

- *лучший случай* — набор данных размером N, на котором алгоритм совершает наименьшее количество действий;
- *худший случай* — набор данных размером N, требующий наибольшего количества действий.

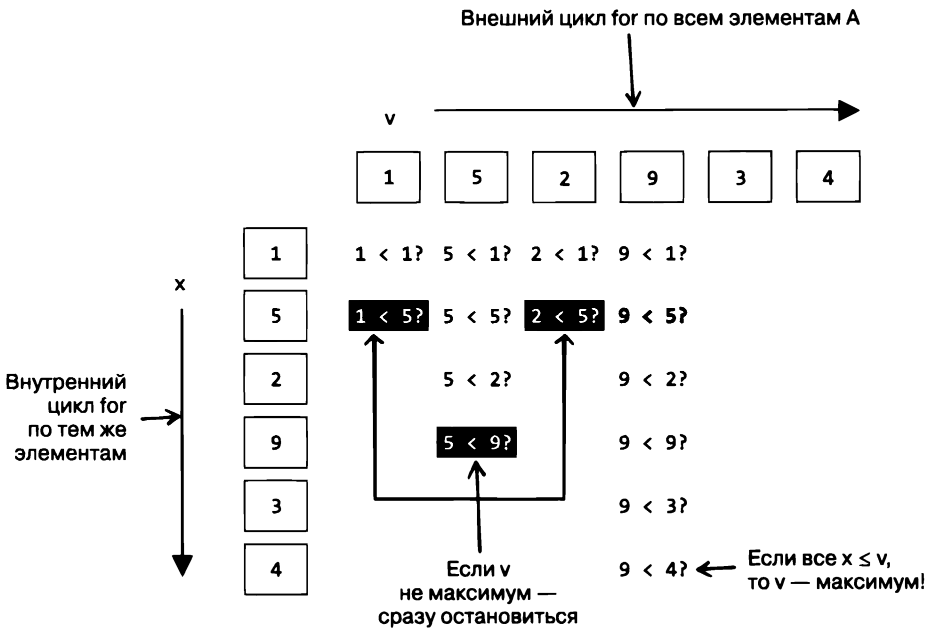


Рис. 1.3. Как работает alternate()

Попробуем отыскать наихудший вариант входных данных для `alternate()`, в котором количество проделанных сравнений было бы наибольшим. Очевидно, что максимум должен лежать в конце `A`, но вдобавок к этому в наихудшем наборе данных все значения в `A` должны быть упорядочены по возрастанию.

На рис. 1.4 показан наилучший случай, в котором `A = [9 5 2 1 3 4]`, и наихудший случай, в котором `A = [1 2 3 4 5 9]`.

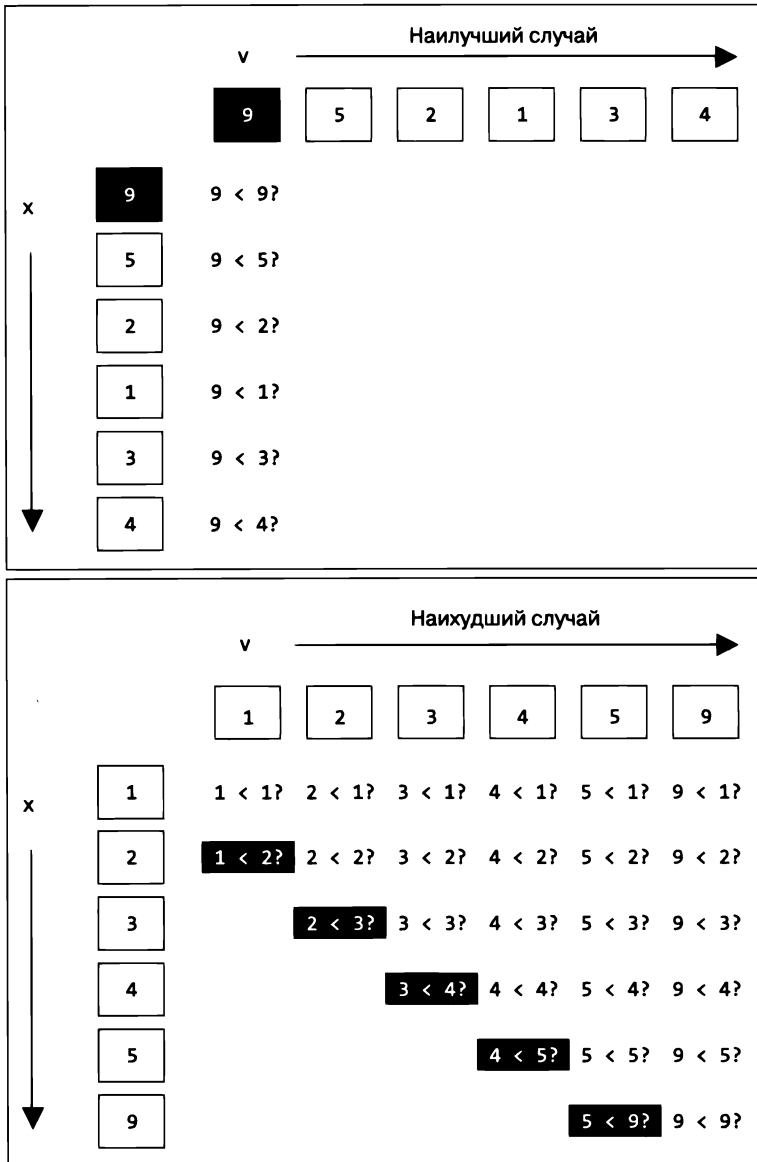


Рис. 1.4. Как работает alternate() в лучшем и худшем случаях

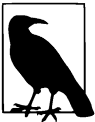
В проиллюстрированном лучшем случае функция делает шесть сравнений >. Если значений всего *N*, сравнений будет *N*. В *худшем случае* подсчет действий слегка сложнее (табл. 1.2). На рис. 1.4 видно, что шесть элементов, упорядочен-

ные по возрастанию, потребовали 26 сравнений. Немного арифметики, и становится понятно, что для N элементов число сравнений будет равно $\frac{N^2 + 3N - 2}{2}$.

Таблица 1.2. Сравнение работы largest() и alternate() в худших случаях

N	largest() (количество сравнений)	alternate() (количество сравнений)	largest() (время в миллисекундах)	alternate() (время в миллисекундах)
8	7	43	0.001	0.001
16	15	151	0.001	0.003
32	31	559	0.002	0.011
64	63	2143	0.003	0.040
128	127	8383	0.006	0.153
256	255	33 151	0.012	0.599
512	511	131 839	0.026	2.381
1024	1023	525 823	0.053	9.512
2048	2047	2 100 223	0.108	38.161

Пока данных мало, это еще терпимо. Но если удвоить размер набора данных, количество сравнений у `alternate()` фактически увеличится вчетверо, а `largest()` останется далеко позади. Последние два столбца табл. 1.2 показывают быстродействие обоих алгоритмов на сотне случайно выбранных наихудших наборов данных размером N. Время работы `alternate()` тоже увеличивается вчетверо.



Здесь измерялось время, потраченное алгоритмом на обработку набора размером N. Представлены данные о самом быстром (потребовавшем меньше всего времени) решении среди всех попыток. Такой подход лучше обычного вычисления среднего времени по всем однотипным наборам данных, потому что в среднее может закрасться измерение, которое оказалось большим не по вине алгоритма.

В книге будут появляться таблицы с подсчетом количества выполненных действий и затраченного времени наподобие табл. 1.2, где измерялось количество сравнений >. Строки таких таблиц соответствуют различным объемам входных данных. Если просмотреть таблицу сверху вниз, становится понятно, как растут показания в каждом столбце по мере удвоения размера данных.

Подсчет количества сравнений показывает, как работают функции `largest()` и `alternate()`. При удвоении *N* количество сравнений в `largest()` удваивается, а в `alternate()` — увеличивается вчетверо. Это поведение вполне стабильно, и несложно предсказать, как оба алгоритма поведут себя на данных большего размера. На рис. 1.5 показано, что количество сравнений в функции `alternate()` (отмечено по оси *Y* с левой стороны) довольно точно соответствует ее производительности (затраченное время отмечено по оси *Y* с правой стороны).

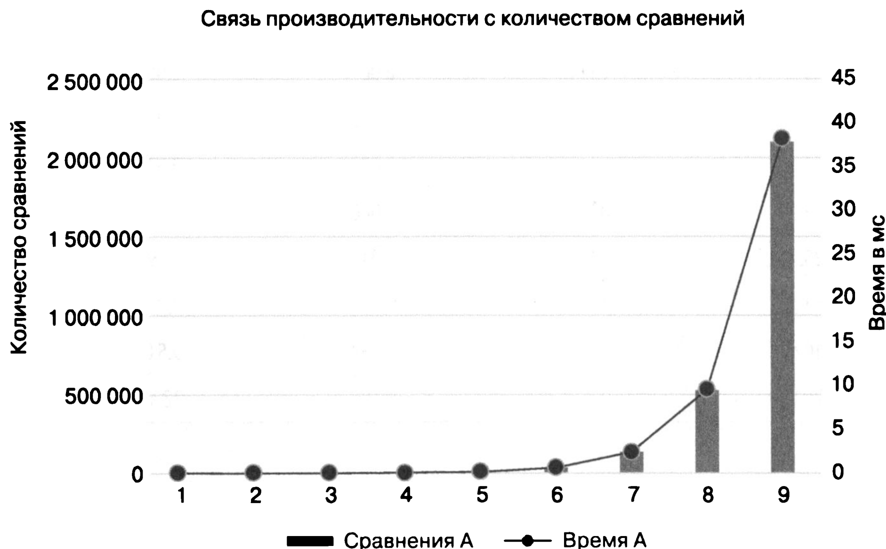


Рис. 1.5. Соотношение количества сравнений и времени работы

Можем себя поздравить: мы только что сделали важный шаг на пути исследования алгоритмов — оценили относительную производительность двух функций, сравнив количество выполняемых ими действий. Конечно, теперь можно взять обе реализации, снабдить тестами производительности, в которых размер входных данных несколько раз удваивается, и измерить ее на практике, как это сделано в примерах к книге. В действительности достаточно того, что оценка схемы алгоритмов неплохо предсказывает их работу, и из оценки следует, что `largest()` быстрее, чем `alternate()`.

Наша функция `largest()` и встроенная функция Python `max()` реализуют один и тот же алгоритм, но, как это видно из табл. 1.3, `largest()` работает значительно — раза в четыре — медленнее, чем `max()`. Дело в том, что Python — *интерпретируемый* язык программирования: написанная программа транслируется в промежуточное представление, называемое байт-кодом, а при выполнении

программы запускается интерпретатор Python, который читает, интерпретирует и выполняет инструкции байт-кода. Встроенные же функции, например `max()`, — часть самого интерпретатора: пока такая функция обрабатывает объект, не нужно ничего дополнительно интерпретировать. Поэтому встроенные функции всегда быстрее тех, что написаны на Python[1]. Следует заметить, что во всех случаях реализация одного и того же алгоритма должна приводить к одинаковому изменению быстродействия при изменении размера данных — например, при удвоении *N* время работы и `largest()`, и `max()` тоже удваивается как в *наихудшем*, так и в *наилучшем* случае.

В табл. 1.3 показано, что время, которое тратится на работу с данными при увеличении их размера, вполне предсказуемо. Если знать, сколько времени потратили `largest()` и `max()` на наборе длиной *N*, можно вычислить время, которое они затратят, если увеличить этот набор вдвое.

Таблица 1.3. Быстродействие largest() и max() в лучшем и худшем случаях

N	largest(), худший случай	max(), худший случай	largest(), лучший случай	max(), лучший случай
4096	0.20	0.05	0.14	0.05
8192	0.40	0.11	0.29	0.10
16 384	0.80	0.21	0.57	0.19
32 768	1.60	0.41	1.14	0.39
65 536	3.21	0.85	2.28	0.78
131 072	6.46	1.73	4.59	1.59
262 144	13.06	3.50	9.32	3.24
524 288	26.17	7.00	18.74	6.50

Теперь слегка поменяем условия задачи — так она станет поинтереснее.

[1] Например, что требуется *интерпретировать* в цикле (то есть многократно) при выполнении largest()? В первую очередь — *поиск имен* переменных: Python не может быть уверен, что, скажем, idx вообще существует на каждом следующем проходе цикла, ведь это имя вполне могли удалить! Дополнительное время тратится на вычисление выражений, ибо для каждого промежуточного результата приходится заводить отдельный объект Python, а затем уничтожать его. Схожая ситуация с операцией индексации и т. д. Встроенная же функция max() тратит время только на обход конкретного выданного ей списка и на сравнение. — *Примеч. пер.*

Поиск двух наибольших значений в произвольном списке

Разработаем алгоритм поиска *двух* наибольших значений произвольного списка (второе может быть меньше первого или равно ему, но точно не меньше всех остальных). Наверняка же достаточно немного подправить уже имеющийся алгоритм `largest()`. Кстати, самое время попробовать справиться с задачей самостоятельно, а потом уже вернуться к нашему примеру!

В примере 1.4 показано возможное решение этой задачи.

Пример 1.4. Поиск двух максимумов с помощью отредактированной функции largest_two()

```
def largest_two(A):
  my_max,second = A[:2]                    ❶
  if my_max < second:
    my_max,second = second,my_max

     for idx in range(2, len(A)):
       if my_max < A[idx]:                 ❷
         my_max,second = A[idx],my_max
       elif second < A[idx]:               ❸
         second = A[idx]
     return (my_max, second)
```

❶ Предположим, что `my_max` и `second` — первые два значения в списке. Если порядок обратный — поменяем их местами.

❷ Если `A[idx]` больше подходит на роль максимума, сделаем `my_max` равным `[idx]`, а `second` — равным `my_max`.

❸ Если `A[idx]` больше `second`, но меньше `my_max`, обновим только его.

Функция `largest_two()` работает похоже на `largest()`. Сначала предполагается, что `my_max` и `second` — это первые два элемента `A` (порядок проверяется). Затем для всех оставшихся элементов `A` (сколько их? Ну да, $N - 2$) проверяется очередной `A[idx]`, и, если он больше `my_max`, обновляются обе переменные, а если он больше только `second`, обновляется только `second`.

Посчитать количество выполненных сравнений не так-то просто, потому что оно опять *зависит от порядка данных в наборе*.

Реже всего `largest_two()` будет сравнивать значения, если условие оператора `if` внутри цикла окажется всегда истинным. Например, если в `A` каждый следующий

элемент больше предыдущего, то сравнение их оператором < всегда истинно, так что всего таких сравнений будет $N - 2$ — плюс еще одно, которое мы сделали в самом начале функции. Выходит, что в *лучшем случае* нам потребуется только $N - 1$ сравнений для поиска двух максимумов. Сравнение в условии при `elif` в *лучшем случае* не используется вообще.

Можно попробовать построить *наихудший* для `largest_two()` набор данных — в нем сравнение в условии оператора `if` внутри цикла всегда должно быть ложным.

Наверняка уже понятно, что больше всего сравнений `largest_two()` делает, когда элементы в `A` упорядочены по убыванию. Строго говоря, в *наихудшем случае* должны выполняться оба сравнения на каждом обороте цикла, что дает нам оценку $1 + 2 (N - 2) = 2N - 3$ действий.

Таким образом, функция `largest_two()`:

- в *лучшем случае* тратит $N - 1$ сравнений на поиск обоих ответов;
- в *худшем случае* тратит на аналогичный поиск $2N - 3$ сравнений.

Но правда ли это и есть «лучший» алгоритм поиска двух наибольших значений в произвольном списке и наши исследования закончены? Среди имеющихся алгоритмов можно выбирать по нескольким признакам.

- *Дополнительная память*. Не требуется ли в алгоритме дублировать исходные данные?
- *Трудоемкость*. Много ли строк в исходной программе?
- *Изменение исходных данных*. Требуется ли в алгоритме менять *непосредственно* введенные данные, или их можно не трогать?
- *Скорость*. Действительно ли алгоритм работает не хуже других на всех возможных входных наборах данных?

В примере 1.5 рассмотрим еще три алгоритма, которые решают ту же задачу. Функция `sorting_two()` создает из `A` новый — отсортированный по убыванию — список, так что первые два его элемента и есть ответ. Функция `double_two()` находит максимум в `A` с помощью `max()`, создает копию `A`, удаляет оттуда этот элемент и ищет второй ответ тем же самым `max()` по урезанной копии. Функция `mutable_two()` находит индекс наибольшего элемента в списке, удаляет его оттуда на время, ищет второй ответ в урезанном списке, после чего вставляет удаленный элемент обратно. Первые два алгоритма нуждаются в дублировании исходных данных, третий изменяет их непосредственно; для всех трех алгоритмов в наборе данных должно быть больше одного элемента.

Пример 1.5. Еще три способа решить задачу средствами самого Python

```
def sorting_two(A):
  return tuple(sorted(A, reverse=True)[:2])      ❶

def double_two(A):
  my_max = max(A)                                ❷
  copy = list(A)
  copy.remove(my_max)                            ❸
  return (my_max, max(copy))                     ❹

def mutable_two(A):
  idx = max(range(len(A)), key=A.__getitem__)    ❺
  my_max = A[idx]                                ❻
  del A[idx]
  second = max(A)                                ❼
  A.insert(idx, my_max)                          ❽
  return (my_max, second)
```

❶ Создать из `A` новый отсортированный список и вернуть первые два его элемента.

❷ Использовать встроенную функцию `max()` и найти максимум.

❸ Создать дубликат списка `A` и удалить из него этот максимум.

❹ Вернуть кортеж из сходного максимума и максимума в урезанной копии.

❺ Трюк Python, который позволяет найти *индекс* наибольшего значения, а не само наибольшее значение.

❻ Запомнить максимум `my_max` и удалить его из `A`.

❼ Найти еще один `max()` в усеченном списке.

❽ Вставить максимальное значение `my_max` на место.

Трюк ❺ работает так. Как и все операции с объектами в Python, операция индексирования (то есть квадратные скобки) имеет эквивалентный ей метод — `.__getitem__()`. Параметр `key=функция` в функции `max(последовательность)` означает, что максимум будет вычисляться не в исходной последовательности *элемент*$_0$, *элемент*$_1$..., а в последовательности *функция(элемент*$_0$*), функция(элемент*$_1$*)*..., а в качестве результата функция вернет некоторый *элемент*$_m$. В нашем примере элементы — это `0`, `1`... `m`... `len(A)-1`, то есть индексы всех объектов в `A`, а максимум вычисляется среди `A.__getitiem__(0)`, `A.__getitiem__(1)`..., то есть среди `A[0]`, `A[1]`..., а возвращается при этом индекс `m`.

Все три способа не используют сравнений явно, потому что используют встроенные функции Python. Функции `sorting_two()` и `double_two()` копируют исходный список `A`, а `largest_two()` — нет; видимо, это не обязательно. Вдоба-

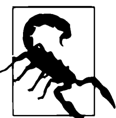
вок *сортировать весь список* ради всего двух первых элементов тоже кажется излишним. Для обеих функций, задействующих *дополнительную память*, эту память можно посчитать тем же способом, каким мы считали количество сравнений — и там и там получится прямо пропорционально *N*. Третий вариант, `mutable_two()`, ненадолго изменяет `A`: удаляет оттуда элемент, а затем добавляет обратно. В программе, откуда была вызвана `mutable_two()`, возможно, не рассчитывали на то, что `A` будут изменять.

Если позволить себе конструкции Python посложнее — ввести специальный класс `RecordedItem`[1], можно отследить, сколько операций сравнения < потребует любой алгоритм. Из табл. 1.4 видно, что `double_two()` делает больше всего сравнений на возрастающей последовательности, а все `largest_two()` и все остальные — на убывающей. Последний столбец — «Вперемежку» — характеризует работу функций на последовательности, в которой на четных местах элементы возрастают, а на нечетных — убывают. Например, для $N = 8$ перемежающаяся последовательность выглядит как `[0,7,2,5,4,3,6,1]`.

Таблица 1.4. Производительность других вариантов решения на нисходящих и восходящих последовательностях

Алгоритм	По возрастанию	По убыванию	Вперемежку
`largest_two`	524 287	1 048 573	1 048 573
`sorting_two`	524 287	524 287	2 948 953
`double_two`	1 572 860	1 048 573	1 048 573
`mutable_two`	1 048 573	1 048 573	1 048 573
`tournament_two`	524 305	524 305	524 305

Функция `tournament_two()`, алгоритм которой описан ниже, стабильно показывает наименьшее количество сравнений на любых входных данных. Ее логика будет знакомой поклонникам баскетбола.



Если один алгоритм решения задачи дает *наихудший* результат на некотором наборе данных, другой алгоритм решения той же самой задачи на том же самом наборе данных вовсе не обязан работать так же медленно. У разных подходов разные слабые места, и надо еще постараться их найти.

[1] Класс-обертка RecordedItem задает собственный метод `__lt__()`, который соответствует операции <. В нем, помимо проверки, делается и увеличение счетчика. То же самое делает `__gt__()` для операции >. — *Примеч. авт.*

Турнирное дерево

В состязаниях на вылет выявляется победитель среди нескольких команд. Лучше всего, если количество команд изначально равно какой-нибудь степени двойки, например 16 или 64. Розыгрыш состоит из нескольких туров, в каждом из которых все оставшиеся команды разбиваются на пары для игры. Каждый проигравший в паре выбывает из соревнований, а остальные переходят на следующий круг. Команда, выигравшая в финале, объявляется победителем.

Предположим, нам надо найти максимум в списке `p = [3,1,4,1,5,9,2,6]` длиной $N = 8$. На рис. 1.6 показан розыгрыш на вылет, в первом туре которого попарно сравниваются восемь значений, и те, что больше, переходят на второй[1]. В туре «Великолепной Восьмерки» выбывают четыре значения, и остается `[3,4,9,6]`, из тура «Чемпионской Четверки» в финал выходят `[4,9]`, и в результате победителем становится `9`[2].

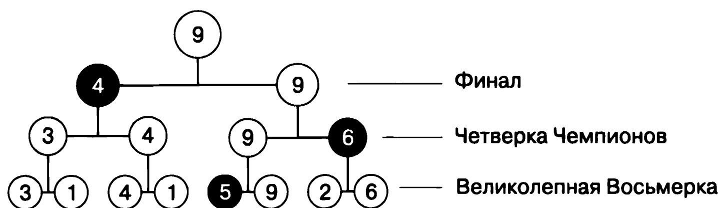


Рис. 1.6. Турнирное дерево с восемью участниками

Для применения турнирного дерева требуется семь сравнений (по одному на игру), и это обнадеживает: как мы уже говорили, это означает, что на поиск максимума в наборе данных размером N требуется $N - 1$ сравнений. Если запоминать все эти сравнения, нетрудно показать, что второе наибольшее значение находится быстро.

Где может «прятаться» второе наибольшее значение, если победителем объявлено `9`? Начнем со значения `4`, раз уж оно добралось до финала и проиграло

[1] В случае ничьей, то есть равных значений, берется первое из них. — *Примеч. авт.*

[2] Автор использует термины турнира баскетбольной лиги США первого дивизиона: региональные полуфиналы (Sweet Sixteen), региональные финалы (Elite Eight) и национальные полуфиналы (Final Four). Знатокам турнира эти термины хорошо известны, но я решил перевести эти названия, сохраняя принцип «первые буквы повторяются». — *Примеч. пер.*

только там. Однако победитель, `9`, участвовал еще в двух играх, так что надо проверить еще двух проигравших — `6` из тура «Четверки Чемпионов» и `5` из тура «Великолепной Восьмерки». Получается, что второй ответ — `6`.

Чтобы выяснить, что `6` — второе наибольшее значение, для длины списка 8 достаточно двух дополнительных сравнений — «4 меньше 6?» и «6 меньше 5?». То, что $8 = 2^3$, а сравнений было $3 - 1 = 2$ — не совпадение. Действительно, для $N = 2^K$ необходимо $K - 1$ дополнительных сравнений, при этом K оказывается количеством туров в розыгрыше.

Для $8 = 2^3$ элементов алгоритму требуется розыгрыш из трех туров. На рис. 1.7 приведен розыгрыш из пяти туров, соответствующий 32 элементам. Если удвоить количество элементов, потребуется еще один тур. Иными словами, в туре под номером K может участвовать 2^K элементов. Нужно найти максимум среди 64 элементов? Потребуется шесть туров, потому что $2^6 = 64$.

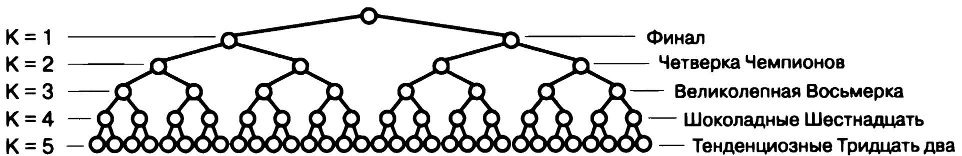
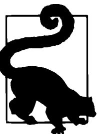


Рис. 1.7. Турнирное дерево с 32 участниками

Чтобы определить, сколько туров потребуется для произвольного N, используем *логарифмическую* функцию `log()` — это обратная к показательной функции, `exp()`. Для $N = 8$ элементов на весь розыгрыш требуется три тура, потому что $2^2 = 8$, и, стало быть, $\log_2 8 = 3$. В нашей книге, как и в большинстве задач оценки сложности, используется логарифм с основанием 2 — двоичный.



Большинство настольных калькуляторов (а заодно Microsoft Excel) по команде log() вычисляют десятичный логарифм (по основанию 10). В них также есть команда ln() для вычисления натурального логарифма, основание которого равно константе e (примерно 2.7182818). Чтобы вычислить двоичный логарифм с помощью любой из этих функций, надо поделить результат на логарифм двух: $\log(N) / \log(2)$.

Если N — степень двойки, например 64 или 65 536, в розыгрыше будет $\log_2(N)$ туров, а это значит, что потребуется еще $\log_2(N) - 1$ сравнений. В примере 1.6 приведен алгоритм, который сводит к минимуму количество сравнений за счет дополнительной памяти, где откладываются результаты этих сравнений.

Пример 1.6. Поиск двух наибольших значений A с помощью турнирного дерева

```
def tournament_two(A):
  N = len(A)
  winner = [None] * (N-1)              ❶
  loser = [None] * (N-1)
  prior = [-1] * (N-1)                 ❷

    idx = 0
    for i in range(0, N, 2):           ❸
      if A[i] < A[i+1]:
        winner[idx] = A[i+1]
        loser[idx] = A[i]
      else:
        winner[idx] = A[i]
        loser[idx] = A[i+1]
      idx += 1

    m = 0                              ❹
    while idx < N-1:
      if winner[m] < winner[m+1]:      ❺
        winner[idx] = winner[m+1]
        loser[idx] = winner[m]
        prior[idx] = m+1
      else:
        winner[idx] = winner[m]
        loser[idx] = winner[m+1]
        prior[idx] = m
      m += 2                           ❻
      idx += 1

    largest = winner[m]
    second = loser[m]                  ❼
    m = prior[m]
    while m >= 0:
      if second < loser[m]:            ❽
        second = loser[m]
      m = prior[m]

    return (largest, second)
```

❶ В этих списках мы станем хранить индексы победителей и проигравших в игре, которых будет $N - 1$.

❷ Когда значение в позиции `m` проходит на очередной тур, в `prior[m]` записывается позиция этого значения в предыдущем туре. Для первого тура такой информации нет, поэтому в начале этого списка хранится `-1`.

❸ Первый тур состоит из $N / 2$ игр, то есть требует $N / 2$ сравнений на «меньше» в парах «победитель — проигравший».

❹ Игры победителей во всех следующих турах с записью позиции выигравшего в каждой игре.

❺ Еще $N / 2 - 1$ сравнение.

❻ Увеличим `m` на 2 — это игра двух следующих победителей. Когда `idx` достигнет $N - 1$, в `winner[m]` окажется наибольшее значение.

❼ Первый кандидат на второе место; надо еще проверить остальных проигравших чемпиону, возможно, они больше подходят для второго места.

❽ Не больше $\log_2(N) - 1$ дополнительное сравнение на «меньше».

Рисунок 1.8 показывает работу нашего алгоритма. Начальный шаг превращает N значений исходного списка `A` в $N / 2$ значений в `winners` и `losers`; на рис. 1.6 это четыре пары. На следующем шаге с каждым проходом цикла `while` победитель и проигравший в очередной игре под номером `idx` (они находятся на соседних позициях `m` и `m+1`) помещаются соответственно в `winner[idx]` и `loser[idx]`.

Массив A

3	1	4	1	5	9	2	6

Начальный шаг (m = 0, idx = 4)

	0	1	2	3	4	5	6
Победитель	3	4	9	6			
Проигравший	1	1	5	2			

Продвижение 1 (m = 2, idx = 5)

	0	1	2	3	4	5	6
Победитель	3	4	9	6	4		
Проигравший	1	1	5	2	3		

Продвижение 2 (m = 4, idx = 6)

	0	1	2	3	4	5	6
Победитель	3	4	9	6	4	9	
Проигравший	1	1	5	2	3	6	

Продвижение 3 (m = 6, idx = 7)

	0	1	2	3	4	5	6
Победитель	3	4	9	6	4	9	9
Проигравший	1	1	5	2	3	6	4

Рис. 1.8. Пошаговое выполнение алгоритма с использованием турнирного дерева

В `prior[idx]` мы записываем предыдущую позицию выигравшего, с которой он попал в эту игру (обозначено стрелкой справа налево). После трех шагов мы имеем всю информацию о розыгрыше, и алгоритм проверяет всех проигравших в играх с чемпионом — последовательность, которую можно проследить по стрелкам от чемпиона назад. Второе наибольшее значение можно найти всего двумя сравнениями, что больше — начальный кандидат (найденный в `loser[6]`), `loser[5]` или `loser[2]`.

Итак, мы описали алгоритм поиска двух наибольших элементов `A`, которому нужно всего $N - 1 + \log_2(N) - 1 = N + \log_2(N) - 2$ сравнений на «меньше» для любого N, равного степени двойки. А насколько практична функция `tournament_two()`? Работает ли она быстрее `largest_two()`? Если считать только сравнения элементов на «меньше», `tournament_two()` должна быть быстрее. На наборе данных длиной N = 65 536 функция `largest_two()` выполняет 131 069 сравнений, а `tournament_two()` — только 65 536 + 16 – 2 = 65 550, то есть примерно половину. Но история на этом не кончается.

Таблица 1.5 показывает, что функция `tournament_two()` значительно медленнее всех своих соперниц! Достаточно посмотреть, сколько времени у нее уходит на обработку ста случайных наборов данных, размер которых растет от 1024 до 2 097 152 элементов. Раз уж мы об этом заговорили, давайте еще добавим в таблицу производительность функций из примера 1.5. Если программу с примерами запускать на другом компьютере, конкретные результаты получатся другими, но *общая закономерность* сохранится.

Таблица 1.5. Сравнение времени работы в миллисекундах всех четырех алгоритмов

N	double_two	mutable_two	largest_two	sorting_two	tournament_two
1024	0.00	0.01	0.01	0.01	0.03
2048	0.01	0.01	0.01	0.02	0.05
4096	0.01	0.02	0.03	0.03	0.10
8192	0.03	0.05	0.05	0.08	0.21
16 384	0.06	0.09	0.11	0.18	0.43
32 768	0.12	0.20	0.22	0.40	0.90
65 536	0.30	0.39	0.44	0.89	1.79
131 072	0.55	0.81	0.91	1.94	3.59
262 144	1.42	1.76	1.93	4.36	7.51
524 288	6.79	6.29	5.82	11.44	18.49
1 048 576	16.82	16.69	14.43	29.45	42.55
2 097 152	35.96	38.10	31.71	66.14	...

С непривычки табл. 1.5 выглядит как стена чисел. Если запускать наши функции на другом компьютере — с меньшей памятью или менее мощным процессором, — это будут другие числа, но некоторые закономерности можно проследить безотносительно к быстродействию компьютера. В первую очередь рост чисел в каждом столбце: при удвоении размера входного набора время выполнения увеличивается тоже примерно вдвое.

В таблице можно заметить кое-что неожиданное: к примеру, `double_two()` поначалу ведет себя как самое быстрое решение, но с ростом N (после $N > 262.144$) уступает пальму первенства `largest_two()`. А хитрая наша `tournament_two()` оказалась ужасно медленной — она тратит много времени на создание и обработку вспомогательных списков, размер которых даже больше, чем объем входных данных. Она настолько медленно работает, что на самых больших наборах даже не проверяется — это было бы слишком долго.

Чтобы получить представление обо всех этих числах, посмотрим на рис. 1.9, где в виде графиков представлена зависимость производительности от роста объема данных.

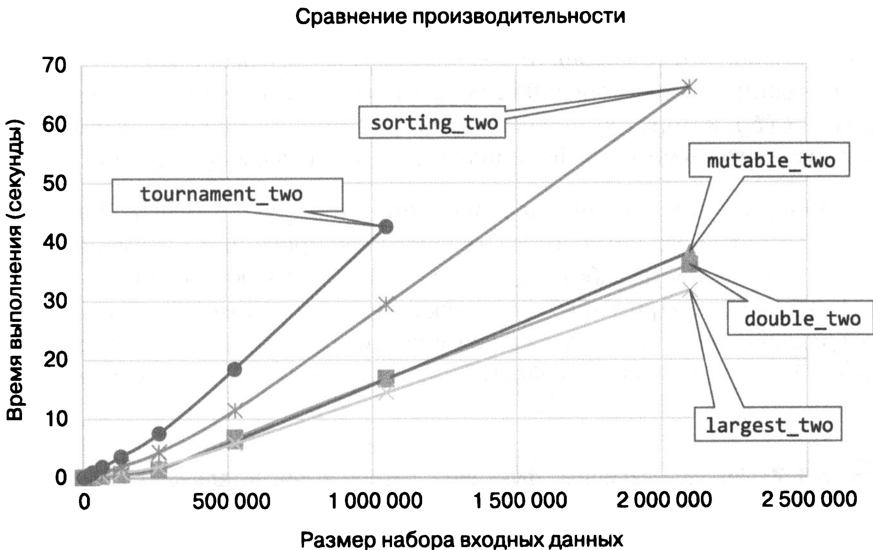


Рис. 1.9. Сравнение тестов производительности

Графики показывают особенности работы всех пяти алгоритмов.

- Видно, что производительность `mutable_two()`, `double_two()` и `largest_two()` примерно одинакова, но явно отличается от производительности двух других функций. Можно сказать, что эти три функции образуют «семейство»:

графики их производительности — прямые линии, предсказать продолжение которых, по-видимому, просто.

- Функция `tournament_two()` — самая медленная, и ее график заметно отличается от прочих. Измерений сделано немного, поэтому до конца не ясно, будет ли график продолжать расти вверх или тоже в конце концов превратится в прямую.
- Функция `sorting_two()` явно получше, чем `tournament_two()`, однако медленнее остальных трех. А ее график? Будет ли он и дальше изгибаться кверху или стремиться к прямой?

Чтобы понять, отчего графики имеют такой вид, надо обратиться к двум важным понятиям, которые определяют сложность, присущую алгоритмам.

Сложность по времени и сложность по памяти

Сложно посчитать, сколько конкретных машинных инструкций (например, сложений, присваиваний или ветвлений) выполнила программа: это зависит от языка программирования, тем более что некоторые из них, например Python или Java, требуют для работы интерпретатор. Но если общее число выполненных инструкций посчитать *можно*, то можно исследовать, как оно зависит от объема входных данных. Задача оценки *сложности по времени* — получить некоторую формулу $C(N)$, которая вычисляла бы количество инструкций, выполненных некоторым алгоритмом, как функцию от N — размера набора входных данных.

Предположим, выполнение одной машинной инструкции центральным процессором некоторого компьютера требует фиксированного времени t. Тогда время T, затраченное на работу алгоритма на этом компьютере, можно выразить как $T(N) = t \times C(N)$. Пример 1.7 подтверждает догадку о том, что самое важное — это *структура* программы. Можно в точности посчитать, сколько потребуется сложений вида `ct = ct + 1` для работы функций `f0()`, `f1()`, `f2()` и `f3()` на входных данных размером N.

Пример 1.7. Четыре различные функции с разной вычислительной сложностью

```
def f0(N):
  ct = 0
  ct = ct + 1
  ct = ct + 1
  return ct

def f1(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
  return ct

def f2(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
  return ct

def f3(N):
  ct = 0
  for i in range(N):
    for j in range(N):
      ct = ct + 1
  return ct
```

Функция f0() всегда выполняет одно и то же количество действий, независимо от *N*. Количество действий в f2() всегда в семь раз больше, чем в f1(); при удвоении *N* обе функции выполняют в два раза больше действий. Количество действий, выполняемых f3(), растет гораздо быстрее. Мы такое уже видели: *N* удваивается, сложность f3(N) вырастает в четыре раза (табл. 1.6). Получается, что f1() и f2() больше похожи друг на друга, чем на f3(). В следующей главе мы обсудим важную роль, которую играют циклы и вложенные циклы при оценке сложности алгоритма.

Таблица 1.6. Количество действий в различных функциях

N	f0	f1	f2	f3
512	2	512	3584	262 144
1024	2	1024	7168	1 048 576
2048	2	2048	14 336	4 194 304

Когда мы исследуем алгоритм, важно установить его *сложность по памяти* — посчитать, сколько дополнительной памяти требуется на обработку входных данных размером *N*. Под *памятью* может подразумеваться занимаемое место в файловой системе или объем оперативной памяти, который требуется для работы. Функция largest_two() потребляет меньше всего памяти: в ней определены лишь три переменные — my_max, second и переменная цикла idx. Каким бы ни был объем входных данных, размер дополнительной памяти не меняется. Следовательно, сложность по памяти *не зависит от размера входных данных*, то есть является константой. Так же ведет себя и функция mutable_two(). А вот tournament_two() заводит аж три дополнительных списка размером *N* – 1 — winner, loser и prior. Так что при увеличении *N* объем дополнительной памяти увеличивается *прямо пропорционально объему входных данных*[1]. Необходимость создавать турнирную таблицу делает работу tournament_two() заметно медленнее по сравнению с largest_two(). Еще две функции — double_two() и sorting_two() — дублируют входные данные (список A), так что их потребление памяти скорее похоже на таковое у tournament_two(), чем у largest_two(). В книге мы будем исследовать и сложность по времени, и сложность по памяти каждого алгоритма.

Если еще раз посмотреть на табл. 1.5, можно заметить, что числа в столбце largest_two с каждой строкой становятся примерно в два раза больше. Мы уже подчеркивали, что столбцы double_two и mutable_two устроены в целом так же.

---

[1] При вычислении сложности по памяти размер самого входного набора не учитывается. Интересен объем данных, которые пришлось породить вдобавок к нему. — *Примеч. авт.*

Значит, время работы этих алгоритмов *прямо пропорционально объему входных данных*, который тоже удваивается на каждой строке. Это важное свойство, ибо эти функции быстрее, чем `sorting_two()`, а у нее и график быстродействия выглядит по-другому — менее эффективным. Самая медленная — `tournament_two()`, скорость ее работы падает настолько быстрее, чем вдвое, что под конец, на самых больших наборах, мы ее даже не запускали.

С научной точки зрения нельзя просто заявить, что графики производительности `largest_two()` и `mutable_two()` похожи. Нужны формальное теоретическое обоснование и способ фиксации этого утверждения. В следующей главе вы познакомитесь с математическим инструментарием, который необходим для грамотного исследования производительности алгоритмов.

Заключение

В этой главе мы обсудили целый спектр хороших и разных алгоритмов. Вы научились предсказывать производительность алгоритма на входных данных размером N путем подсчета количества выполненных действий. Кроме того, мы посмотрели, как опытным путем измерить производительность конкретной реализации алгоритма. В обоих случаях оказалось возможным определить, с какой скоростью падает производительность алгоритма при удвоении объема N входных данных.

Мы рассмотрели несколько важных понятий, в частности:

- *сложность по времени*, которая оценивается подсчетом количества основных действий алгоритма на наборе входных данных размером N;
- *сложность по памяти*, которая оценивается измерением дополнительного объема памяти, который потребуется алгоритму для обработки набора входных данных размером N.

В следующей главе мы рассмотрим математический инструментарий *асимптотического анализа* и на этом завершим обзор методов формального исследования алгоритмов.

Тренировочные задания

1. *Определитель палиндромов.* Палиндром — слово, буквы которого, расположенные в обратном порядке, дают то же самое слово, — например, «казак». Нужно придумать и оформить в виде функции алгоритм, определяющий,

что данная строка — это палиндром. Проверить на практике, что написанная функция работает быстрее двух других, предложенных в примере 1.8.

Пример 1.8. Довольно медленные функции — определители палиндрома

```
def is_palindrome1(w):
  """Сделаем секцию, содержащую все символы в обратном порядке,
  и проверим ее на наличие w."""
  return w[::-1] == w

def is_palindrome2(w):
  """Если первый и последний символы равны, удалим их.
  Если они не равны – вернем False."""
  while len(w) > 1:
      if w[0] != w[-1]:     # если не равны, вернем False
        return False
      w = w[1:-1]           # удаляем в цикле первый и последний символы

    return True             # это точно палиндром
```

После того, как функция заработает, дополните ее: научите определять палиндромы-фразы, в которых пробелы и знаки пунктуации игнорируются, а строчные и прописные буквы не отличаются. Например, вот эта строка — палиндром: `Я не стар, брат Сеня!`.

2. *Вычисление медианы за линейное время.* Есть замечательный алгоритм нахождения медианы в произвольном списке, имеющий линейную сложность (для простоты будем считать, что размер списка нечетен). Изучите исходный текст функций из примера 1.9 и посчитайте количество сравнений с помощью значений упомянутого в этой главе класса `RecordedItem`. В реализации алгоритма из примера используется переупорядочение элементов исходного списка.

Пример 1.9. Алгоритм нахождения медианы в неупорядоченном списке за линейное время

```
import random

def partition(A, lo, hi, idx):
  """Разделение A на две части относительно значения A[idx]."""
  if lo == hi: return lo

  A[idx],A[lo] = A[lo],A[idx]    # ставим в нужную позицию
  i = lo
  j = hi + 1
  while True:
    while True:
      i += 1
      if i == hi: break
      if A[lo] < A[i]: break
```

```
    while True:
      j -= 1
      if j == lo: break
      if A[j] < A[lo]: break

    if i >= j: break
    A[i],A[j] = A[j],A[i]

  A[lo],A[j] = A[j],A[lo]
  return j

def linear_median(A):
  """
  Быстрая реализация поиска медианы в произвольном списке.
  Предполагается, что в списке нечетное число элементов.
  Обратите внимание на то, что при работе алгоритма
  порядок элементов в A меняется.
  """
  lo = 0
  hi = len(A) - 1
  mid = hi // 2
  while lo < hi:
    idx = random.randint(lo, hi)  # случайный выбор допустимого индекса
    j = partition(A, lo, hi, idx)

    if j == mid:
      return A[j]
    if j < mid:
      lo = j+1
    else:
      hi = j-1
  return A[lo]
```

Реализуйте другой способ (для него потребуется дополнительная память): отсортируйте список и верните в качестве ответа его средний элемент (то есть медиану). Сравните быстродействие этого алгоритма с работой `linear_median()`, составив таблицу тестов производительности.

3. *Сортировка подсчетом.* Если известно, что некоторый (произвольный) список `A` состоит только из целых чисел в диапазоне от 0 до M, его можно отсортировать за линейное время, используя дополнительную память размером M.

   Хотя в примере 1.10 есть вложенные циклы — `for` внутри `while`, — количество присваиваний вида `A[pos+idx] = v` все равно ровно N. Докажите это!

Пример 1.10. Сортировка подсчетом за линейное время

```
def counting_sort(A, M):
  counts = [0] * M
  for v in A:
    counts[v] += 1
```

```
pos = 0
v = 0
while pos < len(A):
  for idx in range(counts[v]):
    A[pos+idx] = v
  pos += counts[v]
  v += 1
```

Проанализируйте быстродействие и покажите, что время при удвоении N сортировки N элементов в диапазоне от 0 до M также удваивается.

Внутреннего цикла в `counting_sort()` можно избежать, если воспользоваться присваиванием секций списка. Если написать что-то вроде `список[начало:конец] = [1, 2, 3]`, то элементы **списка** в позициях от **начала** до **конца** заменятся значениями `1, 2, 3`. Измените `counting_sort()` и проверьте на практике, что время работы, как и прежде, удваивается при удвоении N, но новая функция работает на 30 % быстрее старой.

4. Исправьте алгоритм с турнирной таблицей так, чтобы он работал для нечетного количества значений.
5. Насколько хорошо функция из примера 1.11 находит два наибольших значения в списке `A`?

Пример 1.11. Еще одна попытка найти два наибольших значения в неупорядоченном списке

```
def two_largest_attempt(A):
  m1 = max(A[:len(A)//2])
  m2 = max(A[len(A)//2:])
  if m1 < m2:
    return (m2, m1)
  return (m1, m2)
```

При каких условиях функция работает правильно, а при каких — неправильно? Опишите эти условия.

ГЛАВА 2

Анализ алгоритмов

https://t.me/it_boooks/2

В этой главе

- Как пользоваться обозначением «O большое» для определения класса вычислительной сложности алгоритма по времени или по памяти.
- Какие бывают основные классы сложности, например:
  - O(1) — *константный*;
  - O(log N) — *логарифмический*;
  - O(N) — *линейный*;
  - O(N log N) — N-*логарифмический*;
  - O(N^2) — *квадратичный*.
- Как применять *асимптотический анализ* по N для оценки времени работы (или потребляемой памяти) алгоритма на наборе данных размером N.
- Как работать с массивами данных, элементы которых упорядочены по возрастанию.
- Как устроен алгоритм *двоичного поиска* элемента в сортированном списке.

В главе вводятся обозначения и терминология из теории сложности алгоритмов, которые на практике используются для оценки их быстродействия и объема потребляемой ими памяти. Если некоторая программа работает без нареканий — вполне быстро с любыми данными, что попадаются на практике, — скорее всего, ее не стоит улучшать. Книга посвящена тому, с чего начинать, когда требуется улучшить быстродействие, — алгоритмам и структурам данных. И тогда возникают такие вопросы.

- *Точно ли нынешнее решение — самое эффективное?* Вполне могут существовать алгоритмы, которые решают задачу значительно быстрее.

- *Насколько эффективна реализация алгоритма?* В тексте программы могут оказаться конструкции, требующие неожиданно много ресурсов.
- *Может, купить компьютер помощнее?* Одна и та же программа на разных машинах работает с разной скоростью. Мы посмотрим, какими способами специалисты учитывают обновление аппаратного обеспечения при анализе.

Для начала разберемся, как оценивать производительность программ на постоянно растущих объемах данных. Если данных слишком мало, точное время работы измерить сложно — оно будет зависеть и от конкретных значений на входе, и от точности работы компьютерных таймеров. Но на *достаточно больших входных данных* гипотезу о классе вычислительной сложности уже можно сопоставлять с эмпирической моделью.

Как оценить сложность с помощью эмпирической модели

Лучше всего начать сразу с примера, в котором теоретический анализ оказывается в исследовании конкретных программных систем очень даже практическим. Предположим, вы создаете приложение — часть большого задания, которое еженощно обрабатывает большой набор данных. Задание запускается в полночь и должно отработать до шести утра. Ночной набор данных состоит из нескольких миллионов значений — и ожидается, что лет за пять его объем удвоится.

И вот вы уже написали работающий прототип, но тестируете его на небольших наборах данных — по 100, 1000 и 10 000 элементов. В табл. 2.1 зафиксировано время работы прототипа на этих данных.

Таблица 2.1. Время работы прототипа

N	Время в секундах
100	0.063
1000	0.565
10 000	5.946

Можно ли по этим предварительным результатам прогнозировать время работы прототипа на больших наборах данных, например на 100 000 или 1 000 000 значений? Попробуем построить математическую модель на основании *только этих*

результатов и найдем некоторую функцию $T(N)$, которая будет предсказывать время работы на данных заданного объема. Достоверная модель не только вычислит $T(N)$ достаточно близко к трем значениям из табл. 2.1, но еще и предскажет время работы на больших наборах, представленное в табл. 2.2 (известные нам результаты указаны в ней в квадратных скобках).

Возможно, вы пользовались инструментами, которые умеют находить *линейную регрессию* заданного набора данных, например Maple (https://maplesoft.com) или Microsoft Excel (https://microsoft.com/excel). Регрессионные модели можно построить и с помощью SciPy — библиотеки Python для математических, общенаучных и инженерных расчетов. В примере 2.1 показано, как с помощью SciPy подобрать константы `a` и `b` в линейном приближении $TL(N) = a \times N + b$. Функция `curve_fit()` возвращает пару коэффициентов `(a, b)` линейного приближения на основании экспериментальных данных, которые передаются ей в списках `xs` и `ys`.

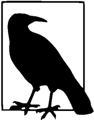
Строго говоря, дело обстоит так. Мы передаем в `curve_fit()` три параметра — векторную функцию `linear_model()`, массив `xs` и массив `ys` с известными значениями этой функции. Функция `linear_model()` — непростая. Во-первых, первый ее операнд, `v`, — это не число, а *массив* `numpy`, умножение которого на число или сложение с числом приводят к поэлементному выполнению соответствующих операций. Остальные операнды этой функции — `a` и `b` — коэффициенты. Их-то и подбирает `curve_fit()` методом наименьших квадратов и возвращает в виде массива. Она также возвращает матрицу ковариации, но для данного исследования она не нужна. По договоренности «ненужные» значения принимает специальная переменная с именем `_`.

Пример 2.1. Построение приближения на неполных данных

```
import numpy as np
from scipy.optimize import curve_fit

def linear_model(v, a, b):
  return a*v + b

# Экспериментальные данные
xs = [100, 1000, 10000]
ys = [0.063, 0.565, 5.946]

# Первое возвращаемое значение - массив из двух коэффициентов
(a, b), _ = curve_fit(linear_model, xs, ys)
print(f'Linear = {a}*N + {b}')
```

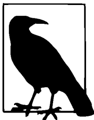
Полученная математическая модель выражается формулой $TL(N) = 0.000596 \times N - 0.012833$. Из табл. 2.2 понятно, что это приближение не слишком достоверно: чем больше размер входного набора, тем сильнее прогноз времени работы прототипа отстает от результатов эксперимента. Можно попробовать повысить степень N до второй и построить *квадратичное приближение*:

```
def quadratic_model(v, a, b):
  return a*v*v + b*v;
```



В приведенном примере можно заметить две хитрости: во-первых, в формуле квадратного трехчлена свободный член c равен нулю (потому что время работы на пустом наборе данных полагается нулевым), во-вторых, вместо универсальной операции возведения в степень используется умножение, потому что оно в данном случае быстрее. На всякий случай стоит еще раз заметить, что v — это целый массив numpy, арифметические операции над которым производятся поэлементно. Например, в результате v * v получается массив квадратов исходных элементов v.

Задача анализа — подобрать такие параметры a и b функции `quadratic_model()`, чтобы полученная формула $TQ(N) = a \times N^2 + b \times N$ соответствовала имеющимся данным. Если исследовать ее по аналогии с примером 2.1, получится формула $TQ(N) = 0.000000003206 \times N^2 + 0.000563 \times N$. Таблица 2.2 показывает, что эта модель тоже недостоверна: ее прогноз времени работы на больших наборах тем сильнее опережает экспериментальные результаты, чем больше данных в наборе.



Большинство найденных нами коэффициентов, например 0.000000003206, то есть 3.206×10^{-9}, — довольно малые числа. Это происходит оттого, что наши алгоритмы работают на больших наборах данных, где N = 1 000 000 и даже больше. Тот же миллион, возведенный в квадрат, — это $1\,000\,000^2 = 10^{12}$, так что стоит ожидать, что нам встретятся и очень большие, и очень маленькие числа.

Последний столбец табл. 2.2 — время работы, оцененное третьей математической моделью, $TN(N) = a \times \log_2 N$: в ней один коэффициент — a, а в формуле присутствует двоичный логарифм. Полученное приближение — $TN(N) = 0.0000448 \times \log_2 N$. Для N = 10 000 000 погрешность приближенного значения находится в пределах 5 % от экспериментального.

Таблица 2.2. Сравнение результатов различных математических моделей с фактическим временем работы

N	Время в секундах	TL	TQ	TN
100	[0.063]	0.047	0.056	0.030
1000	[0.565]	0.583	0.565	0.447
10 000	[5.946]	5.944	5.946	5.955
100 000	65.391	59.559	88.321	74.438
1 000 000	860.851	595.708	3769.277	893.257
10 000 000	9879.44	5957.194	326 299.837	10 421.327

Линейное приближение, `TL`, сильно преуменьшает время работы, а квадратичное, `TQ`, — преувеличивает. Расчет времени для $N = 10\,000\,000$ у `TL` равен 5957 секундам (это минут сто), а у `TQ` — целых 326 300 (почти 91 час). У `TN` предсказывать быстродействие получается куда лучше: ожидаемое время по `TN` — 10 421 секунда (примерно 2.9 часа), а фактическое — 9879 секунд (2.75 часа).

Прототип, слава богу, за ночь все обработал. Но все же имеет смысл просмотреть его исходный текст на предмет задействованных в нем алгоритмов и структур данных. Только тогда можно будет утверждать, что он будет работать примерно столько же на любых данных того же размера.

Отчего формула $a \times N \times \log_2 N$ так хорошо предсказывает поведение программы? Это наверняка зависит от того, какие алгоритмы лежат в основании самой программы. Оценка сложности алгоритмов часто выдает одно из наших трех приближений — линейное, квадратичное или N-логарифмическое. Следующий пример напомнит нам об одном удивительном открытии 60-летней давности[1].

Умножать быстрее, чем в столбик

В примере 2.2 показаны два примера умножения N-значных чисел в столбик, алгоритм которого большинство из нас знает со времен начальной школы. Даже не вдаваясь в подробности алгоритма, мы можем заметить, что в процессе умножения создается N промежуточных произведений, которые записываются

[1] Алгоритм быстрого умножения изобрел в 1960 году 23-летний Анатолий Карацуба, в то время аспирант механико-математического факультета Московского государственного университета. Этот алгоритм используется в Python для умножения больших чисел. — *Примеч. авт.*

ниже исходных чисел, а ответ представляет собой сумму этих (слегка преобразованных) произведений.

Пример 2.2. Школьный алгоритм умножения N-значных чисел

```
   456                      123456
 x 712                    x 712835
   ---                      ------
   912                      617280
  456                      370368
3192                      987648
------                   246912
324672                  123456
                       864192
                       -----------
                       88003757760
```

Чтобы умножить два трехзначных числа друг на друга, надо перемножить девять пар цифр. Два шестизначных числа потребуют уже 36 умножений отдельных цифр. Умножая таким способом два N-значных числа, мы выполняем N^2 умножений цифр. Можно еще заметить, что от увеличения количества цифр в целочисленных множимом и множителе вдвое количество действий «умножение цифры на цифру» растет вчетверо. Другие операции (например, сложение) можно даже не считать — умножение важнее.

Центральный процессор ЭВМ умеет быстро умножать целые числа только фиксированного размера — 32- или 64-разрядные, но с числами большей длины самостоятельно не справляется. Если целое число в Python оказывается слишком большим, оно преобразуется в специальную структуру *PyLong*, которая увеличивается по мере роста числа. Вот на ней и можно исследовать быстродействие умножения произвольных N-значных целых. В табл. 2.3 приведены результаты вычислений четырех эмпирических моделей производительности, которые построены на основании первых пяти экспериментальных данных (в квадратных скобках).

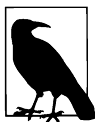
Здесь `TL` — линейное приближение, а `TQ` — квадратичное. `Karatsuba` — это приближение с нестандартной степенью, $a \times N^{1.585}$, а `TKN` — более соответствующая экспериментальным данным формула, $TKN(N) = a \times N^{1.585} + b \times N$, с аккуратно подобранными коэффициентами a и b[1]. `TL` значительно недооценивает время работы умножения. А вот `TQ` — значительно переоценивает, хотя можно было ожидать, что квадратичное приближение окажется достаточно достоверным: количество действий при умножении столбиком растет квадратично, о том же говорит увеличение времени работы вчетверо при росте длины чисел вдвое.

[1] Нестандартная степень 1.585 в формулах — это округленное значение $\log_2 3 \approx$ ≈ 1.58496250072116. — *Примеч. авт.*

Но две другие математические модели оказались намного достовернее для расчета времени, за которое в Python умножаются .-значные целые, потому что для этого используется более эффективный алгоритм Карацубы — умножения длинных целых.

Таблица 2.3. Умножение N-значных чисел

N	Время в секундах	TL	TQ	Karatsuba	TKN
256	[0.0009]	−0.0045	0.0017	0.0010	0.0009
512	[0.0027]	0.0012	0.0038	0.0031	0.0029
1024	[0.0089]	0.0126	0.0096	0.0094	0.0091
2048	[0.0280]	0.0353	0.0269	0.0282	0.0278
4096	[0.0848]	0.0807	0.0850	0.0846	0.0848
8192	0.2524	0.1716	0.2946	0.2539	0.2571
16 384	0.7504	0.3534	1.0879	0.7617	0.7765
32 768	2.2769	0.7170	4.1705	2.2851	2.3402
65 536	6.7919	1.4442	16.3196	6.8554	7.0418
131 072	20.5617	2.8985	64.5533	20.5663	21.1679
262 144	61.7674	5.8071	256.7635	61.6990	63.5884



Внимательный читатель, должно быть, заметил, что «точная» формула (TKN) лучше «упрощенной» (Karatsuba) только на не слишком больших числах — аккурат на тех, для которых известны экспериментальные данные. Для умножения действительно больших чисел в Python используется уже не алгоритм Карацубы, а дискретное разложение Фурье. Класс сложности $N^{\log_2 3}$ у этого алгоритма тот же (кажется, для достижимых чисел он минимально возможный), а накладных расходов на хранение промежуточных результатов, если их предполагается много, меньше. Возможно — эту гипотезу следует проверить! — TKN приближает как раз алгоритм Карацубы, а Karatsuba, вопреки названию, — алгоритм дискретного разложения.

Эмпирический подход, иллюстрируемый данными табл. 2.2 и 2.3, неплох для начала, но недостаточен: мы изучаем не сам алгоритм напрямую, даже не исходный текст программы, а только результаты работы. Когда в книге нам будут встречаться конкретные реализации алгоритмов, мы будем изучать структуру исходных текстов и оценивать быстродействие, выводя подходящую формулу на основании этой структуры.

Классы вычислительной сложности

Иногда довольно просто понять, какой из двух алгоритмов эффективнее справляется с одной и той же задачей: достаточно оценить их вычислительную сложность с помощью математических моделей. Обычно говорят «сложность $O(N^2)$» или в худшем случае «сложность $O(N \log N)$». Чтобы начать в этом разбираться, посмотрим на рис. 2.1. Тем, кто прочел книгу по теории сложности или изучал материалы по анализу алгоритмов, картинка покажется знакомой.

Наша задача — построить математическую модель, которая оценивает *время работы в наихудшем случае* для набора данных размером N. Математики называют это *верхней границей*; проще говоря, алгоритм никогда не проработает дольше приведенной оценки. Соответственно, *нижняя граница* оценки относится к минимальному времени работы и определяется как «алгоритм всегда проработает как минимум столько-то».

Поясню понятия верхней и нижней границы на примере автомобильного спидометра. Задача спидометра — вычислять и показывать приблизительное значение текущей скорости автомобиля. Скорость, которую показывает спидометр, *должна быть не ниже текущей*, чтобы водитель мог соблюдать правила движения. Это и есть *нижняя граница*. В бóльшую сторону показания скорости не должны превышать 110 % плюс 4 км/ч от текущей[1]. В математике это соответствует понятию *верхней границы*.

Текущая скорость автомобиля не может быть больше показаний спидометра. Стало быть, мы можем принять ее за верхнюю границу и относительно несложно высчитать минимальную возможную скорость, которая будет совпадать с соответствующей нижней границей.

На рис. 2.1 изображены три графика, которые представляют три наших приближения «длинного умножения» в Python — TL, TQ и TKN; экспериментальные данные обозначены там же черными квадратиками. $TQ(N)$ — очевидная верхняя граница эксперимента (ибо $TQ(N)$ всегда меньше экспериментального значения на всех N), но по самой иллюстрации видно, что эта граница очень неточная.

[1] Это правила Евросоюза; в США показаниям спидометра достаточно держаться в диапазоне ±5 миль в час. — *Примеч. авт.*

В России правила сложнее: погрешность должна быть положительной (то есть нижняя граница — это текущая скорость); до 60 км/ч верхняя граница не должна превышать +4 % от скорости, а дальше на каждые дополнительные 20 км/ч количество процентов увеличивается на 1: для 80 — +5 %, для 100 — +6 % и т. д. Но это еще не все: дополнительно вводится температурная погрешность, которая тем больше, чем дальше температура от 20 °С. — *Примеч. пер.*

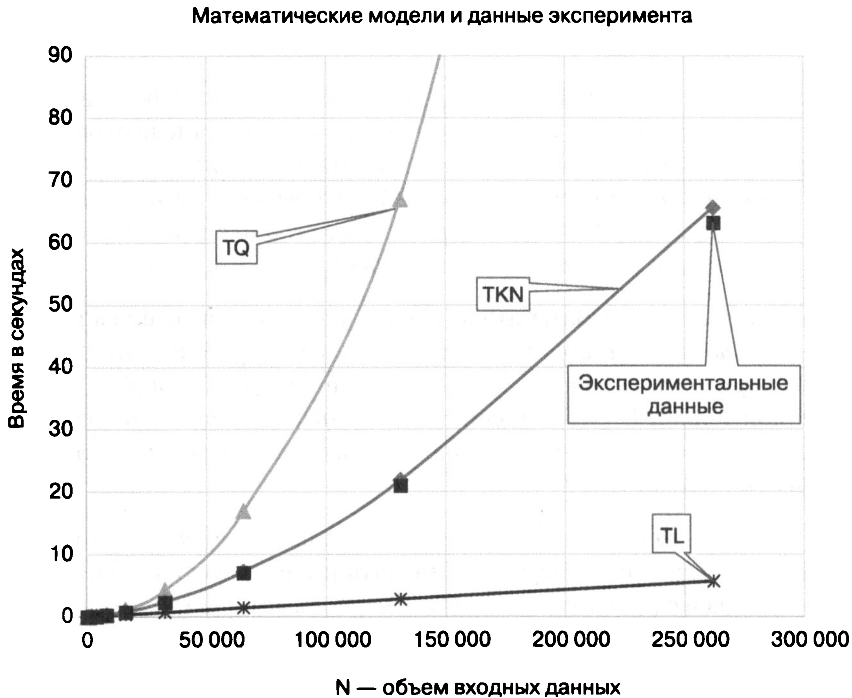


Рис. 2.1. Сравнение математических моделей с результатами измерений производительности

Если повнимательнее посмотреть на табл. 2.3, можно заметить, что, начиная с порогового значения объема входных данных $N = 8192$, приближение $TKN(N)$ тоже всегда больше эмпирических данных для остальных N, но при этом гораздо ближе к ним. Обычно это признак того, что функция стабилизировалась и будет вести себя так впредь для «достаточно больших» N, но для каких именно — зависит от самого алгоритма и его конкретной реализации.

Очевидно также, что линейное приближение $TL(N)$ — нижняя граница времени работы, потому что $TL(N)$ меньше эмпирических данных для любого N. Впрочем, при увеличении N эта формула все сильнее отстает от экспериментальных значений, так что в качестве математической модели становится вполне бесполезной. Более точной нижней границей оказывается формула, названная в табл. 2.3 Karatsuba: $a \times N^{1.585}$.

Оценку производительности можно запускать на разных компьютерах, при этом конкретные числовые характеристики в табл. 2.3 поменяются. Скорость умножения может быть меньше или больше, коэффициенты a и b в $TKN(N)$ могут оказаться другими, сама функция $TKN(N)$ может стабилизироваться на другом

большем или меньшем *N*. Но степень 1.585 в формулах не поменяется, потому что такова организация алгоритма быстрого умножения Карацубы, которым и определяется быстродействие. Никакой суперкомпьютер (за исключением, быть может, квантового) не сможет заставить умножение работать со скоростью линейного приближения *TL*(*N*).

Настало время заняться *асимптотическим анализом* — методикой, которая позволит нам при оценке быстродействия алгоритма вообще не обращать внимания на скорость работы конкретных компьютеров. Мощный компьютер может заставить программу работать быстрее, но не может изменить законы асимптотического анализа!

Асимптотический анализ

Понятие *аддитивной постоянной* часто встречается в обыденной жизни, например, в нашем недавнем разговоре про спидометры. Когда мы говорим «приеду минут через сорок, плюс-минус пять минут», мы имеем в виду как раз аддитивную постоянную.

В *асимптотическом анализе* эта идея развивается и вводится понятие *мультипликативной постоянной* алгоритма. Кто слышал о законе Мура, уже представляет себе, в чем оно состоит. Еще в 1965 году Гордон Мур, один из основателей и генеральных директоров корпорации Intel, высказал предположение, что количество элементов в интегральных микросхемах будет удваиваться ежегодно в течение ближайших десяти лет. В 1975-м он его скорректировал: «количество элементов в интегральных микросхемах удваивается каждые два года». Это наблюдение остается верным более 40 лет, и пока оно верно, скорость работы компьютеров тоже удваивается примерно за два года. Такая «мультипликативная постоянная в истории вычислительной техники» означает, что одна и та же программа на старом компьютере будет *работать в тысячу раз медленнее* (а то и хуже), чем на новом.

Возьмем два алгоритма, решающих одну и ту же задачу. Изучив их уже известным нам способом, предположим, что алгоритм X требует $5N$ действий на наборе данных размером *N*, а алгоритм Y — $2020 \times \log_2 N$ действий на том же входном наборе. Какой из них эффективнее — X или Y?

Мы реализовали оба алгоритма и запускаем программы на двух компьютерах — один назовем C_{fast}, и он в два раза быстрее второго, C_{slow}. На рис. 2.2 приведено количество действий, затраченных каждым алгоритмом на входных данных объема *N*, а также время работы обоих алгоритмов на компьютере C_{fast} (столбцы обозначены как X_{fast} и Y_{fast}) и на компьютере C_{slow} (столбцы X_{slow} и Y_{slow} соответственно).

	Количество действий		Время выполнения			
N	X	Y	X_{slow}	X_{fast}	Y_{fast}	$Y_{fastest}$
4	20	4040	0.0	0.0	2.7	0.0
8	40	6060	0.0	0.0	4.0	0.0
16	80	8080	0.1	0.0	5.4	0.0
32	160	10 100	0.1	0.1	6.7	0.0
64	320	12 120	0.2	0.1	8.1	0.0
128	640	14 140	0.4	0.2	9.4	0.0
256	1280	16 160	0.9	0.4	10.8	0.0
512	2560	18 180	1.7	0.9	12.1	0.0
1024	5120	20 200	3.4	1.7	13.5	0.0
2048	10 240	22 220	6.8	3.4	14.8	0.0
4096	20 480	24 240	13.7	6.8	16.2	0.0
8192	40 960	26 260	27.3	13.7	17.5	0.1
16 384	81 920	28 280	54.6	27.3	18.9	0.1
32 768	163 840	30 300	109.2	54.6	20.2	0.2
65 536	327 680	32 320	218.5	109.2	21.5	0.4
131 072	655 360	34 340	436.9	218.5	22.9	0.9
262 144	1 310 720	36 360	873.8	436.9	24.2	1.7
524 288	2 621 440	38 380	1747.6	873.8	25.6	3.5
1 048 576	5 242 880	40 400	3495.3	1747.6	26.9	7.0
2 097 152	10 485 760	42 420	6990.5	3495.3	28.3	14.0
4 194 304	20 971 520	44 440	13 981.0	6990.5	29.6	28.0
8 388 608	41 943 040	46 460	27 962.0	13 981.0	31.0	55.9

Рис. 2.2. Производительность алгоритмов X и Y на разных компьютерах

На небольших наборах данных одинакового размера алгоритму X требуется меньше действий, чем Y, однако, начиная с $N = 8192$, Y становится экономнее X, и чем N больше, тем разница сильнее. На рис. 2.3 видна *точка пересечения* графиков между 4096 и 8192, после которой Y оказывается эффективнее X по количеству выполняемых действий. На двух разных компьютерах идентичные реализации алгоритма X закономерно покажут, что X_{fast}, запущенный на C_{fast}, всегда быстрее X_{slow}, запущенного на C_{slow}.

Допустим, мы нашли суперкомпьютер $C_{fastest}$, который в 500 раз быстрее, чем C_{slow}. Можно отыскать такой объем входных данных, начиная с которого эффективный алгоритм Y будет работать на C_{slow} быстрее, чем неэффективный — на $C_{fastest}$. Мы здесь, конечно, немножко «сравниваем теплое с мягким» — компьютеры-то разные! — но все равно даже в нашем случае точка пересечения возникает между

наборами размером 4 194 304 и 8 388 608. Если алгоритм в конечном счете более эффективен, то, работая даже на медленном компьютере, он обгонит неэффективный алгоритм, запущенный на суперкомпьютере, — когда объем входных данных достаточно велик.

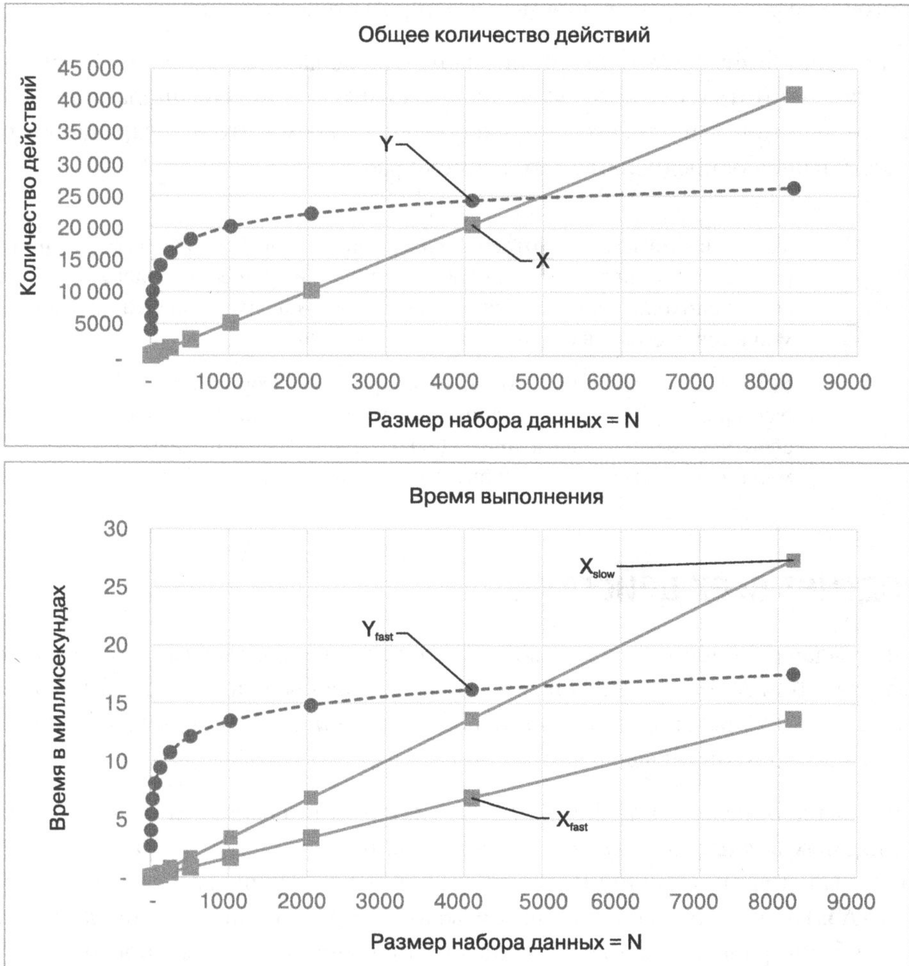
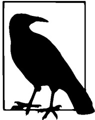


Рис. 2.3. Графики для числовых данных рис. 2.2

Сколько ни трать дополнительных вычислительных мощностей на решение задачи, все равно найдутся такие объемы данных, начиная с которых более эффективный алгоритм работает быстрее.

В математике для классификации вычислительной сложности алгоритмов (и в *худшем*, и в *лучшем* случаях) используется обозначение «О большое». «О» в данном случае означает order, то есть «порядок». Порядок функции — это скорость, с которой она растет при росте ее переменной, N. Например, формула $4N^2 + 3N - 5$ — «порядка N^2» (или квадратичная), потому что быстрее всего в этом многочлене растет первое слагаемое, в котором степень N равна 2.

Для оценки быстродействия алгоритма на наборе данных размером N обычно начинают считать количество выполненных им действий. Предполагается, что каждое действие выполняется за фиксированное время, что и превращает этот подсчет в оценку предполагаемого времени работы.



T(N) — время, которое потребуется алгоритму для обработки набора данных размером N. Для входных данных из лучшего и худшего случаев могут быть свои, непохожие друг на друга T(N). При этом не имеет значения, в чем измеряется время — в секундах или миллисекундах.

S(N) — объем памяти, требуемый для работы алгоритма на наборе данных размером N. Для входных данных из лучшего и худшего случаев могут быть свои, непохожие друг на друга S(N). При этом не имеет значения, в чем измеряется объем памяти — в битах или гигабайтах.

Подсчет всех действий

Наша задача — оценить время работы алгоритма на *произвольном наборе данных* размером N. Поскольку такая оценка должна быть достоверной для всех возможных данных, стоит подобрать набор для *наихудшего случая*, на котором алгоритм работает дольше всего.

Для начала определим функцию $K(N)$ — количество *основных действий*, выполняемых в *наихудшем случае* на наборе данных размером N. Затем оценим число выполняемых машинных инструкций той же функцией с коэффициентом: $c \times K(N)$. Поскольку в современных языках программирования одно действие может быть транслировано в десятки, а то и тысячи инструкций, предосторожность в виде константы c будет нелишней. Вычислять ее не обязательно, хотя подобрать опытным путем с учетом производительности конкретного компьютера, как мы это делали раньше, вполне возможно.

Классификация вычислительной сложности (или потребления памяти) алгоритмов напрямую сопоставляет сложность с некоторой функцией от N. Этой функцией, $f(N)$, описывается класс сложности $O(f(N))$ (поначалу в терминах

легко запутаться). Относя алгоритм к определенному классу, мы составляем формулу в виде функции f от N. Нам уже встречались четыре класса сложности.

1. $O(N)$ — *линейная* сложность, где $f(n) = N$.
2. $O(N^{1.585})$ — сложность Карацубы, где $f(N) = N^{1.585}$.
3. $O(N^2)$ — *квадратичная* сложность, где $f(N) = N^2$.
4. $O(N \log N)$ — *N-логарифмическая* сложность, где $f(N) = N \log N$[1].

Для точного анализа надо изучать исходный текст программы и выявлять организацию алгоритма. Сколько раз выполняется действие `ct += 1` в примере ниже?

```
for i in range(100):
  for j in range(N):
      ct += 1
```

Внешний цикл, по `i`, делает 100 проходов, и на каждом из них внутренний цикл, по `j`, делает N проходов. Итого действие `ct += 1` выполняется $100 \times N$ раз. При должном выборе константы c время выполнения приведенного фрагмента программы $T(N)$ на наборе данных размером N не будет превышать $c \times N$. Запуская программу на конкретном компьютере, можно определить значение соответствующего c. Говоря точнее, в терминах «О большого», сложность этого фрагмента оценивается как $O(N)$.

Можно запускать эту программу тысячи раз на самых разных компьютерах, можно даже каждый раз получать различные значения коэффициента c — порядок сложности алгоритма не изменится, именно это мы и имели в виду, оценивая сложность как $O(N)$. Если не углубляться в дебри теории, достаточно знать вот что: если мы подобрали некоторую функцию $f(N)$, которая оценивает количество действий, выполняемых нашим алгоритмом, тем самым мы автоматически классифицируем сложность этого алгоритма как $O(f(N))$.

Подсчет всех байтов

Такой же анализ можно сделать для оценки «сложности по памяти» — объема памяти, потребляемого алгоритмом на входных наборах размером N. Динамическое выделение дополнительной памяти неизбежно ведет и к понижению

---

[1] Основание логарифма — двоичный логарифм, десятичный, натуральный и т. п. — не имеет значения, так как логарифмы одного числа с разными основаниями отличаются константой, а именно такое отличие допускается оценкой «О большое». — *Примеч. пер.*

производительности, потому что управление памятью — это тоже алгоритм со своей сложностью.

Вот такие конструкции Python требуют совершенно разных объемов памяти.

