

Кори Альтхофф

Computer Science

для программиста-самоучки

Все, что нужно знать
о структурах данных и алгоритмах



The Self-Taught Computer Scientist

The beginner's guide
to data structures &
algorithms

Cory Althoff

WILEY

Кори Альтхофф

Computer Science

для программиста-самоучки

Все, что нужно знать
о структурах данных и алгоритмах



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018
УДК 004.422.63
А58

Альтхофф Кори

A58 Computer Science для программиста-самоучки. Все, что нужно знать о структурах данных и алгоритмах. — СПб.: Питер, 2023. — 240 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2010-9

Книги Кори Альтхоффа вдохновили сотни тысяч людей на самостоятельное изучение программирования.

Чтобы стать профи в программировании, не обязательно иметь диплом в области computer science, и личный опыт Кори подтверждает это: он стал разработчиком ПО в eBay и добился этого самостоятельно.

Познакомьтесь с наиболее важными темами computer science, в которых должен разбираться каждый программист-самоучка, мечтающий о выдающейся карьере, — это структуры данных и алгоритмы. «Computer Science для программиста-самоучки» поможет вам пройти техническое интервью, без которого нельзя получить работу в «айти».

Книга написана для абсолютных новичков, поэтому у вас не должно возникнуть трудностей, даже если ранее ничего не слышали о computer science.

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

ББК 32.973.2-018
УДК 004.422.63

Права на издание получены по соглашению с John Wiley & Sons, Inc. 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. 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 978-1119724414 англ.
ISBN 978-5-4461-2010-9

© 2022 by John Wiley & Sons, Inc. All rights reserved.
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

Об авторе	12
О научном редакторе	12
Благодарности.....	13
От издательства	14
Введение	15
I Введение в алгоритмы.....	25
1 Что такое алгоритм.....	26
2 Рекурсия.....	45
3 Поисковые алгоритмы	51
4 Алгоритмы сортировки.....	64
5 Строковые алгоритмы.....	85
6 Математика.....	94
7 Вдохновение самоучки: Маргарет Гамильтон	111
II Структуры данных	113
8 Что такое структуры данных.....	114
9 Массивы.....	119
10 Связные списки	134
11 Стеки	147
12 Очереди.....	162
13 Хеш-таблицы.....	174
14 Двоичные деревья	184
15 Двоичные кучи.....	202
16 Графы	213
17 Вдохновение самоучки: Илон Маск	230
18 Дальнейшие шаги	233

Оглавление

Об авторе	12
О научном редакторе	12
Благодарности	13
От издательства	14
Введение	15
Чему вы научитесь	17
Для кого эта книга	19
Истории успеха самоучек	20
Мэтт Мансон	20
Тианни Майерс	21
Начало работы	22
Устанавливаем Python	22
Устранение неполадок	23
Практикумы	23
Не бросайте начатое	23
I Введение в алгоритмы	25
1 Что такое алгоритм	26
Анализируем алгоритмы	27
Постоянное время	32
Логарифмическое время	33
Линейное время	34
Линейно-логарифмическое время	35
Квадратичное время	36
Кубическое время	38

Экспоненциальное время.....	39
Сложность в лучшем и худшем случаях	40
Пространственная сложность	41
Почему все это важно	42
Словарь терминов	42
Практикум	44
2 Рекурсия.....	45
Когда использовать рекурсию.....	50
Словарь терминов	50
Практикум	50
3 Поисковые алгоритмы	51
Линейный поиск.....	52
Когда следует использовать линейный поиск	53
Двоичный поиск.....	54
Когда следует использовать двоичный поиск.....	57
Поиск символов.....	60
Словарь терминов.....	62
Практикум	63
4 Алгоритмы сортировки.....	64
Пузырьковая сортировка.....	65
Когда следует использовать пузырьковую сортировку	69
Сортировка вставками	70
Когда следует использовать сортировку вставками	74
Сортировка слиянием	74
Когда следует использовать сортировку слиянием	81
Алгоритмы сортировки в Python.....	82
Словарь терминов	84
Практикум	84
5 Строковые алгоритмы.....	85
Обнаружение анаграмм.....	85
Выявление палиндромов	86

Последняя цифра	87
Шифр Цезаря.....	89
Словарь терминов	92
Практикум	93
6 Математика.....	94
Двоичный код	94
Побитовые операторы.....	98
FizzBuzz	102
Наибольший общий делитель.....	104
Алгоритм Евклида.....	107
Простые числа.....	108
Словарь терминов	110
Практикум	110
7 Вдохновение самоучки: Маргарет Гамильтон	111
 II Структуры данных	 113
8 Что такое структуры данных.....	114
Словарь терминов	117
Практикум	118
9 Массивы.....	119
Производительность массива.....	121
Создание массива	123
Перемещение нулей	124
Объединение двух списков.....	127
Поиск дубликатов в списке	128
Поиск области пересечения двух списков.....	130
Словарь терминов	132
Практикум	133

10	Связные списки	134
	Производительность связанных списков.....	137
	Создаем связанные списки	138
	Поиск по связанному списку	141
	Удаление узла из связанного списка	142
	Реверсия связанного списка.....	143
	Поиск цикла связанного списка.....	144
	Словарь терминов	145
	Практикум	146
11	Стеки	147
	Когда использовать стеки	148
	Создаем стек.....	150
	Использование стеков для реверсирования строк	154
	Минимальный стек.....	155
	Помещенные в стек скобки	158
	Словарь терминов	160
	Практикум	161
12	Очереди.....	162
	Когда использовать очереди.....	163
	Создаем очередь	165
	Встроенный класс Queue	170
	Создаем очередь с помощью двух стеков.....	171
	Словарь терминов	172
	Практикум	173
13	Хеш-таблицы.....	174
	Когда использовать хеш-таблицы	178
	Символы в строке	179
	Сумма двух	181
	Словарь терминов	183
	Практикум	183

14	Двоичные деревья	184
	Когда использовать деревья	188
	Создаем двоичное дерево	191
	Обход дерева в ширину	193
	Другие обходы дерева	196
	Инвертируем двоичное дерево	199
	Словарь терминов	200
	Практикум	201
15	Двоичные кучи	202
	Когда использовать кучи	207
	Создаем кучу	208
	Соединяем канаты с минимальными затратами	210
	Словарь терминов	211
	Практикум	212
16	Графы	213
	Когда использовать графы	218
	Создаем граф	219
	Алгоритм Дейкстры	221
	Словарь терминов	228
	Практикум	229
17	Вдохновение самоучки: Илон Маск	230
18	Дальнейшие шаги	233
	Что дальше?	233
	Карьера фрилансера	234
	Как попасть на собеседование	235
	Как подготовиться к техническому интервью	235
	Дополнительные источники	236
	Заключительные размышления	237

*Посвящаю эту книгу жене Бобби и дочери Луке.
Я так сильно люблю вас!*

Об авторе

Кори Альтхофф — писатель, программист и лектор. Его первая книга *The Self-Taught Programmer* («Программист-самоучка»)¹ была переведена на семь языков, а после ее публикации в обиход вошло понятие *программист-самоучка*. Сайт Book Authority назвал *The Self-Taught Programmer* одной из величайших книг по программированию всех времен, а The Next Web включил ее в топ-10 книг, которые помогут стать отличным программистом. Более 200 тысяч разработчиков состоят в сообществе программистов-самоучек, собранном Кори Альтхоффом через популярную группу на Facebook, через блог, новостную рассылку и курсы на платформе Udemу. Кори живет в Калифорнии с женой и дочерью.

О научном редакторе

Доктор **Ханну Парвиайнен** — астрофизик, изучающий планеты за пределами нашей Солнечной системы, сотрудник Канарского института астрофизики — одного из ведущих астрофизических институтов в мире, в котором находится крупнейший из существующих оптических телескопов. Ханну Парвиайнен несколько лет был аспирантом-исследователем в Оксфордском университете. Среди ключевых тем его работ — научные вычисления и современные численные методы. Имеет более 20 лет опыта программирования на языке Python.

¹ Книга вышла на русском языке под названием «Сам себе программист», но, на наш взгляд, такой перевод названия не в полной мере отражает мысль, лежащую в основе книги. Она адресована людям, которые хотят добиться всего своими собственными силами, то есть программистам-самоучкам. — *Примеч. ред.*

Благодарности

Хочу сказать огромное спасибо всем замечательным людям, благодаря которым эта книга увидела свет. В первую очередь моей жене Бобби, любви всей моей жизни, — ты всегда меня поддерживала! Я благодарен своему отцу Джеймсу Альтхоффу: он потратил огромное количество времени, помогая мне в написании книги. Следующая благодарность Стиву Бушу — спасибо тебе за то, что прочитал написанное и поделился своим мнением! Благодарю редактора Робина Альвареса и научного редактора Ханну Парвиайнена. И наконец, хочу выразить благодарность своему издателю Девону Льюису, который сделал этот проект возможным. Спасибо моей дочери Луке за то, что она лучшая дочь в мире и всегда вдохновляет меня работать изо всех сил. Я люблю тебя, Лука! Без поддержки всех этих людей у меня ничего бы не получилось. Спасибо вам огромное!

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.

Введение

Свой путь в программирование я начал, когда вышел из колледжа со степенью в области политологии. После завершения учебы я активно искал работу. У меня не было тех навыков, которые требовались работодателям, но мои друзья, изучавшие более практические предметы, получили высокооплачиваемую работу. Я же в это время безуспешно ходил на собеседования, ничего не зарабатывал и чувствовал себя полным лузером. Так, живя в Кремниевой долине и окруженный кодерами, я решил научиться программировать. Тогда я даже не думал, что стою на пороге самого сумасбродного и захватывающего приключения в своей жизни.

Это была не первая моя попытка научиться кодировать: я и раньше пробовал, но без особого успеха. В свой первый год обучения в колледже я посещал курс лекций по программированию, однако ничего там не понял и быстро его бросил. К сожалению, большинство учебных заведений в качестве первого языка программирования предлагают изучать Java, в котором новичкам сложно разобраться. Вместо Java я решил самостоятельно изучить Python — один из самых легких языков для начинающих. И хоть я выбрал легкий для изучения язык, я практически бросил и его. Мне приходилось выуживать информацию из множества различных источников, что сильно меня сбивало. Не помогало и чувство одиночества на протяжении всего моего пути. Я не посещал лекций, где полно студентов и можно получить поддержку.

Я был близок к тому, чтобы сдаться и все бросить, когда начал больше времени проводить в онлайн-сообществах по программированию, например в Stack Overflow. Принадлежность к сообществу помогла сохранить мою мотивацию, и я заново ощутил энтузиазм. Было много взлетов и падений, иногда я был готов от всего отказаться, но менее чем через год после моего судьбоносного решения научиться программировать я получил должность разработчика программного обеспечения в компании eBay. Годом ранее для меня было бы удачей получить работу в службе технической поддержки клиентов, а теперь мне платили 50 долларов в час за программирование в известной технологической компании. Я не мог в это поверить! Однако дело было не в деньгах — моя самооценка выросла в разы. После того как я научился кодировать, я почувствовал, что смогу достичь всего, чего захочу.

После работы в eBay я начал работать над стартапом в Пало-Альто. В конце концов я решил отдохнуть и отправился в путешествие по Юго-Восточной Азии.

На заднем сиденье такси, которое везло меня под дождем по узеньким улочкам Семиньяка на Бали, мне пришла в голову идея. Дома меня часто спрашивали о моем опыте инженера-программиста. Быть разработчиком в Кремниевой долине — обычное дело, но я отличался от других тем, что не имел образования в этой области.

Идея заключалась в том, чтобы написать книгу под названием «Программист-самоучка», изложив в ней не только основы программирования, но и все то, чему я научился, работая инженером-программистом. Иными словами, я хотел помочь другим пройти тот же путь. Итак, я взялся за создание дорожной карты для программистов-самоучек. Я потратил год на написание книги, а потом сам же ее и опубликовал. Я не был уверен, что ее будут читать, и даже был уверен в обратном — что никто читать не будет, но все равно хотел поделиться личным опытом. К моему удивлению, я продал тысячи экземпляров в первые несколько месяцев и начал получать множество писем со всего мира от программистов-самоучек и тех, кто только собирался встать на этот путь.

Полученные отзывы вдохновили меня, поэтому я решил помочь другим справиться еще с одной проблемой, с которой и сам столкнулся во время обучения программированию, — проблему одиночества. Я создал группу на Facebook под названием *Self-Taught Programmers* — место, где новички могут поддерживать друг друга. Сейчас она насчитывает более 60 тысяч участников, в ней люди помогают друг другу в различных вопросах, обмениваются знаниями и делятся своими историями успеха. Если хотите стать частью нашего сообщества, присоединяйтесь — <https://facebook.com/groups/selftaughtprogrammers>. Вы можете также подписаться на мою новостную рассылку на сайте theselftaughtprogrammer.io.

Когда мне доводилось в интернет-общении упоминать о том, что я работаю программистом без специального образования в области computer science, я всегда получал как минимум несколько негативных комментариев о том, что невозможно работать программистом без соответствующего образования. Некоторые даже возмущались: «Что вы, программисты-самоучки, творите?! Вам нужно образование! Ни в одной компании вас не будут воспринимать всерьез!» Сегодня такие комментарии встречаются гораздо реже. А если подобное все же пишут, я направляю комментатора в группу *Self-Taught Programmers*: наши участники работают в компаниях по всему миру на самых разных должностях — от младших инженеров-программистов до ведущих разработчиков программного обеспечения.

Между тем моя книга продавалась лучше, чем я мог себе представить, — она даже легла в основу одного из популярных курсов на Udemy. Общение со множеством таких замечательных людей, обучающихся программированию, было для меня

восхитительным опытом, и я рад продолжить это увлекательное путешествие с новой книгой, которая является дополнением и продолжением первой. Я советую начать чтение с первой книги — *The Self-Taught Programmer*, если только вы уже не владеете основами программирования. Вторая книга, которую вы держите в руках — «Computer Science для программиста-самоучки» (в оригинале *The Self-Taught Computer Scientist*), — подразумевает, что вы уже умеете программировать на языке Python. Но если не умеете, то либо обратитесь к моей первой книге, либо пройдите мой курс на Udey, либо выучите Python любым подходящим для вас способом.

Чему вы научитесь

Если моя первая книга знакомит с темами и навыками, необходимыми для профессионального программирования, то вторая представляет собой введение в computer science, в частности в структуры данных и алгоритмы. Computer science¹ — это дисциплина, изучающая компьютеры и их работу. Когда вы поступаете в колледж на инженера-программиста, вы не специализируетесь на программировании — вашей профильной дисциплиной является именно computer science. Студенты данной специальности изучают математику, архитектуру компьютера, компиляторы, операционные системы, структуры данных, сетевое программирование и многое другое.

Каждой из этих тем посвящено множество толстых изданий, и я в своей книге не в состоянии рассмотреть все темы подробно. Computer science — объемная дисциплина: сколько бы вы ее ни изучали, все равно останутся неизведанные области. Я стремился не охватить весь объем знаний, которые вы могли бы получить, изучая компьютерные науки в колледже, а познакомить с некоторыми важными понятиями информатики, чтобы дать вам инструменты для решения различных задач.

Вам, как программисту-самоучке, важнее всего разобраться в двух темах — в структурах данных и в алгоритмах. Именно на них я и сфокусируюсь. Я разделил книгу на две части. Часть I — введение в алгоритмы. Вы узнаете, что такое алгоритмы и чем один алгоритм лучше другого. Вы познакомитесь с разными их видами — линейным и двоичным (бинарным) поиском. Часть II — введение в структуры данных. Вы узнаете о таких понятиях, как структуры данных, и изучите массивы, связанные списки, стеки, очереди, хеш-таблицы, двоичные деревья, двоичные кучи и графы. В завершение я расскажу, что делать после прочтения книги и какие

¹ Или информатика. — *Примеч. науч. ред.*

следующие действия вы можете предпринять, а также предложу другие источники, которые помогут вам на пути изучения программирования.

В предыдущей книге я объяснял, что не имеет смысла изучать computer science, не освоив сначала программирование. Это, однако, не означает, что эту дисциплину можно игнорировать. Вы должны изучить computer science, если хотите стать успешным программистом. Все просто: если вы не разбираетесь в принципах работы компьютеров, то не сможете эффективно писать код. Практически каждая компания, нанимающая программистов, проводит с ними техническое собеседование, которое всегда сфокусировано на одном аспекте — computer science. Особое внимание уделяется структурам данных и алгоритмам. Если вы захотите устроиться в Facebook, Google, Airbnb или любую другую топовую компанию (неважно, какого она размера), вам придется пройти техническое интервью, в центре внимания которого будут структуры данных и алгоритмы. Не имея глубоких знаний по этим двум темам, вы провалите собеседование — пройти его наугад не получится. Потенциальный работодатель будет задавать подробные вопросы о структурах данных, алгоритмах и многом другом, и вам лучше знать эти ответы, иначе работу вы не получите.

Кроме того, нанимая вас на первую работу, работодатели предполагают, что вы знакомы с основами computer science. Если им придется объяснять вам, почему алгоритм $O(n^3)$ не лучшее решение, они навряд ли будут вами довольны. В такой ситуации был я, когда получил свою первую работу в eBay. Я попал в команду невероятно талантливых программистов из Стэнфорда, Беркли и Калифорнийского технологического института. У них у всех были глубокие познания в области компьютерных наук, и я чувствовал себя неуверенно и не на своем месте. Изучение computer science поможет вам как программисту-самоучке избежать подобной участи.

Более того, изучение структуры данных и алгоритмов позволит вам улучшить и навыки разработчика. Возможность немедленно оценить качество проделанной работы — ключ к овладению навыками. Когда вы практикуетесь только в программировании, у вас нет обратной связи. Например, если вы создаете веб-сайт, то он может работать, несмотря на то что код совершенно ужасен, а вы не знаете, как понять, хорош код или нет. Однако когда вы разбираетесь в алгоритмах, ситуация иная. Существует множество известных алгоритмов, а это значит, что вы можете написать код для решения задачи, сравнить свой результат с уже существующим алгоритмом и сразу же понять, насколько ваше решение качественно. Такая практика с положительной обратной связью улучшит ваши навыки программирования.

Самой большой ошибкой, которую я совершил как программист-самоучка, была попытка ворваться в индустрию программного обеспечения, не уделив

должного внимания изучению структур данных и алгоритмов. Если бы я подготовился как следует, мой путь был бы более легким. Вам не нужно повторять моих ошибок!

Как я уже упоминал, компьютерные науки — обширная дисциплина. Есть веская причина того, что студенты тратят четыре года на их изучение: очень многому надо научиться. У вас может не быть этих четырех лет. К счастью, вам это и не нужно. В этой книге описаны наиболее важные вещи, которые вам следует знать для построения успешной карьеры инженера-программиста. Конечно, чтение книги не заменит вам четырех лет обучения в колледже, однако, если вы все прочитаете и отработаете примеры, у вас будет достаточно прочная база, чтобы пройти техническое собеседование. Кроме того, вы будете чувствовать себя комфортно в компании тех, кто получил профильное образование, и значительно улучшите свои навыки в программировании.

Для кого эта книга

Итак, я убедил вас, что самоучки могут программировать на профессиональном уровне и что вам нужно изучать информатику, особенно структуры данных и алгоритмы. Но значит ли это, что вы не можете читать книгу, если учитесь программированию не самостоятельно, а в колледже? Конечно же нет! Мы рады видеть всех в нашем сообществе самоучек! Моя первая книга стала неожиданно популярной среди студентов колледжей. Несколько преподавателей даже связались со мной и сказали, что читают свои лекции по моей книге.

Студенты, изучающие информатику, часто спрашивают меня, надо ли им бросать свои образовательные учреждения. Моя цель — вдохновить как можно больше людей на освоение программирования. То есть я хочу донести до всех мысль, что программировать профессионально без получения соответствующего образования возможно. Но если вы уже в колледже, то это тоже сработает и вам не нужно бросать учебу. Оставайтесь в колледже! Вы все равно можете стать частью сообщества самоучек, руководствуясь нашим девизом «Всегда учиться!» на своих занятиях и стараясь выходить за пределы учебных программ, чтобы узнать больше, чем преподаватель может вам дать.

Как же понять, что вы готовы изучать computer science? Легко. Если вы уже знаете, как программировать, вы готовы! Я написал эту книгу для каждого, кто хочет узнать больше о computer science. Неважно, читаете вы эту книгу для того, чтобы заполнить пробелы в знаниях, или чтобы подготовиться к техническому интервью, или чтобы почувствовать себя компетентным на своей работе и стать лучше в качестве программиста, — я написал эту книгу для вас.

Истории успеха самоучек

На должность инженера-программиста меня взяли без диплома, и с тех пор я каждый день узнаю об успехах в профессии тех, кто учился всему самостоятельно. Несомненно, вы можете построить успешную карьеру программиста, не имея диплома. Я знаю, что для некоторых подобное может казаться нереальным, поэтому, прежде чем погрузиться в computer science, я хочу поделиться с вами несколькими историями успеха программистов-самоучек из моей группы на Facebook.

Мэтт Мансон

Первым будет Мэтт Мансон, член группы *Self-Taught Programmers* на Facebook. Вот его история, рассказанная им же самим.

Все началось, когда я потерял работу в финансово-технологической компании. Чтобы сводить концы с концами, я искал возможность подработать: нарезал линзы для очков, ремонтировал автомобили, работал официантом и занимался небольшими проектами по программированию. Несмотря на все старания, через несколько месяцев я лишился жилья. Эта история о том, как я избежал участи бездомного, став программистом.

Незадолго до потери работы я поступил в колледж. Оставшись без жилья, я продолжил заниматься, живя в машине и палатке. Моя семья не могла мне помочь. Они не понимали, что низкооплачиваемые подработки не могут прокормить человека и что этих денег не хватит ни на бензин, ни на крышу над головой. Тем не менее я все еще не хотел просить помощи у друзей. В сентябре я продал свой фургон, обналичил то, что оставалось на накопительном пенсионном счете, и проехал около 1800 миль от моего родного города Хелена, штат Монтана, до Остина, штат Техас, чтобы попытать счастья там.

В течение недели я побывал на двух или трех интервью, но ни одна компания не хотела дать шанс бездомному парню, даже с опытом работы. Несколько месяцев такой жизни закончились тем, что мои друзья и абсолютно незнакомые люди попытались помочь мне встать на ноги, посылая деньги на краудфандинговую платформу GoFundMe. Тогда я ел примерно раз в день и редко что-то достойное. Моим единственным шансом выбраться из сложившейся ситуации было стать программистом.

В конце концов я решил сделать последний рывок. Я разослал резюме на все виды работ, для которых моя квалификация подходила хотя бы отдаленно. На следующий день небольшая стартап-компания пригласила меня на собеседование. Я сделал все, чтобы выглядеть прилично. Побрился, принял душ (непростая задача для бездомно-

го), надел чистую одежду, завязал волосы в хвост и пришел на интервью. Я говорил начистоту, объяснил свою ситуацию и рассказал, почему решил попытаться счастья в Остине. Во время собеседования я как мог старался показать, что, хотя сейчас у меня и трудные времена, мне можно дать шанс и я буду работать изо всех сил, дабы показать, что однажды я мог бы стать лучшим.

Я вышел с ощущением провала: мне казалось, что честность снизила мои шансы. Однако через полторы недели, когда я уже собирался полностью сдаться, меня позвали на повторное собеседование.

Когда я пришел, был только директор. Босс сказал, что его впечатлила моя искренность и что он хочет дать мне шанс. Он сказал, что у меня приличная база и я напоминаю ему ящик — прочный, но относительно пустой. Я кажусь ему достаточно сильным, чтобы справиться со всем, чем они меня нагрузят, а научусь я во время работы. И в конце он сказал, что к работе я приступлю 6 декабря.

Год спустя я живу в гораздо более комфортной квартире, чем жил до того, как стал программистом. Меня уважают коллеги и даже интересуются моим мнением по важным вопросам компании. Вы можете достичь чего угодно и стать кем угодно. Никогда не бойтесь пробовать, даже если шансы кажутся вам призрачными.

Тианни Майерс

Следующий — Тианни Майерс. Он прочитал мою книгу *The Self-Taught Programmer* и прислал мне на электронную почту свою историю о том, как самостоятельно осваивал профессию.

Мой путь самоучки начался на лекциях по веб-дизайну в колледже, когда я работал над получением степени бакалавра в области медиакоммуникаций. В то время мне нравилось писать тексты и я мечтал о работе в сфере маркетинга. Мои цели поменялись после решения научиться программировать. Я хочу поделиться своей историей самоучки и рассказать, как за 12 месяцев я прошел путь от кассира в магазине до младшего веб-разработчика.

Начал я с изучения основ HTML и CSS в Code Academy. Я написал свою первую программу на языке Python: это была простая игра с числами, когда компьютер выбирал случайное число, а у пользователя было три попытки, чтобы его угадать.

Мое утро начиналось в 4 часа утра с чашки кофе. Я проводил от 6 до 10 часов в день за чтением книг по программированию и написанием кода. Тогда мне был 21 год и я работал на полставки в Goodwill, чтобы сводить концы с концами. Я никогда не был счастливее, потому что большую часть дня занимался любимым делом — творил, используя языки программирования как инструмент.

Однажды я зарегистрировался на сайте по трудоустройству Indeed в поисках работы. Я не ожидал, что мне ответят, но через несколько дней мне написали из рекламного агентства. Я ответил на вопросы SQL-ассесмента на Indeed, затем последовало телефонное собеседование, ассесмент по коду — и вскоре после этого меня пригласили на очное собеседование. Директор по веб-разработке и два senior-разработчика просматривали мои ответы во время прохождения код-ассесмента. Я чувствовал себя достаточно уверенно, так как они были поражены некоторыми моими ответами и приятно удивились, узнав, что я самоучка. Они сказали, что какие-то из моих ответов были лучше, чем у senior-разработчиков, которым они давали те же самые задания. Две недели спустя они меня наняли.

Если вы захотите и приложите достаточно усилий, то вы сможете воплотить свои мечты в жизнь, как это сделал я.

Начало работы

Примеры кода в книге написаны на языке Python. Я выбрал этот язык программирования, так как он один из самых простых для чтения. В книге я иллюстрирую примеры кода следующим образом:

```
for i in range(100):  
    print("Hello, World!")
```

```
>> Hello, World!  
>> Hello, World!  
>> Hello, World!
```

Строка вида `#http://tinyurl.com/h4qntgk` содержит URL-адрес, который открывает веб-страницу с соответствующим кодом: при возникновении проблем с запуском кода вы можете легко его скопировать и вставить в текстовый редактор Python IDLE. Текст, следующий после `>>`, является выводом интерактивной оболочки Python. Многоточие после вывода (...) означает «и так далее». Если после примера нет символов `>>`, это значит, что либо программа не выдает никакого вывода, либо я просто объясняю принцип, а вывод не важен. Все, что в тексте выделено моноширинным шрифтом, является кодом или выводом.

Устанавливаем Python

Чтобы прорабатывать примеры из книги, вам необходимо установить Python версии 3. Загрузить Python для Windows или Unix можно по ссылке <http://python.org/downloads>. Если у вас операционная система Ubuntu, Python 3 установлен по умолчанию. Убедитесь, что вы загрузили Python 3, а не Python 2: некоторые примеры, приведенные в книге, не будут работать с Python 2.

Python доступен для 32-битных и 64-битных компьютеров. Если вы приобрели свой компьютер после 2007 года, скорее всего, он будет 64-битным. Если вы не уверены — поиск в интернете поможет вам это уточнить.

Если вы используете Windows или Mac, загрузите 32- или 64-битную версию Python, откройте файл и следуйте инструкциям. По ссылке <http://theselftaughtprogrammer.io/installpython> вы также можете посмотреть видео с объяснениями, как установить Python для разных операционных систем.

Устранение неполадок

При возникновении сложностей с установкой Python, пожалуйста, напишите сообщение в группу *Self-Taught Programmers* в Facebook: ее вы найдете по ссылке <https://facebook.com/groups/selftaughtprogrammers>. Когда в этой группе (или на любой другой площадке в интернете) вы размещаете код с просьбой о помощи, обязательно поместите его в GitHub Gist. Никогда не присылайте скриншоты кода. Если люди захотят помочь, им может понадобится запустить ваш код. Когда вы присылаете скриншот, приходится набирать код вручную, а если использовать сервис GitHub Gist, код можно быстро скопировать и вставить в свою IDE.

Практикумы

Многие главы заканчиваются упражнением. Эти практикумы сделаны для того, чтобы проверить, насколько хорошо вы поняли материал, улучшить навыки и помочь подготовиться к техническому интервью. Все решения предложенных задач вы найдете на GitHub по ссылке https://github.com/calthoff/tstcs_challenge_solutions.

Пока вы читаете книгу и решаете задачи, я призываю вас делиться своими успехами в сообществе самоучек, используя тег `#selftaughtcoder` в Twitter. Каждый раз, когда вы чувствуете, что достигли впечатляющего прогресса, выкладывайте мотивационный твит, используя тег `#selftaughtcoder`, чтобы вдохновить других членов сообщества. Не стесняйтесь упоминать в посте и меня: `@coryalthoff`.

Не бросайте начатое

Остался еще один вопрос, который я хотел бы обсудить до того, как вы погрузитесь в изучение computer science. Если вы читаете эту книгу, значит, уже знакомы с программированием. Как вы знаете, самое трудное в освоении нового навыка, в том числе и программирования, — не сложность материала, а непрерывная работа. Продолжать изучать новое — это вызов. Я с ним боролся годами,

пока не научился хитрости, которой сейчас поделюсь и с вами: она называется «Не разрывай цепь».

Методику продуктивности «Не разрывай цепь» изобрел Джерри Сайнфелд, когда оттачивал свое первое стендап-выступление. Для начала он повесил в комнате календарь. Затем, если ему удавалось написать шутку к концу дня, он ставил красный крестик в этом дне календаря (мне больше нравятся зеленые галочки). Вот и все. В этом и состоит фокус, и он невероятно действенный.

Как только у вас появится цепь (две и более зеленые галочки подряд), вам не захочется ее разрывать. Две зеленые отметки подряд превращаются в пять, затем в десять, затем в двадцать. Чем длиннее ваша цепь, тем сложнее ее разорвать. Представьте, что в конце месяца вы смотрите на календарь с 29 зелеными галочками. Вам нужна лишь одна, чтобы месяц стал идеальным. Вы обязательно выполните задачу в этот день. Или, как описывает сам Джерри Сайнфелд:


Через несколько дней у вас появляется цепь. Просто продолжайте в том же духе, и цепь начнет увеличиваться с каждым днем. Вам понравится наблюдать за ней, особенно когда позади у вас будет уже несколько недель. Следующая ваша задача — не разорвать цепь.

Однажды мое стремление сохранить целостность одной из цепочек привело меня к такому сумасшедшему поступку, как поход в спортзал посреди ночи. Нет лучшего чувства, чем то, что возникает при просмотре страницы календаря с вашим первым идеальным месяцем, в котором каждый день заполнен зелеными галочками. Если вы погрязли в рутине, вы всегда можете вернуться к этой странице и вспомнить месяц, когда все сделали правильно.

Технические книги трудно читать. Я уж и не помню, сколько из них я бросил недочитанными. Эту книгу я постарался сделать как можно более увлекательной и легкой для чтения, но, чтобы обзавестись дополнительными гарантиями, попробуйте применить метод «Не разрывай цепь». В сотрудничестве с [monday.com](https://hey.monday.com/CoryAlthoff) я создал для программистов-самоучек бесплатный шаблон и приложение, которое отслеживает написанный код, — опробовать его можно по ссылке <https://hey.monday.com/CoryAlthoff>.

С учетом всего вышесказанного вы готовы начать изучение computer science?

Приступим!



Введение в алгоритмы

Глава 1. Что такое алгоритм

Глава 2. Рекурсия

Глава 3. Поисковые алгоритмы

Глава 4. Алгоритмы сортировки

Глава 5. Строковые алгоритмы

Глава 6. Математика

Глава 7. Вдохновение самоучки: Маргарет Гамильтон

1

Что такое алгоритм

Независимо от того, желаете ли вы раскрыть тайны Вселенной или хотите построить успешную карьеру в XXI веке, один из важнейших навыков — основы компьютерного программирования.

Стивен Хокинг

Алгоритм — это последовательность шагов, которая решает задачу. Например, алгоритм приготовления яичницы-болтуньи (скрэмбла) следующий: разбить три яйца в миску, взбить их, разогреть сковородку на плите, вылить яйца на сковородку и перемешать, переложить яичницу на тарелку, как только она будет готова. Данный раздел книги полностью посвящен алгоритмам. Здесь вы узнаете, как решать задачи вроде поиска простых чисел. Кроме того, вы научитесь составлять новые, продуктивные алгоритмы с нуля, искать и сортировать данные.

В этой главе вы научитесь сравнивать два алгоритма, что поможет вам их анализировать. Программисту важно понимать, чем один алгоритм лучше другого, потому что программисты тратят большую часть своего времени на написание алгоритмов и выбор того, какие структуры данных с ними использовать. Если вы понятия не имеете, почему вам следует предпочесть один алгоритм другому, вы не станете профессионалом, поэтому данная глава имеет решающее значение.

Несмотря на то что алгоритмы являются фундаментальным понятием в информатике, ученые пока не пришли к единому определению. Существует несколько конкурирующих между собой формулировок, однако предложенная Дональдом

Эрвином Кнудом — одна из наиболее известных. Он описывает алгоритм как определенный, эффективный и конечный процесс, который получает входные данные и производит выходные данные на основе полученных входных.

Определенность означает, что шаги четкие, лаконичные и недвусмысленные.

Эффективность означает, что вы можете выполнить каждую операцию точно для решения задачи.

Конечность подразумевает, что алгоритм останавливается после определенного количества шагов.

Общепринятое дополнение к этому списку — *правильность*. Алгоритм должен всегда выдавать один и тот же результат для заданных входных данных, и полученный результат должен быть правильным ответом на задачу, которую решает алгоритм.

Не все, но большая часть алгоритмов удовлетворяют этим требованиям, однако некоторые исключения важны. Скажем, когда вы создаете генератор случайных чисел, ваша цель — сгенерировать случайность, поэтому нельзя использовать вывод, чтобы понять, каким был ввод. Кроме того, многие алгоритмы в data science не так строго придерживаются правильности. Например, алгоритму может быть достаточно лишь предположить вывод, если заранее известна его неопределенность. Однако в большинстве случаев ваш алгоритм должен соответствовать всем заявленным требованиям. Если вы пишете алгоритм по приготовлению яичницы-болтуньи, пользователь не обрадуется, если в результате получит омлет или вареные яйца.

Анализируем алгоритмы

Как правило, для решения задачи можно использовать более одного алгоритма. Например, существует несколько разных способов сортировки списка. Когда задачу могут решить несколько алгоритмов, как узнать, который из них лучше? Простейший? Быстрейший? Самый маленький? Или какой-то еще?

Один из способов оценить алгоритм — обратить внимание на время его выполнения. **Время выполнения** алгоритма — это количество необходимого вашему компьютеру времени, чтобы выполнить алгоритм, написанный на языке программирования. Например, ниже представлен алгоритм на языке Python, который считает от 1 до 5 и отображает каждое число:

```
for i in range(1, 6):  
    print(i)
```

Вы можете измерить время выполнения этого алгоритма, используя встроенный в Python модуль `time`. Данный модуль позволяет отследить, сколько времени требуется вашему компьютеру, чтобы выполнить алгоритм:

```
import time

start = time.time()
for i in range(1, 6):
    print(i)
end = time.time()
print(end - start)

>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.15141820907592773
```

Когда вы запускаете программу, она отображает числа от 1 до 5 и выводит значение времени, затраченного на выполнение. В данном случае потребовалось 0,15 секунды.

Теперь перезапустите программу:

```
import time

start = time.time()
for i in range(1, 6):
    print(i)
end = time.time()
print(end - start)

>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.14856505393981934
```

При втором запуске программы время ее выполнения будет иным. Следующий запуск опять же выдаст новое значение времени. Время выполнения алгоритма продолжит меняться, поскольку доступная производительность компьютера в процессе выполнения программы также меняется и влияет на время выполнения.

Более того, время выполнения алгоритма будет отличаться и на другом компьютере. Если вы запустите алгоритм на компьютере с меньшей вычислительной мощностью, он выполнится медленнее, а на более мощном

компьютере — быстрее. Кроме того, на время выполнения программы влияет и язык программирования, на котором она написана. Например, время выполнения будет короче, если вы запустите эту же программу на C, потому что C может быть быстрее, чем Python.

Поскольку на время выполнения алгоритма влияет множество различных переменных, таких как вычислительные мощности компьютера и язык программирования, оценка данного параметра — не очень эффективный способ сравнения двух алгоритмов. Вместо этого специалисты в области информатики сравнивают алгоритмы по количеству шагов, которые требуются для их выполнения. Вы можете ввести количество шагов в формулу для сравнения двух и более алгоритмов без учета языка программирования или мощности компьютера. Обратимся к примеру с той же программой, которая считает от 1 до 5:

```
for i in range(1, 6):  
    print(i)
```

Этой программе требуется пять шагов для завершения (она проходит цикл пять раз и выводит *i* каждый раз). Вы можете выразить количество шагов, необходимых вашему алгоритму, следующим уравнением:

$$f(n) = 5$$

Если вы усложните программу, ваше уравнение изменится. Например, вы можете захотеть отслеживать сумму всех выводимых цифр:

```
count = 0  
for i in range(1, 6):  
    print(i)  
    count += i
```

Теперь алгоритму требуется выполнить 11 шагов для завершения. Сначала он присваивает переменной *count* нулевое значение. Затем выводит пять чисел и пять раз инкрементируется ($1 + 5 + 5 = 11$).

Вот новое уравнение для вашего алгоритма:

$$f(n) = 11$$

Что произойдет, если вы замените 6 на переменную?

```
count = 0  
for i in range(1, n):  
    print(i)  
    count += i
```

Уравнение приобретет следующий вид:

$$f(n) = 1 + 2n$$

Теперь количество шагов алгоритма зависит от значения n . Цифра 1 в уравнении представляет первый шаг: `count = 0`. Затем следуют два шага с переменной n . Например, если $n = 5$, $f(n) = 1 + 2 \times 5$. Программисты называют переменную n в уравнении, которое описывает количество шагов в алгоритме, **размером задачи**. В данном случае вы можете сказать, что время на решение задачи размером n равно $1 + 2n$, или то же самое в математическом обозначении: $T(n) = 1 + 2n$.

Уравнение, описывающее количество шагов в данном алгоритме, не сильно помогает, так как, помимо всего прочего, вы не можете достоверно посчитать количество шагов в алгоритме. Например, если алгоритм содержит много условных операторов и вы не в состоянии заранее предугадать, какие из них будут исполнены. Хорошая новость в том, что для вас как для программиста не имеет значения точное количество шагов алгоритма — вам важно знать, как алгоритм поведет себя при увеличении переменной n . Большинство алгоритмов хорошо работают с малым объемом данных, но могут не справиться с большим. Даже самый неэффективный алгоритм будет хорошо работать при $n = 1$. Однако в реальности n , равное единице, встречается редко: значение переменной может составлять и несколько сотен тысяч, и миллион, и даже более.

Что действительно необходимо знать об алгоритме — не точное, а, скорее, приблизительное количество шагов, которое потребуется при увеличении n . Когда n возрастает, одна часть уравнения затмевает другую до такой степени, что все остальное становится просто неактуальным. Посмотрите на следующий код Python:

```
def print_it(n):
    # цикл 1
    for i in range(n):
        print(i)
    # цикл 2
    for i in range(n):
        print(i)
        for j in range(n):
            print(j)
            for h in range(n):
                print(h)
```

Какая часть этой программы наиболее важна для определения количества шагов, необходимых алгоритму для завершения? Вы можете предположить, что обе части функции (первый цикл и второй цикл, содержащий другие циклы) важны. В конце концов, если n равно 10 000, ваш компьютер выведет много чисел в обоих циклах.

Получается, что данный код не подходит, если говорить об эффективности алгоритма:

```
# цикл 1
for i in range(n):
    print(i)
```

Чтобы понять, почему это так, вам нужно посмотреть, что произойдет при увеличении n . Ниже приведено уравнение для вычисления количества шагов вашего алгоритма:

$$T(n) = n + n^2$$

Когда у вас два вложенных цикла `for` с количеством шагов n , они преобразуются в n^2 (n во второй степени), потому что если n равно 10, то вам необходимо выполнить 10 шагов дважды, или 10^2 . Три вложенных цикла `for` всегда преобразуются в n^3 по той же причине. В данном уравнении, когда n равно 10, первому циклу вашей программы потребуется 10 шагов, а второму циклу потребуется 10^3 шагов, что равно 1000. Когда n равно 1000, первому циклу требуется 1000 шагов, а второму циклу необходимо 1000^3 , что равно 1 миллиарду.

Видите, что происходит? По мере роста n вторая часть вашего алгоритма растет намного быстрее, поэтому первая часть становится неактуальной. Например, если вам необходимо, чтобы программа работала для 100 000 000 записей базы данных, вас не будет волновать количество шагов в первой части, так как шаги во второй части будут расти в геометрической прогрессии. При наличии 100 000 000 записей вторая часть алгоритма будет состоять из более чем септиллиона шагов, то есть 1 с 24 нулями, поэтому использовать данный алгоритм не имеет смысла. Первые 100 000 000 шагов нерелевантны для вас при принятии решения.

Поскольку важной частью алгоритма является та часть, которая растет быстрее по мере увеличения n , для описания эффективности алгоритма программисты вместо равенства $T(n)$ используют нотацию «О большое». **Нотация «О большое»** — это математическое обозначение, описывающее, как по мере роста n возрастают требования алгоритма к времени и объему памяти (о требованиях к объему вы узнаете позднее).

Программисты используют нотацию «О большое» для создания функции порядка величины $T(n)$. **Порядок величины** — тип объекта в системе классификации, в которой каждый тип во много раз больше или меньше предыдущего. В функции порядка величины вы используете ту часть $T(n)$, которая преобладает

в уравнении, и игнорируете все остальное. Часть $T(n)$, преобладающая в уравнении, — это порядок величины алгоритма. Ниже представлены общепринятые классификации порядка величины для нотации «О большое»: от наилучшего (наиболее эффективного) до наихудшего (наименее эффективного):

- постоянное время;
- логарифмическое время;
- линейное время;
- линейно-логарифмическое время;
- квадратичное время;
- кубическое время;
- экспоненциальное время.

Каждый порядок величины описывает временную сложность алгоритма. **Временная сложность** — это максимальное количество шагов, требующихся алгоритму для завершения по мере увеличения n .

Рассмотрим подробнее каждый порядок величины.

Постоянное время

Наиболее эффективным порядком величины является *постоянная временная сложность*. Алгоритм выполняется за **постоянное время**, когда ему требуется одно и то же количество шагов вне зависимости от объема задачи. Нотация «О большое» для постоянной сложности — $O(1)$.

Допустим, у вас онлайн-магазин книг и каждый день вы дарите книгу первому покупателю. Вы храните своих покупателей в списке `customers`. Алгоритм может выглядеть следующим образом:

```
free_books = customers[0]
```

А равенство $T(n)$ выглядит так:

$$T(n) = 1$$

Вашему алгоритму требуется один шаг независимо от количества покупателей. Если у вас 1000 покупателей, алгоритму необходим один шаг. Если у вас 10 000 покупателей, вашему алгоритму все еще требуется один шаг, и даже если у вас триллион покупателей, все равно алгоритм завершится в один шаг.

Если изобразить на графике постоянную временную сложность, где по оси X отображается количество покупателей, по оси Y — количество шагов, линия на графике будет ровной (рис. 1.1).



Рис. 1.1. Постоянная временная сложность

Как вы можете видеть, количество шагов, необходимых вашему алгоритму для завершения, не увеличивается по мере роста объема задачи. Таким образом, это и есть наиболее эффективный алгоритм, который можно написать, потому что его время не меняется с увеличением набора данных.

Логарифмическое время

Логарифмическое время — вторая по эффективности временная сложность. Алгоритм выполняется за **логарифмическое время**, когда время выполнения программы растет пропорционально логарифму размера входных данных. Подобную временную сложность можно увидеть в таких алгоритмах, как двоичный поиск, который может не учитывать множество значений в каждом цикле. (Если сейчас вам что-то непонятно, не переживайте — позднее мы обсудим это более подробно.) Логарифмический алгоритм в нотации « O большое» выражается как $O(\log n)$.

Рисунок 1.2 показывает, как на графике выглядит логарифмический алгоритм.

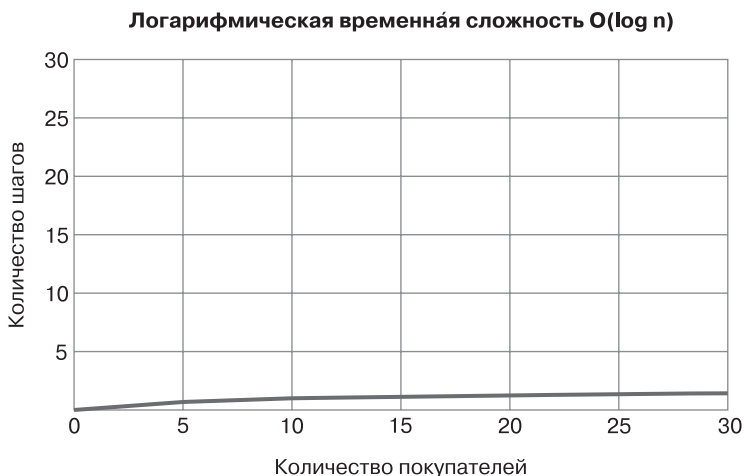


Рис. 1.2. Логарифмическая временная сложность

В логарифмическом алгоритме количество необходимых шагов по мере увеличения набора данных растет достаточно медленно.

Линейное время

Следующий по эффективности тип алгоритма — тот, который выполняется за **линейное время**. Такой алгоритм растет пропорционально росту размера задачи. Линейный алгоритм в нотации «О большое» выражается как $O(n)$.

Предположим, вам нужно изменить программу дарения бесплатных книг таким образом, чтобы дарить одну книгу в день не первому покупателю, а тому, чье имя начинается на букву *B*. Однако в данном случае список ваших покупателей составлен не в алфавитном порядке. Вы вынуждены перебирать весь список и просматривать каждое имя, чтобы найти то, которое начинается на *B*.

```
free_book = False
customers = ["Lexi", "Britney", "Danny", "Bobbi", "Chris"]
for customer in customers:
    if customer[0] == 'B':
        print(customer)
```

Когда список покупателей состоит из пяти позиций, программе для завершения требуется осуществить пять шагов. Для списка из 10 покупателей программе потребуется 10 шагов, для 20 покупателей — 20 шагов и т. д.

Ниже приведено уравнение для временной сложности данной программы:

$$f(n) = 1 + 1 + n$$

А в нотации «О большое» вы можете не учитывать константы и сфокусироваться на части, преобладающей в уравнении:

$$O(n) = n$$

В линейном алгоритме по мере роста n количество шагов алгоритма увеличивается на то же количество, что и n (рис. 1.3).

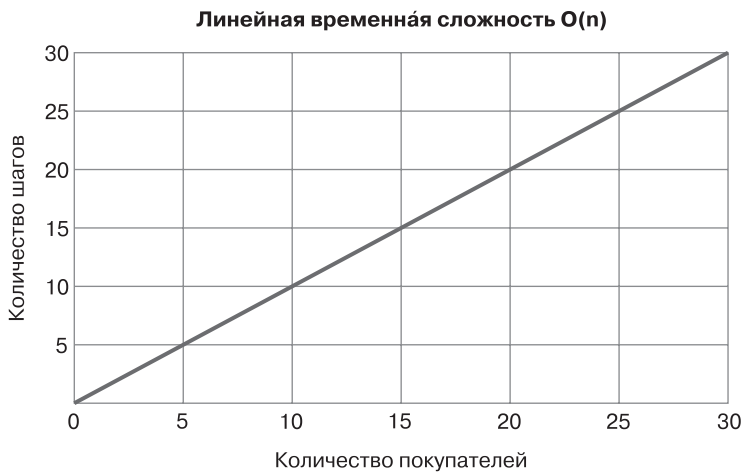


Рис. 1.3. Линейная временная сложность

Линейно-логарифмическое время

Алгоритм, выполняющийся за **линейно-логарифмическое время**, растет как сочетание (умножение) логарифмических и линейных временных сложностей. Например, линейно-логарифмический алгоритм может вычислять операцию $O(\log n)$ n раз. В нотации «О большое» линейно-логарифмический алгоритм выражается как $O(n \log n)$. Линейно-логарифмические алгоритмы часто разделяют набор данных на меньшие части и каждую из них обрабатывают по отдельности. Например, большинство наиболее эффективных алгоритмов сортировки, с которыми вы познакомитесь далее, такие как сортировка слиянием, являются линейно-логарифмическими.

На рис. 1.4 показано, как выглядит линейно-логарифмический алгоритм на графике.

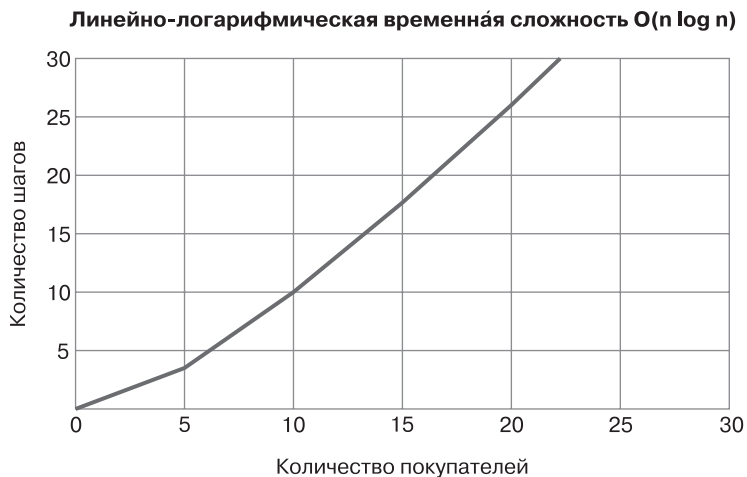


Рис. 1.4. Линейно-логарифмическая временная сложность

Как вы уже поняли, линейно-логарифмическая временная сложность не такая эффективная, как линейная. Однако эта сложность не растет так быстро, как квадратичная, которую вы изучите следующей.

Квадратичное время

Следующим по эффективности после линейно-логарифмического идет квадратичное время. Алгоритм выполняется за **квадратичное время**, когда его производительность прямо пропорциональна размеру задачи в квадрате. В нотации «О большое» квадратичный алгоритм выражается как $O(n^2)$.

Ниже приведен пример квадратичной временной сложности.

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        x = i * j
        print(x)
```

Алгоритм умножает каждое число из списка чисел на каждое следующее число, сохраняет результат в переменной и выводит его на экран.

В данном случае n — это размер вашего списка `numbers`. Уравнение для временной сложности алгоритма следующее:

$$f(n) = 1 + n * n * (1 + 1)$$

В этом уравнении $(1 + 1)$ является результатом умножения и вывода. Вы повторяете умножение и вывод $n * n$ раз с двумя вложенными циклами `for`. Можно упростить уравнение:

$$f(n) = 1 + (1 + 1) * n^{**2}$$

Что идентично следующему:

$$f(n) = 1 + 2 * n^{**2}$$

Как вы уже могли догадаться, преобладает часть уравнения n^{**2} , поэтому в нотации «О большое» равенство будет таким:

$$O(n) = n^{**2}$$

На графике алгоритма с квадратичной временной сложностью количество шагов резко увеличивается по мере увеличения размера задачи (рис. 1.5).

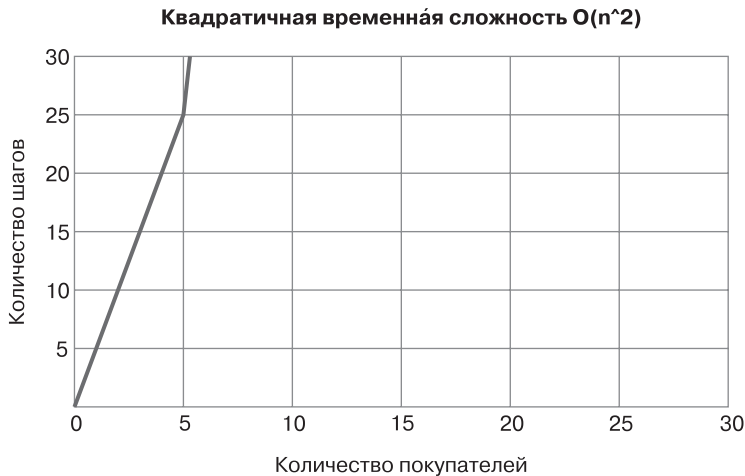


Рис. 1.5. Квадратичная временная сложность

Как правило, если ваш алгоритм содержит два вложенных цикла, выполняемых от 1 до n (или от 0 до $n - 1$), временная сложность будет как минимум $O(n^{**2})$. Большинство алгоритмов сортировки, такие как сортировка методом вставок или пузырьковая сортировка (о них вы узнаете позднее), выполняются за квадратичное время.

Кубическое время

За квадратичной временной сложностью следует кубическая. Алгоритм выполняется за **кубическое время**, когда производительность прямо пропорциональна размеру задачи в кубе. В нотации «О большое» вы выражаете кубический алгоритм как $O(n^3)$. Алгоритм с кубической сложностью похож на квадратичный, только n возводится в третью степень, а не во вторую.

Ниже представлен алгоритм с кубической временной сложностью:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        for h in numbers:
            x = i + j + h
            print(x)
```

Уравнение для этого алгоритма следующее:

$$f(n) = 1 + n * n * n * (1 + 1)$$

Или же:

$$f(n) = 1 + 2 * n^3$$

Как и в алгоритме с квадратичной сложностью, наиболее важная часть данного уравнения — n^3 , которая растет так быстро, что остальная часть уравнения, даже та, что включает в себя n^2 , становится нерелевантной. Таким образом, в нотации «О большое» кубическая сложность выглядит так:

$$O(n) = n^3$$

Два вложенных цикла свидетельствуют о квадратичной временной сложности, а три вложенных цикла, выполняемых от 0 до n , — признак алгоритма с кубическим временем. Весьма вероятно, что вы столкнетесь с кубической временной сложностью, если ваша работа связана с data science или статистикой.

И квадратичная, и кубическая временные сложности представляют собой частные случаи полиномиальной временной сложности. Алгоритм, который выполняется за **полиномиальное время**, вычисляется как $O(n^a)$, где $a = 2$ для квадратичного времени и $a = 3$ для кубического времени. При создании алгоритма, как правило, пытаются избежать полиномиального масштабирования, потому что алгоритм может работать очень медленно при увеличении n . Иногда избежать полиномиального масштабирования не получается, но в таком случае вы можете успокаивать себя тем, что полиномиальная сложность — еще не самый худший вариант.

Экспоненциальное время

Гордый титул «наихудшая временная сложность» по праву принадлежит экспоненциальной. Алгоритм, который выполняется за **экспоненциальное время**, содержит константу, увеличенную до размеров задачи. Другими словами, алгоритму с экспоненциальным временем для завершения требуется s шагов, возведенных в степень n . Нотация «О большое» для экспоненциального времени равна $O(c^{**n})$, где c — это константа. Значение константы не играет роли. Важно лишь то, что n находится в экспоненте.

К счастью, экспоненциальная сложность встречается не так часто. Одним из примеров данной сложности, связанной с попыткой угадать числовой пароль из n -го количества десятичных знаков путем проверки всех возможных комбинаций, является $O(10^{**n})$.

Ниже приведен пример угадывания пароля со сложностью $O(10^{**n})$:

```
pin = 931
n = len(pin)
for i in range(10**n):
    if i == pin:
        print(i)
```

Количество шагов, необходимых этому алгоритму для завершения, растет с невероятной скоростью по мере увеличения n . Когда n равно 1, алгоритму требуется 10 шагов. Когда n равно 2, необходимо 100 шагов. Когда n равно 3, будет затрачено 1000 шагов. Как вы можете заметить, сначала кажется, что экспоненциальный алгоритм растет не так уж и быстро. Но в конце концов рост становится взрывным. Для угадывания пароля из 8 десятичных цифр потребуется 100 миллионов шагов, а чтобы угадать пароль из 10 десятичных знаков, придется выполнить более 10 миллиардов шагов. Экспоненциальное масштабирование позволяет понять, почему так важно создавать длинные пароли. Если кто-то попытается отгадать ваш пароль с помощью подобной программы, он с легкостью это сделает для пароля из четырех знаков. Однако пароль из 20 знаков взломать невозможно, потому что программа будет выполняться дольше, чем человек живет на свете.

Отгадывание пароля — это пример алгоритма **полного перебора**, проверяющего все возможные варианты. Алгоритмы полного перебора, как правило, неэффективны, и выбирать их следует только в крайнем случае.

На рис. 1.6 представлено сравнение эффективности рассмотренных видов алгоритмов.

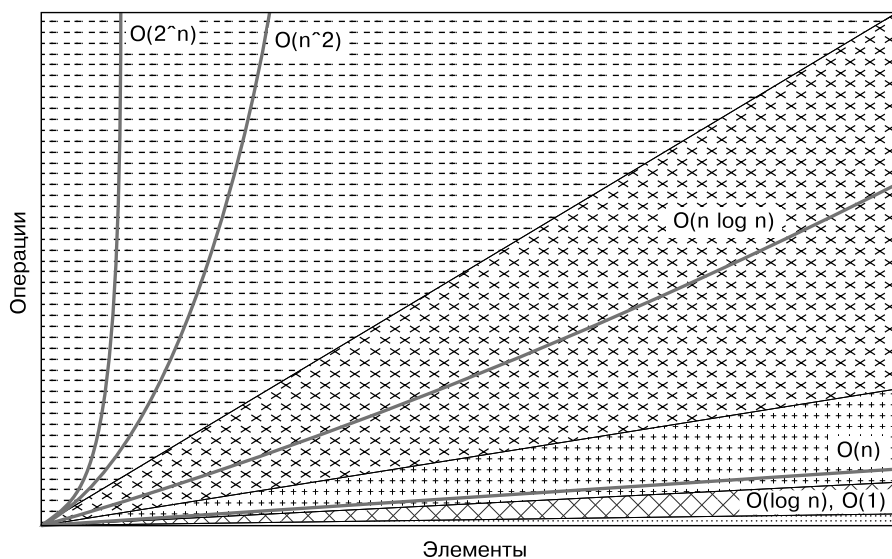


Рис. 1.6. График сложности O большого

Сложность в лучшем и худшем случаях

Производительность алгоритма зависит от различных факторов, в том числе и от типа данных. Поэтому при оценке производительности алгоритма необходимо учитывать его сложность в лучшем, худшем и среднем вариантах.

Сложность алгоритма в лучшем случае — это то, как он работает с идеальными входными данными, **сложность в худшем случае** — его работа в наихудшем из всех возможных сценариев. **Средняя сложность** алгоритма — то, как он работает в среднем.

Например, во время поиска с просмотром каждой позиции списка вам может повезти и вы найдете искомое в первом же элементе списка: это будет наилучший вариант сложности. Однако, если нужного элемента вообще нет в списке, вам придется проверить весь список — и это худший вариант сложности.

Если вам потребуется просмотреть список сотню раз, то в среднем вы найдете то, что ищете, за время $O(n/2)$, идентичное $O(n)$ в нотации « O большое». При сравнении алгоритмов часто начинают с рассмотрения среднего варианта сложности. При проведении более глубокого анализа можно также сравнивать сложности в лучшем и худшем вариантах.

Пространственная сложность

У каждого компьютера есть ограниченные ресурсы, такие как память, поэтому важно учитывать не только временную сложность алгоритма, но и потребление ресурсов. **Пространственная сложность** — это объем необходимой алгоритму памяти, включающий в себя фиксированное пространство, пространство структуры данных и временное пространство. **Фиксированное пространство** — это объем памяти, который требуется программе. **Пространство структуры данных** представляет собой объем памяти, необходимый программе для хранения набора данных, например, размер списка, в котором ведется поиск. Объем памяти, используемый вашим алгоритмом для хранения данных, зависит от объема входных данных конкретной задачи. **Временное пространство** — это объем памяти, необходимый алгоритму для промежуточной обработки: например, если алгоритму нужно временно скопировать список для переноса данных.

К пространственной сложности применимы все изученные ранее понятия временной сложности. Например, можно вычислить факториал n (произведение всех положительных целых чисел меньше или равных n), используя алгоритм с постоянной пространственной сложностью $O(1)$:

```
x = 1
n = 5
for i in range(1, n + 1):
    x = x * i
```

Пространственная сложность является константой, так как объем памяти, необходимый алгоритму, не увеличивается по мере увеличения n . Если вы решили сохранить все факториалы до n в списке, у вашего алгоритма будет линейная пространственная сложность $O(n)$:

```
x = 1
n = 5
a_list = []
for i in range(1, n + 1):
    a_list.append(x)
    x = x * i
```

Пространственная сложность алгоритма — $O(n)$, поскольку объем памяти, используемый алгоритмом, увеличивается в той же степени, что и n .

Как и в случае с временной сложностью, допустимый уровень пространственной сложности зависит от ситуации. В целом чем меньше памяти требуется алгоритму, тем лучше.

Почему все это важно

Программисту важно понимать различные порядки величины, чтобы оптимизировать алгоритм. Когда вы пытаетесь улучшить алгоритм, нужно сфокусироваться на изменении порядка величины, а не на других способах его улучшения. Предположим, у вас есть алгоритм $O(n^2)$ с двумя циклами `for`. Вместо оптимизации происходящих внутри циклов процессов постарайтесь переписать алгоритм так, чтобы у него не было двух вложенных циклов `for` и, соответственно, был меньший порядок величины.

Если вы можете решить задачу, написав алгоритм с двумя невложенными циклами `for`, у вас получится алгоритм $O(n)$ и производительность существенно изменится. Подобное изменение гораздо сильнее влияет на производительность, чем любые попытки подстройки и изменения алгоритма $O(n^2)$. Тем не менее имеет смысл подумать как о наилучших, так и о наихудших случаях сценариев. Возможно, у вас алгоритм $O(n^2)$, но в лучшем случае он принимает вид алгоритма $O(n)$ и ваши данные соответствуют этому сценарию. Тогда выбор такого алгоритма обоснован.

Решения, которые вы принимаете в отношении алгоритмов, могут иметь значительные последствия в реальном мире. Предположим, в качестве веб-разработчика вы ответственны за написание алгоритма для ответов на интернет-запросы покупателей. Ваш выбор между константным и квадратичным алгоритмом может повлиять на время загрузки сайта: в одном случае сайт будет грузиться менее чем за одну секунду и покупатель будет доволен, или же сайту потребуется больше минуты, из-за чего клиент может быть потерян еще до того, как отправит запрос.

Словарь терминов

Алгоритм — последовательность шагов, которая решает задачу.

Время выполнения — объем времени, необходимый вашему компьютеру для выполнения алгоритма, написанного на языке программирования, например Python.

Объем задачи — переменная n в уравнении, описывающем количество шагов алгоритма.

Нотация «О большое» — математическое обозначение того, как возрастают требования алгоритма к времени или объему памяти по мере роста n .

Порядок величины — тип объекта в системе классификации, в которой каждый тип во много раз больше или меньше предыдущего.

Временная сложность — максимальное количество шагов, необходимых алгоритму для завершения по мере увеличения n . Временная сложность оценивается по времени выполнения алгоритма.

Постоянное время — алгоритм выполняется за постоянное время, когда ему требуется одно и то же количество шагов, вне зависимости от объема задачи.

Логарифмическое время — алгоритм выполняется за логарифмическое время, когда время выполнения программы растет пропорционально логарифму размера входных данных.

Линейное время — алгоритм выполняется за линейное время, которое растет пропорционально росту размера задачи.

Линейно-логарифмическое время — алгоритм выполняется за линейно-логарифмическое время, если оно растет как сочетание (умножение) логарифмических и линейных временных сложностей.

Квадратичное время — алгоритм выполняется за квадратичное время, когда производительность алгоритма прямо пропорциональна размеру задачи в квадрате.

Кубическое время — алгоритм выполняется за кубическое время, когда производительность алгоритма прямо пропорциональна размеру задачи в кубе.

Полиномиальное время — алгоритм выполняется за полиномиальное время, которое вычисляется как $O(n^{**a})$, где $a = 2$ для квадратичного времени и $a = 3$ для кубического времени.

Экспоненциальное время — алгоритм выполняется за экспоненциальное время, когда содержит константу, увеличенную до размеров задачи.

Алгоритм полного перебора — тип алгоритма с проверкой всех возможных вариантов.

Сложность в лучшем случае — то, как алгоритм работает с идеальными входными данными.

Сложность в худшем случае — то, как алгоритм работает в наихудшем из всех возможных сценариев.

Средняя сложность — то, как алгоритм работает в среднем.

Пространственная сложность — объем памяти, который требуется алгоритму.

Фиксированное пространство — объем памяти, необходимый программе.

Пространство структуры данных — объем памяти, нужный для хранения набора данных.

Временное пространство — объем памяти, который требуется алгоритму для промежуточной обработки, например в случае временного копирования списка для переноса данных.

Практикум

1. Найдите какую-нибудь программу, которую вы написали ранее. Просмотрите ее и выпишите все временные сложности для разных алгоритмов.

2

Рекурсия

Чтобы понять рекурсию, необходимо для начала понять рекурсию.

Аноним

Итеративный алгоритм решает задачи повтором шагов снова и снова, используя, как правило, цикл. Большинство алгоритмов, которые вы до сих пор писали, наверняка были итеративными. **Рекурсия** — метод решения задачи, когда вы решаете более мелкие экземпляры задачи, пока не придете к решению. Рекурсивные алгоритмы опираются на функции, вызывающие сами себя. Любую задачу, которую можно решить с помощью итеративного алгоритма, можно решить и с помощью рекурсивного. Однако иногда рекурсивный алгоритм является более элегантным решением.

Рекурсивный алгоритм пишется внутри функции или метода, которые вызывают сами себя. Код внутри функции меняет входные данные и передает новые, отличные от предыдущих, когда в следующий раз функция вызывает саму себя. По этой причине у функции должна присутствовать **терминальная ветвь** — условие, которое завершает рекурсивный алгоритм, останавливая его навсегда. Каждый раз, когда функция вызывает саму себя, она приближается к терминальной ветви. В конце концов условие терминальной ветви удовлетворяется, то есть задача решена, и функция перестает вызывать саму себя. Алгоритм, следующий этим правилам, удовлетворяет трем законам рекурсии.

- У рекурсивного алгоритма должны быть как минимум одна рекурсивная и одна терминальная ветвь.

- Рекурсивный алгоритм должен менять свое состояние и стремиться к терминальной ветви.
- Рекурсивный алгоритм должен вызывать себя рекурсивно.

Чтобы помочь вам понять принцип работы рекурсивного алгоритма, рассмотрим задачу нахождения факториала числа с помощью рекурсивного и итеративного алгоритмов. **Факториал** числа — это произведение всех положительных целых чисел, меньших или равных числу. Например, факториал числа 5 равен $5 \times 4 \times 3 \times 2 \times 1$.

$$5! = 5 * 4 * 3 * 2 * 1$$

Ниже представлен итеративный алгоритм, который вычисляет факториал числа n :

```
def factorial(n):
    the_product = 1
    while n > 0:
        the_product *= n
        n = n - 1
    return the_product
```

Функция `factorial` принимает число n , которое вы используете в своих вычислениях.

```
def factorial(n):
```

Внутри функции вы объявляете переменную `the_product` и устанавливаете ей значение 1. Вы используете `the_product` для отслеживания результата при умножении n на предшествующие ему числа, например $5 * 4 * 3 * 2 * 1$.

Далее вы используете цикл `while` для итерации в обратном направлении от n до 1, отслеживая результат.

```
while n > 0:
    the_product *= n
    n = n - 1
```

В конце цикла `while` вы возвращаете `the_product`, которая содержит факториал n .

```
return the_product
```

А вот как написать этот же алгоритм рекурсивно:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Сначала вы объявляете функцию `factorial`, которая принимает число `n` в качестве параметра. Далее следует ваша терминальная ветвь. Функция будет возвращать себя до тех пор, пока `n` не станет равным 0, после чего она вернет число 1 и перестанет вызывать себя.

```
if n == 0:  
    return 1
```

Каждый раз, когда не удовлетворяются условия терминальной ветви, выполняется рекурсивная ветвь, то есть эта строка кода:

```
return n * factorial(n - 1)
```

Как видите, ваш код вызывает функцию `factorial`, которая является самой собой. Если вы впервые видите рекурсивный алгоритм, он может показаться вам странным. Вы можете даже подумать, что такой код не будет работать. Но поверьте, он работает. В данном случае функция `factorial` вызывает саму себя и возвращает результат. Однако она не вызывает саму себя со значением `n`; точнее, она вызывает себя со значением `n - 1`. В конечном итоге `n` будет меньше 1, что удовлетворит условие терминальной ветви:

```
if n == 0:  
    return 1
```

Вот весь код, который вам необходимо написать для рекурсивного алгоритма, состоящего всего лишь из четырех строк:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Как же это работает? Каждый раз, когда функция встречает оператор возврата, она помещает его в стек¹. Стек — тип структуры данных, о котором вы узнаете больше во второй части книги. Это как список в Python, но вы удаляете элементы в том же порядке, в котором их добавляли. Допустим, вы вызываете свою рекурсивную функцию `factorial` следующим образом:

```
factorial(3)
```

Переменная `n` начинается с 3. Функция проверяет условие терминальной ветви. Возвращается значение `False`, поэтому Python выполняет строку кода рекурсивной ветви:

```
return n * factorial(n - 1)
```

¹ Так называемый стек вызовов. — *Примеч. науч. ред.*

Python пока не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

Внутренний стек (не выполняйте этот код)

```
# n = 3
[return n * factorial(n - 1)]
```

Затем функция вызывает саму себя после уменьшения `n` на 1:

```
factorial(2)
```

Функция опять проверяет условие терминальной ветви, и вновь возвращается значение `False`, поэтому Python снова идет по рекурсивной ветви:

```
return n * factorial(n - 1)
```

Python все еще не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

Внутренний стек

```
# n = 3                                # n = 2
[return n * factorial(n - 1), return n * factorial(n - 1),]
```

Снова ваша функция вызывает саму себя после уменьшения `n` на 1:

```
factorial(1)
```

Python опять не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

Внутренний стек

```
# n = 3
[return n * factorial(n - 1),
# n = 2
return n * factorial(n - 1),
# n = 1
return n * factorial(n - 1),]
```

Вновь ваша функция вызывает саму себя после уменьшения `n` на 1, но на этот раз `n` равно 0, а это значит, что условие терминальной ветви удовлетворено и вы возвращаете число 1.

```
if n == 0:
    return 1
```

Python снова помещает возвращаемое значение в стек, но на этот раз он знает, что возвращает: число 1. Теперь внутренний стек Python выглядит следующим образом:


```
# Внутренний стек
```

```
# n = 3
[return n * factorial(n - 1),
# n = 2
return n * factorial(n - 1),
# n = 1
return n * factorial(n - 1), 1]
```

Поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата и удалить его из стека. Другими словами, Python умножает $1 * n$, а n равно 1.

```
1 * 1 = 1
```

Теперь внутренний стек Python выглядит так:

```
# Внутренний стек
```

```
# n = 3                                # n = 2
[return n * factorial(n - 1), return n * factorial(n - 1), 1]
```

И снова, поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата и удалить его из стека:

```
2 * 1 = 2
```

В этот момент внутренний стек Python выглядит так:

```
# Внутренний стек
```

```
# n = 3
[return n * factorial(n - 1), 2]
```

В конечном счете, поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата, удалить этот результат из стека и вернуть ответ.

```
3 * 2 = 6
```

```
# Внутренний стек
```

```
[return 6]
```

Теперь вы видите, что вычисление факториала числа — прекрасный пример задачи, которую можно решить, находя решения для меньших экземпляров одной и той же задачи. Осознав это и написав рекурсивный алгоритм, вы создадите элегантное решение для вычисления факториала числа.

Когда использовать рекурсию

Как часто использовать рекурсию в алгоритме, зависит только от вас. Любой алгоритм, написанный с помощью рекурсии, можно также написать с помощью итерации. Основное преимущество рекурсии — ее изящество. Как вы видели ранее, итеративное решение для вычисления факториалов заняло шесть строк кода, в то время как для рекурсивного решения потребовалось лишь четыре строки. Недостаток рекурсивных алгоритмов в том, что обычно они занимают больше памяти, так как им нужно хранить данные во внутреннем стеке Python. Рекурсивные функции по сравнению с итеративными может быть труднее считать и отлаживать, поскольку отследить, что происходит в рекурсивном алгоритме, сложнее.

Будете ли вы использовать рекурсию для решения задачи, зависит от специфики ситуации — например, от того, насколько важен задействованный объем памяти по сравнению с изяществом применения рекурсивного алгоритма, а не итерационного. Далее в книге вы встретите больше примеров, когда рекурсия предлагает более красивое решение, как в случае обхода двоичного дерева.

Словарь терминов

Итеративный алгоритм — алгоритм, решающий задачи повтором шагов снова и снова, обычно с помощью цикла.

Рекурсия — метод решения задачи, при котором результат зависит от решений меньших экземпляров одной и той же задачи.

Терминальная ветвь — условие, при котором рекурсивный алгоритм завершается, чтобы он не продолжался вечно.

Факториал — произведение всех положительных целых чисел, меньших или равных числу.

Практикум

1. Выведите числа от 1 до 10 рекурсивно.

3

Поисковые алгоритмы

Чтобы поверить в алгоритм, его нужно увидеть.

Дональд Кнут

Как профессиональный программист, вы будете тратить много времени на работу с данными. Если вы станете разработчиком веб-сайтов или приложений, вам придется отвечать за отображение данных для пользователей, которые обращаются к сайту или приложению, — а данными часто приходится манипулировать, прежде чем выводить их на экран пользователя. Если вы станете специалистом по данным (data scientist), то будете тратить еще больше времени, работая с данными. Может быть, компания Netflix наймет вас, чтобы вы использовали ее данные для улучшения алгоритма рекомендации фильмов. Или создатели какой-то соцсети возьмут вас на работу для анализа ее данных, желая дольше удерживать пользователей на платформе.

Одна из самых фундаментальных вещей, которую должен знать программист, работающий с данными, — это как их искать. Программисты ищут данные с помощью **поискового алгоритма**: он находит определенные данные в наборе данных. **Набор данных** — это совокупность собранной информации. Два наиболее распространенных примера поискового алгоритма — линейный и двоичный поиск. Как программист, вы, скорее всего, не будете тратить много времени на разработку поисковых алгоритмов, поскольку они уже встроены в языки программирования, например в Python. Тем не менее, если вы научитесь писать коды для поисковых алгоритмов, то сделаете большой шаг вперед, так как это поможет вам лучше понимать основополагающие понятия в информатике — линейный

и логарифмический порядки величины. Понимать, как эти алгоритмы работают с разными наборами данных, для вас критически важно, чтобы не ошибиться в выборе встроенного в Python поискового алгоритма.

В этой главе вы узнаете, как искать число в списке с помощью алгоритмов линейного поиска и двоичного поиска. А после того, как вы самостоятельно напишете каждый из алгоритмов, я покажу вам, как выполнить тот же поиск с помощью встроенных в Python инструментов.

Линейный поиск

В **линейном поиске** вы перебираете каждый элемент в наборе данных и сопоставляете его с целевым числом. Если ваше сравнение находит совпадение — число находится в списке. Если алгоритм завершается, не найдя совпадения, — числа в списке нет.

Ниже приведен алгоритм линейного поиска на языке Python:

```
def linear_search(a_list, n):
    for i in a_list:
        if i == n:
            return True
    return False

a_list = [1, 8, 32, 91, 5, 15, 9, 100, 3]
print(linear_search(a_list, 91))

>> True
```

Первая часть программы вызывает функцию `linear_search` и передает список и число для поиска — `n`:

```
a_list = [1, 8, 32, 91, 5, 15, 9, 100, 3]
print(linear_search(a_list, 91))
```

В данном случае `n = 91`, поэтому алгоритм будет проверять, есть ли 91 в `a_list`. Затем используется цикл `for` для перебора каждого элемента в `a_list`:

```
for i in a_list:
```

Оператор `if` позволяет сравнить каждый элемент в `a_list` с `n`:

```
if i == n:
```

Если совпадение найдено, возвращается значение `True`. Если весь список пройден и совпадения не нашлось, возвращается значение `False`:

```
for i in a_list:
    if i == n:
        return True
return False
```

Когда вы запускаете программу, она возвращает значение `True`, так как число `n` (в данном примере 91) есть в `a_list`:

```
print(linear_search(a_list, 91))
```

```
>> True
```

Если вы перезапустите программу, но будете искать 1003 вместо 91, ваша программа вернет значение `False`, так как 1003 нет в `a_list`:

```
print(linear_search(a_list, 1003))
```

```
>> False
```

Когда следует использовать линейный поиск

Временная сложность линейного поиска — $O(n)$. Если у вас список из десяти элементов, в худшем случае сценария алгоритму потребуется десять шагов. Лучший случай сценария для линейного поиска — $O(1)$, так как искомый элемент может быть первым в списке, и вашему алгоритму потребуется всего один шаг: найдя совпадение, алгоритм остановится. В среднем линейному поиску требуется $n/2$ шагов.