- `range(N)` занимает константный, не зависящий от N объем оперативной памяти, потому что объект типа `range` — это арифметическая прогрессия, любой элемент которой можно вычислить, а не хранить их все в памяти (когда-то, еще в Python 2, `range()` была просто функцией, возвращавшей список).
- `list(range(N))` сделает из такой последовательности список с элементами от 0 до $N - 1$. Такому объекту требуется оперативная память в объеме, прямо пропорциональном N.

Четко определить объем той или иной конструкции языка программирования непросто уже потому, что не до конца понятно, в чем его измерять. В байтах или битах? Но тогда имеет ли значение, что на одних архитектурах целое число занимает 32 бита, а на других — 64? Допустим, в будущем появится компьютер, оперирующий 128-битными целыми[1], — значит ли это, что сложность по памяти при работе с целыми числами для них удвоится? Размер объекта в байтах в Python показывает функция `sys.getsizeof(объект)`. Давайте на примере убедимся, что использование вычислимой последовательности `range()` в Python 3 позволяет ощутимо сэкономить потребление памяти! Для этого достаточно запустить интерпретатор Python и подать ему такие команды:

```
>>> import sys
>>> sys.getsizeof(range(100))
48
>>> sys.getsizeof(range(10000))
48
>>> sys.getsizeof(list(range(100)))
1008
>>> sys.getsizeof(list(range(1000)))
9112
>>> sys.getsizeof(list(range(10000)))
90112
>>> sys.getsizeof(list(range(100000)))
900112
```

Здесь хорошо видно, что объем `list(range(10000))` в байтах примерно в сто раз больше объема `list(range(100))`. Глядя на остальные значения, можно с уверенностью сказать, что список в Python потребляет порядка O(N) памяти.

[1] Такими «компьютерами будущего» являются, например, игровая консоль Sony PlayStation 2 и большинство графических процессоров. — *Примеч. пер.*

Для сравнения: и `range(100)`, и `range(10000)` занимают один и тот же объем памяти — 48 байт. Это новый для нас класс сложности, в котором значение постоянно и не зависит от N, он так и называется — *константным*.

O(1) — *константная* сложность, где $f(N) = c$ с некоторой постоянной c.

Ну что ж, в этой главе мы рассмотрели довольно много теории; настало время применить наши знания на практике. Пора изучить алгоритм оптимального поиска элемента в упорядоченном списке — по-научному он называется алгоритмом **двоичного поиска** в массивах. Мы выясним, отчего он так эффективен, и попутно познакомимся с новым, *логарифмическим* классом сложности $O(\log N)$.

Одна дверь захлопнулась — другая откроется

За каждой из дверей на рис. 2.4 скрыто по одному произвольному числу. Известно, что числа за дверьми возрастают слева направо. Вопрос: *сколько дверей необходимо в худшем случае открыть* по одной, чтобы найти число 643 — или убедиться, что его за дверьми нет? Можно начать с самой левой и открывать каждую по очереди, пока не появится число 643 или большее (если числа 643 вообще не было за дверьми). Если не повезет, придется открыть все двери. В этом способе никак не учитывается, какие именно числа нашлись за уже открытыми дверьми. В действительности для ответа на вопрос достаточно открыть не более трех дверей. Для начала откроем дверь 4 — среднюю.

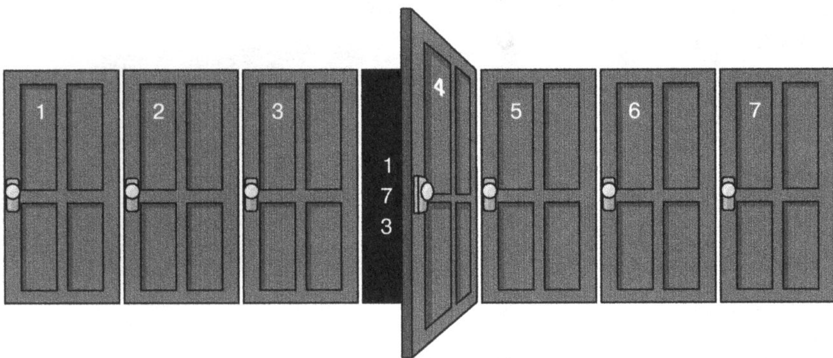


Рис. 2.4. Двери судьбы!

Там окажется число 173. Раз мы ищем 643, все двери слева от четвертой нас больше не интересуют, потому что номера за ними меньше 173. Теперь откроем дверь 6 — за ней число 900. Прекрасно, теперь двери правее шестой нас тоже

не интересуют. Остается только дверь 5 — и либо за ней скрывается 643, либо этого числа вообще не было в исходной последовательности. Пока она закрыта, мы вольны воображать и то и другое.

На любой возрастающей последовательности из семи чисел и для любого искомого числа потребуется открыть не более трех дверей. Кстати, можно заметить, что $2^3 - 1 = 7$. А если возрастающая последовательность будет состоять из миллиона элементов за миллионом дверей и с нами поспорят на 10 000 долларов, что мы не найдем нужное число, открыв только 20 дверей? Надо соглашаться! Ведь $2^{20} - 1 = 1\,048\,575$, так что 20 открытых дверей хватит на то, чтобы проверить возрастающую последовательность из 1 048 575 чисел. Более того, если количество дверей внезапно удвоится (даже дойдет до 2 097 151), понадобится дополнительно открыть всего одну дверь, 21 дверь на все два с лишним миллиона. Удивительно быстрый способ! Вот мы и изобрели алгоритм **двоичного поиска** в упорядоченном массиве.

Двоичный поиск в упорядоченном массиве

Двоичный поиск — один из базовых алгоритмов: порядок его сложности отличается от порядка сложности уже рассмотренных нами алгоритмов. В примере 2.3 реализован двоичный поиск значения `target` в упорядоченном списке `A`.

Пример 2.3. Двоичный поиск

```
def binary_array_search(A, target):
  lo = 0
  hi = len(A) - 1             ❶

  while lo <= hi:             ❷
    mid = (lo + hi) // 2      ❸

    if target < A[mid]:       ❹
      hi = mid-1
    elif target > A[mid]:     ❺
      lo = mid+1
    else:
      return True             ❻

  return False                ❼
```

❶ Будем вести поиск в списке от индекса `lo` до индекса `hi` включительно. Поначалу они равны `0` и `len(A)-1` соответственно.

❷ Цикл закончится, когда диапазон опустеет.

❸ В диапазоне `A[lo ... hi]` найдем середину и значение `A[mid]` в ней.

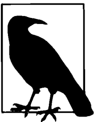
❹ Если `target` меньше `A[mid]`, продолжим поиск *слева* от середины.

❺ Если `target` больше `A[mid]`, продолжим поиск *справа* от середины.

❻ Если это и есть `target`, вернем `True`.

❼ Если `lo` превысило `hi`, значит, искать больше негде. Вернем `False` в знак того, что `target` не найден в `A`.

Поначалу `lo` и `hi` равны наименьшему и наибольшему индексу в `A`. Пока диапазон поиска не исчерпан, с помощью целочисленного деления находим его середину, `mid`. Если `A[mid]` и есть `target`, поиск окончен; иначе становится понятно, сужать ли диапазон до левой части — `A[lo...mid-1]` или до правой — `A[mid+1...hi]`.



Запись A[lo ... mid] в книге означает диапазон значений из списка A, начиная с индекса lo до индекса mid включительно. Если lo < mid, диапазон пуст.

Наша функция определяет, принадлежит ли некоторое значение упорядоченному списку из N элементов. На каждом обороте цикла можно либо отыскать нужный элемент, либо сузить диапазон. Цикл заканчивается, когда элемент найден или когда `hi` оказывается меньше `lo`.

Немногим сложнее, чем π

Рассмотрим **двоичный поиск** числа 53 в списке на рис. 2.5. Переменные `hi` и `lo` поначалу совпадают с границами списка `A`. В цикле `while` вычисляется `mid`. Поскольку `A[mid]` равно 19 и меньше 53, выполнение идет по ветке `elif` и значение `lo` меняется на `mid + 1`. Дальнейший поиск производится уже по диапазону `A[mid+1 ... hi]`. Серым закрашены нерассматриваемые значения. Размер диапазона уменьшается вполовину (с 7 до 3).

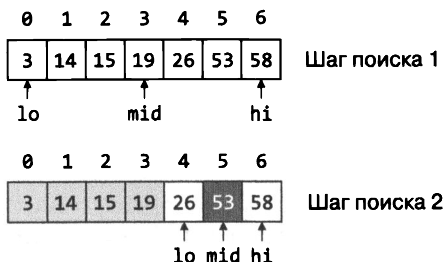
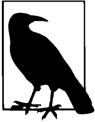


Рис. 2.5. Поиск числа 53

На втором проходе цикла `while` вычисляется новое `mid` и оказывается, что `A[mid]` равно искомому числу, 53, так что функция возвращает `True`.



Имеет ли смысл перед запуском двоичного поиска проверить, точно ли A[0] <= target <= A[-1]? Так мы могли бы избежать бессмысленного поиска числа, которое в принципе не может оказаться в списке. Говоря коротко: нет, не имеет. Это добавит целых две проверки (которых даже на 1 000 000 всего 20) к каждому поиску, а сами проверки окажутся лишними в поиске чисел, которые заведомо принадлежат диапазону значений A.

Теперь поищем число 17, которого нет в списке. Как показано на рис. 2.6, `lo` и `hi` снова установлены по границам `A`. `A[mid]` равно 19; это больше, чем 17, так что вычисление идет по ветке `if`, и диапазон сужается до `A[lo … mid-1]`. Серым закрашены значения, которые нам больше не нужны. `A[mid]` теперь равно 14, это меньше 17, и вычисление идет по ветке `elif`, а диапазон сужается до `A[mid+1 … hi]`.

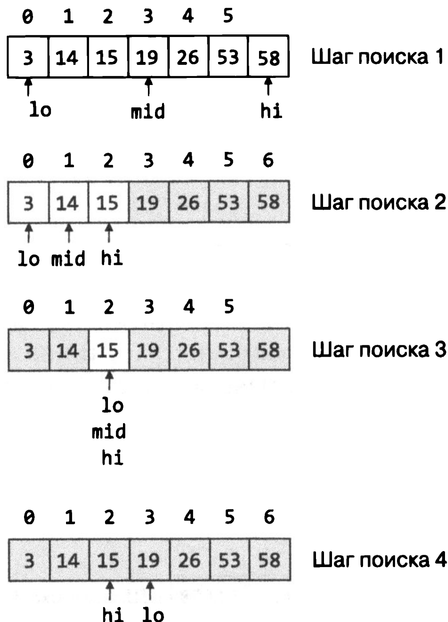


Рис. 2.6. Поиск числа 17

На третьем проходе цикла `while` получаем `A[mid]` равным 15; это меньше, чем искомое 17. Снова идем по ветке `elif`, где `lo` делается больше, чем `hi`: диапазон «схлопывается», как показано в конце рис. 2.6. Условие цикла `while` становится ложным, и возвращается `False`: это означает, что `target` в `A` не найден.

Двух зайцев одним выстрелом

Зачастую требуется найти, где именно в A находится target, а не только проверить, есть ли он там. Пока наша функция возвращала только True или False. Слегка подправим исходный текст, и **двоичный поиск** будет возвращать позицию mid, где находится target (пример 2.4).

Пример 2.4. Вычисление позиции target в A

```
def binary_array_search(A, target):
  lo = 0
  hi = len(A) - 1

  while lo <= hi:
    mid = (lo + hi) // 2

    if target < A[mid]:
      hi = mid-1
    elif target > A[mid]:
      lo = mid+1
    else:
      return mid               ❶

  return -(lo+1)               ❷
```

❶ Возвращаем mid, потому что именно там мы нашли target.

❷ Уведомляем пользователя о том, что target не найден, возвращая отрицательный индекс -lo - 1.

Что бы такого полезного вернуть, если target нет в A? Можно просто None, как это принято в Python, но в нашем случае есть шанс дать пользователю дополнительные сведения. Например, можно не только сообщить о том, что target отсутствует в A, но и заранее вычислить *место*, куда его следует при необходимости вставить, не нарушая порядка A.

Давайте еще раз посмотрим на рис. 2.6. Пока мы искали число 17 в списке, где его не было, значение lo стало указывать на место, куда 17 *можно было бы вставить*, соблюдая порядок в A. Можно было бы в этом случае вернуть просто -lo, но если искомый элемент найден в *начальной позиции* или не найден и это только место вставки, в обоих случаях вернется 0. Так что возвращаем заведомо отрицательное число -(lo + 1). Если пользователь получит отрицательное x, он будет знать, что target отсутствует в A, но его можно туда добавить в позицию -(x + 1). Если x неотрицателен — это позиция target в A.

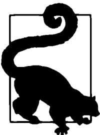
Напоследок еще немного оптимизации. В примере 2.4 над элементами A производится два сравнения, хотя можно было бы обойтись одним, а потом проверять

его результаты. Если значения числовые, таким сравнением может быть их разность — как в примере 2.5, где над элементами `A` производится *всего одно* действие (а не два). Кстати, операция индексирования `A[mid]` в примере тоже одна.

Пример 2.5. Оптимизация, которая требует только одного сравнения элементов списка

```
diff = target - A[mid]
if diff < 0:
  hi = mid-1
elif diff > 0:
  lo = mid+1
else:
  return mid
```

Если `target` был меньше `A[mid]`, `diff` окажется отрицательным, и это аналогично сравнению `target < A[mid]`. Если `diff` положителен, значит, `target` оказался больше `A[mid]`. Допустим, значения `A` — не числовые, но для них существует операция сравнения, которая возвращает отрицательное число, ноль или положительное число, в соответствии с тем, как эти значения можно упорядочить. Сравнение обычных целых вместо дорогостоящего сравнения элементов увеличивает быстродействие.



Если список упорядочен по убыванию, **двоичный поиск** тоже работает — только hi и lo в цикле while должны меняться по-другому.

Как быстро работает **двоичный поиск** на данных объемом *N*? Чтобы ответить на вопрос, надо привести *наихудший случай* входных данных и посчитать, сколько раз должен выполниться цикл `while`. Понятие логарифма нам в помощь!

Что такое логарифм? Попробуем ответить на вопрос: сколько раз нужно удвоить единицу, чтобы получить число 33 554 432? Можно, конечно, попробовать добраться до него вручную — 1, 2, 4, 8, 16 и т. д., — но это довольно утомительно. Что мы ищем с математической точки зрения? Такое x, что $2^x = 33\,554\,432$.

Запись 2^x — это возведение основания 2 в степень x. Логарифм — функция, обратная возведению в степень, примерно как деление — функция, обратная умножению. Чтобы найти такое x, что $2^x = 33\,554\,432$, надо посчитать логарифм по основанию 2 от этого числа, $\log_2 33554432$. Получится 25.0. Если вычислить 2^{25} на калькуляторе, получится 33 554 432.

То же значение — ответ на вопрос «Как долго можно делить 33 554 432 пополам?»: после 25 делений получится 1. Логарифм — вещественное число; напри-

мер, $\log_2 137$ — примерно 7.098032. Что в целом понятно: раз уж $2^7 = 128$, то для получения 137 степень нужна чуть-чуть побольше.

В алгоритме **двоичного поиска** цикл `while` повторяется до тех пор, пока `lo` ≤ `hi`, то есть пока есть где искать. На первом проходе в диапазоне поиска N значений, на втором их уже $N / 2$ (а если N нечетно, то $(N - 1) / 2$). Следовательно, чтобы предсказать количество проходов цикла, надо посчитать, сколько раз придется делить N пополам, пока не получится 1. А это и есть в точности `k` = $\log_2 N$, что дает нам 1 + `k` проходов цикла (первый — для полного N, остальные `k` — для меньших диапазонов). Логарифм — вещественное число, а нам для оценки количества проходов, конечно, нужно целое. Подойдет нижнее округление `floor(x)`, математическая функция, возвращающая наибольшее целое число, не превосходящее x[1].

Далеко не каждый калькулятор или приложение-калькулятор умеют вычислять двоичный логарифм. Но у нас же есть Python! Лучший в мире программируемый калькулятор всегда у вас под рукой. Достаточно запустить любую версию командного интерпретатора — стандартный Python, консоль любого средства разработки (например, IDLE — она есть в базовой поставке Python) или специализированный терминал наподобие iPython — и импортировать требуемую библиотеку. В ответ на подсказку >>> вводим нужную формулу — и вуаля, ответ перед нами.

```
>>> from math import *
>>> log2(16)

>>> log(16)

>>> log10(16)

>>> log(16)/log(2)

>>> log10(16)/log10(2)
```

В примере иллюстрируется важное свойство: логарифм одного и того же числа по разным основаниям отличается только множителем. Скажем, чтобы получить логарифм числа 16 по основанию 2, можно воспользоваться натуральным логарифмом (по основанию $e \approx 2.718281828459045...$) числа 16 и натуральным логарифмом самого основания 2. Функция log(x) в Python вычисляет натуральный логарифм, $\log_2(x)$ — двоичный, а $\log_{10}(x)$ — логарифм по основанию 10.

---

[1] *Наихудший случай* для данной реализации алгоритма — поиск числа, которое заведомо больше всех 2^N элементов массива. Именно в этом варианте каждый диапазон поиска в цикле будет четной длины. Если оказалось, что хотя бы один из диапазонов — нечетной длины, это сэкономит один проход. Поэтому для 2^{N-1} элементов (в которых такая ситуация невозможна) оценка уже будет меньше. — *Примеч. пер.*

Итак, в **двоичном поиске** цикл `while` делает не более `floor`($\log_2 N$) + 1 проходов. Это невероятно быстро! Найти нужное число из миллиона отсортированных значений можно всего за 20 попыток.

Мы можем сами убедиться в корректности формулы: для любого объема входных данных от 8 до 15 элементов требуется не более четырех проходов. Если на первом проходе в диапазоне 15 элементов, то на втором их будет 7, на третьем — 3 и на четвертом останется только один. Если начать с десяти элементов, диапазон будет меняться так: 10 → 5 → 2 → 1 — и это тоже четыре прохода.

Получается, что изучение **двоичного поиска** дало нам новый класс сложности, O($\log N$), он называется *логарифмическим*, потому что определяющая его функция $f(N) = \log N$.

Стало быть, если при изучении алгоритма мы утверждаем, что его сложность по времени — O($\log N$), это означает, что существует такая константа c, для которой, начиная с некоторого достаточно большого N, скорость работы алгоритма $T(N)$ будет всегда меньше, чем $c \times \log N$. Наше утверждение верно, *если неверны аналогичные утверждения* относительно более низких классов вычислительной сложности (рис. 2.7).

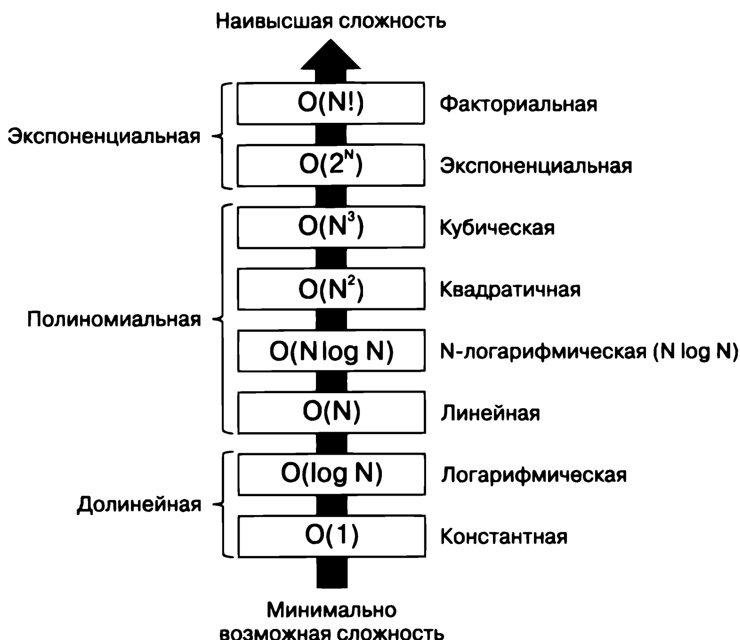


Рис. 2.7. Иерархия классов вычислительной сложности в порядке доминирования

Классов вычислительной сложности в теории бесконечно много, но на практике эти восемь встречаются чаще всего. Константное время выполнения O(1) — это низший класс сложности, при этом время работы алгоритма постоянно и не зависит от размера входных данных. Следующий класс в иерархии — O(log N), логарифмический, мы уже видели, что **двоичный поиск** относится к этой категории. Оба класса — *долинейные*, они содержат чрезвычайно эффективные алгоритмы.

Линейная сложность, O(N), предусматривает увеличение времени работы прямо пропорционально объему данных. Далее идет набор полиномиальных классов в порядке возрастания сложности — O(N^2), O(N^3) и т. д. вплоть до любой наперед взятой константы c, O(N^c)[1]. Между O(N) и O(N^2) затесался класс O($N \log N$); для многих задач именно алгоритм такого класса специалисты считают наилучшим.

Доминирование классов друг над другом выражается еще в том, как определяется алгоритм, в оценке сложности которого встречаются несколько различных компонентов-формул. Например, если сложность одной части алгоритма оценивается как O($N \log N$), а другой — как O(N^2), то какова сложность всего алгоритма? Ответ: O(N^2), потому что сложность второй части *доминирует* над сложностью первой. Если, например, оценка сложности вашего алгоритма $T(N)$ оказалась равной $5N^2 + 10\,000\,000 \times N \times \log_2 N$, то класс его сложности — O(N^2).

Два оставшихся класса в иерархии — экспоненциальный и факториальный — описывают весьма неэффективные алгоритмы, которые могут работать только с очень небольшим числом данных. Тренировочное задание в конце этой главы посвящено в том числе этим классам сложности.

Как все это работает

В табл. 2.4 представлены результаты вычислений $f(N)$ для каждого упомянутого класса сложности. Предположим, что числа в таблице — это время работы некоторого алгоритма в секундах. Сложность алгоритма указана в заголовке столбца, а размер входного набора данных N — в заголовке строки. Значение 4096 минут — это 1 час 8 минут, так что чуть больше чем за час можно решить:

- любую задачу сложности O(1), потому что размер входных данных в ней не имеет значения;
- задачу сложности O(log N) на входном наборе размером не более чем 2^{4096};
- задачу сложности O(N) на входном наборе размером не более чем 4096;

[1] Иными словами, какую бы константу c мы заранее ни взяли, всегда найдется достаточно большой объем данных, на котором наш полиномиальный алгоритм степени c эффективнее данного алгоритма более высокого класса сложности. А вот утверждать то же самое про весь бесконечный ряд {O(N^k)} нельзя. — *Примеч. пер.*

- задачу сложности O($N \log N$) на входном наборе размером не более чем 462;
- задачу сложности O(N^2) на входном наборе размером не более чем 64;
- задачу сложности O(N^3) на входном наборе размером не более чем 16;
- задачу сложности O(2^N) на входном наборе размером не более чем 12;
- задачу сложности O($N!$) на входном наборе размером не более чем 7.

Таблица 2.4. Рост различных оценок сложности

N	log N	N	N log N	N	N	2	N!
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40 320
16	4	16	64	256	4096	65 536	2.1×10^{13}
32	5	32	160	1024	32 768	4.3×10^{9}	2.6×10^{35}
64	6	64	384	4096	262 114	1.8×10^{19}	1.3×10^{89}
128	7	128	896	16 384	2 097 152	3.4×10^{38}	∞
256	8	256	2048	65 536	16 777 216	1.2×10^{77}	∞
512	9	512	4608	262 144	1.3×10^{8}	∞	∞
1024	10	1024	10 240	1 048 576	1.1×10^{9}	∞	∞
2048	11	2048	22 528	4 194 304	8.6×10^{9}	∞	∞

Нужно стараться отыскать алгоритм с самым низким классом сложности, потому что, как это показано на рис. 2.8, всегда может попасться задачка, в которой и для самого быстрого компьютера данных слишком много. А алгоритмы высокого уровня сложности даже на малых объемах данных могут работать практически бесконечно.

Быстродействие различных классов на больших объемах данных выглядит обычно так, как на рис. 2.8. Ось X отвечает за размер входного набора, ось Y — за полное время работы алгоритма, классы алгоритмов надписаны на графиках. Чем выше класс сложности алгоритма, тем меньше объем данных, которые он сможет обработать за «приемлемое время»[1].

[1] Графики наивысших классов сложности — экспоненциального и факториального — на иллюстрации отсутствуют, потому что в таком масштабе они практически совпадают с осью X — сразу уходят в бесконечность. — *Примеч. пер.*

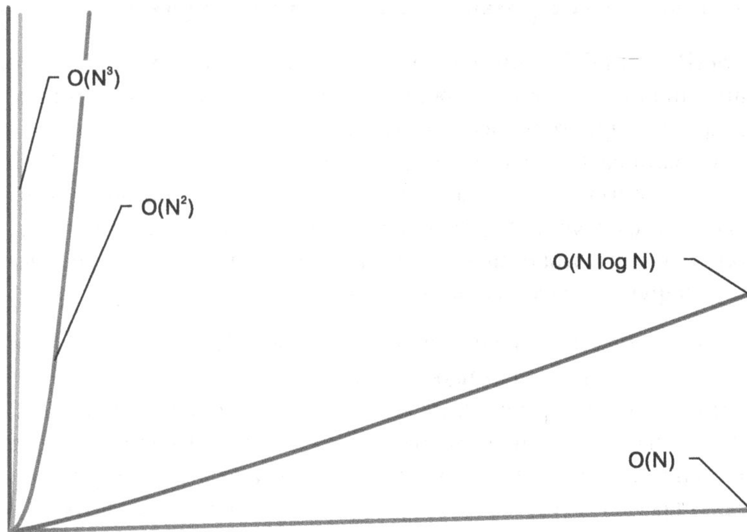


Рис. 2.8. График быстродействия разных классов сложности на возрастающем объеме входных данных

Рассмотрим еще несколько примеров посложнее.

- Если сложность алгоритма оценивается как $O(N^2 + N)$ — к какому классу его отнести? Согласно иерархии на рис. 2.7 N^2 доминирует над N, и, следовательно, оценку можно упростить до $O(N^2)$. По тем же соображениям $O(2^N + N^8)$ упрощается до $O(2^N)$.
- Если сложность оценивается как $O(50 \times N^3)$, оценка упрощается до $O(N^3)$, потому что мультипликативная постоянная значения не имеет.
- Иногда быстродействие алгоритма зависит не только от размера входных данных N, но и от их свойств. Скажем, имеется алгоритм, на первом шаге которого N числовых значений отрабатывается за линейное время (то есть за время, прямо пропорциональное N). На втором шаге этого алгоритма из входных данных обрабатываются только *четные числа* и время этой обработки пропорционально E^2 (где E — количество четных чисел во входном наборе). Если подходить к вопросу аккуратно, оценка получится $O(N + E^2)$. Допустим, четные числа не очень-то и нужны и для решения задачи достаточно входного набора без них. Тогда оценка упадет до $O(N)$, что может быть весьма полезно. Понятное дело, *худший случай* для такого алгоритма — это набор исключительно четных чисел, так что общая оценка будет $O(N^2)$, поскольку $E \leq N$.

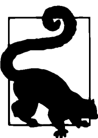
Приближенная кривая или четкие границы?

В пакете SciPy есть функция `curve_fit()`, которая с помощью нелинейного метода наименьших квадратов может подобрать коэффициенты для некоторой модельной функции f, чтобы та как можно лучше описывала имеющиеся данные. Мы использовали ее, когда для измеренного на разных объемах данных N времени работы алгоритма прикидывали, какой функцией оно может выражаться. Как мы уже знаем, `curve_fit()` возвращает коэффициенты, подставив которые в f мы получим приближение с *минимальной суммой квадратов отклонений* значений этой функции от имеющихся данных.

Таким способом можно довольно точно угадать, как будет вести себя конкретная реализация алгоритма на других наборах данных в исследованном нами диапазоне. Однако само по себе наше приближение не может выступать ни в роли верхней, ни в роли нижней границы оценки сложности. Для точной оценки надо определить количество действий, которые выполняет данная реализация алгоритма, — именно оно управляет временем работы — и на этом основании уже делать предположения.

Построив функцию $f(N)$, которая оценивает количество выполняемых действий на наборе данных размером N в *худшем случае*, мы можем оценить сложность алгоритма как $O(f(N()))$ в *худшем случае* — и это будет верхняя граница. Соответствующую нижнюю границу можно вывести из функции $g(N)$, оценивающей минимальное возможное количество действий в *лучшем случае*, — эта граница обозначается как $\Omega(g(N))$.

В разговоре про **двоичный поиск** мы выяснили, что цикл `while` в функции делает не более `floor`($\log_2$ N) + 1 проходов. Это значит, что оценка в *худшем случае* $f(N) = \log N$ и порядок сложности **двоичного поиска** — логарифмический, $O(\log N)$. Как **двоичный поиск** работает в *лучшем случае*? Если искать элемент, равный `A[mid]`, функция завершится всего за один проход цикла. Этот результат *постоянен и не зависит от размера входных данных*, значит, порядок **двоичного поиска** в *лучшем случае* — константный, $O(1)$. Обозначение «O большое» можно применять и для *лучших*, и для *худших случаев*, хотя, с точки зрения большинства программистов, это имеет смысл только для *худших*.



Для классификации сложности алгоритма нередко применяют обозначение вида $\Theta(N \log N)$ (с большой греческой буквой тета). Хотя чаще всего оно применяется для оценки *сложности алгоритма в среднем*, его смысл более узкий. Такая запись означает, что и верхняя, и нижняя границы оценки определяются одной и той же формулой, в данном случае верхняя граница — $O(N \log N)$, а нижняя — $\Omega(N \log N)$. По-английски этот термин звучит как tight bound — «строгая оценка». Строгая оценка производительности алгоритма, если она есть, лучше всего позволяет убедиться, что время его работы хорошо предсказуемо.

Заключение

В первых двух главах мы довольно далеко продвинулись в изучении алгоритмов, но непройденных дорог намного больше. Вы видели достаточно примеров того, что быстродействие алгоритма *не зависит от способа его реализации*. Пока ученые середины XX века изобретали новые алгоритмы, инженеры успешно повышали быстродействие самих компьютеров, на которых эти алгоритмы работают. Чтобы при изучении быстродействия алгоритмов не зависеть от производительности аппаратной платформы и вывести некоторые абсолютные оценки, мы применили основы асимптотического анализа. Мы рассмотрели несколько классов сложности (по времени или по памяти), которые приведены на рис. 2.8, и с их помощью вы научились показывать, как производительность алгоритма зависит от объема обрабатываемых данных. Классы сложности нам понадобятся и впредь, с их помощью можно коротко описать быстродействие алгоритма.

Тренировочные задания

1. Оцените сложность по времени каждого из фрагментов программы в примере 2.6.

Пример 2.6. Оцените фрагменты программ

Фрагмент 1

```
for i in range(100):
  for j in range(N):
    for k in range(10000):
      ...
```

Фрагмент 2

```
for i in range(N):
  for j in range(N):
    for k in range(100):
      ...
```

Фрагмент 3

```
for i in range(0,N,2):
  for j in range(0,N,2):
    ...
```

Фрагмент 4

```
while N > 1:
  ...
  N = N // 2
```

Фрагмент 5

```
for i in range(2,N,3):
  for j in range(3,N,2):
    ...
```

2. Используя методику анализа, описанную в этой главе, оцените значение `ct`, возвращаемое функцией `f4` в примере 2.7.

Пример 2.7. Исследуйте функцию

```
def f4(N):
  ct = 1
  while N >= 2:
    ct = ct + 1
    N = N ** 0.5
  return ct
```

Ни одну из рассмотренных в главе оценок применить не получится. Попробуйте взять за основу формулу $a \times \log_2(\log_2 N)$. Составьте таблицу значений функции вплоть до $N = 2^{50}$ и сравните с вашей оценкой. Полученный класс сложности можно определить как $O(\log(\log N))$.

3. Есть один неэффективный способ сортировки: перебрать все перестановки исходного массива и вернуть ту, которая окажется отсортированной. Этот способ показан в примере 2.8.

Пример 2.8. Сортировка путем перебора всех перестановок списка

```
from itertools import permutations, pairwise
from scipy.special import factorial

def factorial_model(v, a):
  return a * factorial(v)

def check_sorted(a):
  for x, y in pairwise(a):
    if x > y:
      return False
  return True

def permutation_sort(A):
  for attempt in permutations(A):
    if check_sorted(attempt):
      A[:] = attempt          # заполним A отсортированными значениями
      return
```

Составьте таблицу быстродействия функции `permutation_sort()` в *наихудшем случае* (здесь это список, упорядоченный по убыванию); размер списка

от 1 до 12 элементов включительно. Подберите приближенную кривую с помощью функции `factorial_model()` и исследуйте, насколько хорошо она прогнозирует время работы `permutation_sort()`. Сколько *лет*, если верить полученным данным, эта функция будет сортировать упорядоченный по убыванию список из 20 элементов?[1]

4. Соберите данные по быстродействию 50 000 экспериментальных запусков **двоичного поиска** на случайных наборах размером N от 2^5 до 2^{21}. Каждый эксперимент должен использовать случайный входной набор размером N, получаемый с помощью `random.sample()` из диапазона чисел от 0 до $4 \times N$, который затем надо отсортировать. Искать в каждом эксперименте нужно случайное число из того же диапазона.

   Используя методику, описанную в данной главе, подберите с помощью `curve_fit()` логарифмическое ($\log N$) приближение к полученным данным для N в диапазоне от 2^5 до 2^{12}. Определите **пороговое значение** N, начиная с которого результаты экспериментов предсказуемо описываются найденным приближением. Нарисуйте график функции-приближения и точки экспериментальных данных. Насколько достоверно найденная функция описывает эксперимент?

5. В отличие от предыдущих задач будем исследовать сложность по памяти. В примере 2.9 рассмотрим еще один странный алгоритм сортировки списка.

Пример 2.9. Сортировка путем постоянного удаления наибольшего значения из списка

```
def max_sort(A):
  result = []
  while len(A) > 1:
    index_max = max(range(len(A)), key=A.__getitem__)
    result.insert(0, A[index_max])
    A = list(A[:index_max]) + list(A[index_max+1:])
  return A + result
```

Используя методику из этой главы, оцените сложность по памяти функции `max_sort()`.

---

[1] Кое-что в примере нуждается в объяснении. Во-первых, itertools.permutations(A) возвращает в цикл for все перестановки A по одной, не храня их в памяти. Во-вторых, обычный factorial() из библиотеки math() нельзя применить к массиву v, поэтому в factorial_model() используется специальная векторная функция scipy.special.factorial(v), которая возвращает массив факториалов элементов v. В-третьих, itertools.pairwise(A) возвращает в цикл for все соседние пары элементов в последовательности A, сначала нулевой элемент и первый, затем первый и второй и т. д. — *Примеч. пер.*

6. *Галактический алгоритм* — это алгоритм решения некой задачи, который становится эффективнее других известных алгоритмов только на «недостижимо большом» наборе данных. Например, алгоритм умножения N-значных чисел, опубликованный в ноябре 2020 года Дэвидом Харви и Йорисом ван дер Хувеном, имеет сложность $O(N \log N)$, но только для N, превышающих 2^Z, где $Z = 1729^{12}$. Это Z — само по себе астрономическое число, примерно 7×10^{38}, а уж если в такую степень возвести 2, получится уже что-то запредельно огромное! Почитайте, что пишут о галактических алгоритмах. На практике их использовать нельзя, но зачастую осознание того, что для данной задачи *может* существовать алгоритм более низкого класса сложности, подталкивает к дальнейшим открытиям!

7. В табл. 2.1 приведены три ряда измерения для трех размеров набора входных данных. Если бы рядов было только два, удалось бы составить приближение для квадратичного алгоритма? Более общий вопрос: если имеется K замеров производительности, какова наибольшая степень многочлена, которым можно приблизить эти результаты?

Хороший хеш — залог успеха

https://t.me/it_boooks/2

В этой главе

- Как сформировать *ассоциативный массив*, хранящий пары вида (`ключ, значение`), и по известному ключу получать такую пару из него.
- Как хранить пары (`ключ, значение`) в массиве, чтобы при этом поиск по ключу был эффективен, при условии, что размер этого массива достаточно велик по сравнению с количеством хранимых пар.
- Как использовать массив списков для хранения и поиска пар (`ключ, значение`) с возможностью удаления по ключу.
- Как менять размер хеш-таблицы, чтобы не потерять ее производительность.
- Как оценивать *среднее* быстродействие алгоритма, если его работа меняется по мере выполнения некоторых действий с данными (как делать *амортизационный анализ*).
- Как довести оценку операции добавления в хеш-таблицу `put()` до амортизационной сложности O(1) в среднем путем *геометрического масштабирования.*
- Как свойство равномерного распределения значений *хеш-функции* можно использовать для эффективного размещения ключей в хеш-таблице.

Соответствие значений ключам

Часто набор объектов, которые нужно уметь хранить в памяти, представлен не в виде последовательности, а в виде множества пар (`ключ, значение`), в котором каждому ключу однозначно соответствует некоторое значение. Такая структура называется *ассоциативным массивом.* Если наложить дополнительные

требования на свойство ключа, вместо поиска пары (*ключ, значение*) по всему набору можно использовать гораздо более эффективный прием — хеширование. Поиск по хешу работает быстрее любого изученного нами алгоритма поиска! Ассоциативный массив сохраняет эффективность, даже если разрешить удаление ключей и значений. Порядок следования ключей в такой структуре определяется ее реализацией (в словарях Python ключи идут в порядке *добавления*, в устаревшем Python2 определенный порядок не гарантировался вовсе; в любом случае ключи почти наверняка не будут упорядочены по возрастанию, как индексы массива). Взамен мы получаем великолепную скорость поиска и добавления пары.

Скажем, мы хотим написать функцию `print_month(month, year)`, которая будет выводить календарь на определенный месяц определенного года. С параметрами `print_month('February', 2024)` функция должна выводить примерно это:

```
    Февраль 2024
Пн Вт Ср Чт Пт Сб Вс
          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 28 29
```

Что нам нужно для этого знать? День недели, с которого в этом году начинается месяц (в примере это четверг), а еще — сколько дней в феврале этого года (28, а если год високосный, как 2024-й, то 29). Для количества дней в месяце заведем числовой массив, каждый элемент которого будет соответствовать месяцу, начиная с января:

```
month_length = 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

Итак, в январе 31 день, поэтому `month_length[0] == 31`, в феврале — 28, так что следующее значение — `28` и т. д. до последнего `31`, которое соответствует 31 дню в декабре.

Изученные нами ранее примеры позволяют предположить, что понадобится также массив той же длины с названиями месяцев — ключей, по которым мы сможем определить, по какому индексу в `month_length` находится искомое количество дней. Вот такой фрагмент программы выводит количество дней в феврале:

```
key_array = ('Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
             'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь')

idx = key_array.index('Февраль')
print('В феврале', month_length[idx], 'дней')
```

Если название месяца в массиве ключей есть, этот фрагмент будет работать. Однако в *худшем случае*, когда нам нужен `'Декабрь'`, придется просмотреть все значения. Если ключа в массиве нет, `key_array.index()` выдаст исключение, так что предварительно нам придется проверить наличие ключа, например, с помощью `key in key_array`. Это тоже *худший случай*, потому что будут просмотрены все строки в `key_array`. Следовательно, время поиска значения по ключу в изобретенной нами структуре *прямо пропорционально количеству ключей*. Такая скорость поиска довольно быстро станет медленной до полной неприемлемости. Несмотря на это, давайте все-таки напишем `print_month()`. В частности, в примере 3.1 приведена возможная реализация, в которой задействованы два стандартных модуля Python — `datetime` (для определения дня недели) и `calendar` (для определения високосного года).

Пример 3.1. Вывод на экран календаря за произвольный год и месяц в удобном формате

```
from datetime import date
import calendar

month_length = 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
key_array = ('Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
             'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь')

def print_month(month, year):
  idx = key_array.index(month)                ❶
  wd = date(year, idx + 1, 1).weekday()       ❷
  days = month_length[idx]                    ❸
  if calendar.isleap(year) and idx == 1:      ❹
   days += 1

  print(f"{month} {year}".center(20))
  print("Пн Вт Ср Чт Пт Сб Вс")
  print('   ' * wd, end='')                   ❺
  for day in range(days):
    wd = (wd + 1) % 7                         ❻
    eow = " " if wd % 7 else "\n"             ❼
    print(f"{day+1:2}", end=eow)
  print()
```

❶ Найдем номер месяца (целое число от 0 до 11) — индекс в `month_length`.

❷ Вычислим первый день недели данного месяца в данном году, 0 — это понедельник. В функцию `date()` передается `idx + 1`, потому что месяцы в ней нумеруются с единицы, а не с нуля.

❸ Определим количество дней в месяце.

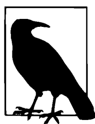
❹ В високосном году в феврале 29 дней (мы нумеруем месяцы с нуля, так что февраль – это 1).

❺ Вместо дней прошедшего месяца в первой неделе выведем пробелы, тогда первый день окажется на своем месте.

❻ Вычислим следующий день недели.

❼ Вычислим, что выводить после текущей даты. Если следующий день не понедельник, то есть `wd` не делится на 7, нужно выводить пробел, а если понедельник — конец строки, потому что неделя закончилась.

Чтобы оценить производительность поиска ключа в наборе из N пар (*ключ, значение*), имеет смысл посчитать количество операций индексирования. Функция `key_array.index()` для поиска нужной строки может просматривать вплоть до N элементов, так что ее сложность – $O(N)$.



Тип данных list реализован в Python как динамический массив, размер которого может меняться. Для краткости его часто называют просто списком. В этой главе мы также будем использовать термин «массив» в знак того, что наши методы применимы и к классическим массивам неизменяемой длины. Кроме того, если мы не собираемся менять и сами элементы массива, в программах на Python лучше использовать кортежи (тип данных tuple), как в примерах выше.

Ассоциативные массивы, хранящие произвольные значения по ключу, реализованы в Python специальным типом данных `dict` (от dictionary – «словарь»). Мы можем составить словарь `days_in_month`, хранящий целые числа – продолжительность месяцев в днях в соответствии с ключами – названиями месяцев (с заглавной буквы):

```
days_in_month = {'Январь':  31, 'Февраль': 28, 'Март':     31,
                 'Апрель':  30, 'Май':     31, 'Июнь':     30,
                 'Июль':    31, 'Август':  31, 'Сентябрь': 30,
                 'Октябрь': 31, 'Ноябрь':  30, 'Декабрь':  31 }
```

А вот пример вывода утверждения «в апреле 30 дней»:

```
print('В апреле', days_in_month['Апрель'], 'дней')
```

В словаре на поиск ключа в среднем уходит $O(1)$ времени, независимо от количества хранящихся в нем пар (*ключ, значение*). Совершенно удивительный результат, будто фокусник только что достал кролика из шляпы! Словарями мы воспользуемся в восьмой главе, а пока попробуем сами достать этого кролика. Пример ниже предлагает интуитивное описание математического механизма,

который мы изучаем в этой главе. Смысл в том, чтобы для начала преобразовать строку в число.

Будем считать, что буква «а» — это 0, буква «б» — 1 и т. д. до «я» — 31. С помощью этих 32 «цифр» любое русское слово (увы, без буквы «е»), записанное строчными буквами, соответствует некоторому числу в системе счисления с основанием 32. Например, в «числе» «июнь» четыре цифры — «и», «ю», «н» и «ь». Если начать переводить из тридцатидвоичной системы счисления в привычную нам десятичную, получится «и» × 32_3 + «ю» × 32_2 + «н» × 32_1 + «ь» × 32_0, или, после вынесения основания за скобки, 32 × (32 × (32 × «и» + «ю») + «н») + «ь». Именно так и работает функция `base32()` в примере 3.2[1].

Пример 3.2. Интерпретация слова как числа с основанием 32

```
def base32(w):
  val = 0
  for ch in w.lower():                ❶
    next_digit = ord(ch) - ord('a')   ❷
    val = 32 * val + next_digit       ❸
  return val
```

❶ Преобразуем буквы строки в строчные.

❷ Вычислим значение очередной цифры.

❸ Припишем цифру в конец результата.

В `base32()` используется функция `ord()`, которая возвращает порядковый номер буквы в таблице всех известных Python символов Unicode[2]. Буквы от «а» до «я» идут в алфавитном порядке[3], `ord('a') == 1072`, `ord('б') == 1073` и т. д. до `ord('я') == 1103`. Так что для того, чтобы вычислить значение «и», достаточно вычесть порядковые номера: `ord('и') - ord('a')` — и получить `8`.

Значение `base32()` очень быстро растет с ростом длины строки. Уже `base32('Июнь')` возвращает `293308`, а `base32('Сентябрь')` — так и вовсе `589940360732`.

---

[1] Буква «а» в примере — русская. — *Примеч. пер.*

[2] В авторском тексте использовались английские буквы и речь шла про кодировку ASCII. Поскольку ord() в действительности работает не с ASCII, а именно с Unicode, мы решили, что перевод примеров на русский не только приблизит их к читателю, но и сгладит неоднозначность. Разумеется, строчные буквы в Unicode отличаются от прописных! Например, ord('я') == 1103, а ord('Я') == 1071. — *Примеч. пер.*

[3] А вот «е» не повезло: ее порядковый номер на 2 меньше, чем у «а», поэтому мы ее не используем в примере. — *Примеч. пер.*

Как бы такие большие числа втиснуть в более приемлемый диапазон? Можно применить операцию «остаток от деления»: в Python, как и во многих языках программирования, это `%`. Остаток сколь угодно большого целого числа (например, `base32(month)`) можно взять относительно достаточно малого делителя.

Немного поэкспериментировав, мы можем обнаружить, что для всех названий месяцев значения `base32(month) % 35` различны. Например, `base32('Август')` равно `2215474`, а `2215474 % 35 == 9`, и для каждого месяца результат будет иным. Теперь, как показано на рис. 3.1, мы можем завести массив на 35 элементов, с помощью которого можно узнать количество дней в данном месяце, *за одно обращение*, и перебирать все пары `(ключ, значение)` не нужно. Вычисляем формулу для февраля, получаем `0`, на нулевой позиции в `day_array` стоит `28` — в феврале 28 дней. Проделываем то же самое с ноябрем — получаем позицию `32` и 30 дней.

Давайте еще раз оценим, что у нас получилось. Вместо цикла по всем ключам в массиве нам нужно проделать несложное преобразование *самого ключа*, которое даст позицию нужного значения в массиве. Время нашего вычисления зависит от длины ключа, но *не зависит от общего количества ключей* — это очень большой шаг вперед!

```
base32('Февраль') % 35 == 0 // Нулевой элемент равен 28
day_array = (
  28, -1, 30, -1, -1, -1, -1,
  -1, 30, 31, -1, -1, 31, -1,
  31, 31, -1, -1, 30, -1, -1,
  31, -1, -1, -1, -1, 31, -1,
  -1, -1, -1, 31, 30, -1, -1
)
base32('Ноябрь') % 35 == 32 // 32-й элемент равен 30
```


Рис. 3.1. Массив с длинами месяцев; если формула не дает соответствующего индекса, в массиве записано –1

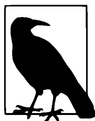
Поработать, правда, пришлось изрядно. Во-первых, подыскать формулу однозначного преобразования имени месяца в индекс, а во-вторых, создать массив на 35 элементов, из которых имеет смысл только дюжина, а значит, больше половины этого массива потрачено впустую. В нашем примере объем хранимых данных *N* равен 12, а объем хранилища *M* — 35.

Если для некоторой строки `s` элемент `day_array` в позиции `base32(s) % 35` равен `-1`, очевидно, эта строка не название месяца. Этим удобно пользоваться. Может показаться, что если `day_array[base32(s)%35] > 0`, то любая строка `s` — допустимое название месяца, но это, конечно, не так. К примеру, `base32("Промахнулось")` выдает то же самое, что и `base32("Февраль")`; получится, что «Промахнулось» — это такой месяц и в нем 28 дней! Это довольно неприятное свойство, и нам придется разобраться, как с ним справиться.

Хеш-функции и хеш-суммы

Формула `base32(s) % 35` — пример *хеш-функции*, которая отображает ключ произвольного размера в *хеш-сумму* (или просто *хеш*) в заданном диапазоне значений[1]. В нашем примере хеш не меньше нуля и не больше 34. Множество хеш-функций возвращают произвольное 32-разрядное целое (в диапазоне от –2 147 483 648 до 2 147 483 647) или произвольное 64-разрядное целое (в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807). Нетрудно заметить, что хеши бывают и отрицательными.

Задача отображения структурированных данных в фиксированные целые числа, то есть задача отыскания подходящей хеш-функции, занимает математиков последние несколько десятилетий. Программисты пользуются плодами их трудов: большинство языков программирования включают в себя вычисление хеша произвольного набора данных. В Python функция `hash()` для неизменяемых объектов встроена в сам язык.



Единственное обязательное свойство хеш-функции — однозначность. Хеш, посчитанный от двух одинаковых объектов, должен совпадать; нельзя каждый раз брать первое попавшееся число. Хеш неизменяемого объекта (например, string в Python) можно вычислять лишь единожды и потом хранить — это снизит общее количество вычислений.

[1] Сама base32() хеш-функцией, согласно этому определению, не является, потому что область ее значений не ограничена. — *Примеч. пер.*

Хеш-функция не обязана для каждого ключа давать *уникальное число*, не равное другим значениям функции (впрочем, в конце главы мы рассмотрим идеальные хеши, которые это умеют). Более того, ценой еще большей неуникальности мы можем резко уменьшить диапазон ее значений до 0... $M - 1$: возьмем остаток от деления на M.

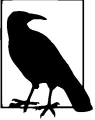
В табл. 3.1 представлено 32-разрядное значение функции `hash()` для некоторых строк и соответствующий им остаток от деления на 15. Формула `hash(key) % 15` обладает свойством однозначности, но если с точки зрения теории вероятности шанс на совпадение значений `hash()` у двух ключей исчезающе мал, то на малом диапазоне совпадение хешей вполне вероятно. В нашем примере таких совпадений два: «рифмы» — «возьми» со значением 10 и «ее» — «скорей» со значением 3. Несмотря на различие вероятностей, всегда стоит ожидать, что две разные строки могут иметь одинаковый хеш[1].

Таблица 3.1. Пример значений функции hash() и хеш-суммы с диапазоном на 15 значений

Ключ	Значение hash(ключ)	Значение hash(ключ) % 15
Читатель	30 711 662 493 708 375	0
ждет	–8 425 886 308 320 884 491	14
уж	4 546 933 634 410 127 049	9
рифмы	5 680 402 382 273 197 615	10
розы	–2 463 836 295 302 140 937	13
На	8 516 920 402 432 418 136	6
вот	5 909 557 570 350 754 617	12
возьми	8 292 399 267 285 665 935	10
ее	2 740 441 431 699 165 093	3
скорей	4 193 948 643 156 477 063	3

Если использовать остаток от деления на M для того, чтобы вычислить хеш, `hash(key) % M`, стоит озаботиться тем, чтобы диапазон оказался не меньше, чем общее количество ключей.

[1] Например, в языке программирования Java хеш 32-битный, и он действительно совпадает у некоторых разных строк; скажем, и у "misused", и у "horsemints" он равен 1 069 518 484. — *Примеч. авт.*



Лет десять назад в Python (а в Java — до сих пор) встроенная функция hash() стабильно возвращала один и тот же конкретный хеш для каждого конкретного параметра. В современном Python в значения хешей при запуске интерпретатора подмешивается соль — независимое случайное значение. В течение работы одного процесса Python соль не меняется, но предсказать ее значение для очередного запуска интерпретатора невозможно — это повышает безопасность приложений. Если хеш предсказуем, хакер может наполнить словарь заранее просчитанными ключами с одинаковым хешем. Это нарушит равномерность их распределения в хеш-таблице, и производительность словаря упадет до $O(N)$. Таким способом можно организовать так называемую DoS-атаку (от Denial of Service — «отказ в обслуживании»).

Больше информации относительно «подсоленного хеша» можно найти по ссылке https://oreil.ly/C4V0W и в Python-документации PEP 456. Кроме того, в конце главы есть упражнение на эту тему.

Хеш-таблица: хранение данных по ключу

Для удобства работы заведем отдельный тип данных для пары (*ключ*, *значение*):

```
class Entry:
  def __init__(self, k, v):
    self.key = k
    self.value = v
```

В примере 3.3 определен класс `Hashtable`, внутри которого в массиве `table` может храниться не более *M* объектов. Каждая из *M* позиций массива `table` — *ячейка* хеш-таблицы (по-английски bucket — «ведро» или «корзина»). Для начала определим, что каждая ячейка либо пуста, либо хранит ровно одну пару (объект типа `Entry`).

Пример 3.3. Неэффективная реализация хеш-таблицы

```
class Hashtable:
  def __init__(self, M=10):
    self.table = [None] * M        ❶
    self.M = M

  def get(self, k):                ❷
    hc = hash(k) % self.M
    return self.table[hc].value if self.table[hc] else None
```

```
  def put(self, k, v):          ❸
    hc = hash(k) % self.M
    entry = self.table[hc]
    if entry:
      if entry.key == k:
        entry.value = v
      else:                     ❹
        raise RuntimeError(f'Key Collision: {k} and {entry.key}')
    else:
      self.table[hc] = Entry(k, v)
```

❶ Заводим массив на `M` объектов.

❷ Метод `.get()` определяет номер ячейки по ключу `k`, для которого вычисляется хеш, и возвращает значение, если оно есть.

❸ Метод `.put()` определяет номер ячейки по ключу `k`, для которого вычисляется хеш, и перезаписывает хранящееся там значение — или записывает новое, если ячейка была пуста.

❹ Если хеш двух различных ключей приводит к одной и той же ячейке — это коллизия, происходит исключение.

Вот так нашей хеш-таблицей можно пользоваться:

```
table = Hashtable(1000)
table.put('Апрель', 30)
table.put('Май', 31)
table.put('Сентябрь', 30)
print(table.get('Август'))    # Промах: должно быть выведено None
print(table.get('Сентябрь'))  # Попадание: должно быть выведено 30
```

Если все работает как задумано, значит, в массиве на 1000 элементов нам удалось разместить три пары — объекты типа `Entry`. Скорость работы `.put()` и `.get()` не зависит от количества пар в таблице, так что их быстродействие можно оценить как O(1).

Если `.get(key)` не находит `key` в соответствующей хешу ячейке — это *промах*. Если `.get(key)` выбрал ячейку по хешу и ключ находящейся там пары равен `key` — это *попадание*. В таких случаях хеш-таблица работает так, как предполагается. Чего нам пока *не хватает* — это поведения на случай *коллизии хеша*, когда два или больше ключа имеют одинаковый хеш. Если не обрабатывать коллизии, метод `.put(ключ)` начнет удалять содержимое непустых ячеек, хеш ключа которых совпадает с хешем нового ключа. Смысла в хеше `Hashtable`, который теряет ключи, нет.

Определение коллизий и их разрешение последовательным просмотром

Вполне может случиться, что у ключей двух пар `e1 = (key1, val1)` и `e2 = (key2, val2)` окажется одинаковый хеш, несмотря на то что *сами ключи будут разными*. В табл. 3.1 и «возьми», и «рифмы» имеют хеш 10. Допустим, `e1` попал в `Hashtable` первым. Тогда при попытке добавить туда еще и `e2` возникнет *коллизия хешей*: для `e2` будет вычислена та же ячейка, что и для `e1`, при этом ячейка будет уже занята, а ее ключ, `key1`, не будет равен `key2` из `e2`. Если как-то это противоречие не разрешать, в одном объекте типа `Hashtable` нельзя будет хранить одновременно `e1` и `e2`.

Справиться с этим помогает стратегия *открытой адресации*: на случай возникновения коллизии (ячейка занята, ключи не совпадают) можно предусмотреть процедуру поиска (просмотра) других незанятых ячеек в таблице. Самый простой способ — *последовательный просмотр*, когда `.put()` просто проверяет следующие за вычисленной ячейки `table` до тех пор, *пока не найдет свободную*; если в процессе поиска массив закончится, а все ячейки заняты, `.put()` начнет с начала. Чтобы этот поиск всегда был успешным, будем проверять, что как минимум одна ячейка все еще свободна, а попытку записать что-то в последнюю свободную ячейку будем считать ошибкой переполнения хеш-таблицы.

Допустим, мы добавили пару в ячейку, индекс которой не равен хешу ключа, потому что соответствующая ячейка была занята. Чтобы убедиться в том, что так делать вообще можно, надо придумать алгоритм поиска по такому ключу. Для начала договоримся, что удалять пары из `Hashtable` мы не будем, будем только добавлять. Понятно, что теперь чем больше пар мы добавляем в хеш-таблицу, тем длиннее становятся участки занятых ячеек в `table`. Тогда поиск довольно прост: по хешу ключа определяем первоначальную ячейку, а если ключи не совпадают, начинаем рассматривать все последующие непустые ячейки. Если в каком-то из этих просмотров ключи совпадут, нужная ячейка найдена, а если попадется пустая — соответствующей пары в таблице нет. То есть пара `e1` найдется либо в ячейке с номером `hash(e1.key) % M`, либо *в последующих непустых ячейках* (с учетом перехода между концом и началом таблицы), либо не найдется.

На рис. 3.2 показано, как в `Hashtable` размером `M` = 7 с помощью последовательного просмотра добавляются пять пар с коллизией ключей (показаны только ключи). Заполненные серым ячейки — пустые. Ключ 20 попадает в `table[6]`, потому что 20 % 7 = 6, ключ 15 — в `table[1]`, а ключ 5 — в `table[5]`.

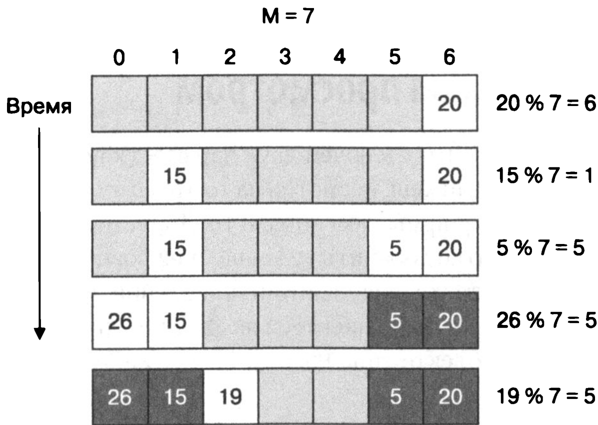
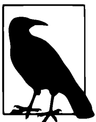


Рис. 3.2. Как выглядит Hashtable после добавления пяти пар (ключ, значение)

Попытка добавить пару с ключом 26 приведет к коллизии, потому что ячейка `table[5]` уже занята (парой с ключом 5; на рис. 3.2 просмотр занятой ячейки выделен темным фоном), и запускается последовательный просмотр: сначала `table[6]`, которая тоже занята, затем придется перевалить через конец таблицы и начать с начала, где для ключа 26 и отыщется первая свободная ячейка — `table[0]`. Добавление пары с ключом 19 тоже приводит к коллизии и просмотру *всех непустых ячеек* вплоть до `table[2]`, которая наконец оказывается свободна.

В экземпляр `Hashtable` с рис. 3.2 можно добавить еще только одну пару: хотя бы одна ячейка должна оставаться незанятой. Куда попадет пара с ключом 44? Хеш ключа 44 % 7 = 2, ячейка 2 занята, а 3 — свободна, так что пара попадет в `table[3]`. Поскольку `.get()` и `.put()` пользуются одним и тем же алгоритмом просмотра, впоследствии эту пару можно будет найти с помощью `.get(44)`.



Цепочкой для определенного хеша hc в таблицах с открытой адресацией наподобие нашего Hashtable называется последовательность ячеек в table. Эта последовательность начинается с table[hc] и продолжается вправо (с переходом из конца в начало table) вплоть до первой неиспользуемой ячейки, не включая ее. На рис. 3.2 цепочка для хеша 5 состоит из пяти элементов, хотя только в трех из них хеш ключа равен пяти. А для хеша 2 ключей вообще нет, но цепочка уже равна 1 из-за предыдущих коллизий. Длина цепочки не может превышать $M - 1$, потому что по правилам одна ячейка должна быть незанятой.

В примере 3.4 показано, как добавить открытую адресацию в `Hashtable`: класс `Entry` не меняется, а количество сохраненных пар запоминается в счетчике `N` для проверки того, что всегда есть хотя бы одна свободная ячейка (при этом за-

поминать, где она расположена, не надо!). Правило «одной свободной ячейки» очень важно, иначе цикл поиска `while` в `.get()` и `.put()` может оказаться вечным.

Пример 3.4. Реализация открытой адресации в Hashtable

```
class Hashtable:
  def __init__(self, M=10):
    self.table = [None] * M
    self.M, self.N = M, 0

  def get(self, k):
    hc = hash(k) % self.M            ❶
    while self.table[hc]:
      if self.table[hc].key == k:    ❷
        return self.table[hc].value
      hc = (hc + 1) % self.M         ❸
    return None                      ❹

  def put(self, k, v):
    hc = hash(k) % self.M            ❶
    while self.table[hc]:
      if self.table[hc].key == k:    ❺
        self.table[hc].value = v
        return
      hc = (hc + 1) % self.M         ❸

    if self.N >= self.M - 1:         ❻
       raise RuntimeError ('Table is Full.')

    self.table[hc] = Entry(k, v)     ❼
    self.N +=  1
```

❶ Начнем с первой же ячейки, в которой может быть ключ `k`.

❷ Если ключ найден, вернем соответствующее `k` значение.

❸ В противном случае просмотрим следующую ячейку, при необходимости продолжая просмотр с начала таблицы.

❹ Если `table[hc]` не занята, очевидно, `k` в `table` нет.

❺ Если ключ найден, обновим `value` соответствующей `k` ячейки.

❻ Если `k` нет в `table`, а свободная ячейка осталась одна, инициируем исключение `RuntimeError`.

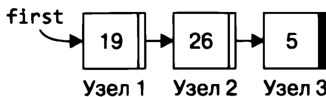
❼ Запишем пару в свободную ячейку `table[hc]` и увеличим счетчик ключей `N`.

Так что, мы изобрели эффективный способ хранения? Теперь быстродействие `.put()` и `.get()` не зависит от `M`, размера таблицы ключей? Не тут-то было!

СВЯЗНЫЙ СПИСОК ИЛИ ДИНАМИЧЕСКИЙ МАССИВ?

В языках программирования с непосредственным выделением памяти часто используется динамическая структура данных под названием *«связный список»*. Для массива выделяется сразу один непрерывный блок памяти фиксированного размера, а для связного списка каждый элемент, называемый *узлом*, располагается в памяти отдельно, так что заведомо известен только *начальный* узел связного списка, в котором вместе со значением хранится *ссылка* (в языках такого типа она обычно называется *указателем*) на следующий *узел* и т. д. Для поиска значения по связному списку необходимо проходить по ссылкам в узлах.

Список на рисунке ниже содержит три узла, в каждом из которых хранится свое значение. Стрелки на картинке — это ссылки на следующий узел списка (обычно для этого используется специальный тип данных, указатель, традиционное имя для такого указателя — `next`). Последний узел вместо ссылки содержит специальное «пустое» значение в знак того, что ссылаться больше не на что, отдельная переменная-указатель `first` хранит ссылку на начальный элемент списка.



Размер связного списка определяется количеством узлов при его просмотре от начала до конца, начиная с указателя `first`, по всем `next` до тех пор, пока они не пусты. Сами узлы могут появляться в любом месте памяти и в любое время. В объектно-ориентированных языках удобно моделировать тип узла классом. В языках программирования с непосредственным выделением памяти необходимо освобождать память, выделенную под узел, если он больше нигде не используется. В других, например в Python и Java, выделением и освобождением памяти занимается сама исполняющая система языка.

- *Вставка в начало.* Чтобы вставить узел в начало списка, надо этот узел, `Node0`, создать, положить в него нужное значение и сделать так, чтобы `next`, указатель на следующий элемент, ссылался на `Node1`. Кроме того, в переменной `first` следует переделать указатель на `Node1` в указатель на `Node0` (а также во всех других местах, ссылающихся на наш список).
- *Добавление в конец.* Можно также хранить `last` — указатель на последний элемент связного списка. В этом случае для добавления значения в конец списка необходимо создать узел `Node4` с пустым полем `next`, сохранить в узел требуемое значение, сходить по ссылке из `last` (в нашем случае это будет узел `Node3`), заменить там поле `next` указателем на `Node4` и сам `last` тоже переделать в указатель на `Node4`.
- *Вставка в произвольное место.* Если заранее известно, что новый узел `q` необходимо вставить после конкретного узла `p`, надо создать этот новый узел `q` с нужным значением и полем `next`, которое совпадает с полем `next` узла `p`. После чего поле `next` узла `p` сделать указателем на новый узел `q`.

- *Удаление произвольного элемента.* Чтобы удалить некое значение из списка, необходимо найти узел, поле `next` которого указывает на узел, содержащий искомое значение. Тогда значение `next` искомого узла записывается в `next` узла-предшественника (где раньше была ссылка на искомый узел). Отдельно обрабатываются случаи, когда искомым оказывается первый или последний узел. Кроме того, следует соблюдать принятые в языке программирования договоренности по освобождению памяти.

Моделирование связного списка — отличное упражнение для изучающих алгоритмы. Это довольно хрупкая конструкция, реализация примитивов работы с ней требует внимательности и аккуратного анализа нескольких граничных ситуаций и т. д. Тем не менее именно с помощью связных списков в большинстве учебников программирования продемонстрирована работа с динамически изменяющимися объемами данных: если из встроенных типов данных в языке программирования есть только переменные, массивы, структуры и указатели, иного способа нет.

Связные списки в Python не используются никогда — или *почти* никогда: вместо них следует применять динамические массивы. Гвидо ван Россум, автор языка Python, даже назвал такой тип данных `list`, прозрачно намекая на область его применения.

Рассмотрим таблицу сравнения быстродействия различных операций над связными списками и динамическими массивами.

Операция	Связные списки	Динамические массивы
Поиск элемента	O(*N*)	O(*N*)
Двоичный поиск элемента	Невозможен	Возможен
Поиск элемента по индексу	O(*N*)	O(1)
Добавление и удаление элемента в конце	O(1)	O(1)
Добавление и удаление элемента в произвольном заранее известном месте	O(1)	В среднем O(*N*)
Добавление и удаление элемента в начале	O(1)	O(*N*)
Количество операций «выделение памяти» при добавлении *N* элементов	O(*N*)	O(log *N*)

Как видно из таблицы, `list` в Python работает не хуже или даже лучше, чем связный список, за исключением случая, когда приходится модифицировать размер списка *не в его конце*. В самом деле, для вставки нового элемента в связный список, допустим между вторым и третьим элементами, достаточно указатель на третий записать в поле `next` нового элемента, а в поле `next` второго записать указатель на этот новый. А вот для вставки третьего элемента в динамический массив нужно добавить пустой элемент

в его конец (это операция быстрая), а затем переставить все элементы, начиная с последнего, на одну позицию вперед — *N* – первый записать в позицию *N*, *N* – второй — в позицию *N* – 1 и т. д. до третьего, который записывается в позицию 4, освобождая тем самым место для нового элемента в позиции 3. Эта операция, очевидно, достаточно неэффективна — порядка *N*.

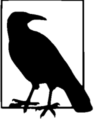
Поначалу кажется, что такое различие быстродействия при работе с *почти любым элементом* списка сводит на нет все преимущества `list`. Однако в действительности важными являются только операции работы с *началом* списка. Например, эффективно реализовать абстракцию «очередь» (FIFO — «первым вошел, первым вышел») с помощью `list` нельзя. Для этого в Python есть отдельный тип данных — `collections.deque`, в котором операции с обоих концов списка одинаково быстры (за счет некоторых накладных расходов).

А вот эффективная работа (удаление или добавление) с элементом в *произвольной* позиции не дает связному списку практически никаких преимуществ перед `list`. Для того чтобы понять, что вставка или удаление должны происходить в позиции *k*, эту *k* надо сначала определить. А все операции поиска в связном списке — как по значению, так и по индексу — линейны. Таким образом, например, в операции «поиск + вставка» линейный поиск доминирует над константной вставкой, и порядок ее сложности оказывается одинаков и для связных списков, и для динамических массивов (в которых, как мы видим, с поиском еще и получше).

Не меньшую роль в выборе динамических массивов в качестве базового типа данных в Python сыграла необходимость управлять памятью, под них выделяемой. Выделение и освобождение памяти — функция ядра операционной системы, она может работать быстро, а может и долго. В языках с явным выделением памяти, таких как C и отчасти C++, программист сам принимает решение, как часто он может обращаться в алгоритме к системным вызовам. В языках с автоматическим учетом памяти, таких как Java и Python, этим занимается интерпретатор (или исполняющая система) — и появляется возможность поэффективнее реализовать работу с памятью. В Python для изменения размера списка используется стратегия масштабирования, которую мы опишем ниже.

Посмотрим, что происходит в *худшем случае*. Поначалу `Hashtable` на N элементов пуст[1] и пара добавляется в пустую ячейку — для определенности пусть в `table[0]`. Если теперь каждый из оставшихся $N - 1$ запросов на добавление будет приводить к коллизии ключа, сколько всего ячеек придется за это время просмотреть? Для первого запроса — одну, для второго — две (из-за коллизий в первой), для третьего — три и т. д. Очевидно, во время k-го запроса придется просмотреть k ячеек. Значит, всего просмотров будет $1 + 2 + ... + (N - 1)$, то есть $N \times (N - 1) / 2$. Если поделить на количество запросов N, получится *в среднем* $(N - 1) / 2$ действий на один `.put()`.

[1] Напомним, что минимум одна ячейка всегда должна быть не заполнена, так что размер таблицы table в этом случае N + 1. — *Примеч. пер.*



Согласно классификации из главы 2, сложность $(N - 1) / 2$ — это $O(N)$. В самом деле, если раскрыть скобки, получится $(N / 2) - 1/2$. Порядок сложности определяется доминирующей составляющей — $N / 2$. Выходит, что в худшем случае количество просмотренных ячеек прямо пропорционально N (в нашем алгоритме равно половине N).

Итак, мы рассмотрели *наихудший случай*, в котором среднее количество просмотров ячеек оценивается как $O(N)$. В алгоритме мы оценивали количество просмотров ячеек, а не время работы или количество инструкций, потому что производительность `get()` и `put()` напрямую зависит от количества просмотров.

Похоже, мы зашли в тупик. Можно увеличить M, размер `table`, так, чтобы он значительно превосходил возможное количество хранимых ключей, N, — тогда для случайных ключей количество коллизий (а следовательно, и время поиска) в таблице уменьшится[1]. Однако стоит ошибиться с прогнозами относительно N — и по мере приближения его к M производительность будет становиться все ужаснее; хуже того, на $N = M - 1$ переполнится `table` и значения вообще нельзя будет добавлять. В табл. 3.2 представлено время вставки N пар в структуру типа «`Hashtable` с открытой адресацией» размером M. Обратим внимание вот на что.

- Для небольшого N, скажем 32, среднее время работы (соответствующая строка таблицы) везде примерно одинаково, независимо от M, размера `Hashtable`, потому что M либо само по себе мало, либо значительно превосходит N.
- В любой `Hashtable` размером M среднее время вставки N ключей постоянно растет с увеличением N — это видно в соответствующем столбце таблицы.
- Любая диагональ «с северо-запада на юго-восток» в таблице содержит примерно одинаковое время работы. С учетом того, как меняются M и N в столбцах и строках таблицы, можно сказать, что для того, чтобы в `Hashtable` можно было вместо N ключей добавлять с той же средней скоростью $2N$ ключей, нужно увеличить ее размер вдвое.

Наша реализация типа данных «хеш-таблица» отлично работает — но только в случае, когда размер хранилища существенно больше предполагаемого количества ключей, которые мы ходим туда поместить. Если ошибиться с оценкой этого количества, быстродействие операции начинает стремительно падать, может и в сто раз уменьшиться. Что еще неприятнее, мы не предусмотрели удаление ключа из таблицы — такой структурой данных далеко не всегда удобно пользоваться. Самый простой способ побороть эти трудности — хранить цепочки в отдельных списках.

---

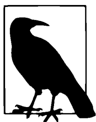
[1] Худший случай при этом не отменял никто, но среди случайных наборов он встречается довольно редко. — *Примеч. пер.*

Таблица 3.2. Среднее время добавления N ключей в таблицу Hashtable размером M (в миллисекундах)

	8192	16 384	32 768	65 536	131 072	262 144	524 288	1 048 576
32	0.048	0.036	0.051	0.027	0.033	0.034	0.032	0.032
64	0.070	0.066	0.054	0.047	0.036	0.035	0.033	0.032
128	0.120	0.092	0.065	0.055	0.040	0.036	0.034	0.032
256	0.221	0.119	0.086	0.053	0.043	0.038	0.035	0.033
512	0.414	0.230	0.130	0.079	0.059	0.044	0.039	0.035
1024	0.841	0.432	0.233	0.132	0.083	0.058	0.045	0.039
2048	1.775	0.871	0.444	0.236	0.155	0.089	0.060	0.047
4096	3.966	1.824	0.887	0.457	0.255	0.144	0.090	0.060
8192	—	4.266	2.182	0.944	0.517	0.276	0.152	0.095
16 384	—	—	3.864	1.812	0.908	0.484	0.270	0.148

Раздельное хранение цепочек в списках

Перепишем `Hashtable` так, чтобы в таблице хранились не сами пары, а цепочки пар с совпадающими ключами. Такой способ называется *хешированием с раздельными цепочками*. В случае *последовательного просмотра* для размещения пары нам приходилось искать очередную свободную ячейку, теперь же в ячейке `table[idx]` будет храниться не пара, а весь список пар, хеши ключей которых совпадают с `idx`. Если таких ключей нет, ячейка пуста: содержит пустой список (пример 3.5). Количество ячеек для удобства оставим *M*.



Понятие цепочки при раздельном хранении намного очевиднее цепочки в открытой адресации: теперь длина списка в ячейке и есть длина цепочки.

Как и прежде, хеш-функцией, определяющей номер ячейки для данного ключа, у нас будет формула `hash(key) % M`. На рис. 3.3 представлена хеш-таблица с семью ячейками; как и прежде, показаны только ключи.

На рис. 3.3 пары (*ключ*, *значение*) добавляются в том же порядке, что и на рис. 3.2. Поначалу все ячейки пустые, это обозначено серым фоном. Первые три пары с ключами 20, 15 и 5 попадают в начало цепочек в соответствующих ячейках

хеш-таблицы. Когда добавляется пара с ключом 26, возникает коллизия и эта пара добавляется в конец цепочки ячейки `table[5]`. То же самое происходит для ключа 19, так что в результате цепочка, соответствующая хешу 5, содержит три пары.

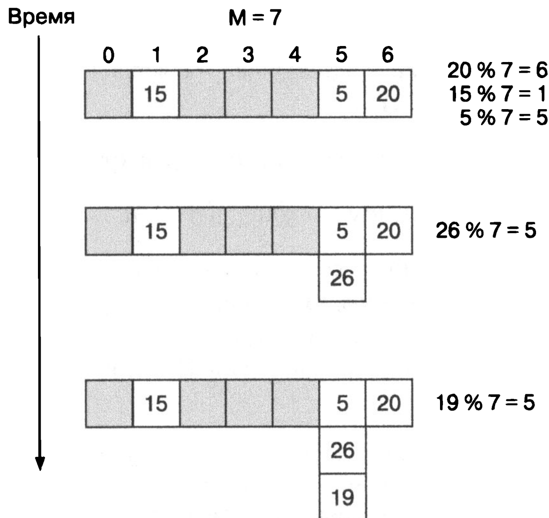


Рис. 3.3. Как выглядят цепочки Hashtable после вставки пяти пар (ключ, значение)

Пример 3.5. Вариант хеш-таблицы с раздельным хранением цепочек