Вам следует рассмотреть вариант использования линейного поиска, если ваши данные не отсортированы. **Отсортированные данные** — данные, упорядоченные определенным образом. Например, вы можете отсортировать список чисел последовательно (либо по возрастанию, либо по убыванию):

```
# Несортированный список
the_list = [12, 19, 13, 15, 14, 10, 18]

# Список, отсортированный по возрастанию
the_list = [10, 12, 13, 14, 15, 18, 19]
```

Если данные отсортированы, можно использовать более эффективный двоичный поиск, о котором вы скоро узнаете.

Работая над реальными задачами, вместо написания собственного алгоритма линейного поиска вы можете использовать встроенное в Python ключевое

слово `in`. Ниже приведен пример, как выполнять линейный поиск по списку чисел, используя встроенное в Python ключевое слово `in`:

```
unsorted_list = [1, 45, 4, 32, 3]
print(45 in unsorted_list)
```

```
>> True
```

С помощью встроенного в Python ключевого слова `in` вы произвели линейный поиск в `unsorted_list` лишь за одну строку кода. В приведенных до этого примерах вы искали только числа, но линейный поиск позволяет находить и символы в строках. В Python вы можете найти символ в строке с помощью линейного поиска следующим образом:

```
print('a' in 'apple')
```

Двоичный поиск

Двоичный (бинарный) поиск — еще один, более быстрый алгоритм для поиска в списке чисел. Однако его нельзя применять для любого набора данных, потому что он работает только с отсортированными данными.

Двоичный поиск ищет элемент в списке, разделяя список пополам. Предположим, у вас есть отсортированный список чисел, представленный на рис. 3.1 (от меньшего к большему), и вы ищете число 19.



Рис. 3.1. Отсортированный набор данных для двоичного поиска

Ваш первый шаг в двоичном поиске — найти среднее число. В данном списке семь элементов, поэтому среднее число — это 14 (рис. 3.2).

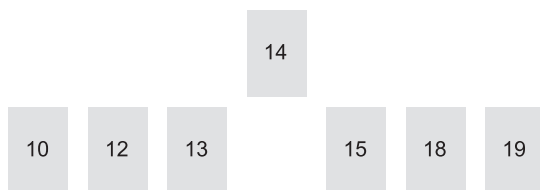


Рис. 3.2. Двоичный поиск сначала определяет среднее число

Поскольку 14 не является искомым числом, вы продолжаете.

Следующий шаг — определить, больше или меньше среднего числа то число, которое вы ищете. Ваше число 19 больше 14, поэтому вам не нужно искать в нижней половине списка и вы можете избавиться от этой половины. Теперь все, что у вас осталось, — верхняя половина списка с тремя числами, в которой вы и продолжаете поиск (рис. 3.3).



Рис. 3.3. Следующий шаг двоичного поиска исключает ту половину данных, которая не может содержать искомое число

Далее вы повторяете процесс, снова определяя среднее число, которое теперь равно 18 (рис. 3.4).

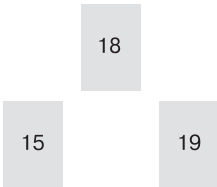


Рис. 3.4. Двоичный поиск опять находит среднее число

Поскольку 18 не является искомым числом, вы опять определяете, какую часть списка оставлять — верхнюю или нижнюю. Так как 19 больше 18, вы оставляете верхнюю половину и исключаете нижнюю.

В результате остается лишь одно число 19, а его-то вы и искали (рис. 3.5). Если бы оставшееся число было не 19, вы бы узнали, что вашего числа нет в списке.



Рис. 3.5. Двоичный поиск нашел нужное число

В линейном поиске вам потребовалось бы семь шагов, чтобы найти число 19. В двоичном поиске вы потратили только три шага, что составляет меньше половины количества шагов линейного поиска.

Вот как можно применить двоичный поиск на Python:

```
def binary_search(a_list, n):
    first = 0
    last = len(a_list) - 1
    while last >= first:
        mid = (first + last) // 2
        if a_list[mid] == n:
            return True
        else:
            if n < a_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return False
```

Функция `binary_search` принимает два аргумента — `a_list` и `n` (искомое число):

```
def binary_search(a_list, n):
```

Вы используете переменные `first` и `last`, чтобы отслеживать начало и конец списка, в котором ведется поиск. Сначала вы устанавливаете значение `first` равным 0, а затем присваиваете переменной `last` значение, равное длине списка минус один. Вы будете менять значения переменных по мере деления `a_list` на все меньшие сегменты:

```
first = 0
last = len(a_list) - 1
```

Цикл алгоритма продолжается до тех пор, пока в списке еще есть элементы:

```
while last >= first:
```

Внутри цикла вы определяете среднюю точку `a_list`, складывая `first` и `last`, а затем деля этот результат на 2:

```
mid = (first + last) // 2
```

Двойная косая черта — это **оператор целочисленного деления**, который возвращает целое значение результата деления, округляя его в меньшую сторону. Например, 7 разделить на 2 будет 3,5, но при целочисленном делении это 3. Вы используете целочисленное деление, поэтому `mid` — всегда целое число, так как индексы элементов всегда являются целыми числами.

Затем вы используете условный оператор, чтобы проверить, является ли центральный элемент вашего списка искомым элементом. Если это так, вы возвращаете значение `True`, потому что искомое число найдено:


```
if a_list[mid] == n:  
    return True
```

Если центральный элемент вашего списка — не ваша цель, вы проверяете, больше или меньше число, которое вы ищете, чем значение средней точки. Когда оно меньше средней точки, вы устанавливаете значение `last` равным среднему значению минус 1, что исключает верхнюю половину списка из дальнейшего поиска:

```
if n < a_list[mid]:  
    last = mid - 1
```

Если искомое число больше среднего значения, вы устанавливаете значение `first` равным среднему значению плюс 1, что исключает нижнюю половину списка из дальнейшего поиска:

```
else:  
    first = mid + 1
```

Затем ваш цикл повторяется в меньшем сегменте списка, который вы создаете, используя переменные `first` и `last`:

```
mid = (first + last) // 2
```

Когда цикл завершается, функция возвращает значение `False`, потому что, если вы дошли до конца, нужного числа в списке нет:

```
return False
```

Когда следует использовать двоичный поиск

Двоичный поиск имеет временную сложность $O(\log n)$. Он более эффективен, чем линейный, так как не нужно просматривать весь список. Вместо этого вы можете исключать из поиска целые сегменты списка. Эффективность двоичного поиска особенно заметна, когда речь идет о работе с большими объемами данных. Например, когда вы ищете в списке, состоящем из миллиона чисел, с линейным поиском может потребоваться миллион шагов для завершения, а с логарифмическим двоичным — только 20.

Теперь подробнее поговорим о том, что значит для алгоритма быть логарифмическим. **Возведение в степень** — математическая операция, которая записывается как b^n (или `b**n` на языке Python): это умножение числа b на само

себя n раз. В этом равенстве b называется **основанием**, а число n — **степенью**. Процесс представляет собой возведение основания b в степень n . Например, $2^{**}2 = 2 \times 2$, $2^{**}3 = 2 \times 2 \times 2$ и т. д. **Логарифм** — это показатель степени, в которую нужно возвести основание, чтобы получить искомое число. Логарифм может сказать вам, сколько раз нужно умножить число 2 на само себя, чтобы получить 8. Математически данный вопрос отображается как $\log_2(8)$. Значение $\log_2(8)$ равно 3, так как вам необходимо число 2 умножить на себя три раза, чтобы получить 8 (рис. 3.6).



Рис. 3.6. Экспоненциальное представление в сравнении с логарифмическим представлением

Когда при двоичном поиске вы первый раз делите список пополам, у вас остается $n/2$ элементов. После второй итерации у вас остается $n/2/2$ элементов, а после третьей — $n/2/2/2$ элементов. Иными словами, после первой итерации двоичного поиска остается $n/2^{*}1$ элементов, а после третьей — $n/2^{**}3$ элементов. Если обобщить, то после x итераций в списке остается $n/2^{**}x$ элементов.

Вы можете использовать логарифм для определения того, сколько итераций потребуется двоичному поиску, чтобы найти число в списке в наихудшем сценарии. Например, у вас есть список из 100 чисел, и вы хотите узнать, сколько итераций потребуется двоичному поиску, чтобы убедиться в отсутствии в списке определенного числа. Для этого вам необходимо решить для n равенство $2^{**}n = 100$, аналогичное $\log_2(100)$. Можно попробовать решить уравнение угадыванием. Например, вы предположите, что n равно 5. Однако $2^{**}5$ равно 32, что слишком мало. Продолжив угадывание, вы убедитесь, что $2^{**}6$ равно 64 — а это тоже слишком мало, но $2^{**}7$, равное 128, больше 100, что и является вашим ответом. Другими словами, если вы выполняете двоичный поиск по списку из 100 элементов и заданного вами элемента в нем нет, то чтобы это определить, вашему алгоритму потребуется 7 шагов. Или — $100/2/2/2/2/2/2/2 < 1$.

Двоичный поиск при каждой итерации делит ваш список пополам, а это значит, что основание логарифма, описывающего время выполнения поиска, равно 2. Тем не менее в нотации «О большое» основание логарифма не играет роли, потому что его можно изменить, умножив логарифм на константу. Математические тонкости вне компетенции данной книги, но следует знать, что основание логарифма не имеет значения в нотации «О большое». Важно то, является ли

алгоритм логарифмическим, а так обычно и бывает, когда алгоритм при каждой итерации сокращает объем вычислений в два раза и более.

Поскольку двоичный поиск особенно эффективен при работе со списком отсортированных данных, лучше использовать именно его. Однако, даже если ваши данные не отсортированы, иногда стоит их специально отсортировать, чтобы воспользоваться преимуществами двоичного поиска. Например, если у вас большой список и вы планируете выполнить много поисковых запросов, вам может быть выгоднее отсортировать свои данные один раз, чтобы значительно ускорить каждый поиск, который вы будете выполнять в будущем.

Как и с линейным поиском, у языка Python есть встроенный модуль для двоичного поиска, который следует использовать для написания реальных приложений. Ключом к применению двоичного поиска с помощью встроенных в Python инструментов является использование `bisect_left` из модуля `bisect`: таким образом с помощью двоичного поиска находится индекс существующего элемента в отсортированном списке:

```
from bisect import bisect_left

sorted_fruits = ['apple', 'banana', 'orange', 'plum']
bisect_left(sorted_fruits, 'banana')

>> 1
```

В данном случае `bisect_left` возвращает 1, потому что 'banana' с индексом 1 находится в `sorted_fruits`. Если элемента, который вы ищете, нет в вашей отсортированной итерации, то `bisect_left` возвращает тот индекс, с которым он был бы, если бы имелся в списке.

```
from bisect import bisect_left

sorted_fruits = ['apple', 'banana', 'orange', 'plum']
bisect_left(sorted_fruits, 'kiwi')

>> 2
```

Как видите, элемента 'kiwi' нет в отсортированной итерации, но если бы он там был, то его индекс был бы равен 2.

Поскольку `bisect_left` показывает, где должен быть элемент, если его нет в списке, то для проверки того, есть ли элемент в итерации, вам следует убедиться, что индекс находится в пределах итерации (`bisect` может вернуть позицию за пределами итерации) и что элемент с индексом, возвращаемым `bisect_left`,

является искомым значением. Вот как использовать `bisect_left` для выполнения двоичного поиска в Python:

```
from bisect import bisect_left

def binary_search(an_iterable, target):
    index = bisect_left(an_iterable, target)
    if index <= len(an_iterable) and an_iterable[index] == target:
        return True
    return False
```

Если `bisect_left` возвращает индекс в пределах итерации и индекс указывает на искомый элемент, вы возвращаете значение `True`, так как элемент находится в вашей итерации. В противном случае, если элемента в итерации нет, вы возвращаете значение `False`.

Поиск символов

Теперь вы знаете, как осуществлять поиск в списке, используя встроенные в Python инструменты линейного и двоичного поиска. Но что, если бы вам понадобилось с нуля написать линейный или двоичный поиск не цифр, а каких-то других символов? Для начала нужно разобраться, каким образом компьютер хранит символы.

Набор символов (character set) — таблица сопоставления символов с двоичными числами. Программисты используют кодировку символов для реализации различных наборов символов. В соответствии с Американской стандартной кодировочной таблицей символов для обмена информацией (American Standard Code for Information Interchange, **ASCII**) компьютер сопоставляет каждую букву алфавита с семибитовым числом. В табл. 3.1 показано, как различные символы в английском языке соотносятся с двоичным кодом.

Например, ASCII-значение *A* равно 1000001 в двоичном формате (вы узнаете больше о двоичном формате далее в книге) и 65 в десятичном (этой системой счисления мы пользуемся каждый день). Двоичное число для *b* равно 01100010. Прописные и строчные буквы, знаки пунктуации, цифры и различные управляющие символы, указывающие на действия, такие как разрыв страницы и перевод строки, — у всех имеются ASCII-коды. Существуют ASCII-коды и для цифр от 0 до 9, так как в таблице ASCII цифры являются не числовыми значениями, а символами для математических целей, таких как выражение чисел в адресе: 26 Broadway Street, New York. Поскольку ASCII сопоставляет каждый символ с семибитовым двоичным числом, таблица может содержать максимум 128 различных символов (2^7 равно 128). Тем не менее большинство компьютеров расширяют ASCII до 8 бит, чтобы можно было использовать 256 символов.

Таблица 3.1. Таблица ASCII

Дес	Двоич	Символ	Дес	Двоич	Символ	Дес	Двоич	Символ	Дес	Двоич	Символ
0	0000 0000	[NUL]	32	0010 0000	Пробел	64	0100 0000	@	96	0110 0000	`
1	0000 0001	[SOH]	33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
2	0000 0010	[STX]	34	0010 0010	"	66	0100 0010	B	98	0110 0010	b
3	0000 0011	[ETX]	35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
4	0000 0100	[EOT]	36	0010 0100	\$	68	0100 0100	D	100	0110 0100	d
5	0000 0101	[ENQ]	37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
6	0000 0110	[ACK]	38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
7	0000 0111	[BEL]	39	0010 0111	'	71	0100 0111	G	103	0110 0111	g
8	0000 1000	[BS]	40	0010 1000	(72	0100 1000	H	104	0110 1000	h
9	0000 1001	[TAB]	41	0010 1001)	73	0100 1001	I	105	0110 1001	i
10	0000 1010	[LF]	42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
11	0000 1011	[VT]	43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
12	0000 1100	[FF]	44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
13	0000 1101	[CR]	45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
14	0000 1110	[SO]	46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
15	0000 1111	[SI]	47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
16	0001 0000	[DLE]	48	0011 0000	0	80	0101 0000	P	112	0111 0000	p
17	0001 0001	[DC1]	49	0011 0001	1	81	0101 0001	Q	113	0111 0001	q
18	0001 0010	[DC2]	50	0011 0010	2	82	0101 0010	R	114	0111 0010	r
19	0001 0011	[DC3]	51	0011 0011	3	83	0101 0011	S	115	0111 0011	s
20	0001 0100	[DC4]	52	0011 0100	4	84	0101 0100	T	116	0111 0100	t
21	0001 0101	[NAK]	53	0011 0101	5	85	0101 0101	U	117	0111 0101	u
22	0001 0110	[SYN]	54	0011 0110	6	86	0101 0110	V	118	0111 0110	v
23	0001 0111	[ETB]	55	0011 0111	7	87	0101 0111	W	119	0111 0111	w
24	0001 1000	[CAN]	56	0011 1000	8	88	0101 1000	X	120	0111 1000	x
25	0001 1001	[EM]	57	0011 1001	9	89	0101 1001	Y	121	0111 1001	y
26	0001 1010	[SUB]	58	0011 1010	:	90	0101 1010	Z	122	0111 1010	z
27	0001 1011	[ESC]	59	0011 1011	;	91	0101 1011	[123	0111 1011	{
28	0001 1100	[FS]	60	0011 1100	<	92	0101 1100	\	124	0111 1100	
29	0001 1101	[GS]	61	0011 1101	=	93	0101 1101]	125	0111 1101	}
30	0001 1110	[RS]	62	0011 1110	>	94	0101 1110	^	126	0111 1110	~
31	0001 1111	[US]	63	0011 1111	?	95	0101 1111	_	127	0111 1111	[DEL]

Хотя в ASCII вы можете выразить 256 символов, но в этой таблице нет поддержки достаточного количества символов для обработки текстов других систем письма, таких как японская или китайская. Чтобы решить эту проблему, программисты разработали таблицу символов **Юникод (Unicode)**. **Кодирование символов** означает присвоение символу числа для цифрового представления. **UTF-8** — один из методов кодирования символов, который используют программисты для реализации таблицы символов Юникода.

Вместо использования 7 и 8 бит, как это делает ASCII, UTF-8 использует до 32 бит для кодирования каждого символа, что позволяет представить более миллиона символов. Система UTF-8 совместима с ASCII, потому что использует то же самое битовое представление для латинского алфавита. Например, прописная A и в ASCII, и в UTF-8 представлена как 1000001.

Вы можете использовать встроенную в Python функцию `ord()`, чтобы получить значение символа ASCII:

```
print(ord('a'))
```

```
>> 97
```

Как видите, ASCII-значение для *a* равно 97 (в десятичной системе счисления).

Функция `ord()` может быть полезна, если вы хотите работать непосредственно с базовыми ASCII-кодами различных символов. Чтобы изменить бинарный поиск, который вы написали ранее для поиска символов, вам нужно получить и сравнить значения символов ASCII. Каждый раз во время прохождения цикла вы будете проверять, является ли ASCII-код каждого символа выше, ниже или равным искомому ASCII-коду. Вместо того чтобы показать решение задачи, в конце главы я предложу вам написать код самостоятельно.

Теперь вы знаете, как работают линейный и двоичный поиски и в каких случаях их следует использовать. Несмотря на свою эффективность, двоичный поиск — не самый быстрый способ. Во второй части книги вы узнаете, как искать данные при помощи хеш-таблиц и почему именно такой способ — самый эффективный.

Словарь терминов

Поисковый алгоритм — алгоритм, который ищет данные в наборе данных.

Набор данных — совокупность собранной информации.

Линейный поиск — поисковый алгоритм, который перебирает все элементы в наборе данных и сопоставляет каждый с целевым.

Отсортированные данные — данные, упорядоченные определенным образом.

Двоичный поиск — еще один алгоритм для поиска в списке чисел, более быстрый по сравнению с линейным.

Оператор целочисленного деления — оператор, который возвращает целое значение результата деления, округляя его в меньшую сторону.

Возведение в степень — математическая операция b^n (или `b**n` на языке Python), которая состоит в умножении числа b на само себя n раз.

Основание — b в уравнении возведения в степень (b^n).

Степень — n в уравнении возведения в степень (b^n).

Логарифм — показатель степени, в которую нужно возвести основание, чтобы получить искомое число.

Набор символов — таблица сопоставления символов с двоичными числами.

ASCII — Американская стандартная кодировочная таблица символов для обмена информацией.

Кодирование символов — присвоение числа символу для его цифрового представления.

UTF-8 — один из методов кодирования символов, который используют программисты для реализации таблицы символов Юникода (Unicode).

Практикум

1. Дан список слов в алфавитном порядке. Напишите функцию, которая выполнит двоичный поиск слова и вернет ответ о том, имеется ли оно в списке.

4

Алгоритмы сортировки

Я думаю, пузырьковая сортировка была бы неправильным решением.

Барак Обама

Будучи программистом, в дополнение к поиску данных вам часто придется заниматься и их сортировкой. **Сортировка данных** означает их упорядочение определенным образом. Если говорить о списке чисел, вы можете отсортировать их от меньшего к большему (по возрастанию). Или представьте, что пишете приложение, которое отслеживает книги, прочитанные каждым пользователем, — в подобном приложении вы можете разрешить пользователю просматривать книги, отсортированные разными способами. Например, от самой короткой до самой длинной, от самой старой до самой новой или от самой новой до самой старой.

Существует множество различных алгоритмов сортировки, которые помогут вам отсортировать данные, и у каждого свои плюсы и минусы. Какие-то алгоритмы сортировки лучше работают в определенных ситуациях, например когда итерируемый объект почти отсортирован. В данной главе вы узнаете о пузырьковой сортировке, сортировке вставками и сортировке слиянием. К другим популярным видам сортировки относятся быстрая сортировка, сортировка Шелла и пирамидальная сортировка. Многие алгоритмы стоит применять лишь в редких случаях, поэтому, познакомив вас с несколькими видами сортировки, в остальной части главы я буду рассказывать, как использовать встроенные

в Python функции для сортировки данных, которые вы будете часто использовать в реальных задачах программирования.

Когда вы создаете свои программы в реальном мире, вы почти всегда должны использовать встроенную в ваш язык программирования функцию сортировки. Не нужно применять классические алгоритмы сортировки, которые мы здесь рассматриваем (за исключением редких случаев), потому что в современных языках программирования, таких как Python, имеются встроенные функции, работающие быстрее. Однако изучение нескольких классических алгоритмов сортировки поможет вам лучше разобраться с временной сложностью и познакомит с важными концепциями, знание которых пригодится и в решении других задач.

Пузырьковая сортировка

Пузырьковая сортировка — это алгоритм сортировки, при котором вы перебираете список чисел, сравнивая каждое число с последующим, и переставляете их, если они идут не по порядку. Программисты называют такую сортировку пузырьковой, потому что наибольшие числа «всплывают» в конце списка, а наименьшие перемещаются в начало списка по мере выполнения алгоритма.

Допустим, у вас есть такой список:

```
[32, 1, 9, 6]
```

Сначала вы сравниваете 1 и 32:

```
[32, 1, 9, 6]
```

Тридцать два больше, поэтому вы меняете эти числа местами:

```
[1, 32, 9, 6]
```

Далее вы сравниваете 32 и 9:

```
[1, 32, 9, 6]
```

Тридцать два больше, поэтому вы опять меняете их местами:

```
[1, 9, 32, 6]
```

И наконец, вы сравниваете 32 и 6:

```
[1, 9, 32, 6]
```

И опять меняете их местами:

[1, 9, 6, 32]

Как видите, 32 «всплыло» в конце списка. Однако числа в вашем списке все еще расположены не по порядку, так как 9 и 6 стоят не на своих местах. Поэтому алгоритм начинает все сначала и сравнивает 1 и 9:

[1, 9, 6, 32]

Ничего не происходит, потому что 1 не больше 9. Затем сравниваются 9 и 6:

[1, 9, 6, 32]

Девять больше шести, поэтому вы меняете их местами, и теперь все числа в списке стоят по порядку:

[1, 6, 9, 32]

При пузырьковой сортировке наибольшее число из списка перемещается в конец списка в конце первой итерации алгоритма, но если наименьшее число списка стоит в конце, алгоритму потребуется множество проходов, чтобы переместить его в начало. В приведенном примере 32 оказалось в конце списка после первой итерации. Предположим, у вас имеется такой список:

[32, 6, 9, 1]

Потребуется четыре итерации, чтобы переместить число 1 из конца в начало списка.

Чтобы лучше понять работу этого алгоритма, можно использовать визуализатор пузырьковой сортировки. В интернете есть множество визуализаторов, которые помогут укрепить ваше понимание работы данного алгоритма. Я советую обратиться к визуализаторам для всех алгоритмов сортировки, о которых вы узнаете в этой главе.

Вот как писать алгоритм пузырьковой сортировки на языке Python:

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        for j in range(list_length):
            if a_list[j] > a_list[j + 1]:
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
    return a_list
```

Функция `bubble_sort` принимает список чисел под названием `a_list` в качестве параметра:

```
def bubble_sort(a_list):
```

Внутри функции вы получаете длину своего списка, вычитаете 1 и сохраняете результат в `list_length`, чтобы контролировать количество итераций, которые осуществит ваш алгоритм:

```
list_length = len(a_list) - 1
```

У функции есть два вложенных цикла, поэтому вы можете перебирать список и сравнивать элементы:

```
for i in range(list_length):  
    for j in range(list_length)
```

Во внутреннем цикле `for` используется оператор `if`, чтобы сравнивать текущее число с последующим, прибавляя 1 к индексу текущего числа:

```
if a_list[j] > a_list[j + 1]:
```

Эта строка кода является текущим числом:

```
a_list[j]
```

А эта строка кода обозначает следующее число в вашем списке:

```
a_list[j + 1]
```

Если текущее число больше последующего, вы меняете их местами. Представленный ниже синтаксис Python позволяет менять местами два элемента, не загружая один из элементов во временную переменную:

```
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
```

Все сопоставления происходят во внутреннем цикле. Внешний цикл нужен лишь для того, чтобы алгоритм выполнялся столько раз, сколько требуется для приведения списка в порядок. Возьмем, например, список из начала этого раздела:

```
[32, 1, 9, 6]
```

После итерации внутреннего цикла он будет выглядеть так:

```
[1, 9, 6, 32]
```

Помните, однако, что это еще не правильный порядок. Если бы у вас был только внутренний цикл, алгоритм завершился бы преждевременно и список не имел бы правильного порядка. Вот для чего нужен внешний цикл — чтобы запустить внутренний цикл алгоритма заново и повторять его, пока числа не расположатся правильно.

Вы можете улучшить эффективность пузырьковой сортировки, включив `-i` во второй цикл `for`. Таким образом, ваш внутренний цикл не будет сравнивать

последние два числа при первом прохождении цикла, при втором прохождении цикла он не будет сравнивать последние три числа и т. д.

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        for j in range(list_length - i):
            if a_list[j] > a_list[j + 1]:
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
    return a_list
```

Вам не нужно делать эти сравнения, так как наибольшие числа всплывают в конце списка. В исходном примере в начале раздела вы видели, как 32 всплыло в конце списка после первой итерации сортировки. Это значит, что наивысшее число находится в конце списка после первой итерации, второе наибольшее число находится на предпоследнем месте после второй итерации и т. д. Поэтому вам не нужно сравнивать другие числа с этими числами, и вы можете завершить цикл быстрее. Взгляните, например, на список:

[32, 1, 9, 6]

После одной полной итерации внутреннего цикла он выглядит следующим образом:

[1, 9, 6, 32]

После одной итерации наибольшее число перемещается в конец списка, поэтому больше не нужно сравнивать другие числа с числом 32, так как вы знаете, что это наибольшее число вашего списка.

Во время второго прохождения внутреннего цикла второе по величине число перемещается на свою итоговую позицию — вторую с конца.

[1, 6, 9, 32]

Так, каждый раз проходя внутренний цикл, вы можете завершать его раньше.

Вы можете сделать пузырьковую сортировку еще более эффективной, добавив переменную для отслеживания перестановки элементов во внутреннем цикле алгоритма. Если вы проходите цикл без перестановок, ваш список отсортирован, и вы можете завершить цикл и вернуть список без дальнейшей обработки.

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        no_swaps = True
```

```
for j in range(list_length - i):
    if a_list[j] > a_list[j + 1]:
        a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
        no_swaps = False
if no_swaps:
    return a_list
return a_list
```

В данном случае вы добавили переменную `no_swaps`, которая принимает значение `True` в начале внутреннего цикла. Если вы меняете местами два числа в своем внутреннем цикле, вы установите для нее значение `False`. Если пройден внутренний цикл и переменная `no_swaps` будет иметь значение `True`, список станет отсортированным и алгоритм завершится. Подобное небольшое изменение значительно ускорит пузырьковую сортировку, если изначально список будет почти отсортирован.

Когда следует использовать пузырьковую сортировку

В рассмотренной нами реализации пузырьковой сортировки алгоритм сортировал числа, но ее (как и любую другую сортировку) можно использовать и для сортировки строк. Например, вам нужно изменить пузырьковую сортировку так, чтобы она отсортировала строки в алфавитном порядке по первой букве каждой строки.

Преимущество пузырьковой сортировки в простоте ее применения, поэтому она хорошо подходит тем, кто только начинает изучать алгоритмы сортировки. Поскольку данная сортировка основана на двух вложенных циклах `for`, ее временная сложность равна $O(n^2)$. Хотя она может быть эффективна для малого набора данных, но для большого набора данных ее эффективность снижается.

Пузырьковая сортировка также является устойчивой. **Устойчивая сортировка** — сортировка, которая не нарушает никакой последовательности, кроме той, что указана ключом сортировки. Например, у вас есть база данных, включающая четыре названия животных:

Акита
Медведь
Тигр
Альбатрос

Если вы отсортируете список по первому символу, то на выходе получите следующий список:

Устойчивая сортировка

Акиа
Альбатрос
Медведь
Тигр

Это пример устойчивой сортировки, потому что *акита* и *альбатрос* расположены в том же порядке, что и в исходном списке, хотя ваша сортировка просматривала только первую букву слова. Неустойчивая сортировка может нарушить исходный порядок двух слов, начинающихся на букву *А*, таким образом, что *альбатрос* может оказаться выше *акиты* в списке, хотя в исходном списке *акита* располагалась выше *альбатроса*.

Неустойчивая сортировка

Альбатрос
Акиа
Медведь
Тигр

Другими словами, в устойчивой сортировке при наличии двух одинаковых ключей элементы сохраняют свой первоначальный порядок.

Несмотря на устойчивость пузырьковой сортировки, вы вряд ли найдете кого-то, кто использует пузырьковую сортировку на практике, поскольку ее временная сложность равна $O(n^2)$. К тому же существуют более эффективные виды сортировки (которые вы скоро изучите).

Сортировка вставками

Сортировка вставками — это алгоритм сортировки, при котором вы сортируете данные как колоду карт. Сначала вы разбиваете список чисел пополам: на отсортированную левую половину и неотсортированную правую. Затем сортируете левую половину так же, как сортировали бы игральные карты. Например, когда вы сортируете пять карт в возрастающем порядке, вы просматриваете каждую карту и размещаете ее справа от меньшей по значению карты.

Вот каким образом работает сортировка вставками для списка из четырех элементов: 6, 5, 8 и 2. Алгоритм начинается с 5 — второго элемента списка:

[6, 5, 8, 2]

Далее вы сравниваете текущий элемент с предыдущим. Шесть больше пяти, поэтому вы меняете их местами:

```
[5, 6, 8, 2]
```

Теперь левая половина вашего списка отсортирована, а правая — нет:

```
[5, 6, 8, 2]
```

Затем вы переходите к третьему элементу списка. Шесть не больше восьми, поэтому вы не меняете их местами:

```
[5, 6, 8, 2]
```

Поскольку левая половина списка отсортирована, вам не нужно сравнивать 8 и 5:

```
[5, 6, 8, 2]
```

Далее вы сравниваете 8 и 2:

```
[5, 6, 8, 2]
```

Так как 8 больше, чем 2, вы просматриваете элементы в левой отсортированной половине списка, сравнивая 2 с каждым числом, пока 2 не окажется в начале и весь список не будет отсортирован:

```
[2, 5, 6, 8]
```

Вот как на языке Python выглядит код для сортировки вставками:

```
def insertion_sort(a_list):
    for i in range(1, len(a_list)):
        value = a_list[i]
        while i > 0 and a_list[i - 1] > value:
            a_list[i] = a_list[i - 1]
            i = i - 1
        a_list[i] = value
    return a_list
```

Вы начинаете с объявления функции `insertion_sort`, которая принимает список чисел в качестве входных данных:

```
def insertion_sort(a_list):
```

Ваша функция использует цикл `for` для перебора всех элементов списка. Затем она использует цикл `while` для сравнений, которые делает ваш алгоритм при добавлении нового числа в отсортированную левую часть списка:

```
    for i in range(1, len(a_list)):
        ...
        while i > 0 and a_list[i - 1] > value:
            ...
```

Цикл `for` начинается со второго элемента списка (индекс 1). Внутри списка вы отслеживаете текущее число в переменной `value`:

```
for i in range(1, len(a_list)):
    value = a_list[i]
```

Цикл `while` перемещает элементы из неотсортированной правой половины списка в отсортированную левую половину. Так продолжается до тех пор, пока верны две вещи: `i` должно быть больше 0, а предыдущий элемент в списке должен быть больше следующего за ним элемента. Переменная `i` должна быть больше 0, так как цикл `while` сравнивает два числа, а если `i` равно 0, это означает, что алгоритм находится у первого элемента списка и нет предшествующего числа для сравнения:

```
while i > 0 and a_list[i - 1] > value:
```

Цикл `while` выполняется, только если число в переменной `value` меньше предыдущего элемента списка. Внутри цикла `while` вы перемещаете большее значение в правую часть списка. Затем алгоритм выясняет, куда поместить меньшее значение в отсортированной левой части списка. Вы уменьшаете `i`, чтобы цикл смог провести еще одно сравнение и выяснить, надо ли переместить меньшее число еще дальше влево.

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
```

Когда цикл `while` завершается, вы помещаете текущее число в переменную `value` на правильную позицию в отсортированной левой половине списка:

```
a_list[i] = value
```

Посмотрим, как алгоритм сортировки вставками работает для следующего списка:

```
[6, 5, 8, 2]
```

Когда цикл `for` проходит первый раз, `i` равно 1, а `value` равно 5:

```
for i in range(1, len(a_list)):
    value = a_list[i]
```

Код, выделенный ниже жирным шрифтом, дает значение `True`, потому что `i > 0` и `6 > 5`, и ваш цикл `while` выполняется:

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
a_list[i] = value
```


Этот код:

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
a_list[i] = value
```

превращает ваш список из такого:

[6, 5, 8, 2]

вот в такой:

[6, 6, 8, 2]

А этот код:

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
a_list[i] = value
```

уменьшает *i* на 1. Переменная *i* теперь равна 0, и это значит, что цикл `while` больше не будет выполняться:

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
a_list[i] = value
```

Далее, эта строка кода:

```
while i > 0 and a_list[i - 1] > value:
    a_list[i] = a_list[i - 1]
    i = i - 1
a_list[i] = value
```

превращает ваш список из такого:

[6, 6, 8, 2]

вот в такой:

[5, 6, 8, 2]

Вы отсортировали первые два элемента списка, и все, что требуется вашему алгоритму, это повторить тот же самый процесс сортировки в правой половине списка, содержащей 8 и 2.

Когда следует использовать сортировку вставками

Как и пузырьковая, сортировка вставками устойчива и имеет временную сложность $O(n^2)$, поэтому данный вид сортировки не очень эффективный. Однако, в отличие от пузырьковой, сортировку вставками программисты нередко используют при написании реальных приложений. Например, сортировка вставками может быть эффективна, если ваши данные почти отсортированы. Когда список почти отсортирован, временная сложность сортировки вставками равна $O(n)$, что очень эффективно.

Предположим, у вас есть следующий список:

[1, 2, 3, 4, 5, 7, 6]

Как видите, все числа отсортированы, кроме последних двух. Поскольку второй цикл (цикл `while`) при сортировке вставками выполняется, только если два числа расположены в неправильном порядке, для обработки этого почти отсортированного списка сортировкой вставками потребуется лишь восемь шагов, что является линейным, так как второй цикл надо будет запустить один раз.

Сортировка слиянием

Сортировка слиянием — это алгоритм рекурсивной сортировки, который постоянно делит список пополам, пока не останется несколько списков (или один), содержащих один элемент, а затем объединяет их вместе в правильном порядке. Вот как она работает. Сначала вы используете рекурсию, чтобы постоянно разбивать список пополам, пока ваш исходный список не превратится в один или несколько подсписков, содержащих только одно число (рис. 4.1).

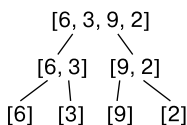


Рис. 4.1. Первая часть сортировки слиянием

Списки из одного элемента отсортированы по умолчанию. Как только вы отсортируете списки, содержащие один элемент, вы объедините свои подсписки по два, сравнивая первый элемент в каждом списке. Объединение списков означает соединение их в отсортированном порядке.

Сначала вы объединяете [6] и [3], затем объединяете [9] и [2]. В данном примере каждый список содержит лишь одно число, поэтому вы сравниваете два числа и помещаете наименьшее в начало вашего нового объединенного списка, а наибольшее — в конец. Теперь у вас два отсортированных списка:

```
[3, 6], [2, 9]
```

Затем вы объединяете эти два списка:

```
# необъединенный  
[3, 6], [2, 9]
```

```
# объединенный  
[]
```

Сначала вы сравниваете 3 и 2. Поскольку 2 меньше, оно попадает в объединенный список:

```
# необъединенный  
[3, 6], [9]
```

```
# объединенный  
[2]
```

Теперь вы сравниваете 3 и 9. Поскольку 3 меньше, оно попадает в объединенный список:

```
# необъединенный  
[6], [9]
```

```
# объединенный  
[2, 3]
```

И наконец, вы сравниваете 6 и 9. Поскольку 6 меньше, оно попадает в объединенный список. Затем вы помещаете 9 в список:

```
# необъединенный  
[], []
```

```
# объединенный  
[2, 3, 6, 9]
```

Теперь, после завершения всех слияний, у вас один отсортированный список.

Вот как применять сортировку слиянием на языке Python:

```
def merge_sort(a_list):  
    if len(a_list) > 1:  
        mid = len(a_list) // 2  
        left_half = a_list[:mid]  
        right_half = a_list[mid:]
```

```

merge_sort(left_half)
merge_sort(right_half)

left_ind = 0
right_ind = 0
alist_ind = 0
while left_ind < len(left_half) and right_ind < len(right_half):
    if left_half[left_ind] <= right_half[right_ind]:
        a_list[alist_ind] = left_half[left_ind]
        left_ind += 1
    else:
        a_list[alist_ind] = right_half[right_ind]
        right_ind += 1
    alist_ind += 1

while left_ind < len(left_half):
    a_list[alist_ind] = left_half[left_ind]
    left_ind += 1
    alist_ind += 1

while right_ind < len(right_half):
    a_list[alist_ind] = right_half[right_ind]
    right_ind += 1
    alist_ind += 1

```

Эта часть алгоритма отвечает за разбивку списков на подспски:

```

if len(a_list) > 1:
    mid = len(a_list) // 2
    left_half = a_list[:mid]
    right_half = a_list[mid:]
    merge_sort(left_half)
    merge_sort(right_half)

left_ind = 0
right_ind = 0
alist_ind = 0

```

А эта часть отвечает за объединение двух списков:

```

left_ind = 0
right_ind = 0
alist_ind = 0
while left_ind < len(left_half) and right_ind < len(right_half):
    if left_half[left_ind] <= right_half[right_ind]:
        a_list[alist_ind] = left_half[left_ind]
        left_ind += 1
    else:

```

```
        a_list[alist_ind] = right_half[right_ind]
        right_ind += 1
        alist_ind += 1

while left_ind < len(left_half):
    a_list[alist_ind]=left_half[left_ind]
    left_ind += 1
    alist_ind += 1

while right_ind < len(right_half):
    a_list[alist_ind]= right_half[right_ind]
    right_ind += 1
    alist_ind += 1
```

Рекурсия является ключом к этому алгоритму. Позднее мы обсудим, как работает вторая половина алгоритма, отвечающая за слияние, а сейчас рассмотрим рекурсивную часть функции шаг за шагом. Предположим, вы берете список, приведенный в начале данного раздела:

```
[6, 3, 9, 2]
```

Когда вы первый раз вызываете функцию `merge_sort`, она создает три переменные:

```
# Первый вызов функции
```

```
a_list = [6, 3, 9, 2]
left_half = [6, 3]
right_half = [9, 2]
```

Вы передаете `a_list` в свою функцию, и этот код создает две другие переменные, разделяя `a_list` на левую и правую половины:

```
mid = len(a_list) // 2
left_half = a_list[:mid]
right_half = a_list[mid:]
```

Затем функция рекурсивно вызывает саму себя и передает `left_half` в качестве параметра:

```
merge_sort(left_half)
```

Далее Python создает еще три переменные:

```
# Второй вызов функции
```

```
a_list = [6, 3]
left_half = [6]
right_half = [3]
```

Критически важно понимать, что `left_half` в первом вызове функции и `a_list` во втором вызове функции *указывают на один и тот же список*. Это значит, что, когда вы во втором вызове функции меняете `left_half`, это изменяет `a_list` в первом вызове функции.

Теперь ваш код снова вызывает сам себя, но на этот раз `left_half` равна `[6]`, и терминальная ветвь останавливает дальнейшую рекурсию.

```
if len(a_list) > 1:
```

Итак, Python переходит к следующей строке кода:

```
merge_sort(right_half)
```

Переменная `right_half` равна `[3]`, и терминальная ветвь останавливает повторный вызов функции:

```
if len(a_list) > 1:
```

Теперь Python вызывает код слияния, который объединяет `left_half`, равную `[6]`, и `right_half`, равную `[3]`, и сохраняет результат, изменяя `a_list`, который теперь отсортирован:

```
# Второй вызов функции
```

```
a_list = [3, 6]
left_half = [6]
right_half = [3]
```

Как вы узнали ранее, когда вы изменяете `a_list` во втором вызове функции, вы также изменяете переменную `left_half` в первом вызове функции.

Изначально состояние Python при первом вызове функции выглядело так:

```
# Первый вызов функции
```

```
a_list = [6, 3, 9, 2]
left_half = [6, 3]
right_half = [9, 2]
```

Но при втором вызове вы изменили `a_list`, что также изменило переменную `left_half` в первом вызове. Теперь первый вызов функции выглядит следующим образом:

```
# Первый вызов функции
```

```
a_list = [3, 6, 9, 2]
```

```
left_half = [3, 6]
right_half = [9, 2]
```

Эта переменная важна, поскольку, как вы уже знаете, при применении рекурсии Python возвращается к предыдущим состояниям, когда вы достигаете терминальной ветви. Как видите, `left_half` теперь отсортирована.

Ваш алгоритм снова вызывает себя рекурсивно, запускается тот же процесс, и `right_half` также сортируется. Это означает, что первое состояние функции выглядит следующим образом:

```
# Первый вызов функции
```

```
a_list = [3, 6, 9, 2]
left_half = [3, 6]
right_half = [2, 9]
```

Как видите, `right_half` теперь отсортирована при первом вызове функции. Когда алгоритм возвращается к первому состоянию функции, он объединяет `left_half` и `right_half`, которые были уже отсортированы в предыдущем вызове рекурсивной функции. Таким образом, когда алгоритм достигает этой точки и вызывает свой код слияния, ваша функция содержит два отсортированных списка `left_half` и `right_half`, и ей нужно лишь объединить их, чтобы получить итоговый отсортированный список.

Теперь посмотрим на код, который объединяет два списка. Ваш код слияния начинается с трех переменных, для которых вы устанавливаете значение 0:

```
left_ind = 0
right_ind = 0
alist_ind = 0
```

Вы используете эти переменные для отслеживания индексов трех списков: `left_half`, `right_half` и `a_list`.

Код сравнивает каждый первый элемент списка `left_half` с каждым первым элементом списка `right_half` и помещает наименьшее число на правильную позицию в список `a_list`:

```
while left_ind < len(left_half) and right_ind < len(right_half):
    if left_half[left_ind] <= right_half[right_ind]:
        a_list[alist_ind] = left_half[left_ind]
        left_ind += 1
    else:
        a_list[alist_ind] = right_half[right_ind]
        right_ind += 1
    alist_ind += 1
```

А этот код завершает каждое слияние и также может обработать ситуацию, когда два объединяемых списка не равны:

```
while left_ind < len(left_half):
    a_list[alist_ind] = left_half[left_ind]
    left_ind += 1
    alist_ind += 1

while right_ind < len(right_half):
    a_list[alist_ind] = right_half[right_ind]
    right_ind += 1
    alist_ind += 1
```

Например, вы объединяете следующие два списка:

[2], [6, 4]

Python будет вызывать код слияния для этих переменных:

```
left_ind = 0
right_ind = 0
alist_ind = 0

a_list = [6, 3]
left_half = [6]
right_half = [3]
```

Выражения `left_ind < len(left_half)` и `right_ind < len(right_half)` принимают значение `True`, поэтому вы входите в свой цикл `while`. Этот код:

```
left_half[left_ind] <= right_half[right_ind]:
```

вычисляется как `6 <= 3`, что равно `False`, поэтому выполняется этот код:

```
if left_half[left_ind] <= right_half[right_ind]:
    a_list[alist_ind] = left_half[left_ind]
    left_ind += 1
else:
    a_list[alist_ind] = right_half[right_ind]
    right_ind += 1
alist_ind += 1
```

Теперь ваши переменные выглядят следующим образом:

```
left_ind = 0
right_ind = 1
alist_ind = 1

a_list = [3, 3]
left_half = [6]
right_half = [3]
```


Во время следующего прохождения цикла `while` код не выполняется, так как `right_ind` не меньше длины списка `right_half`:

```
while left_ind < len(left_half) and right_ind < len(right_half):
    if left_half[left_ind] <= right_half[right_ind]:
        a_list[alist_ind] = left_half[left_ind]
        left_ind += 1
    else:
        a_list[alist_ind] = right_half[right_ind]
        right_ind += 1
    alist_ind += 1
```

Далее выполняется этот код, потому что `left_ind` меньше длины списка `left_half`:

```
while left_ind < len(left_half):
    a_list[alist_ind] = left_half[left_ind]
    left_ind += 1
    alist_ind += 1

while right_ind < len(right_half):
    a_list[alist_ind] = right_half[right_ind]
    right_ind += 1
    alist_ind += 1
```

Теперь переменные выглядят следующим образом:

```
left_ind = 1
right_ind = 1
alist_ind = 2

a_list = [3, 6]
left_half = [6]
right_half = [3]
```

Ваш список отсортирован, и слияние завершено.

Когда следует использовать сортировку слиянием

Сортировка слиянием — пример алгоритма «разделяй и властвуй». **Алгоритм «разделяй и властвуй»** — это алгоритм, который рекурсивно разбивает задачу на две и более связанные подзадачи до тех пор, пока они не станут достаточно просты, чтобы их можно было с легкостью решить. При сортировке слиянием вы разбиваете список на подсписки, пока каждый из подсписков не будет содержать лишь один элемент, что упрощает решение задачи, так как список из одного элемента по умолчанию отсортирован. Временная сложность выполнения

сортировки слиянием равна $O(n \log n)$, потому что, хоть разделение исходного списка на подписки является логарифмическим, алгоритму требуется линейное время, чтобы обработать каждый элемент в подписках при их слиянии. Сортировка слиянием, имеющая линейно-логарифмическую временную сложность, — один из наиболее эффективных алгоритмов сортировки, который широко применяется программистами. Например, Python использует сортировку слиянием в своих встроенных алгоритмах сортировки, о чем вы скоро узнаете. Так же как пузырьковая сортировка и сортировка вставками, сортировка слиянием является устойчивой.

Принцип слияния из сортировки слиянием можно использовать не только при сортировке данных. Предположим, вы находитесь в аудитории из 50 студентов. У каждого студента в кармане есть случайный набор мелочи на сумму до 1 доллара. Как лучше всего узнать, сколько денег в аудитории в целом? Первое, что приходит на ум, — предложить преподавателю обойти всех студентов и спросить, сколько у каждого из них денег в кармане, а затем сложить полученные ответы. Решение задачи — линейный поиск, а время его выполнения — $O(n)$. Однако более эффективным будет вариант, при котором студенты объединятся друг с другом. Вот как это работает: каждый студент выбирает рядом сидящего товарища, достает свою мелочь и добавляет к ней мелочь соседа (и запоминает сумму). Студенты без мелочи покидают аудиторию. Далее процедура повторяется, пока не останется только один студент со всей мелочью и итоговой суммой. При использовании слияния для подсчета мелочи в аудитории вместо 50 шагов потребуется только 6.

Алгоритмы сортировки в Python

В Python есть две встроенные функции сортировки: `sorted` и `sort`. Функции сортировки Python используют Timsort — гибридный алгоритм сортировки, объединяющий в себе сортировку слиянием и сортировку вставками. **Алгоритм гибридной сортировки** сочетает несколько алгоритмов, решающих одну и ту же задачу: он либо выбирает один из них (в зависимости от данных), либо в процессе переключается между ними. Сочетание сортировки слиянием и сортировки вставками делает алгоритм Timsort намного более эффективным, и именно по этой причине программисты в большинстве случаев используют встроенные в Python алгоритмы, а не кодируют сами.

Функция Python `sorted` позволяет сортировать любой итерируемый объект, если Python может сравнивать содержащиеся в нем данные. Например, вы вызываете функцию `sorted` для списка целых чисел, а Python возвращает новый список, содержащий целые числа из исходного списка, отсортированные в возрастающем порядке:

```
a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list))
```

```
>> [1, 4, 8, 10, 33, 103]
```

Если вы вызовете функцию `sorted` для списка строк, Python вернет вам новый список, отсортированный в алфавитном порядке по первой букве каждой строки:

```
a_list = ["Guido van Rossum", "James Gosling", "Brendan Eich", "Yukihiro
Matsumoto"]
print(sorted(a_list))
```

```
>> ['Brendan Eich', 'Guido van Rossum', 'James Gosling', 'Yukihiro
Matsumoto']
```

У функции Python `sorted` есть дополнительный параметр под названием `reverse`, поэтому, если вы хотите отсортировать список в убывающем порядке, можете передать ей параметр `reverse=True`:

```
a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list, reverse=True))
```

```
>> [103, 33, 10, 8, 4, 1]
```

У функции `sorted` также есть параметр `key`, который позволяет передавать функцию. В таком случае Python вызывает эту функцию для каждого элемента итерации и использует результат для сортировки. Например, вы можете передать функцию `len`, которая отсортирует список строк по длине:

```
a_list = ["onehundred", "five", "seventy", "two"]
print(sorted(a_list, key=len))
```

```
>> ['two', 'five', 'seventy', 'onehundred']
```

У другой встроенной функции сортировки Python, `sort`, такие же дополнительные параметры, как и у `sorted`, но функция `sort` работает только со списками. Кроме того, в отличие от `sorted`, функция `sort` сортирует на месте: она изменяет исходный список вместо возвращения нового. Ниже приведен пример использования функции `sort` для списка:

```
a_list = [1, 8, 10, 33, 4, 103]
a_list.sort()
print(a_list)
```

```
>> [1, 4, 8, 10, 33, 103]
```

Как видите, Python отсортировал числа в исходном списке в порядке возрастания.

Словарь терминов

Сортировка данных — упорядочение данных определенным образом.

Пузырьковая сортировка — алгоритм сортировки, при котором вы перебираете список чисел, сравнивая каждое число с последующим, и переставляете их, если они идут не по порядку.

Устойчивая сортировка — сортировка, которая не нарушает никакой последовательности, кроме той, которая указана ключом сортировки.

Сортировка вставками — алгоритм сортировки, при котором вы сортируете данные, как колоду карт.

Сортировка слиянием — алгоритм рекурсивной сортировки, который непрерывно разделяет список пополам, пока не найдет один и более списков, содержащих один элемент, а затем соединяет их обратно в правильном порядке.

Алгоритм «разделяй и властвуй» — алгоритм, который рекурсивно разбивает задачу на две и более связанные подзадачи, пока они не станут достаточно просты, чтобы с легкостью их решить.

Алгоритм гибридной сортировки — алгоритм, сочетающий в себе несколько алгоритмов, которые решают одну и ту же задачу: он либо выбирает один из них (в зависимости от данных), либо в процессе переключается между ними.

Практикум

1. Изучите и напишите алгоритм сортировки, но не пузырьковой, сортировки вставками или сортировки слиянием.

5

Строковые алгоритмы

Одним из самых важных навыков, которым должен овладеть любой предприниматель, является программирование. Если вы хотите запустить технологический стартап, это критически важный навык, но базовые знания кодирования пригодятся даже в традиционных областях, потому что программное обеспечение меняет все.

Рид Хоффман

В этой главе вы узнаете ответы на некоторые из наиболее распространенных на технических собеседованиях вопросов, касающихся применения строк. Несмотря на то, что в качестве разработчика программного обеспечения вам навряд ли потребуется находить анаграммы, подобное умение поможет вам решать задачи с использованием сортировки. Более того, другие понятия, которые вы изучите в данной главе, такие как модульная арифметика и списковые включения, пригодятся в вашей ежедневной работе программиста.

Обнаружение анаграмм

Две строки являются анаграммами, если содержат одни и те же буквы, но не обязательно в одинаковом порядке (регистр не имеет значения). Например, слова *Car* и *arc* — анаграммы. Ключом к выявлению анаграмм является их сортировка.

Если отсортированные строки одинаковы, это анаграммы. Вот как написать алгоритм, который определяет, являются ли две строки анаграммами:

```
def is_anagram(s1, s2):  
    s1 = s1.replace(' ', '').lower()  
    s2 = s2.replace(' ', '').lower()  
    if sorted(s1) == sorted(s2):  
        return True  
    else:  
        return False
```

```
s1 = 'Emperor Octavian'  
s2 = 'Captain over Rome'  
print(is_anagram(s1,s2))
```

```
>> True
```

Иногда анаграммы содержат несколько слов и включают прописные и строчные буквы, поэтому вы начинаете свою функцию с удаления пробелов из строк и преобразования всех символов в строчные:

```
s1 = s1.replace(' ', '').lower()  
s2 = s2.replace(' ', '').lower()
```

Затем вы сортируете обе строки и сравниваете результат, чтобы определить, одинаковы ли они. Если это так, то строки являются анаграммами, и вы возвращаете значение `True`. В противном случае, если они не одинаковы, вы возвращаете значение `False`.

```
if sorted(s1) == sorted(s2):  
    return True  
else:  
    return False
```

Поскольку алгоритм выявляет анаграммы, используя встроенную в Python функцию `sorted`, время его выполнения равно $O(n \log n)$.

Выявление палиндромов

Палиндром — слово, которое читается одинаково слева направо и справа налево. *Шалаш, довод, кабак и еле-еле* — примеры палиндромов. Один из способов их нахождения — скопировать строку, реверсировать ее и сравнить с оригиналом. Если они равны, строка является палиндромом.

Вот как реверсировать строку на языке Python:

```
print("blackswan"[::-1])
```

```
>> nawskcalb
```

А вот как проверить, является ли строка палиндромом:

```
def is_palindrome(s1):  
    if s1.lower() == s1[::-1].lower():  
        return True  
    return False
```

Сначала вы используете встроенную в Python функцию `lower` с целью убедиться, что наличие заглавных букв не влияет на ваше сравнение. Затем используете нарезку¹, чтобы реверсировать строку и сравнить ее с оригиналом:

```
if s1.lower() == s1[::-1].lower():
```

Если две строки одинаковы, то исходная строка является палиндромом и вы возвращаете значение `True`:

```
return True
```

В противном случае вы возвращаете значение `False`:

```
return False
```

Самая медленная часть алгоритма в выявлении палиндрома — это синтаксис Python для реверсирования списка. Поскольку Python необходимо просмотреть каждый элемент списка, чтобы его реверсировать, время выполнения равно $O(n)$, в результате чего время выполнения алгоритма также равно $O(n)$.

Последняя цифра

Еще одно распространенное задание на техническом собеседовании — вернуть самую правую цифру в строке. Например, в строке "Buy 1 get 2 free" ваша функция должна вернуть число 2.

Один из изысканных способов решения данной задачи — использовать функцию спискового включения в Python. **Списковое включение** — синтаксис языка Python для создания нового, отличающегося списка из текущей итерации (например, другого списка).

¹ Slicing — синтаксис манипуляций со строками. — *Примеч. науч. ред.*

Ниже приведен синтаксис для спискового включения:

```
new_list = [expression(i) for i in iterable if filter(i)]
```

`iterable` — итерация, которую вы используете для создания нового списка. `expression(i)` — переменная, которая включает в себя каждый элемент из `iterable`. Например, в этом регулярном выражении с содержит каждый символ строки "selftaught":

```
print([c for c in "selftaught"])
```

```
>> ['s', 'e', 'l', 'f', 't', 'a', 'u', 'g', 'h', 't']
```

Как видите, Python возвращает список, содержащий все буквы из первоначального списка "selftaught".

`filter(i)` позволяет внести изменения в исходную итерацию. Например, вы можете создать фильтр, добавляющий элементы в итерацию только при условии, что он удовлетворяет требованиям фильтра:

```
print([c for c in "selftaught" if ord(c) > 102])
```

```
>> ['s', 'l', 't', 'u', 'g', 'h', 't']
```

Встроенная в Python функция `ord` возвращает ASCII-код для символа. В данном примере вы добавили фильтр, который прибавляет символы к итерации, только если ASCII-код символа больше 102 (символ *f*). Как видите, в вашем списке не хватает символов *e*, *f* и *a*.

Вы можете использовать функцию Python `isdigit`, чтобы отфильтровать все, кроме чисел:

```
s = "Buy 1 get 2 free"
nl = [c for c in s if c.isdigit()]
print(nl)
```

```
>> ['1', '2']
```

Теперь, когда вы знаете, как использовать списковые включения для поиска цифр в строке, осталось выполнить всего один шаг, чтобы найти крайнюю справа цифру в строке, а именно использовать отрицательный индекс для извлечения последней цифры из нового списка:

```
s = "Buy 1 get 2 free"
nl=[c for c in s if c.isdigit()][-1]
print(nl)
```

```
>> 2
```


Сначала списковое включение возвращает список всех цифр в строке. Затем вы используете отрицательный индекс для извлечения последней цифры из нового списка чисел, которая является крайней справа в исходном списке.

Как видите, вы можете использовать списковые включения, чтобы превратить три-четыре строки кода в красивую строку, что поможет вам написать короткий легкочитаемый код, когда вы программируете профессионально.

Поскольку ваш алгоритм перебирает все символы строки, проверяя, являются ли они цифрами, и реверсирует список, время его выполнения равно $O(n)$.

Шифр Цезаря

Шифр — алгоритм шифрования и дешифрования. Юлий Цезарь, известный римский полководец и политик, защищал свои личные послания, зашифровывая их искусным шифром. Для начала он выбирал число, а затем каждую букву смещал на это число. Например, если он выбирал цифру 3, то строка *abc* превращалась в *def*.

Когда смещение доходило до конца алфавита, шифровальщик возвращался к его началу. Скажем, если требовалось сместить букву *z* на два символа, она становилась *b*.

Модульная арифметика является ключом к кодированию шифром Цезаря. **Модульная арифметика** — вид арифметики, когда отсчет начинается заново после достижения определенного значения. Вам уже знакома модульная арифметика, так как вы умеете определять время по часам (рис. 5.1).

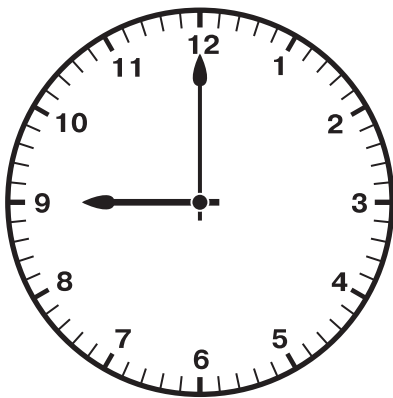


Рис. 5.1. Вы используете модульную арифметику для определения времени

Скажем, рейс из Нью-Йорка в столицу Перу Лиму отправляется в 9 часов вечера. Предположим, что оба города находятся в одном часовом поясе и полет займет 8 часов. В какое время прибудет самолет? 9 плюс 8 равно 17. Но на 12-часовом циферблате нет числа 17. Чтобы определить время прилета, вы складываете 9 и 8 (17) и вычисляете результат по модулю 12.

$$17 \% 12$$

Семнадцать делится на 12 один раз, остаток равен 5, а это значит, что рейс прибудет в 5 часов утра (рис. 5.2).

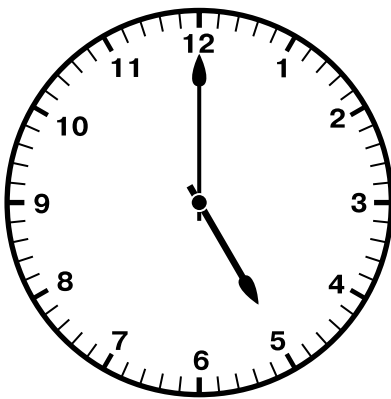


Рис. 5.2. Через восемь часов после 9 будет 5

Модульная арифметика пригодится при написании программ, включающих время. Например, если вы создаете веб-сайт для обработки времени прилетов и отлетов, вы можете использовать модульную арифметику, чтобы выяснять, когда приземлится самолет.

Теперь, понимая принцип модульной арифметики, вы можете использовать шифр Цезаря для написания функции, которая использует строку и число для сдвига каждой буквы и выводит новую зашифрованную строку:

```
import string

def cipher(a_string, key):
    uppercase = string.ascii_uppercase
    lowercase = string.ascii_lowercase
    encrypt = ''
    for c in a_string:
        if c in uppercase:
            new = (uppercase.index(c) + key) % 26
```

```
        encrypt += uppercase[new]
    elif c in lowercase:
        new = (lowercase.index(c) + key) % 26
        encrypt += lowercase[new]
    else:
        encrypt += c
return encrypt
```

Функция `cipher` принимает два параметра: `a_string` (строка, которую вы хотите зашифровать) и `key` (число, на которое вы смещаете каждую букву по алфавиту).

Вы начинаете с использования встроенного в Python модуля `string`, чтобы создать две строки, содержащие все символы алфавита в верхнем и нижнем регистрах:

```
import string

def cipher(a_string, key):
    uppercase = string.ascii_uppercase
    lowercase = string.ascii_lowercase
```

Если вы печатаете в верхнем и нижнем регистрах, ваши выходные данные выглядят следующим образом:

```
>> 'abcdefghijklmnopqrstuvwxyz'
>> 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Затем вы создаете переменную `encrypt`, которая сначала является пустой строкой, но потом будет содержать вашу зашифрованную строку:

```
encrypt = ''
```

Далее вы перебираете строку, отслеживая каждый элемент в переменной `c`:

```
for c in a_string:
```

Если символ в верхнем регистре, вы находите индекс символа в `uppercase` (который, как вы помните, содержит `ABCDEFGHIJKLMNOPQRSTUVWXYZ`). Затем добавляете `key` к этому индексу и получаете новый зашифрованный символ. Например, если символ равен `A`, а `key` равен `2`, то сначала вы извлекаете индекс для `A` из `uppercase`, который равен `0`, а затем добавляете `2`. Индекс `2` в `uppercase` — буква `C`.

Однако использование такого метода для извлечения нового символа может быть проблемным. Что произойдет, если вы переместите букву `Z` на два и более места? Индекс буквы `Z` равен `25`. Если вы прибавите `2` к `25`, то получите индекс `27`, которого нет. Поскольку `Z` — последняя буква алфавита, вам нужно

перейти к началу алфавита, чтобы получить новый зашифрованный символ. В данном примере вам нужен индекс 2, что равно C (индекс $0 + 2$).

Для решения подобной задачи используйте остаток от деления на 26 для суммы начального индекса каждого символа и `key`. Сначала вы извлекаете начальный индекс символа в `uppercase`. Далее добавляете `key` и используете оператор остатка от деления:

```
if c in uppercase:
    new = (uppercase.index(c) + key) % 26
```

Этот код работает, потому что вы используете модульную арифметику, чтобы по достижении определенного значения каждый раз возвращаться в начало. В данном случае вы возвращаетесь в начало, когда индекс превышает 25.

Как только вы получаете новый индекс для зашифрованного символа, вы смотрите, что это за символ в `uppercase`, и сохраняете его в `encrypt`.

```
encrypt += uppercase[new]
```

Если символ в нижнем регистре, вы выполняете те же операции, используя при этом `lowercase`:

```
elif c in lowercase:
    new = (lowercase.index(c) + key) % 26
    encrypt += lowercase[new]
```

Если символа нет ни в `uppercase`, ни в `lowercase`, то это специальный символ, и вы добавляете его в `encrypt` без изменений:

```
else:
    encrypt += c
```

В конце цикла вы возвращаете новую зашифрованную строку:

```
return encrypt
```

Поскольку алгоритму необходимо перебрать все буквы в строке, чтобы зашифровать ее, время его выполнения равно $O(n)$.

Словарь терминов

Анаграммы — две строки, которые содержат одни и те же буквы, но не обязательно в одинаковом порядке (регистр не имеет значения).

Палиндром — слово, которое читается одинаково слева направо и справа налево.

Списковое включение — синтаксис языка Python для создания новой, отличающейся от текущей итерации (например, другого списка).

Шифр — алгоритм шифрования и дешифрования.

Модульная арифметика — вид арифметики, в которой отсчет начинается заново после достижения определенного значения.

Практикум

1. Используйте списковое включение, чтобы из следующего списка вернуть список всех слов, содержащих более четырех символов: ["selftaught", "code", "sit", "eat", "programming", "dinner", "one", "two", "coding", "a", "tech"].

6

Математика

Не слишком беспокойтесь о своих трудностях с математикой. Могу вас заверить, мои трудности все равно больше.

Альберт Эйнштейн

В этой главе вы изучите некоторые основы математики, которые помогут вам пройти техническое интервью и улучшат ваши навыки программирования. Хотя поиск простых чисел, возможно, не пригодится вам в ежедневной работе, понимание различных алгоритмов, которые можно использовать для поиска простых чисел, сделает вас более знающим программистом. Проверка вашей способности использовать в алгоритмах оператор остатка от деления является одной из самых распространенных «ловушек» на технических собеседованиях, но этот навык может быть полезен и при написании реальных приложений. Кроме того, здесь вы познакомитесь с понятием граничных условий. Если вы пишете приложение без учета граничных условий, вы, скорее всего, создадите веб-сайт или приложение, наполненные непредвиденными ошибками. Поэтому следует знать, что представляют собой граничные условия и как к ним подготовиться.

Двоичный код

Компьютеры «мыслят» в двоичном формате. **Двоичное число** — число двоичной системы счисления (с основанием 2). **Система счисления** — система записи чисел. Для обозначения двоичных чисел используются только две цифры: 0 и 1. В двоичной системе цифра называется **битом**, что означает двоичный разряд.

Система счисления, которую вы используете для счета, называется десятичной (с основанием 10) и имеет 10 знаков (от 0 до 9). **Основание** системы счисления — это количество цифр в этой системе. Двоичная и десятичная — не единственные виды систем счисления. Есть, например, система счисления по основанию 16, которая популярна среди программистов и называется *шестнадцатеричной*.

Ниже приведено несколько примеров двоичных чисел:

100
1000
101
1101

Когда вы смотрите на эти числа, вы не знаете, по какому они основанию — 2 или 10. Например, первое число **100** может быть 100 по основанию 10 или 4 по основанию 2.

Существует несколько способов для отображения числа по основанию 2. Например, программисты часто ставят **b** перед числом, чтобы показать, что оно по основанию 2. Вот как еще можно обозначить число по основанию 2:

100b
1000₂
%100
0b100

Вес разряда — это числовое значение, которое принимает цифра в зависимости от своего места в числе. У четырехзначного числа вес разряда представлен в тысячах, сотнях, десятках и единицах. К примеру, число 1452 — это одна тысяча, плюс четыре сотни, плюс пять десятков, плюс две единицы (рис. 6.1).

Тысячи	Сотни	Десятки	Единицы
1	4	5	2

Рис. 6.1. Вес разрядов для числа 1452 по основанию 10

В десятичной системе каждый вес разряда имеет степень 10. Крайний правый вес разряда равен 10 в нулевой степени, что равно 1. Следующий вес разряда — 10 в первой степени, что равно 10. Следующий вес разряда равен 10 во второй степени (10×10), что равно 100. Следующий вес разряда равен 10 в третьей степени ($10 \times 10 \times 10$), что равно 1000 (рис. 6.2).

Тысячи	Сотни	Десятки	Единицы
1	4	5	2
10^3	10^2	10^1	10^0

Рис. 6.2. Десять в разных степенях, используемых в весах разрядов по основанию 10

Вы можете выразить число 1452 как равенство, используя его веса разрядов:

$$(1 * 10 ** 3) + (4 * 10 ** 2) + (5 * 10 ** 1) + (2 * 10 ** 0) = 1452$$

Или представить следующим образом:

$$\begin{aligned} 1 * 10 ** 3 &= 1 * 1000 = 1000 + \\ 4 * 10 ** 2 &= 4 * 100 = 400 + \\ 5 * 10 ** 1 &= 5 * 10 = 50 + \\ 2 * 10 ** 0 &= 2 * 1 = 2 \end{aligned}$$

1452

Двоичная система счисления работает так же, как десятичная, за исключением того, что в ней лишь две цифры — 0 и 1, а веса разрядов имеют степень 2, а не 10.

Крайний правый вес разряда равен 2 в нулевой степени, что равно 1. Следующий вес разряда — 2 в первой степени, что равно 2. Следующий вес разряда равен 2 во второй степени, что равно 4 (2×2). Следующий вес разряда равен 2 в третьей степени, что равно 8 ($2 \times 2 \times 2$) (рис. 6.3).

Восьмерки	Четверки	Двойки	Единицы
2^3	2^2	2^1	2^0

Рис. 6.3. Два в разных степенях, используемых в весах разрядов по основанию 2

Вот как выглядит равенство для числа 1101 по основанию 2:

$$\begin{aligned} (1 * 2 ** 3) + (1 * 2 ** 2) + (0 * 2 ** 1) \\ + (1 * 2 ** 0) = \end{aligned}$$

$$8 + 4 + 0 + 1 = 13$$

Или:

$$\begin{aligned}
 1 * 2 ** 3 &= 1 * 8 = 8 + \\
 1 * 2 ** 2 &= 1 * 4 = 4 + \\
 0 * 2 ** 1 &= 0 * 2 = 0 + \\
 1 * 2 ** 0 &= 1 * 1 = 1 \\
 \hline
 &13
 \end{aligned}$$

Как видите, 1101 в двоичной системе является числом 13 в десятичной.

В десятичной системе вы считаете, начиная с нуля: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. В этот момент у вас заканчиваются цифры. Чтобы представить следующее число, как вам известно, вы создаете 10, используя 2 цифры. Вы представляете 10 с помощью цифры 1, за которой следует 0.

Когда вы считаете в двоичном коде, вы также начинаете с нуля.

0

Как и в десятичной системе, следующее число 1.

1

После 1 у вас заканчиваются цифры. Это означает, что вам нужно использовать две цифры, чтобы представить число 2, так же, как вы их использовали в десятичной системе, чтобы представить число 10.

В двоичном коде вы обозначаете 2 с помощью 1 и 0:

10

Ноль означает отсутствие единиц, а 1 означает одну двойку.

Как же обозначить число 3 в двоичной системе?

11

Первая справа цифра 1 означает одну единицу, а вторая справа 1 означает одну двойку. Если вы прибавите 2 к 1, то получите 3.

Следующая цифра 4 в двоичной системе выглядит следующим образом:

100

Первый 0 справа означает отсутствие единиц, второй 0 справа означает отсутствие двоек, а 1 означает одну четверку. Сложите их и получите 4.

Побитовые операторы

Обычно, когда вы программируете или оперируете числами, вы работаете с целыми числами и числами с плавающей точкой, например с такими, как **100** и **10.5**. Однако бывает полезно поработать и с двоичными числами. Используя двоичные числа, можно быстро решать определенные задачи: скажем, выяснить, является ли число степенью 2.

Для работы с двоичными числами в Python можно использовать метод `bin`.

```
print(bin(16))
```

```
>> 0b1000
```

Когда вы печатаете `bin(16)`, результат равен `0b10000`, что в двоичной системе означает число 16. Как вы узнали ранее, `0b` показывает, что число записано в двоичном формате.

Побитовый оператор — это оператор, который можно использовать в выражении с двумя двоичными числами. Например, побитовый оператор AND в Python выполняет булеву арифметику бит за битом. Если оба бита равны 1 (`True`), Python возвращает число 1; в противном случае Python возвращает 0 (`False`). Логика для каждого бита у побитового AND та же, что и для ключевого слова `and` в Python. Как вы уже знаете, если вы используете ключевое слово `and` в Python и обе стороны выражения принимают значение `True`, то Python возвращает `True`.

```
print(1==1 and 2==2)
```

```
>> True
```

Если обе стороны принимают значение `False`, он возвращает `False`.

```
print(1==2 and 2==3)
```

```
>> False
```

Если одна сторона принимает значение `True`, а другая `False`, Python также возвращает `False`.

```
print(1==1 and 1==2)
```

```
>> False
```

Рассмотрим пример применения побитового AND. Допустим, у вас есть два целых числа 2 и 3. Два в двоичной системе равно `0b10`, а 3 равно `0b11`. Первый справа бит цифры 2 — это 0, первый справа бит цифры 3 — это 1.

```
10 # 2
11 # 3
—
0
```

Применение побитового AND к этим битам приводит к 0, потому что есть значения `True` и `False`, которые возвращают `False` (помните: 0 равно `False`, а 1 равно `True`). Применение побитового AND ко второму набору битов приводит к 1, так как обе цифры принимают значение `True`, и Python возвращает `True`.

```
10 # 2
11 # 3
—
10
```

В данном случае побитовая операция AND выводит `0b10`, что соответствует цифре 2 в двоичном коде (вы скоро узнаете, чем это полезно).

На языке Python побитовый оператор AND — это знак амперсанда (&). Вот как вы можете применить побитовое AND к двоичным цифрам `0b10` и `0b11` на языке Python:

```
print(0b10 & 0b11)

>> 2
```

Вам не нужно использовать побитовое AND для двоичных чисел.

```
print(2 & 3)

>> 2
```

В данном случае вы применили оператор AND для десятичных чисел, но Python использует двоичные значения, определяющие 2 и 3, для выполнения операции.

В Python также есть побитовый оператор OR, который работает побитово и возвращает значение 1, когда один или оба бита принимают значение `True`, и возвращает `False`, когда оба принимают значение `False`: точно так же, как ключевое слово `or` в Python. Посмотрим, что произойдет, если вы используете побитовый оператор OR для чисел 2 и 3. Применение оператора OR для первых двух битов приводит к значению 1, потому что один из битов равен `True`.

```
10 # 2
11 # 3
—
1
```

Когда вы применяете побитовый оператор OR для второго набора битов, Python возвращает 1, потому что оба бита принимают значение True (1).

```
10 # 2
11 # 3
—
11
```

Как видите, результат оператора OR для чисел 2 и 3 равен 0b11, что соответствует десятичному числу 3.

На языке Python побитовый оператор OR представлен в виде вертикальной черты (|).

```
print(2 | 3)
```

```
>> 3
```

Двоичные операторы, с которыми вы уже познакомились, встречаются наиболее часто, однако есть и другие: побитовые XOR, NOT, сдвиг вправо и сдвиг влево. О них вы можете узнать из документации к языку Python.

Рассмотрим некоторые ситуации, когда побитовые операторы бывают полезны. Вы можете использовать побитовое AND для проверки четности или нечетности числа. У четного числа, такого как 2, всегда стоит 0 в конце, тогда как число 1 всегда имеет 1 в конце (и содержит только одну двоичную цифру — 1).

```
10 # 2
11 # 1
```

Когда вы применяете побитовый оператор AND для четного числа и числа 1, Python всегда возвращает False, потому что четное число заканчивается на 0, а у числа 1 лишь одна двоичная цифра 1.

```
10 # 2
11 # 1
--
0
```

С другой стороны, когда вы применяете побитовый оператор AND для нечетного числа и числа 1, Python всегда будет возвращать значение True, потому что нечетное число заканчивается на 1, а у числа 1 лишь одна двоичная цифра 1.

```
11 #3
11 #1
--
1
```

Поскольку у числа 1 лишь одна двоичная цифра, не имеет значения, сколько двоичных цифр у числа, которое вы проверяете на четность, — одна или тысяча.

Поскольку у числа 1 лишь одна двоичная цифра, вы производите только одно сравнение: последней двоичной цифры числа и 1.

Вот как проверить, является число четным или нечетным, используя побитовый оператор AND в Python:

```
def is_even(n):  
    return not n & 1
```

В функции `is_even` вы возвращаете `not n & 1`. Код `n & 1` использует побитовый оператор AND для `n` и 1. Затем вы используете `not`, чтобы поменять результат на противоположный тому, каким он был бы, потому что, когда вы используете побитовый оператор AND для четного числа и 1, он возвращает `False`, а это означает, что число четное. В данном случае вы хотите, чтобы функция возвращала значение `True` для демонстрации того, что число четное, поэтому вы используете `not` для переключения `True` на `False` и `False` на `True`.

Побитовый оператор AND также можно использовать для определения того, является ли целое число степенью 2. Каждое число, являющееся степенью 2, имеет одну 1 в своем двоичном представлении, потому что двоичный код имеет основание 2, а это означает, что у любой степени 2 лишь одна цифра 1: например, число 8 в двоичном коде представлено как `0b1000`. И наоборот, число, которое на 1 меньше степени 2, содержит только биты с 1: число 7, на 1 меньшее 8, в двоичном коде имеет вид `0b111`.

Когда вы примените побитовый оператор AND для этих двух двоичных чисел, вы увидите, что весь итоговый двоичный код состоит из нулей, если первое число является степенью 2.

```
1000 # 8  
0111 # 7  
-----  
0000
```

Если число не является степенью 2, итог будет содержать как минимум одну двоичную цифру, равную 1.