```
class Hashtable:
    def __init__(self, M=10):
        self.table = [[] for i in range(M)]  ❶
        self.M = M
        self.N = 0

    def get(self, k):
        hc = hash(k) % self.M                ❷
        for entry in self.table[hc]:         ❸
            if entry.key == k:
                return entry.value
        return None

    def put(self, k, v):
        hc = hash(k) % self.M                ❷
        for entry in self.table[hc]:         ❸
            if entry.key == k:               ❹
                entry.value = v
                return
        self.table[hc].append(Entry(k, v))   ❺
        self.N += 1
```

➊ Написать `self.table = [] * M` значило бы создать список, все элементы которого являются *одним и тем же* объектом; нам же надо, чтобы это было *M разных* объектов (пускай они поначалу все равны `[]`).

➋ Вычисляем номер ячейки `hc` (остаток от деления хешированного ключа `k` на размер таблицы).

➌ Ищем совпадение `k` с ключом одной из пар (быстродействие этой части пропорционально длине цепочки).

➍ Если пара с данным ключом `k` найдена, изменим хранимое значение.

➎ Добавим пару с новым ключом `k` в конец цепочки.

Методы `.get()` и `.put()` очень похожи: мы проходим по всем парам в цепочке `table[hc]` и сравниваем нулевой элемент пары (ключ) с `k`; если ключ найден — `get()` вернет значение, а `put()` поменяет старое значение на новое, если не найден — `get()` вернет `None`, а `put()` добавит пару `(k, v)` в конец цепочки.

Добавить элемент в конец динамического массива типа list можно за одну операцию, так что это самый эффективный способ. Общая вычислительная сложность добавления пары в хеш-таблицу все равно зависит от длины соответствующей цепочки: прежде чем добавлять пару, надо убедиться, что ее в этой цепочке нет.

Аналогично функции добавления функция удаления пары по ключу, `remove()`, будет искать пару с ключом `k` в соответствующей хешу цепочки таблице и удалять, если такая пара нашлась. По договоренности метод `.remove()` возвращает поле *значение* удаленной пары или `None`, если ключ не нашелся (пример 3.6).

Пример 3.6. Функция удаления пары по ключу

```
def remove(self, k):
    hc = hash(k) % self.M
    for i, entry in enumerate(self.table[hc]):   ➊
        if entry.key == k:
            del self.table[hc][i]                ➋
            self.N -= 1
            return entry.value                   ➌
    return None                                  ➍
```

➊ Функция `enumerate(`*список*`)` возвращает последовательность пар *индекс элемента, элемент*; индекс нам может потребоваться для удаления.

➋ Динамический массив позволяет удалять любой элемент по его индексу.

❸ Если ключ найден, возвращаем значение из удаленной пары.

❹ Если удаления не было, возвращаем `None`.

Стоит заметить, что операции над динамическим массивом (тип `list`), которые работают *не с концом* этого массива, обычно имеют линейную сложность, в среднем пропорциональную длине самого массива. Это относится как к операциям поиска вида *`элемент`* `in` *`список`*, так и к операциям удаления вида `del` *`список`*`[i]`. Поскольку массив — это непрерывный блок элементов в памяти, для удаления из него i-го элемента на место i-го надо переписать $(i + 1)$-й, на место $(i + 1)$-го — $(i + 2)$-й и т. д. до последнего элемента, который должен встать на место предпоследнего. Эта часть алгоритма очевидно зависит от длины списка и в среднем переставляет половину его элементов.

Может показаться, что для моделирования цепочек эффективнее было бы использовать *связные списки*, потому что в них операция удаления заранее известного элемента не зависит от длины списка. Однако в нашем случае (как и в подавляющем большинстве других) операции удаления непременно предшествует операция *поиска*, и, поскольку она заведомо линейна, ее сложность доминирует над константной сложностью удаления.

Насколько эффективны хеш-таблицы с раздельным хранением цепочек? Чтобы ответить на этот вопрос, надо посчитать и количество обращений к ячейкам таблицы, и количество просмотров пар в цепочке. Забегая вперед, скажу, что быстродействие таблиц с раздельными цепочками практически такое же, что и быстродействие таблиц с открытой адресацией. Единственное важное достижение — количество пар в раздельных цепочках может быть актуально бесконечным. Тем не менее, как мы скоро увидим, когда количество хранимых пар N заметно превышает количество ячеек M, производительность значительно падает. Требования к памяти у обоих методов примерно одинаковые, однако в случае с раздельными цепочками структура данных посложнее и нужно дополнительно заводить M пустых списков.

Оценка

Итак, у нас есть две различные структуры, которые реализуют абстракцию «хеш-таблица» и позволяют хранить пары (*`ключ, значение`*). В варианте с раздельными цепочками мы предусмотрели операцию удаления пары. Удаление также можно было бы реализовать и в варианте с открытой адресацией: найти пару, записать на ее место последнюю пару в цепочке, очистить ячейку, в которой раньше лежала последняя пара. Однако такая операция нарушает *порядок добавления* ключей в цепочку — свойство, которое может понадобиться при

использовании хеш-таблицы. В любом случае эффективность обеих структур еще предстоит оценить.

Сначала оценим сложность по памяти: сколько ее требуется для хранения N пар (*ключ, значение*)? Обе структуры изначально содержат массив из M элементов (при открытой адресации это пустые ячейки, при раздельном хранении цепочек — пустые списки ячеек). Для хеш-таблицы с раздельными цепочками N может быть сколь угодно большим, для хеш-таблицы с открытой адресацией N должно быть строго меньше M — значит, это M нужно выбирать, исходя из знания о максимально возможном количестве пар. В обеих структурах размер базового массива `table` прямо пропорционален M.

Когда мы добавим в хеш-таблицу все N пар, общий объем памяти при хранении с открытой адресацией не изменится, потому что базовый массив уже хранит ячейки типа `Entry`. При раздельном хранении цепочек каждая цепочка — это список из ячеек типа `Entry`, поэтому с каждой добавленной парой объем увеличивается на размер одной ячейки, и под конец размер этого дополнительного пространства будет прямо пропорционален N.

- Для открытой адресации требуется объем памяти, пропорциональный сразу и M, и N, но, поскольку $N < M$, можно просто заметить, что сложность по памяти в данном случае — $O(M)$.
- Раздельное хранение цепочек для базового массива требует наличия памяти объемом, пропорциональным M, а для всех цепочек — памяти объемом, пропорциональным N; так что сложность по памяти стоит оценить как $O(M + N)$.

Теперь возьмемся за сложность по времени. Основное действие — обращение к ячейке таблицы, а мы будем подсчитывать количество таких обращений. *Наихудший случай* наступает, когда все вычисленные хеши ключей оказываются равны. В обеих структурах при этом $M - 1$ цепочки пусты, а оставшаяся единственная цепочка, которая соответствует этому одному на всех хешу, содержит все N пар. Для *наихудшего случая* предположим, что искомый элемент всегда оказывается в конце этой цепочки, так что и поиск по списку при раздельном хранении, и поиск по самой таблице при открытой адресации прямо пропорционален N. Можно смело сказать, что *независимо от реализации* сложность *худшего случая* для операции `get()` — $O(N)$.

Казалось бы, к чему тогда все наши усилия? Если хеш-функция исследована математически, она дает очень хороший *разброс* хешей на ожидаемом множестве ключей. В действительности с увеличением M вероятность коллизии уменьшается, а значит, уменьшаются и длины цепочек в наших структурах. В табл. 3.3 приведено сравнение обоих подходов: будем помещать 321 129 английских слов в хеш-таблицы размером M от $N / 2$ до $2N$. Вдобавок в первой строке будут приведены данные по хеш-таблицам размером $M = 20N$, а в пяти последних — малых (непригодных для открытой адресации) размеров.

Таблица 3.3 показывает результаты двух измерений для каждого M, N.

- Средний размер непустой цепочки в хеш-таблице. Этот параметр годится и для открытой адресации, и для раздельного хранения.
- Максимальный размер цепочки в хеш-таблице. Этот параметр тоже годится для обеих структур и подсказывает ухудшение их быстродействия. Когда таблица с открытой адресацией близится к переполнению или хранимая цепочка становится чересчур длинной, быстродействие падает.

Таблица 3.3. Усредненные показатели при добавлении 321 129 ключей в хеш-таблицу размером M по мере уменьшения M

M	Раздельное хранение		Открытая адресация	
	Средняя цепочка	Наибольшая цепочка	Средняя цепочка	Наибольшая цепочка
6 422 580	1.0	4	1.1	5
642 258	1.3	7	3.0	43
610 145	1.3	6	3.3	46
579 637	1.3	7	3.7	56
550 655	1.3	7	4.1	58
523 122	1.3	7	4.6	69
496 965	1.4	7	5.5	98
472 116	1.4	7	6.4	112
448 510	1.4	8	7.9	107
426 084	1.4	7	10.3	204
404 779	1.4	9	13.6	206
384 540	1.5	9	21.4	319
365 313	1.5	8	39.0	700
347 047	1.5	8	94.4	1516
329 694	1.6	8	784.5	9759
313 209	1.6	9	*	*
187 925	2.1	10	*	*
112 755	3.0	13	*	*
67 653	4.8	15	*	*
40 591	7.9	21	*	*
24 354	13.2	30	*	*

Все приведенные значения растут с уменьшением M[1]. Причина в том, что чем меньше хеш-таблица, тем больше случается коллизий при добавлении и цепочки ключей становятся длиннее. Хорошо видно, что в варианте с открытой адресацией этот рост существенно выше, особенно под конец, когда в одну многотысячную цепочку начинают сливаться последовательности ключей с разными хешами. Что еще хуже, стоит M оказаться меньше, чем N, и открытая адресация вообще становится неприменима (в таблице отмечено звездочками). А вот хранение цепочек в списках себя оправдало: в разумных пределах оно нечувствительно к этой проблеме. Когда размер таблицы M достаточно велик — скажем, в два раза больше количества хранимых элементов N, — средний размер цепочки вообще близок к 1, да и длина наибольшей цепочки сравнительно невелика. Тем не менее *M нужно выбирать заранее*, и в случае открытой адресации у нас закончится место, как только N станет равным $M - 1$.

С раздельным хранением не все так страшно. Таблица 3.3 показывает, что, даже когда N превышает размер таблицы M более чем десятикратно, списки растут себе и растут, и даже производительность не падает так ужасающе, как при открытой адресации с N, всего лишь примерно равным M. Это можно понять по размеру максимальной цепочки в списке, который приведен в той же таблице.

Изучив эти показатели, мы можем изобрести для хеш-таблицы размером M простую методику оценки эффективности. Производительность хеш-таблицы зависит от того, насколько она «заполнена», то есть от отношения N к M. В экономической науке есть похожее понятие — *альфа-индекс*[2], но мы будем пользоваться более очевидным термином — «*загруженность* хеш-таблицы».

- При раздельном хранении цепочек *показатель загруженности* — это *средняя длина цепочки* (в отличие от данных в табл. 3.3, где приведена средняя длина только среди непустых цепочек). Этот показатель может превышать 1 и зависит только от объема доступной памяти.
- При открытой адресации *показатель загруженности* — это доля занятых ячеек таблицы. Наибольшее его значение — $(M - 1) / M$, так что он *не может превышать 1*.

Многолетние исследования показали, что хеш-таблицы начинают стремительно терять производительность при показателе загруженности больше 0,75 — на-

---

[1] Можно заметить, что длина наибольшей раздельной цепочки растет не слишком строго, в силу случайности функции hash(). Всегда есть вероятность, что для двух близких значений M в одном случае пары разложатся по цепочкам оптимальнее, чем в другом. — *Примеч. пер.*

[2] Строго говоря, альфа-индекс — это такое число M / N, при котором наблюдается субъективно наиболее эффективная работа экономической модели. — *Примеч. пер.*

пример, когда таблица с открытой адресацией заполнена на 3/4[1]. При раздельном хранении цепочек хеш-таблица не бывает совсем заполнена, но принцип остается тот же[2].

На рис. 3.4 приведены средние и наибольшие длины цепочек после добавления N = 321 129 слов в хеш-таблицу размером M. Значение M меняется по оси ординат, длины отложены по оси абсцисс. Средние длины помечены ромбами, их шкала расположена слева от диаграммы, наибольшие — квадратами, их шкала — справа. Задавшись целью обеспечить определенный средний или наибольший размер цепочки для заранее известного количества ключей N, можно по аналогии с диаграммой на рис. 3.4 подобрать подходящее M.

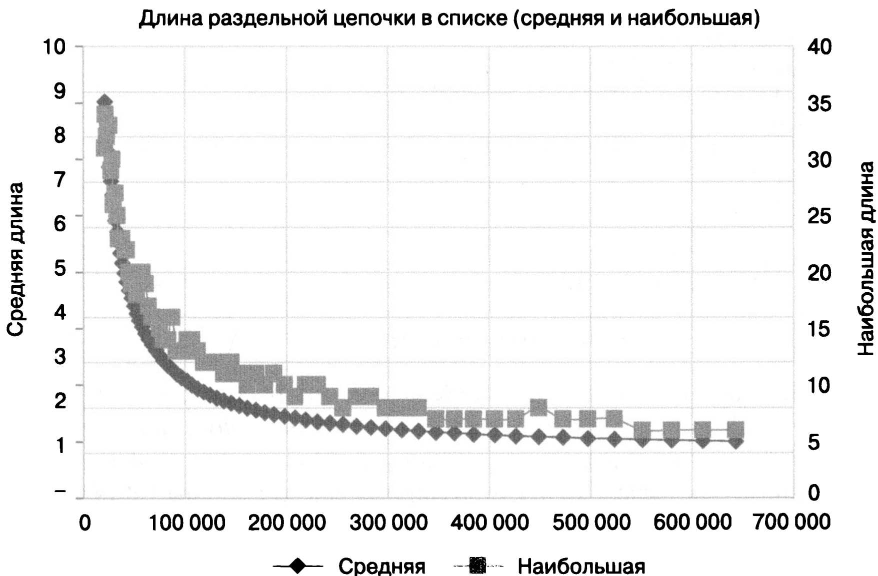


Рис. 3.4. При вставке заданного количества элементов N в хеш-таблицы разных размеров можно уверенно предсказать, какой длины будут средняя и наибольшая цепочки

Если бы мы могли расширять хеш-таблицу, то есть увеличивать M по необходимости, можно было бы удерживать оптимальный показатель загруженности!

[1] В структуре данных dict в Python пороговое значение другое — 2/3. — *Примеч. авт.*

[2] Показатель загруженности задает порог, после которого структурой становится неудобно пользоваться. Например, согласно табл. 3.3, при открытой адресации порог 2/3 соответствует средней длине цепочки примерно 5, а максимальной — примерно 100; а порог 3/4 — 10 и 200 соответственно. — *Примеч. пер.*

Как мы сейчас увидим, при должном старании такую таблицу сделать вполне реально.

Расширяемые хеш-таблицы

В классе `DynamicHashtable` из примера 3.7 заведено поле `load_factor` со значением `0.75`. Это показатель загруженности, на основании которого вычисляется пороговая загруженность `threshold`.

Пример 3.7. Определение показателя и порога загруженности при создании динамической хеш-таблицы

```
class DynamicHashtable:
    def __init__(self, M=10):
        self.table = [[] for i in range(M)]
        if M < 1:
            raise ValueError('Hashtable storage must be at least 1.')
        self.M = M
        self.N = 0

        self.load_factor = 0.75
        self.threshold = min(M * self.load_factor, M-1)       ❶
```

❶ Для $M \leq 3$ порог не должен превышать $M - 1$.

Что делать, если хеш-таблица будет заполнена свыше порогового значения? Распространенный подход *геометрического масштабирования* рекомендует сразу увеличивать таблицу вдвое. Точнее, увеличивать вдвое и еще прибавлять к размеру единицу. Как только количество занятых ячеек таблицы `table` превысит порог (`threshold`), для эффективной работы понадобится таблицу увеличить. В примере 3.8 приведен доработанный метод `put()` для раздельно хранимых цепочек.

Пример 3.8. Доработанный метод put(), который вызывает resize()

```
    def put(self, k, v):
        hc = hash(k) % self.M
        for entry in self.table[hc]:
            if entry.key == k:
                entry.value = v
                return
        self.table[hc].append(Entry(k, v))      ❶
        self.N += 1

        if self.N >= self.threshold:            ❷
            self.resize(2*self.M + 1)           ❸
```

❶ Добавим новую ячейку в конец цепочки `table[hc]`.

❷ Проверим, не превысило ли *N* порог.

❸ Увеличим размер массива ячеек вдвое (плюс еще одна ячейка).

Процедура удвоения создает новый массив (при открытой адресации — ячеек, при раздельном хранении — цепочек) и заново добавляет в эту новую таблицу все пары (*ключ, значение*). Поскольку загруженность таблицы стала равна пороговой, но не превысила ее, мы ожидаем в среднем константное время добавления одной ячейки и, следовательно, в среднем линейную сложность функции `resize()` по отношению к количеству ячеек, то есть предположительно O(*M*). Прежде чем продвинуться дальше, попытаемся (некорректно!) упростить процедуру удвоения. Во многих языках программирования есть операция копирования части или целого массива в другой массив. Эта операция уж точно имеет сложность O(*M*), ибо копировать надо ровно *M* ячеек старого массива. Почему бы нам просто не скопировать старый массив в новый? А в случае Python так и вовсе просто добавить $M + 1$ ячеек прямо в конец старого (тип данных `list` позволяет сделать это очень эффективно)? Не окажется ли такое решение в несколько раз быстрее повторного добавления всех пар в таблицу?

Так-то оно так, только работать как надо эта операция не будет! После механического расширения хеш-таблицы с открытой адресацией некоторые ее элементы *нельзя будет найти*, а раздельно еще и хранимые цепочки окажутся некорректными! На рис. 3.5 показано, к чему приводит механическое удвоение размера таблицы для обоих подходов. Попробуйте, к примеру, проследить, что произойдет с ключами 19 и 26.

Рис. 3.5. Некоторые ячейки «теряются», если таблицу просто увеличить, не меняя содержимого

Начнем искать ключ 19. Хеш этого ключа — 19 % 15 = 4, но соответствующая ячейка будет пуста в обеих структурах, то есть его там как бы нет. При старом размере таблицы с открытой адресацией M = 7 последовательный просмотр цепочки (с перезапуском от начала таблицы) приводил пару с ключом 19 в ячейку 2. А теперь и хеш другой, и размер, так что возврат к началу таблицы происходит после ячейки 15, то есть алгоритм поиска не соответствует действительности.

По чистой случайности в таблице с открытой адресацией некоторые ключи все же можно найти, но алгоритм их поиска не будет адекватен исходному. Например, ключ 20 будет найден, но не сразу (в `table[5]`), а последовательным просмотром, в `table[6]`. Точно так же и ключ 15 найдется не в `table[0]`, а в `table[1]`. И только ключ 5 остается на своем месте — потому что его хеш не изменился.

Дело, конечно же, в том, что при увеличении таблицы *меняется хеш-функция*, так что элементы, скорее всего, окажутся не на своих местах. Придется вернуться к идее *повторного хеширования*: создадим пустую хеш-таблицу размером $2M + 1$ и заново добавим в нее все пары из исходной таблицы. В терминах примера 3.9: каждую пару (*ключ, значение*) исходного массива `self.table` добавим в новую таблицу `temp` с помощью `temp.put(`*ключ, значение*`)` — тогда все пары точно можно будет найти. Воспользуемся динамической природой объектов Python и подменим старый массив новым (никакого копирования ячеек при этом не происходит). В результате у нового массива окажется *два имени* — `self.table` и `temp.table`, а у старого — ни одного, и вскорости он будет удален исполняющей системой Python. При выходе из функции `resize()` прекращает существование переменная `temp`, и все объекты, у которых не осталось имен, также будут удалены — но новый массив никуда не денется: у него останется еще одно имя. Этот способ подходит обеим схемам хранения цепочек.

Пример 3.9. Динамическое масштабирование таблицы с открытой адресацией путем повторного хеширования

```
def resize(self, new_size):
    temp = DynamicHashtable(new_size)                  ❶
    for key, value in self.table:
        temp.put(key, value)                           ❷
    self.table, self.M = temp.table, temp.M            ❸
    self.threshold = self.load_factor * self.M
```

❶ Создаем временную хеш-таблицу нужного размера.

❷ Добавляем в нее все пары из старой таблицы.

❸ Обновляем массив ячеек, значение M и порог `threshold`.

При раздельном хранении цепочек процедура практически не меняется, надо только вспомнить, что элементы массива `table` не сами ячейки, а их списки (пример 3.10).

Пример 3.10. Динамическое масштабирование таблицы с раздельным хранением цепочек путем повторного хеширования

```
def resize(self, new_size):
    temp = DynamicHashtable(new_size)
    for bucket in self.table:                    ❶
      for key, value in bucket:                  ❷
          temp.put(key, value)
    self.table, self.M = temp.table, temp.M
    self.threshold = self.load_factor * self.M
```

❶ Внешний цикл — по спискам ячеек.

❷ Внутренний цикл — по ячейкам в списке.

Итак, мы определили механизм масштабирования хеш-таблицы. На рис. 3.6 показано, как в действительности должны выглядеть таблица с открытой адресацией (см. рис. 3.2) и таблица с раздельными цепочками (см. рис. 3.3) после увеличения размера с 7 до 15.

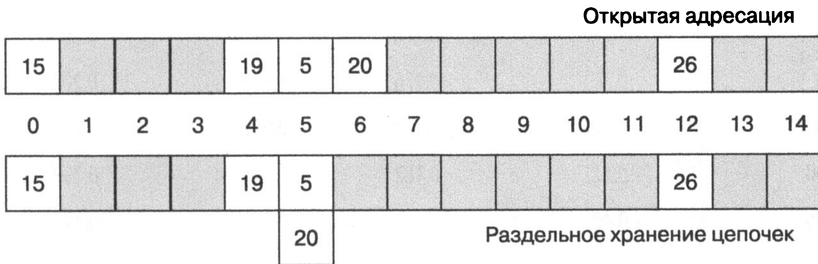


Рис. 3.6. Структура хеш-таблиц после масштабирования с повторным хешированием

Насколько лучше стало с применением масштабирования? Проведем эксперимент: для каждого начального размера таблицы M предпримем 25 тестов, в которых будем измерять:

- *время заполнения* — время, которое потребуется для того, чтобы добавить N = 321 129 ключей в хеш-таблицу с начальным размером M и удвоением размера при превышении порога загруженности;
- *время доступа* — время, за которое мы найдем в получившейся таблице все N ключей.

В табл. 3.4 приведены результаты измерения времени заполнения и времени доступа для хеш-таблиц с открытой адресацией и с раздельными цепочками, причем начальный размер таблицы, *M*, меняется от 625 до 6 400 000. В последней строке приведены аналогичные данные для хеш-таблицы фиксированного размера на 428 172 ячейки — этот размер соответствует пороговому заполнению таблицы 321 129 элементами, потому что 428 172 × 0.75 = 321 129, и сравнение вполне оправданно.

Таблица 3.4. Сравнение масштабируемых хеш-таблиц с таблицами фиксированного размера (время в миллисекундах)

M	Раздельные цепочки		Открытая адресация	
	Время заполнения	Время доступа	Время заполнения	Время доступа
625	0.783	0.173	1.050	0.197
1250	0.783	0.176	1.057	0.198
2500	0.787	0.177	1.053	0.198
5000	0.780	0.175	1.059	0.200
10 000	0.798	0.180	1.069	0.200
20 000	0.782	0.176	1.062	0.201
40 000	0.775	0.177	1.037	0.204
80 000	0.763	0.176	1.015	0.203
160 000	0.722	0.177	0.932	0.209
320 000	0.627	0.182	0.747	0.208
640 000	0.409	0.164	0.327	0.178
...	...	...	...	...
Фиксированное значение	0.378	0.172	0.434	0.285

В последнем ряду приведен «наихудший идеальный» случай: идеальный — потому что размер таблицы заранее подобран так, чтобы показатель загруженности не превысил `0.75` (стало быть, `resize()` не вызывается), а наихудший — потому что под конец заполнения таблицы он вплотную подходит к пороговому значению. Разумно ожидать, что в этом случае и время заполнения, и время доступа в таблице с открытой адресацией будут несколько больше аналогичных параметров в таблице с раздельными цепочками: в первом случае коллизии одного хеша в конце концов начинают неизбежно удлинять несколько разных цепочек, а во втором — только одну, которая этому хешу соответствует. А вот для «слишком

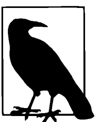
идеального» случая в предпоследней строке (с таблицей, заполненной примерно наполовину) может выиграть открытая адресация — за счет более простой структуры, которая требует линейно меньше операций. Что более важно: время доступа к N = 321 129 элементам динамической хеш-таблицы не зависит от ее начального размера M, так что отпадает необходимость магически предсказывать этот размер заранее.

Оценка производительности динамических хеш-таблиц

Уже говорилось, что чисто теоретически хеши всех ключей могут совпасть и угодить в одну цепочку (как при открытой адресации, так и для цепочек в списках). Как следствие, время выполнения операций `put()` и `get()` будет прямо пропорционально количеству элементов в хеш-таблице N, а значит, в *наихудшем случае* сложность этих операций следует оценить как $O(N)$.

Довольно частое требование к хеш-функции — *равномерное распределение* хешей на всем множестве значений. Если это свойство обеспечить, любой случайный ключ будет иметь равную вероятность попасть в любую цепочку. В идеале средняя длина отдельной цепочки должна быть равна N / M, это доказывается математически. Остается только надеяться на то, что специалист, который разработал для Python функцию `hash()`, предусмотрел и равномерное распределение в ней[1].

Раз уж у нас таблица динамически растет, причем N гарантированно меньше, чем M, можно уверенно утверждать, что среднее значение N / M — это константа, то есть $O(1)$, а стало быть, оно *не зависит* от N.



Если ключ есть в таблице, поиск завершается попаданием, а если нет — промахом. Предположим, хеш-функция равномерно распределяет значения, и оценим среднее количество обращений к ячейкам таблицы с открытой адресацией и пороговым значением alpha для обоих результатов. В случае попадания придется просмотреть в среднем (1 + 1 / (1 – alpha)) / 2 ячеек, для alpha = 0.75 получается 2.5. В случае промаха результат равен 1 + 1 / / $(1 - \text{alpha})^2$ / 2, то есть 8.5, при тех же условиях.

[1] Важно понимать, что худший случай так никуда и не делся! Уже говорилось, что до введения рандомизации в функцию hash() было возможно практически подобрать последовательность ключей с одинаковым хешем. Это, наверное, удастся даже сейчас, но при условии, что известна «соль» — случайное значение, которое подмешивается во все хеши конкретного экземпляра Python-интерпретатора, и есть весьма серьезный вычислительный ресурс для подбора этой последовательности. Случайно же получить худший случай уж совсем невероятно. — *Примеч. пер.*

Стоит понять, во что нам обходится динамическое изменение размера таблицы — допустим, с порогом загруженности 0,75 и геометрическим масштабированием (удвоением). Возьмем для начала $M = 1023$, а N — гораздо большим, чем M. К примеру, используем все тот же словарь из 321 129 английских слов. Попробуем посчитать, сколько раз мы добавляем пару в таблицу, в том числе в новую, внутри функции `resize()`.

Первый раз `resize()` вызовется при добавлении 768-го ключа (потому что $768 \geq 767.25 = 0.75 \times 1023$), и размер таблицы M станет равен $1023 \times 2 + 1$, то есть 2047. В процессе масштабирования нам придется заново добавить все 768 ключей (повторно вычисляя их хеши). Заметим, что в результате показатель загруженности упадет до 768 / 2047, то есть примерно до 0.375.

После добавления еще 768 ключей их количество опять подойдет к пороговому — 1536, и все пары будут перенесены (с повторным хешированием) в новую таблицу размером $M = 2047 \times 2 + 1$, то есть 4095. В третий раз масштабирование произойдет после добавления еще 1536 ключей, и придется добавлять уже 3072 пары в таблицу размером 8191.

Что эти числа нам говорят? Посмотрим на табл. 3.5: там показано количество слов N, которые надо добавить в таблицу, чтобы произошло масштабирование, а еще — каково будет после этого общее число операций добавления ключа в таблицу. После каждого перехеширования будет увеличиваться, а затем снова падать среднее количество операций добавления на один хранимый элемент. Эта характеристика — отношение общего числа добавлений к N — представлена в последнем столбце. В нашем случае она никогда не превышает 3 в силу геометрической природы масштабирования — и это несмотря на то, что при каждой смене размера нужно заново добавлять все ключи из старой таблицы в новую.

Главное, что понятно из этих данных: с увеличением размера таблицы геометрическое масштабирование происходит *значительно реже*. В последней строке показано, как с добавлением заключительного слова среднее количество операций становится равным 2.22, и при этом масштабирование не понадобится еще для 72 087 добавлений. Если первый интервал между операциями масштабирования составлял 768 добавлений, то девятый (196 608) уже в $2^8 = 256$ больше.

Если исследовать алгоритм геометрического масштабирования аналитически, картина не меняется. Для начала заметим, что ни размер таблицы, ни пороговый коэффициент загруженности на подсчет среднего количества добавлений не влияют. В самом деле, перед очередной операцией масштабирования в таблицу размером M уже добавлено $N = M \times 3/4$ ячеек в соответствии с пороговым значением. При масштабировании размер таблицы становится $2M$ (забудем на время про одну дополнительную ячейку), а новое пороговое значение — $2M \times 3/4 = 2N$.

При этом на операцию масштабирования тратится еще N добавлений, N ячеек в новой таблице оказываются заняты, и еще $2N - N = N$ ячеек туда можно добавить до следующего масштабирования. В следующий раз нам потребуется уже $2N$ добавлений, чтобы скопировать старую таблицу, и $2N$ добавлений останется в запасе.

Таблица 3.5. Слова, на добавлении которых происходит изменение размера таблицы, сколько всего для этого требуется добавлений и сколько раз в среднем добавлялось одно слово

Слово	M	N	Всего	В среднем
absinths	1023	768	1536	2.00
accumulatively	2047	1536	3840	2.50
addressful	4095	3072	8448	2.75
aladinist	8191	6144	17 664	2.88
anthoid	16 383	12 288	36 096	2.94
basirhinal	32 767	24 576	72 960	2.97
cincinnatian	65 535	49 152	146 688	2.98
flabella	131 071	98 304	294 144	2.99
peps	262 143	196 608	589 056	3.00
...	...	...	...	...
zyzzyvas	524 287	321 129	713 577	2.22

Итак, сколько добавлений выполнено *между этими двумя* операциями масштабирования? N — для новых элементов и $2N$ — при повторном хешировании. Вот и весь секрет «магического» числа 3, к которому приближается среднее количество операций на элемент в хеш-таблице! После первого же масштабирования на каждое добавление нового элемента приходятся дополнительно еще две операции при копировании. Если бы начинали сразу с *полупустой* таблицы, это отношение было бы равно строго 3. Но, поскольку мы начинаем с пустой таблицы размером M и, следовательно, имеем $M / 2$ «неучтенных» свободных ячеек, да еще после удвоения прибавляем дополнительную ячейку, отношение общего количества добавлений к количеству хранимых ключей в хеш-таблице с геометрическим удвоением всегда меньше 3.

В частности, добавление всех $N = 321\ 129$ слов в нашу динамически расширяемую хеш-таблицу медленнее добавления в неизменяемую таблицу подходящего для N размера *не более чем втрое*. Как мы уже договорились в главе 2, это всего

лишь мультипликативная постоянная и она не влияет на оценку сложности операции `put()` в среднем: даже несмотря на дополнительные затраты на масштабирование сложность остается O(1).

Динамические массивы

Мы уже несколько раз указывали на то, что тип `list` в Python реализован как массив; строго говоря — как *таблица*, то есть массив ссылок на объекты. Объекты Python могут быть разного размера, а вот ссылки — всегда одинакового. Наиболее распространен интерпретатор Python, написанный на C, так называемом CPython; в нем, как в C, ссылки — это просто адреса соответствующих структур в оперативной памяти. Доступа к этим ссылкам программист почти не имеет: в том же CPython функция `id(объект)` вернет число, соответствующее адресу объекта в памяти, но смысла в этом немного, ибо сделать с ним ничего нельзя, да и сам этот `id` не обязан быть уникальным. Например, если создать произвольный объект (скажем, строку), затем удалить его, потом создать объект другого типа (скажем, кортеж), есть шанс, что оба эти объекта получат одинаковый `id`, потому что будут каждый в свое время расположены по одному и тому же адресу оперативной памяти.

О свойствах массива рассказывалось в отдельном подразделе, замечу только, что созданием и удалением самих объектов, ссылки на которые содержит объект типа `list`, занимается исполняющая система Python и в конечном итоге операционная система. Сейчас нас будет интересовать другое очень полезное свойство типа `list`, которое отличает его от простого массива: его *размер можно менять*.

Последовательная организация оперативной памяти предполагает, что «классические» массивы могут иметь только заранее заданный размер. Скажем, в восьми идущих подряд ячейках памяти находится массив из восьми целых чисел A, а в следующих восьми — массив из четырех вещественных чисел двойной точности B. Добавить больше восьми элементов в массив A невозможно: дальше начинается стартовый элемент массива B, и если записать в `A[8]` целочисленное значение, то в `B[0]` окажется что-то совсем невообразимое. Тем не менее в списки Python можно добавлять сколько угодно элементов, и я указывал на то, что операция эта сравнительно дешевая — в среднем O(1).

Ничего не напоминает? Конечно же, ситуацию с переполнением хеш-таблицы, только в упрощенной форме! Допустим, объект `A` типа `list` занимает восемь ячеек памяти и все они уже заняты: `len(A)` равно `8`. Тогда операция `A.append(объект)` начнется с масштабирования массива, причем тоже геометрического и тоже с коэффициентом 2. Опуская подробности: будет выделен фрагмент памяти на 16 ячеек, туда скопировано содержимое восьми ячеек из старого фрагмента,

добавлено значение девятой ячейки (ссылка на *объект*), имя `A` начнет указывать на этот новый фрагмент, а старый будет освобожден. При этом `len(A)` будет возвращать `9`, и в `A` останется еще семь свободных мест для добавления новых объектов за константное время. Как и в случае динамической хеш-таблицы, операция добавления в динамический массив, достигший предела загруженности (здесь он просто равен 1), имеет линейную сложность. Мы уже знаем, что геометрическое масштабирование гарантирует нам O(1) в среднем, с мультипликативной постоянной 3.

Вот и весь секрет! Прежде чем двинуться дальше, остается сделать два важных замечания. Во-первых, мы не рассматривали операцию *удаления* ключа из хеш-таблицы, потому что она сравнительно редко требуется. Однако удаление элемента из динамического массива требуется сплошь и рядом, оно имеет в среднем линейную сложность (из-за «сдвига» элементов) и в среднем константную сложность при удалении последнего элемента, например, для `A.pop()`. Логично предположить, что при достижении некоторого *нижнего* порога загруженности (скажем, в четверть прежнего размера) нам захочется *уменьшить* размер массива. Это делается ровно тем же способом, геометрическим масштабированием, только в меньшую сторону, и ровно по этой причине также не влияет на константность операции `.pop()`.

Во-вторых, здесь кажется уместным ответить (или, скорее, *не ответить*) на вопрос «Как Python справляется с фрагментацией памяти?». Суть вопроса в следующем. Допустим, мы создали сто динамических массивов и они расположились в оперативной памяти, как и полагается, подряд. Затем в *каждый второй* из этих массивов мы добавили столько элементов, что произошло их масштабирование. При этом, как мы договорились, для этих 50 элементов была выделена новая память, а старая освобождена. В результате фрагмент памяти, в котором изначально лежали все 100 массивов, приобрел вид «массив, свободное место, массив, свободное место, массив...». Итак, вопрос: не захламляют ли такие структуры оперативную память и кто в случае Python заведует всем этим хозяйством? Ответ: разумеется, захламляют, и заведует этим исполняющая система Python, пользуясь программным интерфейсом операционной системы.

Эффективное управление памятью — серьезная, активно развивающаяся дисциплина IT, не имеющая ни идеального, ни даже просто устраивающего всех решения. Ей посвящены многие научные и практические труды. В книге мы не станем посягать на эту тему. Чтобы оценить масштаб проблемы, задумаемся еще вот над чем. Допустим, мы создали 100 обычных объектов Python, а потом просто удалили каждый второй. Не произойдет ли и здесь фрагментации памяти, и если да, то кто и что с ней будет делать? Ведь такое происходит, считай, постоянно? Изучение этого вопроса столь же занимательно, сколь и обширно, оно может окончательно увести в сторону от главной темы. Лучше вернемся к ней.

Идеальный хеш

Если заранее знать, чему будут равны все наши N ключей, можно подобрать то, что называется *идеальной хеш-функцией*, в которой каждому ключу взаимно однозначно соответствует уникальный хеш, и, стало быть, и попадание, и промах в хеш-таблице потребуют ровно одного обращения. Неожиданный, но довольно распространенный прием — по набору ключей ***автоматически изготовить фрагмент программы*** на произвольном языке программирования, в которой будет определена соответствующая хеш-функция.

В примере 3.11 приведен вариант такой функции `perfect_hash()`, сгенерированный сторонним модулем `perfect-hash` из файла, в котором в каждой строке расположен один ключ — слово из бессмертного пушкинского «Читатель ждет уж рифмы розы; / На, вот возьми ее скорей!» (без знаков препинания и прописных букв, зато с буквами «е»)[1]. Несущественную отладочную информацию и комментарии мы удалили.

Пример 3.11. Идеальный хеш для десяти пушкинских слов

```
G = [0, 10, 3, 10, 10, 8, 1, 0, 3, 5, 6]
S1 = [4, 10, 4, 6, 9, 7, 1, 6]
S2 = [7, 1, 6, 1, 6, 8, 8, 9]

def hash_f(key, T):
    return sum(T[i % 8] * ord(c) for i, c in enumerate(key)) % 11

def perfect_hash(key):
    return (G[hash_f(key, S1)] + G[hash_f(key, S2)]) % 11
```

Мы уже пользовались встроенной в Python функцией enumerate(), которая возвращает последовательность пар (индекс, элемент) для любой итерируемой последовательности. Строго говоря, исходная последовательность даже не обязана сама быть индексируемой, то есть конструкция *последовательность*[i] может быть синтаксически недопустимой. Но если существует возможность пройтись по ней циклом вида for *элемент* in *последовательность*, то всегда возможна конструкция вида for *индекс*, *элемент* in enumerate(*последовательность*). Переменная *индекс* при этом будет расти от 0 с шагом 1. Если при проходе циклом нужны и элементы, и их индексы, рекомендуется в любом случае использовать enumerate(), как в примере:

```
>>> for index, character in enumerate("QWE"):
...     print(index, character)
```

[1] Модуль можно установить из командной строки с помощью pip install perfect-hash. Текст в примере был сгенерирован тоже из командной строки примерно так: perfect-hash --trials=1000 --hft=2 файл_со_словами_в_столбик.txt. — *Примеч. пер.*

Вспомним, как на рис. 3.1 мы задали массив `day_array` и в дополнение к нему — функцию `base32()`, которая вычисляла хеш каждого из 12 месяцев. Тогда размер этого массива зависел от найденного нами числа — 35, остатки деления хешей каждого месяца на которое все оказываются различными. Тем самым мы задали идеальную для месяцев года хеш-функцию, формула которой — `day_array[base32(месяц)%35]`. Алгоритм из `perfect-hash` достигает того же эффекта, но существенно более остроумным способом. Не пытаясь описать методику *создания* полученных структур данных и функций[1], попробуем убедиться хотя бы в том, что это именно хеш и он идеален для наших ключей.

Итак, у нас есть массив `G`, размер которого (11) превосходит общее количество ключей $N = 10$, два вспомогательных массива `S1` и `S2` меньшей длины (ниже мы убедимся, что эта длина растет очень незначительно по сравнению с ростом N), вспомогательная хеш-функция `hash_f()` и собственно `perfect_hash()`, которая, поверим авторам на слово, будет выдавать идеальный хеш для заданного множества ключей. Чтобы вычислить хеш ключа *«уж»*, необходимо сначала получить два промежуточных значения:

- `hash_f('уж', S1) = (S1[0] * ord('у') + S1[1] * ord('ж')) % 11`, то есть (4 × 1091 + + 10 × 1078) % 11 = 15 144 % 11 = 8 (1091 и 1078 — порядковый номер букв «у» и «ж» в таблице Unicode);
- `hash_f('уж', S2) = (S2[0] * ord('у') + S2[1] * ord('ж')) % 11`, то есть (7 × 1091 + + 1 × 1078) % 11 = 8715 % 11 = 3.

Функция `perfect_hash('уж')` вычисляет `(G[8] + G[3]) % 11`, то есть (3 + 10) % 11 = 2. Таким образом, хеш ключа «уж» равен 2. Повторим упражнение на ключе «вот»:

- `hash_f('вот', S1)` = (4 × 1074 + 10 × 1086 + 4 × 1090) % 11 = 19 516 % 11 = 2;
- `hash_f('вот', S2)` = (7 × 1074 + 1 × 1086 + 6 × 1090) % 11 = 15 144 % 11 = 8;
- `perfect_hash('вот')` = (G[2] + G[8]) % 11 = (3 + 3) % 11 = 6.

Продолжая в том же духе, получим, что хеш «уж» — это 2, хеш «вот» — 6, и вообще хеш каждого слова из «читатель ждет уж рифмы розы на вот возьми ее скорей» уникален. Математика[2] временами творит настоящие чудеса!

В примере 3.12 приведена функция `perfect_hash()`, которую сгенерировал этот модуль для нашего тренировочного словаря из 321 129 слов. На самом первом английском слове — *a* — функция возвращает `0`, а на последнем, *zyzzyvas* — **`321128`**. Сам массив `G`, содержащий 667 596 промежуточных ключей, я здесь, понятное

---

[1] Ссылка на серьезную математическую статью и упрощенное описание алгоритма подбора идеальной хеш-функции есть в документации к модулю. — *Примеч. пер.*

[2] В данном случае — теория графов и теория чисел. — *Примеч. пер.*

дело, не привожу, а вот два массива S1 и S2 для вычисления идеального хеша все еще невелики.

Пример 3.12. Часть текста программы с идеальным хешем для словаря английских слов

```
S1 = [394429, 442829, 389061, 136566, 537577, 558931, 481136, 337378,
      395026, 636436, 558331, 393947, 181052, 350962, 657918, 442256,
      656403, 479021, 184627, 409466, 359189, 548390, 241079, 140332]

S2 = [14818, 548808, 42870, 468503, 590735, 445137, 97305, 627438,
      8414, 453622, 218266, 510448, 76449, 521137, 259248, 371823,
      577752, 34220, 325274, 162792, 528708, 545719, 333385, 14216]

def hash_f(key, T):
        return sum(T[i % 24] * ord(c) for i, c in enumerate(key)) % 667596

def perfect_hash(key):
        return (G[hash_f(key, S1)] + G[hash_f(key, S2)]) % 667596
```

Читатель может самостоятельно изготовить полный пример: установить Python-модуль `perfect-hash` и выполнить команду `python3 -m perfect_hash --hft=2 words.english.txt`, используя файл `words.english.txt` из репозитория с программами к книге. Поскольку при наполнении служебных массивов модуль использует случайный выбор, числа в получившемся примере будут другие и длина `G` наверняка будет слегка отличаться.

Промежуточная функция `hash_f()` генерирует довольно большое число — сумму значений из `S1` или `S2`, — далее это число обрезается с помощью остатка от деления на 667 596, что дает индекс каждой из половин хеша в большом массиве `G`. После того как половины складываются и снова обрезаются до 667 596, получается уникальный ключ для любого допустимого английского слова из нашего словаря в диапазоне от 0 до 667 595. В действительности все еще интереснее: алгоритм рассчитан так, что `perfect_hash(`*слово*`)` возвращает индекс этого *слова* во входном списке, то есть число от 0 до 321 128, и это значение (в отличие от содержимого `S1`, `S2` и `G`) не меняется при повторной генерации текста функции.

Если попробовать внезапно посчитать хеш чего-то, что не является словом из исходного словаря, может возникнуть коллизия. В примере 3.9 слово *watered* и не слово *not-a-word* получали одинаковый хеш; в только что сгенерированном примере на *not-a-word* может не быть коллизии, но она (в силу того что всевозможных не слов гораздо больше, чем 667 595) обязательно появится на каких-то других парах «*слово — не слово*». Интересная задача — отыскание таких пар для конкретного варианта `perfect_hash()`. В этом нет никакой ошибки: за то, что нашей функции будут предъявляться только слова из исходного словаря, отвечает программист.

Проход таблицы циклом

Основная задача хеш-таблицы — эффективные операции `get(k)` и `put(k, v)`. Но еще было бы неплохо иметь доступ ко всем ячейкам таблицы, независимо от того, какой цепочке они принадлежат и какой подход — открытая адресация или раздельное хранение — используется для их хранения.

Генератор, или вычислимая последовательность, — один из самых удобных инструментов Python. В современном языке программирования для последовательного просмотра некоторого набора данных не нужно задействовать отдельную память со списком этих данных. В главе 2 вы уже видели, что вычислимые последовательности range(0, 1000) и range(0, 100000) занимают одинаковый объем памяти, порождая соответствующие последовательности целых чисел. Генератор-функция — общая форма задания вычислимой последовательности в Python.

Вот такая генератор-функция изготовит и вернет вычислимую последовательность целых неотрицательных чисел от 0 до *n*, в десятичном представлении которой нет заданной цифры (пример 3.13).

Пример 3.13. Задание генератор-функции

```
{{{#highlight py3
def avoid_digit(n, digit):
  sd = str(digit)for i in range(n):
    if not sd in str(i):
      yield i
}}}
```

Анализируя синтаксис описания функции, Python обнаружит в ней оператор `yield`, и это будет значить, что это генератор-функция, которая при вызове возвращает собственно генератор. Генератор-функцию надо вызвать *один раз*, получить генератор, а по генератору — пройтись циклом. Когда мы первый раз в цикле обращаемся к генератору, выполняются строки от начала функции до первого встреченного `yield`. В примере — от строки 2 до строки 5, при этом в цикл возвращается выражение, указанное параметром `yield` (в примере — `i`). В следующий раз выполнение *продолжается непосредственно после* выполненного `yield` и до следующего `yield`. Здесь `yield` — последняя строка блока цикла, так что выполнение продолжается со строки 3 до строки 5, затем снова возвращается `i`, и так до тех пор, пока вычисляется цикл. Выход за пределы функции (в примере — когда `i` сравняется с *n*) означает останов цикла, использующего генератор.

Посмотрим, как работает наша `avoid_digit()` в командной строке Python (пример 3.14).

Пример 3.14. Работа генератор-функции

```
>>> gen = avoid_digit(15, "3")            ❶
>>> gen                                   ❷

>>> for element in gen:                   ❸
...     print(element, end=" ")
... print()

>>> for element in gen:                   ❹
...     print(element, end=" ")
... print()
```

❶ Создадим генератор и назовем его `gen`.

❷ Еще раз убедимся в том, что `avoid_digit()` возвращает генератор, а не, скажем, целое число.

❸ Поскольку генератор — это последовательность, его элементы можно пройти циклом `for`; цикл закончится, когда закончится сама последовательность.

❹ Если попробовать еще раз пройтись циклом по тому же самому генератору, выяснится, что он *уже закончился*, и ни одной итерации не произойдет[1].

Если в классе Python задать специальный метод `.__iter__()`, сконструированный из этого класса объект станет *итерируемым* — его можно будет использовать в конструкции вида `for элемент in объект`. В примере 3.15 приведена реализация прохода циклом хеш-таблицы с открытой адресацией, а в примере 3.16 — реализация прохода циклом хеш-таблицы с раздельным хранением цепочек в списках.

Пример 3.15. Метод-итератор по всем элементам хеш-таблицы с открытой адресацией, реализованный как генератор-функция

```
class Hashtable:
    def __iter__(self):
        for entry in self.table:
            if entry:                         ❶
                yield entry.key, entry.value  ❷
```

❶ Пропустим все пустые (содержащие `None`) ячейки.

❷ Вернем в `yield` кортеж (*`ключ, значение`*).

[1] По этой причине генераторы можно считать «одноразовыми» и не хранить их в переменных, а сразу писать что-то вроде for element in avoid_digit(15, "3"). — *Примеч. пер.*

Пример 3.16. Метод-итератор по всем элементам хеш-таблицы с раздельным хранением цепочек

```
class Hashtable:
    def __iter__(self):
        for chain in self.table:                ❶
            for entry in chain:                 ❷
                yield entry.key, entry.value    ❸
```

❶ Цикл по цепочкам.

❷ Цикл по ячейкам в цепочке.

❸ Пара (*ключ, значение*) в качестве возвращаемого значения.

Рассмотрим последовательности пар, порождаемые итераторами в трех случаях: проходом по хеш-таблице размером $M = 13$ с открытой адресацией, по таблице размером 13 с раздельным хранением цепочек и по идеальной хеш-таблице на основе слов английского языка. После добавления в них слов из строки *a rose by any other name would smell as sweet* порядок выдачи пар при проходе циклом будет различен. В табл. 3.6 показан результат для всех трех случаев.

Таблица 3.6. Порядок слов, возвращаемых итераторами хеш-таблиц

Открытая адресация	Раздельное хранение цепочек	Идеальный хеш
a	sweet	a
by	any	any
any	a	as
name	would	by
other	smell	name
would	other	other
smell	as	rose
as	name	smell
sweet	by	sweet
rose	rose	would

Порядок слов в двух наших реализациях выглядит случайным, а для идеальной хеш-функции — словарным. Конечно, никакого случайного выбора из множества ключей мы не писали: дело в алгоритме вычисления хеша, который использует Python-функцию `hash()` с ее равномерным распределением. Более того, если

запустить соответствующий пример из репозитория самостоятельно, последовательность слов в этих двух случаях изменится, ибо современный `hash()` в Python случайно непредсказуем.

Слова в выдаче третьего итератора идут в словарном порядке, потому что при формировании идеальной хеш-функции `perfect_hash(ключ)` исходный список слов был словарно упорядочен. Напомню одно из полезных свойств модуля `perfect-hash`: порождаемый этой функцией хеш — это *номер слова во входном словаре*. Так что, если словарь был упорядочен и ячейки в таблицу записываются в порядке значения хеша, итератор вернет пары тоже в словарном порядке. Еще одно важное следствие этого свойства — хеш-таблица на основе идеального хеша имеет размер, строго равный количеству элементов в исходном словаре, и не нуждается в разрешении коллизий.

В последней, восьмой, главе вы поближе познакомитесь с типом данных `dict` — ассоциативными массивами Python (или *словарями*). Всякий раз, когда в программе нужна хеш-таблица, надо использовать `dict`, а не наши самодельные классы, потому что `dict` и намного быстрее, и больше умеет.

Заключение

В этой главе мы рассмотрели несколько важных понятий.

- Примитивным хеш-таблицам, которые хранят пары (`ключ, значение`) в массиве ячеек размером M, не хватает алгоритма разрешения коллизий — ситуации, когда хеш двух и более ключей приводит к одной и той же ячейке.
- Открытая адресация ключа позволяет рассредоточить хранимые пары по всему массиву, но тогда для действительно эффективной работы нужно, чтобы размер M этого массива был как минимум вдвое больше максимального количества элементов N. Удаление элемента из такой хеш-таблицы — это целое упражнение (в тренировочных заданиях есть два таких упражнения).
- Цепочки можно хранить раздельно, в списках. В этом случае совпадение хешей для двух и более ключей приводит к добавлению их в один и тот же список, а основной массив таблицы оказывается массивом этих списков. Удалить элемент из такой таблицы проще.
- Самому придумать хеш-функцию непросто, но в Python есть встроенная функция `hash()`, ее и стоит использовать.
- Геометрическое масштабирование ассоциативного массива (хеш-таблицы) позволяет сохранить среднее быстродействие операции добавления за счет того, что «тяжелая» процедура увеличения размера с перехешированием выполняется все реже с увеличением размера.

- Тип данных Python `list`, который мы используем для списков, — по сути массив с геометрическим масштабированием при добавлении, то есть динамический массив. Это задает константную, O(1), сложность операциям добавления в конец списка.
- Идеальное хеширование — функция, возвращающая уникальный хеш для каждого элемента из заданного множества, — возможно при условии, что это множество заранее определено. Более того, если это множество как-либо упорядочить, можно использовать алгоритм, в котором хеш элемента совпадает с его порядковым номером. Вычислительная сложность и сложность по памяти идеальной хеш-функции чаще всего выше стандартной `hash()`.

Тренировочные задания

1. Улучшается ли работа *открытой адресации*, если вместо *последовательного просмотра* с шагом 1 и возвратом к началу избрать другую тактику разрешения коллизий? Сделайте размер массива равным степени двойки, а шаг просмотра — переменным, равным *n*-му элементу из последовательности так называемых *треугольных чисел*. После просмотра соседней ячейки (расстояние +1) исследуются ячейки на расстоянии 3, 6, 10, 15, 21 и т. д. (с учетом возврата к началу при выходе за границы). Треугольное число под номером n вычисляется по формуле $n \times (n + 1) / 2$.

   Будет ли производительность такой таблицы лучше, чем при последовательном просмотре?

   Проведите эксперимент. Добавьте в хеш-таблицу размером 524 288 половину словаря английских слов (160 564 слова — это наполнение примерно на треть) и измерьте время поиска всех 321 129 слов. Сравните результаты эксперимента для хеш-таблицы с раздельным хранением, для таблицы с открытой адресацией и последовательным просмотром и для новой таблицы с переменным шагом.

2. Имеет ли смысл сортировать ключи в раздельно хранимых цепочках хеш-таблицы? Напишите вариант хеш-таблицы с раздельным хранением, в которой ячейка в цепочку добавляется так, чтобы `(ключ, значение)` оставались упорядоченными по возрастанию ключа.

   Проведите эксперимент. Создайте хеш-таблицу на 524 287 элементов (524 287 — простое число) и добавьте в нее первую половину словаря — 160 564 английских слова (загруженность будет примерно на треть). Если добавлять слова в порядке возрастания, это сделает линейной сложность поиска нужного места для вставки пары, но константной — сложность вставки (потому что она будет фактически соответствовать добавлению в конец).

Если добавлять слова в порядке убывания – наоборот. Какой из этих вариантов окажется *наихудшим случаем*? Сравните его производительность с производительностью хеш-таблиц того же размера с открытой адресацией и несортированными раздельными цепочками.

Теперь измерьте время поиска первых 160 564 английских слов в этой таблице. Можно ожидать, что это будет *наилучший случай*, потому что все эти слова уже есть в таблице. Сравните производительность с производительностью аналогичных операций на хеш-таблицах с открытой адресацией и несортированными раздельными цепочками. Затем поищите оставшиеся 160 565 слов из второй половины словаря. Это должен оказаться *наихудший случай*, потому что поиск несуществующих слов в списке требует его полного просмотра. Снова сравните производительность с производительностью двух других типов таблиц. Насколько результаты сравнения связаны с индексом загруженности хеш-таблицы? Например, что изменится, если задать размер таблицы 214 129 (загруженность 75 %) или 999 983 (загруженность 16 %)?

3. Чтобы убедиться в опасности предсказуемой хеш-функции, посмотрите на пример 3.17. Класс `BadString` в примере порождает объекты, схожие во всем с типом Python `str`. Но работа встроенной функции `hash()` для них *перегружена*, потому что определен специальный метод `.__hash__()`. В результате `hash()` для таких объектов может быть равен только `0`, `1`, `2` или `3`.

Пример 3.17. Ужасная реализация хеш-функции

```
class BadString(str):
    def __hash__(self):
        return hash(str(self)) % 4
```

Создайте хеш-таблицу с раздельно хранимыми цепочками на 50 000 элементов и вызовите `.put(BadString(w), 1)` для первых 20 000 английских слов из нашего словаря. Сделайте еще одну такую же хеш-таблицу и добавьте туда те же слова непосредственно, без преобразования в `BadString`. Посчитайте:

- среднюю длину цепочки;
- наибольшую длину цепочки.

Имейте в виду, что работать это будет довольно долго! Объясните полученные результаты.

4. Число ячеек в хеш-таблице, M, на практике выгодно делать простым. Если взять для примера числовой ключ и остаток от деления на M в качестве индекса в таблице, окажется, что *ключ, имеющий общий делитель с M, попадает только в ячейку, индекс которой имеет тот же общий делитель с M*. Скажем, пускай $M = 632 = 8 \times 79$, а добавляем мы пару с ключом $2133 = 27 \times 79$; хеш этого ключа будет равен $2133 \% 632 = 237 = 3 \times 79$. Видите 79? Неприятность

в том, что вместо равномерного распределения по всем ячейкам, от которого зависит быстродействие хеш-таблицы, получается, что некоторые ключи будут попадать строго в ограниченный набор ячеек.

Исследуйте влияние M на результат. По аналогии с `base32()` напишите функцию `base26()`, которая каждому слову английского словаря поставит в соответствие числовой ключ. Создайте 101 хеш-таблицу размером M, где M будет меняться от 428 880 до 428 890 включительно, добавьте в каждую такую таблицу все 321 129 английских слов по их числовому ключу `base26()` и сравните средние и наибольшие длины цепочек. Повторите это на хеш-таблицах с открытой адресацией и на хеш-таблицах с раздельными цепочками. Нет ли среди результатов *некоторой общей закономерности*? Сформулируйте гипотезу и попробуйте воспользоваться ею для поиска наихудшего M среди последующих 10 000 размеров (до 438 890). Для этого длина M наибольшей цепочки должна быть почти вдесятеро больше.

5. Из хеш-таблицы с открытой адресацией нельзя просто так удалить произвольную пару — если пометить ячейку как пустую, проходившая *через нее* цепочка прервется и последовательный просмотр не позволит добраться до продолжения этой цепочки. Допустим, у нас есть хеш-таблица на пять элементов, в которую мы добавили ключи 0, 5 и 10. В результате ключи в таблице разместятся так: `[0, 5, 10, None, None]`, потому что с каждой коллизией просмотр будет добираться до следующей ячейки. Если сейчас удалить `5`, получится таблица с ключами `[0, None, 10, None, None]`, в которой внезапно образовались *две* цепочки, причем ключ 10 найти в ней нельзя, потому что цепочка для хеша 0 теперь состоит из всего одного элемента.

   Один из путей решения проблемы — ввести специальное логическое поле в классе `Entry`, в котором будет отмечено, что эта ячейка удалена, но является частью цепочки. Напишите метод `.remove(ключ)`, удаляющий элемент по ключу. Что изменится в работе методов `.get()` и `.put()`? Не забудьте также поправить процедуру масштабирования и метод `.__iter__()` — они должны пропускать удаленные ячейки.

   При таком подходе понадобится также обратное масштабирование — уменьшение размера массива ячеек в случае, когда более половины из них помечены как удаленные. Сравните быстродействие хеш-таблицы с открытой адресацией и удалением и хеш-таблицы с раздельными цепочками (в которых удаление уже реализовано).

6. Масштабирование хеш-таблицы весьма серьезно понижает производительность: требуется повторно хешировать все N ключей и заполнить N ячеек новой таблицы. В *поэтапном* масштабировании хранятся обе таблицы — старая, размером M, и новая, размером $2M + 1$. Вызов `get(ключ)` сначала ищет ключ

в новой таблице, а если его там нет — в старой. Вызов `put(ключ, значение)` всегда добавляет пару в новую таблицу, после чего D раз берет пару из старой таблицы, повторно хеширует ключ, добавляет ее в новую, а из старой удаляет. Когда старая таблица опустеет, ее можно удалить[1].

Реализуйте поэтапное масштабирование для хеш-таблиц с раздельным хранением цепочек и исследуйте различные значения D. В частности, каково минимальное D, при котором к следующему масштабированию старая таблица гарантированно опустеет? Постоянна ли величина D или она зависит от M либо даже от N?

Такой подход должен снизить максимальную сложность операции `.put()`. Проведите эксперимент, составив таблицу, аналогичную табл. 3.5, в которой на этот раз сравните производительность самых неэффективных операций.

7. Обратное масштабирование можно реализовать и в обычной хеш-таблице с удалением. При вызове метода `.remove()` надо проверить, что таблица загружена не более чем на $1/4\ M$, и если да — запустить масштабирование к половинному размеру. Это позволит освободить неиспользуемую память. Доработайте любой из вариантов хеш-таблицы с удалением элемента, добавив в него обратное масштабирование, и проведите несколько испытаний на предмет того, стоит ли результат затраченных усилий.

8. Используйте хеш-таблицу, чтобы найти элемент списка, который встречается в нем не реже любого другого. Если таких несколько, можно возвращать любой из наиболее повторяющихся элементов.

   Например, `most_duplicated([1, 2, 3, 4])` может вернуть `1`, `2`, `3` или `4`, а вот `most_duplicated([1, 2, 1, 3])` обязан вернуть `1`.

9. Еще один способ удалить пару из хеш-таблицы с открытой адресацией — повторно хешировать все пары `(ключ, значение)`, *которые следуют в цепочке за удаляемой парой*. Добавьте в хеш-таблицу с открытой адресацией такой способ удаления и проведите несколько экспериментов, сравнивая производительность хеш-таблицы с раздельным хранением цепочек с производительностью хеш-таблицы с открытой адресацией и удалением цепочки описанным способом[2].

---

[1] Скорее всего, она удалится сама при следующем масштабировании. — *Примеч. пер.*

[2] Не забудьте сначала убедиться, что реализованное таким способом удаление сохраняет работоспособность вашей структуры. — *Примеч. пер.*

ГЛАВА 4

Могучая куча

В этой главе

- Два абстрактных типа данных — *очередь* и *приоритизированная очередь*.
- Изобретенная в 1964 году структура данных — *двоичная куча*, которую можно хранить в обычном массиве.
- *Двоичная куча с порядком убывания*, в которой элемент, имеющий *наибольшее числовое значение* приоритета, считается элементом с наивысшим приоритетом, и *двоичная куча с порядком возрастания*, в которой наивысшим считается приоритет с минимальным числовым значением.
- Как добавить пару (*значение, приоритет*) в двоичную кучу за $O(\log N)$ операций, где N — количество элементов в куче.
- Как и где в двоичной куче найти элемент с наивысшим приоритетом за $O(1)$ операций.
- Как снять с вершины двоичной кучи элемент с наивысшим приоритетом за $O(\log N)$ операций.

Что, если мы будем хранить не просто набор значений, а набор пар, в котором с каждым *значением* будет сопоставлен некий числовой *приоритет*? Будем считать, что чем выше приоритет, тем важнее для нас значение, так что на этот раз нам нужно уметь только добавлять пару (*значение, приоритет*) и получать из нашей структуры *одно значение с наивысшим приоритетом, одновременно удаляя его оттуда* (это называется операцией *снятия*).

Мы сейчас описали работу так называемой приоритизированной очереди — структуры данных, в которой эффективно реализованы две операции: добавления

пары `enqueue(`*`значение, прироритет`*`)` и снятия (удаления + возврата) значения с наивысшим приоритетом *`значение`* `= dequeue()`. Приоритизированная очередь не похожа на уже изученные нами в предыдущей главе хеш-таблицы: *нам не нужно заранее знать приоритет*, чтобы снять наиболее приоритетное значение.

Допустим, ночной клуб переполнен и у входа уже выстроилась очередь, как на рис. 4.1. Для того чтобы попасть внутрь, каждый вновь пришедший должен встать в конец очереди. Первым попадет в клуб человек из начала очереди — он ждал дольше всех. Так работает собственно *очередь* — структура данных, в которой предусмотрена операция добавления последнего появившегося значения в конец очереди, `enqueue(`*`значение`*`)`, и операция снятия первого значения по времени добавления, `dequeue()`. Иными словами, принцип работы очереди — «первым вошел, первым вышел» (First in, first out, или FIFO), что в развернутом виде можно прочесть как «(элемент, который) первым вошел (в очередь), первым вышел (из нее, как только к ней обратились)».

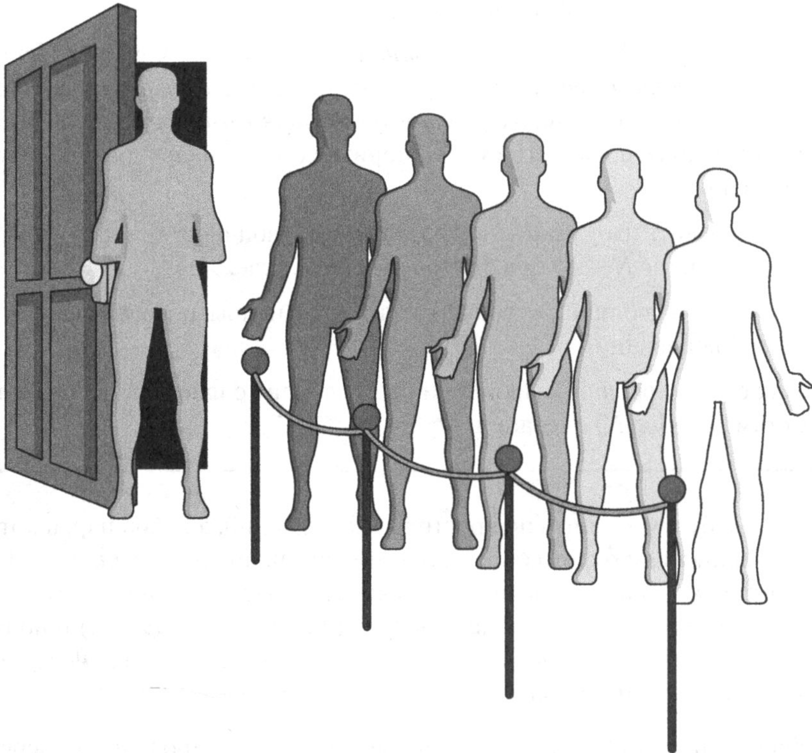


Рис. 4.1. Ночной клуб. Ожидание в очереди

Упомянутую в предыдущей главе структуру данных — связный список — можно было бы смоделировать на Python довольно просто: каждый элемент списка — это пара (*значение, следующий элемент*). Весь список — это «матрешка» таких пар, в каждой из которых, кроме последней, поле `.next` — это следующая пара, а в последней `.next` равен `None`.

```
class Node:
  def __init__(self, val):
    self.value = val
    self.next = None
```

Из таких узлов можно выстроить и очередь. На рис. 4.2 показан результат добавления в клубную очередь «Ивана», «Ирины» и «Игоря» (именно в таком порядке). «Иван» окажется первым, кого снимут из очереди, тогда в ней останется два посетителя, из которых первой окажется «Ирина».

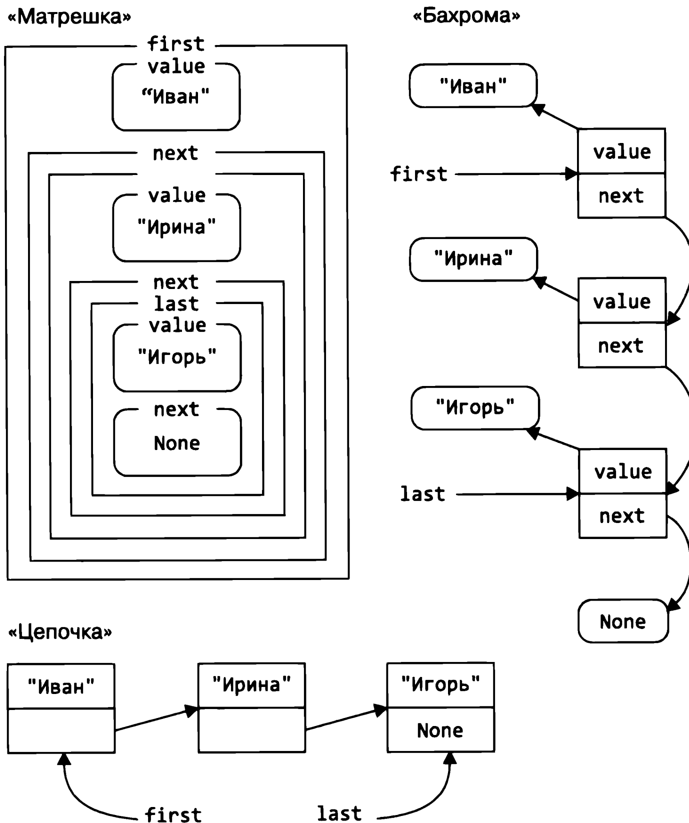


Рис. 4.2. Три варианта представления очереди в ночной клуб

Уровень абстракции и модель памяти языка Python позволяют нам относиться к связным спискам и как к «матрешкам», в которых весь «хвост» списка вложен в его первый элемент, и как к «бахроме», в которой каждый объект представлен изолированно, а в узлах хранятся только ссылки на эти объекты. Действительности, однако, соответствует именно вторая модель, «бахрома»: поэтому, например, фактический размер узла в списке (элемента типа `Node`) не зависит от размера поля `.value`. Поскольку обе модели тяжелы для восприятия, впредь мы будем пользоваться их упрощенным гибридом, «цепочкой», в которой явными ссылками представлены только элементы списка, а все остальные данные проще показывать как хранящиеся внутри соответствующих структур. Сами такие структуры мы время от времени будем называть *узлами*.

В примере 4.1 класс `Queue` имеет два метода — `.enqueue()` для добавления элемента в конец связного списка и `.dequeue()` для снятия элемента из его начала. Обоим методам нужно константное время для работы, независимо от количества элементов в очереди.

Пример 4.1. Реализация очереди при помощи связного списка

```
class Queue:
  def __init__(self):                        ❶
    self.first = None
    self.last = None

  def is_empty(self):
    return self.first is None                ❷

  def enqueue(self, val):
    if self.first is None:                   ❸
      self.first = self.last = Node(val)
    else:
      self.last.next = Node(val)             ❹
      self.last = self.last.next

  def dequeue(self):
    if self.is_empty():
      raise RuntimeError('Queue is empty')
    val = self.first.value                   ❺
    self.first = self.first.next             ❻
    return val
```

❶ Поначалу и `first`, и `last` — это `None`.

❷ Если `first` — пусто, очередь считается пустой.

❸ Если очередь пуста, первый добавляемый элемент и есть последний.

❹ Если очередь не пуста, добавляем элемент после текущего `last`; на этот новый элемент `last` должен указывать в дальнейшем.

❺ Значение, которое нужно вернуть, хранится в `first`.

❻ Первый элемент исключается из списка, и `first` теперь указывает на следующий (или равен `None`, если такого не было).

Поменяем условия. Допустим, ночной клуб стал продавать своим завсегдатаям входные билеты произвольной стоимости, кто сколько заплатит, и проставлять эту стоимость на выданном билете. К примеру, один может купить билет за 50 долларов, а другой — за 100. Люди все так же встают в очередь, когда клуб переполнен, но первым в него войдет тот, *чей билет дороже всех остальных билетов в очереди*. Если самые дорогие билеты оказываются у нескольких человек, очередь на вход предоставляется одному из них. Безбилетники считаются обладателями бесплатного билета.

На рис. 4.3 в середине очереди есть завсегдатай с самым дорогим — стодолларовым — билетом. Если сейчас двери откроются, он первым попадет в клуб. За ним последуют двое с билетами за 50 долларов (в не установленном пока порядке), следом — безбилетники; они тоже считаются равноправными (владельцами бесплатных билетов), так что их порядок также не определен.

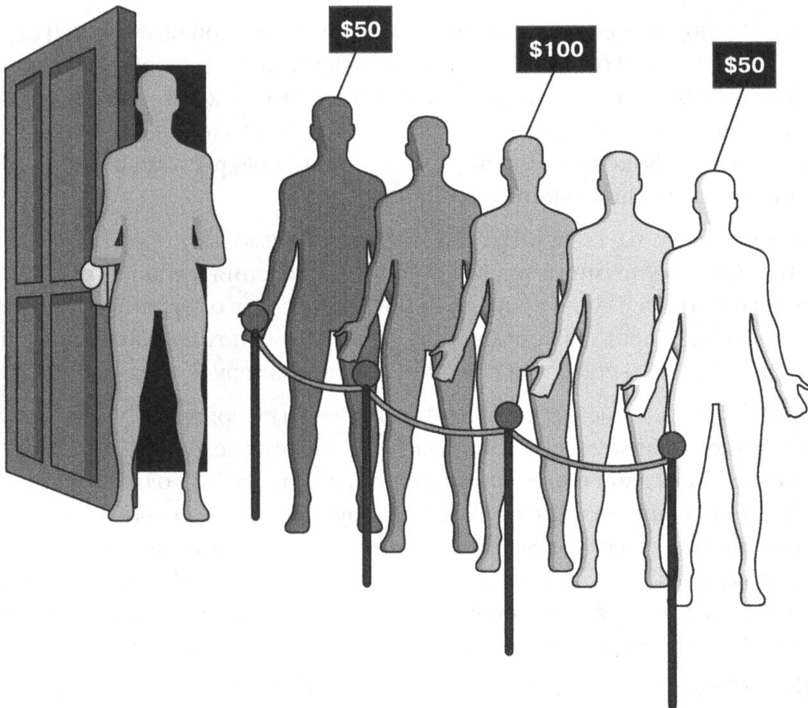
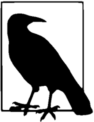


Рис. 4.3. Войдет первым тот, кто больше заплатит



В приоритизированной очереди не задается, что делать с двумя и больше значениями, приоритеты которых совпадают. В действительности приоритизированная очередь может возвращать такие значения не в порядке добавления — это зависит от реализации. Вариант на основе кучи, описанный в этой главе, возвращает элементы с одинаковым приоритетом не в том порядке, в котором их добавляли. Встроенный модуль heapq, о котором мы поговорим в главе 8, тоже использует кучу.

Исправленные таким образом требования определяют абстрактный тип данных — *приоритизированную очередь*. Правда, в ней уже нельзя реализовать обе операции — `enqueue()` и `dequeue()` — за константное время. С одной стороны, если воспользоваться связным списком, то `enqueue()` останется O(1), а вот `dequeue()` придется просматривать все элементы в поисках значения с наивысшим приоритетом, так что в *худшем случае* сложность ожидается O(*N*). С другой стороны, если всегда держать элементы в очереди упорядоченными по приоритету, `dequeue()` окажется O(1), а `enqueue()` в *худшем случае* потребует O(*N*) просмотров в поисках правильного места для вставки в список.

По опыту, есть пять сравнительно простых способов организовать из набора объектов типа `Entry` приоритизированную очередь пар (*`значение, приоритет`*).

- *Array*. Неупорядоченный массив элементов без всякой внутренней структуры — может, и так сойдет. Операция `enqueue()` занимает константное время, а `dequeue()` мало того, что занимается поиском по всему массиву, так еще и вынуждена переставлять его элементы после удаления одного. Вдобавок массив имеет фиксированный размер, так что очередь может еще и переполниться. В Python такого типа данных нет.
- *Built-in*. Вместо массива в Python можно использовать тип `list` и воспользоваться встроенными методами поиска, добавления и удаления элементов. Если элементы в нем не упорядочивать, сложность операций останется той же, что и при использовании массива, разве что мультипликативная константа будет меньше и снимется ограничение по размеру.
- *OrderA*. Массив, элементы которого упорядочены по приоритету. Необходимый для `enqueue()` поиск места для вставки элемента по такому массиву можно ускорить с помощью **двоичного поиска** (описан в примере 2.4), однако мы все равно вынуждены копировать часть массива вручную, чтобы освободить нужную ячейку посреди других элементов. При этом `dequeue()` может иметь константную сложность, так как в массиве, упорядоченном по приоритетам, нужный элемент оказывается всегда в конце. Массив имеет фиксированный размер, так что очередь может еще и переполниться. В Python такого типа данных нет.
- *Linked*. Связный список, элементы которого всегда расположены по убыванию приоритета: начальный элемент имеет наивысший приоритет, все остальные — меньший либо равный приоритету предыдущего элемента.

В этом варианте в операции `enqueue()` линейную сложность имеет поиск места для вставки элемента, а сама вставка и операция `dequeue()` константны.

- *OrderL*. Элементы хранятся в порядке приоритетов в динамическом массиве типа `list` от Python. Для поиска в `enqueue()` можно снова использовать **двоичный поиск**, но сама вставка (встроенными в `list` методами) требует перемещения в среднем половины элементов массива и оттого линейна. Операция `dequeue()` ожидаемо константна, потому что элемент с наибольшим приоритетом находится в конце массива.

Мы сравнили производительность всех описанных способов, для чего провели испытания, в которых благополучно вызвали $3N / 2$ операций `enqueue()` и $3N / 2$ операций `dequeue()`. Для каждого способа измерялось отношение общего времени работы к количеству операций, $3N$, в итоге получалось среднее время выполнения одной операции. Результаты, показанные в табл. 4.1, говорят следующее. Имитация «массива», в которой все действия реализованы явно, оказалась самым медленным вариантом, а использование встроенных методов `list` ускорило тест почти вдвое. Упорядоченное хранение элементов в «массиве» снова почти удвоило скорость, использование связных списков ускорило тест еще примерно на 20 %; однако реализация `OrderL` посредством упорядоченных списков типа `list` и их встроенных методов оказалась вне конкуренции.

Таблица 4.1. Среднее время операции в наносекундах для объема данных размером N

N	Heap	OrderL	Linked	OrderA	Built-in	Array
256	6.4	2.5	3.9	6.0	8.1	13.8
512	7.3	2.8	6.4	9.5	14.9	26.4
1024	7.9	3.4	12.0	17.8	28.5	52.9
2048	8.7	4.1	23.2	33.7	57.4	107.7
4096	9.6	5.3	46.6	65.1	117.5	220.3
8192	10.1	7.4	95.7	128.4	235.8	446.6
16 384	10.9	11.7	196.4	255.4	470.4	899.9
32 768	11.5	20.3	—	—	—	—
65 536	12.4	36.8	—	—	—	—

Так или иначе, для всех упомянутых способов среднее время либо `enqueue()`, либо `dequeue()` прямо пропорционально N. А вот в столбце, помеченном как *Heap*, числа ведут себя по-другому: среднее время выполнения пропорционально $\log(N)$. Этот столбец содержит результаты эксперимента над структурой данных `Heap` **(кучей)**; на рис. 4.4 хорошо видно, насколько она эффективнее нашего

победителя — упорядоченных списков Python. Хороший эмпирический признак логарифмической сложности — константное приращение времени выполнения при удвоении объема входных данных. В табл. 4.1 объем входных данных как раз удваивается в каждой следующей строке — и среднее время работы одной операции с кучей увеличивается примерно на 0,8 наносекунды.

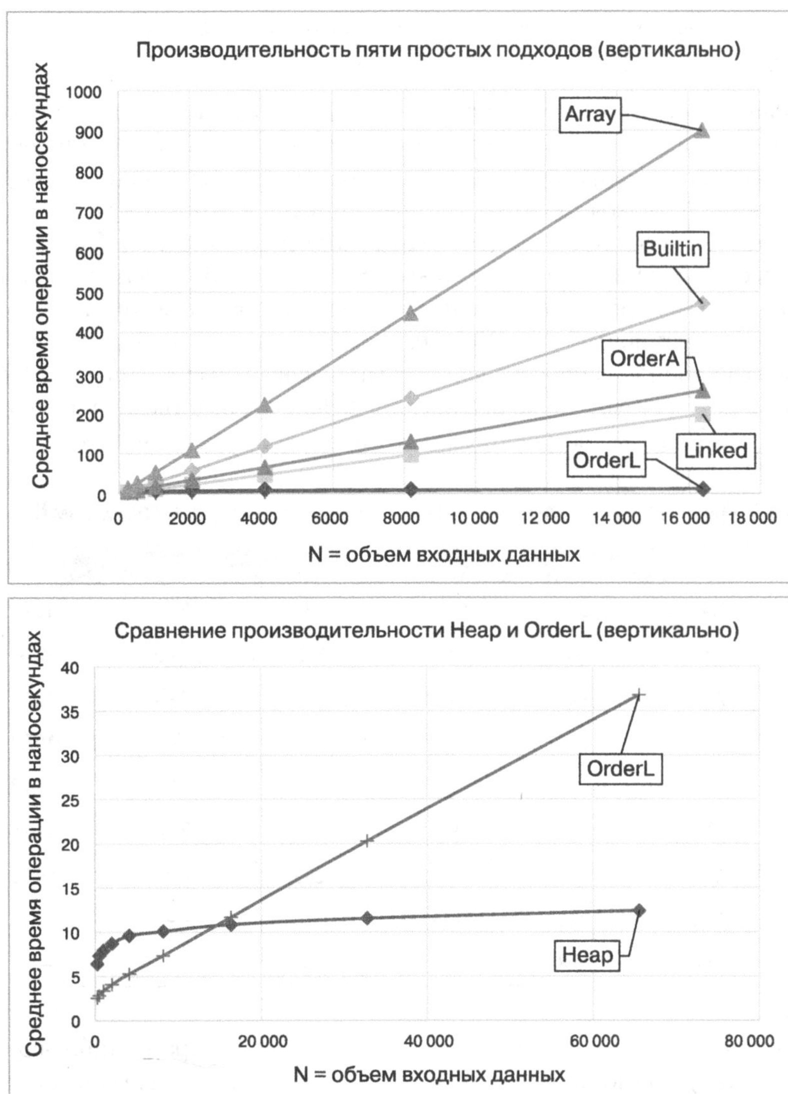


Рис. 4.4. Логарифмическая сложность операций над кучей куда эффективнее линейной сложности других подходов

Простейший вариант кучи — двоичную кучу — изобрел для своего алгоритма сортировки валлийско-канадский математик Дж. В. Дж. Вильямс (в русскоязычной литературе этот алгоритм обычно называют **пирамидальной сортировкой**). С помощью кучи можно добиться логарифмической сложности операций над приоритизированной очередью. В этой главе мы не станем больше обращать внимания на тип поставленного в очередь объекта: это могут быть строки, числа или вообще картинки — значения не имеет. Нам будет важен только числовой приоритет, который хранится в паре с объектом. Так что на всех рисунках, где для простоты показан только приоритет, подразумевается и объект, который с этим приоритетом поставлен в очередь. В куче с порядком убывания у того из двух объектов приоритет выше, у кого он больше.

В нашем варианте куча будет иметь заранее заданный наибольший размер, M, и сможет хранить $N < M$ элементов. Теперь изучим ее структуру, посмотрим, как она растет (в рамках наибольшего размера) и убывает в процессе работы, а также научимся хранить N элементов кучи в обычном массиве.

Двоичная куча

Не вполне понятно зачем, но предположим, что наши данные отсортированы только частично. Например, на рис. 4.5 изображена двоичная куча с частичным порядком — убыванием приоритетов[1]; у каждого элемента показан только приоритет. На уровне 0 находится единственный элемент, и приоритет его будет наивысшим среди всех элементов кучи. Если два элемента x → y связывает стрелка, то приоритет x больше приоритета y либо равен ему.

Частично упорядоченная по убыванию куча — это несортированный список, в ней, например, нельзя сразу найти элемент с наименьшим приоритетом (если что, он на уровне 3). Зато у нее есть иные полезные свойства. На первом уровне кучи находится два элемента, приоритет одного из которых — непременно наивысший для всех уровней, кроме приоритета элемента на уровне 0 (а не то равен и ему). Любой уровень k — *кроме последнего* — содержит 2^k элементов; иными словами, он *полон*. Неполным может быть только самый низкий уровень (в нашем примере там два из возможных 16 элементов); заполняются уровни слева направо. Приоритет неуникален: в куче из примера приоритеты 8 и 14 встречаются более одного раза.

---

[1] В русскоязычной терминологии не принято явно указывать, какой именно частичный порядок установлен в куче, обычно это понятно из контекста. Поэтому, а также во избежание путаницы (ибо куча с порядком убывания у автора называется max heap) мы будем использовать просто термин «куча», а порядок указывать, только если возможна неоднозначность. — *Примеч. пер.*

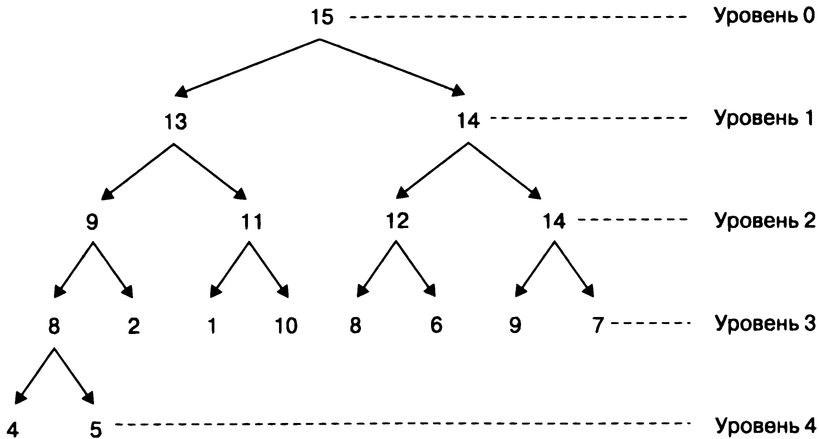


Рис. 4.5. Пример двоичной кучи с частичным убыванием

В *двоичной куче* из каждого узла может исходить не более двух стрелок. Возьмем узел на уровне 0, приоритет которого равен 15. Он связан стрелками с двумя узлами на уровне 1: узел с приоритетом 13 – это его *левый потомок*, а узел с приоритетом 14 – *правый потомок*. Для узлов на первом уровне узел на нулевом уровне (с приоритетом 15) является соответственно *родительским*.

Рассмотрим структурные свойства двоичной кучи.

- *Частичный порядок*. Приоритет узла не меньше приоритетов его левого и правого потомков (если они есть). Приоритет любого узла (кроме того, что на вершине кучи) не больше приоритета его родительского узла.
- *Пирамидальность*. Каждый уровень k кучи, кроме последнего, содержит ровно 2^k узлов, последний может содержать меньше, заполняется он слева направо, и до тех пор, пока он неполон, уровень $k + 1$ не появляется.

Если в двоичной куче только один элемент, в ней только один уровень – нулевой, то $2^0 = 1$, пирамидальность соблюдена. Сколько уровней понадобится, чтобы хранить $N > 0$ элементов? Нужна математическая формула, которая по заданному N позволит вычислить необходимое количество уровней в куче. С помощью рис. 4.6 можно попробовать догадаться, что это за формула. Расположим в виде пирамиды 16 узлов и пронумеруем их согласно принципу пирамидальности: на вершине – e_1, затем последовательно по каждому уровню с увеличением индекса на 1. Получится четыре полных уровня и единственный узел на пятом.

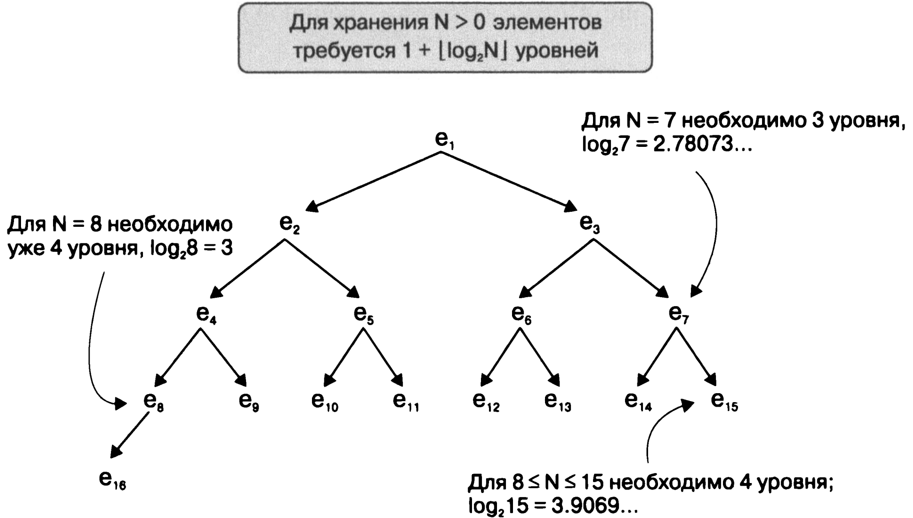
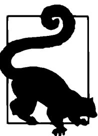


Рис. 4.6. Сколько уровней в куче из N узлов?

Пока в куче только семь узлов, e_1–e_7, они помещаются на трех уровнях. Начиная с восьми узлов уже нужно четыре уровня. Если идти по стрелкам сверху налево, нам встретятся узлы e_1, e_2, e_4, e_8 и e_{16} — очевидно, степень двойки влияет не только на размер уровня, но и на размер всей кучи. В формуле оценки количества слоев логично ожидать двоичный логарифм от N — если быть более точным, количество слоев L для хранения N элементов в двоичной куче равно $L(N) = 1 + \lfloor \log_2 N \rfloor$ (неполные квадратные скобки здесь означают округление до целого числа вниз, аналогичная функция Python — `math.floor()`).



В каждом следующем уровне двоичной кучи помещается больше узлов, чем во всех предыдущих, вместе взятых! В двоичной куче из L полных уровней содержится 2L – 1 элемент. В самом деле, в куче из одного уровня 21 – 1 = 1 узел, в куче из двух уровней — 22 – 1 = 3 узла; предположим, что в куче из L – 1 уровней 2L – 1 – 1 узел — тогда в куче из L уровней с добавлением еще 2L – 1 элемента из очередного слоя окажется 2L – 1 – 1 + + 2L – 1 = 2(2L – 1) – 1 = 2L – 1 узел, и формула индуктивно доказана. Как следствие, если к куче добавить всего один слой, она вместо N элементов будет способна хранить 2N + 1 элемент.

Что же касается двоичных логарифмов, то мы помним правило: при удвоении N логарифм увеличивается на 1, или в виде формулы: $\log_2 2N = 1 + \log_2 N$.

Какие из вариантов на рис. 4.7 соответствуют двоичной куче с порядком убывания?

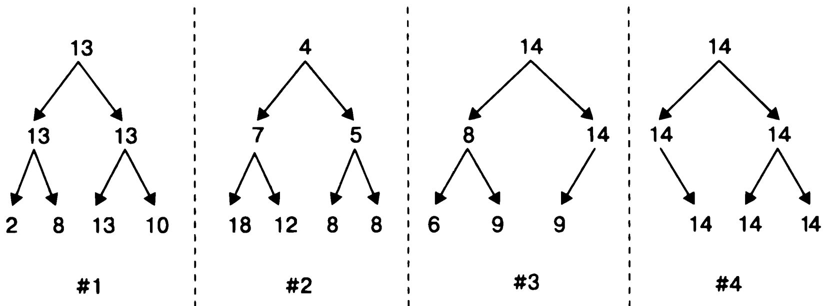
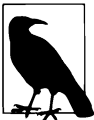


Рис. 4.7. Что из этого можно назвать двоичной кучей с убыванием?

Сначала проверим свойство пирамидальности. Варианты 1 и 2 подходят, потому что в них заполнены все уровни. Вариант 3 тоже подходит, потому что в нем неполон только последний уровень, и заполнен он правильно: содержит три узла из допустимых четырех, и это три самых левых узла. А вот в варианте 4 пирамидальность не соблюдается: последний уровень содержит три узла, при этом самый левый узел пуст.

Теперь проверим частичный порядок. В случае убывания приоритет каждого родителя должен быть не меньше приоритетов его потомков. Это верно для варианта 1, в чем можно убедиться, сравнив все начала и концы стрелок. Вариант 3 не годится, потому что у узла с приоритетом 8 есть потомок с приоритетом 9. Вариант 2 не годится, потому что уже на уровне 0 приоритет узла равен 4, а у обоих его потомков он выше.



В действительности вариант 2 — это пример двоичной кучи с убыванием, в которой приоритет родительского узла не превосходит приоритеты потомков. Этот вариант двоичной кучи мы рассмотрим в главе 7.

Операции добавления элемента в кучу `enqueue()` и снятия с кучи элемента с наивысшим приоритетом `dequeue()` должны сохранять оба ее свойства. Если удастся их реализовать так, чтобы производительность обеих операций оценивалась как $O(\log N)$, мы получим значительное улучшение быстродействия по сравнению с медленными моделями из табл. 4.1, в которых либо `enqueue()`, либо `dequeue()` в *наихудшем случае* показывали быстродействие $O(N)$.

Добавление пары в кучу

Если задаться целью добавить пару (*значение, приоритет*) в двоичную кучу, соблюдая *только свойство пирамидальности*, то в каком конкретно месте функции `enqueue()` придется заводить узел? Ответ всегда одинаков.

- Если последний уровень неполон, заполняется самый левый пустой узел этого уровня.
- Если последний уровень полон, заводится новый уровень и заполняется самый левый его узел.

На рис. 4.8 показан свежедобавленный в кучу узел с приоритетом 12: он занял третье место на уровне 4. Свойство пирамидальности соблюдено — все уровни, кроме последнего, полны, а последний заполнен слева. Однако во многих случаях частичный порядок от такого добавления нарушается.

К счастью, это дело поправимое, причем достаточно переставить местами лишь узлы «по дороге» от вершины кучи к только что добавленному элементу. На рис. 4.10 показана куча с восстановленным частичным порядком; можно заметить, что для этого пришлось переставить в порядке невозрастания узлы отмеченного серым *пути* от вершины до крайнего левого узла на последнем уровне.

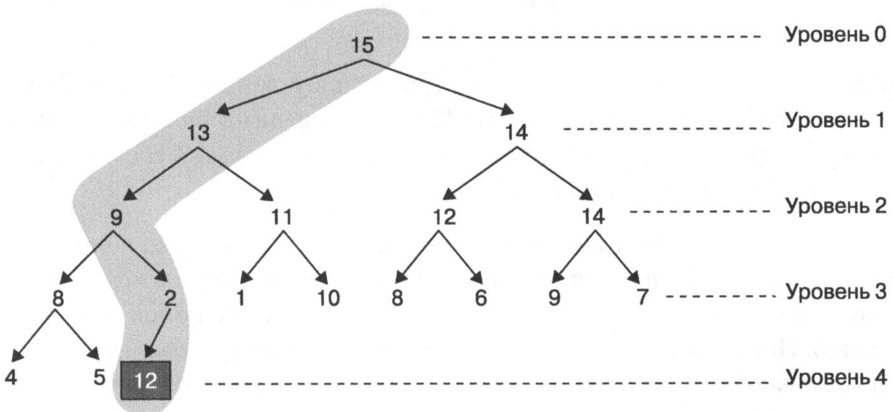
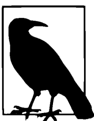


Рис. 4.8. Добавление элемента. Начальный шаг — заполняется первый свободный узел



Путь до заданного узла двоичной кучи — это последовательность узлов и стрелок вправо или влево между ними от единственного узла на уровне 0 до заданного.

Теперь надо позаботиться о восстановлении частичного порядка. Добавленный элемент начинает всплывать вдоль *пути* — если порядок нарушен, он и его родитель меняются местами. В примере на рис. 4.8 свежедобавленный узел с приоритетом 12 нарушает порядок, потому что его приоритет выше 2 — приоритета родителя. Два узла меняются местами; результат показан на рис. 4.9. Между тем *всплытие продолжается*.

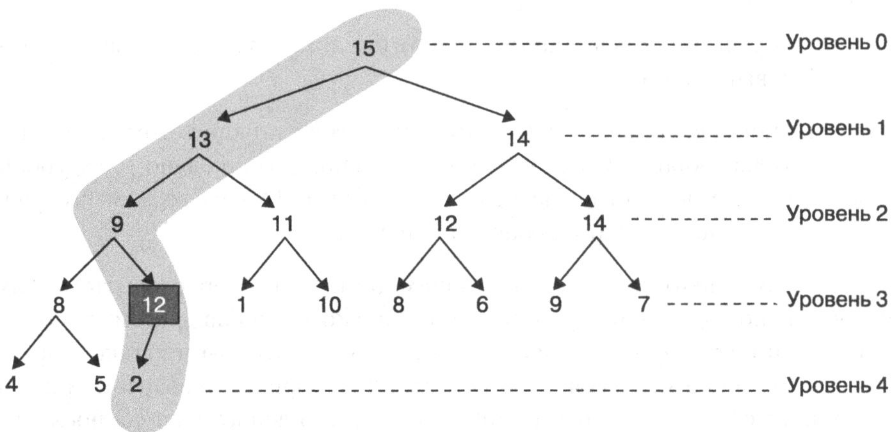


Рис. 4.9. Добавление элемента. Второй шаг — если нужно, узел всплывает на уровень выше

После первого всплытия структура с вершиной в узле 12 будет полноценной двоичной кучей из двух элементов. Но 12 все еще нарушает частичный порядок, потому что приоритет родителя этого узла — 9, а это меньше 12. Надо всплывать еще выше, как показано на рис. 4.10.

После очередного всплытия структура с вершиной в узле 12 будет полноценной двоичной кучей. Меняя местами 9 и 12 (по правой стрелке), мы можем не проверить, соблюдаются ли свойства кучи в структуре с вершиной 8 (по левой стрелке). Поскольку *раньше все элементы этой структуры были не больше 9*, они и подавно будут меньше 12. После третьего шага приоритет родительского узла, 13, оказывается больше 12 — частичный порядок восстановлен.

Попробуем вручную отработать вызов `enqueue(`*`значение`*`, 16)` на куче с рис. 4.10. Сначала узел добавляется в четвертую ячейку уровня 4 в качестве правого потомка узла с приоритетом 9. Затем узел всплывает все выше и выше вплоть до уровня 0. В результате получается двоичная куча как на рис. 4.11.

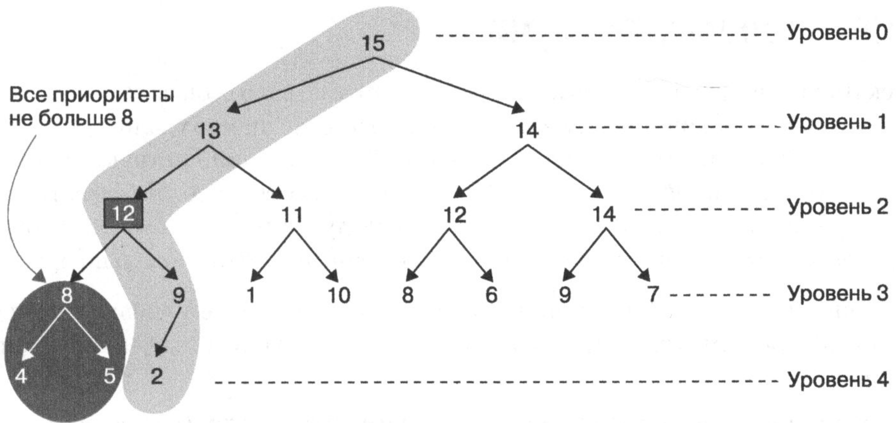


Рис. 4.10. Добавление элемента. Третий шаг — если нужно, узел всплывает на уровень выше

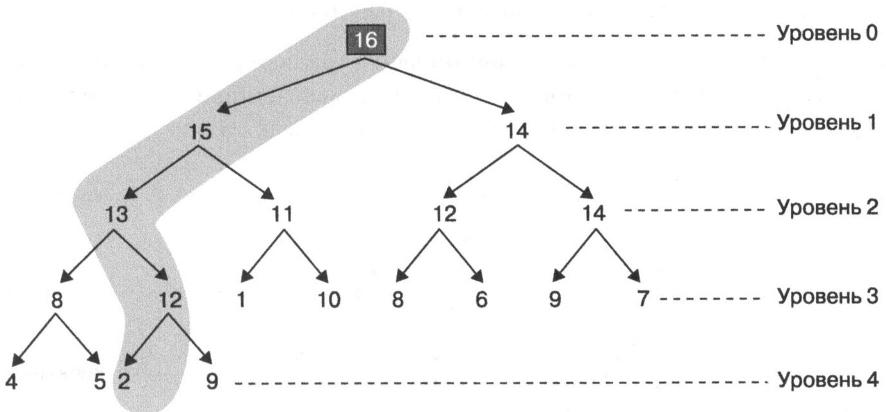


Рис. 4.11. Элемент с приоритетом 16 после добавления всплывает на вершину кучи

Наихудший случай для операции добавления — когда добавляется элемент с приоритетом, превышающим все имеющиеся приоритеты в куче. Длина пути при добавлении равна количеству уровней в куче, $1 + \lfloor \log_2 N \rfloor$, а это значит, что *наибольшее возможное количество операций обмена* будет на одну меньше — $\lfloor \log_2 N \rfloor$. Теперь уже можно уверенно оценивать и процедуру восстановления пирамидальности, и всю операцию `enqueue()` как $O(\log N)$. Большое дело — но это только половина решения всей задачи: теперь надо убедиться в том, что операцию снятия с кучи элемента с наивысшим приоритетом можно реализовать так же эффективно.

Снятие элемента с кучи

Искать в куче элемент с наивысшим приоритетом не надо — он всегда находится на ее вершине, в единственном узле уровня 0. Но, если просто удалить его, нарушится свойство пирамидальности — уровень 0, не последний в структуре, будет содержать пустой узел. К счастью, для dequeue() тоже есть способ эффективной перестройки кучи, как мы это увидим на последующих рисунках. Если сохранять *только свойство пирамидальности*, удаление выглядит довольно просто.

1. Запомним самый последний (самый правый в последнем уровне) элемент кучи и удалим его. Получившаяся структура сохранит все свойства двоичной кучи.
2. Запомним значение элемента с наивысшим приоритетом (с уровня 0) — это значение должна вернуть функция dequeue().
3. Заменим верхний элемент кучи с уровня 0 элементом, который мы удалили из нижнего уровня и запомнили на первом шаге. Это сохранит пирамидальность, но, скорее всего, нарушит частичный порядок.

На примере рис. 4.12: сначала мы запоминаем и удаляем из кучи элемент 9; куча при этом остается кучей. Затем запоминаем возвращаемое значение, приоритет которого наивысший, 16; на рисунке это действие не показано.

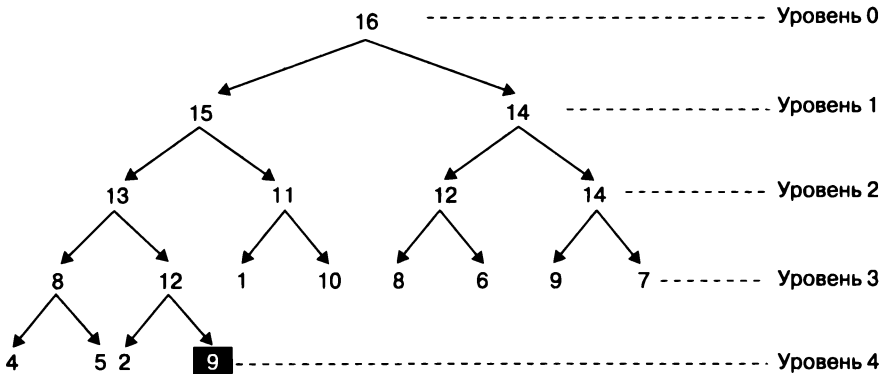


Рис. 4.12. Снятие элемента. Удаляем последний узел

На втором шаге мы подменяем узел на уровне 0 только что удаленным узлом. Как это понятно из рис. 4.13, частичный порядок в куче нарушится. Мы видим, что приоритет единственного узла на уровне 0 меньше и приоритета его левого потомка, 15, и приоритета его правого потомка, 14. Чтобы порядок восстановился, узел должен начать тонуть, погружаясь вглубь кучи вплоть до отведенного ему места.

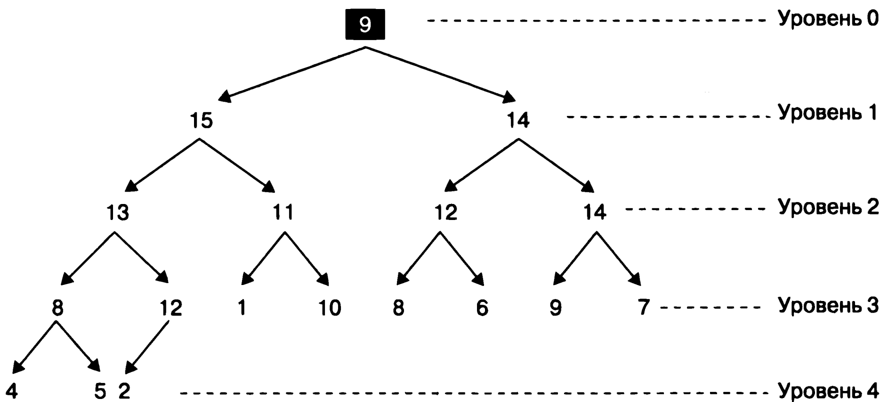


Рис. 4.13. Снятие элемента. Подставляем последний элемент вместо верхнего, порядок при этом нарушается

Поскольку наш узел нарушает порядок (сейчас на уровне 0 оказалась пара с приоритетом 9), надо решить, какой из узлов кучи необходимо поставить на его место. Очевидно, это должен быть узел с наивысшим приоритетом среди тех, что *доступны по стрелкам* на уровнях ниже нашего. Выбор придется делать среди всего двух узлов — правого и левого потомков (приоритеты остальных заведомо не выше). Если правого потомка нет, то остается только левый. В нашем примере мы выбираем левого потомка, чей приоритет равен 15, и это больше, чем 14, приоритет правого потомка. Меняем местами наш узел и выбранного потомка с наивысшим приоритетом, получаем состояние, показанное на рис. 4.14.

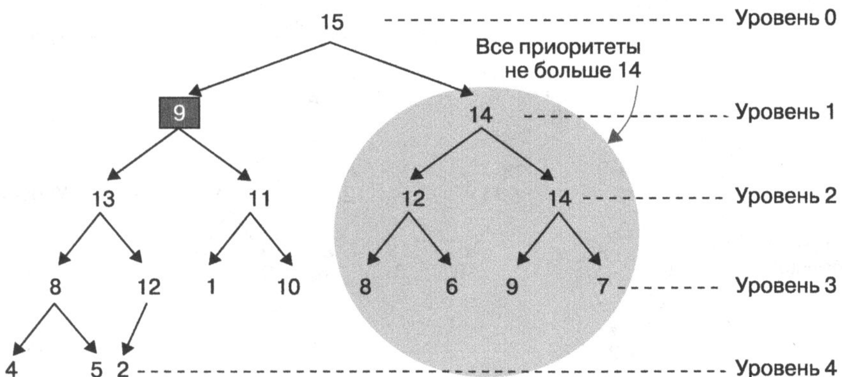


Рис. 4.14. Снятие элемента. Меняем местами верхний узел и его левого потомка, приоритет которого выше

Можно заметить, что вся структура, начинающаяся с узла с приоритетом 14 на уровне 1, — это полноценная двоичная куча; мы ее не трогали, и менять там ничего не надо. А вот в структуре под узлом с приоритетом 9, который оказался на уровне 1 после обмена, порядок нарушен — оба его потомка имеют более высокий приоритет. Значит, наш узел будет продолжать тонуть, меняясь с левым потомком (как это показано на рис. 4.15), потому что приоритет левого потомка 13 выше, чем приоритет правого.

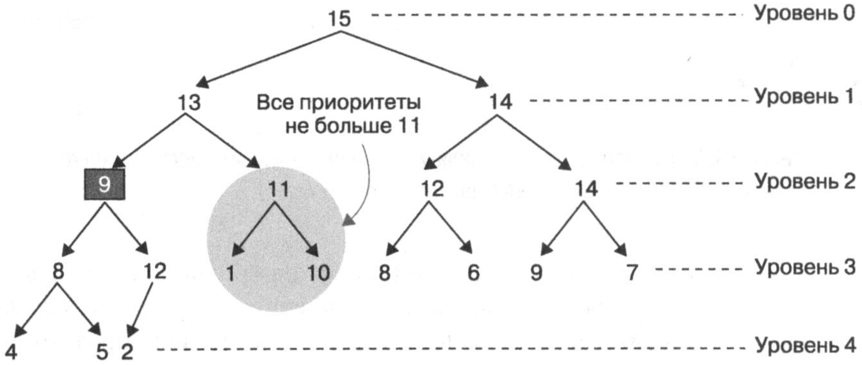


Рис. 4.15. Снятие элемента. Узел продолжает тонуть

Осталось немного! На рис. 4.15 видно, что в новом положении у нашего узла с приоритетом 9 есть правый потомок, а приоритет оставшихся — 12, так что мы меняем эти два узла, после чего частичный порядок в нашей куче восстанавливается, как это видно на рис. 4.16.

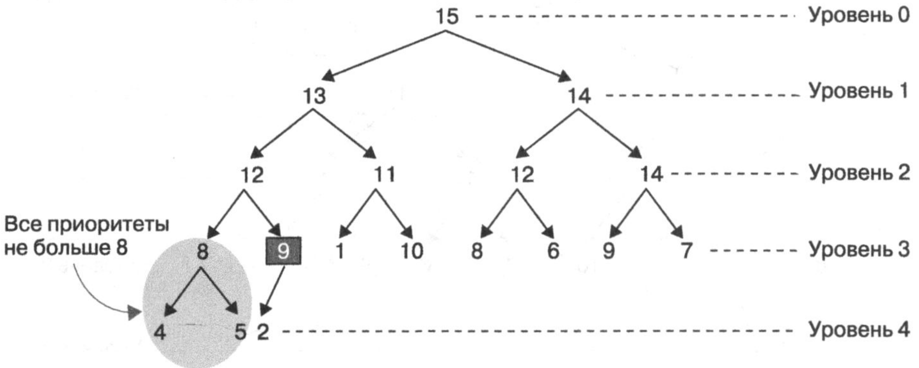


Рис. 4.16. Снятие элемента. Куча после того, как узел погрузился на нужный уровень

В отличие от операции добавления элемента в приоритизированную очередь, в операции снятия нельзя просто предсказать путь до будущего положения тонущего узла. Однако узел тонет с уровня на уровень, так что наибольшее возможное количество погружений не больше, чем переходов между уровнями в двоичной куче. Это число на единицу меньше, чем самих уровней, иными словами, $\lfloor \log_2 N \rfloor$.

Еще можно посчитать, сколько раз сравниваются приоритеты двух узлов за одну операцию снятия. Каждое погружение требует максимум двух сравнений: выбора наибольшего приоритета среди двух потомков и сравнения этого приоритета с приоритетом родителя. Таким образом, можно ожидать, что сравнений будет не больше $2 \times \lfloor \log_2 N \rfloor$.

Мы сделали очень важное предположение: и добавление элемента в двоичную кучу, и снятие элемента с ее вершины занимают время, в *наихудшем случае* прямо пропорциональное $\log N$. Пора переходить от теории к практике, а заодно посмотреть, как можно хранить двоичную кучу в обычном массиве.

Можно заметить, что пирамидальность кучи позволяет перечислить все элементы, если начать перебор с уровня 0 вниз, а по уровню двигаться слева направо. Этой особенностью можно воспользоваться: оказывается, элементы двоичной кучи вполне удобно хранить в обычном массиве подряд по уровням.

Хранение двоичной кучи в массиве

На рис. 4.17 показан один из способов записать двоичную кучу размером $N = 18$ в массив фиксированного размера $M > N$. В такой куче будет пять уровней, каждый из которых будет соответствовать определенному диапазону индексов массива, содержащего хранимые пары. Пунктирные квадраты в правой стороне рисунка отмечают индекс, по которому хранится соответствующий узел двоичной кучи. Как обычно, при изображении кучи мы показываем только приоритеты, а значения опускаем.

Итак, каждой хранимой паре сопоставляется индекс в массиве `storage[]`. Чтобы не упражняться лишний раз с вычитанием и добавлением единицы к этому индексу, оставим `storage[0]` пустым и ничего там хранить не будем. Элемент с наивысшим приоритетом, 15, хранится в `storage[1]`. Согласно рис. 4.17, его левый потомок, с приоритетом 13, хранится в `storage[2]`. В общем случае, если у элемента `storage[k]` есть левый потомок, он будет храниться в `storage[2*k]` (в этом легко убедиться, проследив стрелочки между пунктирными квадратами на рис. 4.17). Если у элемента `storage[k]` есть правый потомок, он будет храниться в `storage[2*k+1]` соответственно.

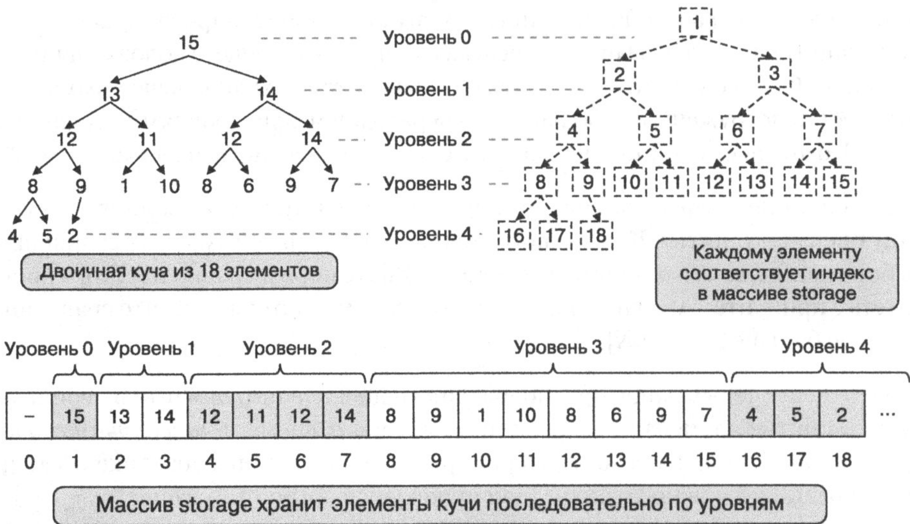


Рис. 4.17. Запись двоичной кучи в массив

Если $k > 1$, родительский узел для `storage[k]` всегда находится в `storage[k//2]` (в Python k // 2 – это целая часть от деления k на 2). Если располагать вершину кучи в `storage[1]`, то, чтобы вычислить индекс родителя любого узла, надо просто целочисленно поделить на 2 индекс узла-потомка. В примере родительский узел для `storage[5]` (с приоритетом 11) окажется в `storage[2]`, потому что 5 // 2 = 2.

Если в куче содержится N элементов, `storage[k]` для любого $0 < k \leq N$ содержит узел из двоичной кучи. Как следствие, если $2k > N$, то у k-го узла нет потомков; например, у узла `storage[10]` (с приоритетом 1) потомков нет, потому что $2 \times 10 = 20 > 18 = N$. А у узла `storage[9]` (с приоритетом, так случайно вышло, тоже 9) нет только правого потомка, потому что $2 \times 9 = 18 = N$, а вот $2 \times 9 + 1 = 19 > N$.

Как погружаться и всплывать

Начнем программировать нашу двоичную кучу с частичным убыванием с задания класса для пары (***значение, приоритет***):

```
class Entry:
  def __init__(self, v, p):
    self.value = v
    self.priority = p
```

В примере 4.2 для хранения узлов кучи применяется массив `storage`. При создании экземпляра класса `PQ` размер `storage` на единицу превосходит максимальный размер кучи — параметр `size` конструктора — потому что, напомню, мы храним элементы кучи в массиве, начиная с `storage[1]`.

Чтобы можно было лучше изучить работу нашего класса, мы написали два вспомогательных метода. Метод `.less(i, j)` проверяет, что приоритет *i*-го узла меньше приоритета *j*-го, и возвращает `True` или `False`. Когда нам понадобится оценить, сколько раз мы сравниваем приоритеты двух узлов, достаточно будет просто посчитать количество вызовов `.less()`. Метод `swap(i, j)` меняет местами *i*-й и *j*-й узлы в массиве, он нам понадобится для подсчета количества обменов при перестройке кучи, когда узел всплывает или тонет.

Пример 4.2. Реализация кучи методами .enqueue() и .swim()

```
class PQ:
  def less(self, i, j):                                        ❶
    return self.storage[i].priority < self.storage[j].priority

  def swap(self, i, j):                                        ❷
    self.storage[i], self.storage[j] = self.storage[j], self.storage[i]

  def __init__(self, size):                                    ❸
    self.size = size
    self.storage = [None] * (size+1)
    self.N = 0

  def enqueue(self, v, p):                                     ❹
    if self.N == self.size:
      raise RuntimeError('Priority Queue is Full!')
    self.N += 1
    self.storage[self.N] = Entry(v, p)
    self.swim(self.N)

  def swim(self, child):                                       ❺
    while child > 1 and self.less(child//2, child):            ❻
      self.swap(child, child//2)                               ❼
      child = child // 2                                       ❽
```

❶ Метод `.less(i, j)` проверяет, что `storage[i]` имеет меньший приоритет, чем `storage[j]`.

❷ Метод `swap(i, j)` меняет местами *i*-й и *j*-й узел.

❸ Узлы хранятся в ячейках от `storage[1]` до `storage[size]`, а `storage[0]` не используется.

❹ Чтобы добавить элемент в кучу, надо разместить его в первой свободной ячейке массива, а затем позволить ему всплыть.

❺ Метод `.swim()` перестраивает массив `storage`, возвращая куче пирамидальность.

❻ Родительский узел для `storage[child]` находится в `storage[child//2]` (`child//2` — это целочисленное деление `child` на 2).

❼ Меняем местами `storage[child]` и его родителя `storage[child//2]`.

❽ Если надо, узел продолжит всплывать, и теперь `child` равен индексу родителя.

Метод `.swim()` оказался весьма лаконичен! Индекс свежедобавленного узла — `child`, индекс его родителя, если таковой имеется, — `child//2`. Если приоритет родителя меньше приоритета `child`, они меняются местами и всплытие продолжается.

На рис. 4.18 представлено, как меняется массив `storage` после вызова `enqueue(`*`значение`*`, 12)` из состояния, которое было показано на рис. 4.8. В каждом следующем ряду ячейки `storage` меняются, как это ранее отображалось на очередном рисунке. В последнем ряду — состояние массива, соответствующее двоичной куче с пирамидальностью и частичным порядком.

Рис. 4.18. Как меняется storage после вызова enqueue() на рис. 4.8

Путь от вершины кучи до только что добавленного элемента с приоритетом 12 состоит из пяти узлов — они помечены серым на рис. 4.18. После двух итераций цикла `while` в методе `.swim()`, в которых узел всплывает, меняясь местами со своим родителем, он попадает в ячейку `storage[4]` и свойство пирамидальности кучи восстанавливается. Количество обменов не может превышать $\log_2 N$, где N — общее число ячеек в двоичной куче.

В примере 4.3 мы реализовали метод `.sink()`, который используется для восстановления свойств двоичной кучи при снятии элемента с ее вершины при помощи `.dequeue()`.

Пример 4.3. После добавления методов .dequeue() и .sink() мы получаем полноценно работающую кучу

```
def dequeue(self):
  if self.N == 0:
    raise RuntimeError ('PriorityQueue is empty!')
  max_entry = self.storage[1]                              ❶
  self.storage[1] = self.storage[self.N]                   ❷
  self.storage[self.N] = None
  self.N -= 1                                              ❸
  self.sink(1)
  return max_entry.value                                   ❹

def sink(self, parent):
  while 2*parent <= self.N:                                ❺
    child = 2*parent
    if child < self.N and self.less(child, child+1):       ❻
      child += 1
    if not self.less(parent, child)                        ❼
      break
    self.swap(child, parent)                               ❽
    parent = child
```

❶ Запомним ячейку на вершине кучи.

❷ Переставим на вершину последний (самый правый в самом нижнем уровне) элемент, а его место освободим.

❸ Уменьшим количество узлов и запустим `sink()` для `storage[1]`.

❹ Вернем значение элемента, снятого с вершины кучи.

❺ Проверка продолжается, пока у узла есть хотя бы левый потомок.

❻ Если правый потомок существует и приоритет левого потомка меньше, нам нужен правый потомок.

❼ Если приоритет родителя не меньше максимального приоритета потомков, пирамидальность восстановлена.

❽ Если нет, поменяем местами родителя и потомка и продолжим топить наш узел уже с нового места.

На рис. 4.19 изображено, как меняется `storage` в процессе работы `dequeue()` — начиная с двоичной кучи с рис. 4.11. Первая строка на рис. 4.19 — это массив из 19 элементов. Во второй строке самый последний элемент с приоритетом 9 перемещается в самое начало (что соответствует вершине двоичной кучи и нарушает пирамидальность); кроме того, теперь в куче 18 узлов, так как последний мы удалили.

Уровень 0 Уровень 1 Уровень 2 Уровень 3 Уровень 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
–	16	15	14	13	11	12	14	8	12	1	10	8	6	9	7	4	5	2	9
–	9	15	14	13	11	12	14	8	12	1	10	8	6	9	7	4	5	2	

Серым показаны потомки узла с приоритетом 9;
на место одного из них мы топим этот узел

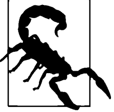
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
–	15	9	14	13	11	12	14	8	12	1	10	8	6	9	7	4	5	2	
–	15	13	14	9	11	12	14	8	12	1	10	8	6	9	7	4	5	2	
–	15	13	14	12	11	12	14	8	9	1	10	8	6	9	7	4	5	2	

После троекратного утопления узел с приоритетом 9
оказывается на своем месте

Рис. 4.19. Изменения массива storage в процессе снятия элемента с вершины кучи, приведенной на рис. 4.11

После трех последовательных итераций цикла `while` в методе `.sink()` узел с приоритетом 9 погружается на позицию, в которой достигается пирамидальность кучи, серым всюду обозначены потомки этого узла. Всякий раз, когда выясняется, что приоритет родителя меньше приоритета потомка, родитель и потомок с более высоким приоритетом меняются местами. Количество таких обменов не может превысить $\log_2 N$.

Метод `.sink()` наглядно показать сложнее, потому что нет однозначного маршрута, по которому тонет переставленный на вершину кучи узел, — в отличие от `.swim()`. В окончательном состоянии `storage` из примера 4.19 погрузившийся на свое место узел с приоритетом 9 имеет еще одного потомка (с приоритетом 2). После отрабатывания `.sink()` мы можем быть уверены, что приоритет узла не меньше (то есть больше либо равен) приоритетов всех имеющихся у него потомков. Как частный случай, индекс ячейки `p` может превышать $N // 2$, тогда у узла потомков нет, так как элементов с индексами `2p` и `2p + 1` нет в `storage`.



В методе .dequeue() важно сначала уменьшить N на 1, а только потом вызвать .sink(1), иначе sink() примет элемент, лежащий за пределами кучи. На всякий случай, если кто-то все-таки решит туда заглянуть, мы предварительно записываем None в storage[N] — тогда уж точно ее не перепутать с узлом. В программировании такой прием — защита от ошибки, которой вроде бы и так не должно случиться. Она называется hardening — усиление надежности исходного текста.

Чтобы убедиться в правильности такой логики метода `.dequeue()`, посмотрим, что случится, если куча содержит единственный элемент, а мы его снимаем. Узел `max_entry` мы запомнили, N уменьшили до 0, затем вызывали `sink()`, а он, как и ожидается, ничего не делает, потому что $2 \times 1 > 0$.

Заключение

С помощью двоичной кучи можно эффективно реализовать абстрактный тип данных — приоритизированную очередь. Довольно много алгоритмов, например те, что мы обсудим в главе 7, пользуются этим типом данных.

- Добавление в очередь пары (`значение, приоритет`) имеет логарифмическую вычислительную сложность $O(\log N)$.
- Снятие из начала очереди элемента с наивысшим приоритетом тоже имеет сложность $O(\log N)$.
- Подсчет количества элементов в очереди имеет константную сложность $O(1)$.

В этой главе мы занимались только двоичной кучей с частичным порядком убывания. Если требуется *частичный порядок возрастания*, когда на вершине содержится наименьший элемент, достаточно изменить совсем немного (это нам понадобится в главе 7). В примере 4.2 нужно взять метод `.less()` и поменять в нем операцию «больше» на «меньше», не трогая сверх того ничего.

```
def less(self, i, j):
  return self.storage[i].priority > self.storage[j].priority
```

Приоритизированная очередь может быть любого размера, но в нашей реализации использовалась куча фиксированного размера M, в которой можно было хранить только $N < M$ элементов. Если куча полна, больше элементов в очередь не добавить. Мы выбрали эту реализацию, чтобы аккуратнее оценить быстродействие: посчитать количество операций сравнения и обмена, не опасаясь, что какие-то другие действия с данными не попадут в оценку. Из главы 4 мы знаем, что массив фиксированного размера всегда можно заменить динамическим

массивом с геометрическим масштабированием (удвоением при заполнении). Это не изменит оценку сложности в среднем, так что сложность `enqueue()` останется O(log N).

Тренировочные задания

1. В действительности для реализации двоичной кучи произвольного размера почти ничего дополнительно делать не надо: все равно мы используем не массив, а `list`, динамические свойства которого как раз и обеспечиваются геометрическим масштабированием. Перепишите класс `PQ`, чтобы в приоритизированной очереди можно было хранить произвольное количество элементов. Сравните быстродействие полученной структуры данных с `PQ`, для чего создайте достаточной длины последовательность со случайными приоритетами, добавьте их по одному в каждую из структур, а потом по одному снимите. Какая структура работает быстрее? Почему? Значимо ли это различие: есть ли основание считать, что какие-то операции над этими структурами имеют различный класс сложности?
   - Отдельно хранить `.size` и `.N` нет необходимости: они всегда равны `len(storage)`.
   - Для работы с *последней* ячейкой массива используйте `.append()` и `.pop()`.
   - Другие методы, такие как `.insert()` или `.pop()` с параметром, использовать нельзя: их сложность в среднем линейна, в то время как сложность `.append()` и `.pop()` константна.

   Обычную очередь фиксированного размера можно весьма эффективно реализовать с помощью массива `storage` фиксированного размера, причем быстродействие операций `enqueue()` и `dequeue()` будет константным, O(1). Один из способов — это так называемый *кольцевой буфер*, использующий простую, но остроумную идею: начальный элемент очереди не обязательно лежит в `storage[0]`. Достаточно просто хранить *два* индекса — `forst`, позицию того, кто стоит в очереди дольше всех, и `last`, позицию, куда мы поместим очередной добавляемый элемент, когда он появится. Посмотрим на рис. 4.20.

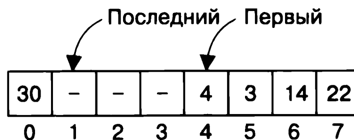


Рис. 4.20. Массив в качестве кольцевого буфера

При каждом добавлении элемента в конец очереди и снятии элемента из ее начала надо аккуратно обновлять `first` и `last`. Скорее всего, будет удобно хранить N — количество элементов в очереди и придется использовать остаток от деления — операцию `%`. Изучите пример 4.4, допишите соответствующие методы и проверьте, что они действительно работают за константное время.

Пример 4.4. Допишите реализацию очереди в виде кольцевого буфера

```
class Queue:
  def __init__(self, size):
    self.size = size
    self.storage = [None] * size
    self.first = 0
    self.last = 0
    self.N = 0

  def is_empty(self):
    return self.N == 0

  def is_full(self):
    return self.N == self.size

  def enqueue(self, item):
    """If not full, enqueue item in O(1) performance."""

  def dequeue(self):
    """If not empty, dequeue head in O(1) performance."""
```

2. Добавьте в порядке возрастания $N = 2^k - 1$ элемент в пустую двоичную кучу размером N. Изучите получившийся массив, который хранит узлы кучи (напомню, позиция 0 в нем не используется). Можно ли предсказать, какие места в массиве будут занимать k наибольших элементов? А если в пустую двоичную кучу добавить N элементов в порядке убывания, нельзя ли предсказать место *каждого* элемента в массиве ячеек?

3. Допустим, у нас есть две кучи размером M и N. Придумайте алгоритм, который заполняет массив размером $M + N$ всеми элементами обеих куч в порядке возрастания за $O(M \log M + N \log N)$ операций. Для подтверждения правильности алгоритма подготовьте таблицу с результатами экспериментов.

4. Придумайте метод получения k наименьших элементов среди произвольного набора из N элементов за время $O(N \log k)$. Для подтверждения правильности алгоритма подготовьте таблицу с результатами экспериментов.

5. В двоичной куче у каждого узла бывает не больше двух потомков. Рассмотрим другой вариант, в котором узел на вершине кучи имеет двух потомков, каждый из них имеет трех потомков (назовем их внуками), каждый

из внуков — четырех потомков и т. д., как показано на рис. 4.21. Назовем эту структуру *факториальной кучей*: в ней с каждым уровнем у узлов прибавляется по дополнительному потомку. Куча должна обладать свойствами пирамидальности и частичного порядка. Реализуйте факториальную кучу, используя хранение ячеек в массиве, поэкспериментируйте с ней и убедитесь, что она работает медленнее двоичной. Попробуйте оценить вычислительную сложность; это задача похитрее, но в конце концов у вас должно получиться $O((\log N) / (\log(\log N)))$.

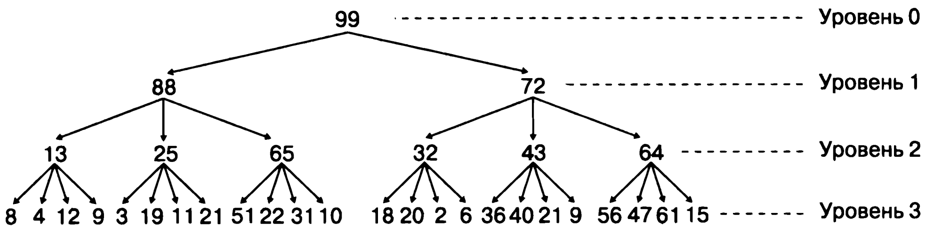


Рис. 4.21. Инновационная факториальная куча

6. Используя геометрическое масштабирование, описанное в главе 3, доработайте *кольцевой буфер* из упражнения 2 так, чтобы он мог работать с произвольным количеством элементов. Переносить ячейки из старого массива в новый удобнее не по одной, а сегментами (таких сегментов будет один или два). Реализуйте уменьшение размера массива, если показатель заполнения опустился ниже 1/4.

7. Допустим, мы хотим сделать итератор по всей куче, который возвращает по одному значения в том порядке, *в котором мы снимали бы их с кучи*. С одной стороны, такой итератор не должен модифицировать саму структуру, с другой — при снятии элемента с кучи нужно переструктурировать массив ячеек. Как совместить оба требования? Одно из решений — написать генератор-функцию `iterator(pq)`, которая получает на вход приоритизированную очередь `pq` и создает ее копию, `pqit`, откуда потом и снимает по одному все элементы. Начнем с более надежного варианта, в котором `pqit` хранит не сами элементы `pq`, а их индексы в массиве `pq.storage`; приоритеты при этом остаются такими же, как и у соответствующих значений. По этим индексам мы и будем обращаться к `pq.storage`, до самого последнего момента не трогая его содержимого и вообще не меняя его структуры.

Допишите следующий пример, в начале которого в `pqit` добавляется индекс 1, который соответствует элементу `pq` с наивысшим приоритетом. Продолжите заполнение `pqit` и допишите цикл `while` из примера:

```
def iterator(pq):
  pqit = PQ(len(pq))
  pqit.enqueue(1, pq.storage[1].priority)
  ...
  while pqit:
    idx = pqit.dequeue()
    yield (pq.storage[idx].value, pq.storage[idx].priority)
    ...
```

Такой итератор должен возвращать все элементы pq в порядке приоритетов, не изменяя сам pq.

Стоит заметить, что особой надежности использование в куче-копии индексов вместо самих элементов, по-видимому, не добавляет. Напишите вариант нашей функции, iteratorc(pq), в которой задается новая куча pqit минимального размера, а затем все поля pq просто копируются туда (включая storage, который можно получить с помощью pq.storage.copy()). Будет ли такая структура надежной: нет ли случаев, в которых что-то в pq модифицируется? Какая из функций работает быстрее и почему?

ГЛАВА 5

Сортировка без магии

В этой главе

- Алгоритмы сортировки основаны на сравнении и пользуются двумя основными действиями:
  - определением порядка элементов `A[i] < A[j]`;
  - обменом элементов `A[i]` и `A[j]` местами.
- Как использовать компаратор и сравнение по ключу для сортировки последовательности в произвольном порядке или для сортировки объектов, порядок которых вообще не определен, например координат точек `(x, y)` на плоскости.
- Как по тексту программы понять, что некоторые алгоритмы, например **сортировка вставками** и **сортировка выбором**, имеют неоправданно высокую сложность $O(N^2)$.
- Как использовать одно из важнейших понятий в программировании — *рекурсию*, когда функция вызывает сама себя, — для решения задач с помощью алгоритмов типа «разделяй и властвуй».
- Как стратегия «разделяй и властвуй» помогает **сортировке слиянием** и **быстрой сортировке** упорядочивать массив из N элементов за $O(N \log N)$ действий и как **пирамидальная сортировка** достигает того же быстродействия без этой стратегии.
- Как **сортировка Тима** (которая используется в Python по умолчанию) объединяет преимущества **сортировки вставками** и **сортировки слиянием** и тоже показывает быстродействие $O(N \log N)$.

В этой главе мы будем рассматривать алгоритмы, которые решают такую задачу: переставить элементы в массиве размером N так, чтобы они шли в порядке возрастания. Многие более сложные задачи можно упростить и решать более

эффективно, если набор данных предварительно отсортировать; временами без сортировки вообще не обойтись. Сортировка очень часто возникает и в повседневных задачах — например, листок со списком сотрудников и их телефонами обычно отсортирован по именам, а список рейсов на табло в аэропорту — по времени вылета.

Если данные в массиве не отсортированы, поиск элемента в *наихудшем случае* имеет сложность $O(N)$, а если отсортированы — можно применить **двоичный поиск**, сложность которого в *наихудшем случае* $O(\log N)$.

Обмен элементов в сортировке

Попробуем отсортировать массив A, изображенный на рис. 5.1, *сверху*. Можно написать числа на клочках бумаги или делать пометки карандашом прямо в книге (только стереть их потом!). Условие: можно использовать только два действия — сравнение двух элементов и *обмен местами двух элементов*. Какое наименьшее количество обменов потребуется? Заодно стоит посчитать количество сравнений. В примере массив отсортирован посредством пяти обменов. А можно ли было меньше[1]?

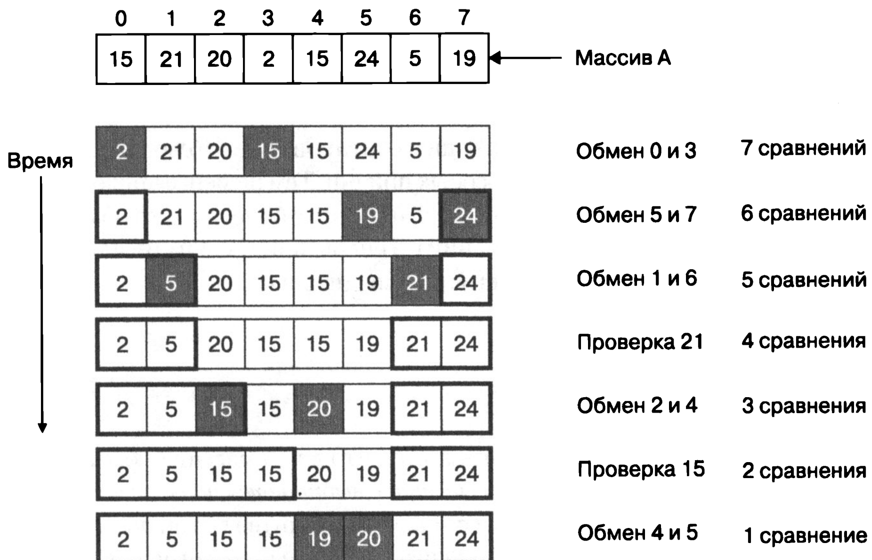


Рис. 5.1. Сортировка некоторого массива A

[1] Нельзя. См. пример в конце тренировочных заданий. — *Примеч. авт.*

Считать *количество попарных сравнений* элементов не менее важно, чем считать число их обменов. Итак, найти наименьший элемент в A — число 2 — можно за семь сравнений, как это было показано в главе 1. Минимум найден в A[3], так что меняем его местами с A[0], и теперь в начале массива, на своем месте, стоит наименьший элемент. Элементы, которые меняются местами на очередном шаге, выделены на рис. 5.1 серым. Жирной рамкой обведены области массива, элементы которых гарантированно стоят на своих местах и больше не переместятся. Это две непрерывные области с двух концов массива, а всю середину нужно сортировать дальше.

Осталось семь элементов, из которых за шесть сравнений мы находим наибольший, 24 в позиции A[5], и меняем его местами с последним элементом, A[7]. На оставшемся отрезке из шести элементов за пять сравнений мы находим наименьший, 5 в позиции A[6], и меняем его местами с начальным элементом отрезка, A[1]. Теперь снова находим наибольший, 21, оказывается, он уже на своем месте, и обменов не требуется.

Среди четырех элементов отрезка за три сравнения находим наименьший, 15, и меняем местами с началом отрезка — A[4] меняем с A[2]. Снова поищем минимум, уже за два сравнения, это снова окажется 15, и оно стоит на своем месте. Теперь сравним оставшиеся два элемента, A[4] и A[5], и поменяем их местами. Это последнее действие на рис. 5.1, и можно заметить, что оба элемента при этом оказываются на своих местах, например, 20 не меньше каждого элемента слева от него и не больше каждого элемента справа. Итак, массив отсортирован за 28 сравнений и пять обменов.

Никакого конкретного алгоритма в нашей сортировке не наблюдалось — мы искали то минимум, то максимум без особых причин. Тем не менее число сравнений на каждом шаге уменьшалось на единицу, и было не более одного обмена за шаг (что куда меньше количества сравнений). Теперь опишем четкий алгоритм для массива из N элементов и оценим его производительность.

Сортировка выбором

Сортировка выбором названа так из-за того, что в ней массив просматривается справа налево и *на каждую очередную позицию выбирается наименьший из оставшихся элементов* и меняется местами с тем, что находится в этой позиции сейчас. Для сортировки N элементов массива A сначала надо найти наименьший по всему массиву и поменять его местами с A[0], затем — наименьший среди оставшихся $N - 1$ элементов и поменять его местами с A[1]. Останется $N - 2$ неотсортированных элементов, и так следует повторять, пока не останется один — наибольший.

Что делать, если на некотором шаге наименьший элемент уже находится на своем месте? В примере ниже это значит, что по окончании цикла по j значение i окажется равным min_index. Да ничего! Программа попытается поменять местами A[i] и A[min_index], и массив не изменится. Можно попробовать проверять, что i и min_index не равны, и только тогда менять местами ячейки, но заметного прироста производительности это не даст.

В примере 5.1 внешний цикл по i перебирает почти все позиции в массиве, от 0 до $N - 2$. Внутренний цикл по j перебирает позиции оставшихся неотсортированными элементов, от $i + 1$ до $N - 1$ — среди них нужно найти наименьшее. В конце цикла по i элемент массива в позиции i меняется местами с найденным наименьшим элементом в позиции `min_index`.

Пример 5.1. Сортировка выбором

```
def selection_sort(A):
  N = len(A)
  for i in range(N-1):                          ❶
    min_index = i                               ❷
    for j in range(i+1, N):
      if A[j] < A[min_index]:                   ❸
        min_index = j

      A[i], A[min_index] = A[min_index], A[i]   ❹
```

❶ На каждом шаге цикла по `i` мы знаем, что `A[0 … i-1]` уже отсортированы.

❷ Индекс наименьшего элемента среди `A[i … N-1]` будет занесен в `min_index`.

❸ Если какой-то `A[j]` окажется меньше `A[min_index]`, надо обновить `min_index` — запомнить позицию наименьшего на данный момент элемента.

❹ После обмена `A[i]` и `A[min_index]` отсортированы будут уже `A[0 … i]`.

Внешний цикл **сортировки выбором** работает с набором данных размером N (всем массивом), а затем размер задействованного набора уменьшается на 1 с каждым оборотом цикла — сначала до $N - 1$, затем — до $N - 2$ и так до конца сортировки. Рисунок 5.2 показывает, что для этого требуется *ровно* $N - 1$ *обмен*.

После $N - 1$ обменов все элементы оказываются на своих местах, включая последний, `A[N - 1]`, — он не участвовал в поиске минимума, но остался один, и значит, он наибольший. Теперь попробуем посчитать количество сравнений — это чуть-чуть сложнее. На рис. 5.2 сравнений всего 28, что равно сумме чисел от 1 до 7.

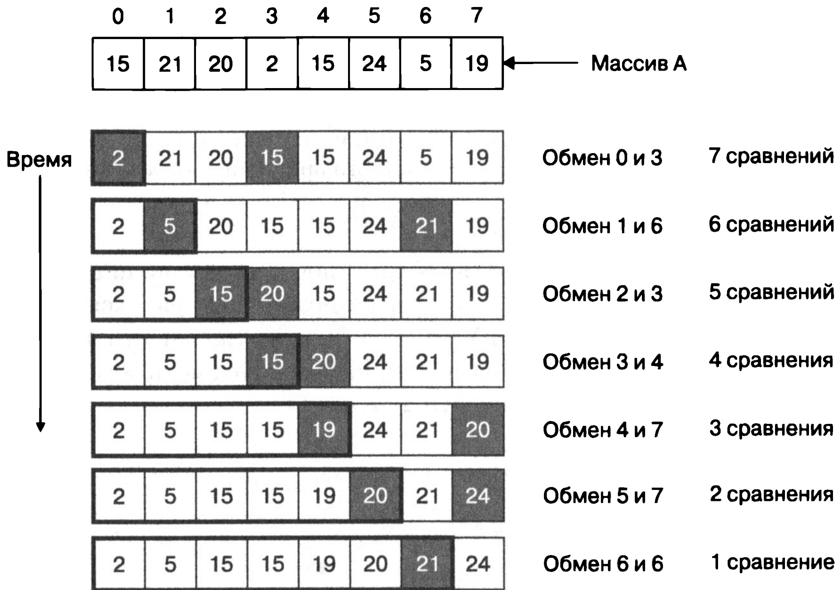


Рис. 5.2. Сортировка произвольного массива выбором

Сумму значений от 1 до K можно вычислить как $K \times (K + 1) / 2$. Такие числа в математике называются *треугольными*, потому что $K \times (K + 1) / 2$ предметов можно расположить в виде треугольника. На рис. 5.3 наглядно представлено само треугольное число 28 и идея, на которой основана формула.

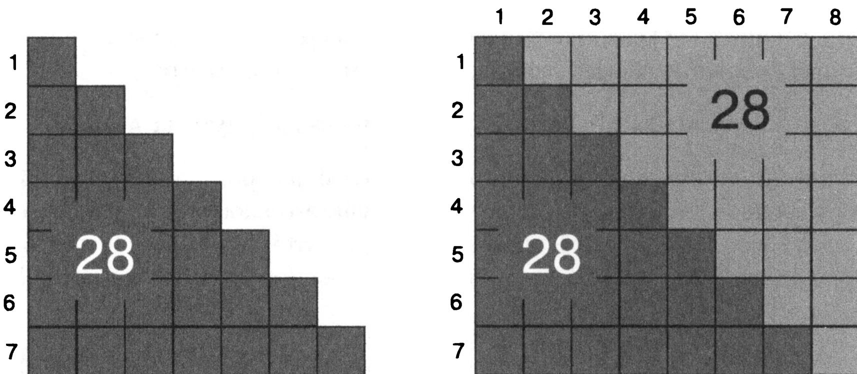


Рис. 5.3. Наглядное представление формулы треугольного числа: 28 как сумма от 1 до 7

Расположим предметы в виде треугольника из K строк и K элементов в самой длинной строке. Продублируем этот треугольник, перевернем вверх ногами

и приложим к исходному. Получим прямоугольник размером K на $K + 1$. Выходит, что количество предметов в треугольнике — это ровно половина количества предметов в прямоугольнике. Для треугольника из семи строк двойной прямоугольник будет состоять из $7 \times 8 = 56$ предметов, значит, в треугольнике их 28. В сортировке $K = N - 1$ это количество сравнений на первом шаге, необходимое для поиска минимума. Общее число сравнений равно $(N - 1) \times N / 2$.

Структура квадратичных алгоритмов сортировки

Доминирующая составляющая в оценке сложности *сортировки выбором* — это количество сравнений, их порядка N^2, значит, и быстродействие окажется ей под стать — $O(N^2)$. На самом деле в алгоритме сортировки N элементов ровно $N - 1$ шаг. На первом шаге для поиска минимума необходимо $N - 1$ сравнений, и только один элемент встает на свое место. На оставшихся $N - 2$ шагах количество сравнений уменьшается, но так медленно, что только под самый конец в них отпадает необходимость. При таком подходе количество сравнений будет строго соответствовать формуле, то есть иметь квадратичный порядок. Можно ли как-то уменьшить это число?

Сортировка вставками — это другой подход, в котором для упорядочения массива слева направо тоже нужно проделать $N - 1$ отдельных шагов. Начнем с того, что массив из одного элемента всегда упорядочен; а вдруг `A[0]` и правда наименьший, ведь и такое случается? Если на первом шаге окажется, что `A[1]` меньше `A[0]`, `A[1]` надо поставить в начало — в данном случае поменять местами с `A[0]`. На втором шаге мы рассматриваем `A[2]` и пробуем *вставить* его на свое место, считая, что массив пока состоит только из трех элементов. Возможны три варианта: `A[2]` и так на своем месте; его надо вставить между `A[0]` и `A[1]`; его надо вставить в начало, перед `A[0]`. Однако нельзя просто так вставить в классический массив один элемент между другими — можно только сдвинуть все ячейки поэлементно. Для этого надо запомнить исходный элемент, поставить соседа на его место, поставить следующего соседа на место соседа и т. д. до нужной позиции, на которую поставить исходный элемент. Другой вариант — менять местами элементы попарно до тех пор, пока исходный не встанет на свое место[1].

Как показано на рис. 5.4, каждый шаг завершается последовательным обменом соседних ячеек, пока те не упорядочены.

[1] В результате автор избрал второй вариант, который не является сортировкой вставками в точном смысле — это скорее гибрид между ней и сортировкой пузырьком. — *Примеч. пер.*

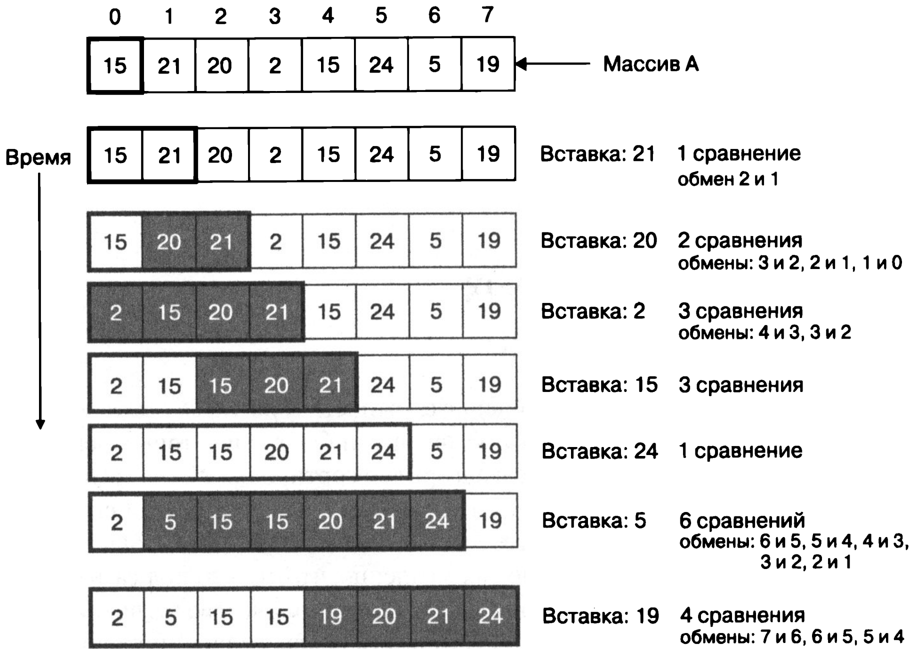


Рис. 5.4. Сортировка вставками в произвольный массив

Все элементы, которые пришлось менять местами, выделены серым, а отсортированная область массива обведена жирной чертой. На рисунке видно, что элементы в отсортированной области время от времени тоже переставляются (в **сортировке выбором** было не так). Количество последовательных обменов, приводящих к вставке элемента на свое место, иногда довольно велико (как, например, при вставке 5), а иногда (как, например, при вставке 21 и 24) обменов вовсе не требуется — очередное значение и так больше остальных, и порядок сохраняется. В нашем примере 20 сравнений и 14 обменов. При **сортировке вставками** число сравнений всегда больше числа обменов, но по сравнению с **сортировкой выбором** обменов больше, а сравнений меньше. Неожиданно простая реализация этого алгоритма приведена в примере 5.2.

Пример 5.2. Сортировка вставками

```
def insertion_sort(A):
  for I in range(1, len(A)):               ❶
    for j in range(I, 0, -1):              ❷
      if A[j-1] <= A[j]:                   ❸
        break
      A[j], A[j-1] = A[j-1], A[j]          ❹
```

❶ На каждом шаге цикла по `i` мы знаем, что `A[0 … i-1]` уже отсортированы.

❷ Индекс `j` уменьшается от `i` до 1 включительно — это возможные положения элемента, с которым нужно менять местами вставляемый.

❸ Если `A[j-1]` не больше `A[j]`, место для вставки найдено и цикл останавливается.

❹ В противном случае меняем местами два элемента: они нарушали порядок.

Больше всего работы достается **сортировке вставками**, когда очередное добавляемое значение меньше всех уже отсортированных. *Наихудший случай* для **сортировки вставками** — значения, упорядоченные по убыванию. При этом на каждом шаге и количество сравнений, и количество обменов увеличивается на 1, так что обе характеристики оказываются уже знакомыми нам треугольными числами.

Оценка производительности сортировки выбором и сортировки вставками

Сортировка N значений **выбором** всегда делает $N \times (N - 1)$ сравнений и $N - 1$ обменов. В случае **сортировки вставками** посчитать количество действий труднее, потому что оно зависит от порядка самих значений в массиве. **Сортировка выбором** в среднем менее эффективна, но *наихудший случай* для **сортировки вставками** — значения, упорядоченные по убыванию, — дает $N \times (N - 1)$ сравнений и столько же обменов. Как бы мы ни улучшали эти алгоритмы, порядок сравнений в них — N^2; на рис. 5.5 приведены графики быстродействия. Чем это нас не устраивает? Как обычно, сравним, насколько понижается быстродействие, когда входной набор данных увеличивается. Набор из 524 288 элементов в 512 раз больше набора из 1024 элементов, а оба алгоритма сортировки работают на нем почти в 275 000 раз медленнее[1]. Сортировка 524 288 значений **вставками** заняла в эксперименте около двух часов, а **сортировка выбором** — почти четыре. Если объемы данных будут больше, очень скоро счет пойдет на дни и недели. Это никуда не годится, но квадратичные алгоритмы (сложность которых $O(N^2)$) на лучшее не способны.

Что, если мы хотим отсортировать массив в каком-нибудь другом порядке? Хоть бы и по убыванию? В общем случае у нас даже может *не быть операции сравнения* самих элементов массива, а порядок их важен, что делать тогда?

[1] Значение 275 000 примерно равно квадрату 512. — *Примеч. авт.*

Простой подход состоит в том, чтобы вместо арифметической операции «меньше» использовать заранее заданную функцию-*компаратор*, которая определит, не нарушается ли порядок в паре значений. Эта функция передается в качестве дополнительного параметра алгоритму сортировки. Все алгоритмы в этой главе можно модифицировать таким образом, так что впредь будем для простоты рассматривать только сортировку по возрастанию. В примере 5.3 представлена **сортировка вставками** с компаратором.

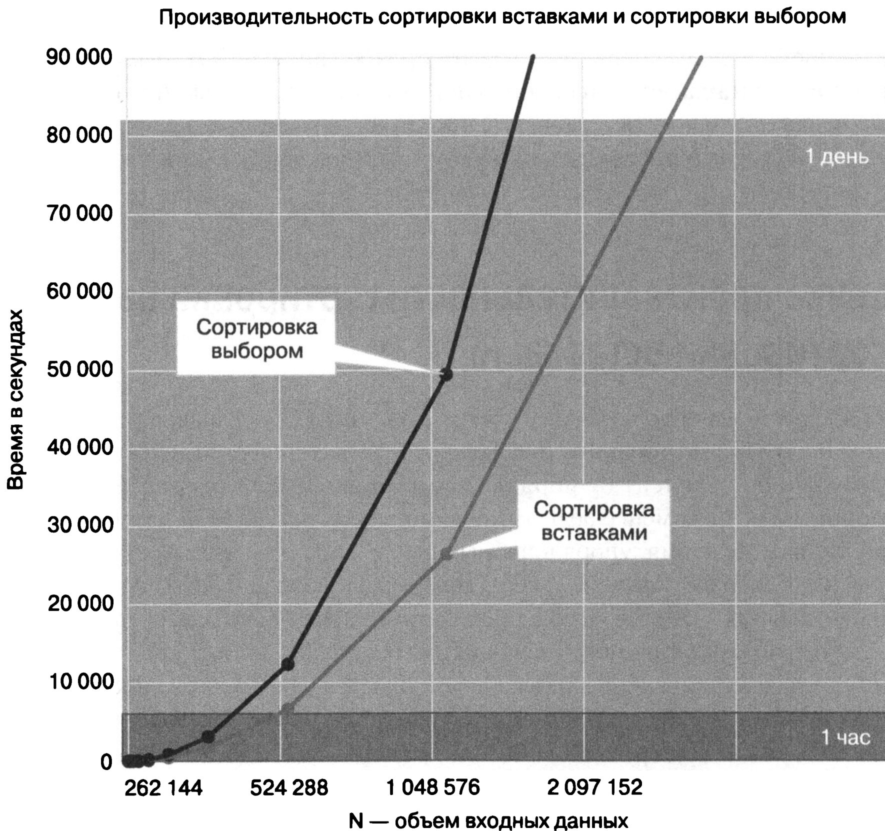


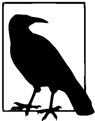
Рис. 5.5. Время работы сортировки вставками и сортировки выбором

Пример 5.3. Использование компаратора в алгоритме сортировки