```
0111 # 7  
0110 # 6  
-----  
0001
```

Вот как использовать побитовый оператор AND в Python, чтобы определить, является ли число степенью 2:

```
def is_power(n):  
    if n & (n - 1) == 0:  
        return True  
    return False
```

Функция `is_power` принимает рассматриваемое число. Внутри функции вы используете оператор `if`, чтобы проверить, равно ли использование оператора AND для `n` и `n - 1` нулю. Если это так, то `n` является степенью 2 и вы возвращаете `True`. В противном случае вы возвращаете `False`.

FizzBuzz

FizzBuzz — одна из классических задач на собеседовании. Однажды я слышал историю об инженерере, который проходил собеседование на должность главного инженера-программиста, и там его попросили решить FizzBuzz. Он не смог и очень смутился. Но не переживайте, в этом разделе вы как раз научитесь решать данную задачу, и с вами подобного конфуза не случится.

Вот в чем состоит задача FizzBuzz: написать программу, которая выводит числа от 1 до 100; если число кратно 3, вывести `Fizz`; если число кратно 5, вывести `Buzz`; если число кратно 3 и 5, вывести `FizzBuzz`.

Ключом к решению этой задачи является использование оператора остатка от деления, который делит два значения и возвращает остаток. Если остаток равен 0, вы понимаете, что делимое (число, которое вы делили) кратно делителю (числу, на которое делили). Например, `6 % 3` делит 6 на 3 и возвращает остаток 0.

```
print(6 % 3)
```

```
>> 0
```

Поскольку остатка нет, вы знаете, что 6 кратно 3.

Когда вы оцениваете `7 % 3`, остаток есть, поэтому вы знаете, что 7 не кратно 3.

```
print(7 % 3)
```

```
>> 1
```

Чтобы решить задачу FizzBuzz, вы перебираете числа от 1 до 100 и используете оператор по модулю, проверяя, является ли число кратным 3 и 5, только 3 или только 5.

Вот как это сделать:

```
def fizzbuzz(n):  
    for i in range(1, n + 1):  
        if i % 3 == 0 and i % 5 == 0:  
            print('FizzBuzz')  
        elif i % 3 == 0:  
            print('Fizz')
```

```
elif i % 5 == 0:
    print('Buzz')
else:
    print(i)
```

Даже если вы хотите найти числа от 1 до 100, лучше всего ввести число `n` вместо жесткого кодирования 100. Введение `n` сделает вашу программу более гибкой, если вы захотите запустить ее с другим числом. Итак, функция `fizzbuzz` принимает `n` в качестве параметра.

```
def fizzbuzz(n):
```

Для начала вы используете цикл `for` для перебора каждого числа от 1 до `n + 1`.

```
for i in range(1, n + 1):
```

Затем вы используете условный оператор с оператором остатка от деления, чтобы определить, делится ли число `i` и на 3, и на 5. Если это так, выводите `FizzBuzz`.

```
if i % 3 == 0 and i % 5 == 0:
    print('FizzBuzz')
```

Далее вы используете еще один условный оператор, чтобы проверить, делится ли число на 3. Если это так, выводите `Fizz`.

```
elif i % 3 == 0:
    print('Fizz')
```

Затем вы используете еще один условный оператор, чтобы проверить, делится ли число на 5. Если это так, выводите `Buzz`.

```
elif i % 5 == 0:
    print('Buzz')
```

Если число не делится ни на 3, ни на 5, ни на оба этих числа, выводите само число.

```
else:
    print(i)
```

Когда вы запустите программу, вы увидите следующее: числа, которые делятся на 3, например 6 и 27, заменяются на `Fizz`; числа, кратные 5, например 10 и 85, заменяются на `Buzz`; те числа, которые делятся на оба числа, например 15 и 30, заменяются на `FizzBuzz`.

```
>> 1 2 Fizz 4 Buzz Fizz 7 8...Buzz Fizz 97 98 Fizz Buzz
```

Вашему алгоритму требуется `n` шагов, поэтому он линейный. Если вы введете 100, алгоритму потребуется 100 шагов, если введете 1000, потребуется 1000 шагов.

Оператор остатка от деления был ключом к решению данной задачи. Однако он может быть полезен не только на техническом интервью. Оператор остатка

от деления также используется при написании реальных приложений. Например, у вас имеется текстовый файл длиной 50 000 строк, и вы можете на страницу поместить 49 строк. Сколько текста будет на последней странице? На последней странице будет 20 строк, потому что $50000 \% 49 = 20$. А что, если у вас база данных с 20 000 наименований и вы хотите что-то сделать с каждым вторым наименованием? Один из способов — перебирать каждый элемент и менять только элементы с четным индексом.

Наибольший общий делитель

Наибольший общий делитель — наибольшее положительное целое число, на которое делятся без остатка два или более целых числа. В этом разделе вы узнаете, как для двух целых чисел, например 20 и 12, найти их наибольший общий делитель.

Числа 20 и 12 вы можете разделить без остатка на 1, 2 и 4. Поскольку 4 — самое большое число, оно и является наибольшим общим делителем.

Делители числа 20: 1, 2, 4, 5, 10.

Делители числа 12: 1, 2, 3, 4, 6.

Один из алгоритмов для нахождения наибольшего общего делителя для двух чисел — проверка всех возможных делителей на предмет того, на какие из них без остатка делятся оба числа. Например, чтобы найти наибольший общий делитель для чисел 20 и 12, вы начинаете с деления их на 1, затем на 2, на 3 и т. д. Вам не нужно проверять числа, которые больше самого меньшего из двух чисел, потому что оно не может делиться без остатка на превышающее его число. Например, число больше 12 не разделит 12 без остатка.

Ниже представлен код на языке Python для вашего алгоритма:

```
def gcf(i1, i2):
    gcf = None
    if i1 > i2:
        smaller = i2
    else:
        smaller = i1
    for i in range(1, smaller + 1):
        if i1 % i == 0 and i2 % i == 0:
            gcf = i
    return gcf

gcf(20, 12)
```


Функция `gcf` принимает в качестве параметров два целых положительных числа, для которых вы ищете наибольший общий делитель.

```
def gcf(i1, i2):
```

Внутри функции вы определяете, какое из двух целых чисел меньше, и присваиваете его значение переменной `smaller`, чтобы прекратить проверку делителей, дойдя до этого числа.

```
    if i1 > i2:
        smaller = i2
    else:
        smaller = i1
```

Затем вы используете цикл `for`, чтобы проверить каждый делитель от 1 до значения переменной `smaller` плюс 1 (чтобы проверить и наименьшее число).

```
    for i in range(1, smaller + 1):
```

Далее вы используете оператор `if`, чтобы увидеть, делит ли делитель без остатка оба целых числа. Если это так, вы присваиваете значение делителя переменной `gcf`.

```
        if i1 % i == 0 and i2 % i == 0:
            gcf = i
```

Но то, что вы нашли один общий делитель, еще не означает, что вы нашли наибольший общий делитель. Если вы найдете еще один делитель — больший, то в следующий раз в своем цикле вы присвоите его переменной `gcf`. Таким образом, когда цикл закончится, `gcf` будет содержать наибольший делитель.

Однако у вашего кода есть проблема. Что, если одно из целых чисел равно 0?

```
print(gcf(0, 22))
```

```
>> None
```

Программа вернет неверный ответ, если одно из целых чисел будет равно 0.

Неспособность кода справиться с числом 0 является примером **граничного условия** — входных данных, находящихся за пределами тех входных данных, которые ожидаемо должна была получить ваша программа. Если при вычислении наибольшего общего делителя для двух чисел одно из чисел равно 0, то наибольший общий делитель будет равен второму целому числу. Например, наибольший общий делитель для чисел 0 и 12 равен 12.

Когда вы пишете алгоритм, вам всегда нужно учитывать непредвиденные входные данные, которые могут нарушить его работу. В данном случае ваш алгоритм неверен, так как во входных данных есть 0. Вот как можно изменить программу, чтобы она возвращала правильные выходные данные, если одно из двух целых чисел равно 0:

```
def gcf(i1, i2):
    if i1 == 0:
        return i2
    if i2 == 0:
        return i1

    if i1 > i2:
        smaller = i2
    else:
        smaller = i1

    for divisor in range(1, smaller + 1):
        if(i1 % divisor == 0) and (i2 % divisor == 0):
            gcf = divisor

    return gcf
```

Ваша программа также не может обрабатывать отрицательные числа, поэтому в начале вам следует добавить проверку на то, что оба числа положительные.

```
def gcf(i1, i2):
    if i1 < 0 or i2 < 0:
        raise ValueError("Numbers must be positive.")
    if i1 == 0:
        return i2
    if i2 == 0:
        return i1

    if i1 > i2:
        smaller = i2
    else:
        smaller = i1

    for divisor in range(1, smaller+1):
        if(i1 % divisor == 0) and (i2 % divisor == 0):
            gcf = divisor

    return gcf
```

Код для нахождения наибольшего общего делителя линейен, так как алгоритм решает задачу за n шагов. Линейный код — это неплохо, но существует более удачный способ решения данной задачи.

Алгоритм Евклида

Более эффективным решением задачи поиска наибольшего общего делителя является алгоритм Евклида. Для начала вы делите одно число, x , на другое число, y , чтобы найти остаток. Затем вы снова делите, используя остаток в качестве y , а предыдущий y в качестве нового x . Вы продолжаете эту процедуру, пока остаток не будет равен 0. Последний делитель и есть наибольший общий делитель.

Например, чтобы найти наибольший общий делитель для чисел 20 и 12, вы сначала делите 20 на 12 и получаете остаток 8. Далее вы делите 12 на остаток — то есть 12 на 8, получая остаток 4. Теперь вы делите 8 на 4. На этот раз остатка нет, что означает, что 4 и есть наибольший общий делитель.

$$20 / 12 = 1, \text{ остаток } 8$$

$$12 / 8 = 1, \text{ остаток } 4$$

$$8 / 4 = 2, \text{ остаток } 0$$

Вот как выглядит алгоритм Евклида на языке Python:

```
def gcf(x, y):
    if y == 0:
        x, y = y, x
    while y != 0:
        x, y = y, x % y
    return x
```

Ваша функция `gcf` принимает в качестве параметров два числа, для которых вы ищете наибольший общий делитель.

В первой строке кода вы задаете граничное условие. Если y равен 0, Python вызовет исключение позже в вашей программе, пытаясь разделить на 0. Если y равен 0, вы меняете местами содержимое переменных x и y , чтобы решить задачу.

```
if y == 0:
    x, y = y, x
```

Далее вы создаете цикл `while`, который повторяется до тех пор, пока y не станет равным 0.

```
while y != 0:
```

Внутри цикла `while` в этом выражении меняются местами значения x и y на y и остаток от деления x на y .

```
x, y = y, x % y
```

Когда цикл `while` заканчивается, это означает, что `x % y` вернуло остаток 0. Затем вы возвращаете значение `x`, которое представляет собой наибольший общий делитель для `x` и `y`.

```
return x
```

Можно математически доказать, что данный алгоритм является логарифмическим, а не линейным, а это значительно повышает его производительность при работе с большими числами по сравнению с вашим исходным алгоритмом поиска наибольшего общего делителя двух чисел.

Простые числа

Простое число — положительное целое число, которое делится только на себя и на 1. Такие числа, как 2, 3, 5, 7 и 11, являются простыми. В этом разделе вы узнаете, как написать функцию, которая определяет, является ли число простым.

Вот как это сделать:

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

Функция `is_prime` принимает число `n`, которое вы проверяете.

```
def is_prime(n):
```

Затем вы используете цикл `for` для перебора каждого числа от 2 до `n`. Вы начинаете с числа 2 (не с 1), потому что простые числа делятся на 1.

```
for i in range(2, n):
```

Если `n` равно 10, ваш код будет перебирать числа от 2 до 9, а это означает, что число 10 будет пропущено (так нужно, потому что простые числа делятся на самих себя).

Затем вы используете оператор остатка от деления, чтобы проверить, есть ли остаток, когда вы делите `n` на `i`. Если остатка нет, вы нашли делитель, отличный от 1 и самого числа. Это означает, что `n` не является простым числом, поэтому вы возвращаете `False`.

```
if n % i == 0:
    return False
```

Если вы перебрали весь диапазон чисел, не найдя делителя, значит, n является простым числом, поэтому вы возвращаете `True`.

```
return True
```

Поскольку для завершения алгоритму требуется n шагов, это линейный алгоритм.

Вы можете его улучшить, заканчивая цикл на квадратном корне из n вместо $n - 1$.

```
import math

def is_prime(n):
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

Вот почему вы можете остановиться на квадратном корне из n . Если $a * b == n$, то либо a , либо b должны быть меньше или равны квадратному корню из n . Потому что если и a , и b больше квадратного корня из n , тогда $a * b$ должно быть больше n , а не равным n . Таким образом, у вас не может быть двух чисел a и b , перемноженных, чтобы стать n , где оба больше квадратного корня из n , так как тогда $a * b$ будет больше n . Поскольку один делитель должен быть меньше или равен квадратному корню из n , вам не нужно осуществлять проверку до n . Вместо этого вы можете остановиться на первом целом числе, большем, чем квадратный корень из n . Если вы не найдете ни одного числа меньше квадратного корня из n или равного ему, которое делит n без остатка, значит, вы не найдете ни одного числа, которое это делает.

Вы легко можете изменить свою программу так, чтобы она выводила список простых чисел в пределах диапазона чисел.

```
def is_prime(n):
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def find_primes(n):
    return [i for i in range(2, n) if is_prime(i)]
```

Сначала вы создаете новую функцию под названием `find_primes`, которая принимает n в качестве параметра, представляющего собой число, до значения которого вы ищете простые числа.

Затем вы используете списковое включение, чтобы перебрать числа от 2 до n , и добавляете число в новый список, только если функция `is_prime` возвращает `True`.

```
return [is_prime(i) for i in range(2, n)]
```

Ваш алгоритм для поиска всех простых чисел в заданном диапазоне вызывает линейную функцию `is_prime` внутри цикла, а это означает, что его временная сложность равна $O(n^2)$, и это не очень эффективно. Но алгоритм, с которым вы познакомились в этом разделе, не единственный алгоритм для поиска простых чисел. Существуют и другие, более сложные алгоритмы, имеющие эффективную временную сложность, и вы можете их также изучить.

Словарь терминов

Наибольший общий делитель — наибольшее положительное целое число, на которое делятся без остатка два и более целых числа.

Граничное условие — входные данные за пределами тех входных данных, которые ожидаемо должна была получить ваша программа.

Простое число — положительное целое число, что делится только на себя и на 1.

Побитовый оператор — оператор, который вы используете в выражении с двумя двоичными числами.

Двоичное число — число двоичной системы счисления (с основанием 2).

Система счисления — система записи чисел.

Бит — в двоичной системе цифра называется битом, что означает двоичный разряд.

Основание системы счисления — количество цифр этой системы.

Вес разряда — числовое значение, которое принимает цифра в зависимости от своей позиции в числе.

Практикум

1. Изучите тему и напишите другой способ нахождения простых чисел.

7

Вдохновение самоучки: Маргарет Гамильтон

Всем моим друзьям, у которых есть младшие братья и сестры, которые учатся в колледже или старшей школе, мой совет номер один: пусть учатся программировать.

Марк Цукерберг

В наши дни существует множество способов научиться программированию и легко забыть, что так было не всегда. Маргарет Гамильтон, одна из первых программистов космической миссии «Аполлон» и одна из лучших программистов-самоучек за все время, оставила свой след в области высоких технологий задолго до того, как курсы программирования стали доступны повсеместно.

Несмотря на то что Гамильтон получила диплом по математике в Мичиганском университете, программированию она обучалась исключительно самостоятельно. В 50-е и 60-е годы прошлого столетия все были самоучками, потому что компьютерной науки, какой мы знаем ее сейчас, еще не существовало, и программистам приходилось учиться самим. В то время даже термина «*разработка программного обеспечения*» еще не придумали: Гамильтон помогла его создать! После окончания университета в 1960 году Маргарет начала свою карьеру программиста в Массачусетском технологическом институте, работая над проектом под названием *Whirlwind* («Вихрь») для прогнозирования погоды. В процессе работы она помогала создавать код для первого в мире портативного компьютера.

Успех Гамильтон в проекте *Whirlwind* привел ее в проект компании SAGE — организации, помогающей обнаруживать потенциальные авиаудары со стороны Советского Союза во времена холодной войны. Если вы фанат *Star Trek*, вы знаете, что такое Кобаяси Мару — тренировочное задание для курсантов Звездного флота, в котором нельзя было победить. Будущие офицеры не могли выиграть, но они могли проявить важные черты характера, делая выбор. Гамильтон столкнулась с проблемой Кобаяси Мару в реальной жизни и, как и знаменитый капитан Кирк, смогла ее преодолеть. Каждому новичку в SAGE давали практически неразрешимую программу, которую нужно было запустить. Создатель программы усложнил ее тем, что написал комментарии на греческом и латинском языках. Гамильтон была первым инженером, запустившим программу, что обеспечило ей место в компании.

Благодаря способности Маргарет решать сложные задачи ее пригласили принять участие в космическом проекте NASA «Аполлон», кульминацией которого стала высадка Нила Армстронга, Базза Олдрина и Майкла Коллинза на Луну в 1969 году. Хотя наибольшее признание за эту историческую миссию получили астронавты, но в проекте было задействовано около 400 000 человек, и Маргарет Гамильтон среди прочих внесла свой ценный вклад в успешность проекта.

Одним из наиболее значимых достижений ее команды была разработка системы оповещения экипажа о чрезвычайных ситуациях. Этот процесс был важной частью успеха высадки на Луну и существенно выиграл от настойчивости Гамильтон в проведении тщательных испытаний. Доктор Пол Курто, который номинировал Маргарет на премию NASA за выдающиеся достижения в области космоса, заявил, что ее работа является «основой для разработки сверхнадежного программного обеспечения», потому что на сегодняшний день никто не нашел ошибки в программном обеспечении проекта «Аполлон».

Двадцать второго ноября 2016 года президент США Барак Обама наградил Маргарет Гамильтон Президентской медалью свободы в знак признания ее выдающихся достижений в области программного обеспечения, укрепив Маргарет в статусе одного из величайших программистов-самоучек всех времен.



Структуры данных

Глава 8. Что такое структуры данных

Глава 9. Массивы

Глава 10. Связные списки

Глава 11. Стеки

Глава 12. Очереди

Глава 13. Хеш-таблицы

Глава 14. Двоичные деревья

Глава 15. Двоичные кучи

Глава 16. Графы

Глава 17. Вдохновение самоучки: Илон Маск

Глава 18. Дальнейшие шаги

8

Что такое структуры данных

Алгоритмы + структуры данных = программы.

Никлаус Вирт

Структура данных — способ организации данных в компьютере, позволяющий программистам эффективно их использовать. В первой части книги вы уже научились работать с некоторыми встроенными в Python структурами данных, например со списками и словарями. В этом разделе о структурах данных вы узнаете больше: познакомитесь с другими их видами, о которых, возможно, еще не слышали, такими как массивы, связанные списки, стеки, очереди, деревья, кучи, графы и хеш-таблицы. У каждой из перечисленных структур данных есть свои преимущества и недостатки. Выбор наилучшей структуры для вашей программы зависит от типа решаемой задачи и от того, какую цель вы ставите при оптимизации. Во второй части книги вы узнаете о плюсах и минусах различных структур данных и в дальнейшем, когда будете писать приложения, сможете выбирать наиболее подходящую для ваших целей структуру. Кроме того, вы узнаете ответ на один из самых распространенных вопросов технического интервью относительно структуры данных, и когда вам его зададут во время собеседования, с легкостью на него ответите.

Нельзя стать отличным программистом без глубокого понимания структур данных, потому что программирование — это написание алгоритмов и выбор подходящей структуры данных для этого алгоритма. Вот откуда идет знаменитое

высказывание Никлауса Вирта: «Алгоритмы + структура данных = программы». Алгоритм говорит компьютеру, что делать, а структура данных говорит компьютеру, как хранить данные из алгоритма. Линус Торвалдс, создатель Linux, подчеркнул важность структуры данных своим известным выражением: «На самом деле я уверен, что разница между плохим программистом и хорошим заключается в том, что он считает более важным — свой код или свою структуру данных. Плохие программисты беспокоятся о коде. Хорошие программисты заботятся о структурах данных и их взаимосвязях». Я хочу, чтобы вы стали хорошим программистом, вот почему в оставшейся части книги мы сфокусируемся на структуре данных.

Абстрактный тип данных — это характеристика структуры данных, тогда как структура данных — фактическая реализация. Скажем, список является абстрактным типом данных: он описывает структуру данных, содержащую группу элементов, где каждый элемент имеет позицию относительно других. В списке также есть операции для управления элементами, такие как добавление и удаление элементов. Когда вы работаете со списком в Python, вы используете структуру данных, а не абстрактный тип данных, потому что структура данных является фактической реализацией абстрактного типа данных. Например, в Python может быть две разные структуры данных, реализованные совершенно разными способами, но обе основанные на абстрактном типе данных — списке.

Программисты классифицируют структуры данных по различным признакам, например по их линейности/нелинейности. **Линейная структура данных** упорядочивает элементы данных последовательно, в то время как **нелинейная структура данных** связывает данные непоследовательно. Например, списки в Python являются линейными структурами данных: у каждого элемента может быть один предшествующий элемент и один последующий. Напротив, графы (подробнее о которых вы узнаете позднее) представляют собой нелинейные структуры: в них каждый элемент может быть связан со множеством других элементов.

Обход структуры данных подразумевает переход от первого элемента в структуре к последнему. В линейной структуре данных можно легко перейти от первого элемента к последнему без отката (движения назад), в то время как в нелинейной структуре данных часто приходится возвращаться к предыдущему состоянию. Линейная структура считается более удобной для доступа к отдельным элементам, потому что в нелинейной для этого обычно требуется использовать откаты или рекурсию. По линейной структуре можно легко перемещаться, поэтому в сравнении с нелинейной структурой вносить изменения в каждый ее элемент

намного удобнее. Поскольку нет необходимости использовать откат для доступа к каждому элементу, линейные структуры данных проще и разрабатывать, и использовать. Несмотря на это, нелинейные структуры данных могут быть более эффективны для определенных типов задач. Их можно использовать для хранения данных, которые было бы затруднительно хранить в линейной структуре, например контактов в социальной сети.

Программисты также делят структуры данных на статические и динамические. У **статической структуры данных** фиксированный размер, в то время как **динамическая структура данных** может увеличиваться или уменьшаться в размерах. Как правило, вы определяете размер статической структуры данных, когда создаете ее в своей программе. Как только вы это сделали, размер структуры данных зафиксирован и вы не можете его изменить, но можете изменить значение данных внутри нее. В Python нет статических структур данных, но в более низкоуровневых языках программирования, например в C, они есть.

Одна из проблем статических структур данных заключается в том, что под них необходимо выделить определенный объем памяти. **Память компьютера** — это место, где компьютер хранит данные. Существуют различные виды памяти, и подробное рассмотрение каждого из них выходит за рамки данной книги, но в этой главе под памятью мы будем понимать место хранения данных с адресом, который компьютер может использовать для последующего обращения к этим данным. С одной стороны, если количество элементов данных, с которыми вы работаете, оказывается меньше выделенного вами размера, вы тратите впустую объем памяти. С другой — вы не сможете добавить больше элементов, чем вмещает объем, который был задан вашей структуре данных. Часто единственный способ добавить больше элементов в статическую структуру данных — это выделить для новой структуры память достаточного объема, чтобы она могла вместить и старые, и новые элементы, а затем скопировать данные из старой структуры во вновь выделенную память вместе с новыми элементами. Вот почему статическая структура не лучший выбор, если вы не знаете заранее количество элементов, которые потребуется в ней хранить. Однако если вы точно знаете, каким будет это количество, и уверены, что оно не изменится, тогда статическая структура данных, как правило, превосходит по эффективности динамическую. Например, если вы осуществляете поддержку списка функций отмены для программы, которая позволяет пользователям отменить до десяти операций, применение статической структуры данных будет уместно.

Целый ряд структур данных могут быть и статическими, и динамическими. Например, массивы (которые мы обсудим в следующей главе) часто являются

статическими структурами данных, но многие современные языки программирования, в частности Python, предлагают динамические массивы (списки).

В отличие от статической, размер динамической структуры данных можно с легкостью изменить: компьютер выделяет дополнительную память, как только добавляется больше элементов. Когда вы удаляете элементы, компьютер освобождает этот объем памяти для других данных. Благодаря своему адаптивному размеру динамические структуры позволяют эффективно добавлять и удалять элементы данных и рационально использовать ресурсы памяти. Тем не менее доступ к элементам в динамической структуре может быть медленнее, чем в статической, а для хранения определенного количества элементов в динамической структуре часто требуется больше памяти, чем для хранения такого же количества элементов в статической структуре данных. Если вы имеете дело с неопределенным объемом данных, особенно в ситуациях, когда объем памяти ограничен, динамическая структура будет хорошим выбором.

Пока вы не пишете низкоуровневый код для операционной системы или другого проекта, где нужно «выжимать» всю возможную производительность, вы вряд ли потратите много времени на выбор между статической и динамической структурами данных. Скорее всего, больше времени вы потратите на принятие решения о линейности или нелинейности структуры данных, а потом на обдумывание того, какую именно структуру нужно использовать. Как вы узнали ранее, у различных структур данных есть свои плюсы и минусы. Они связаны в основном с их эффективностью при вставке, удалении, поиске и сортировке данных, а также с тем, насколько эффективно они используют пространство памяти. Например, проверка нахождения элемента в словаре Python — вероятно эффективная операция, даже если в нем миллиард фрагментов данных. И совершенно противоположная ситуация — поиск фрагмента данных в графе. В следующей главе мы более подробно рассмотрим, когда и как использовать структуры данных.

Словарь терминов

Структура данных — способ организации данных в компьютере, позволяющий программистам эффективно их использовать в программах.

Линейная структура данных — структура данных, упорядочивающая элементы данных последовательно.

Нелинейная структура данных — структура данных, которая связывает данные непоследовательно.

Обход — проход от первого элемента в структуре данных к последнему.

Статическая структура данных — структура данных с фиксированным размером.

Динамическая структура данных — структура данных, которая может увеличиваться или уменьшаться в размерах.

Память компьютера — место, где компьютер хранит данные.

Практикум

1. Составьте список всех структур данных, которые вы использовали при работе с Python.

9

Массивы

Мы приветствуем кодеров, дизайнеров и программистов, которые уже усердно трудятся на своих рабочих местах, и призываем каждого студента, который не может решить, стоит ли ему изучать курс computer science, попробовать его посетить.

Майкл Блумберг, бывший мэр Нью-Йорка

Список — это абстрактный тип данных, описывающий структуру данных, в которой хранятся упорядоченные значения. У списков обычно есть операции для создания нового, пустого списка, проверки его пустоты, вставки элемента в начало, добавления элемента в конец и доступа к элементу по индексу. Вы уже знакомы со списками в Python — одной из многочисленных реализаций абстрактного типа данных. Массивы — еще одна реализация данного типа. В этой главе мы поговорим о них.

Массив — это структура данных, которая сохраняет элементы с индексами в непрерывном блоке памяти. Массивы часто однородны и статичны. **Однородная структура данных** — структура, которая может хранить элементы только одного типа данных, например целые числа (integer) или строки. **Статическая структура данных** — структура данных, размер которой нельзя изменить после ее создания. Когда вы создаете массив на таком низкоуровневом языке программирования, как C, вы решаете, сколько элементов конкретного типа данных вы хотите в нем хранить. После чего компьютер назначает блок памяти для вашего массива на основе количества элементов и того, сколько памяти требуется одному элементу этого типа данных. Блок памяти состоит из элементов, хранящихся один за одним в памяти компьютера.

Список Python представляет собой **неоднородный массив переменной длины**. **Массив переменной длины** — это массив, размер которого может измениться после создания. **Гетерогенный массив** — массив, содержащий разные типы данных. Гвидо ван Россум написал Python на языке C, но Python скрывает сложности создания и обработки массивов. Вместо этого он предоставляет вам структуру данных списка, которую вы можете использовать, не беспокоясь о том, чтобы заранее задавать длину списка или указывать типы данных, которые он может содержать.

На рис. 9.1 приведен пример того, как компьютер хранит массив в своей памяти.



Рис. 9.1. Пример данных в массиве

Вы можете получить доступ к каждому элементу этого массива с уникальным целочисленным индексом. Как правило, первый индекс в массиве равен 0. Однако разные языки программирования используют и разные индексные схемы. И Python, и C используют индексацию, основанную на нуле, но некоторые другие языки, например MATLAB и Fortran, работают с индексацией, основанной на единице (первый элемент имеет индекс 1). Некоторые языки позволяют использовать любое целочисленное значение для индекса первого элемента. Адрес ячейки памяти первого элемента массива называется **базовым адресом**. Когда компьютеру требуется добавить новый элемент в массив, адрес ячейки памяти, в которую он должен поместить элемент, рассчитывается с помощью формулы:

$$\text{base_address} + \text{index} * \text{size_of_data_type}$$

Компьютер принимает базовый адрес и добавляет его к индексу, умноженному на объем памяти, потребляемый этим типом данных в массиве. Такую же формулу компьютер использует для поиска элемента в массиве.

Массивы могут быть одномерными и многомерными. В **одномерном массиве** вы получаете доступ к каждому элементу массива по целочисленному индексу:

```
array = [1, 2, 3]
print(array[0])
```

```
>> 1
```


В **многомерном массиве** вы получаете доступ к элементам массива с помощью двух индексов (целочисленных индексов для каждого измерения):

```
multi_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(array[1][2])
```

```
>> 6
```

Независимо от того, одномерный массив или многомерный, компьютер хранит его элементы в непрерывном блоке памяти и для доступа к элементам использует математическую формулу, которая сопоставляет индекс с ячейкой памяти.

Производительность массива

Вы можете получить доступ к любому элементу массива и изменить его за постоянное время. Предположим, вы ищете в массиве данные с индексом 3. В таком случае вашему компьютеру потребуется извлечь информацию лишь из одной ячейки памяти, даже если в массиве миллион элементов. Поиск по несортированному массиву равен $O(n)$, потому что вам надо проверить каждый элемент массива. Тем не менее вы можете отсортировать массив, например список имен и адресов или большой набор чисел, — в таком случае временная сложность для поиска по массиву будет равна $O(\log n)$. В табл. 9.1 показано время выполнения операций с массивами.

Таблица 9.1. Время выполнения операций

Структура данных	Временная сложность							
	Средний результат				Худший результат			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Если массивы так эффективны, почему бы тогда не использовать их в любой ситуации? В то время как доступ к отдельному элементу и его изменение в массиве происходят очень быстро, изменение формы массива любым способом

(добавлением или удалением элементов) равно $O(n)$. Поскольку ваш компьютер хранит элементы в массиве в одном непрерывном блоке памяти, вставка элемента в массив означает, что нужно сдвинуть все элементы, следующие за добавленным, а это не очень эффективно. Предположим, у вас есть массив, как на рис. 9.2.

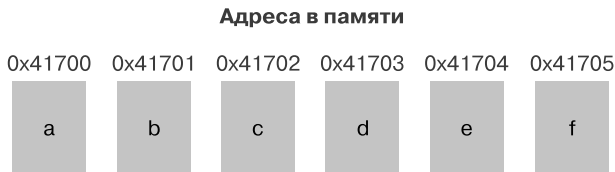


Рис. 9.2. Массив, сохраненный в памяти компьютера

Посмотрите, что произойдет, если вы добавите *z* после *a* и *b*, как показано на рис. 9.3.



Рис. 9.3. Добавление данных в массив часто приводит к изменению большого количества ячеек памяти

Чтобы сделать третий элемент в этом списке равным *z*, компьютеру нужно передвинуть четыре элемента на другие ячейки памяти.

Сдвиг ячеек не будет проблемой при работе с малыми массивами, но если вашей программе потребуется добавлять элементы в произвольные ячейки (особенно ближе к началу) в большом массиве, на переписывание памяти уйдет много времени. Проблема усугубляется при работе со статическими массивами, так как компьютер не может гарантировать, что адреса памяти, которые следуют за зарезервированным для массива блоком памяти, свободны. Это значит, что, когда вы добавляете элемент в массив на таком языке программирования, как C, вам может понадобиться зарезервировать новый блок памяти для массива, скопировать все элементы из старого блока в новый, добавить новый элемент и затем освободить старый блок. Python добавляет элементы в свои списки более эффективно за счет такого процесса, как **превышение доступности**. В этом случае для списка резервируется больший, чем ему необходимо, объем памяти и отслеживается количество элементов в списке и количество неиспользуемого пространства списка.

Работая программистом, вы часто будете обращаться к массивам в своих программах. Каждый раз, когда потребуется сохранять последовательные данные и получать к ним доступ, вам следует задумываться об использовании массива. Предположим, вы программируете игру типа *Call of Duty* и хотите создать страницу, на которой отображается топ-10 игроков игры. Вы могли бы использовать массив, чтобы легко отслеживать топ-10 игроков, их очки и их порядок, сохраняя игрока с самым высоким рейтингом под индексом 0, а с самым низким рейтингом — под индексом 9. Массив — одна из наиболее важных структур данных для математических вычислений. Если вы будете иметь дело с большим объемом числовых данных, вы познакомитесь с массивами. Программисты также используют массивы для реализации других структур данных. Например, в следующих главах вы узнаете, как реализовать структуры данных стека и очереди, используя массивы.

Среды выполнения многих языков программирования используют массивы для реализации высших по уровню структур, таких как списки, которые широко используются программистами на Python. Массивы, подобные тем, что содержатся в пакете Python Numerical Python (NumPy), полезны для математических, научных и финансовых приложений, для статистики и многого другого. Массивы NumPy поддерживают такую математическую операцию, как умножение матриц, которая используется, например, в графических приложениях для масштабирования, преобразования и поворота графических объектов. В операционных системах массивы, как правило, обрабатывают любую операцию, управляющую последовательностью данных, например управление памятью и буферизацию.

Массив — не лучший вариант для большого набора данных, в который вам нужно добавлять данные, так как сложность добавления данных в массив имеет вид $O(n)$. В такой ситуации предпочтителен связный список, о котором мы поговорим в следующей главе. Когда вы вставляете элемент в массив, он изменяет индекс других элементов, поэтому, если вы хотите, чтобы ваши данные сохраняли тот же индекс, словарь Python, вероятно, станет оптимальным выбором.

Создание массива

Если вы программируете на Python, можете использовать список в большинстве случаев, когда вам нужен массив. Однако для одномерного массива можно использовать встроенный в Python класс `array`. Вот как это работает:

```
import array

arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
print(arr[1])

>> 1
```

Сначала вы импортируете встроенный в Python модуль `array`:

```
import array
```

Затем передаете `array.array` два параметра. Первый параметр сообщает Python, какой тип данных для массива вы определяете. В этом случае `f` означает `float` — тип данных в Python для десятичных чисел с плавающей точкой, но можно создавать массивы и с другими типами данных. Второй параметр — это список с данными, которые вы хотите поместить в массив.

```
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
```

Как только вы создали массив, его можно использовать как список Python:

```
print(arr[1])
```

Однако, если вы попытаетесь добавить в массив данные с типом, отличным от изначально определенного, программа выдаст ошибку:

```
arr[1] = 'hello'
```

```
>> TypeError: "must be real number, not str"
```

Пакет NumPy в Python предлагает массив, в котором вычисления можно делать почти так же быстро, как и на низкоуровневом языке программирования, таком как C. Больше о создании массивов с помощью пакета NumPy вы можете прочесть в документации на сайте numpy.org.

Перемещение нулей

Во время прохождения технического интервью вам может понадобиться определить местоположение всех нулей в списке и переместить их в конец, оставив остальные элементы в первоначальном порядке. Предположим, у вас есть такой список:

```
[8, 0, 3, 0, 12]
```

Вы хотите его взять и вернуть новый список, в котором все нули будут стоять в конце:

```
[8, 3, 12, 0, 0]
```

Вот как решить эту проблему на языке Python:

```
def move_zeros(a_list):
    zero_index = 0
    for index, n in enumerate(a_list):
        if n != 0:
```

```

        a_list[zero_index] = n
        if zero_index != index:
            a_list[index] = 0
            zero_index += 1
    return(a_list)

a_list = [8, 0, 3, 0, 12]
move_zeros(a_list)
print(a_list)

```

Сначала вы создаете переменную под названием `zero_index` и устанавливаете ее значение равным 0:

```
zero_index = 0
```

Затем перебираете каждое число в `a_list`, используя функцию `enumerate` для отслеживания как индекса, так и текущего числа в списке:

```
for index, n in enumerate(a_list):
```

Далее следует этот код, который выполняется, только если `n` не равно 0:

```

if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
        zero_index += 1

```

Когда `n` не равно 0, вы используете индекс, сохраненный в `zero_index`, чтобы заменить все, что находится в `zero_index`, на `n`. Затем проверяете, являются ли `zero_index` и `index` все еще одним и тем же числом. Если они не равны, это означает, что ранее в списке был 0, поэтому вы заменяете любое число в текущем индексе на 0 и увеличиваете `zero_index` на 1.

Посмотрим, почему это работает. Допустим, так выглядит список, и алгоритм только что достиг первого нуля, а значит, `index` равен 1.

```
[8, 0, 3, 0, 12]
```

Поскольку число равно нулю, на этот раз в вашем цикле код не будет выполняться:

```

if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
        zero_index += 1

```

Это означает, что вы не инкрементировали `zero_index`. В следующем цикле `index` равен 2, `n` равно 3, а ваш список все тот же.

```
[8, 0, 3, 0, 12]
```

Поскольку на этот раз n не равно 0, этот код выполняется:

```
if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
    zero_index += 1
```

Когда выполняется эта часть кода:

```
a_list[zero_index] = n
```

она меняет список с такого:

```
[8, 0, 3, 0, 12]
```

на такой:

```
[8, 3, 3, 0, 12]
```

Затем эта строка кода:

```
if zero_index != index:
    a_list[index] = 0
```

меняет список с такого:

```
[8, 3, 3, 0, 12]
```

на такой:

```
[8, 3, 0, 0, 12]
```

Ваш код заменил 0 в конце списка на следующее ненулевое число в начале списка.

Теперь, когда алгоритм вновь достигнет нуля, процесс повторится.

```
[8, 3, 0, 0, 12]
```

Переменные `zero_index` и `index` больше не совпадают, и `zero_index` равен 3, что является индексом самого дальнего нуля в списке. В следующий раз n равно 12, поэтому код снова выполняется:

```
if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
    zero_index += 1
```

Этот код:

```
a_list[zero_index] = n
```

меняет ваш список с такого:

```
[8, 3, 0, 0, 12]
```

на такой:

```
[8, 3, 12, 0, 12]
```

Этот код:

```
if zero_index != index:  
    a_list[index] = 0
```

меняет ваш список с такого:

```
[8, 3, 12, 0, 12]
```

на такой:

```
[8, 3, 12, 0, 0]
```

Как видите, нули теперь в конце списка, а остальные числа предшествуют им в своем изначальном порядке.

Алгоритм содержит один основной цикл, который перебирает элементы в `a_list`, что означает временную сложность $O(n)$.

Объединение двух списков

При подготовке к техническому интервью вам следует изучить объединение двух списков, которое вы будете часто применять в своей повседневной работе. Предположим, у вас есть список фильмов:

```
movie_list = [ "Interstellar", "Inception",  
               "The Prestige", "Insomnia",  
               "Batman Begins"  
             ]
```

и список рейтингов:

```
ratings_list = [1, 10, 10, 8, 6]
```

Вы хотите объединить эти два набора данных в один список кортежей, содержащий название фильма и его рейтинг:

```
[('Interstellar', 1),  
 ('Inception', 10),  
 ('The Prestige', 10),  
 ('Insomnia', 8),  
 ('Batman Begins', 6)]
```

Можно использовать встроенную в Python функцию `zip` для объединения списков:

```
print(list(zip(movie_list, ratings_list)))

>> [('Interstellar', 1), ('Inception', 10), ('The Prestige', 10),
      ('Insomnia', 8), ('Batman Begins', 6)]
```

Функция `zip` принимает одну или более итераций и возвращает объект `zip`, содержащий один фрагмент данных из каждой итерации, которую вы затем превращаете в список. Ваши выходные данные — это список кортежей, в котором каждый список содержит название фильма и соответствующий ему рейтинг.

Поиск дубликатов в списке

Еще один популярный вопрос на техническом интервью касается проверки повторяющихся элементов в списке — это вам также придется часто делать в реальной работе. Есть такое решение — сравнивать каждый элемент в списке с каждым другим элементом. К сожалению, для подобного сравнения требуются два вложенных цикла, а временная сложность составляет $O(n^2)$. В Python лучший способ поиска дубликатов обеспечивают множества. **Множество** — структура данных, которая не может содержать повторяющиеся элементы. Если множество содержит такую строку, как `'Kanye West'`, невозможно добавить в него еще одну `'Kanye West'`.

Вот как создавать множество и добавлять в него данные:

```
a_set = set()
a_set.add("Kanye West")
a_set.add("Kendall Jenner")
a_set.add("Justin Bieber")
print(a_set)

>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```

Ваш код создает множество с тремя строками: `"Kanye West"`, `"Kendall Jenner"` и `"Justin Bieber"`. Теперь попробуйте добавить в множество второй экземпляр `"Kanye West"`:

```
a_set = set()
a_set.add('Kanye West')
a_set.add('Kanye West')
a_set.add('Kendall Jenner')
a_set.add('Justin Bieber')
print(a_set)

>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```


Как видите, в множестве по-прежнему три элемента: Python не добавил второй экземпляр "Kanye West", потому что это дубликат.

Поскольку множества не допускают дублирования, вы можете добавлять элементы из итерации один за другим, и если длина множества не меняется, вы понимаете, что элемент, который вы пытаетесь добавить, уже существует.

Ниже приведена функция, которая использует множества для проверки наличия дубликатов в списке:

```
def return_dups(an_iterable):
    dups = []
    a_set = set()

    for item in an_iterable:
        l1 = len(a_set)
        a_set.add(item)
        l2 = len(a_set)
        if l1 == l2:
            dups.append(item)
    return dups

a_list = [
    "Susan Adams",
    "Kwame Goodall",
    "Jill Hampton",
    "Susan Adams"]

dups = return_dups(a_list)
print(dups)
```

Оцениваемый вами список содержит четыре элемента, один из которых дублируется: "Susan Adams".

Функция `return_dups` принимает итерацию `an_iterable` в качестве параметра:

```
def return_dups(an_iterable):
```

Внутри функции вы создаете пустой список `dups` для хранения дубликатов:

```
dups = []
```

Затем создаете пустое множество `a_set`:

```
a_set = set()
```

Далее используете цикл `for` для перебора каждого элемента в `an_iterable`:

```
for item in an_iterable:
```

Затем получаете длину множества, добавляете элемент из `an_iterable` и проверяете, изменилась ли длина:

```
l1 = len(a_set)
a_set.add(item)
l2 = len(a_set)
```

Если длина множества не изменилась, текущий элемент является дубликатом, поэтому вы добавляете его в список `dups`:

```
if l1 == l2:
    dups.append(item)
```

Вот как выглядит вся ваша программа:

```
def duplicates(an_iterable):
    dups = []
    a_set = set()
    for item in an_iterable:
        l1 = len(a_set)
        a_set.add(item)
        l2 = len(a_set)
        if l1 == l2:
            dups.append(item)
    return dups

a_list = [
    'Susan Adams',
    'Kwame Goodall',
    'Jill Hampton',
    'Susan Adams']

dups = duplicates(a_list)
print(dups)

>> ['Susan Adams']
```

Когда вы запускаете функцию и передаете итерацию с дубликатами, она выводит список `dups`, содержащий все дубликаты.

Поиск области пересечения двух списков

Следующее популярное задание на технических собеседованиях — написание функции для поиска области пересечения двух списков, что также пригодится вам в повседневном программировании. Предположим, у вас есть список выигрышных номеров лотереи и другой список с самыми распространенными выигрышными номерами лотереи за все время.

```
this_weeks_winners = [2, 43, 48, 62, 64, 28, 3]
most_common_winners = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
```

Ваша цель — найти, сколько текущих выигрышных номеров есть в списке прошлых победителей.

Один из способов решения задачи — обратиться к списковым включениям, чтобы создать третий список, и использовать фильтр для проверки наличия каждого значения из списка `list1` в списке `list2`:

```
def return_inter(list1, list2):
    list3 = [v for v in list1 if v in list2]
    return list3

list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
print(return_inter(list1, list2))

>> [2, 62, 28]
```

Как видите, числа 2, 62 и 28 есть в обоих списках.

Эта строка кода использует цикл `for` для перебора списка `list1` и добавляет элемент в новый список, только если значение присутствует и в списке `list2`:

```
list3 = [v for v in list1 if v in list2]
```

Ключевое слово Python `in` ищет значение в итерируемом объекте. Поскольку вы имеете дело с неотсортированными списками, Python выполняет линейный поиск, когда вы используете ключевое слово `in` внутри спискового включения. Так как вы используете ключевое слово `in` внутри цикла (в первой части спискового включения), временная сложность алгоритма равна $O(n^2)$.

Другой вариант решения задачи — использование множества. В Python у множеств есть функция `intersection`, которая возвращает любой элемент, присутствующий в двух и более множествах.

Вы с легкостью можете превратить списки во множества следующим образом:

```
set1 = set(list1)
set2 = set(list2)
```

После преобразования можно применить функцию `intersection` для поиска местоположения дубликатов в двух множествах. Ниже приведен синтаксис для вызова функции `intersection` для двух множеств:

```
set1.intersection(set2)
```

Далее вы преобразуете множество с пересекающимися элементами обратно в список, используя функцию `list`:

```
list(set1.intersection(set2))
```

Соберем все вместе:

```
def return_inter(list1, list2):
    set1 = set(list1)
    set2 = set(list2)
    return list(set1.intersection(set2))

list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
new_list = return_inter(list1, list2)
print(new_list)

>> [2, 28, 62]
```

Первая строка объявляет функцию под названием `return_inter`, которая принимает два списка в качестве параметров:

```
def return_inter(list1, list2):
```

Затем вы преобразуете списки во множества:

```
set1 = set(list1)
set2 = set(list2)
```

Далее вызываете функцию `intersection` и находите дубликаты:

```
list(set1.intersection(set2))
```

И наконец, преобразуете множество обратно в список и возвращаете результат:

```
return list(set1.intersection(set2))
```

Функция `intersection` не ограничивает вас двумя множествами — ее можно вызывать для любого количества множеств. Этот код позволяет найти общие элементы:

```
(s1.intersection(s2, s3, s4))
```

Словарь терминов

Список — абстрактный тип данных, описывающий структуру данных с упорядоченными значениями.

Массив — структура данных, сохраняющая элементы с индексами в непрерывном блоке памяти.

Однородная структура данных — структура данных, которая может хранить элементы только одного типа данных, такие как целое число или число с плавающей точкой.

Неоднородный массив переменной длины — массив, размер которого может меняться после его создания, позволяющий хранить несколько типов данных.

Массив переменной длины — массив, размер которого может изменяться после создания.

Неоднородный массив — массив, содержащий различные типы данных.

Базовый адрес — адрес ячейки памяти первого элемента массива.

Одномерный массив — массив, в котором вы получаете доступ к каждому элементу массива по целочисленному индексу.

Многомерный массив — массив, в котором вы получаете доступ к каждому элементу массива с помощью кортежа индексов.

Превышение доступности — резервирование для списка большего, чем ему необходимо, объема памяти и отслеживание количества хранящихся в списке элементов и количества неиспользуемого пространства списка.

Множество — структура данных, которая не может содержать повторяющиеся элементы.

Практикум

1. У вас есть массив `an_array`, состоящий из неотрицательных целых чисел. Верните массив, в котором все нечетные элементы массива `an_array` следуют за всеми четными элементами `an_array`.

10

СВЯЗНЫЕ СПИСКИ

Обучение написанию программ тренирует мозг и помогает лучше думать, формируя образ мышления, который, как мне кажется, полезен во всех областях.

Билл Гейтс, сооснователь компании Microsoft

Связный список представляет собой еще одну реализацию списка как абстрактного типа данных. Вы можете добавлять, вставлять, удалять и искать элементы в связном списке таким же образом, как и в массиве. Однако у элементов связного списка нет индексов, потому что компьютер не хранит элементы в последовательных ячейках памяти. Вместо этого связный список имеет цепочку **узлов**, каждый из которых содержит фрагмент данных и информацию о местоположении следующего узла в цепи. Информация о местоположении следующего узла в связном списке называется **указателем**. Первый узел в связном списке называется **головным**. Указатель из последнего элемента в связном списке часто указывает на **None**, поэтому вы знаете, что это последний узел в списке (рис. 10.1).



Рис. 10.1. Связный список представляет собой цепь узлов

В отличие от массива, в связном списке компьютер может хранить узлы в не-последовательных ячейках памяти (рис. 10.2).



Рис. 10.2. Связному списку не нужно хранить узлы в последовательных ячейках памяти

Как показано на рис. 10.2, компьютер хранит знак *a* в ячейке памяти 41860. Компьютеру не нужно хранить следующий элемент списка, *b*, в следующей ячейке памяти (41861): он может хранить *b* в любой ячейке. В данном случае он хранит его в ячейке 41862.

Каждый узел содержит указатель на адрес следующего узла в списке, связывая таким образом все элементы списка (рис. 10.3). Первый элемент связанного списка, *a*, содержит указатель на адрес ячейки памяти 1862, куда помещен элемент *b*, второй элемент списка. Элемент *b* содержит указатель на местоположение ячейки следующего узла, *c*, и т. д. Эта схема создает последовательность элементов, которые отображаются все вместе.

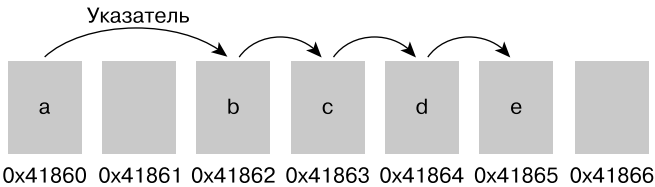


Рис. 10.3. Указатели отображают узлы связанного списка

Когда вы вставляете элемент в связный список, компьютеру не нужно сдвигать какие-либо данные — достаточно скорректировать два указателя. На рис. 10.4 показано, что произойдет, если вы добавите в список элемент *f* после *a*.

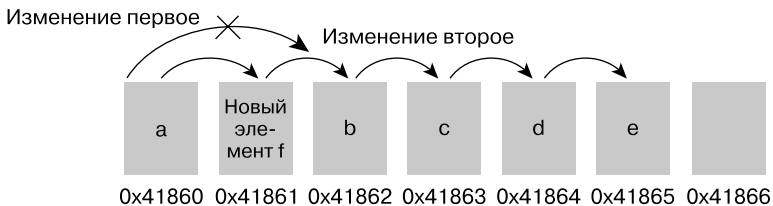


Рис. 10.4. Для вставки элемента в связный список требуется корректировка двух указателей

Компьютер изменяет указатель для *a* так, что теперь он указывает на ячейку памяти *f*, а в *f* добавляет указатель к следующему элементу — *b* (см. рис. 10.4). Больше ничего менять не нужно.

Существует много различных типов связанных списков. Связный список на рис. 10.4 — односвязный. **Односвязный список** — тип связанного списка с указателями, которые указывают только на следующий элемент. В **двусвязном списке** у каждого узла есть два указателя: один указывает на следующий узел, а другой — на предыдущий. Это позволяет перемещаться по двусвязному списку в обоих направлениях (рис. 10.5).

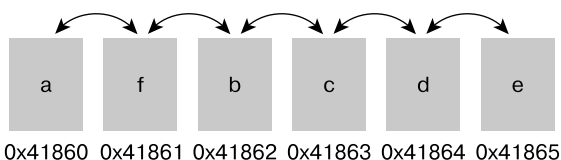


Рис. 10.5. У двусвязного списка есть указатели для обоих направлений

По односвязному списку можно перемещаться только от начала к концу, по двусвязному — и от начала к концу, и в обратном направлении; причем необязательно доходить каждый раз до конца (или начала) списка.

В **круговом связанном списке** последний узел указывает на первый узел, что позволяет переходить от последнего элемента списка обратно к началу (рис. 10.6). Этот тип структуры данных востребован в приложениях, которые многократно перебирают данные, не имеющие четкой начальной или конечной точки. Круговой связный список можно применять, например, в круговой онлайн-игре для отслеживания игроков или в среде объединения ресурсов, где пользователи по очереди используют выделенные фрагменты процессорного времени. Связный список является **закольцованным**, когда любой узел в связанном списке указывает на предыдущий узел.

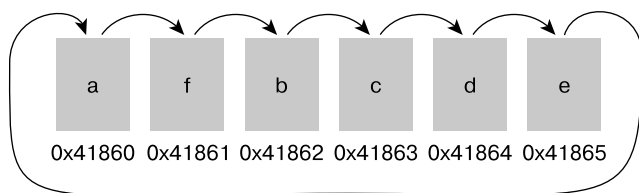


Рис. 10.6. Круговой связный список указывает от конца обратно в начало

Производительность связанных списков

В массиве доступ к элементу, зная его индекс, можно получить за постоянное время. Однако единственный способ получить доступ к элементу в связанном списке — это выполнить линейный поиск, что равно $O(n)$ (табл. 10.1). С другой стороны, добавление и удаление узла из связанного списка равно $O(1)$, в то время как вставка и удаление элементов из массива равно $O(n)$. Это различие является наиболее значимым преимуществом использования связанного списка по сравнению с использованием массива. Поиск по связанному списку, как и по массиву, равен $O(n)$.

Таблица 10.1. Время выполнения операций связанного списка

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

На понятийном уровне связный список идентичен списку или массиву Python, и вы можете использовать связный список в любой ситуации, где использовали бы список или массив. Как вы уже знаете, добавление и удаление данных в массиве равно $O(n)$, в то время как в связанном списке — $O(1)$, и это стоит учитывать при написании алгоритма. Системы управления памятью в операционных системах широко используют связанные списки. Их применяют базы данных и бизнес-системы для бухгалтерского учета, финансовых операций и операций купли-продажи. Можно использовать связанные списки и для создания других структур данных, о двух из которых вы узнаете далее в этой главе: стеки и очереди. И наконец, связанные списки необходимы для технологии блокчейна, которая породила движение web 3.0, поддерживающее криптовалюту. Сами по себе блокчейны идентичны связным спискам, и некоторые из них используют связанные списки в своей технологии.

Несмотря на то что связанные списки могут быть полезны в определенных ситуациях, у них имеются и недостатки. Первым недостатком связанного списка является то, что каждый узел должен содержать указатель на следующий узел. Указатели в связанном списке требуют ресурсов, поэтому связным спискам необходимо больше памяти, чем массивам. Если размер данных, хранящихся в одном узле, небольшой, например одно целое число, размер связанного списка может в два раза превышать размер массива, содержащего такие же данные.

Второй недостаток связанного списка состоит в том, что он не допускает произвольного доступа. В информатике **произвольный доступ** — это возможность произвольно получать доступ к данным за постоянное время. Например, невозможно сразу перейти к третьему элементу в связанном списке, как вы могли бы сделать в массиве. Вместо этого придется начинать с начала списка и следовать за каждым указателем, чтобы последовательно добраться до третьего элемента. Однако есть более продвинутые версии связанных списков, которые справляются с данной проблемой.

Создаем связанные списки

Существует множество различных способов реализации связанного списка в Python. Один из них — объявление классов, представляющих связный список и его узлы. Вот как объявляется класс для представления узла:

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

В вашем классе есть две переменные: первая, `data`, содержит фрагмент данных, вторая, `next`, содержит следующий узел списка. В Python вам не нужно иметь дело непосредственно с адресами памяти (как в языке программирования C), потому что он делает это за вас. Когда вы создаете объект, например экземпляр класса под названием `Node`, Python возвращает указатель (или ссылку) на объект. Указатель является адресом постоянного хранения фактических данных в памяти вашего компьютера. Когда в Python вы назначаете объекты переменным, вы имеете дело с указателями (ссылками), поэтому можете легко связывать объекты вместе, так как Python выполняет всю базовую работу за вас.

Далее вы объявляете класс для представления вашего связанного списка с переменной класса под названием `head`, содержащей начало вашего списка:

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Внутри класса `LinkedList` вы создаете метод `append`, который добавляет новый узел в ваш список:

```
class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        if not self.head:
            self.head = Node(data)
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = Node(data)
```

Метод `append` принимает фрагмент данных в качестве параметра, создает с ним новый узел и добавляет его в связный список.

Если в списке все еще нет головного узла, вы создаете новый узел, который становится головным узлом связного списка:

```
if not self.head:
    self.head = Node(data)
    return
```

Если же в списке уже имеется головной узел, вы находите последний узел, создаете новый узел и устанавливаете его переменную экземпляра *рядом* с ним. Чтобы выполнить это, вы создаете переменную под названием `current` и назначаете ее заголовок вашего списка:

```
current = self.head
```

Затем вы используете цикл `while`, который продолжается до тех пор, пока `current.next` не примет значение `None`. В этот момент вы поймете, что находитесь в конце связного списка:

```
while current.next:
```

Внутри цикла `while` вы непрерывно присваиваете переменной `current` значение `current.next`, пока `current` не примет значение `None` (значит, вы достигли конца списка) и цикл `while` не завершится:

```
while current.next:
    current = current.next
```

Теперь переменная `current` содержит последний узел вашего списка, поэтому вы создаете новый узел и присваиваете его `current.next`:

```
current.next = Node(data)
```

Ниже приведен пример использования метода `append` для добавления новых узлов в связный список:

```
a_list = LinkedList()
a_list.append("Tuesday")
a_list.append("Wednesday")
```

Вы также можете добавить метод `__str__` в класс `LinkedList`, чтобы без труда выводить все узлы в вашем списке:

```
def __str__(self):
    node = self.head
    while node is not None:
        print(node.data)
        node = node.next
```

```
a_list = LinkedList()
a_list.append("Tuesday")
a_list.append("Wednesday")
print(a_list)
```

```
>> Tuesday
>> Wednesday
```

В Python метод `__str__` — «волшебный метод». Если вы объявили `__str__` внутри класса, Python его вызывает, когда вы выводите объект.

Поскольку в Python нет встроенных связных списков, в нем есть встроенная структура данных под названием *deque* (дек, или двусторонняя очередь), которая использует связные списки внутри. Использование встроенной в Python структуры данных `deque` позволяет пользоваться всеми преимуществами эффективности связного списка, не кодируя его при этом самостоятельно.

```
from collections import deque
```

```
d = deque()
d.append('Harry')
d.append('Potter')
```

```
for item in d:
    print(item)
```

```
>> 'Harry'
>> 'Potter'
```

Поиск по связному списку

Вы можете немного изменить метод `append` класса `LinkedList` из предыдущего раздела, чтобы осуществить поиск элемента в связном списке:

```
def search(self, target):
    current = self.head
    while current.next:
        if current.data == target:
            return True
        else:
            current = current.next
    return False
```

Метод под названием `search` принимает один параметр под названием `target`, который является тем фрагментом данных, что вы ищете. Вы перебираете свой связный список и, если данные текущего узла совпадают с искомым значением, возвращаете `True`:

```
if current.data == target:
    return True
```

Если данные текущего узла не совпадают с искомым значением, вы присваиваете переменную `current` следующему узлу в связном списке и продолжаете итерацию:

```
else:
    current = current.next
```

Если вы достигаете конца связного списка, не найдя при этом совпадения, вы понимаете, что данных в списке нет, и возвращаете `False`:

```
return False
```

Вы можете увидеть этот алгоритм в действии, создав связный список из 20 случайных чисел со значениями от 1 до 30 и выполнив по нему поиск числа 10:

```
import random

a_list = LinkedList()

for i in range(0, 20):
    j = random.randint(1, 30)
    a_list.append(j)
    print(j, end= " ")
```

Удаление узла из связного списка

Удаление узла из связного списка — еще одно распространенное задание на технических интервью. Можно использовать линейный поиск, чтобы найти узел в связном списке и удалить его. Вы удаляете узел, изменяя указатель предыдущего узла, чтобы он больше не указывал на узел, который вы хотите удалить (рис. 10.7).

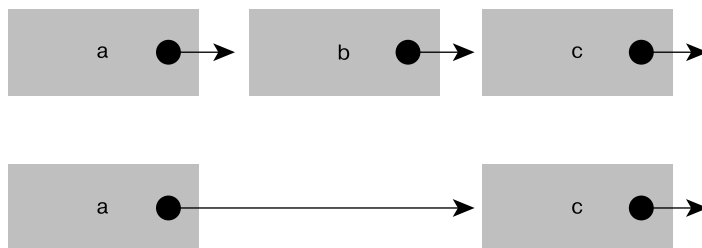


Рис. 10.7. Чтобы удалить узел, измените указатель предыдущего узла

Вот как удалить узел из связного списка:

```
def remove(self, target):
    if self.head == target:
        self.head = self.head.next
        return
    current = self.head
    previous = None
    while current:
        if current.data == target:
            previous.next = current.next
            previous = current
            current = current.next
```

Метод `remove` принимает один параметр, `target`, который является фрагментом данных, находящимся в узле, который вы хотите удалить.

Внутри метода в первую очередь вы прорабатываете, что произойдет, если узел, который вы собираетесь удалить, является головным узлом вашего списка:

```
if self.head == target:
    self.head = self.head.next
    return
```

Если это так, вы присваиваете `self.head` следующему узлу в списке и делаете возврат.

В противном случае вы перебираете связный список, отслеживая и текущий узел, и предыдущий в переменных `current` и `previous`:

```
current = self.head
previous = None
```

Далее вы используете цикл `while` для перебора связного списка. Если вы нашли искомые данные, то присваиваете `previous.next` значение `current.next`, что удаляет узел из списка:

```
while current:
    if current.data == target:
        previous.next = current.next
    previous = current
    current = current.next
```

Реверсия связного списка

Вам также стоит знать, как реверсировать (развернуть задом наперед) связный список. Чтобы это сделать, вы перебираете связный список, отслеживая текущий и предыдущий узлы. Затем вы делаете так, чтобы текущий узел указывал на предыдущий. Как только вы поменяли все указатели в связном списке, вы его реверсировали (рис. 10.8).

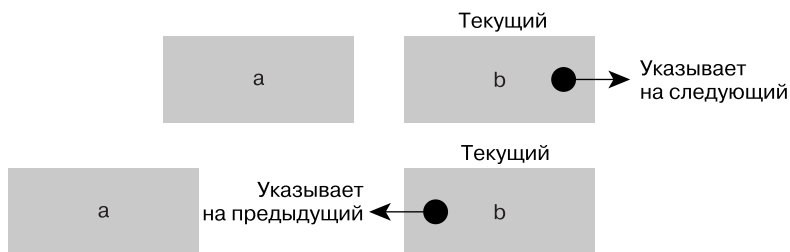


Рис. 10.8. Реверсия связного списка

Посмотрим на код для реверсии связного списка:

```
def reverse_list(self):
    current = self.head
    previous = None
    while current:
        next = current.next
        current.next = previous
        previous = current
        current = next
    self.head = previous
```

Вы используете цикл `while` для перебора связанного списка, отслеживая с помощью переменных `current` и `previous` текущий и предыдущий узлы.

Внутри цикла `while` вы сначала присваиваете значение `current.next` переменной `next`, чтобы сохранить эти данные при присвоении `current.next` значения `previous.next` в следующей строке. Как только вы присвоили `current.next` значение `previous`, вы реверсировали указатель для этого узла.

```
next = current.next
current.next = previous
```

Затем вам остается только присвоить `previous` значение `current`, а `current` — значение `next`, чтобы продолжить итерацию связанного списка и изменение остальных указателей:

```
previous = current
current = next
```

Как только вы поменяли все указатели, вы присваиваете `self.head` значение `previous`. Вы присваиваете `self.head` именно `previous`, а не `current`. Как только вы достигнете конца связанного списка, `current` примет значение `None`, а `previous` будет содержать то, что было последним узлом связанного списка, который вы превратили в первый узел, когда установили его в начало списка.

Поиск цикла связанного списка

Ранее вы узнали, что в круговом связанном списке последний элемент указывает обратно на начало списка (см. рис. 10.6). На технических собеседованиях часто предлагают определить, содержит ли связный список цикл. Другими словами, проверить, указывает ли последний элемент списка на любой элемент списка, вместо того чтобы принимать значение `None` в качестве значения для его «следующей» переменной. Один из алгоритмов для обнаружения цикла связанного списка называется *алгоритмом черепахи и зайца*. В этом алгоритме вы перебираете связный список на двух разных скоростях, отслеживая узлы в переменной `slow` и в переменной `fast`. В случае кругового связанного списка в конечном итоге переменная `fast` перекроет переменную `slow` и обе переменные будут одинаковы. Если подобное случилось, значит, ваш связный список круговой. Когда вы достигаете конца связанного списка, а этого не произошло, вы понимаете, что цикла в нем нет.

Вот как вы реализуете алгоритм черепахи и зайца:

```
def detect_cycle(self):
    slow = self.head
    fast = self.head
```



```
while True:
    try:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    except:
        return False
```

Вы начинаете с двух переменных — переменной `fast` и переменной `slow`:

```
slow = self.head
fast = self.head
```

Затем создаете бесконечный цикл:

```
while True:
```

Внутри бесконечного цикла вы присваиваете `slow` значение следующего узла списка, а значение следующего за ним узла — соответственно, переменной `fast`. Вы помещаете код внутрь блока `try`, потому что, если связный список не круговой, в конце концов переменная `fast` примет значение `None`, а это значит, что вы вызовете `fast.next.next` для `None` и произойдет ошибка. Блок `try` также предотвращает сбой вашей программы, если входные данные представляют собой пустой список или некруговой список с одним элементом.

```
try:
    slow = slow.next
    fast = fast.next.next
```

Далее вы проверяете, являются ли `slow` и `fast` одним объектом. Вы не проверяете, совпадают ли значения двух узлов связных списков, потому что одни и те же данные могут отображаться более чем в одном узле. Вместо этого вы используете ключевое слово `is`, чтобы проверить, являются ли два узла одним и тем же объектом. Если это так, вы возвращаете `True`, так как связный список круговой.

```
if slow is fast:
    return True
```

Если произошла ошибка, это означает, что вы вызвали `.next.next` для `None` и ваш связный список не круговой. Тогда вы возвращаете `False`.

Словарь терминов

Связный список — одна из реализаций списка как абстрактного типа данных.

Узел — часть структуры данных, которая содержит фрагмент данных и может подключаться к другим фрагментам данных.

Указатель — фрагмент данных в каждом узле, содержащий информацию о месте расположения следующего узла в связном списке.

Головной узел — первый узел в связном списке.

Односвязный список — тип связного списка с указателями, которые указывают только на следующий элемент.

Двусвязный список — тип связного списка, в котором каждый узел имеет два указателя: один указывает на следующий узел, а другой — на предыдущий, что позволяет перемещаться по списку в обоих направлениях.

Круговой связный список — тип связного списка, в котором последний узел указывает обратно на первый узел, что позволяет переходить от последнего элемента списка обратно к началу списка.

Цикл — когда любой узел в связном списке указывает на предыдущий узел.

Произвольный доступ — возможность произвольно получать доступ к данным за постоянное время.

Практикум

1. Создайте связный список, содержащий числа от 1 до 100. Затем выведите каждый узел списка.
2. Создайте два связных списка: один содержащий цикл, а другой — без цикла. Убедитесь, что в каждом из них есть метод `detect_cycle` для определения того, имеется ли в списке цикл. Вызовите `detect_cycle` для обоих списков.

11

Стеки

Если вы хотите творить и быть провидцем, то, вероятно, вы каким-то образом будете работать с технологиями.

Стеф Карри

Стек — это абстрактный тип данных и линейная структура данных, которая позволяет удалять только последний добавленный элемент. Можно представить себе стек в образе стопки книг: вы можете доложить или убрать только верхнюю книгу, а чтобы добраться, скажем, до третьей книги, сначала потребуется убрать все книги над ней.

Стек является примером структуры данных «последним вошел, первым вышел», или Last In — First Out (LIFO). **Последним вошел, первым вышел** — структура данных, в которой элемент, помещенный в структуру последним, является первым элементом, который выходит из нее. Поскольку вы можете получить доступ к содержимому стека лишь по одному элементу за другим, стек также является примером **структуры данных с ограниченным доступом**, которая вынуждает получать доступ к ее информации только в определенном порядке.

В стеке есть две главные операции: проталкивание (push) и выталкивание (pop, рис. 11.1). **Push** означает помещение в стек нового элемента. **Pop** — удаление из стека последнего элемента. У стеков также могут быть дополнительные операции, например считывание (peek). **Peek** — это просмотр верхнего элемента стека без его удаления.

Стеки могут быть ограниченного и неограниченного размера. **Ограниченный стек** — это стек, который ограничивает количество добавляемых в него элементов, в то время как **неограниченный стек** позволяет добавлять сколько угодно. Если вы еще до конца не понимаете разницу между абстрактным типом данных и структурой данных, стек поможет вам в этом разобраться. Стек как абстрактный тип данных характеризует идею структуры данных, которая позволяет получать доступ только к последнему добавленному в эту структуру элементу. Однако существует несколько способов создания подобной структуры. Например, вы можете создать стек, объявив класс, который для отслеживания элементов стека использует внутри либо связный список, либо массив. Когда вы пишете код для стека, используя массив или связный список, вы переходите от абстрактной идеи стека к структуре данных — фактической реализации абстрактного типа данных.



Рис. 11.1. Данные можно проталкивать в стек и выталкивать из него

Когда использовать стеки

Проталкивание элементов в стек и выталкивание из него имеют временную сложность $O(1)$. Стеки эффективны для добавления и удаления данных, но не так эффективны для операций, требующих доступа ко всему стеку (табл. 11.1). Предположим, вам требуется вывести содержимое стека. Одно из решений состоит в том, чтобы выводить каждый объект во время выталкивания его из стека. Выведение каждого выталкиваемого элемента будет иметь временную

сложность $O(n)$. Однако этот процесс также создает список объектов в обратном порядке, и ваш стек будет пуст, потому что вы вытолкнули из него все элементы. Другое решение задачи — выталкивать каждый элемент из исходного стека и добавлять его во временный стек. Затем можно выводить каждый элемент, выталкивая его из временного стека, и добавлять обратно в исходный стек. Но такое решение потребует большей затраты ресурсов, потому что нужно будет сохранять данные во временном стеке. Временная сложность этого решения равна $O(2^n)$, что в два раза дольше по времени, чем требуется для вывода элементов в массиве.

Таблица 11.1. Время выполнения операций стека

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Стек — одна из наиболее востребованных структур данных при обработке данных. Программисты применяют стеки для реализации алгоритмов поиска в ширину, чтобы искать данные в деревьях и графах (о которых вы узнаете в дальнейших главах). Среда выполнения для таких языков, как Python и Java, используют стек внутри для обработки вызовов функции. Компиляторы пользуются стеками для разбора выражений, особенно если речь идет о выражениях, имеющих вложенные пары круглых скобок, как в стандартных арифметических выражениях, или вложенные пары квадратных и фигурных скобок. Программисты также применяют стеки в алгоритмах поиска с возвратом (backtracking algorithms), которые можно встретить в машинном обучении и других областях искусственного интеллекта. Поскольку временная сложность добавления и удаления элементов из стека равна $O(1)$, стеки являются отличным выбором, когда требуется часто удалять и добавлять элементы. Например, если программе

нужен механизм отмены, во многих случаях используют один или два стека для обработки как отмены, так и повтора. К примеру, браузеры нередко используют стеки для перемещения назад или вперед по истории браузера. Поскольку временная сложность доступа к элементу в стеке равна $O(n)$, стек — не самый лучший выбор для алгоритмов, которым постоянно нужен доступ к каждому фрагменту данных в наборе данных.

Создаем стек

Как вы уже знаете, существует несколько способов использовать стек в Python. Один из них — создать класс `Stack` и управлять его данными внутри с помощью массива:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        return self.items[-1]
```

Внутри метода `__init__` класса `Stack` вы объявляете переменную экземпляра под названием `items` и присваиваете ей пустой список. Этот список является местом, в котором вы отслеживаете элементы в стеке.

```
class Stack:
    def __init__(self):
        self.items = []
```

Далее вы объявляете метод `push` стека. Вы используете встроенный в Python метод `append`, чтобы добавить новый фрагмент данных в конец `items`:

```
def push(self, data):
    self.items.append(data)
```

Следующий метод стека называется `pop`. Внутри `pop` вы используете встроенный в Python метод `pop`, чтобы вернуть последний добавленный в ваш стек элемент:

```
def pop(self):
    return self.items.pop()
```

Следующий метод класса `Stack` называется `size`, и он использует метод `len`, чтобы вернуть длину вашего стека:

```
def size(self):
    return len(self.items)
```

Метод `is_empty` проверяет, пустой ваш стек или нет:

```
def is_empty(self):
    return len(self.items) == 0
```

И наконец, последний метод под названием `peek` возвращает последний элемент стека.

```
def peek(self):
    return self.items[-1]
```

Вы также можете реализовать класс `Stack`, используя связный список внутри. Вот как создать простой стек (только с проталкиванием и выталкиванием) с применением связного списка:

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

class Stack:
    def __init__(self):
        self.head = None

    def push(self, data):
        node = Node(data)
        if self.head is None:
            self.head = node
        else:
            node.next = self.head
            self.head = node

    def pop(self):
        if self.head is None:
            raise IndexError('pop from empty stack')
        poppednode = self.head
        self.head = self.head.next
        return poppednode.data
```

Сначала вы объявляете класс `Node`, который будет представлять узлы во внутреннем связанном списке вашего стека:

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

Внутри класса `Stack` вы объявляете переменную экземпляра для начала вашего связанного списка:

```
class Stack:
    def __init__(self):
        self.head = None
```

Затем объявляете метод под названием `push`. Внутри `push` вы создаете новый узел. Если у связанного списка нет головного элемента, вы назначаете его новому узлу. В противном случае вы делаете этот узел головным.

```
def push(self, data):
    node = Node(data)
    if self.head is None:
        self.head = node
    else:
        node.next = self.head
        self.head = node
```

Далее вы объявляете метод под названием `pop`:

```
def pop(self):
    if self.head is None:
        raise IndexError('pop from empty stack')
    poppednode = self.head
    self.head = self.head.next
    return poppednode.data
```

Если в каком-то случае происходит попытка выталкивания элемента из вашего стека, когда он пуст, вы создаете исключение:

```
if self.head is None:
    raise IndexError('pop from empty stack')
```

В противном случае вы удаляете первый элемент в связанном списке и возвращаете его:

```
poppednode = self.head
self.head = self.head.next
return poppednode.data
```


Ниже приведен пример создания стека с использованием этого кода и проталкивания и выталкивания элементов:

```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

for i in range(3):
    print(stack.pop())

>> 3
>> 2
>> 1
```

Наконец, вы используете список Python как стек. Вот как он работает:

```
stack = []
print(stack)
stack.append('Kanye West')
print(stack)
stack.append('Jay-Z')
print(stack)
stack.append('Chance the Rapper')
print(stack)
stack.pop()
print(stack)

>> []
>> ['Kanye West']
>> ['Kanye West', 'Jay-Z']
>> ['Kanye West', 'Jay-Z', 'Chance the Rapper']
>> ['Kanye West', 'Jay-Z']
```

Списки Python содержат методы `append` и `pop`. Метод `append` добавляет элемент в конец списка, что идентично проталкиванию элемента в стек. Метод `pop` удаляет элемент из конца списка. Если вы не указали точно, какой элемент нужно удалить, он удаляет последний.

Когда стек выводится первый раз, он пуст, потому что в него еще ничего не добавляли.

```
>> []
```

Затем вы проталкиваете в стек три элемента: "Kanye West", "Jay-Z" и "Chance the Rapper" — с помощью следующих строк кода:

```
stack.append('Kanye West')
stack.append('Jay-Z')
stack.append('Chance the Rapper')
```

Далее вы выталкиваете последний элемент, "Chance the Rapper", из вашего стека, оставляя первые два элемента:

```
stack.pop()
```

Вот почему, когда вы выводите стек в последний раз, элемент "Chance the Rapper" отсутствует:

```
>> ['Kanye West', 'Jay-Z']
```

Стек можно также реализовать и через список. Но, конечно, когда вы используете список Python в качестве стека, вы не ограничены удалением элементов из стека в том порядке, в котором вы их туда помещали. Поэтому, если хотите это осуществить, вам нужно создать класс *Stack*, как было показано в последнем примере.

Использование стеков для реверсирования строк

Распространенное задание на собеседованиях при приеме на работу, где нужен язык Python, — реверсировать строку тремя разными способами. Если вы знакомы с Python, вы знаете, что реверсировать строку можно или так:

```
a_string[::-1]
```

или так:

```
''.join(reversed('a string'))
```

Однако когда дело доходит до реверсирования строки третьим способом, возникают проблемы. Ключом к решению задачи является то, что вы узнали ранее в этой главе: вы можете использовать стек для переворота строки, так как при выталкивании из стека символы выходят оттуда в обратном порядке (рис. 11.2).