```
def less_than(one, two):
  return  one <= two

def insertion_sort_cmp(A, less=less_than):
  N = len(A)
```

```
for i in range(1, N):
  for j in range(i, 0, -1):
    if less(A[j-1], A[j]):        ❶
      break
    A[j], A[j-1] = A[j-1], A[j]
```

❶ Порядок сортировки определяется возвращаемым значением функции-компаратора, `less()`. Если `less(A[j-1], A[j])` истинно, `A[x]` должно предшествовать `A[y]`.

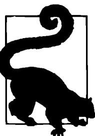


Компаратор не просто функция, возвращающая True или False. Он должен обладать свойством сохранения порядка (на математическом языке — транзитивностью), при котором из того, что A предшествует B, а B предшествует C, с необходимостью следует, что A предшествует также и C. Типичный пример нетранзитивного компаратора — определение победителя в игре «камень, ножницы, бумага», в которой ножницы побеждают бумагу, камень побеждает ножницы, но бумага побеждает камень. Задание нетранзитивного компаратора может привести к вечному зацикливанию алгоритма или неоднозначности результата. Для большей безопасности используют так называемое сравнение по ключу, когда переданная алгоритму сортировки функция вычисляет из элемента массива некоторое значение (обычно числовое или строковое) — ключ, а для определения порядка ключи элементов сравниваются заведомо транзитивной операцией «меньше». Именно так устроена стандартная функция sort() в Python.

И **сортировка выбором**, и **сортировка вставками** упорядочивают массив за $N - 1$ шаг, на каждом из которых объем входных данных уменьшается на 1. Совершенно другой подход, который можно назвать «разделяй и властвуй», состоит в том, чтобы разбить задачу на две подзадачи и решать их по отдельности.

Рекурсия: разделяй и властвуй!

Идея рекурсии возникла в математике сотни лет назад. В программировании рекурсия появляется, когда функция вызывает саму себя.



Последовательность Фибоначчи начинается с двух целых чисел, 0 и 1. Каждый следующий член этой последовательности — сумма двух предыдущих. Первые несколько чисел последовательности Фибоначчи: 1, 2, 3, 5, 8, 13 и т. д. Можно записать рекуррентное соотношение для n-го члена последовательности Фибоначчи: $F(n) = F(n - 1) + F(n - 2)$. Если запрограммировать это вычисление рекурсивно, функция $F(n)$ будет дважды вызывать саму себя.

Факториал натурального числа N — это произведение всех натуральных чисел, не превосходящих N. Обозначается он $N!$, так что $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Можно записать и рекуррентное соотношение: $N! = N \times (N - 1)!$. Действительно, $120 = 5! = 5 \times 4!$, где $4! = 24$. Пользуясь этим соотношением, можно реализовать вычисление факториала рекурсивно (пример 5.4).

Пример 5.4. Рекурсивное вычисление факториала

```
def fact(N):
  if N <= 1                ❶
    return 1
  return N * fact(N - 1)   ❷
```

❶ Основание рекурсии: если $N \leq 1$, возвращается 1.

❷ Рекурсивный вызов: вычислим $(N - 1)!$ и домножим его на N.

Функция `fact()` вызывает саму себя. Это подозрительно! А вдруг она никогда не закончится? Чтобы этого не произошло, в каждую рекурсивную функцию добавляют *основание рекурсии* — условие, при котором вызова не происходит и действия не становятся бесконечными. Основание для факториала — $N = 1$. В этом случае функция сразу возвращает `1` и не выполняет рекурсивного вызова[1]. Рекурсивный вызов в `fact(N)` — это `fact(N - 1)`, результат домножается на N, и это и есть факториал.

На рис. 5.6 показано, как Python интерпретирует команду `y = fact(3)` (каждое следующее по времени состояние интерпретатора отображается под предыдущим). Прямоугольниками обозначены *контексты вызова* функции `fact()` с соответствующими параметрами (`3`, `2` и `1`). Вызов `fact(3)` приводит к рекурсивному вызову `fact(2)`. В это время вычисление `fact(3)` «приостанавливается» (на рисунке выделено серым) до тех пор, пока не станет известно значение `fact(2)`. Вызову `fact(2)`, в свою очередь, требуется рекурсивно вызвать `fact(1)`, так что он тоже приостанавливается (и мы его отмечаем серым) до вычисления `fact(1)`. Тут мы достигаем основания рекурсии, и `fact(1)` возвращает `1`. Контекст вызова `fact(1)` уничтожается, остается только возвращаемое значение (оно выделено пунктирной окружностью), которое передается в контекст вызова `fact(2)`, и вычисления в нем возобновляются: возвращается $2 \times 1 = 2$. Остается только исходный контекст вызова `fact(3)` со значением `2`, теперь возобновляется он, вычисляет и возвращает $3 \times 2 = 6$, и в результате `y` становится равным `6`.

[1] Если проверять только равенство, интерпретатор Python все равно может войти в бесконечную рекурсию и аварийно завершиться (если N отрицательное или не целое). Поэтому мы считаем основанием все N, не превосходящие 1. — *Примеч. авт.*

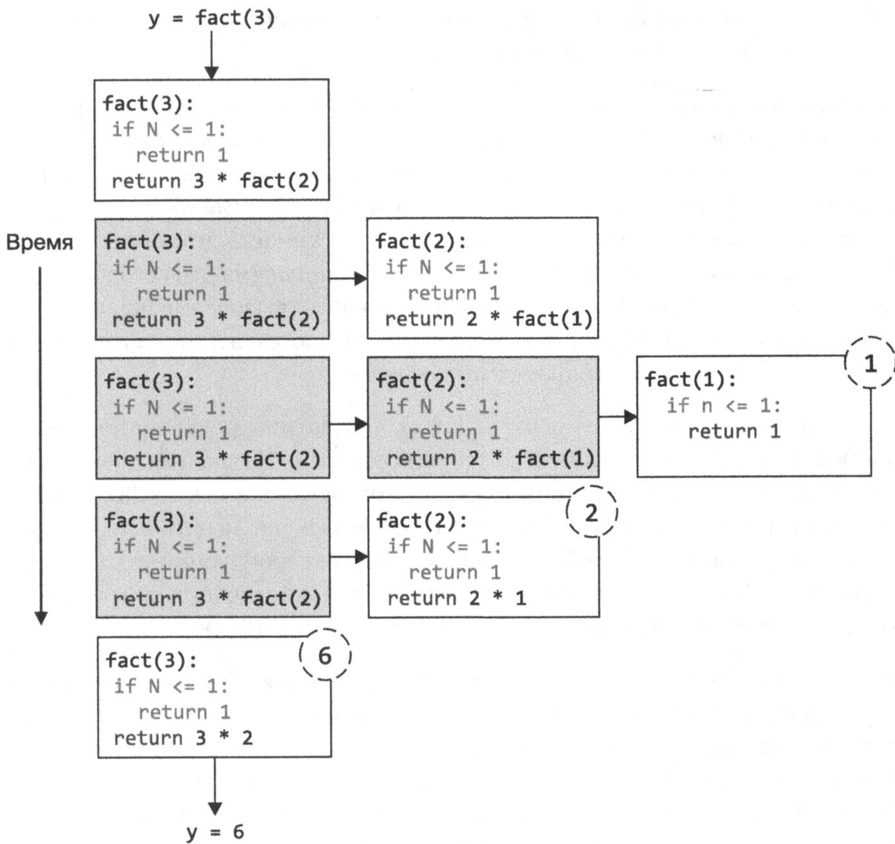


Рис. 5.6. Рекурсивный вызов fact(3)

Итак, рекурсивный вызов `fact(N)` порождает цепочку контекстов, приостановленных до тех пор, пока в самом последнем не будет достигнуто основание рекурсии. Затем Python начинает «отматывать» эту цепочку обратно, по одному вызову, до тех пор пока не завершится самый первый.

Рекурсивное вычисление факториала — самый популярный, но, к сожалению, ***неправильный*** пример использования рекурсии. Из рис. 5.6 понятно, что на реализацию каждого шага рекурсии затрачивается ощутимое количество памяти. Кроме того, язык программирования вынужден выполнять довольно много работы по обслуживанию самого механизма рекурсии: например, Python на каждом шаге создает отдельное пространство имен и заводит там локальные переменные; при выходе из рекурсивного вызова все это надо еще удалить. Реализация того же алгоритма обычным циклом будет работать в несколько

раз быстрее и, что еще важнее, будет иметь *более низкий порядок сложности по памяти* (в нашем случае — O(1), а не O(*N*)).

Поэтому стоит запомнить одно простое правило. Если в рекурсивно организованном алгоритме, который обрабатывает входные данные размером *N*, *глубина рекурсии* (сколько раз максимум функция вложенно вызвала сама себя,) пропорциональна *N*, то вы, скорее всего, реализовали *обычный цикл* и так делать не надо. Как мы уже знаем, для этого достаточно увидеть, что объем входных данных при каждом вызове уменьшается на *константу* (чаще всего на 1). А вот если объем входных данных при каждом вложенном вызове уменьшается *в несколько раз* (например, вдвое), глубина рекурсии окажется логарифмической — O(log *N*) — и это хорошее, годное ее применение.

По этим двум причинам (относительная «дороговизна» рекурсивного вызова и бессмысленность замены линейного цикла рекурсией) максимальное количество вложенных вызовов функций друг другом ограничено в Python тысячей. С одной стороны, тысяча контекстов вызова — все еще не слишком большое потребление памяти, с другой — очевидно, что даже для двоичного логарифма логарифмическая глубина рекурсии 1000 актуально недостижима (размер входных данных не может быть 2^{1000}, то есть примерно 10^{302}).

Наш алгоритм этому логарифмическому критерию не соответствует: в нем объем входных данных *N* на каждом шаге уменьшается только на единицу. Рисунок 5.6 наглядно показывает, что числа от *N* до 1 — это именно входные данные: все они одновременно хранятся в контекстах вызова и занимают память. Вот если бы можно было разбить входные данные на две примерно равные *N* / 2 части и решать две получившиеся подзадачи раздельно! Две подзадачи с наборами размера *N* / 2, в свою очередь, разделятся на четыре размером *N* / 4, и т. д. Будет ли в этом случае останавливаться рекурсия? Да, потому что последовательное деление конечного *N* пополам непременно выведет это значение за пределы основания рекурсии и вычисление каждой подзадачи остановится.

Вспомним знакомую нам задачу поиска максимума в неупорядоченном массиве из *N* элементов. Напишем вспомогательную функцию `rmax(lo, hi)`, которая ищет наибольшее значение на отрезке массива `A[lo ... hi]`; она-то и будет вызываться рекурсивно[1]. В примере 5.5 вызов `find_max(A)` превращается в вызов `rmax(0, len(A)-1)`, в котором задаются исходные границы поиска — `lo = 0` и `hi = N - 1` (`N` — это размер массива `A`). Основанием для `rmax()` будет единичный размер отрезка — равенство `lo` и `hi`, потому что в наборе из одного элемента максимум не нужно искать. Результатом `rmax()`, наибольшим числом в объединении отрезков, будет наибольший из максимумов в левом и правом отрезке.

---

[1] Название rmax происходит от recursive max. — *Примеч. авт.*

Пример 5.5. Рекурсивный алгоритм поиска максимума в неупорядоченном списке

```
def find_max(A):

  def rmax(lo, hi):
    if lo == hi:                ❷
        return A[lo]

    mid = (lo+hi) // 2          ❸
    L = rmax(lo, mid)           ❹
    R = rmax(mid+1, hi)         ❺
    return max(L, R)            ❻

  return rmax(0, len(A)-1)      ❶
```

❶ Исходный вызов рекурсивной функции с заданием стартовых значений для `lo` и `hi`.

❷ Основание рекурсии: единичный размер отрезка `A[lo … hi]` (равенство `lo` и `hi`), единственный элемент и есть максимум.

❸ Примерная середина отрезка `A[lo … hi]`. Индексы массива могут быть только целыми, так что используем целочисленное деление.

❹ `L` — максимум в левой половине отрезка, `A[lo … mid]`.

❺ `R` — максимум в правой половине отрезка, `A[mid+1 … hi]`.

❻ Максимум на `A[lo … hi]` — это наибольшее среди `L` и `R`.

Функция `rmax(lo, hi)` ищет максимум так: делит отрезок исходного массива от `lo` до `hi` включительно на два отрезка половинной длины, ищет максимумы в них и сравнивает результаты. На рис. 5.7 показано, как работает `rmax(0, 3)` на массиве `A` длиной 4. Для этого решаются две подзадачи — поиск максимума в левой половине массива, `rmax(0, 1)`, и в правой, `rmax(2, 3)`. Поскольку `rmax()` дважды вызывает рекурсивно сама себя, важно знать, *в каком именно месте* функции вычисление приостановилось. Понадобится новая раскраска: серым фоном, как и прежде, мы будем обозначать контекст функции, чье выполнение сейчас приостановлено, и там же вдобавок черным фоном будем отмечать строки, которые *еще предстоит выполнить, когда вычисления возобновятся*.

Места на рис. 5.7 хватило только на три первых рекурсивных вызова, в результате которых был найден максимум левой части массива, 21. Мы видим, что две последние строки в соответствующем контексте все еще помечены черным, напоминая про оставшуюся часть вычислений: продолжением будет очередной рекурсивный вызов, `rmax(2,3)`. Он приведет еще к трем похожим рекурсивным вызовам, а уже после них наконец можно будет вычислить и исходное `rmax(0, 3)`: возвращаемое значение вычислится из `max(21, 20)`.

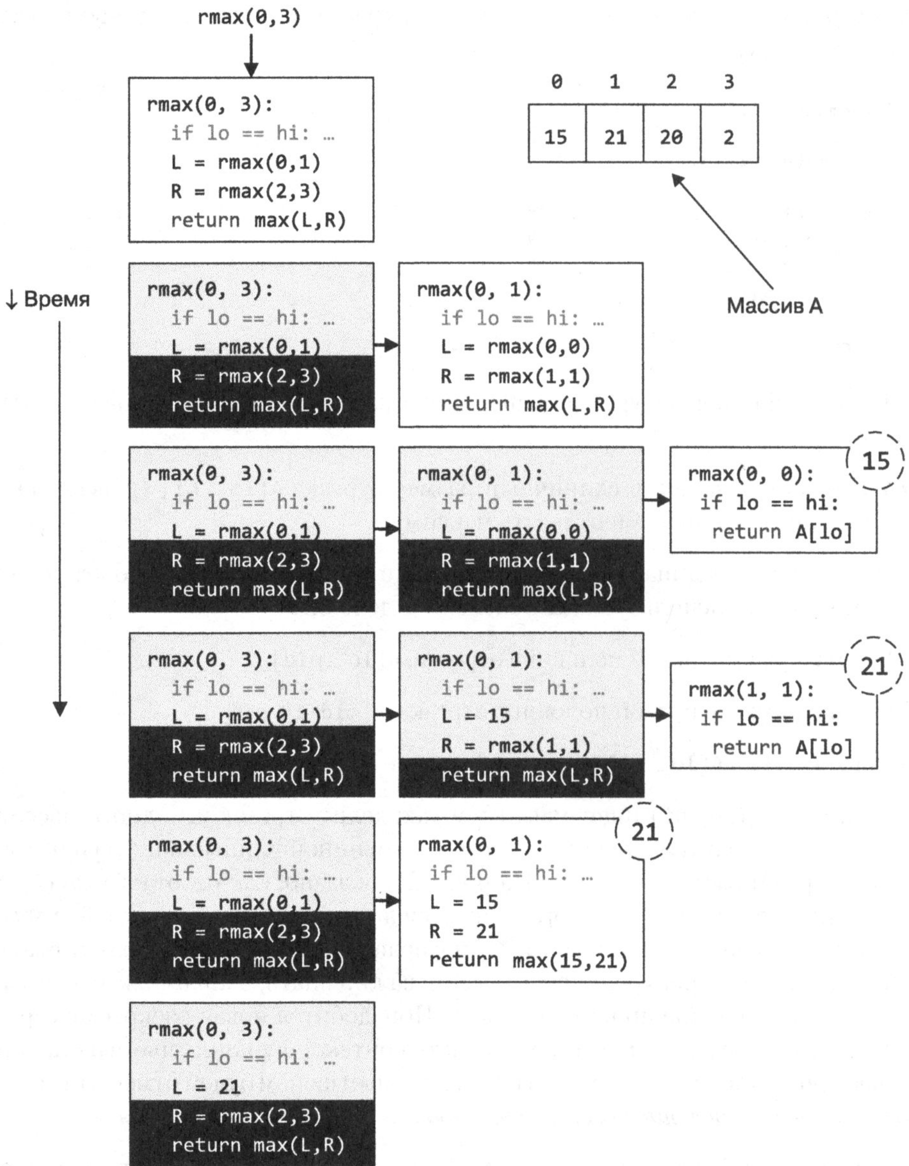


Рис. 5.7. Протокол рекурсивного вызова rmax(0, 3) для массива A = [15, 21, 20, 2]

На рис. 5.8 представлен полный протокол работы `rmax(0, 7)`. Здесь так же, как и в разборе `fact()`, серым обозначена приостановка вычислений: например, во время того, как `rmax(0, 3)` дожидается ответа от рекурсивного вызова `rmax(0, 1)`. Исходный массив делится на все меньшие отрезки до тех пор, пока в очередном

вызове оба параметра rmax() не окажутся равными. На рисунке мы видим восемь таких случаев, потому что в массиве $N = 8$ элементов. Во всех восьми случаях мы имеем дело с основанием рекурсии, и, следовательно, дальнейших вызовов не происходит. Максимум массива A, очевидно, 24, и на рисунке отдельно отмечены рекурсивные вызовы, которые возвращают этот максимум.

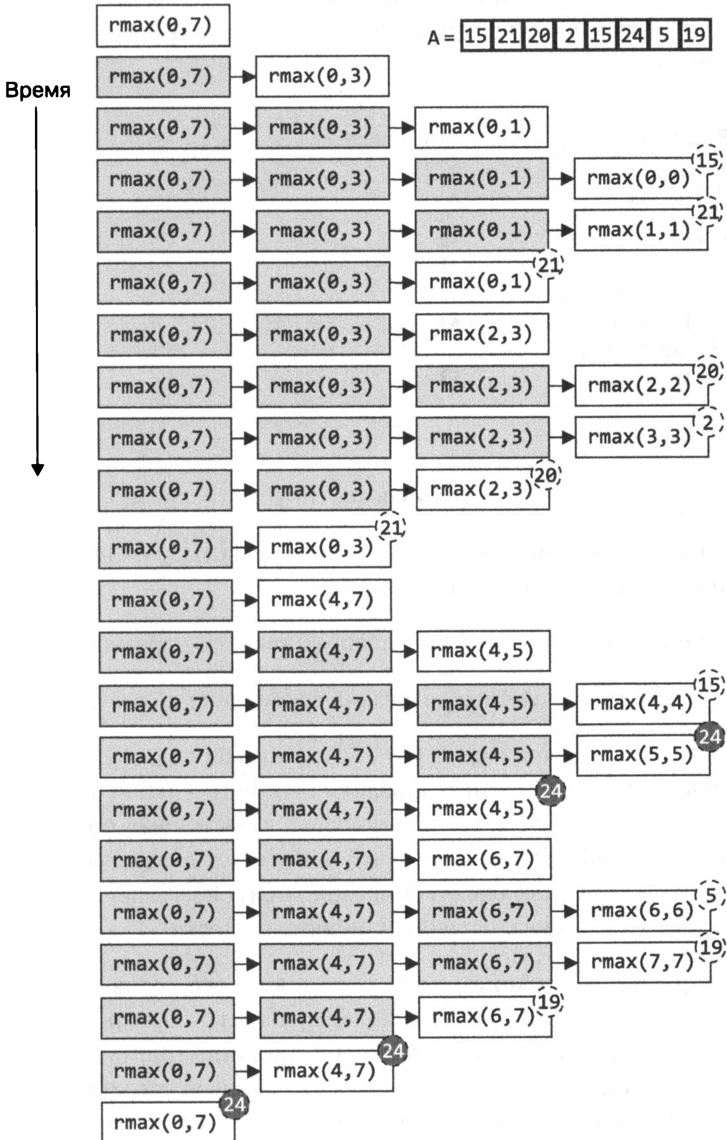


Рис. 5.8. Полный протокол рекурсивного вызова rmax(0, 7)

Сортировка слиянием

Смотрим мы на наши примеры, и возникает вопрос: а нельзя ли и для сортировки применить методику «разделяй и властвуй»? Идея в том, чтобы, как это показано в примере 5.6, рекурсивно отсортировать левую половину массива, затем рекурсивно отсортировать правую, а потом как-нибудь быстренько объединить *уже отсортированные половины*. Тогда и весь массив окажется упорядоченным.

Пример 5.6. Идея рекурсивной сортировки

```
def sort(A):

  def rsort(lo, hi):            ❶
    if hi > lo:                 ❷
      mid = (lo+hi) // 2
      rsort(lo, mid)            ❸
      rsort(mid+1, hi)
      merge(lo, mid, hi)        ❹

  rsort(0, len(A)-1)
```

❶ Вспомогательная функция, которая сортирует отрезок `A[lo … hi]`.

❷ Пустой или единичный отрезок сортировать не надо, это основание рекурсии. В противном случае нужно вычислять дальше.

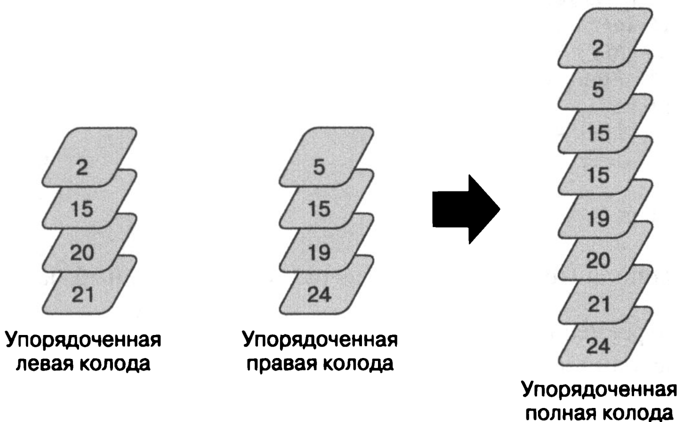
❸ Рекурсивные вызовы: сортировка левого и правого отрезков массива.

❹ Слияние двух упорядоченных отрезков и замена их на результат слияния.

Здесь мы воспользовались схемой, которую придумали для функции `find_max()` и показали в примере 5.5. Если реализация вспомогательной функции `merge()` окажется достаточно простой, у нас получится настоящая **сортировка слиянием** — рекурсивный алгоритм, который использует дополнительную память, но в результате сортирует исходный массив за $O(N \log N)$ действий, то есть мы наконец-то преодолеем квадратичный порог сложности!

Главное в **сортировке слиянием** — это функция `merge()`, слияние двух отрезков массива «по месту», когда отсортированная правая и отсортированная левая половины массива заменяются результатом их слияния. Принцип работы `merge()` знаком всем, кто когда-либо превращал две упорядоченные колоды карт в одну, тоже упорядоченную. Наглядно этот принцип показан на рис. 5.9. «Карты» на этом рисунке — просто карточки с номерами.

Если у нас есть две упорядоченные колоды, сделать из них одну с сохранением порядка можно так. Сравним две карты на вершинах колоды, выберем наименьшую из них и положим на место, где будет формироваться полная колода. Все остальные карты мы будем *подсовывать* под полную колоду, чтобы сохранить тот же порядок (в нашем случае — возрастания снизу вверх). В примере на первых двух шагах сначала подсовываем 2 из левой колоды, а затем — 5 из правой. Теперь в обеих колодах значения одинаковы — 15; возьмем сначала из левой, затем из правой. Продолжим повторять эти действия до тех пор, пока какая-нибудь из колод не опустеет — это будет предпоследний шаг слияния. На последнем шаге просто заберем все карты из оставшейся непустой колоды — они и так не меньше всех остальных и уже отсортированы.



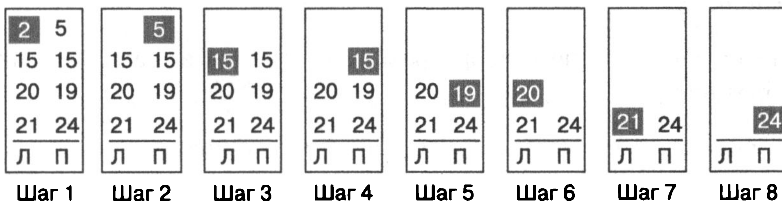


Рис. 5.9. Слияние двух колод в одну

Обратите внимание на то, что нам понадобилось место, где будет формироваться полная колода. Это означает, что при программировании такого алгоритма понадобится дополнительное хранилище данных. Самым эффективным для **сортировки слиянием** оказался самый простой способ: завести еще один массив размером с исходный и хранить там подлежащие сортировке отрезки, выполняя слияние обратно в исходный массив. Вариант функции `merge()` приведен в примере 5.7.

Пример 5.7. Рекурсивная реализация сортировки слиянием

```
def merge_sort(A):
    aux = [None] * len(A)                                              ❶

    def rsort(lo, hi):
        if hi > lo:                                                    ❷
          mid = (lo+hi) // 2
          rsort(lo, mid)                                               ❸
          rsort(mid+1, hi)
          merge(lo, mid, hi)

    def merge(lo, mid, hi):
        aux[lo:hi+1] = A[lo:hi+1]                                      ❹
        left, right = lo, mid+1                                        ❺
        for i in range(lo, hi+1):
            if left > mid or right <= hi and aux[right] < aux[left]:   ❻
                A[i] = aux[right]
                right += 1                                             ❼
            else:                                                      ❽
                A[i] = aux[left]
                left += 1

    rsort(0, len(A)-1)                                                 ❾
```

❶ Заводим дополнительное хранилище размером с исходный массив.

❷ Основание рекурсии — отрезок не более чем единичной длины; в противном случае вычисления продолжаются.

❸ Два рекурсивных вызова: сортировка левой и правой половин отрезка, а затем их слияние.

❹ Копируем элементы двух отсортированных отрезков в хранилище: их предстоит объединить.

❺ Задаем начало левого отрезка, `left`, и правого — `right`.

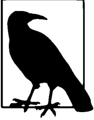
❻ Элемент правого отрезка мы берем в двух случаях:

- левый отрезок уже пуст;
- оба отрезка не пусты, и элемент в начале правого отрезка меньше.

❼ Сдвигаем начало правого отрезка на следующий его элемент.

❽ В двух других случаях — правый отрезок пуст или оба не пусты и левый элемент меньше — берем элемент и сдвигаем начало левого отрезка.

❾ Исходный вызов рекурсивной функции.



В условном операторе внутри цикла функции merge() проявляется свойство ленивых вычислений Python: правый операнд, B, логического выражения A or B вычисляется, только если значение A ложно, а выражения A and B – только если A истинно. В нашем случае нельзя проверять right <= hi и aux[right] < aux[left] одновременно: если первое сравнение ложно, второе, скорее всего, просто ошибочно, и его не нужно даже пытаться интерпретировать. Именно так и работает операция and! То же самое и со сравнением left > mid: если оно истинно, не стоит пытаться выполнить остальные сравнения, но именно так (с учетом приоритета and над or) и работает операция or.

На рис. 5.10 по шагам показано, как работает `merge()`. На первом шаге вызова `merge(lo, mid, hi)` элементы массива `A[lo … hi]` копируются в `aux[lo … hi]` – потому что именно этот отрезок и надо отсортировать.

aux: указатели	aux	A: указатели	A
lo (0), mid (3), hi (7)	- - - - - - - -	mid (3), hi (7)	2 15 20 21 5 15 19 24
	2 15 20 21 5 15 19 24		2 15 20 21 5 15 19 24
left (0), right (4)	**2** 15 20 21 **5** 15 19 24	i (0)	**2** 15 20 21 5 15 19 24
left (1), right (4)	2 **15** 20 21 **5** 15 19 24	i (1)	2 **5** 20 21 5 15 19 24
left (1), right (5)	2 **15** 20 21 5 **15** 19 24	i (2)	2 5 **15** 21 5 15 19 24
left (2), right (5)	2 15 **20** 21 5 **15** 19 24	i (3)	2 5 15 **15** 5 15 19 24
left (2), right (6)	2 15 **20** 21 5 15 **19** 24	i (4)	2 5 15 15 **19** 15 19 24
left (2), right (7)	2 15 **20** 21 5 15 19 **24**	i (5)	2 5 15 15 19 **20** 19 24
left (3), right (7)	2 15 20 **21** 5 15 19 **24**	i (6)	2 5 15 15 19 20 **21** 24
left (4), right (7)	2 15 20 21 5 15 19 **24**	i (7)	2 5 15 15 19 20 21 **24**

Рис. 5.10. Слияние отрезков массива по четыре элемента в каждом

Цикл по `i` выполнится восемь раз, потому что всего в массиве восемь элементов, из которых состоят два объединяемых отрезка. Начиная с третьего ряда на рис. 5.10, мы можем обращаться к переменным `left`, `right` и `i`, которые хранят интересующие нас индексы.

- В переменной `left` хранится индекс очередного элемента левого отрезка. Если он окажется меньшим, возьмем именно его.
- В переменной `right` хранится индекс очередного элемента правого отрезка, он тоже может участвовать в слиянии.
- В переменной `i` хранится индекс элемента основного массива, куда мы записываем значения в порядке возрастания. На последнем шаге весь отрезок `A[li … hi]` окажется отсортированным.

В цикле `for` при необходимости сравниваются два значения из `aux` (выделены серым на рис. 5.10), выбирается меньшее, и оно записывается в `A[i]`. Переменная `i` увеличивается на каждом шаге, а `left` и `right` — только когда соответствующее значение, `aux[left]` или `aux[right]`, оказывается очередным наименьшим и переносится в `A`. Время работы `merge()` прямо пропорционально сумме длин обоих отрезков, то есть `hi - lo + 1`.

Сортировка слиянием — отличный пример алгоритма в стиле «разделяй и властвуй», быстродействие которого оценивается как $O(N \log N)$. В целом произвольную задачу можно решить за $O(N \log N)$ действий, если в ней соблюдаются такие условия.

1. Исходный набор данных можно поделить пополам и независимо решить две получившиеся подзадачи. Допустимо делить не строго пополам, различие в длине на небольшую константу погоды не делает.
2. Если в основании рекурсии лежит фиксированное количество действий или не лежит никаких действий, как в **сортировке слиянием**.
3. Если на каждом шаге рекурсии — и до и после разделения данных и рекурсивного вызова — количество выполняемых действий прямо пропорционально объему обрабатываемых данных. Например, в функции `merge()`, которая выполняется после рекурсивного вызова, количество итераций цикла `for` равно размеру сортируемого отрезка.

Стоит вспомнить, что *время* работы программы зависит от конкретной реализации и его нужно *измерять*. Таким образом, время работы не может являться свойством *задачи*. А вот количество действий в алгоритме — может. Если адекватно оценить количество действий, можно предположить, что быстродействие будет удовлетворять той же оценке.

Быстрая сортировка

Быстрая сортировка — еще один весьма известный и очень эффективный алгоритм в стиле «разделяй и властвуй»[1]. Идея в том, чтобы выбрать в массиве некоторый элемент p (назовем его *опорным*) и *поставить его на свое место*, туда, где он окажется в отсортированном массиве. Для этого предлагается переставить местами элементы массива так, чтобы слева от p оказались только элементы, не большие чем p, а справа — только не меньшие. Затем надо рекурсивно отсортировать два отрезка массива слева и справа от позиции p. На рис. 5.11 показан такой массив; можно проверить, что он и правда «разгорожен» элементом p.

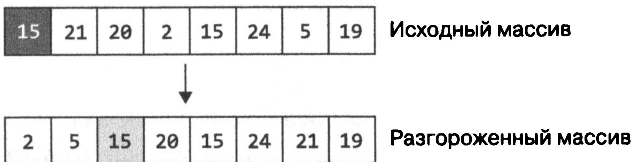


Рис. 5.11. Результат работы partition(A, 0, 7 ,0) с опорным элементом A[0]

Это кажется каким-то трюком на грани мошенничества: как можно знать заранее, куда попадет p в отсортированном массиве? Оказывается, массив A для этого не нужно сортировать — достаточно переставить местами некоторые элементы по отношению к p. В действительности мы уже видели алгоритм, который умеет это делать: функция `partition()` встречалась в тренировочных заданиях к главе 1. Именно она и превратила массив на рис. 5.11 в два отрезка: левый состоит из двух элементов, а правый — из семи. Теперь достаточно рекурсивно применить к этим отрезкам **быструю сортировку**, как в примере 5.8, и задача будет решена.

Пример 5.8. Рекурсивная реализация быстрой сортировки

```
def quick_sort(A):

  def qsort(lo, hi):
    if hi > lo:                                         ❶
      pivot_idx = lo                                    ❷
      location = partition(A, lo, hi, pivot_idx)        ❸
      qsort(lo, location-1)                             ❹
      qsort(location+1, hi)

  qsort(0, len(A)-1)                                    ❺
```

[1] Изобрел ее Тони Хоар в далеком 1959 году, так что этому алгоритму уже за 60! — *Примеч. авт.*

❶ Основание рекурсии: если отрезок не длиннее одного элемента, сортировать его не надо, иначе работу следует продолжить.

❷ Опорным элементом `p` массива выберем `A[lo]`.

❸ Переставим элементы `A` и вернем позицию `p` в нем:

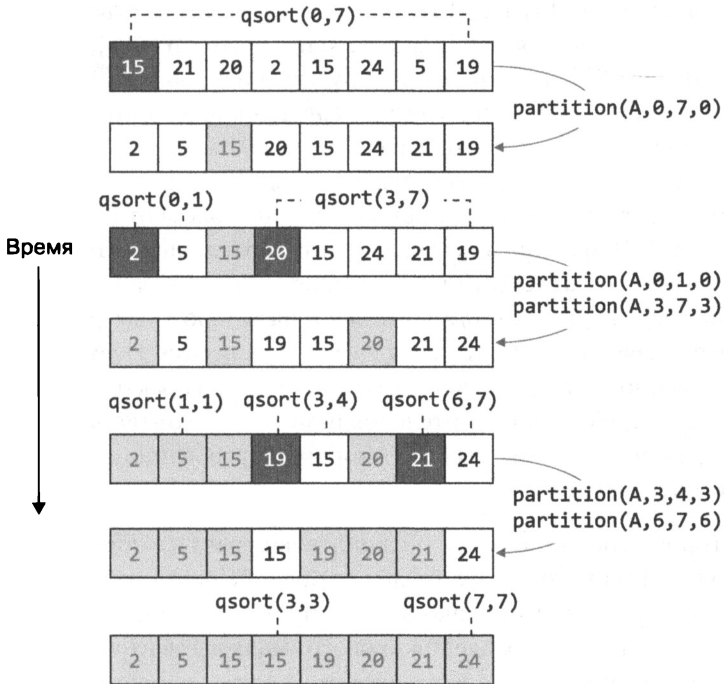
- `location` — это позиция `p`;
- все элементы левого отрезка `A[lo … location-1]` не больше `p`;
- все элементы правого отрезка `A[location+1 … hi]` не меньше `p`.

❹ Рекурсивный вызов: отсортируем элементы обоих отрезков *по месту*, а `p` и так уже находится где следует — в `A[location]`.

❺ Исходный вызов рекурсивной функции.

Быстрая сортировка — красивое рекурсивное решение задачи, но эффективность каждого ее шага зависит от того, где в отсортированном массиве располагается опорный элемент. Например, если так случится, что функция `partition()` выберет в качестве опорного *наименьший* элемент отрезка `A[lo … hi]` длиной N элементов, результатом будет пустой левый отрезок и правый отрезок длиной $N - 1$. Если на каждом шагу объем данных уменьшается только на 1, как в **сортировке вставками** или **сортировке выбором**, сложность алгоритма станет такой же неэффективной — $O(N^2)$. Верхняя часть рис. 5.12 отображает основные шаги **быстрой сортировки** массива с рис. 5.11. В нижней части перечислены все рекурсивные вызовы. В этой части справа показан сам массив `A`, и можно понаблюдать, как меняются его значения при каждом вызове. Здесь опорной точкой отрезка `A[lo … hi]` всегда выбирается `A[lo]`, так что функция `partition()` всегда вызывается с параметрами `lo, hi, lo`. С течением времени — сверху вниз — становится понятно, что каждый вызов `partition()` сопровождается одним или двумя рекурсивными вызовами `qsort()`. Например, `partition(0, 7, 0)` ставит на свое место в массиве `A` число `15` (это место выделено серым), после чего дважды вызывается рекурсивно сначала `qsort(0, 1)` на левом отрезке, а затем — `qsort(3, 7)` на правом. Понятно, что `qsort(3, 7)` запустится только после того, как завершит работу `qsort(0, 1)`.

Каждый вызов `partition()` приводит к тому, что некоторый элемент массива оказывается на своем месте — на рисунке он подкрашивается серым. Если при вызове `qsort(lo, hi)` отрезок оказывается единичной длины, то есть `lo == hi`, соответствующий элемент уже на своем месте и тоже подкрашивается серым.



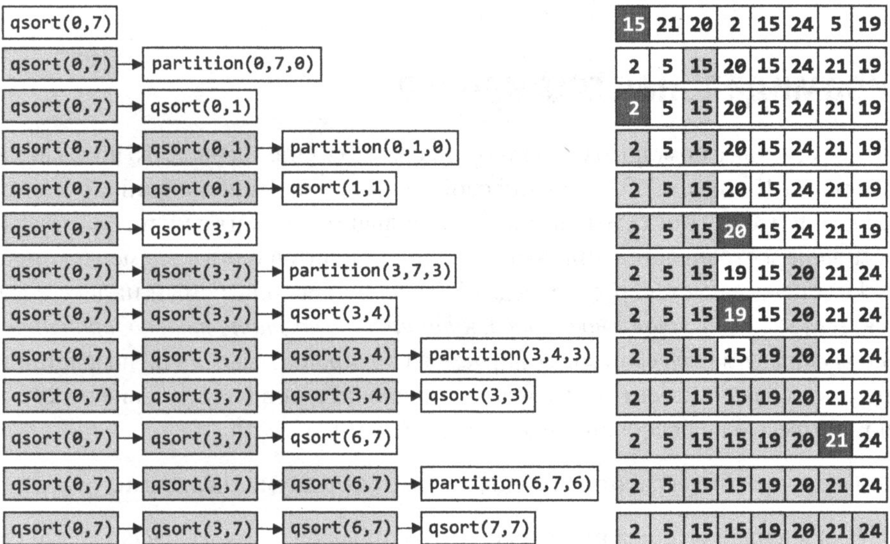


Рис. 5.12. Полный протокол рекурсивного вызова функции quicksort()

Если вызов `partition(lo, hi, lo)` привел только к единственному вызову `qsort()`, значит, опорный элемент оказался или в позиции `A[lo]`, или в позиции `A[hi]` — а это означает, что длина сортируемого отрезка уменьшилась лишь на 1. Таким образом, реализация **быстрой сортировки** из примера 5.8 покажет рекордно низкую — $O(N^2)$ — производительность на... уже отсортированном массиве! Поскольку уже отсортированные данные встречаются часто, **быструю сортировку** пытаются улучшить, выбирая опорный элемент случайно из диапазона `A[lo ... hi]`. В примере для этого достаточно заменить `pivot_idx = lo` на `pivot_idx = random.randint(lo, hi)`. Десятилетия исследований показали, что теоретическая вероятность *наихудшего случая*, на котором **быстрая сортировка** показывает производительность всего $O(N^2)$, есть всегда. Несмотря на такой недостаток, именно этот алгоритм часто используют на практике — потому, например, что он не требует дополнительной памяти (как **сортировка слиянием**). Так-то **быстрая сортировка** отвечает всем трем требованиям к алгоритму со сложностью $O(N \log N)$.

Есть и другой способ добиться производительности $O(N \log N)$: выполнить N шагов, каждый из которых будет не сложнее $O(\log N)$. В **пирамидальной сортировке** используется уже знакомая нам структура данных — куча, и можно ожидать, что N операций добавления произвольных элементов в кучу и N операций снятия (уже упорядоченных элементов) при сложности этих операций не более $\log N$ и дадут нам искомое $O(N \log N)$.

Пирамидальная сортировка

Разберемся, как с помощью кучи отсортировать массив. На рис. 5.13 показан массив с рис. 4.17, то есть обладающий свойствами кучи. Наибольший его элемент находится в `A[1]`. Когда мы снимаем наибольший элемент с кучи, содержимое массива перестраивается, при этом в куче становится одним элементом меньше. И тут оказывается, что последний, теперь неиспользуемый индекс в массиве — `A[18]` — это *то самое место, куда надо поместить только что снятый наибольший элемент*, если мы собираемся сортировать массив по возрастанию. Еще раз снимем элемент — это будет второе максимальное значение из кучи — и поместим его на освободившуюся позицию `A[17]`.

Чтобы эта красота заработала, надо разобраться с двумя нюансами.

- Куча в нашей реализации не использует нулевой элемент массива (это сделано для наглядности алгоритма), так что для хранения N элементов понадобится массив размером $N + 1$.

- Изначально куча пуста, и мы умеем добавлять в нее только по одному элементу. Возможно, имеет смысл научиться добавлять все *N* сразу, поскольку мы заранее знаем размер сортируемого массива.

Рис. 5.13. Схема использования двоичной кучи с частичным убыванием для сортировки

Разберемся, как вычислять индексы в сортируемом массиве. Куча из 18 элементов (которые показаны на рис. 5.13) использовала массив из 19 элементов. Можно было считать, что индексация массива начинается с 1, то есть `A[1]` — это начальный элемент кучи, а `A[N]` — последний. В примере 5.9 мы переопределили `less(i, j)` и `swap(i, j)`: когда `i` и `j` используются как индексы массива `A`, из них вычитается по единице. Таким образом, в самом алгоритме индексация у нас идет с единицы, а для массива `A` она превращается в индексацию с нуля. Наибольшее значение кучи находится в `A[0]`; вызов `swap(1, N)` из метода `sort()` на самом деле меняет местами `A[0]` и `A[N-1]`; так же преобразует индексы и `less()`. Это небольшое изменение позволяет нам не менять метод `sink()`, а метод `swim()` **пирамидальная сортировка** не использует.

Пример 5.9. Частичная реализация пирамидальной сортировки

```
class HeapSort:
  def __init__(self, A):
    self.A = A
    self.N = len(A)
```

```
    for k in range(self.N//2, 0, -1):          ❷
      self.sink(k)

  def sort(self):
    while self.N > 1:                           ❸
      self.swap(1, self.N)                      ❹
      self.N -= 1                               ❺
      self.sink(1)                              ❻

  def less(self, i, j):
    return self.A[i-1] < self.A[j-1]            ❶

  def swap(self, i, j):
    self.A[i-1], self.A[j-1] = self.A[j-1], self.A[i-1]
```

❶ Обе функции — `less()` и `swap()` — вычитают по единице из индексов, при этом можно считать, что индексация в них идет с 1 и $i\,//\,2$ — это индекс родителя для i-й ячейки.

❷ Все элементы, у которых есть хотя бы один потомок, могут нарушать частичный порядок, поэтому их следует утопить, начиная с последнего (в позиции $N//\,2$) и заканчивая первым. Наш сортируемый массив `A` при этом превращается в двоичную кучу с убыванием.

❸ Цикл `while` заканчивается с опустошением кучи, в которой лежат сортируемые элементы.

❹ Обменяем местами наибольшее и последнее значение в куче.

❺ Тем самым мы сняли наибольшее значение, теперь нужно уменьшить размер кучи (иначе `sink()` может работать неправильно).

❻ На вершине кучи лежит только что добавленное значение. Его следует утопить на место, дабы восстановить частичный порядок.

Самое важное происходит в начале **пирамидальной сортировки**: нужно превратить сортируемый массив в двоичную кучу с убыванием. Для этого в конструкторе `HeapSort` есть цикл `for` — его работа показана на рис. 5.14. На превращение конкретного массива в кучу ушло всего 23 сравнения и пять обменов. Цикл обрабатывает все элементы, *у которых есть хотя бы один потомок*, начиная с позиции $N//\,2$ вплоть до вершины кучи. По индексу `k` элементы перебираются в обратном порядке, и для каждого вызывается `sink()` — таким образом, в куче в конце концов восстановится частичный порядок. Элемент в позиции `k` отмечен на рис. 5.14 полужирной рамкой.

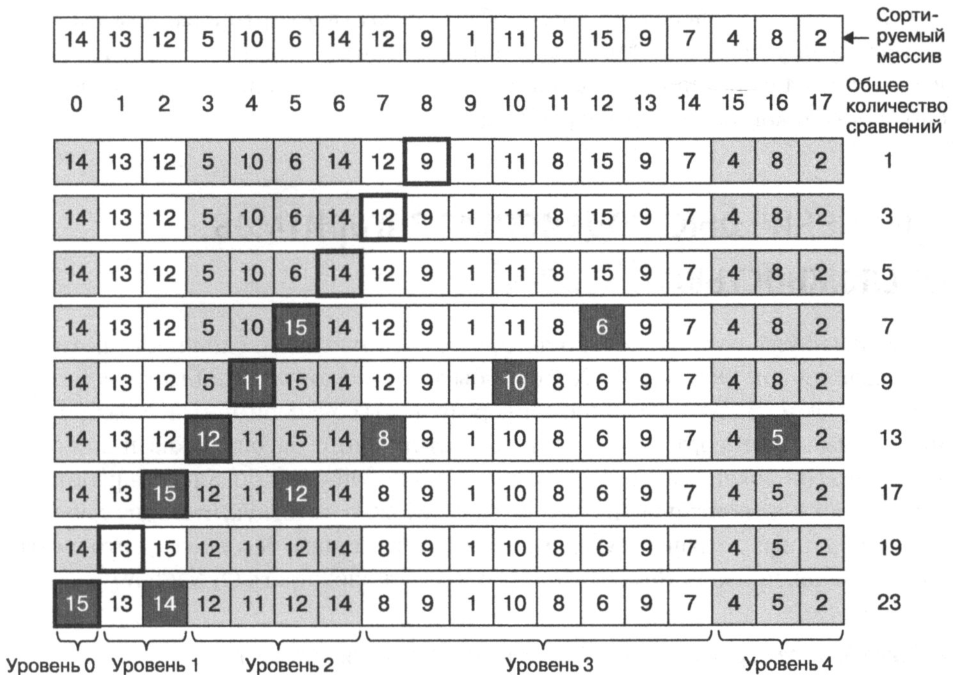


Рис. 5.14. Преобразование массива в двоичную кучу с убыванием

Часть элементов не нуждаются в погружении — у них нет потомков. Половина из оставшихся погружаются не более чем на один уровень, еще в два раза меньше — не более чем на два и т. д., вплоть до вершины, которая может погрузиться максимум на $\log_2 N$ уровней, а на каждый уровень погружения тратится не более двух сравнений. Аккуратный математический анализ показывает, что преобразование произвольного массива в кучу неожиданно выполняет не больше $2N$ сравнений в *худшем случае*. На рис. 5.14 хорошо видно, как медленно (хотя и неуклонно) увеличивается количество сравнений. Элементы, находящиеся на одном и том же уровне кучи, как и в предыдущем примере, отмечены либо серым, либо белым — тогда становится видно, как при обмене они переходят с уровня на уровень.

Последняя строка — это уже настоящая двоичная куча (на самом деле в точности такая же, что и на рис. 4.16, только индексирование в ней идет с нуля, а не с единицы, чтобы использовать все N ячеек). Теперь метод `sort()` из примера 5.9 может снимать наибольшее значение с кучи и помещать его в ее конец (пользуясь уловкой с рис. 5.13). При этом конец постоянно уменьшающейся кучи — это

и есть то место, где очередной элемент будет стоять в отсортированном массиве. Размер кучи уменьшает сам `sort()`, и он же топит элемент, некогда стоявший в ее конце, — так восстанавливается ее частичный порядок. В главе 4 мы оценили сложность такой процедуры, как O(log *N*).

Сравнение быстродействия алгоритмов со сложностью O(N log N)

Итак, у нас уже есть несколько алгоритмов сортировки, сложность которых оценивается как O(*N* log *N*). Хорошо бы теперь сравнить их быстродействие друг с другом. В табл. 5.1 приведены результаты экспериментов с различными реализациями сортировки. Размер сортируемого массива указан в первой колонке — он увеличивается вдвое с каждой новой строкой, а время работы — в колонках, соответствующих алгоритмам. Можно заметить, что при удвоении входных данных затраченное время растет чуть больше чем вдвое — это и есть, грубо говоря, признак того, что алгоритм имеет сложность O(*N* log *N*).

Таблица 5.1. Время работы различных алгоритмов сортировки в секундах

N	Слиянием	Быстрая	Пирамидальная	Тима	Python
1024	0.002	0.002	0.006	0.002	0.000
2048	0.004	0.004	0.014	0.005	0.000
4096	0.009	0.008	0.032	0.011	0.000
8192	0.020	0.017	0.073	0.023	0.001
16 384	0.042	0.037	0.160	0.049	0.002
32 768	0.090	0.080	0.344	0.103	0.004
65 536	0.190	0.166	0.751	0.219	0.008
131 072	0.402	0.358	1.624	0.458	0.017
262 144	0.854	0.746	3.486	0.970	0.039
524 288	1.864	1.659	8.144	2.105	0.096
1 048 576	3.920	3.330	16.121	4.564	0.243

Время работы разных алгоритмов на одном и том же наборе данных в каждой строке различно. В главе 2 мы рассматривали понятие мультипликативной по-

стоянной, которая может отличать быстродействие алгоритмов одного класса. Подтверждение этому — последние строки таблицы: начиная с некоторого достаточного размера массива **быстрая сортировка** на 15 % превосходит **сортировку слиянием**, а **пирамидальная сортировка** медленнее ее вчетверо.

Две последние колонки табл. 5.1 — это тест производительности еще двух алгоритмов. Предпоследняя колонка — это **сортировка Тима**: в 2002 году Тим Петерс реализовал этот метод сортировки в качестве стандартного для Python. Метод пришелся по душе многим разработчикам языков программирования и теперь используется в большинстве современных языков — в самом Python, в Java, Swift, Rust и т. д., по сути превратившись в стандарт. В таблице рассматривается упрощенный вариант **сортировки Тима** — по этой причине он слегка уступает в производительности **быстрой сортировке**, тем не менее он тоже показывает быстродействие $O(N \log N)$. Последняя колонка — это работа встроенной в Python функции `sort()` на типе данных `list`. Функция `sort()` также реализует **сортировку Тима**, но в полном варианте, допускающем массу оптимизаций. К тому же она встроенная и с такими простыми объектами, как `list`, работает непосредственно — в результате чего она раз в 15 быстрее **быстрой сортировки**. Саму **сортировку Тима** имеет смысл рассмотреть отдельно: в ней объединены достоинства сразу двух алгоритмов, что делает ее весьма эффективной.

Сортировка Тима

В **сортировке Тима** используется подход **сортировки вставками** с применением функции `merge()` из **сортировки слиянием**, но в несколько иных условиях. На повседневных данных получившийся алгоритм обгоняет все уже известные нам алгоритмы сортировки! В частности, **сортировка Тима** пользуется тем, что некоторые отрезки массива уже отсортированы — это позволяет основательно поднять производительность.

Когда сортируешь относительно небольшой отрезок, разница между алгоритмом $O(N^2)$ и $O(N \log N)$ может оказаться не в пользу последнего! В самом деле, допустим, что точная (с учетом мультипликативных коэффициентов) оценка сложности алгоритма `A` — это $2\ N^2$ действий, а алгоритма `B` — это $50\ N \log_2 N$ действий. Получится, что отрезки длиной менее 190 элементов выгоднее сортировать алгоритмом `A`, а не алгоритмом `B`, потому что $2 \times 189^2 = 71\ 442 <$ $< 71\ 841 \approx 50 \times 189 \times \log_2 189$. Как видно из примера 5.10, в **сортировке Тима** так и происходит. Сначала сортируются вставками все N / `size` отрезков длиной `size`, при этом `size` заранее вычисляется функцией `compute_min_run()`. Поскольку обычно `size` — это число в диапазоне от 32 до 64, можно считать его

константой, не зависящей от N[1]. В результате мы получаем массив с упорядоченными отрезками, достаточно длинными для того, чтобы на них эффективно работала вспомогательная функция `merge()` из **сортировки слиянием**, которая два отсортированных отрезка сливает в один.

Пример 5.10. Базовая реализация сортировки Тима

```
def tim_sort(A):
  N = len(A)
  if N < 64:                                  ❶
    insertion_sort(A, 0, N-1)
    return

  size = compute_min_run(N)                   ❷
  for lo in range(0, N, size):                ❸
    insertion_sort(A, lo, min(lo+size-1, N-1))

  aux = [None]*N                              ❹
  while size < N:
    for lo in range(0, N, 2*size):
      mid = min(lo + size - 1, N-1)           ❺
      hi = min(lo + 2*size - 1, N-1)
      merge(A, lo, mid, hi, aux)              ❻

    size = 2 * size                           ❼
```

❶ Небольшие массивы можно сразу сортировать вставками.

❷ Вычислим `size` — размер отрезков, которые мы будем сортировать вставками (обычно это целое в диапазоне от 32 до 64).

❸ Отсортируем вставками все отрезки `A[lo ... lo+size-1]` (не забудем, что последний из них может быть короче `size`).

❹ Для сортировки слиянием требуется дополнительная память размером с исходный массив.

❺ Вычислим границы отрезков, которые не надо объединять, `A[lo ... mid]` и `A[mid+1 ... hi]`; не забудем о том, что отрезок может оказаться меньшей длины.

❻ Объединим два отрезка и получим отсортированный `A[lo ... hi]`; в качестве дополнительной памяти используем `aux`.

[1] Одна из оптимизаций состоит, например, в том, что, видя N равным 80, compute_min_run() вернет size == 40 — это уменьшит суммарное время сортировки вставками. — *Примеч. пер.*

❼ На следующем проходе цикла `while`, после того как все отрезки длиной `size` попарно объединены, начнем объединять отрезки вдвое большего размера.

Дополнительный массив `aux` создается единожды, а затем используется в каждом вызове `merge()`. Полная реализация **сортировки Тима** намного сложнее, чем наша. В ней сразу определяются отрезки, уже упорядоченные по возрастанию или убыванию, и функция `merge()` работает не с изолированными элементами, как у нас, а сразу с отрезками. На рис. 5.15 показаны основные преобразования сортируемого массива. Сортировку изучали так долго и так основательно, так часто применяли — тем удивительнее то, что самым эффективным для повседневных задач оказался новый алгоритм, изобретенный в XXI веке.

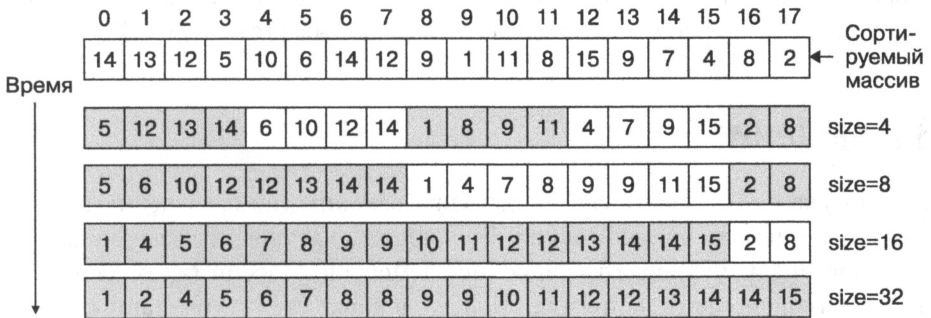


Рис. 5.15. Как меняется массив во время сортировки Тима с начальным размером отрезка 4

Размер `min_run` на рис. 5.15 равен всего четырем — так проще рассмотреть работу сортировки Тима. На первом этапе четыре отрезка длиной 4 сортируются **вставками** (последний отрезок меньше, он содержит только два элемента — 2 и 8). Отсортированные отрезки помечены на рисунке белым и серым фоном поочередно. Количество таких отрезков — N / `size` или на один больше, если N не делится на `size` нацело. Мы помним, что сортировка `size` элементов вставками требует `size` × (`size` – 1) / 2 действий, а поскольку самих отрезков у нас N / `size` штук, время работы первого этапа будет прямо пропорционально N × (`size` – 1) / 2. Значение `size` в алгоритме — константа, так что сложность первого этапа можно оценить как $O(N)$.

На втором этапе мы попарно объединяем соседние отрезки. Разбирая работу **сортировки слияниями**, мы нашли, что общее время работы всех вызовов `merge()` на одном шаге пропорционально N. После первого прохода цикла `while` размер

отсортированных отрезков удвоился до 8 — это показано на рис. 5.15 соответствующим фоном. В нашем примере за три прохода `size` вырастает с 4 до 32, а это уже больше, чем N. В общем случае, если исходный размер равен `size` и вся сортировка выполняется за k проходов цикла `while`, этот цикл заканчивается, когда `size` $\times 2^k$ превысит N, то есть когда N / `size` $< 2^k$.

Чтобы оценить k, возьмем логарифм обеих частей неравенства и получим $k > \log_2$ (N / `size`). Логарифм частного — это разность логарифмов, так что $k > \log_2 N - \log_2$ `size`. Вспомним, что `size` — это константа, и получим, что k — это наименьшее целое число, которое не меньше $\log_2 N$ за вычетом небольшой константы.

Подытожим: первый этап — с **сортировкой вставками** — мы оценили как $O(N)$; второй — со слияниями — как $O(N \times k)$, где k не превосходит $\log_2 N$. Вторая оценка доминирует над первой, так что общая оценка оказывается $O(N \log N)$.

Заключение

Сортировка — одна из основных задач в программировании, изучению которой посвящено немало времени и сил. Большинство встроенных типов данных в Python предусматривают сравнение элементов. Массив таких элементов можно отсортировать напрямую. Более сложные структуры данных могут не иметь функции сравнения, либо алгоритм сравнения по умолчанию может быть неподходящим — например, довольно бессмысленно сравнивать координаты точек *лексикографически*, то есть сначала сравнивать их абсциссы, а если те равны — то ординаты. В этом случае к каждому элементу можно применить функцию-ключ, которая вернет значение простого типа, после чего сортировать исходный массив, сравнивая ключи элементов. Например, ключом для точки может быть расстояние от центра координат — тогда сортировка будет по степени удаления точки от центра.

Вот что в этой главе вы узнали про сортировку.

- Простейшие алгоритмы сортировки имеют сложность $O(N^2)$, и это делает их совершенно непригодными для упорядочения больших объемов данных.
- Если задачу можно разбить на несколько подзадач, это отличный повод применить рекурсию.
- **Сортировка слиянием** и **пирамидальная сортировка** имеют одинаковую сложность $O(N \log N)$, хотя достигается это различными путями.

- В отличие от **сортировки слиянием быстрая сортировка** не требует дополнительной памяти и тоже показывает быстродействие $O(N \log N)$.
- Во многих языках программирования алгоритм сортировки по умолчанию — это **сортировка Тима**, некогда изобретенная специально для Python.

Тренировочные задания

1. Напишите рекурсивную функцию `count(A, t)`, которая возвращает количество вхождений элемента `t` в массив `A`. Рекурсивный вызов должен быть устроен так же, как в `find_max(A)`.
2. Напишите функцию `num_swaps(A)`, на вход которой подается массив размером N, содержащий набор различных целых чисел от 0 до $N - 1$ в произвольном порядке. Функция должна возвращать наименьшее количество обменов двух элементов массива, которое необходимо для его сортировки. Сам массив сортировать не надо, нужно лишь количество обменов.

   Дополните алгоритм так, чтобы можно было обрабатывать массив из N различных произвольных объектов, используя для этого хеш-таблицу из главы 3. Проверьте, что для сортировки массива с рис. 5.1 требуется пять обменов.
3. Сколько всего сравнений требуется для вычисления наибольшего значения в неупорядоченном массиве из N элементов с помощью рекурсивной функции `find_max(A)`? Функция `largest(A)` из главы 1 требует больше сравнений или меньше?
4. Функция `merge()` в **сортировке слиянием** в какой-то момент оказывается в ситуации, когда один из объединяемых отрезков закончился, а другой — еще нет. Наша функция продолжает обрабатывать оставшийся отрезок поэлементно. Исправьте это поведение: пускай в данный момент функция сразу заменяет один сегмент массива другим — так, как мы это делали в `aux[lo:hi+1] = A[lo:hi+1]`. Проведите серию испытаний и определите, приводит ли это к повышению быстродействия и если да, то насколько.
5. Напишите рекурсивную функцию `recursive_two(A)`, которая возвращает два значения — наибольший элемент массива `A` и второй наибольший его элемент. Сравните ее быстродействие с другими вариантами поиска двух наибольших, которые мы видели в главе 1. Также сравните количество задействованных операций «меньше».
6. Числа Фибоначчи — это последовательность, которая вычисляется по рекуррентному соотношению $F_N = F_{N-1} + F_{N-2}$, где $F_0 = 0$, а $F_1 = 1$. *Числа Люка*

задаются примерно так же: $L_N = L_{N-1} + L_{N-2}$, где $L_0 = 2$, а $L_1 = 1$. Пользуясь стандартным рекурсивным подходом, напишите две функции — `fibonacci(n)` и `lucas(n)`, которые будут вычислять n-е число Фибоначчи и Люка соответственно. Измерьте время, которое требуется для вычисления F_N и L_N вплоть до $N = 40$. Если ваш компьютер достаточно быстрый, N стоит увеличить для повышения точности, а если слишком медленный — уменьшить, чтобы измерения заканчивались за приемлемое время. Затем напишите функцию `fib_with_lucas(n)` и вспомогательную к ней, `lucas_with_fib(n)`, пользуясь такими свойствами этих последовательностей:

- `fib_with_lucas(n)` — если поделить n нацело пополам и взять $i = n // 2$, а $j = n - i$, то $F_n = F_{i+j} = (F_i + L_j)(F_j + L_i) // 2$;
- `lucas_with_fib(n)` — в свою очередь, $L_N = F_{N-1} + F_{N+1}$.

Сравните быстродействие `fibonacci()` и `fib_with_lucas()`.

ГЛАВА 6

Двоичные деревья: бесконечность под рукой

В этой главе

- Как создавать, добавлять, удалять двоичные деревья и искать значения в них.
- Как поддерживать свойства *двоичного дерева поиска*:
  - чтобы в каждом узле значения из левого поддерева не превосходили значения самого узла;
  - чтобы в каждом узле значения из правого поддерева были не меньше значения самого узла.
- Как сложность добавления, удаления и поиска в сбалансированном дереве (O(log *N*)) может при неосторожной потере баланса доходить до неприемлемого порога O(*N*).
- Как поэлементно проходить двоичные деревья в восходящем порядке, затратив на это O(*N*) действий.
- Как из двоичного дерева смоделировать хеш-таблицу, которую можно проходить поэлементно в порядке возрастания ключей.
- Как из двоичного дерева смоделировать приоритетную очередь, которую можно проходить поэлементно в порядке удаления согласно приоритету, не меняя при этом структуру очереди.

Введение

Информация в связных списках хранится последовательно — в виде цепочки. В этой главе вы познакомитесь с новой структурой данных — *двоичным деревом*. Идея двоичного дерева чрезвычайно важна для программирования и теории

алгоритмов. В главе 5 вы изучили *рекурсию* — с точки зрения Python так называется ситуация, когда функция вызывает сама себя. Теперь же мы рассмотрим двоичное дерево как *рекурсивную структуру данных* — потому что двоичное дерево состоит из двоичных деревьев! Для начала убедимся в том, что уже известный нам связный список тоже может быть смоделирован как рекурсивная структура.

В самом деле, каждый узел связного списка — это `value`, полезная нагрузка, и `next` — ссылка на первый узел хвостовой части списка, которая сама по себе — тоже список. Список, в отличие от статического массива, можно оперативно увеличивать и сокращать, то есть менять N, количество элементов в нем. В примере 6.1 представлена рекурсивная функция `sum_list()`, которая вычисляет сумму всех элементов списка. Сравним ее работу с обычным суммированием в цикле, `sum_iterative()`.

Пример 6.1. Рекурсивный и итеративный способ посчитать сумму элементов связного списка

```
class Node:
  def __init__(self, val, rest=None):
    self.value = val
    self.next = rest

def sum_iterative(n):
  total = 0                          ❶
  while n:
    total += n.value                 ❷
    n = n.next                       ❸
  return total

def sum_list(n):
  if n is None:                      ❹
    return 0
  return n.value + sum_list(n.next)  ❺
```

❶ Сумма изначально нулевая.

❷ Добавим поле `n.value` каждого узла списка `n` к сумме.

❸ Перейдем к следующему узлу связного списка (или к маркеру конца `None`).

❹ Основание рекурсии: сумма элементов пустого списка нулевая.

❺ Рекурсивный вызов: сумма элементов пустого списка `n` — это значение `n.value` его первого узла плюс сумма элементов хвостового списка.

В цикле `while` мы сначала полагаем сумму `total` нулевой, а затем прибавляем к ней значения всех элементов списка. В рекурсивной функции наоборот: ну-

левое значение суммы пустого списка — это *основание рекурсии*, то есть то, на чем *рекурсивный вызов* остановится. Если список `n` состоит хотя бы из одного узла, происходит рекурсивный вызов, который вычисляет сумму элементов хвостового списка (который начинается с `n.next`), и к результату прибавляется `n.value` — это и есть общая сумма.

Список из N узлов рекурсивно раскладывается на первый узел и хвостовой список из $N - 1$ узлов. Это разложение по определению рекурсивно: хвостовой список — тоже список, причем меньшего размера. Но критерию эффективной рекурсии наша функция не соответствует, потому что сводит обработку объема данных размером N (сумму списка из N элементов) к обработке данных размером $N - 1$, ибо в хвостовом списке всего на один элемент меньше. Вообразим рекурсивную структуру данных, в которой разделение на подструктуры более эффективно. Например, возьмем арифметическое выражение, состоящее только из двухместных операций умножения, вычитания, сложения и деления. Атомарное арифметическое выражение — некоторое число, составное — это операция над двумя арифметическими выражениями. Например (для простоты заключим все операции в скобки):

- 3: число — это выражение;
- (3 + 2): сложение двух чисел — это выражение;
- $(((1 + 5) \times 9) - (2 \times 6))$: вычитание двух выражений, (2×6) справа из $((1 + 5) \times 9)$ слева, — выражение (при условии, что уменьшаемое и вычитаемое — это действительно выражения).

Таким образом выражения можно сочетать в любом объеме. На рис. 6.1 показано арифметическое выражение из семи операций над восемью числами. Такую нелинейную структуру уже нельзя представить в виде списка. Кто когда-нибудь разбирался с генеалогическим семейным древом, легко согласится с тем, что нашу структуру можно назвать *деревом разбора*.

Узел с умножением имеет два дочерних узла (с делением и разностью), потомками которых, в свою очередь, являются четыре внучатых узла (один из них — число 4), шесть правнучатых и два праправнучатых узла.

Арифметическое выражение с рис. 6.1 — это произведение двух других арифметических выражений, в чем, собственно, и выражается его рекурсивность. Чтобы его посчитать, надо сначала вычислить — рекурсивно! — левое выражение; получится 1. Затем рекурсивно же вычислить правое; получится 42. Теперь наконец можно и умножить, так что значение всего выражения — $1 \times 42 = 42$.

На рис. 6.1 исходное выражение представлено в виде рекурсивной структуры. На ее вершине — квадрат с операцией умножения, от которого идут стрелки

к левому и правому подвыражениям. Если на вершине подвыражения находится квадрат, то это снова некоторая операция, а если круг — это основание рекурсии, числовое значение, на котором рекурсивный разбор останавливается. В примере 6.2 смоделирована такая схема разделения на левое и правое подвыражение в классе `Expression`.

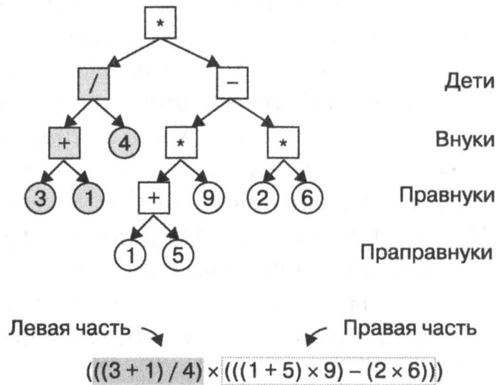


Рис. 6.1. Дерево разбора арифметического выражения

Пример 6.2. Класс Expression, моделирующий арифметическое выражение

```
class Value:                               ❶
  def __init__(self, e):
    self.value = e

  def __str__(self):
    return str(self.value)

  def eval(self):
    return self.value

class Expression:                          ❷
  def __init__(self, func, left, right):
    self.func = func
    self.left = left
    self.right = right

  def __str__(self):                       ❸
    return f"({self.left} {self.func.__doc__} {self.right}")

  def eval(self):                          ❹
    return self.func(self.left.eval(), self.right.eval())

def add(left, right):                      ❺
  """+"""
  return left + right
```

➊ Экземпляр класса `Value` может хранить любые значения, например числа. Класс нужен для того, чтобы и у узла-числа, и у узла-операции был метод `.eval()`. При преобразовании в строку возвращается строковое представление поля `.value`.

➋ Экземпляр класса `Expression` хранит выражение, то есть функцию `func` и ее операнды — левое подвыражение, `left`, и правое, `right`.

➌ Строковое представление выражения рекурсивно собирается из трех строк, заключенных в скобки.

➍ Чтобы вычислить значение выражения, надо вычислить значение левого и правого его потомков и обработать результат функцией `func`.

➎ Пример функции, реализующей сложение. Нам понадобится строковое представление соответствующей арифметической операции. В Python мы можем хранить его в так называемой *строке документации*, которую потом можно использовать, обратившись к полю `.__doc__` объекта-функции. Другие арифметические операции, например `mult()` для умножения, реализуются точно так же.

Вычисление выражения — рекурсивная операция, которая останавливается, когда дойдет до объекта типа `Value`. Разложим рекурсивное вычисление выражения $((1 + 5) \times 9)$ на стадии, как мы это делали в главе 5. Результат разложения приведен на рис. 6.2.