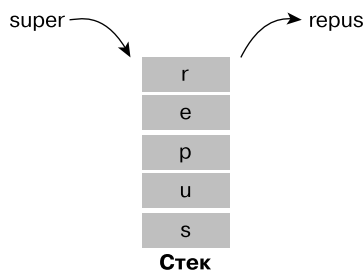


Рис. 11.2. Когда вы выталкиваете символы слова *super*, вы получаете *repus*

Вот как использовать стек для реверсирования строки:

```
def reverse_string(a_string):
    stack = []
    string = ""
    for c in a_string:
        stack.append(c)
    for c in a_string:
        string += stack.pop()
    return string

print(reverse_string("Bieber"))

>> "rebeiB"
```

Функция `reverse_string` принимает строку в качестве параметра:

```
def reverse_string(a_string):
```

Внутри функции вы используете цикл `for` для проталкивания символа в стек:

```
    for c in a_string:
        stack.append(c)
```

Затем вы используете еще один цикл, чтобы провести итерацию стека и добавить символ в переменную `a_string`, когда вытаскиваете его из стека:

```
    for c in a_string:
        string += stack.pop()
```

И наконец, вы возвращаете реверсированную строку:

```
    return string
```

Минимальный стек

Следующее популярное задание на технических интервью, которое мы рассмотрим, — разработать структуру данных, поддерживающую такие операции стека, как проталкивание и вытаскивание, и включающую в себя метод возврата наименьшего элемента. Временная сложность всех операций стека должна быть равна $O(1)$. Ключом к решению данной задачи является использование двух стеков: главного и минимального. Главный стек будет отслеживать все операции проталкивания и вытаскивания, а минимальный стек будет отслеживать наименьший элемент в стеке. Изучение способа решения данной задачи полезно не только для прохождения технического интервью — умение создавать стек, отслеживающий наименьшее число, пригодится в различных ситуациях, с которыми вы столкнетесь в своей повседневной работе программистом.

Вот как на языке Python реализовать стек для отслеживания наименьшего элемента:

```
class MinStack():
    def __init__(self):
        self.main = []
        self.min = []

    def push(self, n):
        if len(self.main) == 0:
            self.min.append(n)
        elif n <= self.min[-1]:
            self.min.append(n)
        else:
            self.min.append(self.min[-1])
        self.main.append(n)

    def pop(self):
        self.min.pop()
        return self.main.pop()

    def get_min(self):
        return self.min[-1]
```

Сначала вы объявляете класс под названием `MinStack`. Внутри метода `_init_` объявляете две переменные экземпляра: `main` и `min`. Общим переменным присваиваете пустые списки. Переменную `main` вы будете использовать для отслеживания главного стека, а переменную `min` — для отслеживания наименьшего элемента.

```
class MinStack():
    def __init__(self):
        self.main = []
        self.min = []
```

Затем вы объявляете метод `push` вашего стека. Внутри метода `push` проверяете, является ли `self.main` пустым, потому что если это так, то вне зависимости от того, какое значение у `n`, в вашем стеке оно является наименьшим. Если `self.main` пуст, вы добавляете `n` в `min`.

```
def push(self, n):
    if len(self.main) == 0:
        self.min.append(n)
```

Если `self.main` не пуст, вы проверяете, меньше или равна переменная `n` последнему элементу в `self.min`. Последний элемент в `self.min` всегда должен быть

наименьшим числом вашего стека, поэтому, если `n` меньше или равна последнему элементу в `self.min`, вы добавляете `n` в `self.min`.

```
elif n <= self.min[-1]:  
    self.min.append(n)
```

Если `n` не меньше и не равна последнему элементу в `self.min` (следовательно, она больше), вы добавляете последний элемент из `self.min` в `self.min`:

```
else:  
    self.min.append(self.min[-1])
```

Добавление последнего элемента из `self.min` в `self.min` сохраняет количество элементов в `self.main` таким же, как и в `self.min`, чтобы вы могли отслеживать наименьший элемент стека.

Рассмотрим пример. Когда вы проталкиваете первое число в стек, два внутренних стека выглядят следующим образом:

```
min_stack = MinStack()  
min_stack.push(10)  
print(min_stack.main)  
print(min_stack.min)  
>> [10]  
>> [10]
```

Вот что произойдет, когда вы протолкнете еще одно, большее, число в стек:

```
min_stack.push(15)  
print(min_stack.main)  
print(min_stack.min)  
  
>> [10, 15]  
>> [10, 10]
```

Обратите внимание, что `min_stack.main` — это обычный стек, содержащий элементы в том порядке, в котором они были в него добавлены:

```
>> [10, 15]
```

Однако `min_stack.min` не отслеживает элементы по мере их попадания в стек — он отслеживает наименьший элемент. В данном примере это дважды число 10:

```
>> [10, 10]
```

Пятнадцать не попало в `min_stack.min`, потому что 15 никогда не будет наименьшим элементом в стеке.

Когда вы вызываете метод `get_min`, он возвращает последний элемент в `self.min`, который является наименьшим числом в стеке:

```
print(min_stack.get_min())
```

```
>> 10
```

В данном примере он возвращает `10`.

После выталкивания элемента два стека выглядят следующим образом:

```
min_stack.pop()
print(min_stack.main)
print(min_stack.min)
```

```
>> [10]
```

```
>> [10]
```

Когда вы вызываете `get_min` второй раз, метод снова возвращает число `10`.

```
print(min_stack.get_min())
```

```
>> 10
```

Когда вы вызываете метод `pop` в последний раз, оба стека пусты:

```
min_stack.pop()
print(min_stack.main)
print(min_stack.min)
```

```
>> []
```

```
>> []
```

Как можете видеть, `self.min` отслеживает наименьшее число стека без необходимости сохранения числа `15`.

Помещенные в стек скобки

Однажды на собеседовании в стартап мне предложили решить такую задачу: «Вам дана строка. Используйте стек для проверки наличия в ней сбалансированных круглых скобок. То есть нужно проверить, что за каждой открывающей круглой скобкой следует закрывающая круглая скобка».

```
(str(1)) # Balanced
print(Hi!) # Not balanced
```

К сожалению, я сильно промахнулся с выполнением задания. Вашей первоначальной реакцией на задачу может быть быстрое ее решение путем анализа

строки с использованием одного счетчика для открывающих круглых скобок и другого счетчика для закрывающих круглых скобок. Если счетчики равны в конце строки, тогда скобки сбалансированы. Однако что произойдет, если счетчики столкнутся с подобной строкой?

```
a_string = ") ( ) ( "
```

Ваше решение не сработает.

Лучше всего использовать стек. Сначала вы перебираете каждый символ в строке. Найдя открывающую круглую скобку, вы помещаете ее в стек, а найдя закрывающую — проверяете, есть ли уже в стеке открывающая. Если ее там нет, значит, строка не сбалансирована, а если есть — вы выталкиваете открывающую круглую скобку из стека. При одинаковом количестве открывающих и закрывающих круглых скобок стек будет пуст к концу вашего цикла. Если стек не пуст, тогда количество открывающих и закрывающих круглых скобок не одинаково.

Посмотрим на код:

```
def check_parentheses(a_string):
    stack = []
    for c in a_string:
        if c == "(":
            stack.append(c)
        if c == ")":
            if len(stack) == 0:
                return False
            else:
                stack.pop()
    return len(stack) == 0
```

Функция `check_parentheses` принимает строку для проверки сбалансированных круглых скобок в качестве параметра:

```
def check_parentheses(a_string):
```

Внутри функции вы создаете стек с помощью списка:

```
stack = []
```

Вы используете цикл `for` для перебора символов в `a_string`:

```
for c in a_string:
```

Если символ является открывающей круглой скобкой, вы проталкиваете его в стек:

```
if c == "(":
    stack.append(c)
```

Если символ является закрывающей круглой скобкой и стек пуст, вы возвращаете `False`, потому что в вашем стеке нет соответствующей открывающей круглой скобки, а значит, строка не сбалансирована. Если в стеке есть открывающая круглая скобка, вы выталкиваете ее, чтобы сопоставить с закрывающей круглой скобкой.

```
if c == ")":
    if len(stack) == 0:
        return False
    else:
        stack.pop()
```

Как только цикл `for` завершен, вы возвращаете ответ, является ли длина вашего стека равной нулю:

```
return len(stack) == 0
```

Если функция возвращает `True`, круглые скобки сбалансированы, в противном случае они не сбалансированы.

Понимание способа решения этой задачи пригодится не только на техническом интервью. У компиляторов для языков Python и Java есть такой код для разбора и оценки выражений. Если вам когда-нибудь понадобится создать или собственный язык программирования, или код для анализа данных с открывающими и закрывающими символами, вы можете написать подобный код.

Словарь терминов

Стек — абстрактный тип данных и линейная структура данных, позволяющие удалять только последний добавленный элемент.

Последним вошел, первым вышел — структура данных, в которой элемент, помещенный в структуру последним, является первым элементом, который из нее выходит.

Структура данных с ограниченным доступом — тип структуры данных, который вынуждает вас получать доступ к ее информации только в определенном порядке.

Проталкивание — помещение нового элемента в стек.

Выталкивание — удаление последнего элемента из стека.

Считывание — просмотр верхнего элемента стека без его удаления.

Ограниченный стек — стек, который ограничивает количество добавляемых в него элементов.

Неограниченный стек — стек, который не ограничивает количество добавляемых в него элементов.

Практикум

1. Измените свою программу сбалансированных строк, чтобы проверить, сбалансированы ли в строке круглые скобки () и фигурные скобки { }.
2. Создайте максимальный стек, который позволит проталкивать, выталкивать и отслеживать самое большое число вашего стека за время $O(1)$.

12

Очереди

У всех студентов должна быть возможность изучать программирование. Компьютерная наука является основой современной креативности и самовыражения. Программисты завтрашнего дня станут революционерами в медицине.

Энн Воджицки

Очередь представляет собой абстрактный тип данных и линейную структуру данных, в которой можно добавлять элементы только в конец и удалять из начала (рис. 12.1). Очередь как абстрактный тип данных описывает структуру данных, работающую подобно кассовым очередям в продуктовом магазине: первый человек в очереди является первым на оплату, а вновь приходящие встают в конец очереди.



Рис. 12.1. В очереди вы добавляете элементы в конец и удаляете из начала

Очередь является примером структуры данных «первым вошел, первым вышел», или First In — First Out (FIFO). Как следует из названия, в структуре данных «**первым вошел, первым вышел**» первый элемент, который

вошел в структуру данных, это и первый элемент, который из нее выходит. Очереди, как и стеки, представляют собой структуры данных с ограниченным доступом.

У очереди есть две основные операции: постановка в очередь и выведение из очереди (рис. 12.2). **Постановка в очередь** означает добавление элемента в очередь, **выведение из очереди** — удаление элемента из очереди. Вы ставите элементы в очередь в конце очереди, а выводите их из очереди в ее начале.

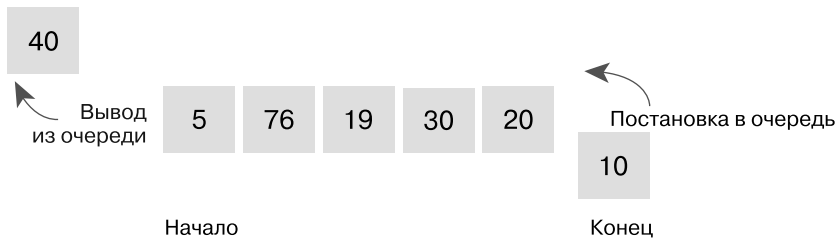


Рис. 12.2. Основные операции очереди — это постановка в очередь и вывод из очереди

Существует несколько различных способов реализации очереди в виде структуры данных. Например, как и в случае со стеками, вы можете реализовать очередь с помощью массива или связного списка. Как и стеки, очереди могут быть ограниченные и неограниченные. **Ограниченная очередь** ограничивает количество добавляемых в нее элементов, **неограниченная**, соответственно, — не ограничивает. Можно создать ограниченную очередь с помощью массива или неограниченную с помощью связного списка. Ограниченную очередь также можно реализовать с помощью связного списка, если отслеживать количество элементов, сохраненных в очереди.

Когда использовать очереди

Как и стеки, очереди эффективны для операций добавления и удаления данных (табл. 12.1). Временная сложность постановки в очередь и выведения из нее равна $O(1)$ вне зависимости от размера очереди. Как и стеки, очереди не эффективны для оценки отдельных фрагментов данных, потому что для нахождения элемента требуется просмотреть все элементы очереди: это означает, что временная сложность доступа к элементу в очереди и поиска в очереди равна $O(n)$.

Таблица 12.1. Время выполнения операций очереди

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

В реальной работе вы часто будете использовать очереди. Это идеальные структуры данных для программирования всего, что относится к структуре «первым вошел, первым вышел». Например, очередь пригодится при разработке автоматизированной телефонной системы, которая ставит звонящего в очередь, если все операторы заняты, и подключает самого первого звонившего, за которым следуют те, кто набрал номер позже. Операционные системы используют очереди для обработки запросов на запись данных на жесткий диск, для потоковой передачи аудио и видео и отправки и получения сетевых пакетов. Веб-серверы применяют очереди для обработки входящих запросов.

Всякий раз, когда вы видите сообщение «буферизация», это означает, что используемая вами система программного обеспечения, вероятно, ожидает добавления входных данных в свою очередь для обработки. Например, чтобы обеспечить бесперебойную потоковую передачу данных, потоковые системы аудио и видео часто устанавливают очередь для входных данных. Допустим, вы смотрите кино на Netflix. Программное обеспечение вашего телевизора, отвечающее за показ вам фильма, может подождать некоторое время перед запуском фильма, чтобы заполнить свою очередь видеоданными, которые отправляет Netflix. Как только программное обеспечение телевизора позволит вам начать просмотр, поступит больше данных и система добавит их в свою очередь. Использование очереди дает возможность видеоплееру извлекать пакеты данных из начала очереди за фиксированное время, что делает процесс просмотра бесперебойным, даже если входные данные поступают с несогласованной частотой. Если чересчур много пакетов данных поступает слишком быстро, система сохраняет пакеты в очереди до тех пор, пока не будет готова их принять. Когда пакеты поступают медленно, система продолжает воспроизводить пакеты, находящиеся в очереди, пока они

не закончатся. В идеале такого не должно происходить, но когда вы видите сообщение «буферизация», значит, в очереди закончились данные и потоковой передачи данных не будет, пока очередь опять не заполнится.

Попробуем представить, как выглядит подобная программа. У нее, вероятно, есть цикл, который выполняется, пока вы не закончите просмотр фильма. Внутри цикла есть алгоритм, который отвечает за добавление данных в очередь, удаление из очереди и отображение данных пользователю в виде видео. Алгоритм и подходящая структура данных (очередь) — все, что нужно для потоковой передачи фильма на вашем телевизоре или ноутбуке. И это еще раз иллюстрирует, что «программы = структуры данных + алгоритмы».

Создаем очередь

Существует несколько различных способов реализации очереди на языке Python. Один из них — объявить класс `Queue`, который использует связанные списки для внутреннего отслеживания данных. Вот каким образом создается очередь с помощью связанного списка:

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

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self._size = 0
    def enqueue(self, item):
        self._size += 1
        node = Node(item)
        if self.rear is None:
            self.front = node
            self.rear = node
        else:
            self.rear.next = node
            self.rear = node
    def dequeue(self):
        if self.front is None:
            raise IndexError('pop from empty queue')
        self._size -= 1
        temp = self.front
        self.front = self.front.next
        if self.front is None:
```

```
        self.rear = None
    return temp.data

    def size(self):
        return self._size
```

Для начала вы объявляете класс `Node` для представления узлов во внутреннем связанном списке вашей очереди:

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

Внутри очереди вы отслеживаете первые и последние элементы в переменных `self.front` и `self.rear`. Вы отслеживаете начало и конец очереди, чтобы ставить в очередь и выводить из очереди за постоянное время. Вы также отслеживаете размер очереди в переменной `self._size`.

```
def __init__(self):
    self.front = None
    self.rear = None
    self._size = 0
```

Далее вы объявляете метод под названием `enqueue`, добавляющий элемент в конец очереди:

```
def enqueue(self, item):
    self._size += 1
    node = Node(item)
    if self.rear is None:
        self.front = node
        self.rear = node
    else:
        self.rear.next = node
        self.rear = node
```

Метод принимает данные, которые вы хотите сохранить в очереди, в качестве параметра:

```
def enqueue(self, item):
```

Внутри метода `enqueue` вы сначала увеличиваете `self._size` на 1, потому что добавляете новый элемент в очередь. Затем создаете новый узел, чтобы сохранить элемент во внутреннем связанном списке очереди:

```
    self._size += 1
    node = Node(item)
```

Если `self.rear` принимает значение `None`, это означает, что ваша очередь пуста, поэтому вы присваиваете `self.front` и `self.rear` только что созданному узлу (поскольку в очереди есть только один элемент, этот элемент является одновременно первым и последним). В противном случае вы назначаете новый узел `self.rear.next`, чтобы добавить его во внутренний связный список очереди. И следом назначаете новый узел `self.rear`, таким образом он становится в конец очереди.

```
if self.rear is None:
    self.front = node
    self.rear = node
else:
    self.rear.next = node
    self.rear = node
```

Затем вы объявляете метод под названием `dequeue`, чтобы удалить элемент из начала очереди:

```
def dequeue(self):
    if self.front is None:
        raise IndexError('pop from empty queue')
    self._size -= 1
    temp = self.front
    self.front = self.front.next
    if self.front is None:
        self.rear = None
    return temp.data
```

Первая строка кода в методе выдаст исключение, если вы попытаетесь вывести элемент из пустой очереди:

```
if self.front is None:
    raise IndexError('pop from empty queue')
```

Когда вы вызываете `dequeue`, вы удаляете элемент из начала очереди и возвращаете его. Чтобы это сделать, вы сохраняете узел в начале очереди (`self.front`) в `temp`, чтобы обратиться к нему позже после удаления его из внутреннего связного списка:

```
temp = self.front
```

Потом вы удаляете элемент из начала очереди из внутреннего связного списка очереди, присваивая `self.front` значение `self.front.next`:

```
self.front = self.front.next
```

Если в очереди больше нет элементов, после того, как вы удалили элемент из начала очереди, вы устанавливаете `self.rear` значение `None`, потому что в конце очереди не осталось ни одного элемента:

```
if self.front is None:
    self.rear = None
```

Последним вы объявляете метод `size`, который возвращает количество элементов в очереди:

```
def size(self):
    return self._size
```

С помощью этих трех методов вы создали простую очередь, используя связный список, позволяющий добавлять в него данные, удалять их из него и проверять его размер. Теперь вы можете использовать очередь следующим образом:

```
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.size())
for i in range(3):
    print(queue.dequeue())
```

```
>> 3
>> 1
>> 2
>> 3
```

В предыдущем коде вы создали очередь, добавили числа 1, 2 и 3, вывели размер очереди, а затем вывели каждый элемент очереди.

Теперь посмотрим, что произойдет внутри класса `queue` при запуске этой программы. Когда вы вызываете `enqueue`, в вашей очереди нет элементов, поэтому вы добавляете узел во внутренний связный список очереди, и он становится ее началом и концом (рис. 12.3).

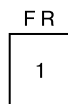


Рис. 12.3. Когда в очереди находится один элемент, он является и началом, и концом

Затем вы добавляете цифру 2 в очередь. Теперь во внутреннем связном списке очереди два узла и узел с цифрой 1 больше не является последним, так как узел с цифрой 2 стоит в конце (рис. 12.4).

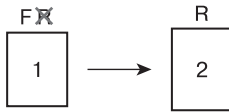


Рис. 12.4. Узел с цифрой 1 находится в начале, а узел с цифрой 2 — в конце

В конце вы добавляете цифру 3 в очередь. Теперь во внутреннем связном списке очереди три узла и узел с цифрой 2 больше не последний, так как узел 3 становится в конец (рис. 12.5).

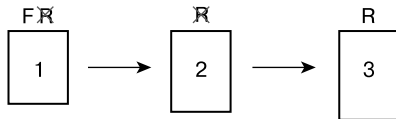


Рис. 12.5. Узел с цифрой 1 находится в начале, а узел с цифрой 3 — в конце

Когда вы вызываете `dequeue` первый раз, вы удаляете узел с цифрой 1. Теперь узел с цифрой 2 располагается в начале очереди (рис. 12.6).

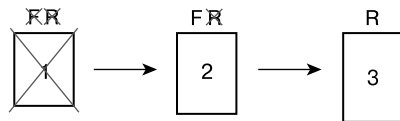


Рис. 12.6. Когда вы выводите из очереди цифру 1, начало очереди перемещается к узлу с цифрой 2

Когда вы вызываете `dequeue` второй раз, вы удаляете узел с цифрой 2. Теперь узел с цифрой 3 является и началом, и концом очереди (рис. 12.7).

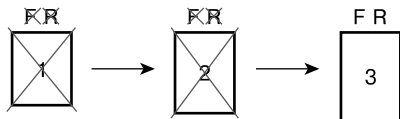


Рис. 12.7. Когда вы снова выводите элемент из очереди, остается лишь один элемент, поэтому он одновременно является и концом, и началом

Когда вы вызываете `dequeue` третий раз, вы удаляете узел с цифрой 3. Ваша очередь пуста, а `self.front` и `self.rear` указывают на `None` (рис. 12.8).

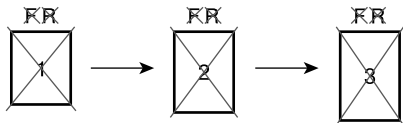


Рис. 12.8. Теперь ваша очередь пуста

Встроенный класс Queue

Вы также можете воспользоваться встроенным в Python классом для создания очереди. Вот как он работает:

```
from queue import Queue
```

```
q = Queue()
q.put('a')
q.put('b')
q.put('c')
print(q.qsize())
for i in range(3):
    print(q.get())
```

```
>> 3
>> a
>> b
>> c
```

Сначала вы импортируете `Queue` из модуля `queue`:

```
from queue import Queue
```

Далее вы создаете очередь, вызвав метод `Queue`:

```
q = Queue()
```

Вы добавляете три строки в вашу очередь с помощью встроенного метода `put`:

```
q.put('a')
q.put('b')
q.put('c')
```

Затем проверяете размер очереди с помощью встроенного метода `qsize`:

```
print(q.qsize())
```

И наконец, используете цикл `for`, чтобы вытолкнуть все элементы из очереди и вывести их:

```
for i in range(3):
    print(q.get())
```

Создаем очередь с помощью двух стеков

К распространенным вопросам на техническом интервью относится вопрос о создании очереди с помощью двух стеков. Вот как это сделать:

```
class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    def enqueue(self, item):
        while len(self.s1) != 0:
            self.s2.append(self.s1.pop())
        self.s1.append(item)
        while len(self.s2) != 0:
            self.s1.append(self.s2.pop())

    def dequeue(self):
        if len(self.s1) == 0:
            raise Exception("Cannot pop from empty queue")
        return self.s1.pop()
```

Сначала вы объявляете класс `Queue` с двумя внутренними стеками: `self.s1` и `self.s2`:

```
class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []
```

Затем объявляете метод под названием `enqueue`, чтобы добавить новый элемент в очередь:

```
def enqueue(self, item):
    while len(self.s1) != 0:
        self.s2.append(self.s1.pop())
    self.s1.append(item)
    while len(self.s2) != 0:
        self.s1.append(self.s2.pop())
```

Когда вы добавляете новый элемент в очередь, вам нужно поместить его в конец первого стека. Но так как вы можете добавлять элементы только в начало стека,

то, чтобы поместить элемент в конец стека, необходимо сначала все из стека вытолкнуть, добавить новый элемент, а затем все поместить обратно.

В данном примере вы выталкиваете все из первого стека и помещаете во второй стек, добавляете новый элемент в первый стек (как только он оказывается пуст), а затем возвращаете все обратно в первый стек. Когда вы закончите, в вашем первом стеке будут все исходные элементы, плюс новый элемент в конце.

```
while len(self.s1) != 0:
    self.s2.append(self.s1.pop())
self.s1.append(item)
while len(self.s2) != 0:
    self.s1.append(self.s2.pop())
```

Теперь, когда вы объявили `enqueue`, вы создаете метод под названием `dequeue`, чтобы удалить элемент из очереди:

```
def dequeue(self):
    if len(self.s1) == 0:
        raise Exception("Cannot pop from empty queue")
    return self.s1.pop()
```

Для начала вы проверяете, является ли стек `self.s1` пустым. Если да, это означает, что пользователь пытается применить метод `dequeue` для пустой очереди, поэтому вы создаете исключение:

```
if len(self.s1) == 0:
    raise Exception("Cannot pop from empty queue")
```

В противном случае вы выталкиваете элемент из начала первого стека и возвращаете его:

```
return self.s1.pop()
```

В этой реализации очереди временная сложность постановки в очередь равна $O(n)$, потому что приходится перебирать каждый элемент в стеке. Однако временная сложность выведения из очереди равна $O(1)$, потому что необходимо удалить только один последний элемент из внутреннего стека.

Словарь терминов

Очередь — линейная структура данных, похожая на стек.

Структура данных «первым вошел, первым вышел» — структура данных, в которой элемент, помещенный в структуру первым, является и первым элементом, который из нее выходит.

Постановка в очередь — добавление элемента в очередь.

Выведение из очереди — удаление элемента из очереди.

Ограниченная очередь — очередь, которая ограничивает количество добавляемых в нее элементов.

Неограниченная очередь — очередь, которая не ограничивает количество добавляемых в нее элементов.

Практикум

1. Реализуйте очередь с помощью двух стеков, чтобы временная сложность постановки в очередь была равна $O(1)$.

13

Хеш-таблицы

Для выучившихся самостоятельно ученых и мыслителей, таких как Чарльз Дарвин, Сриниваса Рамануджан, Леонардо да Винчи, Майкл Фарадей, я сам и другие, образование — это непрекращающееся путешествие к открытиям. Для нас образование — это вечный поиск знания и мудрости.

Абхиджит Наскар

Ассоциативный массив — абстрактный тип данных, в котором хранятся пары «ключ — значение» с уникальными ключами. **Пара «ключ — значение»** состоит из двух фрагментов данных, отображаемых вместе: ключа и значения. **Ключ** — это фрагмент данных для извлечения значения. **Значение** — фрагмент данных, для извлечения которого используется ключ. Будучи программистом на языке Python, вы уже знаете пары «ключ — значение», поскольку пользуетесь словарями Python.

Существует много различных реализаций ассоциативного массива, но в этой главе вы изучите хеш-таблицы. **Хеш-таблица** — линейная структура данных, в которой хранятся пары «ключ — значение» с уникальными ключами, а это значит, что вы не можете хранить дубликаты ключей в хеш-таблице. Разница между ассоциативным массивом и хеш-таблицей в том, что ассоциативный массив представляет собой абстрактный тип данных, в то время как хеш-таблица является структурой данных и, следовательно, воплощением ассоциативного массива. Python реализует словари при помощи хеш-таблиц.

Когда вы программируете, система, в которой запускается программа, сохраняет данные в хеш-таблице внутри структуры данных массива. При добавлении фрагмента данных в хеш-таблицу компьютер использует хеш-функцию, чтобы определить, где в массиве его хранить. **Хеш-функция** — код, принимающий ключ в качестве входных данных и выводящий целое число. Это число сопоставляет ключ хеш-таблицы с индексом массива, который ваш компьютер использует для хранения значения. Индекс, созданный хеш-функцией, называется **хеш-значением**. Вы можете сохранить любой тип данных как хеш-таблицу, но ключ должен быть чем-то, что хеш-функция сможет обратить в индекс, например целым числом или строкой. Этот процесс делает извлечение значений из хеш-таблицы невероятно эффективным, о чем вы узнаете немного позже.

Вкратце поговорим о том, как работают словари Python. Вы храните пары «ключ — значение» в словаре Python. У ключей не может быть дубликатов, но к значениям это не относится. Ниже приведен пример хранения пары «ключ — значение» в словаре Python:

```
a_dict = {}  
a_dict[1776] = 'Independence Year'
```

Теперь вы можете использовать ключ 1776 для поиска значения "Independence Year" следующим образом:

```
print(a_dict[1776])  
  
>> 'Independence Year'
```

Посмотрим для примера, как работает хеш-функция, определяя местоположение нескольких ключей в виде целых чисел. Сейчас вы увидите закулисную работу вашего компьютера при использовании словаря Python. Предположим, у вас есть хеш-таблица с семью слотами и вы хотите сохранить в них несколько целых чисел (рис. 13.1). В данном примере для наглядности мы имеем дело только с ключами, а не с ключами и их значениями.

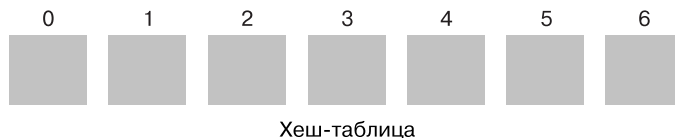


Рис. 13.1. Хеш-таблица хранит пары «ключ — значение» в массиве

Первое число, которое вам нужно сохранить, — 86. Чтобы сохранить 86 в хеш-таблице, необходима хеш-функция. Одна из простых хеш-функций состоит в том, чтобы взять каждое число и выполнить операцию остатка от деления

на количество доступных слотов (рис. 13.2). Например, чтобы получить хеш-значение для числа 86, вы вычисляете $86 \% 7$. Результат равен 2, а значит, вы помещаете 86 в индекс 2 в массиве, который используете для сохранения данных хеш-таблицы.

$$\begin{array}{r} 86 \\ \% 7 \\ = 2 \end{array}$$

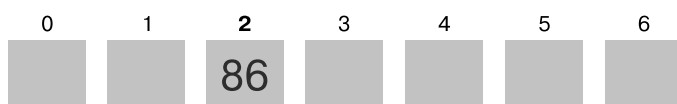


Рис. 13.2. Чтобы сохранить 86 в хеш-таблице, вы выполняете операцию остатка от деления на количество слотов и получаете 2

Следующее число, которое вам нужно сохранить в хеш-таблице, — 90, поэтому вы вычисляете $90 \% 7$, что равно 6. Итак, вы помещаете 90 в индекс 6 в вашем массиве (рис. 13.3).

$$\begin{array}{r} 86 \\ \% 7 \\ = 2 \end{array}$$

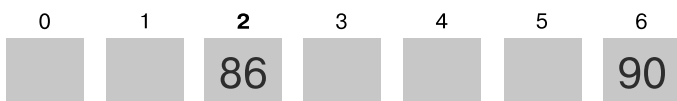


Рис. 13.3. Чтобы сохранить 90 в хеш-таблице, вы выполняете операцию остатка от деления на количество слотов и получаете 6

И наконец, вам нужно добавить 21, 29, 38, 39 и 40 в хеш-таблицу. Вот что произойдет, когда вы примените модуль 7 для этих чисел:

$$27 \% 7 = 0$$

$$29 \% 7 = 1$$

$$38 \% 7 = 3$$

$$39 \% 7 = 4$$

$$40 \% 7 = 5$$

Когда вы добавите числа в хеш-таблицу, она будет выглядеть как на рис. 13.4.

$$\begin{array}{r} 86 \\ \% 7 \\ = 2 \end{array}$$

0	1	2	3	4	5	6
21	29	86	38	39	40	90

Рис. 13.4. Хеш-таблица после добавления всех чисел

До сих пор добавление данных в вашу хеш-таблицу происходило по плану. Предположим, вы хотите еще добавить число 30. Поскольку $30 \% 7$ равно 2, вы должны добавить 30 в слот 2. Здесь, однако, возникает проблема, потому что в этом слоте уже есть число 86. Когда два числа попадают в один и тот же слот, возникает **коллизия**. Чтобы разрешить ее, вы помещаете 30 в следующий пустой слот. Такое решение работает, однако, если вам потребуется найти число 30, придется сделать следующее: использовать хеш-функцию для поиска местоположения числа в массиве, проверить слот 3, понять, что это не 30, а затем просматривать последующие слоты до тех пор, пока вы не найдете искомое число. Все это добавляет временной сложности. Существуют и другие способы решения коллизий, такие как ведение списков (обычно связанных) в каждой ячейке и помещение каждой конфликтующей пары в список, соответствующий исходной конфликтной ячейке. Когда вы создаете хеш-таблицу, ваша цель состоит в применении правильного количества слотов и хеш-функции, которая производит наименьшее количество коллизий. Однако при использовании языка Python вам не нужно беспокоиться о коллизиях, потому что словари решают эту проблему за вас.

Как я уже упоминал ранее, в предыдущем примере вы не сохраняли пары «ключ — значение». Вы можете скорректировать пример, чтобы сохранить пары «ключ — значение» с помощью двух массивов: одного — для хранения ключей и второго — для хранения значений. Так, например, если бы вы сопоставляли `self` и `taught`, хеш-функция превратила бы `self` в индекс в массиве. Затем вы бы сохранили `self` в индексе в массиве для ключей и `taught` в индексе в массиве для значений.

Когда использовать хеш-таблицы

В отличие от других структур данных, которые вы уже изучили (и которые еще изучите), временная сложность поиска данных в хеш-таблице в среднем равна $O(1)$. Вставка и удаление данных из хеш-таблицы в среднем также равны $O(1)$. Коллизии могут снизить эффективность хеш-таблиц, сделав поиск, вставку и удаление равными $O(n)$ в худшем сценарии. И все же хеш-таблицы являются одной из самых эффективных структур для хранения больших наборов данных. Причина такой эффективности хеш-таблиц заключается в следующем: для определения того, находится ли фрагмент данных в хеш-таблице, нужно всего лишь запустить данные через хеш-функцию и проверить массив по этому индексу, а это только один шаг. В табл. 13.1 показано время выполнения операций хеш-таблицы. В таблице нет данных в колонке «Доступ» для хеш-таблиц, потому что хеш-таблицы не разрешают доступ к n -му элементу, как это происходит в массиве или связном списке.

Таблица 13.1. Время выполнения операций хеш-таблицы

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Ранее вы изучили поисковые алгоритмы и то, как при сортировке данных можно выполнить двоичный поиск, который значительно быстрее линейного. Вы также узнали, что существует другой способ поиска данных, даже более эффективный в сравнении с двоичным поиском. Метод заключается в поиске данных в хеш-таблице: его временная сложность составляет $O(1)$, а это значит, что поиск данных в хеш-таблице — самый быстрый из всех возможных способов поиска данных. Возможность поиска данных за постоянное время вместо

осуществления линейного или двоичного поиска имеет огромное значение при работе с большими наборами данных.

Как программист, вы будете часто использовать хеш-таблицы. Например, если вы занимаетесь веб-разработкой, вы имеете дело с **нотацией объектов JavaScript (JSON)** — форматом обмена данными. Многие API отправляют данные в формате JSON, и такие данные можно легко преобразовать в словарь Python. **Интерфейс прикладного программирования (API)** — это программа, которая позволяет приложениям взаимодействовать друг с другом. Каждый раз, когда вы работаете в Python с базой данных «ключ — значение», вы пользуетесь словарями. Популярная система контроля версий Git использует хеш-значения из криптографической хеш-функции для хранения различных версий данных в ваших проектах, а операционные системы часто применяют хеш-таблицы для управления памятью. Сам Python пользуется словарями (хеш-таблицами) для хранения имен и значений объектных переменных.

Всякий раз, когда у вас имеется большой объем данных и нужно быстро получить доступ к отдельным элементам, стоит рассмотреть применение хеш-таблицы. Например, вы пишете программу для поиска в английском словаре или хотите создать приложение, имеющее быстрый доступ к любому телефонному номеру в телефонной книге с сотнями тысяч или миллионами записей. Хеш-таблицы подойдут в обеих ситуациях. В целом они являются подходящей структурой данных во всех случаях, когда требуется быстрый произвольный доступ к данным. Однако, если вы часто работаете с данными в последовательном порядке, лучшим выбором может быть массив или связный список.

Символы в строке

Когда вы над чем-то работаете, стоит подумать о том, чтобы применить хеш-таблицу, которая имеет исключительную эффективность. Например, на собеседовании вас просят посчитать все символы в строке. Одним из решений этой задачи является использование словаря Python. Вы можете сохранить каждый символ как ключ в вашем словаре, а то, сколько раз он встречается в строке, сохранить как значение. Это работает таким образом:

```
def count(a_string):
    a_dict = {}
    for char in a_string:
        if char in a_dict:
            a_dict[char] += 1
        else:
            a_dict[char] = 1
    print(a_dict)
```

Функция `count` принимает строку в качестве параметра (строку, в которой вы хотите посчитать количество символов):

```
def count(a_string):
```

Внутри функции вы создаете словарь:

```
a_dict = {}
```

Затем используете цикл `for` для перебора строки:

```
for char in a_string:
```

Если символ уже есть в словаре, вы инкрементируете его значение.

```
if char in a_dict:
    a_dict[char] += 1
```

Если символа еще нет в словаре, вы добавляете его как новый ключ со значением 1, поскольку символ впервые появляется в словаре.

```
else:
    a_dict[char] = 1
```

Когда в конце вы выводите словарь, он содержит как каждую букву строки, так и то, сколько раз она встречается.