```
>>> a = Expression(add, Value(1), Value(5))
>>> m = Expression(mult, a, Value(9))
>>> print(m, '=', m.eval())
```

Чтобы вычислить `m`, необходимо вызвать два метода `.eval()`: для правого и для левого подвыражений соответственно. Каждый из этих вызовов может оказаться рекурсивным. Здесь левое подвыражение, `a` = (1 + 5), приводит к рекурсии, а правое, 9, — нет. Затем вычисляется и возвращается результат — 54. Становится понятно, чем хорошо рекурсивное представление в виде двоичного дерева `Expression`, да и сами рекурсивные алгоритмы выглядят коротко и ясно.

Очень важно, чтобы данные, которые мы помещаем в рекурсивную структуру, не ломали ее. Вот как легко испортить список:

```
>>> n = Node(3)
>>> n.next = n           # Внимание! Бездонная пропасть!
>>> print(sum_list(n))
```

Вот мы задали связный список из одного узла, `n`, но вместо добавления элемента добавили в поле `.next` сам этот узел! Функция `sum_list()` отлично работает на конечных списках, но здесь нарушена сама структура данных — и `sum_list(n)`

никогда не доберется до основания рекурсии. Та же история и с `Expression`. Нарушение структуры данных – это ошибка алгоритма, найти которую помогает аккуратное тестирование.

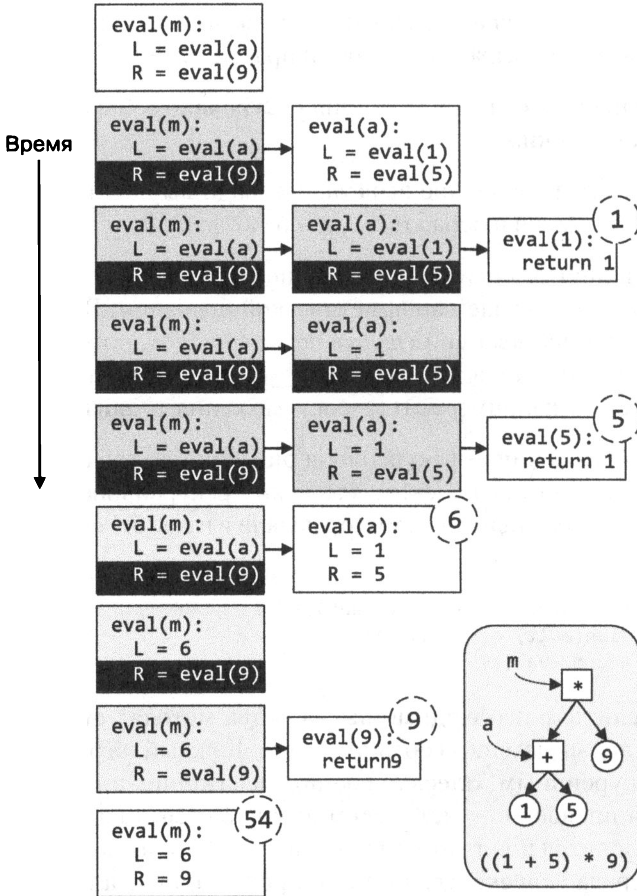


Рис. 6.2. Пошаговый проход рекурсивного вычисления выражения $((1 + 5) \times 9)$

Двоичные деревья поиска

Двоичное дерево – прародитель всех рекурсивных структур данных. Двоичное дерево поиска хранит произвольные значения (если их можно сравнивать), предоставляя прежде всего эффективную операцию поиска, а также вставки и удаления.

Если хранить данные в упорядоченном массиве, к ним применим *двоичный поиск*, вычислительная сложность которого O(log *N*). Есть еще немало выгод в том, чтобы хранить данные упорядоченно: например, человеку тоже будет проще искать нужное. Технически может возникнуть трудность с созданием очень большого массива, потому что он должен занимать один непрерывный фрагмент памяти, и неизвестно, может ли это обеспечить операционная система. Но главная неприятность — это произвольная вставка и удаление элемента, причем даже для такого динамического массива, как `list` в Python.

- Если операция снятия с конца `list` или добавления в конец не приводит к масштабированию, требуется только одно действие.
- Если вставка или удаление происходят не в конце массива, действий требуется больше. При удалении элемента нужно сдвинуть все элементы справа от него на одну позицию влево. При добавлении — сдвинуть вправо все элементы, начиная с нужной позиции, а затем записать туда добавляемый. В обоих случаях может потребоваться масштабирование.
- Если масштабирование все-таки понадобилось, нужно сначала выделить память под массив большего размера, скопировать туда старый массив (останется место для добавления новых значений), а затем освободить память, занимаемую старым массивом.

В программе на Python все эти действия проделываются с динамическим массивом типа `list` без участия программиста. Но вычислительная сложность *наихудшего случая* и, что еще более важно, средняя сложность вставки в начало или удаления из произвольного места неизбежно оказываются O(*N*). Второй и третий столбцы табл. 6.1 показывают время тысячи операций вставки одного значения в динамический массив и тысячи операций добавления одного значения в его конец.

Как и следовало ожидать, время добавления элемента в конец списка — константа, 0,004; это *наилучший случай* для вставки в значения структуры типа `list`. Время добавления тысячи элементов в начало списка практически удваивается с удвоением длины списка *N*. Стало быть, сложность этого действия линейна — O(*N*). Возможно, сразу этого видно не было, но теперь становится понятно, что поддерживать порядок элементов в массиве — дело довольно затратное. Столбец «Удаление» табл. 6.1 показывает, что удалить тысячу элементов из начала списка столь же затратно: вычислительная сложность этого действия тоже линейна. Время добавления удваивается с увеличением размера списка вдвое, в каждой следующей строке этого столбца записано число, примерно вдвое больше, чем число над ним.

Таблица 6.1. Сравнение операций вставки и добавления для массива типа list и двоичного дерева (время в миллисекундах)

N	Вставка в начало	Добавление в конец	Удаление	Дерево
1024	0.07	0.004	0.01	0.77
2048	0.11	0.004	0.02	0.85
4096	0.20	0.004	0.04	0.93
8192	0.38	0.004	0.09	1.00
16 384	0.72	0.004	0.19	1.08
32 768	1.42	0.004	0.43	1.15
65 536	2.80	0.004	1.06	1.23
131 072	5.55	0.004	2.11	1.30
262 144	11.06	0.004	4.22	1.39
524 288	22.16	0.004	8.40	1.46
1 048 576	45.45	0.004	18.81	1.57

В главе 2 мы уже выяснили: если вставка тысячи элементов имеет сложность $O(N)$, то и вставка одного тоже $O(N)$. Вставка десяти тысяч элементов — по тем же соображениям — также будет $O(N)$. Отличаются все три числа только мультипликативной постоянной.

Итак, опыт показал, что вставка или удаление произвольного элемента имеет линейную сложность. Это значит, что если в алгоритме имеется массив, то постоянное соблюдение порядка элементов в нем может сильно замедлить программу. А теперь посмотрим на столбец «Дерево» табл. 6.1, в котором приведены замеры тысячи операций вставки значения в *сбалансированное двоичное дерево*. Когда количество элементов в дереве увеличивается вдвое, время работы увеличивается — но не в разы, а на фиксированное значение, а это признак логарифмической сложности, $O(\log N)$. К тому же, помимо эффективной вставки в дерево, мы ожидаем от него не менее эффективных поиска и удаления.

Возьмем упорядоченный массив, который мы использовали для изучения **двоичного поиска**, и превратим его в двоичное дерево поиска, для чего возьмем центральный элемент массива и поместим его в *корень* дерева. Два поддерева справа и слева от корня формируются по тому же принципу из отрезков массива справа и слева от центрального элемента. По причинам историческим то, что, например, в стеке или куче называлось вершиной, в дереве называется корнем:

да, наши деревья растут сверху вниз! В сумме в нашем дереве получится семь элементов, как и в массиве. Поместим обе структуры данных рядом на рис. 6.3.

Корень
19
14
53
Корень правого поддерева
Левое поддерево
3 15 26 58
Двоичное дерево поиска

0	1	2	3	4	5	6
3	14	15	19	26	53	58

lo mid hi

Упорядоченный массив при двоичном поиске

Рис. 6.3. Дерево двоичного поиска из семи элементов

Каждый узел двоичного дерева поиска соответствует примеру 6.3: поле `left` ссылается на узел, и этот узел — корень собственного поддерева, поле `right` — аналогично. В дереве поиска вводится два дополнительных общих правила для всех узлов.

- Если у узла `n` есть левое поддерево, все значения поля `value` в нем не превосходят `n.value`.
- Если у узла `n` есть правое поддерево, все значения поля `value` в нем не меньше `n.value`.

Оба свойства соблюдены в дереве на рис. 6.3. Назовем *листом* узел, в котором оба поддерева пусты, — в нашем дереве четыре узла со значениями 3, 15, 26 и 58. Вот тут уже большинство согласится с тем, что деревья в программировании растут сверху вниз: у них не только корень вверху, но и листья внизу!

Пример 6.3. Структура двоичного дерева (в частности, дерева поиска)

```
class BinaryNode:
  def __init__(self, val):
    self.value = val          ❶
    self.left = None          ❷
    self.right = None         ❸
```

❶ Каждый узел хранит значение `value`.

❷ Поле `left` — это (возможно, пустое) поддерево, все хранимые значения которого ≤ `value`.

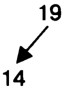
❸ Поле `right` — это (возможно, пустое) поддерево, все хранимые значения которого ≥ `value`.

Еще раз посмотрим на рис. 6.3. Левое поддерево корневого узла — это дерево, корень которого хранит значение 14, левый потомок — лист со значением 3, а правый потомок — лист со значением 15. Это те самые значения, что лежат в массиве в правой части иллюстрации, все они не больше 19, центрального элемента массива. При добавлении элемента по одному двоичные деревья прирастают снизу, как это показано в табл. 6.2.

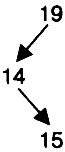
Таблица 6.2. Последовательное добавление значений 19, 14, 15, 53, 58, 3 и 26 в двоичное дерево поиска

19

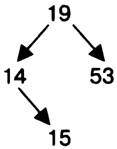
Вставим 19 — новое поддерево с корнем 19



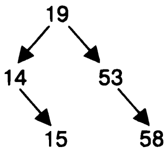
Вставим 14. Оно не больше 19 и должно попасть в левое поддерево. Но левое поддерево узла 19 пусто, так что создадим новое поддерево с корнем 14



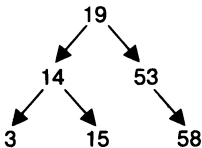
Вставим 15. Оно не больше 19, значит, должно попасть в левое поддерево с корнем 14. Оно больше этого 14 и попадает в его правое поддерево, которое пока пусто, так что создаем новое поддерево с корнем 15



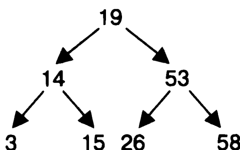
Вставим 53. Оно больше 19 и попадает в правое поддерево — пока пустое, и мы создаем там новое с корнем 53



Вставим 58. Оно больше 19, попадает в правое поддерево с корнем 53; оно больше и 53, попадает в его правое поддерево, которое пусто, и мы создаем новое поддерево с корнем 58



Вставим 3. Оно не больше 19 (попадает в левое поддерево с корнем 14), не больше 14 — должно попасть в левое поддерево узла 14, но там пусто, и мы создаем новое поддерево с корнем 3



Вставим 26. Оно больше 19 (попадает в правое поддерево с корнем 53), не больше 53 — должно попасть в левое поддерево узла 53, но там пусто, и мы создаем новое поддерево с корнем 26

Заведем для удобства класс `BinaryTree`, в котором пока будет только работа с корнем дерева, но далее в этой главе мы добавим туда другие полезные методы. В примере 6.4 задана также процедура добавления элемента в дерево.

Пример 6.4. Класс BinaryTree, в который мы будем добавлять методы работы с двоичными деревьями поиска

```
class BinaryTree:
  def __init__(self):
    self.root = None                              ❶

  def insert(self, val):
    self.root = self._insert(self.root, val)      ❷

  def _insert(self, node, val):
    if node is None:
      return BinaryNode(val)                      ❸

    if val <= node.value:                         ❹
      node.left = self._insert(node.left, val)
    else:                                         ❺
      node.right = self._insert(node.right, val)
    return node                                   ❻
```

❶ Корневой узел древа — `self.root` (если дерево пусто, равно `None`).

❷ Вспомогательный метод `._insert()` добавляет `val` в дерево с корнем `self.root`.

❸ Основание рекурсии: поддерево пусто, возвращаем новый узел `BinaryNode`.

❹ Если `val` не больше значения узла, его следует добавить в левое поддерево, `node.left`.

❺ В противном случае (`val` больше значения узла) `val` следует добавить в правое поддерево, `node.right`.

❻ По договоренности метод `._insert()` *всегда* возвращает — существующий или только что созданный — корень дерева, в которое было добавлено значение.

Метод `.insert(`*`значение`*`)` рекурсивно вызывает `._insert(`*`узел, значение`*`)` с параметром `self.root` — таким образом либо создается прежде пустой корень дерева, либо *`значение`* добавляется в одно из поддеревьев `self.root`[1].

Достоинство рекурсивных структур данных — процедуры работы с ними выглядят коротко и получаются как бы сами собой, например, `.insert()` — всего

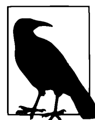
[1] В Python действует соглашение об именовании атрибутов класса: если имя метода начинается со знака подчеркивания (_), желательно пользоваться им только в других методах того же класса. — *Примеч. авт.*

лишь частный случай работы `._insert()`, который и добавляет элемент в поддерево, и возвращает (возможно, обновленный) его корень.

В нашем примере все добавляемые значения были разными, но это, конечно, не обязательно. Значения, хранящиеся в дереве, могут повторяться — именно поэтому _insert() сравнивает их операцией «меньше либо равно».

Основание рекурсии в `._insert(node, val)` — пустой (равный `None`) параметр `node`. Это бывает, когда `val` нужно вставить в пустое поддерево. В таком случае просто создается новый `BinaryNode` — это и будет корень возвращаемого поддерева. В рекурсивном вызове выбирается, в какое поддерево нужно добавить `val` — левое (`node.left`) или правое (`node.right`). Во всех случаях должна выполняться договоренность: `_insert()` добавляет `val` в дерево с корнем `node` и возвращает корень дерева, в которое добавлен `val`.

Вызов `._insert(node, val)` следит за сохранением основного свойства двоичного дерева поиска: все значения в левом поддереве не должны превосходить `node.value`, а все значения в правом должны быть больше.



Узел n двоичного дерева может иметь левый и правый дочерний узел. Для n.left и .right узел n называется родительским. Соответственно, потомки n — это все узлы правого и левого поддеревьев. Каждый узел, кроме корневого, имеет по меньшей мере одного или больше предков, чьим наследником он и является.

Попробуем добавить число 29 в двоичное дерево поиска с рис. 6.3. Число 29 больше корня, 19, значит, его нужно добавить в правое поддерево с корнем 53; 29 меньше 53, значит, добавлять нужно в левое поддерево с корнем 26; 29 больше 26, соответствующее правое поддерево пусто, так что число 26 становится корнем нового правого поддерева. Результат приведен на рис. 6.4.

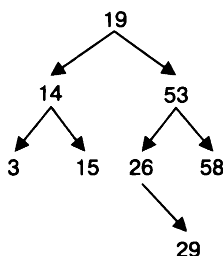


Рис. 6.4. Как добавить 29 в двоичное дерево поиска

Дерево может получиться очень разным в зависимости от того, в каком порядке мы добавляем туда элементы. Например, дерево в левой части рис. 6.5 очевидно началось с добавления числа 5 (которое стало корнем), и вообще всякий предок точно был добавлен раньше своих потомков.

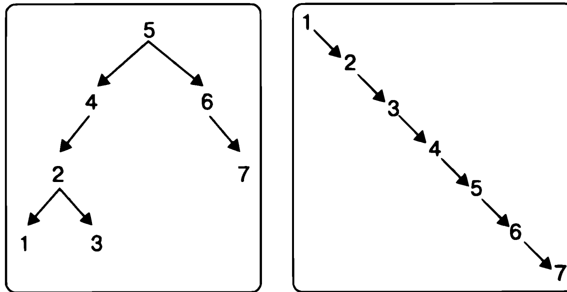


Рис. 6.5. Как различаются деревья двоичного поиска

На том же рисунке справа — также двоичное дерево поиска, которое получилось после добавления семи последовательно растущих значений. Образовался *наихудший случай* для двоичного дерева поиска: если повернуть картинку против часовой стрелки на 45°, получится связный список, с одним потомком у каждого узла. Это резко понижает производительность. В этой главе мы попробуем придумать, как, добавляя и удаляя элементы, не нарушать баланс наших древовидных структур.

Поиск значения в двоичном дереве

У нас уже есть метод поиска нужного места в дереве — этим занимается `._insert()`, когда рекурсивно проходит до соответствующего пустого поддерева и добавляет туда узел. Можно было бы поступить так же: если нужное значение встретилось, поиск успешен, а если добрались до пустого узла — нет. Однако, как показано в примере 6.5, здесь проще обойтись без рекурсии, обычным циклом `while`.

Пример 6.5. Как проверить, содержится ли значение в двоичном дереве

```
class BinaryTree:
  def __contains__(self, target):
    node = self.root                ❶
    while node:
      if target == node.value:      ❷
        return True
```

```
    if target < node.value:        ❸
      node = node.left
    else:
      node = node.right            ❹

  return False                     ❺
```

❶ Начнем поиск с корня.

❷ Если значение текущего узла совпадает с `target`, поиск успешен; вернем `True`.

❸ Если `target` меньше значения текущего узла, продолжим поиск в левом поддереве.

❹ В противном случае (если `target` был больше) продолжим поиск в правом дереве.

❺ Если мы встретили пустой узел, то больше искать негде — значения в дереве нет; вернем `False`.

Метод `.__contains__()` мы дописали в класс `BinaryTree` вдобавок к уже реализованным процедурам[1]. Он работает подобно процедуре поиска значения в связном списке, но различие в том, что всякий раз выбирается только одно следующее поддерево для поиска — левое или правое.

Удаление значения из двоичного дерева

Удалить значение из связного списка весьма просто, но в случае дерева все несколько сложнее. Допустим, мы хотим удалить значение, а оно оказалось в корне дерева. Если удалить корень, как потом «срастить» два осиротевших поддерева? А если не в корне, все равно должен быть какой-то простой и понятный общий способ? Попробуем догадаться, что это за способ, на примере, в котором удаляется значение именно из корня. Рассмотрим приведенные на рис. 6.6 исходное двоичное дерево поиска и два варианта двоичных деревьев поиска, в которых уже нет удаленного узла.

[1] Имя выбрано не случайно. Если в классе реализовать метод __contains__(self, target), для всех экземпляров этого класса сама собой начнет работать операция in. Например, value in tree, где tree — экземпляр класса BinaryTree, будет приводить к вызову этого метода и возвращать True или False. — *Примеч. авт.*

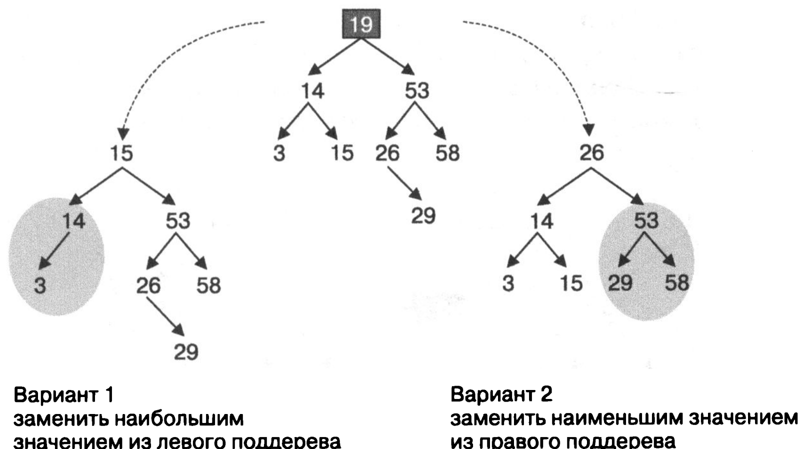


Рис. 6.6. Два возможных варианта удаления 19 из двоичного дерева поиска

Еще раз замечу, что в обоих вариантах получаются именно деревья двоичного поиска: значения в левом поддереве не превосходят значения в корне, а значения в правом поддереве не меньше корня. И того и другого варианта можно достичь сравнительно несложно.

- Вариант 1: найти и удалить наибольшее значение в левом поддереве и использовать его в качестве корня.
- Вариант 2: найти и удалить наименьшее значение в правом поддереве и использовать его в качестве корня.

Разницы между вариантами никакой, давайте выберем второй. Убедимся, что полученное дерево — двоичное дерево поиска. Его новый корень, 26, — это наименьшее значение из правого поддерева, и значит, все элементы перестроенного правого поддерева на рис. 6.6 не меньше 26. При этом левое поддерево не изменилось, и все его элементы были не больше 19 — старого корня, — а 26 не меньше этого корня и, стало быть, всех остальных элементов левого поддерева.

Для начала разберемся с тем, как удалить наименьшее значение в поддереве. Если подумать, становится ясно, что у минимального элемента поддерева *нет левого потомка*, иначе этот левый потомок сам был бы минимальным элементом[1]. Например, на рис. 6.7 наименьшее значение правого поддерева

[1] Строго говоря, если хранить в дереве в том числе одинаковые значения, то в нем могут встретиться узлы с наименьшим значением и левым потомком. Однако значение в этом левом потомке также будет наименьшим (потому что он левый), и, следовательно, в дереве непременно будет узел с наименьшим значением и без левого потомка. — *Примеч. пер.*

с корнем 53 — это 26, и у него нет левого потомка. Если этот узел удалить, останется только поднять его правое поддерево (с корнем 29) на место левого поддерева узла 53. Наименьший узел всегда можно при удалении заменять его правым поддеревом, потому что *левого поддерева у него нет*, и других узлов мы при этом не потеряем.

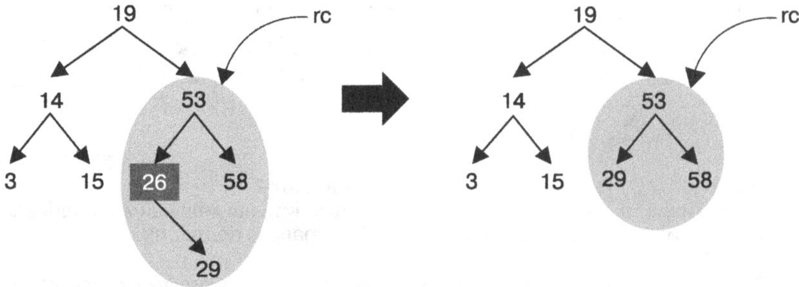


Рис. 6.7. Удаление наименьшего значения в дереве

В примере 6.6 мы завели в `BinaryTree` вспомогательный метод `._remove_min(узел)`, который удаляет наименьший элемент дерева с корнем *узел*; метод можно вызывать только на непустых узлах. Если передать этому методу правого потомка дерева с рис. 6.7, произойдет рекурсивный вызов, а именно попытка удалить наименьшее значение из левого поддерева с корнем 26. Это уже основание рекурсии, потому что у 26 нет левого потомка, значит, настала пора поднять правое поддерево (с корнем 29) на место удаленного узла и вернуть это правое поддерево в качестве нового левого поддерева 53.

Пример 6.6. Как удалить наименьшее значение

```
def _remove_min(self, node):
  if node.left is None:                          ❶
    return node.right

  node.left = self._remove_min(node.left)        ❷
  return node                                    ❸
```

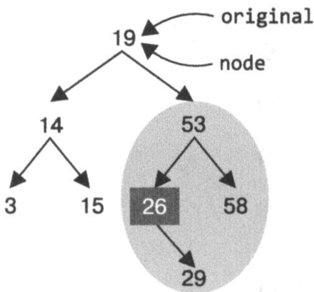
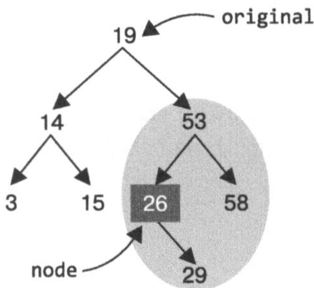
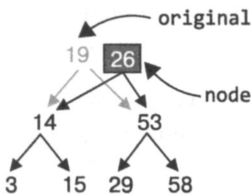
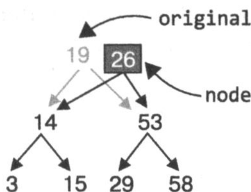
❶ Основание рекурсии: у узла нет левого потомка, значит, *он содержит наименьшее значение* в поддереве. Поднимаем на его место правое поддерево (которое может быть и пустым) и возвращаем его.

❷ Рекурсивный вызов: удаляем наименьшее значение из левого поддерева, а то, что нам возвратил метод `._remove_min()`, записываем в левое поддерево узла.

❸ При рекурсивном вызове `._remove_min()` сам узел не меняется (возможно, меняется его левое поддерево), возвращаем его.

Снова у нас вышло коротко и ясно! Метод `._remove_min()`, как и предыдущие рекурсивные методы, возвращает корень обновленного поддерева. Теперь можно дополнить класс `BinaryTree` методом `.remove()`, который удаляет значение из двоичного дерева поиска. На примере диаграмм в табл. 6.3 рассмотрим, как именно должно происходить удаление значения 19, которое находится в корне двоичного дерева поиска. На диаграммах упоминаются две переменные: `original`, указывающая на удаляемый узел, и `node`, в которой окажется узел на замену удаляемому.

Таблица 6.3. Как удалить корень двоичного дерева поиска

<table>
<tr><td>
</td><td>Поскольку удаляемый узел находится в корне дерева, <code>original</code> и <code>node</code> изначально указывают на него</td></tr>
<tr><td>
</td><td>По окончании цикла <code>while</code> переменная <code>node</code> указывает на наименьший узел в правом поддереве <code>original</code>, в данном случае — на узел со значением 26. Этот узел будет новым корнем всего дерева. Заметим, что, во-первых, у него нет левого потомка, во-вторых, это наименьшее из значений поддерева с корнем 53, и, в-третьих, 26 не меньше значений дерева с корнем 14</td></tr>
<tr><td>
</td><td>После того как наименьший узел удален из поддерева <code>original.right</code> (с корнем 53), записываем это обновленное поддерево (с узлами 29, 53 и 58) в <code>node.right</code>. Ненадолго <code>original.right</code> и <code>node.right</code> становятся одинаковыми и указывают на поддерево с корнем 53</td></tr>
<tr><td>
</td><td>Осталось только записать <code>original.left</code> в <code>node.left</code>, и, когда <code>_return()</code> вернет <code>node</code>, он займет место <code>original</code> — и как корень обновленного дерева поиска, и как потомок какого-нибудь узла (если у <code>original</code> был предок)</td></tr>
</table>

Теперь посмотрим на реализацию `.remove()` в примере 6.7.

Пример 6.7. Удаление значения из BinaryTree

```
def remove(self, val):
  self.root = self._remove(self.root, val)                ❶

def _remove(self, node, val):
  if node is None: return None                            ❷

  if val < node.value:
    node.left = self._remove(node.left, val)              ❸
  elif val > node.value:
    node.right = self._remove(node.right, val)            ❹
  else:                                                   ❺
    if node.left is None: return node.right
    if node.right is None: return node.left               ❻

    original, node = node,                                ❼
    node.right
    while node.left:                                      ❽
      node = node.left

    node.right = self._remove_min(original.right)         ❾
    node.left = original.left                             ❿

  return node
```

❶ Используем вспомогательный метод `._remove()`: ему можно указать корень конкретного поддерева, из которого надо удалять элемент. В нашем случае это `self.root`.

❷ Основание рекурсии: если дерево пусто, вернем `None`.

❸ Первый вариант рекурсивного вызова: удаляемое значение меньше `node.value`, значит, удалять надо из левого поддерева. Запишем в `node.left` обновленное поддерево — результат удаления оттуда `val`.

❹ Второй вариант рекурсивного вызова: удаляемое значение больше `node.value`, значит, удалять надо из правого поддерева, `node.right`. Обновим `node.right`, записав туда результат удаления `val` из него.

❺ Третий вариант рекурсивного вызова: узел хранит удаляемое значение. Возможно, надо удалять именно этот узел, а может, стоит поискать кандидата получше.

❻ Сначала немного оптимизации. Если одно из поддеревьев пусто, можно смело удалять именно этот узел, а возвращать оставшееся поддерево. Если и оно было пусто, значит, узел — это лист, и вернется `None`.

❼ Поищем узел без левого потомка. Запомним исходный узел в `original`: нам еще работать с его поддеревьями, `.left` и `.right`. В этом месте оба поддерева не пусты.

❽ Согласно выбранному правилу, начнем поиск наименьшего элемента в правом поддереве. Теперь `node` равен `node.right`, и, пока у `node` есть левое поддерево, мы не можем быть уверены, что его значение наименьшее. Так что мы в цикле будем выбирать самый левый узел без левого потомка — наименьшее значение правого поддерева `original` будет в нем.

❾ Мы нашли узел `node`, который должен стать новым корнем с поддеревьями `original.left` и `original.right`. Его левое поддерево не изменится, а вот из правого надо удалить наименьшее значение. Воспользуемся для этого уже написанным методом `._remove_min()` и положим результат в `original.right`. Можно заметить, что рекурсивный вызов `._remove_min()` выполнит заново все то, что мы только что уже делали, но так все-таки понятнее, чем пытаться объединить логику работы обоих методов в одном цикле `while`.

❿ Теперь оба поддерева `original` перенесены в `node`.

Что еще нам нужно от двоичного дерева поиска? Хотелось бы получить упорядоченную последовательность всех его значений! В программировании это называется *обходом* дерева.

Обход двоичного дерева

Связный список пройти поэлементно просто: начать с первого узла и в цикле `while` переходить к элементу, на который указывает поле `next`, до тех пор, пока не будут пройдены все узлы. Но такой линейный подход не годится для деревьев: есть сразу два пути дальнейшего просмотра, `left` и `right`. По-видимому, раз уж деревья рекурсивны, то и обход их тоже должен быть рекурсивным. В примере 6.8 мы зададим рекурсивный метод-генератор и с его помощью аккуратно обойдем все узлы дерева.

Пример 6.8. Метод-генератор, который обходит узлы двоичного дерева при восходящем порядке

```
class BinaryTree:

  def __iter__(self):
    yield from self._inorder(self.root):           ❶

  def _inorder(self, node):
    if node is None:                               ❷
      return

    for v in self._inorder(node.left):             ❸
      yield v

    yield node.value                               ❹

    for v in self._inorder(node.right):            ❺
      yield v
```

❶ Отдадим управление вспомогательному генератор-методу `._inorder(self, node)`, который обходит в порядке возрастания дерево с корнем `node`. В качестве параметра передадим ему корень всего дерева, `self.root`.

❷ Основание рекурсии – пустое поддерево.

❸ Сначала пройдем по левому поддереву, `node.left`.

❹ Следом вернем значение текущего узла.

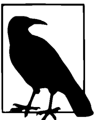
❺ Наконец, пройдем по правому поддереву, `node.right`. Таким образом, все значения окажутся перечислены в порядке возрастания.

Вспомогательный метод `._inorder(self, node)` проходит все узлы дерева с корнем `node` и возвращает их по одному. Основание рекурсии наступает, когда соответствующее поддерево пусто, – `._inorder()` ничего не делает. Все узлы левого поддерева, `node.left`, в двоичном дереве поиска по построению не больше корня `node.value`, а узлы правого поддерева, `node.right`, не меньше его. Значит, если рекурсивно обойти левое поддерево, затем вернуть значение в узле, а затем обойти правое поддерево, значения будут возвращаться в порядке возрастания. Рисунок 6.8 представляет этот процесс в виде диаграмм для двоичного дерева T с пятью значениями.

Метод `.__iter__()` – принятым в Python способом – отдает перечисление элементов методу `._inorder()` с соответствующим параметром – корнем всего дерева[1].

[1] На всякий случай напомню, что имя __iter__() позволяет использовать сам объект типа BinaryTree как итератор – например, проходить его циклом for. – *Примеч. пер.*

Рис. 6.8. Проход по всем узлам двоичного дерева поиска в порядке возрастания их значений

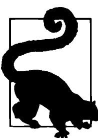


Обойти двоичное дерево можно еще двумя способами. При префиксном обходе сначала возвращается значение узла, а потом выполняется обход дочерних поддеревьев. Это удобно для копирования структуры дерева. При постфиксном обходе, наоборот, сначала идет обход поддеревьев, а затем возвращается значение узла — таким образом можно, например, вычислить выражение на рис. 6.1.

Итак, теперь мы умеем искать значения в дереве двоичного поиска, добавлять их туда и удалять оттуда, а еще мы научились получать упорядоченную последовательность значений дерева. Настала пора посмотреть, с какой скоростью все это работает.

Исследование быстродействия двоичных деревьев поиска

Для того чтобы замерить быстродействие поиска, добавления и удаления, нужно знать *высоту* дерева. Высота некоторого узла — это количество ссылок на правое или левое поддерево, которые нужно пройти, чтобы достигнуть наиболее отдаленного листового узла. Высота листового узла, соответственно, нулевая, а высота всего дерева поиска — это высота его корневого узла.



Высота листового узла равна 0, следовательно, высота пустого узла — узла со значением None — не может быть тоже нулем, она должна быть еще меньше. В наших вычислениях удобно использовать –1 в качестве высоты пустого узла.

Количество узлов, которые в *худшем случае* придется просмотреть при поиске по двоичному дереву, равно высоте этого дерева, то есть высоте корневого узла. Допустим, в дереве N узлов — какова его высота? Мы помним, что высота определяется порядком добавления элементов в дерево. *Наилучший случай*, то есть максимум узлов при минимуме высоты, — это *полное двоичное дерево*, в котором заполнены все уровни. В дереве при этом $2^k - 1$ узлов, а высота его равна $k - 1$. К примеру, на рис. 6.9 представлено полное дерево высотой 5, в котором 63 узла. Поиск по такому дереву займет не более шести сравнений: если начать с корневого узла и переходить по ссылкам `left` и `right`, мы посетим не более шести узлов. Поскольку $63 = 2^6 - 1$, время поиска по такому дереву пропорционально $\log(N + 1)$. А вот в *худшем случае*, когда значения добавлялись в возрастающем или убывающем порядке, двоичное дерево поиска превращается в последовательную цепочку и высота его будет $N - 1$ (как на рис. 6.5). В общем, если высота двоичного дерева поиска — h, то производительность оценивается на $O(h)$.

Добавить новый узел занимает столько же времени, что и найти несуществующий, — надо пройти до какого-то листового узла, разница только в том, что у этого узла появится левое или правое поддерево, куда придется добавить еще один листовой узел. Но сложность при этом остается $O(h)$.

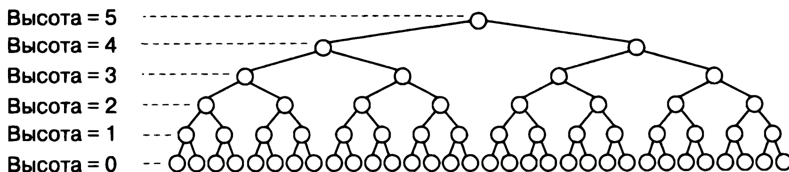


Рис. 6.9. Полное дерево содержит максимум узлов при минимальной возможной высоте

Чтобы удалить узел из двоичного дерева поиска, надо выполнить три операции.

1. Найти узел с нужным значением.
2. Найти в правом поддереве этого узла узел с наименьшим значением.
3. Удалить наименьший узел из правого поддерева (и заменить значение текущего узла).

В *наихудшем случае* количество действий в каждой из этих операций пропорционально высоте[1], так что в самом худшем варианте время, потраченное на удаление, будет пропорционально $3h$ (где h – высота всего дерева). В главе 2 мы уже выяснили, что мультипликативная постоянная не влияет на общую оценку, так что время удаления элемента также оказывается порядка $O(h)$.

Структура двоичного дерева поиска определяется порядком добавления элементов в него и удаления их оттуда, и нет способа этот порядок заранее определить. Значит, нужна возможность оценить, насколько эффективна сложившаяся структура дерева. В главе 3 мы посмотрели, как масштабируется хеш-таблица при достижении пороговой загруженности: размер увеличивается вдвое, а элементы хешируются заново. Геометрическое масштабирование позволяет сделать эту дорогостоящую ($O(N)$) операцию настолько редкой, что порядок производительности остается неизменным: как мы помним, константным, $O(1)$ для действия `get()`.

О геометрическом масштабировании можно было бы задуматься, если бы мы хранили узлы дерева в массиве (как кучу), но при таком подходе совершенно непонятно, когда именно хранилище нуждается в расширении и как часто оно будет требоваться. Пороговое значение должно зависеть от N, а, например, если хранить дерево как кучу, поуровнево, порог определяется не количеством элементов, а высотой. Посмотрим на рис. 6.10: всего пара неудачных вставок – и нам требуется в четыре раза больше места под еще два уровня! Для наглядности узлы одинаковой высоты обозначены на рисунке одинаковым фоном. Замечу,

[1] В тренировочных заданиях будет упражнение на эту тему. – *Примеч. авт.*

что само дерево при этом получается *несбалансированным*: кратчайший путь от корня до листового узла оказывается вдвое меньше высоты.

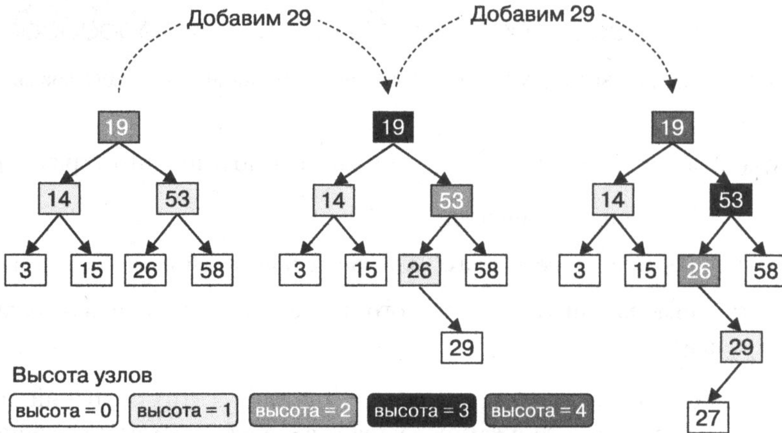


Рис. 6.10. Несбалансированное дерево после добавления двух элементов

Полное двоичное дерево на рис. 6.10 целиком сбалансировано: его высота (и высота корня) — 2, высота всех узлов первого уровня — 1, все узлы второго уровня листовые, и их высота — 0. Как только мы добавляем 29 (средняя диаграмма на рис. 6.10), в соответствующем месте появляется новый листовой узел. При этом *высота всех предков узла 29 увеличивается на единицу* (обозначено на диаграмме фоном). После добавления 27 дерево окончательно разбалансируется (правая диаграмма): левое поддерево с корнем 14 имеет высоту 1, а правое — с корнем 53 — уже 3. Такая же картина и для других узлов, например для 26 и 53. В следующем разделе мы посмотрим, как определить, что дерево нуждается в балансировке, и как эту балансировку делать.

Сбалансированные двоичные деревья

Самая ранняя из известных сбалансированных древовидных структур данных, АВЛ-дерево, была изобретена в 1962 году[1]. Идея была в том, что при всяком добавлении или удалении значения отслеживается эффективность получившегося дерева и, если она недостаточна, дерево реорганизуется. В АВЛ-дереве

[1] АВЛ-дерево названо по фамилиям его изобретателей: Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса. — *Примеч. авт.*

гарантируется, что в каждой вершине высота правого и левого поддеревьев отличаются не более чем на 1 (то есть разница между высотой правого поддерева и высотой левого может быть только −1, 0 или 1).

Как показано в примере 6.9, высоту поддерева удобно знать заранее и хранить в классе `BinaryNode` в поле `height`. Всякий раз при добавлении элемента в дерево значения высоты узлов, изменивших место в иерархии, вычисляются заново, так что *несбалансированный узел будет виден сразу*.

Пример 6.9. Структура узла в АВЛ-дереве

```
class BinaryNode:
  def __init__(self, val):
    self.value = val                                          ❶
    self.left = None
    self.right = None
    self.height = 0                                           ❷

  def height_difference(self):                                ❸
    left_height = self.left.height if self.left else -1       ❹
    right_height = self.right.height if self.right else -1
    return left_height - right_height                         ❺

  def compute_height(self):                                   ❻
    left_height = self.left.height if self.left else -1
    right_height = self.right.height if self.right else -1
    self.height = 1 + max(left_height, right_height)
```

❶ `BinaryNode` в целом такой же, как и в простом двоичном дереве поиска.

❷ Здесь хранится высота узла.

❸ Вспомогательный метод, который вычисляет разность между высотами правого и левого поддеревьев.

❹ Если поддерево пусто, его высота по договоренности −1; в противном случае используем настоящую высоту.

❺ Вернем разницу высот. С учетом высоты пустого поддерева это просто разность `left_height` и `right_height`.

❻ Вспомогательный метод, который вычисляет высоту узла *в предположении, что поле* `height` *левого и правого поддеревьев соответствует действительности* (включая высоту −1 для пустых поддеревьев).

Теперь допишем метод `_insert()`: он должен вычислять значение узла при добавлении (пример 6.10).

Пример 6.10. Дополним метод _insert() вычислением высоты

```
def _insert(self, node, val):
  if node is None:
    return BinaryNode(val)                    ❶

  if val <= node.value:
    node.left = self._insert(node.left, val)
  else:
    node.right = self._insert(node.right, val)

  node.compute_height()                       ❷
  return node
```

❶ Основание рекурсии — создание нового листового узла — не требует вычисления длины, которая и так по умолчанию 0.

❷ Когда отработает рекурсивный вызов и `val` будет добавлено в правое или левое поддерево, высота узла вполне может измениться, так что ее надо высчитать заново.

Посмотрим на рис. 6.11. Вот мы вызвали `insert(27)` и после нескольких рекурсивных вызовов создали новый листовой узел для 27, добавив его в двоичное дерево поиска. Концевой вызов `_insert()` достиг основания рекурсии и вернул только что созданный листовой узел 27. На рисунке видно, что оба узла — новый листовой (27) и старый листовой (29, будущий родитель) — ненадолго оба имеют высоту 0. Осталось только добавить единственный оператор — вычисление новой высоты узла — в конец `_insert()`, перед возвратом из рекурсивного вызова. Пока рекурсивные вызовы завершаются, высота каждого родителя пересчитывается (отмечено на рисунке фоном) — и заметим, что только у этих узлов двоичного дерева могла от добавления элемента измениться высота. Метод `compute_height()` высчитывает высоту узла из простого логического соображения: высота узла на единицу больше, чем высота самого высокого из поддеревьев этого узла.

Итак, с каждым завершением рекурсивного вызова пересчитывается высота соответствующего предка узла 27. Поскольку к этому моменту у всех потомков высота своевременно посчитана, `_insert()` может определить, что узел стал несбалансированным — это когда *высоты правого и левого поддеревьев узла отличаются более чем на единицу*.

Разность высот узла определяется как разность между высотой правого поддерева данного узла и высотой его левого поддерева. Высота пустого поддерева равна –1. В АВЛ-деревьях разность высот любого узла должна быть –1, 0 или 1.

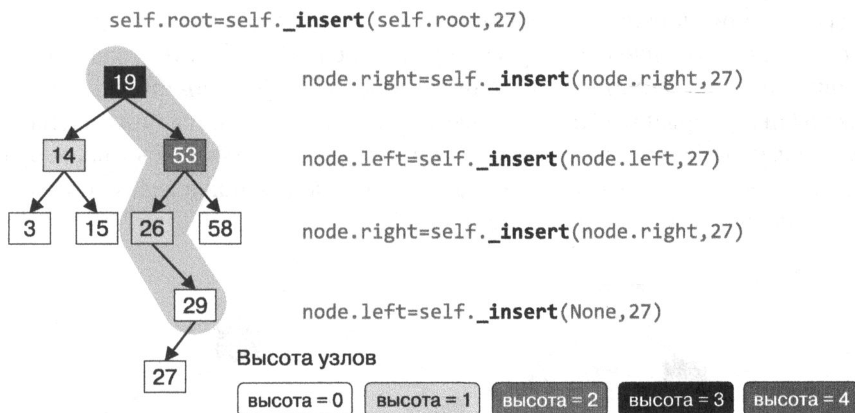


Рис. 6.11. Рекурсивные вызовы при добавлении элемента в дерево

Будем говорить, что узел 26 перевешивает вправо, потому что разница его весов равна −1 − (−1) = −2. Соответственно, узел 53 перевешивает влево, потому что разница его весов — 2 − 0 = 2, а корень — вправо с разницей весов 1 − 3 = −2. После того как мы отметили все потерявшие равновесие узлы, хотелось бы применить какой-нибудь алгоритм для восстановления баланса. Это можно сделать, например, в процессе возврата из рекурсивных вызовов, когда `_insert()` уже определил, что добавление узла разбалансировало дерево. Возврат из рекурсивных вызовов начинается с конца, с основания рекурсии, то есть первым найденным несбалансированным узлом будет 26.

Здесь изобретатели АВЛ-дерева вводят понятие «поворот узла», которое проще всего представить наглядно, как на рис. 6.12. Три узла со значениями 10, 30 и 50 раскрашены в соответствии с их высотами. Корневой узел, 50, имеет вес h. Серые треугольники — это поддеревья, причем они являются двоичными деревьями поиска, и притом сбалансированными. Левое поддерево узла 10 обозначено `10L`, в нем содержатся узлы, значение которых не превосходит 10. Нам достаточно знать, что по построению высота такого дерева равна $h - 3$. Поскольку все поддеревья на диаграмме (10L, 10R, 30R и 50R) раскрашены одинаково, можно сделать вывод о том, что их высоты тоже равны $h - 3$.

Тогда все дерево перевешивает влево: высота левого поддерева — $h - 1$, а правого — $h - 3$, значит, разница весов составляет +2. После того как в АВЛ-дереве определился дисбаланс, происходит вращение узлов и дерево меняет конфигурацию, как это показано на рис. 6.12. После вращения высота всего дерева оказывается $h - 1$, а узел 30 становится новым корнем. На рисунке показано *правое вращение* дерева. Если представить себе дерево в виде сцепленных

подвесок с одним кольцом сверху и двумя — снизу, то вращение — это когда вы перестаете держать всю цепочку за старый корень (50), беретесь за новый (30) и встряхиваете всю конструкцию. При этом новый корень поднимается на верхний уровень, а старый корень оказывается справа и опускается ниже. Ненадолго получится, что у 30 *три* потомка, зато у 50 теперь точно осталось только правое поддерево (50R), а 30R отлично может стать его левым поддеревом, потому что его узлы не превосходят 50 и не меньше 30.

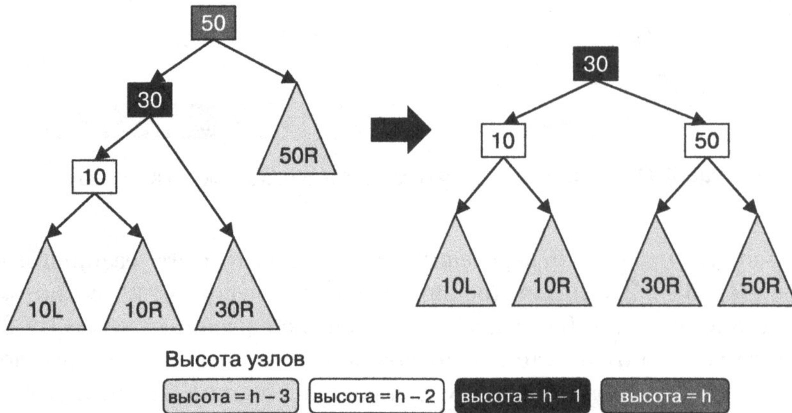


Рис. 6.12. Как сбалансировать двоичное дерево поиска, вращая его корень вправо

Если взять дерево всего из трех элементов, получится четыре варианта дисбаланса — как на рис. 6.13. В частности, лево-левый дисбаланс — это упрощенная версия дерева с рис. 6.12. Чтобы восстановить равновесие, такое дерево достаточно *повернуть вправо* — так же как и зеркальное к нему дерево с право-правым дисбалансом нужно с той же целью *повернуть влево*. Варианты дисбаланса поименованы сообразно тому, как расположены в нем узлы, начиная с корневого. Во всех четырех случаях мы получаем сбалансированное дерево с корнем 30, левым потомком 10 и правым — 50.

Лево-правый дисбаланс чуть сложнее: такое дерево приводится к норме в два шага. Сначала надо *повернуть влево* левое поддерево с корнем 10, при этом узел 10 опустится на уровень, а узел 30 — поднимется. Все дерево при этом приобретает уже знакомый нам лево-левый дисбаланс, и теперь его достаточно *повернуть вправо* — равновесие восстановится. Назовем такую двухступенчатую операцию лево-правым поворотом. Дерево с право-левым дисбалансом зеркально воспроизводит эту ситуацию, и для восстановления равновесия его нужно повернуть вправо-влево. В репозитории примеров к книге есть оптимизированные версии всех этих операций.

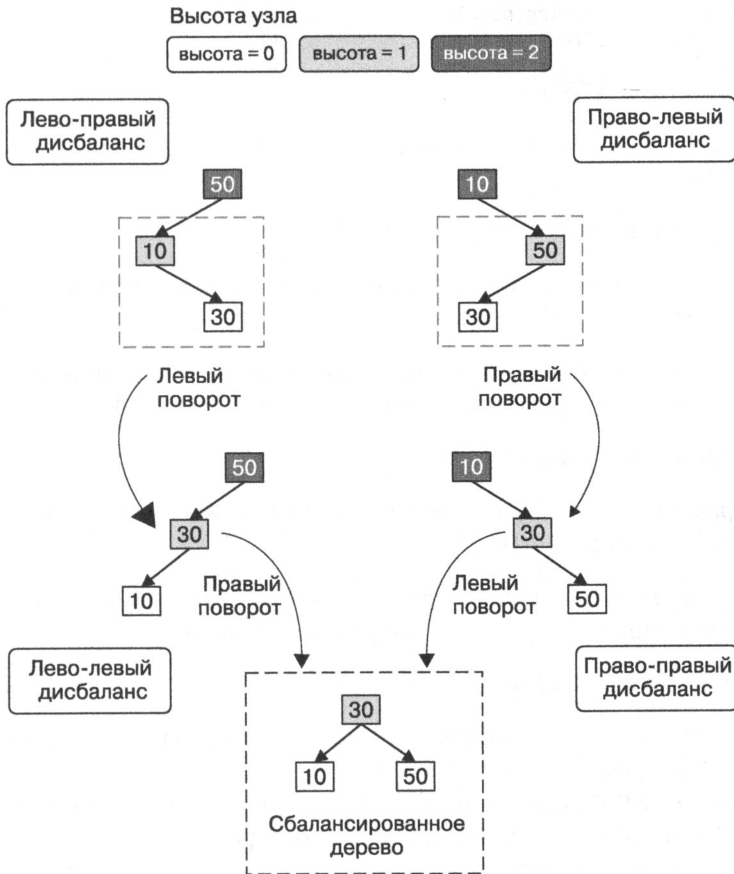


Рис. 6.13. Четыре типа вращения узла

В примере 6.11 приведены функции, которые используют готовые операции поворота и восстанавливают равновесие, когда дерево перевешивает влево или вправо.

Пример 6.11. Вспомогательные функции, которые выбирают подходящие операции поворота

```
def resolve_left_leaning(node):
  if node.height_difference() == 2:            ❶
    if node.left.height_difference() >= 0:     ❷
      node = rotate_right(node)
    else:
      node = rotate_left_right(node)           ❸
  return node                                  ❼
```

```
def resolve_right_leaning(node):
  if node.height_difference() == -2:           ❹
    if node.right.height_difference() <= 0:    ❺
      node = rotate_left(node)
    else:
      node = rotate_right_left(node)           ❻
  return node                                  ❼
```

❶ Узел перевешивает влево с разницей +2.

❷ Если левое поддерево не перевешивает вправо, можно просто повернуть узел влево (вызвать `rotate_right()`).

❸ В противном случае левое поддерево перевешивает вправо и нужно поворачивать узел влево-вправо (вызывать `rotate_left_right()`).

❹ Узел перевешивает вправо с разницей –2.

❺ Если правое поддерево не перевешивает влево, можно просто повернуть узел вправо (вызвать `rotate_left()`).

❻ В противном случае правое поддерево перевешивает влево и нужно поворачивать узел вправо-влево (вызывать `rotate_right_left()`).

❼ Возвращаем узел (если он был разбалансирован, то мы вернули ему равновесие).

Идея в том, чтобы сразу ликвидировать дисбаланс, как только в процессе работы с деревом он проявился. В последний раз дополним метод `._insert()` — как показано в примере 6.12, где для этого сразу вызываются наши вспомогательные функции. Если добавлять значение в левое поддерево, сам узел не может перевесить вправо, так что достаточно исправить перевес влево, если он появился. Аналогично, если добавлять элемент в правое поддерево, сам узел не перевесит влево.

Пример 6.12. Если узел не сбалансирован, сразу повернем его

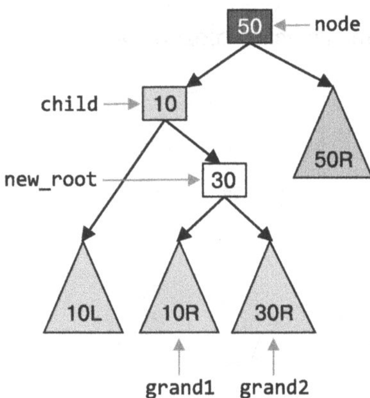
```
def _insert(self, node, val):
  if node is None:
    return BinaryNode(val)

  if val <= node.value:
    node.left = self._insert(node.left, val)
    node = resolve_left_leaning(node)            ❶
  else:
    node.right = self._insert(node.right, val)
    node = resolve_right_leaning(node)           ❷

  node.compute_height()
  return node
```

❶ Левое поддерево может перевешивать — это надо исправить!

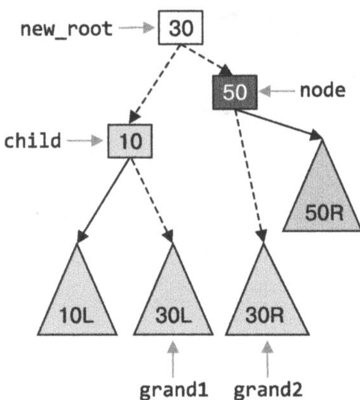
❷ Правое поддерево может перевешивать — это тоже надо исправить!

Текст функций поворота можно найти в репозитории к книге. В табл. 6.4 показано, как по мере выполнения функции `rotate_left_right()` меняется балансируемое дерево. Начинается все с определения `new_root` и остальных затронутых дисбалансом узлов, а заканчивается сбалансированным деревом. Нелишне заметить, что высоты `child` и `node` в нем пересчитаны заново.

Таблица 6.4. Реализация лево-правого вращения




```
def rotate_left_right(node):
    child = node.left
    new_root = child.right
    grand1 = new_root.left
    grand2 = new_root.right
```




```
    child.right = grand1
    node.left = grand2
    new_root.left = child
    new_root.right = node
    child.compute_height()
    node.compute_height()
    return new_root
```

Обратим внимание, что `rotate_left_right()` возвращает корень нового сбалансированного дерева: если мы восстанавливаем равновесие некоторого узла в составе большего дерева, нам понадобится поменять старый корень поддерева

на новый. При этом пересчитывать высоту `new_root` не надо — это уже сделано в `._insert()` или `._remove()`. По последней диаграмме видно, что полученное двоичное дерево — это дерево поиска: например, все значения в 30L не меньше 10 и не больше 30, так что это поддерево вполне может быть правым для узла 10. Точно так же 30R может быть левым поддеревом узла 50.

Дополненный метод `._insert()` теперь при необходимости восстанавливает баланс двоичного дерева поиска. Доделать `._insert()` и `._remove_min()` с помощью наших вспомогательных функций тоже довольно просто — это сделано в примере 6.13. В тексте метода отслеживаются все четыре случая, когда меняется структура дерева и требуется вмешательство.

Пример 6.13. Улучшенный метод ._remove() восстанавливает свойства АВЛ-дерева

```
def _remove_min(self, node):
  if node.left is None: return node.right

    node.left = self._remove_min(node.left)
    node = resolve_right_leaning(node)                    ❶
    node.compute_height()
    return node

def _remove(self, node, val):
  if node is None: return None

    if val < node.value:
      node.left = self._remove(node.left, val)
      node = resolve_right_leaning(node)                  ❷
    elif val > node.value:
      node.right = self._remove(node.right, val)
      node = resolve_left_leaning(node)                   ❸
    else:
      if node.left is None: return node.right
      if node.right is None: return node.left

        original = node
        node = node.right
        while node.left:
          node = node.left

        node.right = self._remove_min(original.right)
        node.left = original.left
        node = resolve_left_leaning(node)                 ❹

    node.compute_height()
    return node
```

❶ Если мы удаляем наименьший элемент, это по договоренности происходит в левом поддереве. При этом узел может перевесить вправо — тогда для восстановления баланса понадобится его повернуть.

❷ Если мы удаляем элемент из левого поддерева, узел может перевесить вправо, понадобится восстановление баланса и поворот.

❸ Если мы удаляем элемент из правого поддерева, узел может перевесить влево, понадобится восстановление баланса и поворот.

❹ Если мы нашли узел с нужным значением и удалили в правом поддереве минимальный узел, наш узел может перевесить влево, понадобится восстановление баланса и поворот.

После того как мы добавили в `BinaryTree` поддержку свойств АВЛ-дерева, каждое добавление или удаление элемента приводит к балансировке всей структуры. Балансировка выполняет всегда одинаковое количество действий, так что оценка времени ее работы должна быть константой — O(1). Методы поиска и обхода менять не нужно, потому что АВЛ-дерево продолжает оставаться двоичным деревом поиска.

Производительность сбалансированных деревьев

И вспомогательный метод `.compute_height()`, и все процедуры поворота работают константное время — ни в одной из них нет ни рекурсивного вызова, ни циклов. Все три метода вызываются, *только* когда обнаружился дисбаланс. Если внимательно посмотреть, как элемент добавляется в АВЛ-дерево, обнаружится, что на это потребуется не более одного поворота. Удаление элемента посложнее: теоретически возможен *наихудший случай*, когда поворотов потребуется несколько (одно из тренировочных заданий посвящено разбору этой ситуации). Но даже в *наихудшем случае* поворотов будет не больше $\log_2 N$, так что общее быстродействие операций поиска, добавления и удаления элементов можно оценить как $O(\log N)$.

Потренировавшись в этой главе на деревьях, мы можем начать исследовать любую рекурсивную структуру данных. А сейчас сравним работу уже известных нам структур — хеш-таблиц и приоритетных очередей — с работой двоичных деревьев: не окажутся ли они эффективнее?

Хранение пар (ключ, значение) в двоичном дереве

Структура двоичного дерева поиска позволяет без качественных изменений хранить в нем соответствующие ключу произвольные значения, то есть реализовать абстракцию «ассоциативный массив», которую мы уже наблюдали в хеш-таблицах.

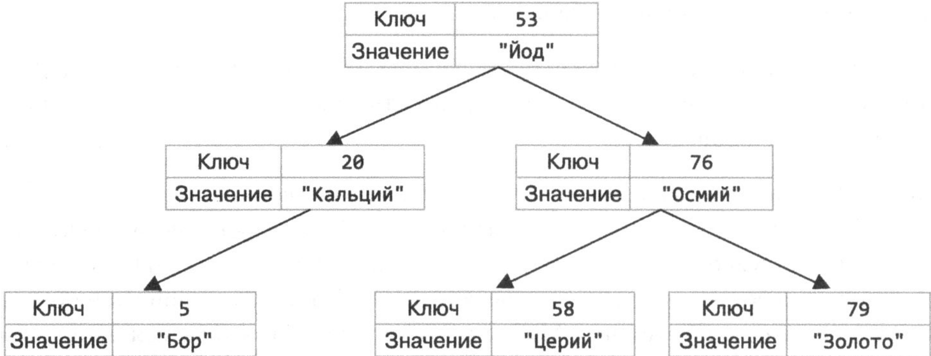


Рис. 6.14. Двоичное дерево поиска как ассоциативный массив: ключ — атомный вес химического элемента

Чтобы реализовать структуру, изображенную на рис. 6.14, необходимо доработать класс `BinaryNode` — в нем, помимо ключа, должно храниться значение. Пример 6.14 содержит такую доработку.

Пример 6.14. Класс BinaryNode, дополненный полем «значение»: теперь двоичное дерево может быть использовано как ассоциативный массив

```
class BinaryNode:
  def __init__(self, k, v):
    self.key = k                    ❶
    self.value = v                  ❷
    self.left = None
    self.right = None
    self.height = 0
```

❶ Ключ используется для поиска по дереву.

❷ Значение может содержать любые данные: они никак не используются при работе с двоичным деревом поиска.

Теперь методы `.insert(`*`ключ`*`)` и `__contains__(`*`ключ`*`)` надо заменить методами `.put(`*`ключ, значение`*`)` и `.get(`*`ключ`*`)` — тогда `BinaryTree` станет совместим

с нашей реализацией хеш-таблиц. Изменения минимальны и сводятся только к заполнению и возврату поля `.value`, так что мы их здесь не приводим. Полный текст всех примеров можно посмотреть в репозитории к книге (https://oreil.ly/fUosk). Обход дерева, то есть выбор между левым и правым поддеревьями, как и прежде, происходит на основании значения `node.key`.

Чем двоичное дерево несомненно лучше хеш-таблицы: значения из него можно извлекать *в порядке возрастания ключей*, потому что так работает метод прохода дерева `.__iter__()`.

Вспомним, что в главе 3 мы рассматривали хеш-таблицы с открытой адресацией и с раздельным хранением цепочек. Имеет смысл сравнить эффективность двоичного дерева поиска, хранящего значения, и с той и с другой реализацией хеш-таблиц, которые приведены в табл. 3.4. Для эксперимента мы взяли 321 129 слов английского словаря и добавили каждое во все три структуры данных. Какова минимальная высота дерева, в котором помещается 321 129 слов? Мы помним, что она равна $\log(N + 1) - 1$, что дает 17.293. И действительно, если все эти слова добавить в АВЛ-дерево *в порядке возрастания*, его высота окажется равной 18. В очередной раз убеждаемся, что хранить информацию в АВЛ-дереве весьма удобно.

Оба варианта хеш-таблиц из главы 3 оказываются *гораздо производительнее* двоичного дерева поиска[1], как это видно из табл. 6.5. Даже если понадобится отсортированный список ключей, выгоднее получить его как есть из хеш-таблицы, а затем отсортировать.

Таблица 6.5. Сравнение АВЛ-дерева и хеш-таблиц из главы 3 в качестве ассоциативных массивов (время в секундах)

Тип	Открытая адресация	Раздельные цепочки	АВЛ-дерево
Время добавления	0.54	0.38	5.00
Время поиска	0.13	0.13	0.58

Двоичное дерево как приоритетная очередь

Поскольку двоичная куча, описанная в главе 4, очевидно, основывается на двоичном дереве, было бы совершенно естественным сравнить производительность приоритетной очереди и аналогичной структуры на базе АВЛ-дерева, в которой ключом для обхода дерева является приоритет, как это изображено на рис. 6.15.

[1] Этого стоило ожидать, если вспомнить, что сложность операций над хеш-таблицей константная, а над деревом — все-таки логарифмическая. — *Примеч. пер.*

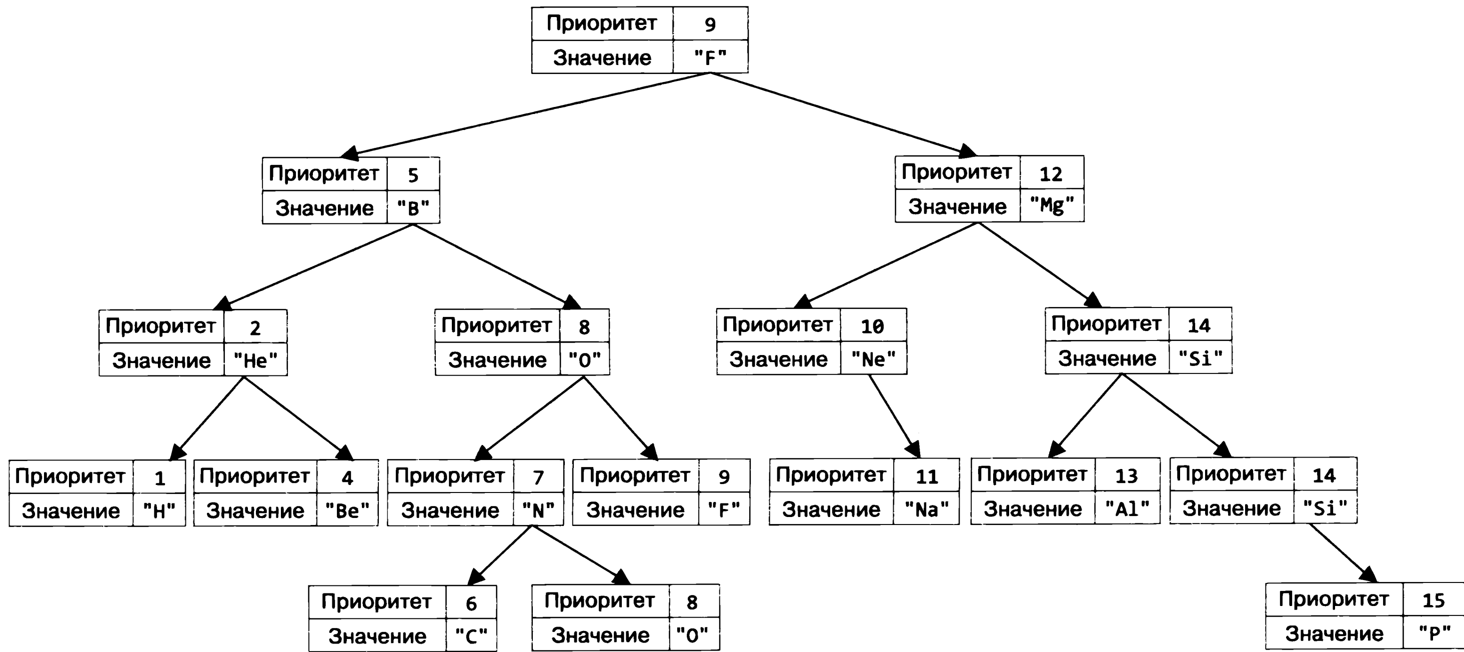


Рис. 6.15. Двоичное дерево поиска как приоритетная очередь: приоритет — это атомный вес химического элемента

Двоичное дерево поиска в качестве приоритетной очереди имеет два достоинства.

- Куча, основанная на массиве определенного размера, способна хранить только фиксированное количество элементов. Двоичное дерево может расти, как нам заблагорассудится[1].
- Поскольку в куче нет строгого порядка, а есть только частичный, итератор по элементам кучи *в порядке возрастания приоритетов* — это либо последовательное *снятие* элементов, либо имитация такого процесса. Так или иначе это требует отдельных ресурсов (см. соответствующее тренировочное задание в конце главы). В двоичном дереве порядок гарантируется процедурой обхода.

Для начала, как это сделано в примере 6.15, добавим в `BinaryNode` поля `.value` и `.priority`: первое поле, значение, как обычно, ни на что не влияет, а второе, приоритет, — это ключ для обхода дерева и операций добавления и удаления элемента.

Пример 6.15. Класс BinaryNode, дополненный полем «значение»: теперь двоичное дерево можно использовать как приоритетную очередь, в которой поле «приоритет» задает порядок

```
class BinaryNode:
  def __init__(self, v, p):
    self.value = v              ❶
    self.priority = p           ❷
    self.left = None
    self.right = None
    self.height = 0
```

❶ Значение может содержать любые данные: они никак не используются при работе с двоичным деревом поиска.

❷ Приоритет используется для поиска по дереву.

Мы помним, что в двоичной куче с порядком убывания элемент с наивысшим приоритетом всегда находится в `storage[1]` и поэтому доступен за время O(1). Если использовать двоичное дерево поиска, будет по-другому: узел с наивысшим

[1] Откровенно говоря, это фиктивное преимущество. В случае кучи мы можем сами масштабировать массив-хранилище; это относительно несложно сделать, как мы уже видели в одном из тренировочных заданий. Созданием и удалением экземпляра BinaryNode — иными словами, выделением и освобождением оперативной памяти под каждый узел дерева — занимается для нас Python, что в конечном итоге приводит к похожей стратегии — только не для одного массива, а для всего набора Python-объектов. — *Примеч. пер.*

приоритетом — крайний правый в дереве. Если дерево сбалансировано (например, как в этой главе), на поиск такого узла потребуется O(log *N*) времени.

Однако главное отличие приоритетной очереди от дерева в том, что произвольный элемент очереди удалять не приходится никогда: *удаляется только элемент с наивысшим приоритетом*. Стало быть, и общий метод `.remove()` не нужен, а вместо него, как это сделано в примере 6.16, добавим в класс `PQ` вспомогательный метод `._remove_max()`. Остальные вспомогательные методы — те же, что у стандартного класса, реализующего приоритетную очередь. Замечу, что класс `PQ` хранит и отслеживает количество пар в куче, *N*.

Пример 6.16. Класс PQ моделирует двоичную кучу с помощью BinaryTree и методов .enqueue() и .dequeue()

```
class PQ:
  def __init__(self):
    self.tree = BinaryTree()                              ❶
    self.N = 0

  def __len__(self):
    return self.N

  def is_empty(self):
    return self.N == 0

  def is_full(self):
    return False

  def enqueue(self, v, p):
    self.tree.insert(v, p)                                ❷
    self.N += 1

  def _remove_max(self, node):                            ❸
    if node.right is None:
      return (node.value, node.left)                      ❹

    (value, node.right) = self._remove_max(node.right)    ❺
    node = resolve_left_leaning(node)                     ❻
    node.compute_height()                                 ❼
    return (value, node)

  def dequeue(self):                                      ❽
    (value, self.tree.root) = self._remove_max(self.tree.root)
    self.N -= 1
    return value                                          ❾
```

❶ В качестве хранилища используем сбалансированное двоичное дерево поиска.

❷ Поставим в очередь пару (*`ключ, значение`*) — добавим ее в дерево и увеличим счетчик *N*.

❸ Вспомогательный метод `._remove_max()`, во-первых, удаляет узел с наивысшим приоритетом из дерева с корнем `node`, а во-вторых — возвращает кортеж вида (*значение, дерево*). Значение берется из удаленного узла, а дерево — это поддерево, получившееся после удаления.

❹ Основание рекурсии: правое поддерево пусто, значит, мы нашли узел с наивысшим приоритетом. Возвращаем его значение и его левое поддерево (оно займет место найденного узла).

❺ Рекурсивный вызов: получаем значение удаленного узла и поддерево, образовавшееся после удаления.

❻ Если узел потерял баланс, восстановим его с помощью поворотов.

❼ Вычислим новую высоту узла и возвратим ее вместе с удаленным значением.

❽ Метод `.dequeue()` удаляет узел с наивысшим приоритетом из двоичного дерева поиска и возвращает значение этого узла.

❾ Уменьшим счетчик, N, и возвратим значение, соответствующее наивысшему приоритету.

У нас получилась приоритетная очередь, быстродействие которой в два раза ниже, чем у аналогичной структуры поверх двоичной кучи (из главы 4), но порядок сложности остался $O(\log N)$, то есть отличие только в мультипликативной постоянной. Операции с АВЛ-деревом требуют больше действий, чем операции с кучей, но если, например, требуется регулярный обход пар (*значение, приоритет*) в порядке удаления, удобно использовать этот новый вариант.

Заключение

Двоичное дерево — это динамическая *рекурсивная* структура данных, в которой набор из N значений делится на *левое* и *правое* поддеревья, в каждом из которых содержится примерно половина всех элементов. Великое множество более сложных и эффективных структур данных основано на двоичных деревьях, например:

- красно-черные деревья, в которых используется более эффективный, но и существенно более сложный алгоритм балансировки, чем в АВЛ-деревьях;
- сбалансированные сильно ветвистые B- и B+-деревья (обычно используются в базах данных и файловых системах);
- R- и R*-деревья, ориентированные на хранение координатной информации (и включающие в себя понятия дистанции, достижимости и т. п.);

- K-мерные деревья (в частности, деревья квадрантов и октодеревья), используемые для хранения и поиска информации в многомерных пространствах.

По итогам этой главы:

- деревья имеют рекурсивную природу, поэтому и функции для работы с ними часто бывают рекурсивными;
- наиболее популярен *центрированный* (LNR, left-node-right) обход двоичного дерева, позволяющий получить элементы в порядке возрастания значений. В примере со структурой `Expression` мы использовали *обратный* (LRN, left-right-node) обход, позволяющий представить дерево в виде выражения в постфиксной записи — как для большинства микрокалькуляторов;
- двоичным деревьям поиска необходима постоянная балансировка, иначе пострадает заявленное быстродействие основных операций, O(log *N*). В АВЛ-деревьях балансировка обеспечена дополнительным полем — *высотой* поддерева — и тем, что любое действие должно сохранять базовое свойство: высота правого и левого поддеревьев не должна отличаться более чем на единицу;
- сбалансированное двоичное дерево поиска можно превратить в приоритетную очередь, хранящую пары (*значение*, *приоритет*), в которой сравнение узлов — это сравнение их приоритетов. Быстродействие такой структуры значимо не пострадает, а вдобавок можно будет центрированно, то есть в порядке возрастания, обходить весь набор данных *без изменения самой очереди*;
- с помощью двоичного дерева можно смоделировать и хеш-таблицу, если соблюдать уникальность ключа, по которому производится поиск. Впрочем, быстродействие такой модели оказывается хуже быстродействия хеш-таблицы в ее классической реализации, описанной в главе 3.

Тренировочные задания

1. Напишите рекурсивный метод `BinaryNode.count(n)`, который подсчитывает количество элементов двоичного дерева, кратных `n`, с вершиной в данном узле.
2. Нарисуйте двоичное дерево поиска, в котором для нахождения двух наибольших значений среди *N* элементов понадобится O(*N*) действий. А теперь нарисуйте двоичное дерево поиска, в котором для нахождения двух наибольших значений среди *N* элементов понадобится O(1) действий.
3. Когда возникает необходимость найти *k*-й по величине элемент в дереве, можно просто обойти *k* узлов, но такой способ неэффективен. Добавьте в класс `BinaryTree` метод `.select(k)`, который будет возвращать *k*-е наименьшее значение в дереве, где *k* выбирается из диапазона от 0 до *N* – 1. Для этого до-

бавьте в класс `BinaryNode` дополнительное поле N, в котором будет храниться общее количество значений в поддереве с корнем в данном узле (включая значение самого узла). Например, в каждом листовом узле N будет равно 1.

Добавьте еще один метод, `.rank(key)`, который будет возвращать число от 0 до $N - 1$ — позицию ключа в упорядоченном списке элементов. Иными словами, требуется подсчитать количество ключей в дереве, строго меньших, чем заданное.

4. Значения из списка `[3,14,15,19,26,53,58]` можно организовать в виде двоичного дерева поиска 7! = 5040 различными способами. Подсчитайте, сколько из таких деревьев будут строго сбалансированными с высотой 2, как на рис. 6.3. Можно ли обобщить ответ и вывести рекуррентное соотношение для `c(k)`, которое с произвольным k давало бы количество различных строго сбалансированных деревьев из $2^k - 1$ значений?
5. Добавьте в `BinaryTree` метод `.__contains__()`, который вызывал бы метод `.__contains__()` из `BinaryNode`.
6. АВЛ-деревья, как мы узнали в этой главе, сбалансированы, но не обязательно столь же компактны, как полное двоичное дерево. Какова наибольшая возможная высота АВЛ-дерева из N элементов? Сконструируйте 10 000 случайных АВЛ-деревьев из N элементов и зафиксируйте наибольшую высоту, которой удалось достичь при каждом N. Составьте таблицу, по которой можно будет отследить, когда наибольшая высота увеличивается. Попробуйте предсказать такие N, для которых наибольшая высота АВЛ-дерева из N элементов на единицу больше высоты АВЛ-дерева из $N - 1$ элементов.
7. Допишите класс `SpeakingBinaryTree` из примера 6.17. Он будет отчитываться о производимых с ним действиях при выполнении метода `.insert(`*значение*`)` примерно так, как это сделано в табл. 6.2. Данная рекурсивная процедура отличается от других, изучаемых нами в этой главе: отчет формируется «от начала», в то время как большинство рекурсивных процедур работают «от конца», то есть от основания рекурсии.

Пример 6.17. Допишите метод _insert() так, чтобы он возвращал описание того, что делает

```
class BinaryNode:
  def __init__(self, val):
    self.value = val
    self.left = None
    self.right = None

class SpeakingBinaryTree:
  def __init__(self):
    self.root = None
```

```
def insert(self, val):
  (self.root,explanation) = self._insert(self.root, val,
      'To insert `{}`, '.format(val))
  return explanation

def _insert(self, node, val, sofar):
  """
  Return (node, explanation) resulting from inserting val
  into subtree rooted at node.
  """
```

Измените стандартный метод `._insert()` так, чтобы он возвращал кортеж (***узел***, ***описание***), где ***узел*** — это корень дерева после вставки, а ***описание*** — строка, в которую с каждым рекурсивным вызовом добавляется описание выполняемого действия.

8. Напишите функцию `check_avl_property(n)`, которая позволяет убедиться, что в дереве с корнем в `n`, во-первых, все поддеревья имеют правильно подсчитанную высоту и, во-вторых, все узлы соответствуют требованию для разности высот АВЛ-дерева.

9. Напишите функцию `tree_structure(n)`, которая конструирует строку, отражающую *прямой обход* дерева с корнем в `n`. Каждое поддерево должно быть заключено в скобки. *Прямой обход* (NLR, node-left-right) предусматривает, что сначала выводится значение самого узла, затем — левое поддерево, а после — правое. Например, полное двоичное дерево с рис. 6.3 должно выглядеть так: `'(19,(14,(3,,),(15,,)),(53,(26,,),(58,,)))'`, а дерево из левой части рис. 6.5 — так: `'(5,(4,(2,(1,,),(3,,)),),(6,,(7,,)))'`. Напишите парную функцию `recreate_tree(`***выражение***`)`, которой на вход передается ***выражение***, представляющее структуру в формате `tree_structure(n)`. Эта функция должна возвращать корень реконструированного двоичного дерева.

10. Если считать *лево-правый* и *право-левый* поворот АВЛ-дерева такими же атомарными операциями, как *левый* и *правый* поворот, добавление элемента в дерево требует не более одной такой операции. Однако удаление элемента из АВЛ-дерева может потребовать более одной операции поворота.

11. Каков наименьший размер АВЛ-дерева, при удалении элемента из которого может потребоваться более одного поворота? В таком дереве, очевидно, должно быть не менее четырех узлов. Чтобы ответить на этот вопрос экспериментально, модифицируйте методы поворота так, чтобы для каждого удаления вычислялось количество проделанных поворотов. Кроме того, потребуется `tree_structure()` из предыдущего упражнения — чтобы можно было восстановить дерево *после того*, как оказалось, что удаление потребовало более одного поворота. Напишите функцию, которая конструирует 10 000 случайных АВЛ-деревьев, содержащих от 4 до 40 узлов, и в каждом таком дереве

удаляет случайный элемент. Должны обнаружиться АВЛ-деревья, для удаления элемента из которых требуется минимум три поворота. В частности, один поворот может потребоваться для дерева уже из четырех элементов, а два — для дерева из 12 элементов. Каков наименьший размер АВЛ-дерева, при удалении элемента из которого может потребоваться три поворота?

12. Наиболее компактное двоичное дерево, хранящее $N = 2^k - 1$ узлов, — полное, его высота равна k. А каково наименее компактное возможное АВЛ-дерево? Так называемые *деревья Фибоначчи* — это такие АВЛ-деревья, у которых в каждом узле высота левого поддерева больше высоты правого (всего на единицу, ибо больше нельзя). Замечу, что после добавления любого элемента в такое дерево потребуется балансировка. Напишите рекурсивную функцию, `fibonacci_avl(N)`, которая для любого $N > 0$ возвращала бы объект типа `BinaryNode` — корень дерева Фибоначчи из N первых натуральных чисел. Вероятно, проще будет не использовать `BinaryTree`. Корень такого дерева будет содержать N-е число Фибоначчи! Например, `fibonacci_avl(6)` вернет корневой узел дерева, показанного на рис. 6.16.

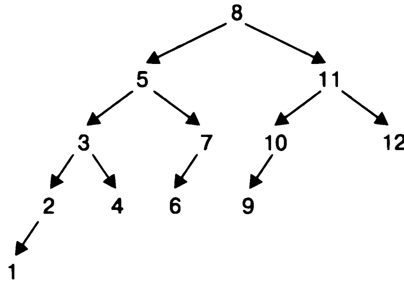


Рис. 6.16. Дерево Фибоначчи из 12 узлов

ГЛАВА 7

Графы: всегда на связи!

В этой главе

- Свойства *индексированной приоритетной очереди* — и больше новых типов данных изобретать в нашей книге не станем.
- *Граф* как набор вершин и соединяющих их ребер. В ориентированном графе ребра имеют направления, во взвешенном — числовые веса.
- **Обход графа в глубину** с помощью стека, хранящего план разведки.
- **Обход графа в ширину**, при котором план разведки хранится в очереди. Если выходной узел достижим из входного, **обход в глубину** вернет кратчайшее расстояние между ними.
- Поиск *цикла* — последовательности дуг, выходящей из некоторой вершины и входящей в нее же, — в ориентированном графе.
- **Топологическая сортировка** ориентированного графа, *которая строит* линеаризацию *его узлов — такую последовательность, в которой одна вершина всегда встречается раньше другой, если между ними есть дуга.*
- *Задача поиска кратчайшего пути из входной вершины взвешенного графа ко всем* остальным.
- Задача полного поиска кратчайших путей из каждой вершины взвешенного графа к каждой.

В графе удобно хранить полезную информацию

Каждый рассмотренный нами алгоритм решал некоторую задачу обработки информации: ввод данных, вычисление, вывод результата. Неисчислимое множество практических задач можно решить применением какого-нибудь известного алгоритма, если только удастся формализовать постановку и смо-

делировать данные. Вот три распространенные задачи, которые мы попробуем решить при помощи графов.

- Лабиринт – это комнаты и двери, соединяющие произвольную пару комнат. Найти кратчайший путь от входа к выходу.
- Проект – это набор заданий, причем некоторые задания можно выполнять только после завершения других заданий. Составить расписание, в котором описано, как последовательно выполнить все задания и тем самым завершить проект.
- Карта – это набор дорог различной длины, соединяющих города. Найти кратчайший суммарный путь между двумя заданными городами.

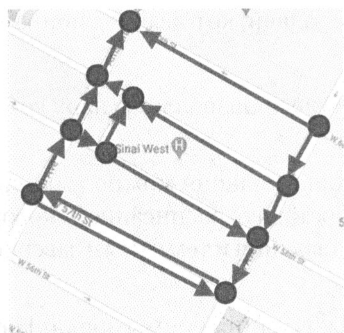
Решения каждой из этих задач можно построить в виде алгоритма на *графах*. Математики уже сотни лет изучают графы, потому что смоделировать связи *между* данными зачастую не менее важно, чем формализовать сами данные. *Граф* описывает данные как набор *узлов*, соединенных *ребрами*. Любую связь между двумя элементами данных u и v можно представить ребром $e = (u, v)$. Самих ребер может быть сколько угодно. Таким способом моделируются самые разные структуры из обыденной жизни; некоторые из них представлены на рис. 7.1. Химические связи между атомами углерода и водорода в структурной формуле молекулы пропана образуют *неориентированный граф*. Мобильное приложение может показать, как проехать по Нью-Йорку с учетом одностороннего движения на некоторых улицах – с помощью *ориентированного графа*. *Взвешенный граф* подходит для атласа автодорог Новой Англии, в котором отмечены расстояния между столицами штатов. Немного арифметики, и становится понятно, что кратчайший путь из Хартфорда, штат Коннектикут, до Бангора, штат Мэн, – 278 миль.

Граф как структура данных – это набор из N различных *узлов*, каждый из которых снабжен уникальным идентификатором[1]. В граф можно добавить *ребро*, которое соединяет два различных узла, u и v. Ребро записывается в виде пары (u, v), при этом узлы u и v называются *вершинами* ребра или *смежными* узлами.

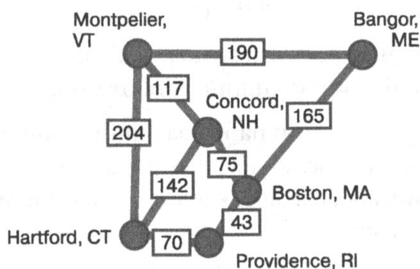
Граф на рис. 7.2 содержит 12 различных узлов и 12 ребер. Представим, что узел – это остров, а ребро – мост между двумя островами. С острова B2 можно добраться до острова C2, а с острова C2 – до островов B2 и C3. При этом добраться *напрямую* с острова B2 до острова B3 нельзя. Сначала надо пройти по мосту между B2 и C2, затем – по мосту между C2 и C3 и только потом – по мосту между C3 и B3. Если поизучать схему островов и мостов, можно подобрать цепочку мостов от любого острова B до любого острова C, а вот с островов A на остров B не попасть, хотя какие-то мосты есть и между островами A.

---

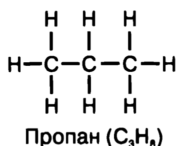
[1] Вслед за модулем networkx мы будем называть узлами элементы графа, а вершинами ребра – два узла, которые это ребро соединяют. – *Примеч. авт.*



Ориентированный граф — движение по улицам Нью-Йорка



Взвешенный граф — автодорожный атлас



Неориентированный граф — структурная формула молекулы

Рис. 7.1. Различные данные в виде графов

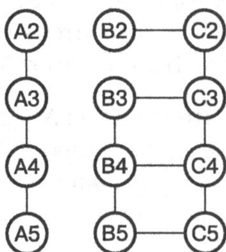


Рис. 7.2. Неориентированный граф с 12 вершинами и 12 ребрами

Если есть граф с узлами и ребрами, часто бывает нужно построить путь от некоторой *начальной* вершины (например, узла, соответствующего острову B2) к *конечной* (узлу, соответствующему острову B3), пользуясь при этом только ребрами графа.

Путь — это последовательность ребер, которая начинается в начальной вершине, а заканчивается в конечной. В простом случае каждое ребро — путь между вершинами этого ребра, но бывает и так, что ребра между двумя узлами

нет, а путь — есть. На рис. 7.2 путь между узлами B2 и B3 — это цепочка ребер (B2, C2), (C2, C3) и (C3, B3).

Каждое следующее ребро пути должно начинаться тем узлом, в котором заканчивалось предыдущее, поэтому путь можно представить последовательностью посещаемых узлов, например [B2, C2, C3, B3]. От B2 к B3 есть путь и подлиннее — [B2, C2, C3, C4, B4, B3]. Если путь начинается и заканчивается одним и тем же узлом, например [C4, C5, B5, B4, C4], — это *цикл*. Узел v называется *достижимым* из другого узла, u, если существует путь из u в v. Некоторые узлы графа могут быть недостижимы друг из друга, например, граф на рис. 7.2 не имеет пути из A2 в B2. Такой граф называется *несвязным*, а если между любыми двумя вершинами есть путь, то это *связный* граф.

Мы рассмотрим три типа графов.

- *Неориентированный граф*. Если ребро соединяет вершины u и v, то u считается смежной с v, а v — смежной с u. Можно считать такое ребро мостом, проходимым в обе стороны.
- *Ориентированный граф*. Каждое ребро (u, v) соединяет вершину u с вершиной v, и это значит, что v — смежная с u, но не обязательно наоборот. Получается, что ребро имеет направление от u к v, в котором можно было бы пройти по соответствующему мосту (но не обратно).
- *Взвешенный граф*. Ребро, соединяющее вершины u и v, имеет дополнительное числовое поле — *вес* — и записывается как (u, v, *вес*). Граф при этом может быть ориентированным, а может и не быть. Это поле отражает некоторую измеряемую характеристику связи u и v. Это может быть, например, стоимость проезда или расстояние в милях между городами, которым соответствуют узлы u и v.

Все графы в этой главе — *простые*, то есть не имеют ни дублирующих (соединяющих одну и ту же пару вершин) ребер, ни петель — ребер, в которых начальная и конечная вершины совпадают. Ребра в графе либо все ненаправленные, либо все направленные — и либо все с весами, либо все без весов.

Для наших алгоритмов понадобится достаточно сложная структура данных, моделирующая абстракцию графа. Хорошо бы, чтобы с соответствующим объектом-графом можно было делать следующее:

- получать количество узлов N и количество ребер E данного графа;
- получать полный набор узлов или вершин;
- для данного узла получить список ребер или смежных с ним узлов;
- модифицировать граф, добавляя туда узел или ребро;

- удалять из графа узел или ребро (вот это нам не понадобится, но для полноты картины хорошо, чтобы так можно было делать).

Среди стандартных библиотек Python есть модуль, реализующий некоторый алгоритм на графах, но он решает ровно одну прикладную задачу и не обладает почти никакими нужными нам свойствами. Нам понадобится куда более мощная сторонняя библиотека для работы с графами, NetworkX (https://networkx.org). Библиотека разрабатывается открыто и распространяется под свободной лицензией. Нам не придется изобретать колесо: модуль `networkx` предоставляет великое множество готовых алгоритмов на графах, а вдобавок с этим модулем прекрасно сочетаются библиотеки для визуализации графов. В примере 7.1 показана программа, в которой строится и используется граф с рис. 7.2, и результаты ее вывода.

Пример 7.1. Программа, в которой строится граф с рис. 7.2

```
import networkx as nx
G = nx.Graph()                                          ❶
G.add_node('A2')                                        ❷
G.add_nodes_from(['A3', 'A4', 'A5'])                    ❸
G.add_edge('A2', 'A3')                                  ❹
G.add_edges_from([('A3', 'A4'), ('A4', 'A5')])          ❺

for i in range(2, 6):
  G.add_edge(f"B{i}", f"C{i}")                          ❻
  if 2 < i < 5:
    G.add_edge(f"B{i}", f"B{i+1}")
  if i < 5:
    G.add_edge(f"C{i}", f"C{i+1}")

print(G.number_of_nodes(), 'nodes.')                    ❼
print(G.number_of_edges(), 'edges.')
print('adjacent nodes to C3:', list(G['C3']))           ❽
print('edges adjacent to C3:', list(G.edges('C3')))     ❾
12 nodes.
12 edges.
adjacent nodes to C3: ['C2', 'B3', 'C4']
edges adjacent to C3: [('C3', 'C2'), ('C3', 'B3'), ('C3', 'C4')]
```

❶ Создадим новый граф с помощью `nx.Graph()`.

❷ Узел может хранить любой константный объект Python (кроме `None`). Хорошая идея — хранить строки.

❸ Метод `.add_nodes_from()` добавляет сразу несколько узлов.

❹ Добавим ребро между узлами `u` и `v` с помощью метода `.add_edge(u, v)`.

❺ Добавим несколько ребер с помощью `.add_edges_from()`.

❻ Если вершина добавляемого ребра в графе *пока отсутствует*, соответствующий узел заводится автоматически.

❼ Мы можем узнать количество узлов и ребер графа.

❽ Мы можем получить список смежных с `v` узлов с помощью простой конструкции `G[v]`.

❾ Мы можем получить список исходящих из `v` ребер с помощью `G.edges(v)`.

Если задуматься, в какой последовательности нужно возвращать смежные вершины или ребра, окажется, что в разных случаях порядок может потребоваться разный. Поэтому в последующих алгоритмах мы вообще не будем рассчитывать на то, что такой порядок существует.

Обход лабиринта в глубину

Как написать программу, которая находит выход из лабиринта с рис. 7.3? Лабиринт размером 3 на 5 состоит из 15 комнат, вход в него – сверху, выход – снизу. Перемещаться по лабиринту можно только вертикально или горизонтально между теми комнатами, которые не разделены стеной. Первым делом надо смоделировать лабиринт с помощью неориентированного графа из 15 узлов, каждый из которых, с меткой (*строка, столбец*), будет обозначать соответствующую комнату лабиринта. Например, *вход* в лабиринт – это (0, 2), а *выход* – (2, 2). Затем в эту модель надо добавить ребра: если между узлами *u* и *v* *нет стены*, в графе должно быть ребро (*u*, *v*). Рисунок совмещает лабиринт с получившимся графом, оттого особенно заметно, что комнаты лабиринта в точности соответствуют узлам графа.

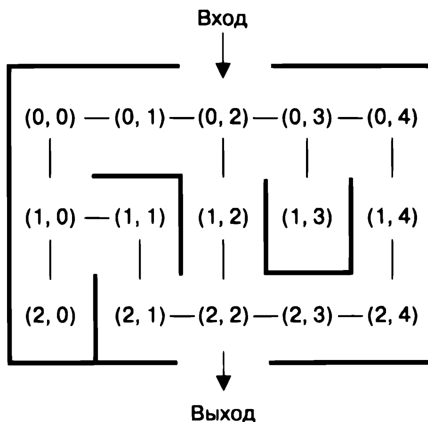


Рис. 7.3. Моделирование прямоугольного лабиринта графом

Найти путь между входным узлом (0, 2) и выходным (2, 2) — это то же самое, что пройти исходный лабиринт. Мы посмотрим, как проходить любой такой лабиринт независимо от размеров. Если самому проходить лабиринт, начинаешь исследовать то один, то другой путь, отбрасывая тупиковые, пока наконец один из путей не окажется от входа к выходу. Это, возможно, неочевидно, но человек имеет перед компьютером важное преимущество: *он видит лабиринт как целое* и может принимать решения, руководствуясь ощущениями, *близко ли к выходу* лежит предполагаемый путь. Представим себе, что мы угодили в такой лабиринт[1] и можем видеть только комнаты, напрямую связанные с той, в которой стоим. В таком положении подход к решению должен быть совсем другим.

Нам нужен способ проходить лабиринт, пользуясь соответствующим неориентированным графом, как это сделано на рис. 7.4. Случайное порождение таких графов — задача сама по себе интересная[2].

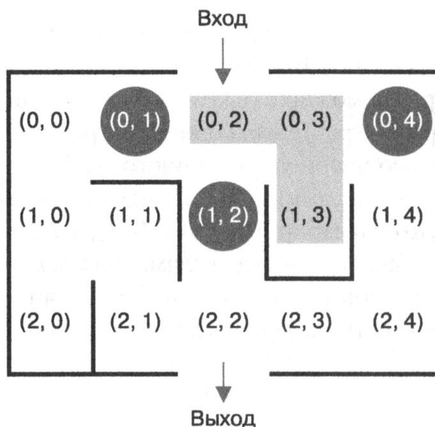


Рис. 7.4. Обходили лабиринт и уперлись в тупик

На входе — это узел (0, 2) — мы видим три смежных узла, к каждому из которых ведет ребро. Можно пойти на восток к (0, 3), но запомнить при этом два других направления, (0, 1) и (1, 2), как возможные альтернативы исследования. Узел (0, 3) также имеет три смежных, но *мы помним, откуда пришли*, так что не возвращаемся туда, где уже были. Из оставшихся двух выбираем, например, южный,

[1] Это может быть настоящий кукурузный лабиринт! Самый большой кукурузный лабиринт в мире — Корн Патч Пампкинс в Диксоне, штат Калифорния, — занимает площадь 63 акра, и проходить его можно часами. — *Примеч. авт.*

[2] См., например, программу ch07.maze в репозитории. — *Примеч. авт.*

(1, 3), а (0, 4) запоминаем как альтернативный. Вот мы и прошли по маршруту, отмеченному на рис. 7.2 серым, — и уперлись в тупик.

Узел (1, 3) — это тупик, потому что, когда мы туда попадаем, мы *не видим ни одного смежного не посещенного нами узла*. Что делать теперь? На рис. 7.4 кружками обведены узлы, которые были нам доступны, но мы туда не пошли: возможно, если мы *вернемся к одному из этих ранее отмеченных узлов и продолжим обход с него*, нам улыбнется удача?

Как превратить наши соображения в алгоритм? Вот примерный план.

- Будем отмечать каждый посещенный узел.
- Найдем все пока не посещенные узлы, смежные с данным. *Выберем любой из таких узлов*, а остальные добавим в план разведки. Переместимся в этот узел.
- Если упремся в тупик, вернемся к плану разведки — выберем оттуда последний добавленный (и еще не посещенный) узел и начнем исследование с него.
- Продолжим в том же духе до тех пор, пока не окажется, что мы посетили все достижимые узлы.

Хотелось бы, чтобы по окончании алгоритма мы могли легко построить путь от входного узла до любого достижимого. Результатом работы алгоритма должна быть структура данных с достаточным для решения этой задачи количеством информации. Обычно создают ассоциативный массив `node_from[]`, в котором в ячейке `node_from[v]` лежит значение `None`, если узел v недостижим из входного узла, или узел u, *из которого мы попали в* v, когда исследовали лабиринт. Для лабиринта с рис. 7.4 `node_from[(1, 3)]` будет хранить узел (0, 3).

Алгоритм по нашему плану нельзя реализовать как простой цикл наподобие поиска значения в списке. Нам придется ввести новую абстракцию — *стек* — и использовать соответствующую структуру данных. В стеке мы будем хранить план разведки, то есть узлы, которые еще предстоит исследовать при обходе графа.

Перед обедом в столовой вы наверняка берете верхний поднос из стопки. Абстракция «стек» имитирует поведение такой стопки. В стеке предусмотрено два действия: «положить значение на стек», `.append(`*`значение`*`)`[1], которое добавляет `значение` на вершину стека, и «снять значение со стека», `.pop()`, которое атомарно выполняет двойное действие: удаляет значение с вершины стека и возвращает это значение. Эту абстракцию также называют LIFO, по первым

[1] В теории операция «положить значение на стек» обычно называется push, но мы будем пользоваться методами типа list, который реализует абстракцию стека в Python. — *Примеч. пер.*

буквам фразы Last in, first out, то есть «последний добавляемый [элемент на стек] первым снимается [с него]».

Абстракцию стека в Python реализует уже знакомый нам тип `list`: будучи динамическим массивом, он, во-первых, позволяет хранить произвольное количество элементов, а во-вторых, за счет геометрического масштабирования операции по добавлению элемента в конец массива и удалению его оттуда имеет константную сложность.

Алгоритм **обхода в глубину** хранит в стеке план разведки — узлы, которые были доступны, но еще не исследованы при обходе лабиринта. В примере 7.2 приведена возможная реализация такого алгоритма. «В глубину» в названии происходит от того, что в алгоритме мы пытаемся сперва сделать очередной шаг, как если бы выход был в соседней комнате, а другие смежные узлы оставляем на потом.

Обход лабиринта начинается во входном узле `src`, который помечается как посещенный (то есть в `marked[src]` записывается `True`) и сразу кладется на стек как текущее окончание маршрута. Все время работы цикла `while` в стеке находятся уже посещенные узлы, которые встретились нам на пути к текущему. В цикле узлы снимаются по одному из стека, после чего все не посещенные доселе смежные с данным узлы кладутся обратно на стек — их еще предстоит исследовать.

Пример 7.2. Обход графа из указанного входного узла src в глубину

```
def dfs_search(G, src):                     ❶
  node_from, stack = {}, [src]              ❷
  marked = {src: True}                      ❸

  while stack:                              ❹
    v = stack.pop()                         ❺
    for w in G[v]:
      if not w in marked:
        node_from[w] = v                    ❻
        marked[w] = True                    ❼
        stack.append(w)

  return node_from                          ❽
```

❶ Обойдем граф `G`, начиная с входного узла `src` в глубину.

❷ В `node_from[w]` будет храниться узел, из которого мы попали в `w`, — это история обхода вплоть до `src` (которого в ней нет). На вершине стека — единственный узел, входной.

❸ Во входном узле мы уже побывали, так что он единственный помечен как посещенный.

❹ Если стек не пуст, значит, обход не завершен: есть еще не исследованные узлы.

❺ Следующий узел для проверки, v, лежит на вершине стека.

❻ Просмотрим все узлы w, смежные с v, и выберем из них еще не посещенные. Запомним, что каждый такой w доступен из v.

❼ Отметим w как посещенный (чтобы не исследовать потом повторно) и положим его на стек.

❽ Возвратим *историю обхода*: для каждого исследованного нами узла v в ней хранится узел, из которого мы попали в этот v, когда начали обход в стартовом узле src.

На рис. 7.5 показан **обход в глубину** с отображением состояния стека на каждом проходе цикла while. Выделенный на вершине стека узел — это комната, которую мы в данный момент исследуем, остальные узлы на стеке — комнаты, *которые мы исследуем в будущем*. Что не дает обходу в глубину бесцельно и бесконечно обшаривать одни и те же соседние комнаты? Каждый узел, который мы кладем на стек, помечается как уже исследованный, а это означает, что вторично на стек мы его никогда не положим. В цикле for (с переменной w) мы ищем еще не помеченные смежные с v узлы, помечаем их и кладем на стек, чтобы потом исследовать.

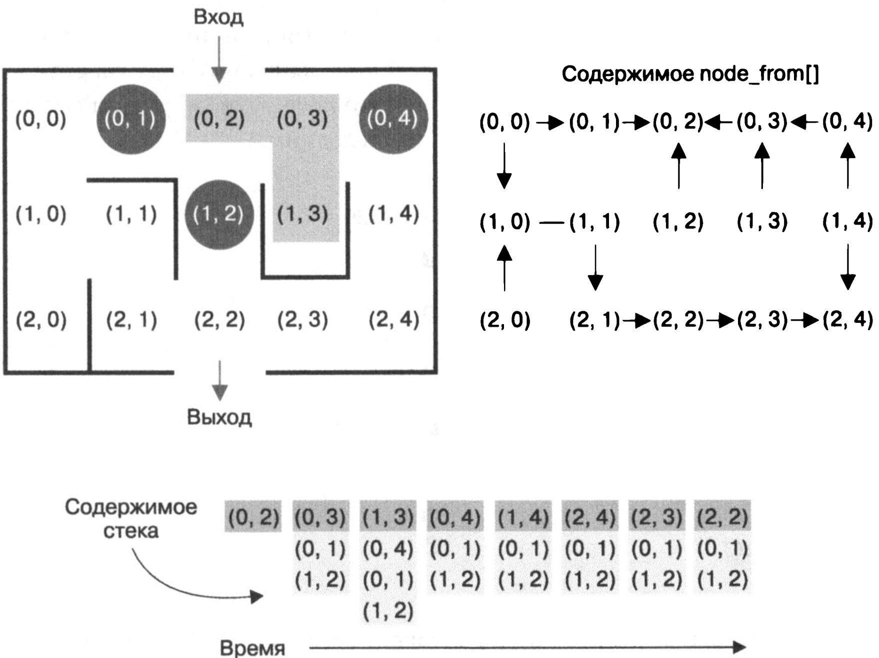


Рис. 7.5. Обход вглубь найдет любую комнату, доступную из входной

Обход сначала упирается в тупик на (1, 3), но немедленно откатывается до (0, 4) и продолжается дальше. Рисунок 7.5 показывает момент, когда выход уже найден, но обход продолжается, потому что не все узлы, достижимые из входного, помечены и стек не пуст.

Стек неизбежно опустеет, потому что, во-первых, количество узлов в графе конечно, а во-вторых, каждый помещаемый в стек узел предварительно помечается как посещенный. Снять пометку с узла невозможно, и значит, в конце концов любой доступный из `src` узел будет помечен и помещен на стек, а затем снят оттуда на очередном шаге цикла `while`.

Полученная *история обхода* вглубь показана на рис. 7.5 в правой части: это содержимое словаря `node_from[]`. Этот словарь можно назвать *деревом*, потому что стрелки не образуют циклы. По содержимому словаря можно восстановить путь от входа (0, 2) до любого достижимого узла в графе, если идти от *конца* пути. Например, `node_from[(0, 0)] = (1, 0)`, то есть предпоследний узел на пути к от (0, 2) к (0, 0) – это (0, 1).

Вообще-то хранящийся в истории обхода путь от входа к выходу далеко не самый короткий: он состоит из шести переходов. **Обход в глубину** не обязан выдавать кратчайшие пути, но зато непременно найдет путь до каждого узла, если до него вообще можно добраться от входа. Функция `path_to()` в примере 7.3 вычисляет по данному `node_from[]` путь от входа `src` к произвольному узлу `target` в виде последовательности узлов, пользуясь тем, что `node_from[v]` всегда содержит узел, который на пути от `src` до `v` был предпоследним.

Пример 7.3. Построение пути по истории обхода

```
def path_to(node_from, src, target):     ❶
  if not target in node_from:
    raise ValueError('Unreachable')       ❼

  path, v = [], target                    ❷
  while v != src:
    path.append(v)                        ❸
    v = node_from[v]                      ❹

  path.append(src)                        ❺
  path.reverse()                          ❻
  return path
```

❶ Для того чтобы найти путь из `src` в `target` (если он есть), достаточно знать историю обхода.

❷ Переменная v будет проходить по обратному пути от `target` до `src`.

❸ Пока v не добралось до `src`, добавляем v в обратный путь.

❹ Продолжаем идти назад, теперь v — это предыдущий узел в пути, `node_from[v]`.

❺ Цикл заканчивается, когда мы доходим до `src`. Сам `src` для полноты картины надо добавить в `path`.

❻ Переставляем элементы `path` в обратном порядке, превращая обратный путь в прямой, и возвращаем его.

❼ Сразу убедимся, что `target` достижим из `src`, то есть содержится в `node_from[]`.

Функция `path_to()` строит обратную последовательность узлов — от `target` до `src`, — а затем просто обращает ее, и получается искомый путь от `src` до `target`. Если попытаться построить путь до недостижимого узла, `path_to()` породит исключение `ValueError`.

Обходя лабиринт вглубь, мы всякий раз выбираем одно случайное направление, как бы предполагая, что находимся в шаге от выхода. Зачастую нужен другой, более методичный, способ обхода.

Другой способ обхода: в ширину

Обход лабиринта в ширину устроен таким образом, что комнаты нам встречаются *в порядке их удаленности от входа*. Возьмем тот же лабиринт, что и на рис. 7.3, и напишем на каждой комнате *кратчайшее расстояние* от входа. Если нам удастся придумать, как сделать это достаточно эффективно, мы и получим **обход в ширину**, при котором вычисляется кратчайший (состоящий из наименьшего количества ребер) путь от входного узла до любого достижимого. Например, из рис. 7.6 очевидно, что кратчайший путь от входа к выходу пролегает всего через три узла.

Начнем с некоторых идей относительно **обхода в ширину**. Входной узел лабиринта на рис. 7.6 имеет три смежных узла, до которых от него всего один шаг. Любой из этих узлов может оказаться началом кратчайшего пути к выходу, но, конечно, нельзя заранее сказать какой. Поэтому вместо того, чтобы выбирать одно направление и продвигаться на следующий узел, при обходе в ширину мы должны сначала исследовать каждый из уже возможных шагов вперед — тогда у нас получится множество узлов, до которых в сумме *два* шага. Выглядит так, как если бы мы, обходя граф, перестали наспех выбирать первое попавшееся направление, а начали методично обшаривать одно за другим.

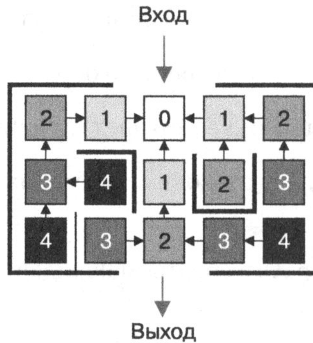


Рис. 7.6. Обход в ширину определяет кратчайший путь до каждой вершины

Продолжая аналогию: **обход в глубину** оптимистичен, ибо ожидает выхода за первым же поворотом, а **обход в ширину** предпочитает *порядок* и просматривает все узлы, расстояние до которых от входа на один шаг больше. Это дает еще четыре узла (на рис. 7.6 помечены двойкой) — они от входа в двух шагах. Если продолжать исследовать узлы таким манером, то после того, как *в свою очередь* мы рассмотрим все узлы, что в двух шагах от входа, мы найдем четыре узла, до которых от входа три шага. Процесс будет повторяться до тех пор, пока мы не побываем во всех достижимых от входа узлах.

Для обхода в ширину нам понадобится особая структура данных, которая гарантировала бы нам, что узел, находящийся на расстоянии $d + 1$ от входа, попадется не ранее всех узлов, находящихся от входа на расстоянии d. Такую структуру данных мы знаем — это очередь, описанная нами в главе 4, ибо в ней поддерживается дисциплина «первым вошел, первым вышел» (FIFO, First in first out): узлы, добавленные в очередь сначала, непременно будут рассмотрены первыми. Программа из примера 7.4 практически такая же, что и для **обхода в глубину**, за исключением того, что используется очередь и в очереди хранится *пространство активного поиска*, то есть набор узлов, которые в данный момент необходимо исследовать.

Пример 7.4. Обход графа в ширину от заданного узла

```
def bfs_search(G, src):          ❶
  marked = {src: True}           ❷
  node_from = {}                 ❸

  q = Queue()
  q.enqueue(src)                 ❹

  while not q.is_empty():        ❺
    v = q.dequeue()
```

```
    for w in G[v]:
      if not w in marked:
        node_from[w] = v        ❻
        marked[w] = True        ❼
        q.enqueue(w)

  return node_from              ❽
```

❶ Функция обхода графа `G` в ширину начиная с входного узла `src`.

❷ В словаре `marked` мы храним уже исследованные узлы — вход уже там.

❸ В словаре `node_from` будет храниться история обхода: для каждого узла `w` в `node_from[w]` содержится узел, из которого мы попали в `w` при обходе, то есть очередной узел по дороге обратно к `src`.

❹ Добавим входной узел в очередь разведки. В начале очереди будет всегда находиться узел, который нам нужно проверить следующим.

❺ Если наш **обход в ширину** не завершен, снимем из очереди очередной исследуемый узел, `v`.

❻ Перебираем все смежные с `v` и притом непомеченные узлы `w`. Запоминаем, что путь до них лежит через `v`.

❼ Ставим `w` в конец очереди (плана разведки) и помечаем его, чтобы не исследовать повторно.

❽ Возвращаем *историю обхода*, в которой для каждого узла `v` хранится узел, из которого мы попали в `v`, когда начинали обход из `src`.

История **обхода в ширину** содержит узлы в порядке их удаления от входа, поэтому любой воссозданный по этой истории путь до любого достижимого узла окажется кратчайшим[1]. Воссоздать путь из `src` до любого достижимого оттуда узла можно с помощью все той же функции `path_to()`. На рис. 7.7 хорошо видно, как методично исследуется граф при **обходе в ширину**.

Мы используем очередь для хранения плана разведки в порядке удаления от входа, и узлы на рис. 7.7 помечены разными оттенками в зависимости от расстояния до входного узла. Если мы упираемся в тупик, новые узлы в очередь не добавляются. Замечу, что выходной узел (2, 2) мы добавляем в очередь внутри цикла `for`, но иллюстрация сделана в момент, когда этот узел снимается из очереди во внешнем цикле `while`. К этому времени все узлы, расстояние

[1] В принципе, может быть несколько путей одинаковой кратчайшей длины, и единственный найденный при обходе в ширину путь в любом случае будет не длиннее возможных альтернатив. — *Примеч. авт.*

до которых меньше двух шагов, мы уже просмотрели, а самый последний узел в очереди уже в трех шагах от входа.

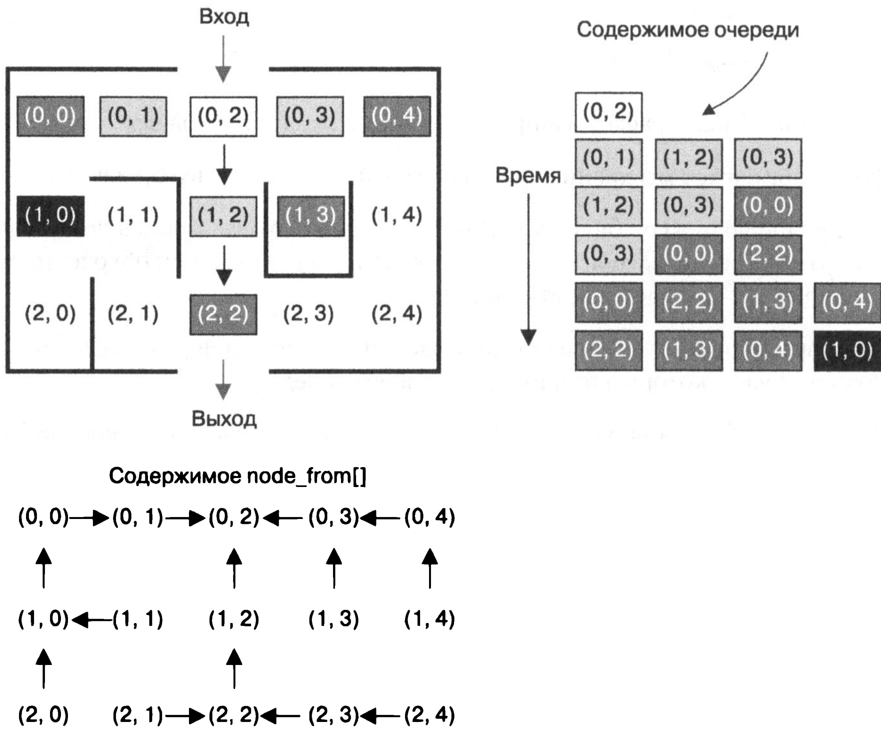


Рис. 7.7. Обход в ширину дает кратчайший путь

Из чистого любопытства рассмотрим еще один способ найти путь от входа к выходу, при котором мы *учитываем, как далеко от выхода находимся*. И обход в глубину, и обход в ширину — это *слепой поиск*: план обхода в них строится исходя только из информации об узлах, смежных с данным. Если подключить к поиску выхода какой-никакой искусственный интеллект, мы получим немало новых, более эффективных алгоритмов решения нашей задачи, но для этого надо снабдить их дополнительной информацией о предмете исследования.

При **направленном обходе** мы будем просматривать узлы в порядке их приближения к выходу. Чтобы определить приближение, надо понять, как измерять расстояние до узла. Нам подойдет так называемое *расстояние городских кварталов*[1],

[1] Названное так потому, что в городе с прямоугольной планировкой невозможно двигаться по диагонали — только по вертикали и по горизонтали. — *Примеч. авт.*

иначе называемое *манхэттенским расстоянием* или *прямоугольной метрикой*. Если мы посчитаем, сколько в сумме рядов и столбцов разделяют два узла, мы и получим манхэттенское расстояние. Например, узел (2, 0) в левом нижнем углу нашего лабиринта окажется по этой метрике в четырех шагах от узла (0, 2), потому что и по вертикали он отстоит на две строки, и по горизонтали — на два столбца.

Из трех узлов, смежных с (0, 2), в **направленном обходе** мы должны сначала исследовать (1, 2), потому что он всего в шаге от выхода, (2, 2), а два других узла — оба от выхода в трех шагах (согласно прямоугольной метрике). Получается, что для этого алгоритма узлы должны храниться так, чтобы всякий раз можно было получить узел, ближайший к выходу.

Проще всего применить здесь приоритетную очередь с порядком возрастания — или уже реализованную нами в главе 4 приоритетную очередь с порядком убывания, но в качестве приоритета использовать *отрицательное число, равное по модулю манхэттенскому расстоянию от узла до выхода*. Таким образом получится, что два узла — `u` на расстоянии десяти шагов от выхода и `v` на расстоянии пяти — будут помещены в приоритетную очередь как `(u, -10)` и `(v, -5)`, и, следовательно, приоритет узла `v`, который ближе к выходу, окажется больше. Пример 7.5 похож на аналогичные функции для **обхода в глубину** и **обхода в ширину**, но план разведки — узлы, которые нужно исследовать, — будет храниться в приоритетной очереди.

Пример 7.5. Направленный обход, порядок которого обусловлен манхэттенским расстоянием от выхода

```
def guided_search(G, src, target):                  ❶
  from ch04.heap import PQ
  marked, node_from = {src: True}, {}               ❷

  pq = PQ(G.number_of_nodes())                      ❸
  pq.enqueue(src, -distance_to(src, target))        ❹

  while not pq.is_empty():                          ❺
    v = pq.dequeue()
    for w in G.neighbors(v):
      if not w in marked:
        node_from[w] = v                            ❻
        marked[w] = True
        pq.enqueue(w, -distance_to(w, target))      ❼

  return node_from                                  ❽

def distance_to(from_cell, to_cell):
  return abs(from_cell[0] - to_cell[0]) + abs(from_cell[1] - to_cell[1])
```

❶ Функция обхода графа `G`, начиная с входного узла `src` по направлению к заранее известному выходу.

❷ В словаре `marked` будем хранить уже просмотренные узлы. Вход уже там. Историю обхода — в словаре `node_from` (напомню, что в `node_from[w]` хранится узел, который был предыдущим на пути от входа в `w`).

❸ Используем приоритетную очередь на базе кучи из главы 4: размер очереди надо задать заранее. Очереди, способной разместить все узлы графа, явно достаточно.

❹ Поместим вход, `src`, в приоритетную очередь — с него и начнется исследование. Приоритет — отрицательное число, по модулю равное расстоянию до выхода, `target`.

❺ До тех пор пока очередь не пуста, обход не закончен. В начале очереди — всегда *ближайший* к выходу помеченный узел, с него и продолжим исследование.

❻ Запоминаем в истории обхода, что до каждого непомеченного узла `w`, смежного с `v`, мы дошли из `v`.

❼ Ставим `w` в приоритетную очередь. Он займет там подобающее место с приоритетом, равным *отрицательному манхэттенскому расстоянию* до входа; пометим этот узел.

❽ Возвратим *историю обхода*, в которой для каждого узла `v` хранится предыдущий (при обходе с началом из узла `src`).

Толику искусственного интеллекта **направленному обходу** придает функция `distance_to()`, которая вычисляет манхэттенское расстояние между двумя узлами.

Не факт, что **направленный обход** выдаст кратчайший путь к выходу, к тому же предполагается, что выход единственный и мы заранее знаем, где он. В частности, путь может оказаться длиннее, чем то, что выдает **обход в ширину**, потому что там гарантируется, что он будет кратчайшим, причем не только к выходу, но и к каждому достижимому узлу в графе. Зато можно надеяться, что **направленный обход** реже будет заглядывать в те части лабиринта, где искать выход нет смысла, — как минимум в случайных лабиринтах[1]. На рис. 7.8 можно сравнить изображенные рядом планы обхода одного и того же лабиринта всеми тремя нашими способами.

[1] *Наихудший случай* можно подобрать практически к любому направленному обходу. Если взять лабиринт, в котором есть всего одна развилка: два смежных со входом узла и ближайший к выходу из них ведет в тупик, направленный обход с прямоугольной метрикой осмотрит весь граф целиком. — *Примеч. пер.*

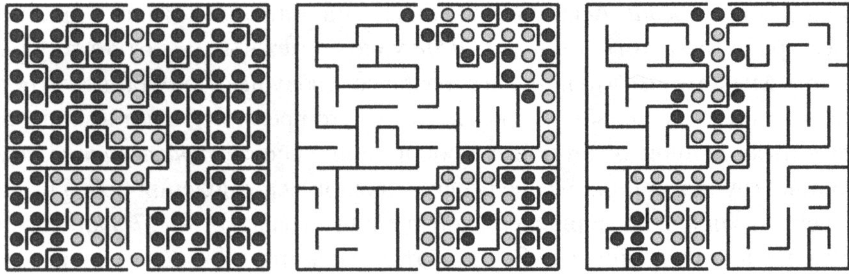


Рис. 7.8. Сравнение обхода лабиринта в глубину, в ширину и по направлению к выходу

Вероятно, что в поиске выхода из лабиринта больше всего узлов мы исследуем при **обходе в ширину**: слишком уж последователен этот алгоритм. **Обход в глубину** вряд ли выдаст кратчайший путь: для этого каждое выбранное при обходе направление должно оказаться верным. Нет гарантии, что **направленный обход** выдаст кратчайший путь от входа к выходу, но в случайном лабиринте с рис. 7.8 направление на выход помогло существенно сократить неуспешные попытки поиска.

Все наши алгоритмы обхода использовали словарь `marked`, узлы из которого не надо исследовать, и в результате в любой из N узлов графа мы заглядываем не больше раза. Тогда можно предположить, что сложность любого из них — $O(N)$, но, чтобы это доказать, надо оценить производительность различных действий внутри цикла. Операции `push()` и `pop()` над стеком имеют константную сложность. Осталось только понять, как работает цикл `for w in G[v]`, в котором перебираются все смежные с `v` узлы. Здесь все несколько сложнее: необходимо знать, как именно хранятся ребра в графе. Есть два способа хранить ребра (оба показаны на рис. 7.9): матрица смежности и список смежности.

- *Матрица смежности*. Двумерный массив M размером $N \times N$, хранящий N^2 булевых значений. Все узлы нумеруются, то есть каждому узлу `u` ставится в соответствие число u^{idx} в диапазоне от 0 до N. Если существует ребро из `u` в `v`, значение $M[u^{idx}][v^{idx}]$ истинно. На иллюстрации значение `True` обозначено закрашенной клеткой. С помощью матрицы смежности мы можем найти все узлы, смежные с данным, за N шагов, так как для этого необходимо проверить одну полную строку матрицы. Соответственно, *независимо от действительного количества смежных узлов для каждого узла лабиринта* сложность такого действия будет $O(N)$. Поскольку сам цикл `while` проходит N итераций, а внутренний цикл сам по себе имеет сложность $O(N)$, выходит, что наше исходное предположение неверно и сложность всего алгоритма — квадратичная, $O(N^2)$.

- *Список смежности.* Ассоциативный массив, ключом которого выступает узел графа, а хранится в нем *список* узлов, смежных с соответствующим узлом. Мы помним, что поиск в словаре по ключу имеет константную сложность, а поиск в списке — линейную, так что время выборки всех вершин, смежных с данной, будет прямо пропорционально *степени вершины*, то есть количеству исходящих из нее ребер. Порядок перечисления ребер не определен, он зависит от порядка их добавления в граф. На рис. 7.9 для каждого узла представлен список ребер, при этом у некоторых узлов смежных мало, а у некоторых — больше.

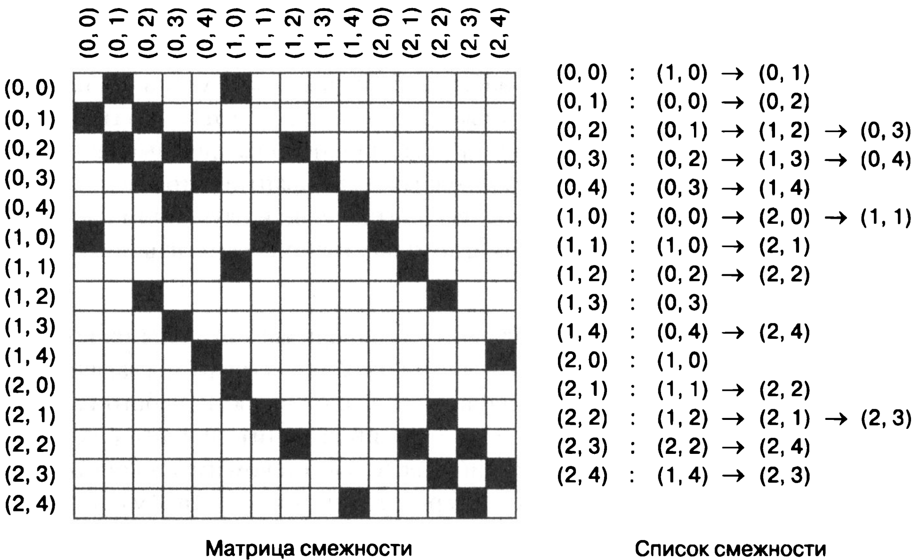


Рис. 7.9. Как выглядят матрица смежности и список смежности

В примере 7.6 приведена часть программы, из которой понятно: если граф представлен с помощью списка смежности, быстродействие **обхода в глубину** зависит от общего количества ребер в графе. Можно легко подсчитать и сколько узлов обрабатывается в программе, и сколько ребер.

Пример 7.6. Часть программы, иллюстрирующая зависимость быстродействия от количества ребер

```
while stack:
  v = stack.pop()
  for w in G[v]:
    if not w in marked:
```

```
        marked[w] = True
        stack.append(w)
        ...
```

Мы помним, что каждый узел v добавляется на стек не более одного раза. Это значит, что оператор `if` выполняется *для каждого смежного с* v *узла*. Возьмем простейший граф из двух узлов, u и v, и одного ребра (u, v) — уже на нем `if` выполнится *дважды*: сначала при поиске вершин, смежных с u, а затем — при поиске вершин, смежных с v. В общем, на ненаправленных графах этот `if` будет выполняться $2E$ раз, где E — общее число ребер в графе.

Если учесть все действия — вызовы `.append()`/`.pop()`, которых может быть не более N, и $2E$ раз выполненный оператор `if`, можно смело утверждать, что быстродействие алгоритмов, использующих список смежности, равно $O(N + E)$, где N — количество узлов, а E — количество ребер графа. Утверждение верно и для **обхода в ширину**, в котором используется не стек, а очередь, но производительность оказывается такой же.

В каком-то смысле все наши оценки совпадают: в самом деле, неориентированный граф с N узлами может содержать не более $E = N \times (N - 1) / 2$ ребер[1]. Так что независимо от того, используется ли для хранения графа матрица смежности или список смежности, обход графа с достаточно большим количеством ребер потребует порядка $N \times (N - 1) / 2$ действий в *наихудшем случае*, то есть сложность окажется $O(N^2)$.

А вот **направленный обход** для хранения узлов использует приоритетную очередь с порядком — расстоянием до выхода. Операции `enqueue()` и `dequeue()` в *наихудшем случае* имеют сложность $O(\log N)$. Сами методы вызываются N раз, и каждое ребро просматривается дважды, так что быстродействие **направленного обхода** в *худшем случае* оказывается $O(N \log N + E)$.

Ориентированные графы

Графы можно использовать для представления данных в задачах, где связь между узлами — односторонняя (обычно обозначается стрелкой). *Дуга* — ориентированное ребро — (u, v) соединяет u с v, и это значит, что вершина v смежна с u, но и только: дуга не делает вершину u смежной с v. В ориентированном ребре (u, v) вершина u называется, естественно, *началом* дуги, а v — ее *концом*. В графе на рис. 7.10 есть дуга (B3, C3), но нет дуги (C3, B3).

[1] И снова здравствуйте, треугольные числа! — *Примеч. авт.*

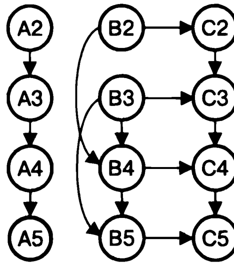


Рис. 7.10. Пример ориентированного графа с 12 узлами и 14 ребрами

Ориентированный граф (или *орграф*) можно обходить как в глубину, так и в ширину. Единственное отличие в том, что дуга (u, v) означает, что вершина v смежна с u, а для того, чтобы u была смежна с v, нужно, чтобы в графе была и дуга (v, u). Многие алгоритмы на орграфах можно упростить, если реализовать их рекурсивно, как в примере 7.7. Можно видеть, что эта программа заимствует основные компоненты из нерекурсивной реализации.

Пример 7.7. Рекурсивная реализация обхода графа в глубину

```
def dfs_search(G, src):              ❶
  marked, node_from = {}, {}         ❷

  def dfs(v):                        ❸
    marked[v] = True                 ❹
    for w in G[v]:
      if not w in marked:
        node_from[w] = v             ❺
        dfs(w)                       ❻

  dfs(src)                           ❼
  return node_from                   ❽
```

❶ Функция обхода графа `G` вглубь начиная с входного узла `src`.

❷ Словарь `marked` хранит уже посещенные узлы, словарь `node_from` — историю обхода графа функцией `dfs()`: в `node_from[w]` содержится предыдущий узел в пути от `src` до `w`.

❸ Рекурсивная функция, при каждом вызове которой обход продолжается с непомеченного узла `v`.

❹ Пометим узел `v` как посещенный.

❺ Для каждого непомеченного смежного с `v` узла `w` запоминаем, что дойти до него можно было от `v`.

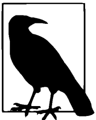
❻ Рекурсивный вызов: продолжим исследование с непомеченного узла `w`, а после возврата из вызова выберем очередной узел `w` в цикле `for`.

❼ Стартовый рекурсивный вызов `dfs()` с параметром — входным узлом `src`.

❽ Возвращаем *историю обхода*, в которой для каждого узла `v` записано, каким был предыдущий узел на пути от `src` до `v`.

Обратите внимание на то, что в рекурсивном алгоритме нет выделенного стека, в котором хранится план разведки. В действительности такой стек задан неявно — это последовательность контекстов рекурсивных вызовов функции `dfs()`, в каждом из которых есть локальные переменные `u` и `v`, а также список `G[v]`.

Каждый контекст `dfs(v)` в стеке рекурсивных вызовов содержит собственную локальную переменную `v`, которая является частью активного пространства поиска. Основание рекурсии в `dfs(v)` наступает, когда у узла `v` больше нет непомеченных смежных узлов `w`, при этом ничего не происходит, и контекст после возврата уничтожается. Для каждого непомеченного смежного узла `w` происходит рекурсивный вызов `dfs(w)`. После возврата из рекурсивного вызова цикл `for` продолжается: возможно, есть еще непомеченные смежные с `v` узлы, которые надо будет исследовать с помощью `dfs()`.

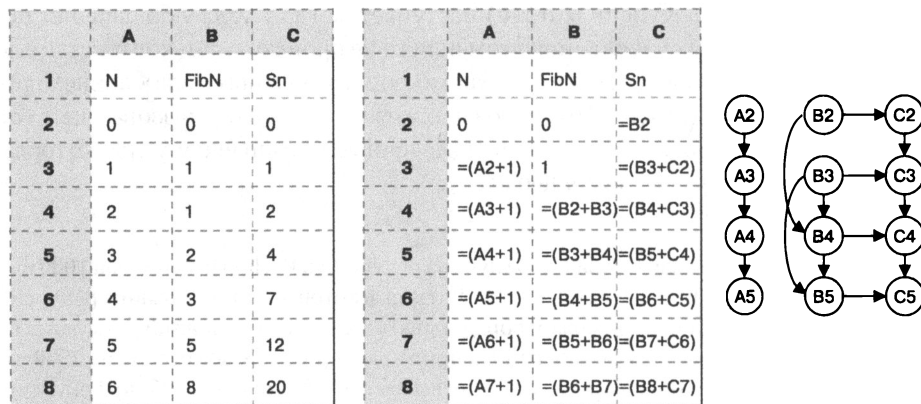


Я в книге уже отмечал, что глубина рекурсии в Python ограничена 1000 вызовами, так что с помощью алгоритмов, в которых использование рекурсии неоправданно, мы можем обрабатывать только небольшие объемы данных. Например, прямоугольный лабиринт размером 50 × 50 состоит из 2500 узлов, и наш обход вглубь явно превысит ограничение на глубину рекурсии. Если заменить стек рекурсивных вызовов настоящим стеком, в котором хранится то же, что было в локальных переменных рекурсивной функции (в нашем случае — пространство активного поиска), проблема исчезнет. Увы, текст программы при этом становится более сложным и менее читаемым, так что в этой главе мы будем использовать «неоправданный» рекурсивный вариант.

Исходный текст функции, которая обходит граф вглубь с использованием стека вместо рекурсии, можно найти в файле ch07/search.py репозитория[1].

[1] Напомним о логарифмическом принципе при выборе рекурсивного алгоритма: если в наихудшем случае можно ожидать, что на наборе данных размером N мы получаем глубину рекурсии порядка $O(N)$, мы в действительности написали обычный цикл и лучше заместить рекурсию стеком; если же глубина рекурсии имеет меньший порядок (например, $O(\log N)$ или даже константу), то можно смело пользоваться рекурсивным алгоритмом. — *Примеч. пер.*

С помощью орграфов можно моделировать объекты из самых разных предметных областей и для этих объектов решать бесчисленное множество задач. На рис. 7.11 представлена электронная таблица, ячейки которой однозначно определяются названием столбца и номером строки. В ячейке B3 записана единица. Слева таблица изображена так, как ее видит пользователь, а затем в виде чисел, ссылок на другие ячейки или формул, которые в действительности записаны в ячейках. Формула в ячейке может использовать константы или содержимое других ячеек, которое, в свою очередь, тоже может вычисляться по формуле. Формат представления формул — инфиксный, такие формулы мы видели в главе 6. Например, в ячейке A4 записана формула =(A3 + 1). В ячейке A3 записано =(A2 + 1), что равно 1, потому что A2 равно 0 — а значит, A4 равно 2. В репозитории к книге есть программа, реализующая отношения в этой таблице.



	A	B	C
1	N	FibN	Sn
2	0	0	0
3	1	1	1
4	2	1	2
5	3	2	4
6	4	3	7
7	5	5	12
8	6	8	20

	A	B	C
1	N	FibN	Sn
2	0	0	=B2
3	=(A2+1)	1	=(B3+C2)
4	=(A3+1)	=(B2+B3)	=(B4+C3)
5	=(A4+1)	=(B3+B4)	=(B5+C4)
6	=(A5+1)	=(B4+B5)	=(B6+C5)
7	=(A6+1)	=(B5+B6)	=(B7+C6)
8	=(A7+1)	=(B6+B7)	=(B8+C7)

Рис. 7.11. Пример электронной таблицы, соответствующей ориентированному графу

В первой строке электронная таблица содержит заголовки столбцов — N, FibN и Sn. Во всех остальных строках в столбце A содержатся первые семь целых неотрицательных чисел N, в столбце B — первые семь чисел Фибоначчи[1]. В каждой строке столбца C записана *частичная сумма* первых N чисел Фибоначчи (например, в C7 хранится 12, что равно 0 + 1 + 1 + 2 + 3 + 5). В правой части рис. 7.11 изображен орграф, отражающий зависимость ячеек друг от друга. Например, дуга между A2 и A3 означает, что для вычисления A3 надо знать значение A2, то есть, во-первых,

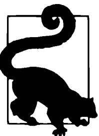
[1] Мы помним из главы 5, что это 0, 1, 1, 2, 3, 5 и 8 — последовательность, в которой каждый следующий член равен сумме двух предыдущих. — *Примеч. авт.*

если изменится A2, то изменится и A3 и, во-вторых, перед тем, как вычислять A3, необходимо вычислить A2.

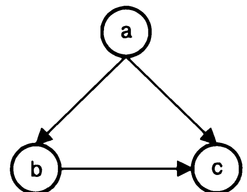
Ячейки электронной таблицы не могут содержать *взаимных ссылок*: например, если в ячейке C2 написано =B2, то в ячейке B2 не должно быть =C2 — значение таких ячеек вычислить нельзя, это считается ошибкой. В орграфах подобная ситуация называется *контуром*. Контур — это последовательность дуг, которая начинается и заканчивается в некоем узле *u*. Любая система, работающая с электронными таблицами, должна убедиться, что в ней нет контуров, а значит, ячейки без взаимных ссылок можно вычитать без ошибок. Вернемся к рис. 7.11. Обрабатывать ячейки, в которых записаны числа, не нужно. К моменту вычисления A4 значение A3 должно быть уже известно (а к моменту вычисления A5 нужно знать соответственно значения обеих ячеек). Зависимость между ячейками в столбцах B и C посложнее, а значит, и порядок их вычисления определить не так легко, не говоря уже о том, чтобы обнаружить в них контур.

Для аккуратной работы приложению — электронной таблице нужно завести орграф, в котором будут отражены зависимости ячеек друг от друга. Если пользователь заменяет в какой-то ячейке формулу числом, приложение должно удалить соответствующие дуги из графа. Если число заменяется формулой — добавить новые дуги, моделирующие зависимость текущей ячейки от упомянутых в формуле. Если формула заменяется на формулу — часть дуг надо удалить, часть добавить. Например, в таблице на рис. 7.11 можно ошибочно создать контур, если в ячейку B2 вписать формулу =C5. Это приведет к добавлению в орграф новой дуги C5 → B2, что породит несколько контуров; вот один из них: [B2, B4, B5, C5, B2].

Найти контур в орграфе поможет *обход вглубь*. Идея в том, чтобы аварийно останавливать обход, когда *в пространстве активного поиска* (плане разведки) внезапно оказывается *помеченный узел*, потому что это и есть признак контура. Когда `dfs(v)` возвращает историю обхода, мы можем быть уверены, что *все достижимые из* v *узлы помечены*, а v не присутствует в пространстве активного поиска.



Обходя орграф вглубь, мы можем наткнуться на помеченный узел, даже если это граф из трех узлов и контуров в нем нет. Если запустить dfs() с узла a, то обход вглубь встретит сначала b, а затем c, зайдя в тупик. Откатившись обратно к a, мы обратимся к следующему смежному узлу, c, а он уже помечен, хотя контуров в графе нет.



Поиск контура в графе — алгоритм, который, в отличие от других алгоритмов в этой главе, не начинает исследование с какого-то конкретного входного узла, потому что должен найти контур в любом месте графа. Как показано в примере 7.8, для этого придется исследовать многие (возможно, все) узлы графа. Словарь `marked` теперь хранит *состояние* узла: если `marked[v]` равно 1, узел помечен и присутствует в пространстве активного поиска (плане разведки), а если 2 — помечен, но в плане разведки его уже нет. При рекурсивном обходе вглубь пространством активного поиска (планом разведки) является стек контекстов вызовов, доступа к которому у нас нет. Поэтому информацию о том, есть ли `v` в одном из таких контекстов, мы будем хранить в словаре `marked`: если `marked[v]` равно 1, узел помечен и присутствует в плане разведки, а если 2 — помечен, но в плане разведки его уже нет.

Пример 7.8. Определение контура в орграфе путем обхода в глубину

```
def has_cycle(DG):
    marked = {}

    def dfs(v):                                  ❶
        marked[v] = 1                            ❷
        for w in DG[v]:
            state = marked.get(w, 0)             ❸
            if state == 1:                       ❹
                return True
            if state == 0 and dfs(w):           ❺
                return True
        marked[v] = 2                            ❻
        return False

    for v in DG.nodes():                         ❼
        if v not in marked and dfs(v):           ❽
            return True
    return False
```

❶ Функция `dfs(v)` обходит граф `DG` в глубину, начиная с узла `v`.

❷ Узел `v` помечен как посещенный и добавлен в план разведки (значение 1 в словаре `marked`).

❸ Состояние каждого узла `w`, смежного с данным `v`: 0 — не помечен, 1 — помечен и есть в плане разведки, 2 — помечен и уже разведан.

❹ Если смежный узел `w` помечен и при этом есть в плане разведки, то это по определению контур, можно сразу возвращать `True`.

❺ Если смежный узел `w` еще не помечен, рекурсивно вызываем `dfs(w)`, и, если там нашелся контур, возвращаем `True`.

❻ Если контур пока не найден, изменяем состояние узла `v` на «помечен, в плане разведки уже не присутствует» (значение `2`).

❼ **Обход в глубину** нужно проделать из каждого узла в нашем орграфе — мало ли, где найдется контур.

❽ Достаточно искать не из каждого узла, а только из каждого еще не помеченного. Если `dfs(v)` на таком узле `v` выдаст `True` — значит, найден контур и надо возвращать `True`.

С каждым рекурсивным вызовом `dfs()` граф исследуется все больше до тех пор, пока все узлы не окажутся помеченными — даже если они не являются вершинами дуг.

Определить состав самого контура тоже несложно — этому посвящено одно из тренировочных упражнений в конце главы, в котором требуется дописать `has_cycle()` таким образом, чтобы оно составляло и возвращало первый найденный в орграфе контур. На рис. 7.12 показан рекурсивный вызов `dfs()`. Каждый исследуемый узел сначала помечается единицей — в знак того, что он определен как посещенный, и при этом находится в *пространстве активного поиска*, — а на выходе из рекурсивного вызова узел помечается двойкой, это признак посещенного узла, которого уже нет в плане разведки. Пометки — это содержимое словаря `marked`, но на рисунке соответствующая пометка для наглядности проставлена прямо на узле и выделена фоном. История заканчивается тем, что в процессе обхода обнаруживается контур `[a, b, d, a]`: мы просматриваем узлы, смежные с `d`, и среди них оказывается узел `a`, помеченный двойкой, — а это значит, что он присутствует в плане разведки. Иными словами, двигаясь из узла `a` по дугам, мы снова попали в `a`, что в точности соответствует определению контура. Мы уже определили контур и возвращаем `True`, однако некоторые узлы все еще помечены двойкой, то есть остаются в (уже неактуальном) плане разведки, потому что рекурсивный вызов не доработал до конца.

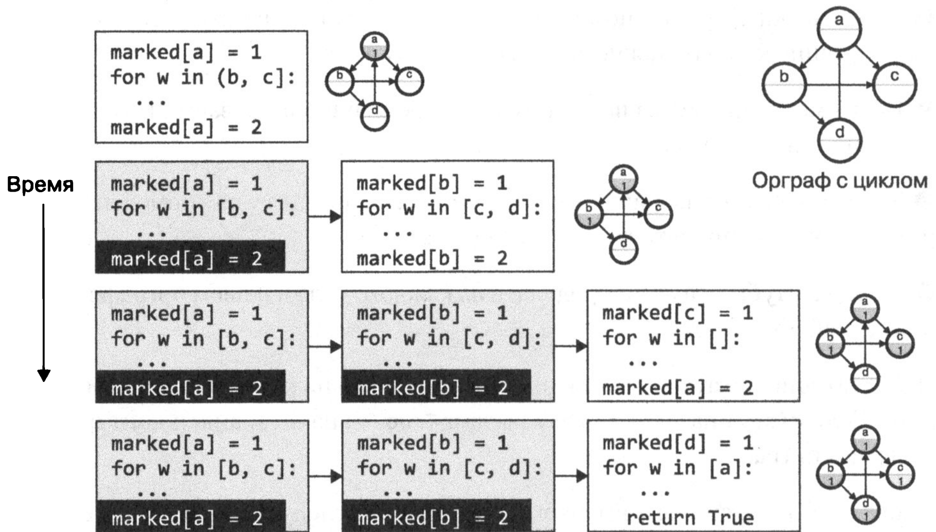


Рис. 7.12. Как найти контур, обходя орграф в глубину

Предположим, контуров в электронной таблице нет. Теперь важно определить порядок, в котором нужно вычислять зависящие друг от друга ячейки. В старой таблице с рис. 7.11 ячейки с константами (например, A1) вообще не надо вычислять и они не влияют на порядок. Ячейка B4 содержит формулу, в которой явно упомянуты B2 и B3, так что их надо посчитать раньше, чем B4. Вот один из возможных вариантов *линеаризации* зависимостей в таблице:

B2, C2, B3, C3, B4, C4, B5, C5, A2, A3, A4, A5

Такой порядок может быть результатом работы функции `topological_sort()`, приведенной в примере 7.9. **Топологическая сортировка** похожа на только что рассмотренный нами алгоритм **поиска контуров** — в нем тоже используется рекурсивная природа **обхода в глубину**. Всякий раз перед выходом `dfs(v)` из рекурсивного вызова, когда все достижимые из `v` узлы помечены, то есть `dfs()` уже обошла все узлы, «лежащие за `v`», можно добавить этот `v` в историю обхода как узел, все зависимости которого уже удовлетворены. Вычислять ячейки электронной таблицы следует с конца этой истории.

Пример 7.9. Топологическая сортировка ориентированного графа

```
def topological_sort(DG):
  marked, postorder = {}, []      ❶
```

```
  def dfs(v):                     ❷
    marked[v] = True              ❸
    for w in DG[v]:
      if not w in marked:
        dfs(w)                    ❹
    postorder.append(v)           ❺

  for v in DG.nodes():
    if not v in marked:           ❻
      dfs(v)

  return reversed(postorder)      ❼
```

❶ В списке postorder хранится история обхода — линеаризация узлов графа в обратном порядке.

❷ Функция dfs(v) выполняет обход графа DG в глубину начиная с узла v.

❸ В словаре marked отмечается, что узел уже посещен.

❹ Вызовем dfs(w) для всех непомеченных смежных с v узлов w.

❺ В этом месте алгоритма функция dfs(v) уже исследовала все узлы, которые (рекурсивно) зависят от v, — самое время добавить v в историю обхода.

❻ Запустим dfs(v) для каждого еще не помеченного узла v из графа DG. Замечу, что dfs() сам помечает вершины, так что каждый новый вызов обрабатывает отдельное подмножество узлов DG.

❼ История обхода содержит линеаризацию в обратном порядке, так что обратим ее и вернем.

В этой функции практически все повторяет **поиск контура**, кроме того, что вместо явного хранения плана разведки в ней ведется история обхода. Исследуя эту функцию уже известным нам методом, мы увидим, что каждый узел исследуется с помощью dfs() только раз, а внутренний оператор if выполняется по разу для каждой дуги графа. Добавление в список имеет константную сложность (см. табл. 6.1), что дает нам линейную оценку производительности **топологической сортировки** $O(N + E)$, где N — это количество узлов в графе, а E — количество дуг.

На рис. 7.13 видно, что, как только очередной вызов dfs() из примера 7.9 заканчивается, в списке postorder содержатся все узлы, зависимости которых уже удовлетворены; эти узлы перечислены в виде квадратов с пунктирными

границами. Под конец работы функции `topological_sort()` этот список — в обратном порядке — возвращается функцией. После того как приложение загрузило электронную таблицу, оно должно пересчитать содержимое ячеек — и порядок подсчета может указать **топологическая сортировка**.

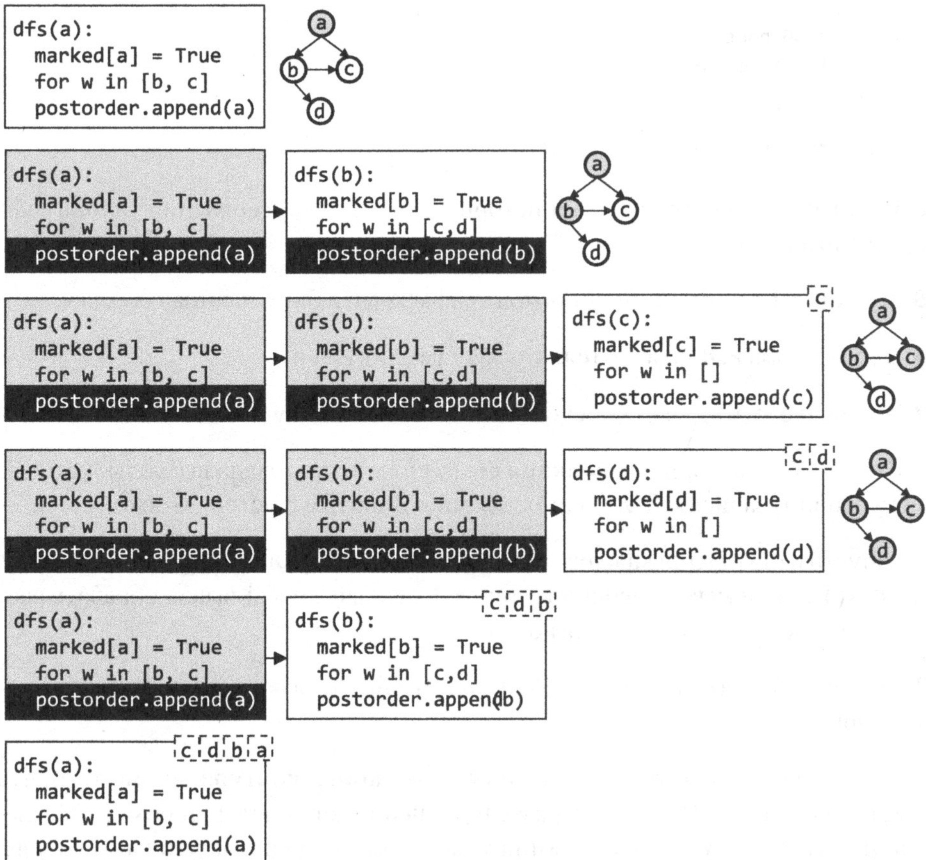
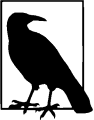


Рис. 7.13. Как работает топологическая сортировка путем обхода в глубину

Взвешенные графы

В некоторых предметных областях возникают задачи, решить которые можно с помощью графов, каждое ребро которых снабжено числовым значением — *весом* ребра. Веса можно приписывать и ребрам обычного графа, и дугам орграфа. Для простоты договоримся, что вес — это положительное число.



Стэнфордская большая коллекция сетевых данных (https://oreil.ly/hXqcg) содержит множество наборов данных, например, о связях в социальных сетях. В библиотеке TSPLIB (https://oreil.ly/MdMWm) собраны наборы данных для так называемой задачи коммивояжера (traveling salesman problem, TSP). Программисты десятилетиями изучают методы решения этой задачи, так что примеров накопилось немало. Большая коллекция графов есть на сайте «Карты путешествий» (https://oreil.ly/qWYsr). Отдельную благодарность хотелось бы выразить Джеймсу Тереско за любезно предоставленную таблицу дорог штата Массачусетс (https://oreil.ly/wlEy2).

По таблице дорожных участков штата Массачусетс можно сконструировать граф, в котором каждый узел — это *путевая точка* (остановка), однозначно определяемая координатами: парой чисел, широтой и долготой. Например, в таблице есть остановка на пересечении шоссе I-90 и I-93 в районе Бостона.

Широта этой точки (угол на север от экватора из центра Земли) равна 42.34642, а долгота (угол на запад от Гринвичского меридиана) составляет –71.060308. Ребро меду двумя узлами — это участок дороги, а вес ребра — длина пути между остановками в милях. Если соединить все остановки дорогами, получится схема, представленная на рис. 7.14.

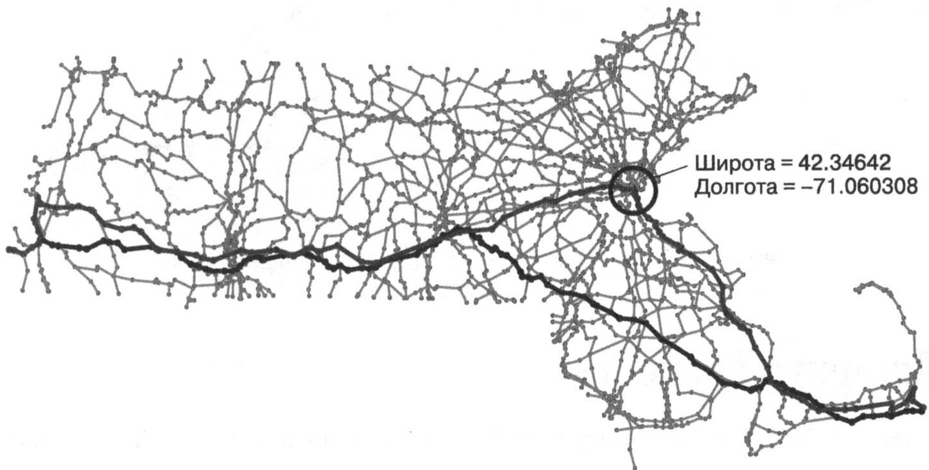


Рис. 7.14. Схема дорожной сети штата Массачусетс

Как теперь определить кратчайшее — в милях — расстояние от самой западной остановки (на границе со штатом Нью-Йорк) до крайней восточной точки штата — остановки на Тресковом мысе? Попробуем **обход в ширину**. Результат

показан на рисунке серой полужирной линией, его длина — 236.5 мили. Путь состоит из 99 ребер, проходит через ту самую остановку на перекрестке I-90/I-93 в Бостоне, и это самый короткий путь от старта до финиша, *если считать количество остановок* (или ребер графа). А вот будет ли этот путь кратчайшим, если измерять *пройденное расстояние*? Похоже, что нет.

Мы знаем, что **обход в глубину** не находит кратчайшего пути: на рис. 7.15 мы видим найденный с его помощью извилистый путь, он имеет длину аж 485.2 мили и содержит 267 ребер. **Направленный обход** (на рисунке не показан) сделал неправильный выбор где-то в самом начале пути и нашел путь из 141 ребра длиной 245.2 мили. А вот выделенный черным на рис. 7.14 кратчайший путь, в котором 210 ребер, но длина которого всего 210.1 мили, был найден с помощью **алгоритма Дейкстры** — им-то мы сейчас и займемся.

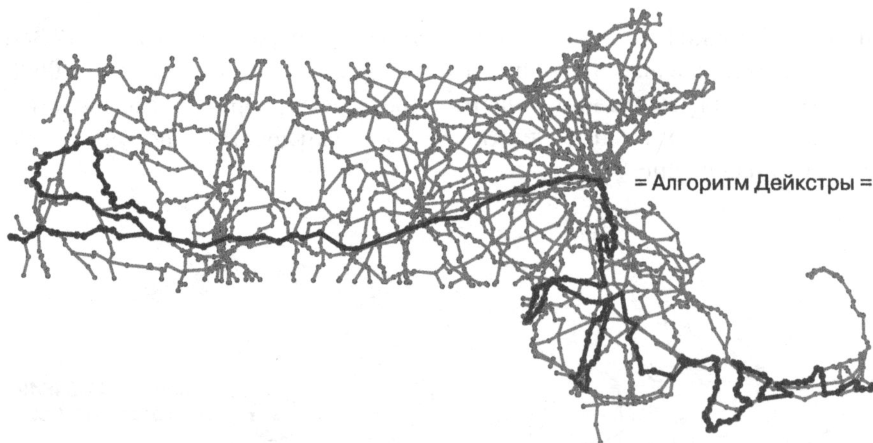


Рис. 7.15. Очень длинный путь, найденный при обходе в глубину

Алгоритм Дейкстры

Эдсгер Вибе Дейкстра — нидерландский ученый, один из создателей академического базиса компьютерной науки и отменный программист, автор нескольких уникальных алгоритмов, столь же впечатляющих, сколь и изящных. Алгоритм поиска *кратчайшего суммарного пути* от заданного узла ко всем достижимым узлам во взвешенном графе так и называется — **алгоритм Дейкстры**, а класс задач, им решаемых, — нахождение кратчайшего пути из заданной точки. Несколько позже мы рассмотрим пример, в котором по заданному графу (или орграфу)

с неотрицательными весами ребер[1] алгоритм Дейкстры заполняет два словаря: `dist_to[]`, где хранится кратчайшее суммарное расстояние от входного узла до данного, и `edge_to[]`, содержащий историю обхода.

На рис. 7.16 приведен простейший взвешенный орграф. Вес дуги `(a, b)` равен 6, дуги `(a, c)` — 10, а `(b, c)` — 2. Поэтому кратчайшим путем от `a` до `c` оказывается не прямой, длиной 10, а окольный — из двух дуг суммарной длиной 8.

Не в каждом графе, особенно ориентированном, можно построить путь между двумя любыми узлами. Кратчайшее расстояние от `b` до `c` равно 2, а вот кратчайшее расстояние от `c` до `b` — бесконечность, потому что вдоль имеющихся дуг путь от `c` до `b` проложить нельзя.

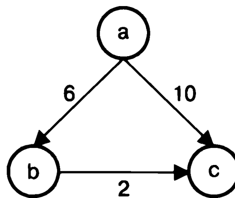


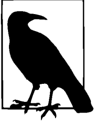
Рис. 7.16. Суммарная длина кратчайшего пути из a в c — 8

Для работы с графом **алгоритму Дейкстры** нужна особая структура данных — индексированная приоритетная очередь с порядком возрастания. Индексированная приоритетная очередь — это некоторое расширение функций приоритетной очереди, которую мы изучали в главе 4. В обеих структурах каждому хранимому значению ставится в соответствие приоритет. И там и там для хранения используется массив из *N* элементов — в нашем случае это количество узлов в графе. В отличие от очереди с порядком убывания, которую мы рассмотрели в главе 4, метод `.dequeue()` снимает из начала очереди элемент с *наименьшим приоритетом*.

Самое важное дополнение индексированной очереди — это метод `.decrease_priority(`*`значение, приоритет`*`)`. Он *понижает текущий приоритет **значения*** до более низкого ***приоритета***. В результате соответствующий элемент *продвигается ближе к началу очереди*, обгоняя другие элементы с бóльшими значениями приоритета. Для обычной приоритетной очереди, которую мы изучали ранее, такая

[1] Нулевые веса в алгоритме Дейкстры разрешены. Для учета весов с разными знаками нужно использовать другой алгоритм — Беллмана — Форда, его мы тоже разберем позже. — *Примеч. авт.*

операция была бы довольно неэффективной, потому что для поиска конкретного значения (которому надо понизить приоритет) пришлось бы просматривать всю очередь, затратив на это $O(N)$ действий.



Обычно в индексированной приоритетной очереди в качестве значений используются целые числа от 0 до N – 1. Приоритеты при этом хранятся в обычном массиве, а доступ к ним происходит просто по значению-индексу. Но в Python мы можем использовать словарь — тогда значения могут быть любыми константами.

Как видно из примера 7.10, класс `IndexedMinPQ` устроен и работает почти так же, как приоритетная очередь на основе кучи, которую мы исследовали в главе 4. Вместо того чтобы хранить объекты типа `Entry`, мы заведем два списка: `.values[n]` будет хранить значения элементов, а `.priorities[n]` — их приоритеты. Методы `.swim()` и `.sink()` такие же, как в очереди на основе кучи (подробнее см. в примерах 4.2 и 4.3), так что мы их не приводим. Главное отличие — словарь `.location`, который хранит информацию, обратную списку `values`: в `.location[v]` находится индекс значения элемента `v` в списке `.values` (и его приоритета в списке `.priorities`)[1]. Дополнительная информация помогает нам не искать значение по всей очереди, а получать его индекс за константное (с учетом возможного масштабирования) время $O(1)$ (о том, как при этом используется хеширование, мы говорили в главе 3).

Метод `.swap()` нужно доработать, потому что, помимо обмена местами двух элементов очереди, нужно менять местами соответствующие значения в списке `priorities` и обновлять содержимое словаря `location`, чтобы в `IndexedMinPQ` сохранялась способность быстрого поиска любого значения.

Пример 7.10. Основные свойства индексированной приоритетной очереди

```
class IndexedMinPQ:
  def less(self, i, j):
    return self.priorities[i] > self.priorities[j]                    ❶

  def swap(self, i, j):
    self.values[i], self.values[j] = self.values[j], self.values[i]      ❷
    self.priorities[i], self.priorities[j] = self.priorities[j], self.priorities[i]
```

[1] Стоит заметить, что в нашем варианте мы оставляем без внимания динамическую природу list: сразу создаем хранилище размером size + 1 и не пользуемся .pop()/.append(), что, как и в приоритетной очереди из главы 4, позволяет считать эти списки массивами. — *Примеч. пер.*

```
    self.location[self.values[i]] = i                          ❸
    self.location[self.values[j]] = j

  def __init__(self, size):
    self.N          = 0
    self.size       = size
    self.values     = [None] * (size+1)                       ❹
    self.priorities = [None] * (size+1)
    self.location   = {}                                       ❺

  def __contains__(self, v):                                   ❻
    return v in self.location

  def enqueue(self, v, p):
    self.N += 1
    self.values[self.N], self.priorities[self.N] = v, p       ❼
    self.location[v] = self.N                                  ❽
    self.swim(self.N)
```

❶ Поскольку мы строим очередь с порядком возрастания, более высокий приоритет (и положение ближе к началу очереди) будет у элемента `j` с *меньшим числовым значением приоритета*, чем у элемента `i`.

❷ В методе `.swap()` поменяем местами значения и приоритеты двух элементов, `i`-го и `j`-го.

❸ Там же в методе `.swap()` обновим соответствующие значения в словаре сохраненных индексов `location`.

❹ Для каждого из `N` элементов в массиве `values` будет под соответствующим индексом лежать значение, а в массиве `priorities` — приоритет.

❺ В словаре `location` хранятся индексы элементов очереди в массивах `values` и `priorities`.

❻ В отличие от обыкновенной приоритетной очереди, индексированная позволяет проверить, есть ли в ней элемент `v`, за (в среднем) константное время O(1) путем поиска в словаре `location`.

❼ Еще одно небольшое исправление — в методе `.enqueue()`. Чтобы добавить пару `(v, p)` в очередь, `v` записывается в `values[N]`, а `p` — в `priorities[N]`, где `N` — индекс очередной свободной ячейки.

❽ Также в методе `.enqueue()` нужно запомнить индекс `v` и только после этого выполнить `.swim()`, который восстановит порядок очереди, если тот нарушился.

Как это и полагается в куче, метод `.enqueue()` сначала добавляет значение v и его приоритет p в конец массивов `values[]` и `priorities[]` соответственно. Кроме того, в IndexedMinPQ заявлен быстрый поиск элемента, и в словарь `location` нужно записать, что значение v лежит под индексом N (напомню, что в куче для простоты вычислений массивы индексируются с единицы). Затем вызывается метод `.swim()`, который восстанавливает возможно утраченный порядок IndexedMinPQ.

Итак, словарь `location` нужен для быстрого поиска индекса любого хранящегося в очереди значения. Метод `.decrease_priority()`, представленный в примере 7.11, переставляет данное значение *ближе к началу очереди* IndexedMinPQ путем замены приоритета. Единственное требование: новый приоритет может быть только *меньше* исходного (как мы помним, это означает более высокий приоритет), после чего элемент всплывает на свое место.

Пример 7.11. Уменьшение приоритета для некоторого значения из IndexedMinPQ

```
def decrease_priority(self, v, lower_priority):
  idx = self.location[v]                            ❶
  if lower_priority >= self.priorities[idx]:        ❷
     raise RuntimeError('Wrong priority')

  self.priorities[idx] = lower_priority             ❸
  self.swim(idx)                                    ❹
```

❶ Найдем индекс элемента v в хранилище.

❷ Если `lower_priority` в действительности не меньше текущего приоритета `priorities[idx]`, сгенерируем исключение `RuntimeError`.

❸ Уменьшим приоритет v.

❹ При необходимости восстановим порядок в очереди, позволив элементу всплыть на свое место.

Метод `.dequeue()` удаляет значение с наименьшим (то есть наивысшим) приоритетом из начала очереди. Реализация этого метода в IndexedMinPQ несколько сложнее уже виденной нами, потому что дополнительно надо обновлять словарь `location`, — как показано в примере 7.12.

Пример 7.12. Удаление элемента с наивысшим приоритетом из IndexedMinPQ

```
def dequeue(self):
  min_value = self.values[1]                              ❶

  self.values[1] = self.values[self.N]                    ❷
  self.priorities[1] = self.priorities[self.N]
  self.location[self.values[1]] = 1
```

```
self.values[self.N] = self.priorities[self.N] = None   ❸
del self.location[min_value]                            ❹

self.N -= 1                                             ❺
self.sink(1)
return min_value                                        ❻
```

❶ Запомним min_value — значение с наивысшим приоритетом.

❷ Перенесем последнее в хранилище значение (с индексом N) в его начало (с индексом 1) и обновим location, хранящий индексы в соответствии со значениями.

❸ Заметем в хранилище следы существования удаленного min_value.

❹ Удалим также индекс min_value в словаре location.

❺ Уменьшим на единицу количество элементов в очереди, а затем вызовем .sink(1), дабы восстановить порядок.

❻ Вернем значение, приоритет которого был наивысшим (у этого значения была наименьшая величина приоритета).

В структуре данных IndexedMinPQ постоянно поддерживается следующее свойство: если значение v есть в приоритетной очереди, то location[v] содержит индекс idx соответствующего элемента в хранилище, то есть values[idx] — это v, а priorities[idx] — его приоритет, p.

Индексированная приоритетная очередь используется в **алгоритме Дейкстры** для вычисления длины наименьшего пути от заданного входного узла src до любого узла орграфа. В процессе обхода составляется словарь dist_to[v], в котором хранится минимальное вычисленное на данный момент расстояние от src до любого узла v в графе. Если узел недостижим из src, в словаре хранится некоторое актуально бесконечное значение. По мере продвижения обхода мы ищем такие пары u и v, дуга между которыми имеет вес wt, что dist_to[u] + wt < dist_to[v] — иными словами, расстояние от src до v больше, чем суммарное расстояние от src до u, а оттуда — до v по дуге (u, v).

Последовательный просмотр узлов в поисках таких укорачивающих расстояние дуг делает **алгоритм Дейкстры** похожим на **обход в ширину**, где порядок вершин в очереди просмотра зависит от количества шагов до входа. Результаты уже проделанного поиска хранятся в dist_to[v], а остальные узлы добавляются в индексированную очередь сообразно с приоритетом, который равен текущему удалению узла от src. Сначала известно только, что dist_to[src] — это 0 (потому что src и есть вход), а остальные расстояния бесконечны. В очереди IndexedMinPQ в это время находятся все узлы, и у всех у них приоритет бесконечен, кроме нулевого приоритета входного узла src.

Нет необходимости помечать в **алгоритме Дейкстры** узлы как посещенные, потому что в приоритетной очереди всегда находятся только неисследованные узлы. В процессе обхода из очереди поочередно удаляется по одному узлу, расстояние до которого от входа наименьшее (пример 7.13).

Пример 7.13. Алгоритм Дейкстры находит кратчайшие расстояния до всех узлов графа из заданного узла

```
from math import inf

def dijkstra_sp(G, src):
  N = G.number_of_nodes()

  dist_to = {v:inf for v in G.nodes()}          ❶
  dist_to[src] = 0

  impq = IndexedMinPQ(N)                        ❷
  for v, dist in dist_to.items():
    impq.enqueue(v, dist)

  def relax(e):
    n, v, weight = e[0], e[1], e[2][WEIGHT]     ❺
    if dist_to[n] + weight < dist_to[v]:        ❻
      dist_to[v] = dist_to[n] + weight          ❼
      edge_to[v] = e                            ❽
      impq.decrease_priority(v, dist_to[v])     ❾

  edge_to = {}                                  ❸
  while not impq.is_empty():
    n = impq.dequeue()                          ❹
    for e in G.edges(n, data=True):
      relax(e)

  return (dist_to, edge_to)
```

❶ В словаре `dist_to` расстояние до каждого узла равно `inf`[1], а расстояние до `src` — нулю.

❷ Поставим все узлы в очередь `impq` с соответствующими приоритетами.

❸ Каждый элемент `edge_to[v]` будет хранить дугу, которая кратчайшим путем привела к `v`.

❹ Возьмем из очереди узел `n`, путь до которого от `src` — кратчайший. Просмотрим все исходящие из него дуги `e` вида `(n, v, вес)` (чтобы получить дуги

[1] В стандартном модуле math есть такое удобное значение, которое и правда больше любого допустимого числа Python. — *Примеч. пер.*

с весами, надо передать методу `networkx.Graph.edges()` дополнительный параметр `data=True`). Для каждого `v` проверим, не короче ли путь до него через `n` нынешнего кратчайшего пути.

❺ Дуга — это последовательность вида *узел* `n`, *узел* `v`, *данные*, где `(n, v)` — это собственно дуга, а ее вес — одна из составляющих объекта *данные*.

❻ Если путь от `src` до `v` оказался длиннее, чем суммарный путь от `src` до `n` и далее от `n` до `v`, значит, мы нашли новый кратчайший путь.

❼ Обновим информацию о кратчайшем пути до `v`.

❽ Запишем дугу `e` (от `n` до `v`) в `edge_to[v]`: именно она привела нас в `v` кратчайшим путем.

❾ Самый важный момент: уменьшим значение приоритета в `impq` до длины нового кратчайшего пути. При этом в цикле `while` будем продолжать выбирать узел с наименьшим вычисленным расстоянием от `src`.

На рис. 7.17 показаны первые три шага цикла `while` из примера 7.13. Каждый узел `n` хранится в `IndexedMinPQ` с приоритетом, равным наименьшему вычисленному расстоянию `dist_to[n]` (оно показано в небольшой пунктирной рамке рядом с каждым узлом). На очередном проходе `while` мы удаляем узел `n` из `impq` и проверяем, нет ли более короткого пути из `src` в некоторый узел `v` через `n`. Назовем этот процесс *сокращением контура*. Поскольку кратчайшее расстояние служит приоритетом `IndexedMinPQ`, мы можем быть уверены, что в **алгоритме Дейкстры** *путь до каждого снятого из начала очереди* `impq` *элемента действительно кратчайший*.

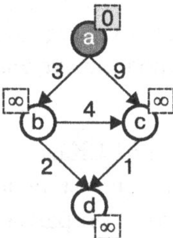
Что следует учитывать при оценке быстродействия **алгоритма Дейкстры**?

- *Время постановки в очередь всех узлов.* Сначала в очередь добавляются все N узлов: `src` с приоритетом 0 и $N - 1$ оставшихся узлов с бесконечным приоритетом. При добавлении узлов с бесконечным приоритетом `.swim()` не делает ничего, потому что условие порядка и так выполнено, и единственный раз при добавлении `src` узел с приоритетом 0 всплывает в начало очереди не более чем за логарифмическое время. Быстродействие этого шага, следовательно, $O(N)$.
- *Время выборки всех узлов из очереди.* В процессе обхода элементы снимаются из начала очереди по одному. Мы помним, что `impq` — это двоичная куча и сложность операции `.dequeue()` в ней логарифмическая ($O(\log N)$), стало быть, сложность снятия всех N элементов можно оценить как $O(N \log N)$.
- *Время перебора всех дуг графа.* Быстродействие перебора всех ребер графа зависит от того, как они в нем смоделированы. Если с помощью матрицы

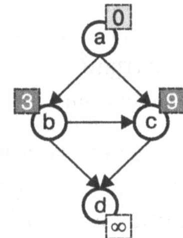
смежности — на полный перебор потребуется порядка $O(N^2)$ действий. Если с помощью списка смежности — то порядка $O(N + E)$.

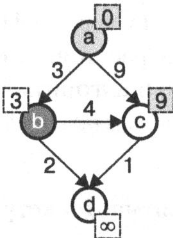
- *Время сокращения контуров*. Функция `relax()` вызывается для каждой дуги графа. Если при этом вычисленная длина кратчайшего пути уменьшилась, в вызове `relax()` может встретиться вызов `.decrease_priority()`. Этот метод, в свою очередь, вызывает метод двоичной кучи `.swim()`, сложность которого — $O(\log N)$. Следовательно, общее время, затраченное на вызов `relax()`, для каждой дуги будет $O(E \log N)$, где E — количество дуг в графе.

Состояние обхода в dist_to	Шаг алгоритма Дейкстры	Новое состояние dist_to

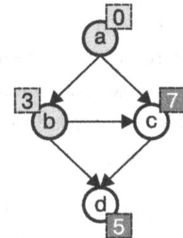


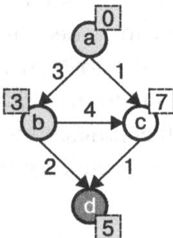

```
# impq содержит[a,b,c,d]
a = impq.dequeue()
for  e  in [(a,b), (a,c)]:
    relax(e)
```






```
# impq содержит[b,d,c]
c = impq.dequeue()
for  e  in [(b,c), (b,d)]:
    relax(e)
```






```
# impq содержит[d,c]
d = impq.dequeue()
for  e  in [(d,c)]:
    relax(e)
```

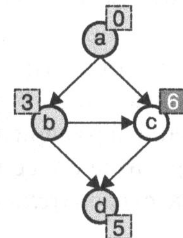


Рис. 7.17. Обход небольшого графа алгоритмом Дейкстры

Если хранить дуги в списке смежности, **алгоритм Дейкстры** имеет сложность порядка $O((E + N) \log N)$, а если в матрице смежности — порядка $O(N^2)$. С ро-

стом графа использование матрицы смежности довольно быстро становится неэффективным.

В процессе обхода **алгоритмом Дейкстры** создаются два словаря: dist_to[v], в котором записан суммарный вес — кратчайший вычисленный путь от src до каждого v, и edge_to[v], в котором хранится последняя на кратчайшем пути из src в v дуга (u, v). Из edge_to сравнительно несложно восстановить полный путь от src до произвольного v. Механизм восстановления очень похож на тот, что мы видели в примере 7.3, но edge_to[], как это показано в примере 7.14, нужно проходить от конца к началу.

Пример 7.14. Восстановление пути из edge_to[]

```
def edges_path_to(edge_to, src, target):         ❶
  if not target in edge_to:
    raise ValueError('Unreachable')              ❼

  path = []
  v = target                                     ❷
  while v != src:
    path.append(v)                               ❸
    v = edge_to[v][0]                            ❹

  path.append(src)                               ❺
  path.reverse()                                 ❻
  return path
```

❶ Восстановим путь от src до произвольного target с помощью edge_to.

❷ Начнем с конца пути, target.

❸ Пока мы не добрались до входного узла src, добавим v в список path, в котором в обратном порядке, начиная с target, строится путь от src до target.

❹ Нулевой элемент дуги (u, v), хранящейся в edge_to[v], — это u, предыдущий узел на пути. Подставим его на место v.

❺ Мы вышли из цикла while, когда добрались до src, так что его надо добавить в путь отдельно.

❻ Переставляем элементы path в обратном порядке — теперь там содержится путь от src до target.

❼ Если узел target отсутствует в edge_to[], значит, из src он недостижим.

Алгоритм Дейкстры работает *только с неотрицательными весами*. Вес может быть отрицательным, например, если граф моделирует финансовые трансакции,

которые предусматривают и платежи, и возврат денег. С отрицательным весом дуги **алгоритм Дейкстры**, как мы видим на рис. 7.18, может и не справиться.

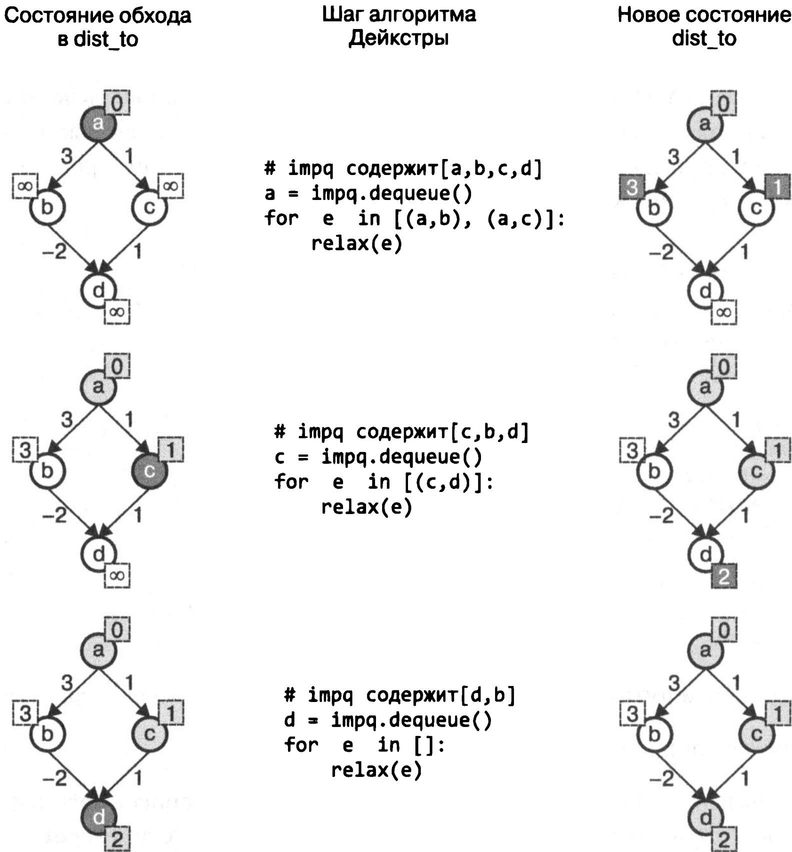


Рис. 7.18. Неудачно присвоенный отрицательный вес дуги нарушает работу алгоритма Дейкстры

На рис. 7.18 мы видим, как **алгоритм Дейкстры** обрабатывает три узла из `impq`. Остается узел `b`. Кратчайший путь от `a` до `d` уже вычислен, и на третьем шаге мы снимаем `d` из `impq`. Поэтому, когда на следующем шаге (которого на иллюстрации уже нет) мы снимем `b` из `impq` и сократим дугу `(b, d)`, обновить внезапно образовавшееся кратчайшее расстояние до `d` уже не догадаемся: этого узла нет в плане разведки. Очевидно, проблема с отрицательными весами в том, что уже пройденный путь может *уменьшиться* после добавления в него еще одной дуги, но **алгоритм Дейкстры** не умеет возвращаться и пересматривать уже вычисленные кратчайшие пути.

Алгоритм Дейкстры может не отследить дугу с отрицательным весом, так как мы предполагаем, что при добавлении очередной дуги в некоторый путь его длина не уменьшается. **Алгоритм Беллмана – Форда** лишен этого недостатка, он может находить кратчайшие пути из входного узла в графе до произвольного, даже если веса ребер имеют разные знаки. Но есть и исключение: если в графе присутствует *отрицательный контур* (то есть контур, суммарный вес которого отрицателен), любой путь, включающий вершину этого контура, можно сделать меньше, чем любое наперед взятое число, – достаточно пройтись по нему несколько раз! На рис. 7.19 показаны два графа, в каждом из которых по две положительные и две отрицательные дуги. В том, что слева, нет отрицательного контура: начнем обход с a и сперва пройдем единственную дугу (a, b) с весом 1. Если потом пройтись один раз по контуру b → d → c → b (суммарный вес которого 1), общая длина a → b → d → c → b будет равна 2, так что кратчайшая дистанция от a останется равной 1. А вот в графе справа присутствует *отрицательный контур* b → d → c → b, суммарная длина которого равна –2. Таким образом, мы можем сделать путь от a до b равным любому нечетному отрицательному числу, просто пройдясь по этому контуру соответствующее число раз. Например, вес пути a → b → d → c → b → d → c → b равен –3.

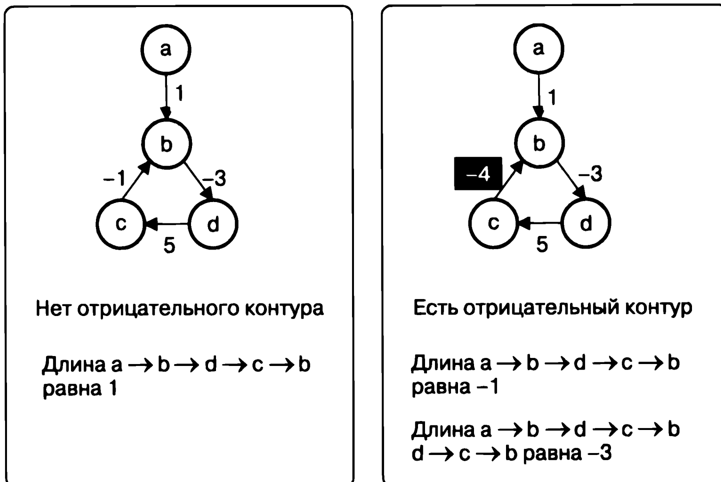


Рис. 7.19. Два графа с отрицательным весом дуг, только один из которых содержит отрицательный контур

Алгоритм Беллмана – Форда находит кратчайшее расстояние от входного узла до любого другого, применяя совсем другой подход, в котором годятся и отрицательные веса дуг. Реализация этого алгоритма приведена в примере 7.15,

многое в ней взято из **алгоритма Дейкстры**. К счастью, нам не надо сначала проверять граф на наличие отрицательных контуров (как это приходилось делать перед **топологической сортировкой**): в процессе обхода графа **алгоритмом Беллмана — Форда** отрицательный контур определится сам — в этом случае порождается исключение.

Пример 7.15. Реализация алгоритма Беллмана — Форда

```
from math import inf

def bellman_ford(G, src):
  dist_to = {v: inf for v in G.nodes()}          ❶
  dist_to[src] = 0
  edge_to = {}                                    ❷

  def relax(e):
    u, v, weight = e[0], e[1], e[2][WEIGHT]
    if dist_to[u] + weight < dist_to[v]:          ❺
      dist_to[v] = dist_to[u] + weight            ❻
      edge_to[v] = e                              ❼
      return True                                 ❽
    return False

  for i in range(G.number_of_nodes()):            ❸
    for e in G.edges(data=True):                  ❹
      if relax(e):
        if i == G.number_of_nodes() - 1:          ❾
          raise RuntimeError('Negative Cycle exists in graph.')

  return (dist_to, edge_to)
```

❶ В словаре `dist_to` будет храниться текущий кратчайший путь. Поначалу там бесконечность для всех узлов, а для `src` — ноль.

❷ В `edge_to[v]` хранится дуга, по которой поиск привел нас в `v`.

❸ Цикл на *N* проходов (по количеству узлов графа).

❹ Цикл по всем дугам графа `(u, v)`. Используем ту же функцию `relax()`, что и в **алгоритме Дейкстры**, — сокращаем вес пути от `src` до `v`, если суммарный вес аналогичного пути через `u` оказывается меньше.

❺ Если путь до `u` и вес дуги в сумме меньше текущего значения кратчайшего пути до `v`, мы нашли более короткий путь до `v`.

❻ Запишем это значение на место старого.

❼ Запомним дугу, которая привела нас в `v` кратчайшим путем.

❽ Если `relax()` вернул `True`, мы нашли более короткий путь до `v`.

❾ В **алгоритме Беллмана — Форда** мы N раз перебираем все E дуг. Если на последнем проходе выясняется, что какая-то дуга `e` все еще укорачивает какой-то путь, значит, мы нашли отрицательный контур.

Почему этот алгоритм работает? Просто потому, что в графе из N узлов в самом длинном пути не может быть больше $N - 1$ узлов. Так что если в цикле `for` переменная `i` прошла $N - 1$ итераций, попытавшись сократить все дуги графа, то даже если единственный путь состоял из всех дуг одновременно, мы и его уже проверили. Значит, больше не должно остаться дуг, позволяющих сократить уже отмеренное расстояние. Поэтому мы продлили цикл еще на один проход: если и на N-м шаге можно сократить какую-то дугу, значит, происходит это посредством отрицательного контура.

Полный поиск кратчайших путей

Пока обход графа в алгоритмах из этой главы *начинался из заранее заданного узла*. Полный поиск кратчайших путей должен выдать информацию о наименьшем суммарном весе дуг на пути из любого узла в графе до любого другого. В неориентированном графе кратчайшее расстояние от любого узла `u` до любого другого `v` совпадает с кратчайшим расстоянием от `v` до `u`. В графе любого типа какой-то узел `v` может быть недостижим из какого-то узла `u`, в этом случае расстояние считается бесконечным. В орграфе кратчайшие пути от `u` до `v` и от `v` до `u` могут не совпадать или `u` может быть недостижим из `v`, даже если `v` достижим из `u`.

Идея подсчитать все возможные кратчайшие расстояния от произвольного `u` до произвольного `v` кажется запредельно трудоемкой. В самом деле, даже в простейшем графе с рис. 7.20 на поиск кратчайшего расстояния между одними только `d` и `c` нужно порядочно времени, сколько же его уйдет для исследования всех пар узлов! Между `d` и `c` есть дуга весом 7, но путь `d` → `b` → `a` → `c` оказывается короче: его вес равен 6.

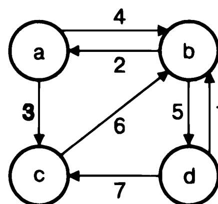


Рис. 7.20. Граф, на котором мы будем вычислять кратчайшие пути

Прежде чем мы углубимся в изучение алгоритма, который решает эту задачу, стоит заранее договориться, что он будет возвращать в качестве результата. Воспользуемся структурой данных из **алгоритма Дейкстры** и его предшественников и слегка доработаем ее. Всякий раз, когда в предыдущих алгоритмах шла речь о пути из заданного узла `src` к произвольному `v`, мы использовали словарь с индексом `v`. Теперь же, поскольку речь зайдет о пути от произвольного `u` к произвольному `v`, индексом словаря станет пара — кортеж `(u, v)`.

1. В словаре `dist_to[u, v]` будем хранить вычисленный кратчайший путь между двумя узлами, `u` и `v`. Пока путь из `u` в `v` не подсчитан, `dist_to[u, v]` бесконечно.
2. В словаре `node_from[u, v]` будем записывать последний узел на вычисленном кратчайшем пути от `u` до `v`. Это поможет нам восстановить весь кратчайший путь от `u` до `v`.

Что предлагается делать.

Начнем с того, что занесем в `dist_to[u, v]` вес дуги `(u, v)`, если такая есть, а если нет — бесконечность для разных `u` и `v` и ноль, когда `u` и `v` совпадают. Для каждой дуги `(u, v)` запишем в `node_from[u, v]` информацию о том, что это последняя (она же первая) дуга на кратчайшем пути из `u` в `v`, и, стало быть, в `v` этим путем мы попали из `u`. На рис. 7.21 показаны словари `node_from` и `dist_to` в виде квадратных таблиц: по вертикали идет первый узел пары, по горизонтали — второй. Для заполнения таблиц мы использовали граф с рис. 7.16.

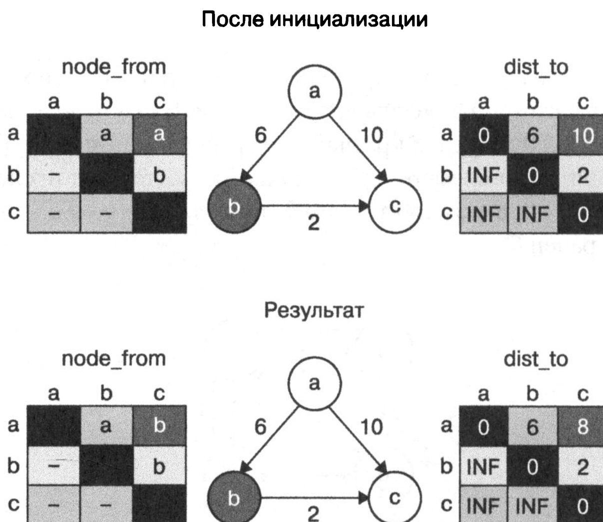


Рис. 7.21. Что хранится в node_from и dist_to при полном поиске кратчайших путей

Теперь выберем узел b в качестве t и проверим, можно ли найти два таких узла, u и v, чтобы сумма расстояний от u до t и от t до v была меньше вычисленного кратчайшего пути dist_to[u, v]. В нашем небольшом примере dist_to[a, b] равно 6, a dist_to[b, c] равно 2, так что кратчайший путь от a до c лежит именно через b — теперь он равен 8. Вдобавок нужно обновить node_from[a, c], потому что предыдущий узел на кратчайшем пути теперь b. Принцип тот же, что и в **алгоритме Дейкстры**, когда мы сокращаем контур.

Как мы видим, словарь node_from[u, v] выполняет ту же функцию, что и аналогичный node_from[] в предыдущих алгоритмах: хранит последний узел на кратчайшем пути до v, только путь задается не один, от входного узла, а много — от любого узла в графе.

Мы поочередно просматриваем все узлы в цикле с переменной t и перебираем все пары узлов u и v в надежде на то, что расстояние между ними можно сократить с помощью этого t (как мы это сделали двумя абзацами раньше). Как только видим, что dist_to[u, t] + dist_to[y, v] меньше dist_to[u, v], сокращаем значение dist_to[u, v] до этой суммы, а заодно обновляем node_from[u, v] — теперь там должен быть узел на пути от t, node_from[t, v].

Иными словами, поскольку мы уже вычислили dist_to[t, v], мы знаем, что node_from[t, v] содержит последний узел на кратчайшем пути от t к v, а значит, он будет последним и на новом кратчайшем пути от u к v, *который идет через* t, и следует приравнять node_from[u, v] к node_from[t, v]. Когда мы будем восстанавливать полный путь, мы пройдем вспять от v к этому t, а в node_from[u, t] будет лежать предыдущий узел на кратчайшем пути от u.

Все наши соображения выглядят не слишком простыми, потому что в самом алгоритме нет выделенной части, которая, скажем, вычисляет кратчайший путь от конкретного u к конкретному v. Мы храним не всю информацию: вычисляемые в процессе работы пути не обязаны быть кратчайшими — мы можем пересчитать их позже. Возьмем граф с рис. 7.20 и заполним наши структуры его информацией о кратчайших путях; результат изображен на рис. 7.22. В таблице dist_to записаны кратчайшие пути между двумя узлами, ее координаты соответствуют первому и второму узлам в паре. Например, в строке, помеченной как a, хранятся кратчайшие дистанции от узла a до всех остальных узлов графа. В частности, dist_to[a, c] равно 3, потому что кратчайший путь от a до c — это a → c, то есть единственная дуга (a, c) весом 3. Кратчайший путь от a до d, dist_to[a, d], — это a → b → d, с суммарным весом 9.

Решение задачи полного поиска кратчайших путей — словари node_from и dist_to. На рис. 7.22 приведены сами эти пути: по вертикали — u, по горизонтали — v, на пересечении — полный путь от u до v. Становится понятнее, откуда такие

значения в `dist_to`, при этом *предпоследний узел в полном кратчайшем пути от* `u` *до* `v` выделен серым. Нетрудно заметить, что выделенный узел соответствует `node_from[u, v]`.

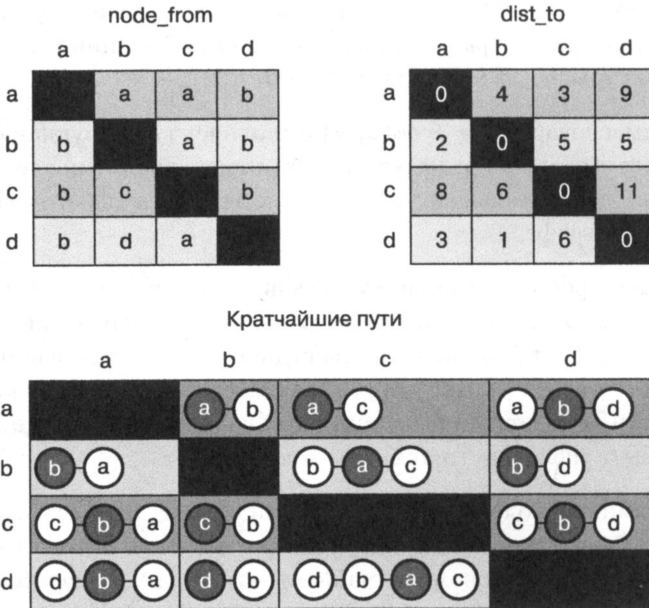


Рис. 7.22. Какими получаются dist_to, node_from и полные кратчайшие пути для графа с рис. 7.20

Рассмотрим кратчайший путь от `d` до `c`: `d` → `b` → `a` → `c`. Он состоит из пути от `d` до `a` и финальной дуги `(a, c)`. Таким образом наши два словаря подсказывают *рекурсивное решение задачи*: последний узел на пути из `d` в `c` — это `a`, который как раз и хранится в `node_from[d, c]`.

Продолжая в том же духе, обнаруживаем, что кратчайший путь от `d` к `a` идет через `b`, и именно `b` записан в `node_from[d, a]`.

Алгоритм Флойда — Уоршелла

Задачу полного поиска кратчайших путей и идею ее решения мы обсудили, теперь рассмотрим формализацию этой идеи: **алгоритм Флойда — Уоршелла**. Напомню: идея была в том, чтобы найти такие три узла u, v и t, чтобы окольный путь из u в v через t был короче прямого.

В примере 7.16 **алгоритм Флойда — Уоршелла** для начала заполняет `node_from[]` и `dist_to[]` данными из списка дуг — это часть исходного графа. На рис. 7.23 представлен результат инициализации обоих словарей.

node_from

	a	b	c	d
a	–	**a**	a	–
b	b	–	–	**b**
c	–	–	**c**	–
d	–	**d**	**d**	–

dist_to

	a	b	c	d
a	0	4	3	INF
b	2	0	INF	**5**
c	INF	6	0	INF
d	INF	1	7	0

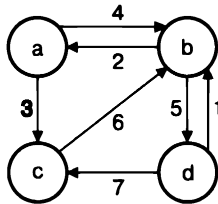


Рис. 7.23. Инициализация dist_to[] и node_from[] данными из графа G

В `node_from[u, v]` записано либо `None` (на рисунке — прочерк), либо `u`. В `dist_to[u, v]` лежит либо 0, если `u` — это `v`, либо вес дуги `(u, v)`, если такая есть, а если ее нет, то бесконечность (на рисунке — `INF`)[1].

Пример 7.16. Алгоритм Флойда — Уоршелла

```
from math import inf

def floyd_warshall(G):
    dist_to = {(u, v): 0 if u is v else inf for u in G.nodes() for v in
G.nodes()}
    node_from = {(u, v): None for u in G.nodes() for v in G.nodes()}     ❶
    for u, v, data in G.edges(data=True):                                 ❷
        dist_to[u, v] = data[WEIGHT]                                      ❸
        node_from[u, v] = u                                               ❹
```

[1] В тексте функции мы, как и раньше, используем inf из математической библиотеки Python. — *Примеч. пер.*

```
    for t in G.nodes():
        for u in G.nodes():
            for v in G.nodes():
                new_len = dist_to[u, t] + dist_to[t, v]              ❺
                if new_len < dist_to[u, v]:
                    dist_to[u, v] = new_len                          ❻
                    node_from[u, v] = node_from[t, v]

    return (dist_to, node_from)                                      ❼
```

❶ Создадим словари по всем возможным парам узлов. Заполняем `dist_to` нулями для путей из узла в тот же узел и `inf` для путей между разными узлами, потому что еще не знаем, есть ли эти пути. По той же причине заполняем `node_from` значением `None`.

❷ Теперь добавим информацию из дуг графа. В цикле используется представление дуги в виде кортежа из двух вершин и массива с данными.

❸ Изначальная длина пути из `u` в `v` — это вес дуги `(u, v)`.

❹ Поскольку пока кратчайший путь из `u` в `v` совпадает с дугой `(u, v)`, последний узел на пути к `v` — это `u`.

❺ Вычислим длину обходного пути `u` → `t` → `v`.

❻ Если обходной путь короче предыдущей вычисленной длины, запомним новую длину `dist_to[u, v]` и обновим последний узел на пути от `u` к `v`, теперь `node_from[u, v]` совпадает с `node_from[t, v]`.

❼ По вычисленным словарям `dist_to` и `node_from` можно восстановить кратчайший путь между любыми узлами. Вернем эти словари.

Сам алгоритм оказался на удивление компактным. Сначала мы заполняем `node_from` и `dist_to` в предположении, что никакой узел не достижим ниоткуда, кроме себя самого. Затем добавляем в них сведения о дугах: вес пути и начальную вершину. Внешний цикл по `t` перебирает узлы, через которые мы будем пытаться строить обходные пути в паре узлов, два внутренних перебирают узлы этой пары — `u` и `v`. Мы проверяем, не короче ли путь `u` → `t` → `v` текущего пути `u` → `v`. По мере перебора узлов мы рассмотрим все улучшения и в конце концов найдем настоящий ответ.

Скажем, `t` — это `a`. Тогда во внутренних циклах по `u` и `v` в какой-то момент окажется, что `u = b`, а `v = c`. Проверяя, не короче ли путь от `b` к `c` через `a`, мы обнаружим, что `dist_to[b, c]` на данный момент бесконечно (см. рис. 7.23), а `dist_to[b, a] = 2` и `dist_to[a, c] = 3`, и 5 явно короче бесконечности. Обновляем

`dist_to[b, c]`, а заодно записываем `a` в `node_from[b, c]` в знак того, что в только что найденном коротком пути от `b` к `c` предпоследний узел — это `a`.

Еще один способ понять работу алгоритма — посмотреть, как именно меняются значения `dist_to[u, v]`. До первого прохода внешним циклом `dist_to` содержит *только прямые пути из* `u` *в* `v`, без возможности пойти в обход. Допустим, на первом проходе `t` стало равно `a`. Тогда под конец этого прохода в `dist_to[u, v]` будет лежать длина кратчайшего пути из `u` в `v`, в котором *может присутствовать* `a`. Например, найдется более короткий путь `b` → `a` → `c` из `b` до `c`.

На втором проходе `t` станет равно `b` и мы найдем все случаи, когда уже имеющийся путь длиннее, чем возможный путь через `b`. Например, кратчайший путь между `d` и `c` был длиной 7 — по длине соответствующей дуги в графе. Но теперь мы посчитали обходной путь через `b`: сначала от `d` к `b` (длиной 1), затем от `b` к `c` (длиной 5), и он оказался короче — равен 6. На рис. 7.24 приведен результат работы **алгоритма Флойда — Уоршелла** после прохода циклом первых двух узлов — `a` и `b`.

k = a

node_from

	a	b	c	d
a	–	a	a	–
b	b	–	a	b
c	–	c	–	–
d	–	d	d	–

dist_to

	a	b	c	d
a	0	4	3	INF
b	2	0	5	5
c	INF	6	0	INF
d	INF	1	7	0

k = b

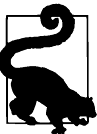
node_from

	a	b	c	d
a	–	a	a	b
b	b	–	a	b
c	b	c	–	b
d	b	d	a	–

dist_to

	a	b	c	d
a	0	4	3	9
b	2	0	5	5
c	8	6	0	11
d	3	1	6	0

Рис. 7.24. Как изменились node_from и dist_to после проверки a и b



Интересно, а почему мы не проверяем, равны ли t, u и v друг другу? Во-первых, dist_to[u, u] уже равно нулю, так что на результат эта проверка не повлияет. Во-вторых, заметного прироста производительности она не даст, а вот читать и понимать программу без нее легче.

В результате **алгоритм Флойда — Уоршелла** не потребовал каких-то особенных структур данных и просто последовательно перебрал все N_3 троек `(t, u, v)` из узлов графа.

1. Во время инициализации `node_from` и `dist_to` запоминаются все кратчайшие пути от u к v, состоящие из одной дуги.
2. На первом шаге вычисляются кратчайшие пути между любыми двумя узлами при условии, что в них не больше двух дуг и, помимо самих узлов, в пути может присутствовать только узел a.
3. На втором шаге вычисляются кратчайшие пути, которые состоят не более чем из трех дуг и не более чем из четырех узлов — начального, конечного, a и b.
4. На k-м шаге рассматриваются пути, в которых, помимо начального и конечного узлов, могут встречаться первые k элементов из списка узлов, а ребер — не более $k + 1$.

После того как внешний цикл по `t` переберет все N узлов, в словарях окажутся сведения о кратчайших путях между любыми узлами, причем в них могут встречаться любые узлы графа и *вплоть до $N + 1$ дуг*. В действительности путь из N вершин не может содержать больше $N - 1$ дуг, так что упомянутых пределов хватит на то, чтобы подсчитать настоящие кратчайшие расстояния для любых пар u и v в графе.

Осталось только написать функцию, которая по результатам работы алгоритма восстанавливает конкретный кратчайший путь. Эта функция приведена в примере 7.17 — она мало чем отличается от такой же функции из примера 7.3, только в качестве индексов в словарях используется пара узлов, а не один.

Пример 7.17. Функция all_pairs_path_to() строит кратчайший путь по результатам работы алгоритма Флойда — Уоршелла

```
def all_pairs_path_to(node_from, src, target):     ❶
  if node_from[src, target] is None:
    raise ValueError('Unreachable')                 ❼

  path, v = [], target                              ❷
  while v != src:
    path.append(v)                                  ❸
    v = node_from[src, v]                           ❹

  path.append(src)                                  ❺
  path.reverse()                                    ❻
  return path
```

❶ Чтобы построить весь путь от `src` до `target`, функции нужен словарь `node_from`.

❷ Начнем с конца — с `target`.

❸ Пока не добрались до начала, то есть пока `v` не совпадает с `src`, добавляем `v` в список `path` (узлы пути от `src` до `target` в обратном порядке).

❹ Отступаем на шаг: теперь `v` – это предыдущий узел на пути, который мы взяли из `node_from[src, v]`.

❺ Когда мы добираемся до `src`, цикл завершается, остается только добавить в список сам `src`.

❻ Восстанавливаем порядок узлов в пути, меняя обратный на прямой.

❼ Если `node_from[src, target]` так и остался равным `None`, значит, `target` недостижим из `src`.

Заключение

С помощью графов можно моделировать данные самых разнообразных предметных областей – от географических карт до социальных сетей или последовательностей в биоинформатике. Граф состоит из узлов и соединяющих их ребер, причем ребра могут быть или не быть ориентированными, а также иметь или не иметь числовые веса. Множество естественно возникающих интересных задач на графах имеют практическое применение, и мы умеем их решать.

1. Связный ли граф? Обойдем его **в глубину** и посмотрим, все ли узлы затронул обход.
2. Есть ли контур в ориентированном графе? Обойдем его **в глубину** и будем проверять, не привел ли обход к вершине, все еще находящейся в множестве активного поиска, – это и будет означать цикл.
3. Как попасть из входной вершины в некоторую другую, чтобы путь занимал наименьшее количество ребер? Обойдем граф из этой вершины **в ширину**: длину кратчайшего пути можно вычислить сразу, а сам путь можно восстановить, если для каждого узла запомнить, откуда мы пришли в него кратчайшим путем.
4. Как найти кратчайшее расстояние (сумму весов дуг в пути) во взвешенном графе из входного узла до любого другого? **Алгоритм Дейкстры** вычисляет эти расстояния, а также запоминает предпоследний узел каждого кратчайшего пути: по этим данным можно восстановить любой кратчайший путь до узла, достижимого из входного.
5. Можно ли решить ту же задачу – поиск кратчайшего расстояния из входного узла до любого другого, – если веса дуг в графе имеют разные знаки, но отрицательных контуров в нем нет? Применим **алгоритм Беллмана – Форда**.

6. Как эффективно решить задачу поиска *всех* кратчайших расстояний (суммарных весов дуг в путях) во взвешенном графе от любого узла к любому другому? Воспользуемся **алгоритмом Флойда – Уоршелла**, который составит таблицы, где для каждой пары узлов будет храниться кратчайшее расстояние и предпоследний узел в кратчайшем пути, — по этим таблицам можно будет восстановить полный путь.

Изобретать собственные структуры данных для моделирования графа в Python не стоит: лучше воспользоваться имеющимися сторонними библиотеками, например NetworkX, в которой есть несколько различных типов графов и уже реализовано множество алгоритмов.

Тренировочные задания

1. Хоть это и не рекомендуется, **обход в глубину** можно написать в виде рекурсивной функции. Как всегда, получится еще более читаемый текст программы, но большие графы обрабатывать этим алгоритмом будет невыгодно, а на Python, глубина рекурсии в котором ограничена 1000 вызовами, просто невозможно. Однако на небольших лабиринтах такое решение работать будет, и посмотреть на это стоит. Доработайте пример 7.18 так, чтобы пространство активного поиска содержалось не в явно заданном стеке, а было распределено по контекстам рекурсивного вызова внутренней функции `dfs()`. Достаточно вызывать `dfs()` на каждом помеченном узле и откатываться к предыдущему вызову для просмотра еще не исследованных направлений.

   Пример 7.18. Допишите рекурсивную реализацию поиска в глубину

   ```
   def dfs_search_recursive(G, src):
     marked = {}
     node_from = {}

     def dfs(v):
       """Допишите эту рекурсивную функцию."""

     dfs(src)
       return node_from
   ```

2. Функцию `path_to()`, которая строит полный путь для **обхода в ширину** и **обхода в глубину**, тоже можно написать рекурсивно (с теми же ограничениями, что и в предыдущем задании). Реализуйте рекурсивную генератор-функцию `path_to_recursive(node_from, src, target)`, которая с помощью `yield` порождает последовательность узлов на пути от `src` до `target`.

3. Напишите функцию `recover_cycle(G)`, которая определяет, есть ли контур в орграфе `G`, и *возвращает его* (или `None`, если контура нет).

4. Напишите функцию `recover_negative_cycle(G)`, которая дополняет **алгоритм Беллмана — Форда**, отыскивая отрицательный контур. Сконструируйте свое исключение `NegativeCycleError`, унаследовав его от `RuntimeError`, — это исключение будет порождаться, когда обнаружится, что некоторый путь можно сокращать постоянно. Возьмите последнюю дугу этого пути и постарайтесь найти весь цикл. Передайте цикл в качестве параметра порождаемого исключения[1].

5. Подберите такой простой орграф из $N = 5$ узлов, чтобы **алгоритму Беллмана — Форда** для вычисления кратчайшего пути от определенного узла понадобилось четыре прохода. Для простоты веса всех дуг пускай будут единичными. Подсказка: результат зависит от порядка добавления дуг в граф. В частности, в **алгоритме Беллмана — Форда** дуги перебираются в том порядке, в каком их возвращает `G.edges()`.

6. Исследуйте эффективность всех трех вариантов обхода — **в глубину**, **в ширину** и **направленного** для случайного лабиринта размером $N \times N$. Для этого: 1) останавливайте обход, как только выход найден, и 2) отмечайте количество помеченных узлов.

   Теперь для разных N, равных степеням двойки от 4 до 128, сгенерируйте по 512 случайных графов и подсчитайте среднее количество помеченных узлов для каждого из вариантов обхода. Должно получиться, что **направленный обход** в среднем самый быстрый, а **обход в ширину** в среднем самый медленный.

   Затем создайте *наихудший случай* для **направленного обхода**, в котором он работает почти так же медленно, как **обход в ширину**. На рис. 7.25 приведен пример довольно плохого лабиринта размером 15×15: стены в нем образуют своеобразное корыто, которое не дает добраться до выхода. **Направленный поиск** сначала безрезультатно обшаривает все $(N - 2)^2$ ячейки «корыта» и только затем переваливает через стенку и обходным путем добирается до выходного узла. В файле `ch07/maze.py` репозитория есть класс `Maze` с методом `.initialize()`, который заполняет весь лабиринт стенками и формирует соответствующие структуры данных, останется только удалить бо́льшую часть вертикальных и горизонтальных стен (они хранятся в массивах `.south_wall[]` и `.east_wall[]` соответственно).

[1] Примерно так: raise RuntimeError('Negative Cycle exists in graph.', cycle). — *Примеч. пер.*

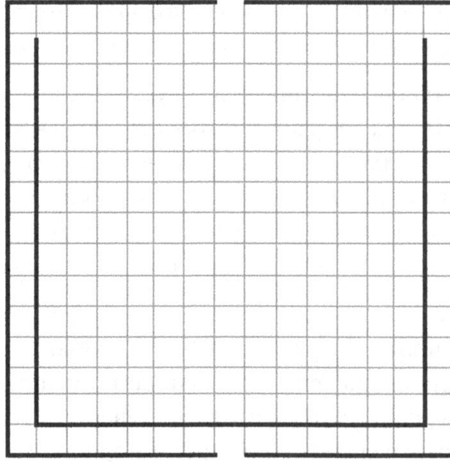


Рис. 7.25. Вариант наихудшего случая для направленного обхода

7. Не содержащий контуров орграф называется *сетью*. **Алгоритм Дейкстры** в *худшем случае* показывает производительность $O((E+N)\log N)$, но на сети он работает быстрее и может отыскать кратчайшие пути из заданного узла за $O(E+N)$ действий. С помощью **топологической сортировки** построим линеаризацию вершин графа и пройдемся по ним в линеаризованном порядке, сокращая каждую дугу, исходящую из очередной вершины. Сконструируйте достаточное количество сетей в виде квадратных *решеток* и экспериментально подтвердите оценку сложности. Для простоты веса всех дуг будем считать единичными. Например, длина кратчайшего пути из узла 1 к узлу 16 в решетке с рис. 7.26 равна 6.

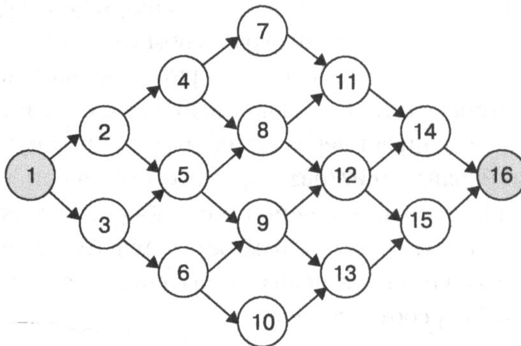


Рис. 7.26. Орграф без контуров — сеть, где поиск кратчайших путей из заданного узла работает быстрее

8. Некоторые водители не пользуются платными дорогами, такими как I-90 в Массачусетсе. В нашем графе дорог Массачусетса ребро `(u, v)` принадлежит трассе I-90, если метки обоих узлов содержат маркировку `'I-90'`. Из всех 2826 ребер 51 принадлежит I-90. Удалите эти ребра из графа и высчитайте кратчайшее расстояние из самой западной точки штата в центр Бостона (точка, обведенная кружком на рис. 7.14, помеченная как `I-90@134&I-93@20&MA3@20(93)&US1@I-93(20)`, потому что там встречаются шесть дорог). Если ехать и по платным дорогам, путешествие занимает 72 ребра суммарной длиной 136.2 мили. Если же не заезжать на I-90, количество ребер увеличится до 104, а длина – до 139.5 мили. Напишите программу, которая получает все эти результаты и создает файл с изображением бесплатного пути.

ГЛАВА 8

Подведем итоги

Цель книги — описать алгоритмы, которые лежат в основе современных информационных технологий, и структуры данных в поддержку этих алгоритмов, от эффективной реализации которых напрямую зависит быстродействие программ. Здесь приведены примеры таких структур данных.

- *Набор.* Несколько (как правило, однотипных) *элементов*, объединенных единой структурой, в которую можно добавлять элементы, а также искать их в ней. Удаление элементов из набора обычно не предусмотрено. Если набор реализован с помощью связного списка, добавление — это вставка в начало, и она имеет константную (O(1)) сложность. Если набор реализован с помощью динамического массива, можно добиться также константной сложности добавления *в среднем*, хотя в редких случаях масштабирования массива сложность будет возрастать до O(N).
- *Стек.* В Python нет необходимости моделировать стек (LIFO — Last in, first out) связным списком, так как с этим легко справляется встроенный тип данных, динамический массив `list`. Тем не менее связный список — классический пример реализации стека. Операции добавления значения на вершину стека и снятия значения оттуда, `push()` и `pop()` (для типа `list` — `.append()` и `.pop()`), должны иметь константную сложность O(1).
- *Очередь.* Если набор данных организован по принципу FIFO — First in, first out, его можно смоделировать связным списком, в котором хранятся ссылки как на первый, так и на последний элементы. При этом обе операции — добавление в конец очереди `enqueue()` и удаление из начала очереди `dequeue()` — будут иметь сложность O(1).
- *Хеш-таблица.* Структура, хранящая пары (`ключ, значение`), эффективность которой зависит от удачного выбора хеш-функции, распределяющей значения по хранилищу, сообразуясь с хешем ключа. Хорошо себя показывает открытая адресация хешей в тандеме с геометрическим масштабированием хранилища при его заполнении выше порогового. Геометрическое масшта-

бирование, например удвоение, позволяет сделать вычислительно тяжелую операцию перехеширования настолько редкой, что это не влияет на среднее быстродействие.

- *Приоритетная очередь*. Если реализовать кучу — структуру данных, которая хранит пары (*значение, приоритет*) таким образом, чтобы на ее вершине всегда находилась пара с наивысшим приоритетом, ее можно рассматривать как очередь. При этом и операция добавления enqueue(), и операция снятия с вершины dequeue() будут иметь логарифмическую сложность $O(\log N)$. Как правило, вместимость приоритетной очереди N известна заранее, если же нет, избежать значимого падения быстродействия нам снова поможет геометрическое масштабирование.
- *Индексированная приоритетная очередь*. Если добавить в реализацию кучи отдельный словарь, в котором записана позиция элемента кучи в массиве-хранилище, получится структура данных, в которой можно быстро (за константное время) найти любой, а не только верхний, элемент. Для классических графов, в которых узлы не имеют названий, а просто пронумерованы от 0 до $N - 1$, вместо словаря можно использовать массив, что еще повысит быстродействие. Индексированная приоритетная очередь поддерживает операции добавления, удаления и *повышения приоритета* элемента, и сложность всех этих операций — логарифмическая, $O(\log N)$.
- *Граф*. Структура, состоящая из узлов и ребер, в которой ребра обычно моделируются матрицей смежности — особенно когда есть подозрение, что в графе могут встретиться вообще все возможные ребра. Если узлы пронумерованы от 0 до $N - 1$, для реализации матрицы смежности можно воспользоваться двумерным массивом. Однако в большинстве случаев удобнее применять список смежности, который можно смоделировать словарем, хранящим списки смежных узлов (или словарем с ключом — ребром, то есть парой смежных узлов). Вручную делать собственную реализацию графа смысла нет — и писать долго, и работать будет небыстро, лучше воспользоваться какой-нибудь готовой удобной реализацией в стороннем Python-модуле. В книге мы применяли модуль NetworkX, поддержка графов в котором эффективна и разнообразна.

В предисловии приведен рисунок, на котором перечислены эти абстрактные типы данных. Разбираясь с алгоритмами, мы смоделировали абстрактные типы теми или иными структурами данных в Python и исследовали быстродействие наших моделей. Результаты представлены в табл. 8.1[1].

[1] Таблица показывает свойства моделей, представленных в этой книге. В практическом программировании стоит пользоваться готовыми типами данных — самого Python или в составе специализированных модулей. — *Примеч. пер.*

Таблица 8.1. Быстродействие абстрактных типов данных

Тип данных	Действие	Производительность	Замечания

Набор	`size()`
	`add()`
	`iterator()`
		O(1)
		O(1)
		O(*N*)
			Проще всего использовать тип `list` — динамический массив с геометрическим масштабированием, которое позволяет добавлять элемент к набору в среднем за константное время. Будучи массивом, `list` поддерживает операцию индексирования, которая тоже работает константное время

Стек	`push()`
	`pop()`
	`is_empty()`
		O(1)
		O(1)
		O(1)
			Эту абстракцию прямо реализует тип `list`, удаление последнего элемента в котором также в среднем константно по времени. Операция `is_empty()` при этом сводится к проверке объекта такого типа на пустоту

Очередь	`enqueue()`
	`dequeue()`
	`is_empty()`
		O(1)
		O(1)
		O(1)
			Для моделирования очереди можно использовать связный список, в котором есть ссылки и на первый, и на последний элемент. Добавление в конец очереди модифицирует последний элемент, удаление из начала — первый. Можно использовать и `list`, в котором эти границы задаются по типу *кольцевого буфера* из главы 4, однако масштабированием такой структуры надо заниматься самостоятельно, иначе она может переполниться

Хеш-таблица	`put()`
	`get()`
	`iterator()`
	`is_empty()`
		O(1)
		O(1)
		O(*N*)
		O(1)
			N пар (*ключ*, *значение*) можно хранить в виде *M* наборов, хеши ключей в которых совпадают. Чем больше *N*, тем больше размер набора, и, чтобы сохранить производительность, нужно примерять геометрическое масштабирование. Другой вариант — *открытая адресация*, при которой все пары хранятся подряд, хеш ключа используется для определения начального индекса, а коллизии разрешаются последовательным просмотром.

Тип данных	Действие	Производительность	Замечания
			В обоих случаях легко организовать итератор по всем парам. Если вдобавок нужно, чтобы в линеаризации пары были упорядочены, для хранения можно воспользоваться двоичным деревом поиска — правда, тогда сложность `put()` и `get()` возрастет до O(log *N*)

Приоритетная очередь	`add()`
	`remove_max()`
	`is_empty()`
		O(log *N*)
		O(log *N*)
		O(1)
			Подойдет новая структура данных — куча, которая хранит пары (*значение*, *приоритет*). Если *N* заранее неизвестно, стоит предусмотреть геометрическое масштабирование, иначе хранилище кучи может переполниться. Внутренние операции `sink()` и `swim()`, описанные в главе 4, определяют общее быстродействие — O(log *N*)

Индексированная приоритетная очередь	`add()`
	`remove_min()`
	`decrease_priority()`
	`is_empty()`
		O(log *N*)
		O(log *N*)
		O(log *N*)
		O(1)
			Используем модифицированную кучу, в которой дополнительно хранится хеш-таблица с позициями значений, так что время поиска такой позиции оказывается O(1). Применение хеш-таблицы позволяет достичь логарифмической — O(log *N*) — производительности действий с очередью. В книге описана приоритетная очередь с порядком возрастания

Встроенные типы данных Python

Язык программирования Python активно развивается более 30 лет, и за это время встроенные типы данных стали весьма удобны и эффективны. Разработчики постоянно — буквально в каждом выпуске Python — пробуют улучшить быстродействие, даже если и ненамного. Довольно показателен текст *The Design and History FAQ* («Часто задаваемые вопросы про историю разработки») с официального сайта Python[1].

[1] Хороший пример — как в Python 3.6 переработали реализацию типа dict, совершенно прозрачно для сообщества и при этом значительно увеличив быстродействие и снизив потребление памяти. — *Примеч. пер.*

Непосредственно в синтаксис языка встроено четыре типа для хранения данных — *кортеж* (`tuple`), *список* (`list`), *словарь* (`dict`) и *множество* (`set`).

- `tuple` (кортеж). Неизменяемая последовательность произвольных объектов Python, с которой можно обращаться как с массивом, только элементам нельзя присваивать новые значения. Реализован как *таблица* — массив ссылок на объекты Python. Часто используется для объединения элементов в группы, например попарно, или для того, чтобы вернуть из функции несколько значений.
- `list` (список). Динамический массив произвольных объектов Python, один из главных типов данных. Удивительно гибкий инструмент, в котором, как и в кортеже, есть очень прозрачно оформленное *секционирование* — изготовление новых объектов-подпоследовательностей. Если нужно пройти такую подпоследовательность элементов циклом, с помощью объекта типа `range()` создается *вычислимая* последовательность индексов и применяется *индексирование*. Список тоже реализован как массив ссылок на объекты, но эти ссылки можно изменять, а также добавлять и удалять. Константную сложность операциям удаления и добавления (в конце списка) придает геометрическое масштабирование.
- `dict` (словарь). Второй главный тип данных в Python — хеш-таблица, которая хранит данные в соответствии с их ключами. Все, что мы знаем из главы 3 о хеш-таблицах, применимо и к `dict`. В реализации Python для разрешения коллизий используется открытая адресация с повторным хешированием. Тройки `полный хеш ключа, ключ, значение` хранятся в динамическом массиве, а их индексы (равные остатку от деления хеша на M) — в списке заранее заданного размера M, равного степени двойки. Это несколько отличается от обычной структуры хеш-таблиц. Размер словаря (массива индексов) начинается с $M = 8$, а при превышении *порога заполнения* в 2/3 применяется геометрическое масштабирование: размер удваивается и хеши вычисляются заново. Размер меньше 8 смысла не имеет — слишком часто происходило бы масштабирование, и так-то в массиве хранится не более пяти элементов. Если большинство элементов таблицы не занято, список объектов достаточно мал, что обеспечивает эффективность по памяти в среднем. Поскольку значения хранятся в списке, всегда известен *порядок их добавления* в словарь; чтобы сохранить константную сложность удаления произвольного элемента словаря, по соответствующему индексу делается пометка о том, что элемент удален, а сам элемент остается в списке-хранилище (таким образом, удаление не уменьшает размер хранилища до следующего масштабирования).

Python — свободное программное обеспечение с открытыми исходными текстами, так что реализацию любого типа данных (Python написан на языке C) всегда можно посмотреть в исходниках[1]. Вместо последовательного просмотра цепочки в случае коллизии хеш-суммы `hc` очередной индекс вычисляется как $((5 \times \texttt{hc}) + \texttt{perturb}) \% 2^n$, где 2^n — это M, размер массива с индексами, а `perturb` — небольшая числовая константа, помогающая равномерно рассеивать хеши по области значений. Довольно интересно поизучать, как небольшое изменение в формуле приводит к более эффективной реализации хеширования. Остаток от деления на степень двойки, как известно, можно взять и без деления — с помощью ***побитовой конъюнкции*** (в Python обозначается знаком `&`).

Таким образом можно значительно увеличить быстродействие. Например, остаток от деления M на 2^n — это M & $(2^n - 1)$[2]. В табл. 8.2 показано время выполнения десяти миллионов операций взятия остатка от деления на 2^n и побитовой конъюнкции с $2^n - 1$ в Python и C: на C написана основная реализация словарей в Python, и там быстродействие отличается более чем впятеро!

Таблица 8.2. Остаток деления на степень двойки быстрее вычислять с помощью побитовой конъюнкции

Язык	Выражение	Время выполнения
Python	1989879384 % *M*	0.6789181 с
Python	1989879384 & (*M* – 1)	0.3776672 с
C	1989879384 % *M*	0.1523320 с
C	1989879384 & (*M* – 1)	0.0279260 с

- `set` (множество). Набор различающихся хешируемых (в случае Python это означает неизменяемых) данных. Попросту говоря, множество — это хеш-таблица, в которой *ничего не хранится*[3]. В отличие от словаря, основное

[1] Например, реализацию словарей можно увидеть здесь: https://oreil.ly/jpI8F. — *Примеч. авт.*

[2] Поясним: 2^n — это число, двоичное представление которого состоит из единицы и n нулей, а двоичное представление $2^n - 1$ соответственно — это n единиц подряд. Остаток от деления M на 2^n — это число от 0 до $2^n - 1$, то есть n младших бит числа M без изменения. Тот же результат мы получим, если взять побитовую конъюнкцию M и $2^n - 1$: младшие n бит числа M не изменятся, а остальные превратятся в 0. — *Примеч. пер.*

[3] Исходный текст реализации множеств на C можно посмотреть в репозитории Python: https://oreil.ly/FWttm. — *Примеч. авт.*

назначение которого — накопление элементов и поиск конкретного значения по (скорее всего) существующему в словаре ключу, множество обычно нужно для того, чтобы проверить, принадлежит ли ему некоторый ключ. Соответственно, в реализации Python оптимизирован не только успешный поиск ключа в множестве (когда ключ в множестве есть), но и неуспешный (когда его там нет). Кроме того, множества в Python поддерживают теоретико-множественные операции, такие как объединение, пересечение, дополнение и т. п.

Реализация стека в Python

В главе 7 мы довольно подробно обсудили, как именно тип `list` в Python реализует абстрактный тип данных «стек». Если коротко: будучи динамическим массивом с геометрическим масштабированием, `list` предоставляет методы `.append()`, который добавляет элемент в конец списка, и `.pop()`, который снимает оттуда элемент. Оба метода имеют константную в среднем сложность, как это видно, например, из табл. 6.1.

В состав дистрибутива Python входит модуль `queue`, в котором реализованы и стек (LIFO), и очередь (FIFO), причем основной упор сделан на структуры данных фиксированного размера и их использование в многопоточных окружениях. Такой стек или очередь поддерживает несколько потоков выполнения, часть из которых может добавлять туда элементы, а часть — снимать их оттуда. Например, `queue.LifoQueue()` создаст стек актуально бесконечного размера, который отличается от обычного списка тем, что методу `.get()` (аналог `.pop()`) по умолчанию передается дополнительный параметр `block=True`, тогда попытка снять значение из пустого стека не вызовет исключения, а приостановится до тех пор, пока в каком-нибудь другом потоке выполнения в этот стек не *положат хотя бы одно значение*. Если создать стек или очередь фиксированного размера, то операция `.put()` (аналог `.append()`) также будет приостанавливаться, если соответствующая структура данных полна, и ждать, пока кто-то другой не снимет оттуда хотя бы одно значение.

В частности, такая программа зависнет на операции `.put()`, и нам придется остановить ее средствами операционной системы:

```
import queue
q = queue.LifoQueue(3)
q.put(9)
q.put(7)
q.put(4)
q.put(3)  # ...эту операцию можно прервать, только остановив программу
```

Если многопоточность не нужна, но вдобавок мы хотим использовать наше хранилище и как стек, и как очередь (то есть наш алгоритм часто удаляет или добавляет элементы и в начало, и в конец хранилища), самый быстрый вариант — это упоминавшаяся в главе 3 двусторонняя очередь (double-ended queue) `deque` из модуля `collections`[1]. Операции добавления и в начало, и в конец `deque` имеют сложность O(1) в среднем, и при этом его быстродействие раз в 30 выше `LifoQueue` (хотя порядок сложности остается константным).

Реализация очередей в Python

Очередь, в принципе, можно было бы смоделировать на базе `list`: использовать метод `.append()` для добавления в конец очереди (последний элемент списка) и `.pop(0)` — для снятия элемента из ее начала (нулевой элемент списка). Однако мы знаем, что работа с началом списка требует уже не константного, а линейного времени, и, как это видно из табл. 6.1, лучше так вообще никогда не делать.

Очередь с поддержкой многопоточности (несколько потоков выполнения добавляют в очередь, несколько снимают) и блокировок при переполнении и опустошении реализована классом `queue.Queue`, но, как уже говорилось выше относительно класса `queue.LifoQueue`, если все эти свойства не нужны, он оказывается слишком мощным инструментом с большими накладными расходами. Как и в `LifoQueue`, методы `.put()` и `.get()` используются для добавления и удаления элементов, и, если ограничить размер хранилища, возникает блокировка. Например, такой диалог в командной строке Python придется прерывать вручную средствами операционной системы:

```
>>> import queue
>>> q = queue.Queue(2)
>>> q.put(2)
>>> q.put(5)
>>> q.put(8)
... blocks until terminated
```

Классы `queue.Queue` и `queue.LifoQueue` можно использовать для создания очередей с *заданиями*: каждый поток выполнения задания должен вызвать метод `.task_done()` после того, как выполнит последнюю, по его мнению, операцию с очередью, а поток выполнения, запустивший эти задания, вызовет метод `.join()`, который приостановится до тех пор, пока все задания не завершатся. Если вы не пользуетесь абстракцией «задание», можно задействовать класс

[1] По сходству звучания deque и deck deque иногда называют колодой. — *Примеч. пер.*

`queue.SimpleQueue` — в нем нет этих двух методов (и механизма их реализации), и работает он несколько эффективнее. Еще раз напомню, что классы из модуля `queue` обладают впечатляющим набором свойств, которые, возможно, не будут нужны в простых программах. Они не только адаптированы к многопоточным средам (тредобезопасны), но и допускают использование в повторно-входимых процедурах (сопрограммах) и т. п. — все это за счет падения производительности. Используйте эти классы, если вам в самом деле нужны их свойства.

Самая быстрая реализация очереди в Python — класс `collections.dequeue`: в нем неплохо оптимизирована производительность и, если важно быстродействие очереди, стоит использовать именно его. В табл. 8.3 приведено сравнение быстродействия очередей на базе различных типов — `dequeue` оказывается лучшим, а `list` — худшим, причем операция снятия из очереди на нем имеет сложность $O(N)$ — в отличие от остальных классов; так что еще раз напомню: для очередей списки использовать не надо.

Таблица 8.3. Быстродействие снятия элемента из очереди для различных реализаций

N	list	deque	SimpleQueue	Queue
1024	0.012	0.004	0.114	0.005
2048	0.021	0.004	0.115	0.005
4096	0.043	0.004	0.115	0.005
8192	0.095	0.004	0.115	0.005
16384	0.187	0.004	0.115	0.005

Базовые типы данных Python можно использовать для решения самых разнообразных задач, но в каждом случае стоит тщательно выбирать, с помощью чего моделировать ту или иную структуру данных, чтобы программа заработала быстрее.

Реализация кучи и приоритетной очереди

В базовом дистрибутиве Python есть модуль `heapq`, который реализует кучу с порядком возрастания — примерно так, как мы это сделали в главе 4, но в качестве хранилища используется обычный список с индексацией элементов от 0, а не от 1, как в нашем варианте.

Более того, никакой специальной структуры данных для кучи создавать не надо. Достаточно передать процедуре `heapq.heapify(h)` список исходных значений `h`,

и она превратит `h` в кучу с порядком возрастания, после чего можно будет пользоваться им и как кучей, и как списком (только не стоит нарушать порядок). Другой способ — завести пустой список `h=[]` и добавлять значения в кучу с помощью `heapq.heappush(h, `*`значение`*`)`. Наименьшее значение можно снять с кучи с помощью `heapq.heappop(h)`. Вдобавок к этому в `heapq` имеются две полезные функции.

- `heapq.heappushpop(h, value)`. Сначала добавляет в кучу новый элемент, а затем снимает с ее вершины наименьшее значение (если новый элемент *и был* минимальным, ничего добавлять и снимать не надо, достаточно просто вернуть его).
- `heapq.heapreplace(h, value)`. Сначала снимает с кучи минимальный элемент, а затем добавляет туда новый (если он оказался не больше предыдущего минимума, достаточно просто заменить один на другой).

Если изучить содержимое `h` — списка, в котором поддерживается порядок кучи, мы увидим в нем прямое сходство с хранилищем элементов кучи из нашей реализации в главе 4.

Приоритизированную очередь реализует класс `PriorityQueue` из модуля `queue` (https://oreil.ly/sUiZd) — как и другие структуры данных этого модуля, он поддерживает многопоточность и блокировку операций при выходе за размер хранилища. Нетрудно заметить, что это просто обертка `queue.Queue` вокруг кучи из `heapq`: метод `.put(item)`, добавляющий в очередь `item` — кортеж `(`*`приоритет, значение`*`)`, просто хранит этот кортеж в куче, а метод `.get()`, снимающий элемент из начала очереди, просто выполняет `heappop()`.

А вот реализации индексированной приоритетной очереди среди стандартных структур данных Python нет, что неудивительно, ибо чаще всего она нужна при обработке графов — например, в **алгоритме Дейкстры**, который мы рассматривали в главе 7. Метод `IndexedMinPQ.decrease_priority()` из нашей реализации в этой главе оказался коротким и эффективным как раз за счет задействованной в нем совокупности различных структур данных.

Что изучать дальше?

В книге мы только пробежались по верхам необъятного пространства — науки об алгоритмах. Углубляться в это пространство можно по нескольким направлениям, в различных предметных областях и с различными подходами к пониманию алгоритмов.

- *Вычислительная геометрия.* Великое множество практических задач включают в себя обработку наборов точек в двумерных или даже многомерных

координатах. Алгоритмы для такой обработки весьма разнообразны, например, нередко используется подход «разделяй и властвуй», описанный в книге, и для эффективной их реализации создаются специализированные структуры данных. Самые распространенные — K-мерные деревья и их подвиды: дерево квадратов (для разбиения двумерного пространства), октодерево (для разбиения трехмерного пространства) и R-дерево (для индексации многомерной информации). Как видим, идея двоичных деревьев живет и побеждает повсеместно, в различных предметных областях.

- *Динамическое программирование.* Наглядный представитель этого класса алгоритмов — **алгоритм Флойда — Уоршелла** (поиск кратчайших путей в графе от заданной вершины). Теория, лежащая в основе динамического программирования, позволила выработать целый спектр эффективных алгоритмов. Подробнее об этом можно прочитать в книге *Algorithms in a Nutshell*[1], которая также вышла в издательстве O'Reilly (https://oreil.ly/1lXRF).
- *Параллельные и распределенные алгоритмы.* В нашей книге мы изучали алгоритмы, использующие единственный поток выполнения на единственном компьютере. Если решение задачи можно разбить на несколько независимо вычисляемых подзадач, эти решения можно запустить одновременно, например, на одном компьютере с несколькими процессорами или на нескольких вычислительных узлах, объединенных сетью. Параллелизмом довольно непросто управлять, и сами алгоритмы становятся сложнее, но игра стоит свеч: выигрыш в быстродействии от умело спроектированного параллелизма может быть очень значительным.
- *Численные методы.* Вычисления в практических задачах совершенно не обязаны быть алгебраически точными. Более того, для некоторых важных задач вычислительная сложность точного решения может быть слишком большой для необходимого набора данных или даже точного решения может не существовать вообще. Такие задачи можно решить приближенно, с заданной степенью точности — и на этот счет существует множество алгоритмов.
- *Вероятностные методы.* Иногда для решения задачи невозможно построить *алгоритм* в собственном смысле: невозможно (или вычислительно сложно) написать программу, которая на одном и том же наборе входных данных давала бы одинаковый удовлетворительный результат. Можно применить принципиально иной подход: ввести в вычисления случайный компонент и запустить программу довольно много раз, получив множество отлича-

[1] *Хайнеман Д., Селков С., Поллис Г.* Алгоритмы. Справочник с примерами на C, C++, Java и Python.

ющихся результатов. В одних случаях подходящим ответом будет среднее из полученных значений, в других — если есть функция, позволяющая оценить правильность результата, — процесс повторяется, пока проверочная функция не даст добро (в действительности подходов, конечно, больше).

Конечно же, ни одна книга не может объять необъятного — всего многообразия возможных алгоритмов. В 1962 году Дональд Кнут, один из столпов программирования, начал писать грандиозную книгу — «Искусство программирования». К нынешнему времени — более чем 60 лет спустя — вышло три первых тома этой книги (в 1968, 1969 и 1973 годах), а также первая часть четвертого тома (в 2011-м). Впереди еще три тома, так что проект далек от завершения!

Изучать и применять алгоритмы можно до бесконечности, и я надеюсь, что знания, которые вложены в эту книгу, помогут вам в разработке быстрых и эффективных программ.

https://t.me/it_boooks/2

Об авторе

Джордж Хайнеман — профессор computer science с более чем 20-летним стажем разработки программного обеспечения и исследования эффективности алгоритмов. Хайнеман — автор книги *Algorithms in a Nutshell* и множества учебных курсов O'Reilly, таких как Exploring Algorithms in Python («Исследование алгоритмов на языке Python») и Working with Algorithms in Python («Работа с алгоритмами на языке Python»). С давних пор он интересуется логическими и математическими головоломками и даже изобрел несколько, например Sujiken® (вариант судоку) и Trexagon.

Иллюстрация на обложке

Животное на обложке книги — чесапикский голубой краб *(Callinectes sapidus)*. Название рода *Callinectes* происходит от древнегреческого «хороший пловец», а название вида — *sapidus* — на латыни значит «вкусный», «пикантный». Голубой цвет крабу придают пигменты в панцире, в частности ярко-синий альфа-крустацианин, который, взаимодействуя с красным астаксантином, дает зелено-голубой окрас. Если краба сварить, альфа-крустацианин разрушается, и панцирь приобретает яркий оранжево-красный оттенок.

Родина голубого краба — атлантическое побережье Северной и Южной Америки и Мексиканский залив. В Европу и Японию он попал, по-видимому, с водяным балластом, и уже в 1901 году его можно было встретить в тамошних водах. Прежде считалось, что причина расширения ареала голубого краба — повышение температуры океана вследствие глобального потепления.

Эти крабы откладывают икру на мелководье, после чего приливные течения относят ее на глубину. Планктонная личинка краба проходит восемь стадий развития, прежде чем станет похожей на взрослую особь. Как и все ракообразные, голубые крабы растут во время линьки: старый панцирь сбрасывают и отращивают новый — побольше. Считается, что за свою жизнь краб успевает перелинять около 25 раз и дорасти примерно до 12 см в ширину. Брюшко самцов плоское, самок — обширное и округлое; окраской они почти не различаются.

Многие животные с обложек O'Reilly находятся под угрозой исчезновения, и все животные без исключения — важный элемент биосферы.

Иллюстрация на обложке выполнена Карен Монтгомери (Karen Montgomery): за основу взята черно-белая гравюра из книги 1887 года *Animal Life in the Sea and on the Land* («Жизнь животных в море и на суше. Зоология для юношества»).

Джордж Хайнеман

Алгоритмы. С примерами на Python

Перевел с английского Г. Курячий

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

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: 07.2023. 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 17.05.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 8680.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н

O'REILLY®

Алгоритмы

с примерами на Python

Когда нужно, чтобы программа работала быстро и занимала поменьше памяти, профессионального программиста выручают знание алгоритмов и практика их применения. Эта книга — как раз про практику. Ее автор Джордж Хайнеман предлагает краткое, но четкое и последовательное описание основных алгоритмов, которые можно эффективно использовать в большинстве языков программирования. О том, какими методами решаются различные вычислительные задачи, стоит знать и разработчикам, и тестировщикам, и интеграторам.

Книга поможет вам:

- исследовать важнейшие для теории и практики программирования алгоритмы;
- познакомиться с основными методиками решения вычислительных задач, например подходом «разделяй и властвуй» и динамическим программированием;
- научиться анализировать исходные тексты программ с точки зрения вычислительной сложности (в терминах «O большое»);
- представить важнейшие алгоритмы в виде последовательности шагов.

«Доступно изложенное введение в алгоритмы, которое можно сразу применить, — и ваши программы станут быстрее. Руководство по важнейшим алгоритмам и структурам данных в программировании. Нелишне изучить эту книгу перед тем, как устраиваться на работу: собеседование наверняка пройдет блестяще».

— Цви Галиль,
почетный декан
Технологического института
штата Джорджия, кафедра
компьютерных наук
им. Фредерика Дж. Стори

Джордж Хайнеман — профессор computer science с более чем двадцатилетним стажем разработки программного обеспечения и исследования эффективности алгоритмов. Хайнеман — автор книги Algorithms in a Nutshell и множества учебных курсов O'Reilly, таких как Exploring Algorithms in Python («Исследование алгоритмов на языке Python») и Working with Algorithms in Python («Работа с алгоритмами на языке Python»). Автор логических головоломок Sujiken® (вариант судоку) и Trexagon.

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

ISBN:978-5-4461-1963-9