```
print(a_dict)
```

Посмотрим, что произойдет при запуске функции. Когда вы вызываете функцию и передаете строку "Hello", в первой итерации программа добавляет заглавную H в качестве ключа в словарь и число 1 в качестве ее значения. Это означает, что ваш словарь будет выглядеть следующим образом:

```
{"H": 1}
```

В следующей итерации символом будет `e`. Поскольку `e` еще нет в словаре, вы добавляете этот символ в словарь как ключ со значением 1. Теперь ваш словарь выглядит следующим образом:

```
{"H": 1, "e": 1}
```

Те же самые операции вы производите для буквы `l`. Словарь имеет уже такой вид:

```
{"H": 1, "e": 1, "l": 1}
```

В следующей итерации вновь будет символ `l`. На этот раз символ присутствует в словаре, поэтому вы инкрементируете значение его ключа. Теперь словарь выглядит так:

```
{"H": 1, "e": 1, "l": 2}
```

Процесс продолжается до тех пор, пока вы не просмотрите каждый символ в строке. Когда цикл завершится и вы выведете словарь, он будет выглядеть следующим образом:

```
{ 'н': 1, 'е': 1, 'л': 2, 'о': 1 }
```

Используя данный подход, вы не только решили задачу, но и решили ее за время $O(n)$ с помощью хеш-таблицы (в данном примере n — это количество символов в строке).

Сумма двух

Еще один распространенный вопрос технического интервью касается использования хеш-таблиц для решения задачи под названием *сумма двух*. В этом типе задачи интервьюер попросит вас вернуть индексы двух чисел в неотсортированном списке, которые в сумме дают целевое значение. Вы можете предположить, что только одна пара дает в сумме целевое значение, и не использовать одно и то же число из списка дважды.

Предположим, ваше целевое значение — 5, а работаете вы с таким списком чисел:

```
[-1, 2, 3, 4, 7]
```

В этом примере числа с индексом 1 и 2 в сумме дают целевое значение 5, поэтому ответом будет индекс 1 и индекс 2 ($2 + 3 = 5$).

Один из способов решить задачу — примитивно перебрать весь список, то есть использовать так называемый метод грубой силы, проверяя пары чисел на то, дают ли они в сумме 5. Вот как кодировать данное решение методом грубой силы:

```
def two_sum_brute(the_list, target):
    index_list = []
    for i in range(0, len(the_list)):
        for j in range(i, len(the_list)):
            if the_list[i] + the_list[j] == target:
                return [the_list[i], the_list[j]]
```

Такое решение предполагает два вложенных цикла. Внешний цикл перебирает список чисел с помощью переменной i , в то время как внутренний цикл — с помощью переменной j . Вы используете обе переменные для создания таких пар, как 1 и 2, 2 и 3 и т. д., и проверяете, дают ли они в сумме целевое значение. Решение методом грубой силы простое, но неэффективное. Поскольку алгоритму необходимы два вложенных цикла для перебора каждой комбинации, его временная сложность равна $O(n^2)$.

Более эффективно решить задачу суммы двух можно с помощью словаря:

```
def two_sum(a_list, target):
    a_dict = {}
    for index, n in enumerate(a_list):
        rem = target - n
        if rem in a_dict:
            return index, a_dict[rem]
        else:
            a_dict[n] = index
```

Функция `two_sum` принимает два аргумента: список чисел и целевое значение суммы чисел:

```
def two_sum(a_list, target):
```

Внутри функции вы создаете словарь:

```
a_dict = {}
```

Затем вызываете `enumerate` для списка, что позволяет перебирать список, отслеживая каждое число и его индекс в списке:

```
for index, n in enumerate(a_list):
```

Далее вычитаете `n` из целевого числа:

```
rem = target - n
```

Результатом будет число, суммируя которое с текущим числом вы получите целевое число. Если число из переменной `rem` находится в вашем словаре, вы понимаете, что нашли ответ, и поэтому возвращаете текущий индекс и просматриваете индекс числа, сохраненного с помощью ключа `rem` внутри словаря:

```
return index, a_dict[rem]
```

Если числа из переменной `rem` нет в словаре, вы его туда добавляете, указывая `n` в качестве ключа, а его индекс — в качестве значения:

```
else:
    a_dict[n] = index
```

Посмотрим, как это работает. Допустим, вы запустили программу с таким списком и целевым числом 5:

```
[-1, 2, 3, 4, 7]
```

В первом цикле `n` равно `-1`, и в вашем словаре ничего нет, поэтому вы добавляете `-1` с индексом `0` в словарь. Во втором цикле `n` равно `2`, поэтому переменная `rem` равна `3` (`5 - 2`), и на этот раз вы добавляете `2` с индексом `1` в словарь. В третьем цикле `n` равно `3`, значит, `rem` равна `2`. Вы уже поместили число `2` в ваш словарь, а это значит, что вы нашли ответ.

В отличие от метода грубой силы, временная сложность подобного решения равна $O(n)$, так как применение хеш-таблиц означает, что больше не требуется использовать два вложенных списка `for`, а это значительно эффективнее.

Словарь терминов

Ассоциативный массив — абстрактный тип данных, хранящий пары «ключ — значение» с уникальными ключами.

Пара «ключ — значение» — два фрагмента данных, отображаемых вместе: ключ и значение.

Ключ — фрагмент данных для извлечения значения.

Значение — фрагмент данных, для извлечения которого используется ключ.

Хеш-таблица — линейная структура данных, хранящая пары «ключ — значение» с уникальными ключами.

Хеш-функция — код, принимающий ключ в качестве входных данных и выводящий уникальный фрагмент данных, который сопоставляет ключ с индексом массива, используемого компьютером для хранения значения.

Хеш-значение — уникальное значение, которое производит хеш-функция.

Коллизия — когда два числа получают одинаковое хеш-значение и попадают в один и тот же слот.

Нотация объектов JavaScript (JSON) — формат обмена данными.

Интерфейс прикладного программирования (API) — программа, которая позволяет приложениям взаимодействовать друг с другом.

Практикум

1. В предлагаемой строке удалите все повторяющиеся слова. Например, вам дана строка "I am a self-taught programmer looking for a job as a programmer.". Ваша функция должна вернуть "I am a self-taught programmer looking for a job as a.".

14

Двоичные деревья

Самые влиятельные люди — те, кто никогда не перестают учиться.

Реджойс Денхиер

До сих пор все структуры данных, которые вы изучали, были линейными. В следующих главах вы узнаете о нескольких основных нелинейных структурах и абстрактных типах данных. Первое, что мы обсудим, — **дерево**. Это нелинейный абстрактный тип данных, состоящий из узлов, объединенных в иерархическую структуру (рис. 14.1). Общие операции с деревом включают вставку, поиск и удаление.

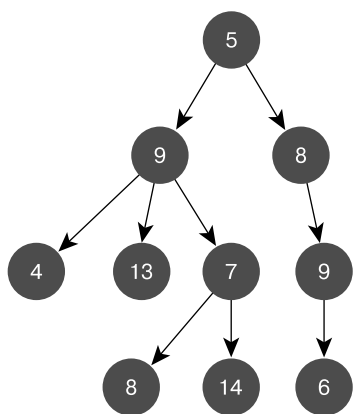


Рис. 14.1. Пример древовидной структуры данных

Существует несколько типов древовидных структур данных: общие деревья, AVL-деревья, красно-черные деревья, двоичные деревья, двоичные деревья поиска и многие другие. В этой книге вы узнаете об общих деревьях, деревьях двоичного поиска и двоичных деревьях — с акцентом на последних. Хотя описание всех типов древовидных структур выходит за рамки данной книги, я призываю вас узнать о них больше уже самостоятельно.

Общее дерево представляет собой структуру данных, которая начинается с узла наверху (рис. 14.2). Верхний узел дерева называется **корневым**. Каждый узел, соединенный с узлом, находящимся над ним в дереве, — **дочерний узел**. Узел с одним и более дочерними узлами — **родительский узел**. У **братских узлов** имеется общий родительский узел. Соединение между двумя узлами в дереве называется **ребром**.

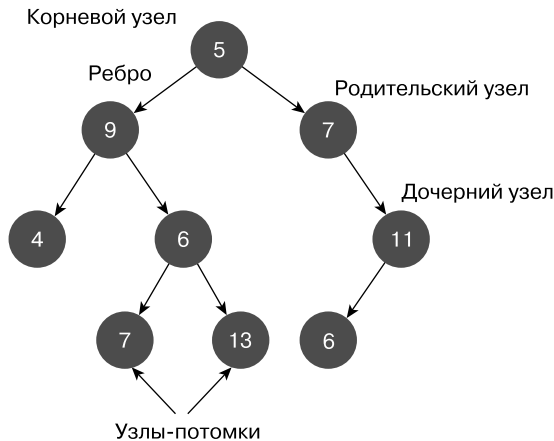


Рис. 14.2. Дерево с корневым узлом, родительскими узлами, дочерними узлами и ребрами

Вы можете перемещаться от одного узла к другому, если они имеют общее ребро (рис. 14.3).

У каждого узла дерева, за исключением корневого, есть один родительский узел. Узел без дочерних узлов — **лист**, а узел с дочерними узлами — **узел ветвления**.

Двоичное дерево — древовидная структура данных, в которой у каждого узла может быть только два дочерних узла (иногда их называют *ребенком* или *детью*). Каждый узел двоичного дерева (кроме корневого) является либо левым, либо правым дочерним элементом родительского узла (рис. 14.4).

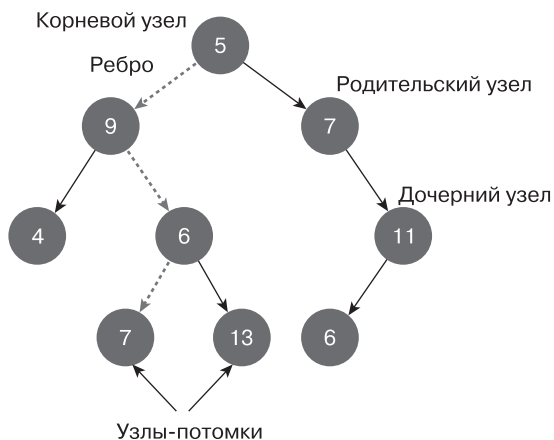


Рис. 14.3. Путь по дереву

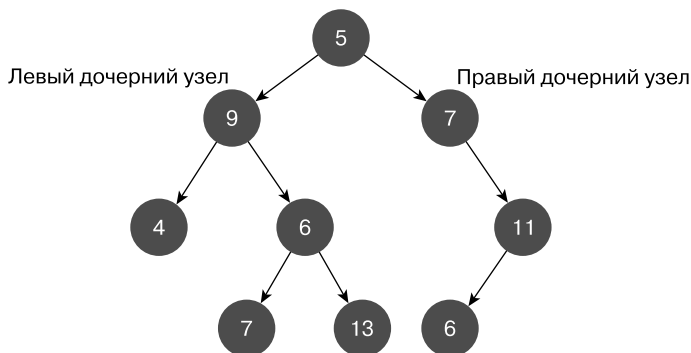


Рис. 14.4. В двоичном дереве у родительского узла может быть только два дочерних

Все остальное в двоичном дереве идентично общему дереву. Единственная разница заключается в ограниченном количестве дочерних узлов.

Двоичное дерево поиска — это древовидная структура данных, в которой у каждого узла может быть только два дочерних узла, причем дерево хранит свои узлы в отсортированном порядке, где значение каждого узла больше любого значения в его левом поддереве и меньше любого значения в его правом поддереве (рис. 14.5). Как и в случае с ключами хеш-таблицы, нельзя хранить дубликаты значений в двоичном дереве поиска. Вы можете обойти это ограничение и обрабатывать дубликаты значений, добавив поле счетчика

в объекты узлов вашего дерева для отслеживания количества появлений данного значения.

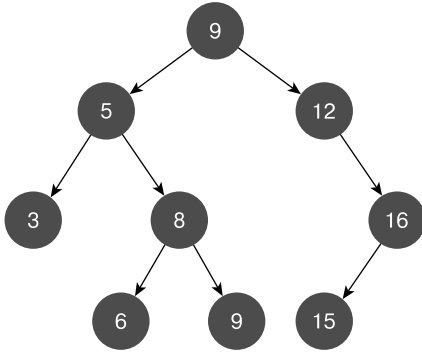


Рис. 14.5. Пример двоичного дерева поиска

В отличие от таких линейных структур данных, как массивы и связанные списки, вы не можете все время перемещаться по дереву без возврата назад. Вы можете попасть в каждый узел дерева, начав с корневого узла, но как только вы ушли от корневого узла, вы можете попасть только в потомки узла. **Потомки** узла — это его дети, и их дети, и дети их детей и т. д. На рис. 14.6 приведен пример простого дерева с корневым узлом А, листьями В, D и Е и узлом ветвления С. У узла А есть два дочерних (В и С), и у узла С есть два дочерних (D и Е). Узлы В, С, D и Е являются потомками узла А.

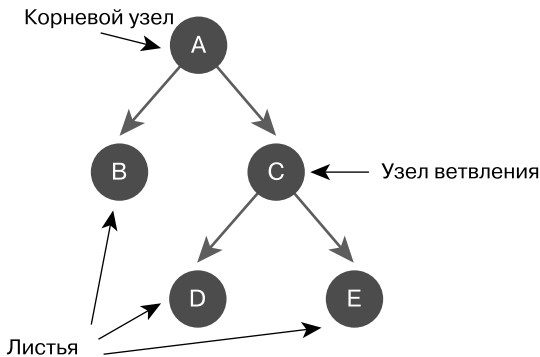


Рис. 14.6. Простое двоичное дерево с корневым узлом А и его потомками

Начав с корневого узла и перемещаясь только по правым дочерним элементам, вы пройдете узлы А, С и Е. Если вы будете перемещаться только по левым

дочерним узлам, вы пройдете через узлы А и В. Можно заметить, что ни один из этих переходов не включает узел D. Чтобы попасть в узел D, вы должны сначала перейти к правой дочке узла А, а затем к левой дочке узла С. Другими словами, чтобы дойти до узла D в двоичном дереве, вам нужно вернуться назад.

Когда использовать деревья

Вставка, удаление и поиск данных в общем дереве и в двоичном дереве равны $O(n)$. Двоичные деревья поиска более эффективны: все три операции в двоичном дереве поиска являются логарифмическими, потому что можно выполнять двоичный поиск для вставки, удаления и поиска узлов в двоичном дереве поиска (табл. 14.1).

Таблица 14.1. Время выполнения операций двоичных деревьев поиска

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Массив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Стек	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Очередь	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Связный список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Хеш-таблица	Неизвестно	$O(1)$	$O(1)$	$O(1)$	Неизвестно	$O(n)$	$O(n)$	$O(n)$
Двоичное дерево поиска	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Возможно, вам будет интересно узнать: если каждая операция линейна, почему тогда вы должны когда-либо использовать общее или двоичное дерево? Даже двоичные деревья поиска, позволяющие искать данные логарифмически, медленнее хеш-таблиц. Так зачем вообще использовать деревья? Деревья позволяют хранить иерархическую информацию, которую было бы сложно или невозможно представить в такой линейной структуре данных, как массив. Например, вы хотите представить папки в вашем компьютере программным путем. У вас может быть папка **Документы** с десятью папками в ней, в каждой из которых еще по 20 папок, а в них еще по четыре папки и т. д. Представление взаимосвязи между папками на вашем компьютере и папкой, в которой находится пользователь, с помощью массива может быть слишком сложным и запутанным, но в дереве сделать это легко (рис. 14.7).

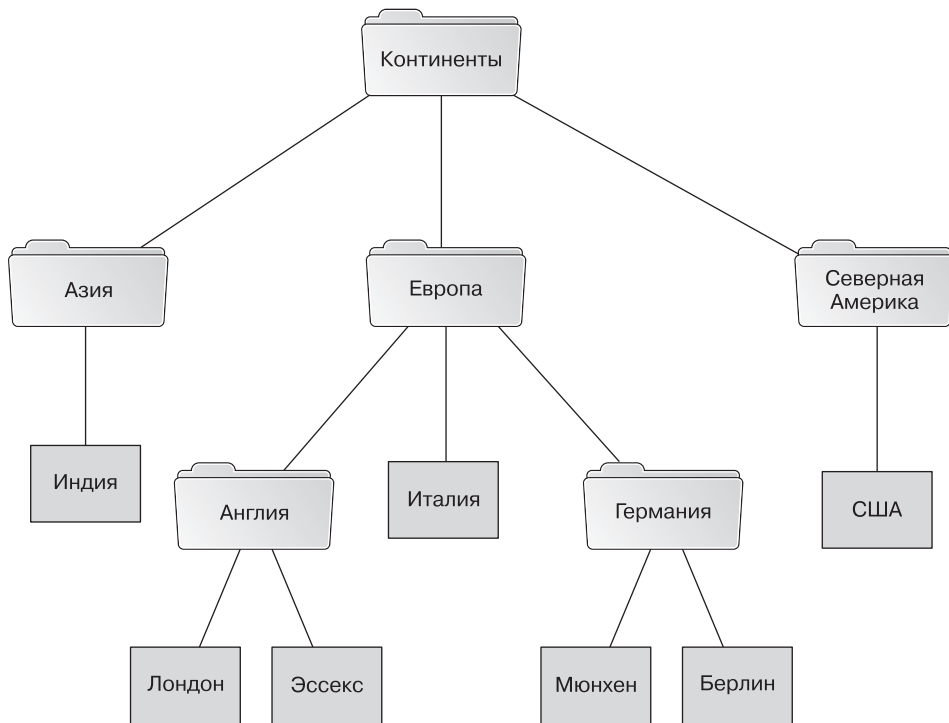


Рис. 14.7. Пример папок в дереве

Документы формата HTML и XML являются еще одним примером иерархических данных, представляемых программистами в виде дерева. **HTML** — это язык разметки, который используется для создания веб-страниц. **XML** — язык разметки для документов. Теги HTML и XML можно вкладывать друг в друга, поэтому программисты часто хранят их в виде деревьев, когда каждый узел представляет отдельный элемент в HTML и XML. При фронтенд-программировании веб-сайта язык программирования JavaScript дает доступ к объектной модели документа (DOM). **Объектная модель документа** — это не зависящий от языка интерфейс, моделирующий документы в форматах XML и HTML в виде дерева (рис. 14.8).

Вы можете использовать дерево для разбора арифметических выражений. Например, можно вычислить выражение $2 + 3 * 4$, создав дерево, такое как на рис. 14.9. Сначала вы вычисляете низ дерева ($3 * 4$), а потом перемещаетесь на уровень выше, чтобы вычислить итоговое решение ($2 + 7$). Дерево, подобное изображенному на рис. 14.9, называется деревом разбора. **Дерево разбора** — это упорядоченное дерево, которое хранит данные в соответствии

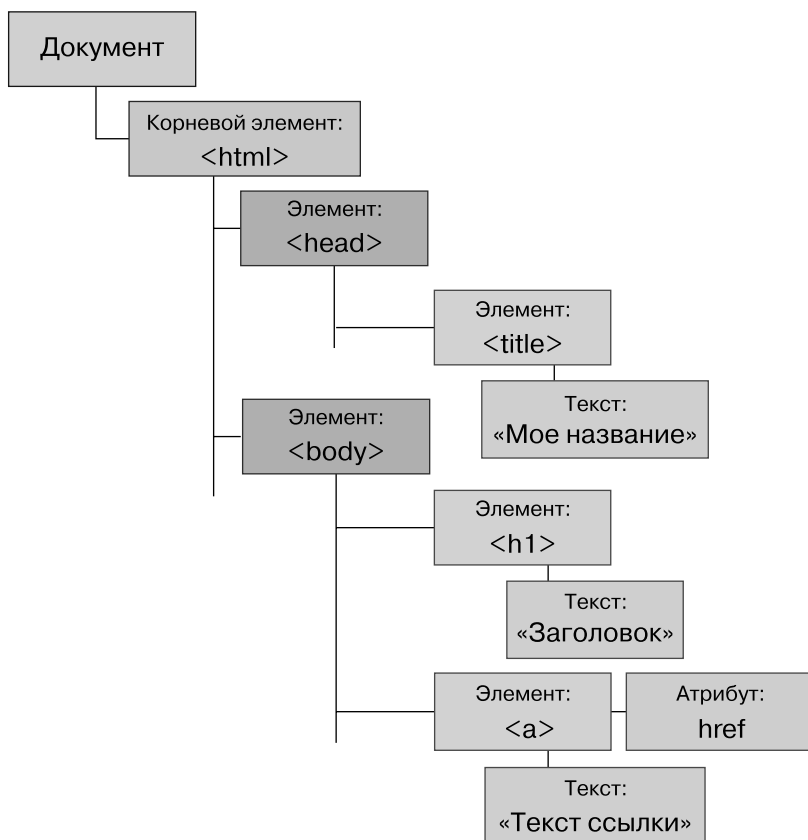


Рис. 14.8. Объектная модель документа

с определенным типом синтаксиса, например в соответствии с правилами вычисления выражения.

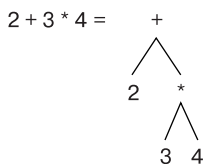


Рис. 14.9. Дерево для вычисления математического выражения

Представление иерархических данных — не единственная причина, по которой программисты используют деревья. Как вы уже знаете, поиск в отсортированном

двоичном дереве осуществляется за логарифмическое время, и он не такой быстрый, как поиск за постоянное время в хеш-таблице, однако двоичные деревья поиска имеют ряд преимуществ перед хеш-таблицами. Первое преимущество касается использования памяти. Из-за наличия коллизий хеш-таблицы могут в десять и более раз превышать объем данных, который вы в них храните. Двоичным деревьям поиска, с другой стороны, не нужна дополнительная память. Двоичное дерево поиска также позволяет совершать быстрые проходы по данным как в прямом, так и в обратном порядке сортировки, в то время как хеш-таблицы вы не можете просматривать подобным образом, потому что они не хранят данные по порядку.

Создаем двоичное дерево

Вот каким образом можно создать двоичное дерево на языке Python:

```
class BinaryTree:
    def __init__(self, value):
        self.key = value
        self.left_child = None
        self.right_child = None

    def insert_left(self, value):
        if self.left_child == None:
            self.left_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.left_child = self.left_child
            self.left_child = bin_tree

    def insert_right(self, value):
        if self.right_child == None:
            self.right_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.right_child = self.right_child
            self.right_child = bin_tree
```

Сначала вы объявляете класс под названием `BinaryTree`, который будет представлять ваше дерево. У `BinaryTree` есть три переменные экземпляра: `key`, `left_child` и `right_child`. Переменная `key` содержит данные узла (например, целого числа), переменная `left_child` отслеживает левый дочерний элемент узла, а переменная `right_child` — правый. Когда вы создаете дочерний узел для дерева, вы создаете новый экземпляр класса `BinaryTree`, у которого тоже есть ключ, левый дочерний узел и правый дочерний узел. Каждый дочерний узел является поддеревом. **Поддерево** — это узел в дереве, отличный от корневого узла и его потомков. У поддерева могут быть еще поддеревья.

Далее вы объявляете метод под названием `insert_left`, чтобы создать дочерний узел и вставить его в левую часть вашего дерева.

```
def insert_left(self, value):
    if self.left_child == None:
        self.left_child = BinaryTree(value)
    else:
        bin_tree = BinaryTree(value)
        bin_tree.left_child = self.left_child
        self.left_child = bin_tree
```

Для начала метод проверяет, имеет ли `self.left_child` значение `None`. Если это так, вы создаете новый класс `BinaryTree` и присваиваете его `self.left_child`.

```
if self.left_child == None:
    self.left_child = BinaryTree(value)
```

В противном случае вы создаете новый объект класса `BinaryTree`, присваиваете объект, находящийся на данный момент в `self.left_child`, новому объекту `self.left_child` класса `BinaryTree`, а затем присваиваете новый класс `BinaryTree` объекту `self.left_child`.

```
else:
    bin_tree = BinaryTree(value)
    bin_tree.left_child = self.left_child
    self.left_child = bin_tree
```

После объявления метода `insert_left` вы объявляете метод под названием `insert_right`: он делает то же самое, что и `insert_left`, но добавляет новый узел в правую сторону вашего дерева:

```
def insert_right(self, value):
    if self.right_child == None:
        self.right_child = BinaryTree(value)
    else:
        bin_tree = BinaryTree(value)
        bin_tree.right_child = self.right_child
        self.right_child = bin_tree
```

Теперь вы можете создать двоичное дерево и добавить в него узлы следующим образом:

```
tree = BinaryTree(1)
tree.insert_left(2)
tree.insert_right(3)
tree.insert_left(4)
tree.left_child.insert_right(6)
tree.insert_right(5)
```


Этот код создает двоичное дерево, представленное на рис. 14.10.

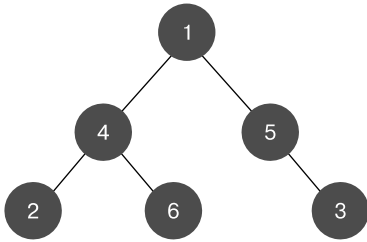


Рис. 14.10. Двоичное дерево с пятью узлами

Обход дерева в ширину

Как вы узнали ранее, вы не можете всегда перемещаться по дереву от узла к узлу без возврата назад. Однако это не значит, что в дереве нельзя искать данные. Чтобы найти фрагмент данных в дереве, нужно посетить каждый узел и посмотреть, содержит ли он искомую информацию. Существует несколько способов просмотра каждого узла в двоичном дереве. Один из них — **обход в ширину**, метод просмотра каждого узла в дереве уровень за уровнем. Например, в двоичном дереве на рис. 14.11 корень находится на уровне 0, а далее следуют уровни 1, 2 и 3.

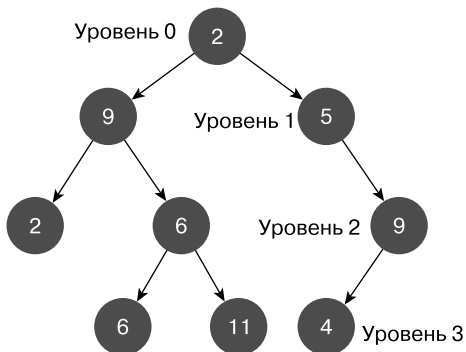


Рис. 14.11. Уровни в двоичном дереве

Когда вы используете обход в ширину для осуществления поиска, вы называете это **поиском в ширину**. Поиск в ширину начинается с корневого узла

дерева (уровень 0) и проходит уровень за уровнем с посещением всех узлов, пока не достигнет последнего уровня. Можно закодировать поиск в ширину с помощью двух списков для отслеживания текущего и следующего уровней дерева. Посещая узлы в вашем текущем уровне, вы проверяете, совпадает ли узел с искомыми данными, и добавляете его дочерние узлы в ваш список следующего уровня. Когда приходит время перемещаться к следующему узлу, вы переключаете списки. Вот как осуществить поиск числа в двоичном дереве с помощью поиска в ширину:

```
class BinaryTree:
    def __init__(self, value):
        self.key = value
        self.left_child = None
        self.right_child = None

    def insert_left(self, value):
        if self.left_child == None:
            self.left_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.left_child = self.left_child
            self.left_child = bin_tree

    def insert_right(self, value):
        if self.right_child == None:
            self.right_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.right_child = self.right_child
            self.right_child = bin_tree

    def breadth_first_search(self, n):
        current = [self]
        next = []
        while current:
            for node in current:
                if node.key == n:
                    return True
                if node.left_child:
                    next.append(node.left_child)
                if node.right_child:
                    next.append(node.right_child)
            current = next
        next = []
        return False
```

Метод `breadth_first_search` принимает один параметр `n`, который является искомыми данными:

```
def breadth_first_search(self, n):
```

Затем вы объявляете два списка. Первый список, `current`, используется для отслеживания узлов на текущем уровне, на котором вы осуществляете поиск. Второй список, `next`, предназначен для отслеживания узлов на следующем уровне. Вы также добавляете `self` в `current`, поэтому ваш алгоритм начинает поиск с корня дерева (уровень 0).

```
current = [self]
next = []
```

Цикл `while` продолжается, пока в `current` есть узлы для поиска.

```
while current:
```

Затем вы используете цикл `for` для итерации через каждый узел в `current`.

```
for node in current:
```

Если значение узла совпадает с `n` (искомым значением), вы возвращаете `True`.

```
if node.key == n:
    return True
```

В противном случае, если правый и левый дочерние узлы вашего узла не принимают значение `None`, вы добавляете их в список `next`, чтобы осуществить в них поиск на следующем уровне.

```
if node.left_child:
    next.append(node.left_child)
if node.right_child:
    next.append(node.right_child)
```

В конце цикла `while` вы меняете местами `current` и `next`. Список узлов для следующего поиска становится списком узлов для текущего поиска, и вы устанавливаете `next` в пустой список.

```
current = next
next = []
```

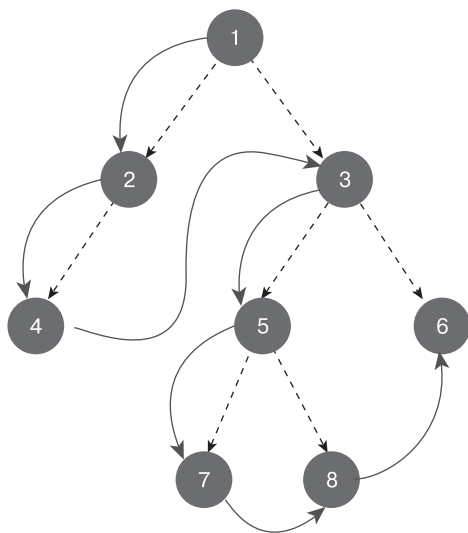
Если цикл `while` заканчивается, вы возвращаете `False`, потому что поиск в ширину не нашел `n` в дереве.

```
return False
```

Другие обходы дерева

Обход в ширину — не единственный способ обойти двоичное дерево: можно выполнить обход и в глубину. При **обходе в глубину** вы посещаете все узлы двоичного дерева путем погружения как можно глубже в одном направлении, прежде чем перейти к следующему потомку. Обход в глубину предлагает три варианта посещения каждого узла: прямой, обратный и симметричный. Реализации всех трех подходов похожи, но их использование различно.

Допустим, у вас есть двоичное дерево, представленное на рис. 14.12. При прямом обходе вы начинаете с корня и двигаетесь влево, а затем вправо.



Прямой обход: 1, 2, 4, 3, 5, 7, 8, 6

Рис. 14.12. Прямой обход дерева в глубину

Ниже представлен код для прямого обхода дерева:

```
def preorder(tree):  
    if tree:  
        print(tree.key)  
        preorder(tree.left_child)  
        preorder(tree.right_child)
```

Функция рекурсивно вызывает саму себя, пока не достигнет терминальной ветви, которая представляет собой такую строку кода:

```
if tree:
```

Эта строка кода выводит значения каждого узла дерева:

```
print(tree.key)
```

Следующие строки кода вызывают `preorder` для каждого левого и правого дочернего узла дерева:

```
preorder(tree.left_child)
preorder(tree.right_child)
```

Подобный обход должен быть вам знаком, так как он похож на ваши действия при написании сортировки слиянием в главе 4. Когда вы писали код для сортировки слиянием, вы помещали рекурсивный вызов в левую половину списка, за которым следовал рекурсивный вызов в правой половине списка. Алгоритм вызывал сам себя с левой половиной, пока не оставался список лишь из одного элемента. Когда это происходило, алгоритм вызывал рекурсивный код, чтобы разбивать вашу правую половину списка. По достижении терминальной ветви вы перемещались на уровень выше в рекурсивном стеке и объединяли два списка с помощью кода, написанного под двумя рекурсивными вызовами. Алгоритм сортировки слиянием похож на прямой обход, но называется *обратным обходом*. Разница между обратным обходом и прямым состоит в том, что вы выводите значение каждого узла (или делаете что-то другое) после ваших рекурсивных вызовов в обратном обходе.

```
def postorder(tree):
    if tree:
        postorder(tree.left_child)
        postorder(tree.right_child)
        print(tree.key)
```

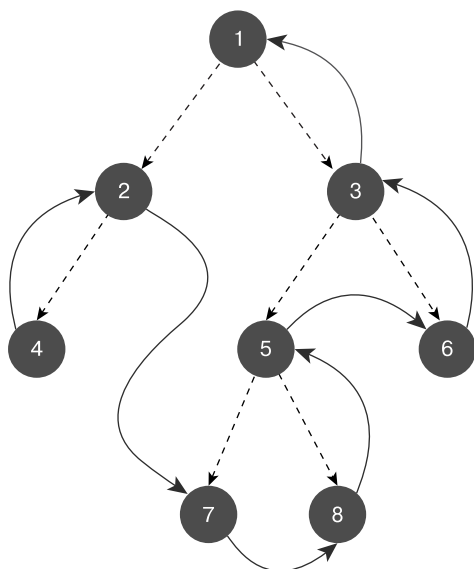
При обратном обходе вы передвигаетесь по дереву, начиная слева, затем двигаетесь вправо и заканчиваете в корневом узле, как показано на рис. 14.13.

Если вы представляете обратный обход как алгоритм сортировки слиянием, вы выводите узел каждый раз, когда осуществляете слияние.

И наконец, есть симметричный обход.

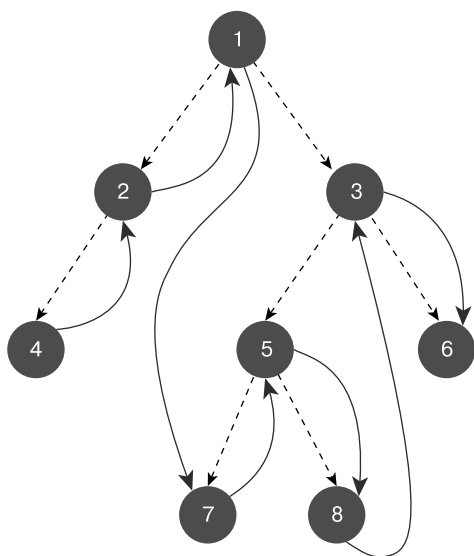
```
def inorder(tree):
    if tree:
        inorder (tree.left_child)
        print(tree.key)
        inorder (tree.right_child)
```

Симметричный обход похож на прямой и обратный обходы, но значение узла выводится (или делается что-то другое) в промежутках между двумя рекурсивными вызовами. Когда вы используете симметричный обход, вы перемещаетесь по дереву слева до корня и направо, как показано на рис. 14.14.



Обратный обход: 4, 2, 7, 8, 5, 6, 3, 1

Рис. 14.13. Обратный обход дерева в глубину



Симметричный обход: 4, 2, 1, 7, 5, 8, 3, 6

Рис. 14.14. Симметричный обход дерева в глубину

Инвертируем двоичное дерево

Макс Хауэлл — основатель Homebrew, популярного менеджера пакетов. Как известно, он проходил собеседование в Google на должность инженера-программиста, но получил отказ. После собеседования он написал в Twitter: «Google: 90 % наших программистов используют написанное тобой программное обеспечение (Homebrew), но ты не можешь инвертировать двоичное дерево на доске, так что иди ####». **Инвертирование двоичного дерева** означает перестановку всех узлов в нем: каждый правый узел становится левым, а каждый левый узел становится правым. В этом разделе вы узнаете, как инвертировать двоичное дерево, поэтому ваше техническое интервью не закончится так, как у Макса Хауэлла.

Для инвертирования двоичного дерева нужно посетить каждый узел и отследить его дочерние элементы, чтобы поменять их местами. Один из способов сделать это — использовать поиск в ширину, который позволяет с легкостью отслеживать все левые и правые узлы и переключать их.

Код для инвертирования двоичного дерева выглядит следующим образом:

```
class BinaryTree:
    def __init__(self, value):
        self.key = value
        self.left_child = None
        self.right_child = None

    def insert_left(self, value):
        if self.left_child == None:
            self.left_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.left_child = self.left_child
            self.left_child = bin_tree

    def insert_right(self, value):
        if self.right_child == None:
            self.right_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.right_child = self.right_child
            self.right_child = bin_tree

    def invert(self):
        current = [self]
        next = []
        while current:
            for node in current:
```

```
if node.left_child:
    next.append(node.left_child)
if node.right_child:
    next.append(node.right_child)
tmp = node.left_child
node.left_child = node.right_child
node.right_child = tmp
current = next
next = []
```

Код такой же, как при обходе в ширину для поиска числа, но вместо проверки того, равно ли значение узла `n`, вы меняете местами правый и левый дочерние узлы при каждой итерации.

Чтобы выполнить это, сначала вы сохраняете `node.left_child` во временной переменной под названием `tmp`. Затем устанавливаете для `node.left_child` значение `node.right_child`, а для `node.right_child` — значение `tmp`, что меняет местами два дочерних узла.

```
tmp = node.left_child
node.left_child = node.right_child
node.right_child = tmp
```

Когда алгоритм завершится, вы успешно инвертируете двоичное дерево.

Другой, более изысканный способ инвертирования двоичного дерева — обход в глубину, который вы можете опробовать в практикуме.

Словарь терминов

Дерево — нелинейный абстрактный тип данных, состоящий из узлов, объединенных в иерархическую структуру.

Корневой узел — верхний узел дерева.

Дочерний узел — узел, соединенный с родительским узлом, находящимся над ним в дереве.

Родительский узел — узел с одним и более дочерними узлами.

Братские узлы — узлы, имеющие общий родительский узел.

Ребро — соединение между двумя узлами в дереве.

Лист — узел без дочерних узлов.

Узел ветвления — узел, имеющий дочерние узлы.

Двоичное дерево — древовидная структура данных, в которой у каждого узла может быть только два дочерних узла.

Двоичное дерево поиска — древовидная структура данных, в которой у каждого узла может быть только два дочерних узла, причем дерево хранит узлы в отсортированном порядке, где значение каждого узла больше значения его левого дочернего узла и меньше значения его правого дочернего узла.

Потомки — дочерние узлы и их дети, и дети их детей и т. д.

HTML — язык разметки, который используется для создания веб-страниц.

XML — язык разметки для документов.

Объектная модель документа — не зависящий от языка интерфейс, моделирующий XML- и HTML-документы в виде дерева.

Дерево разбора — упорядоченное дерево, которое хранит данные в соответствии с определенным типом синтаксиса, например в соответствии с правилами вычисления выражения.

Поддерево — узел в дереве, отличный от корневого узла и его потомков.

Обход в ширину — метод просмотра всех узлов в дереве путем посещения каждого его узла уровень за уровнем.

Поиск в ширину — поиск с использованием обхода в ширину.

Обход в глубину — посещение всех узлов двоичного дерева путем погружения как можно глубже в одном направлении, перед тем как переходить к следующему братскому узлу.

Инвертирование двоичного дерева — перестановка в нем всех узлов.

Практикум

1. Добавьте в код вашего двоичного дерева метод под названием `has_leaf_nodes`. Метод должен вернуть `True`, если у дерева есть узлы без листьев, и `False`, если их нет.
2. Инвертируйте двоичное дерево с помощью обхода в глубину.

15

Двоичные кучи

Я считаю, что расцвет Google, расцвет Facebook, расцвет Apple являются доказательством того, что компьютерная наука может рассматриваться как нечто, решающее проблемы, с которыми люди сталкиваются каждый день.

Эрик Шмидт

Очередь с приоритетом — это абстрактный тип данных, описывающий структуру данных, в которой каждый фрагмент обладает определенным приоритетом. В отличие от очереди, освобождающей элементы по принципу «первым вошел, первым вышел», очередь с приоритетом обслуживает элементы в соответствии с приоритетом: сначала она удаляет данные с наивысшим приоритетом, затем данные с последующими по рангу приоритетами (или, наоборот, начинает с наименьших значений). Куча является одной из многочисленных реализаций очереди с приоритетом. **Куча** — древовидная структура данных, в которой каждый узел отслеживает два фрагмента данных: сами данные и их приоритет. Вы можете называть значение узла кучи **ключом**. Хотя ключ узла и его приоритет могут быть не связаны, но если данные узла представляют собой числовое значение, например целое число или символ, его также можно использовать в качестве приоритета. В этой главе я использую в кучах ключи, которые, как вы увидите, обозначают и приоритет.

Программисты создают кучи с помощью деревьев. Существует множество видов куч (в зависимости от типа используемого дерева), но в данной главе вы изучите

двоичные кучи. **Двоичная куча** — та, что создана с помощью двоичного дерева (рис. 15.1).

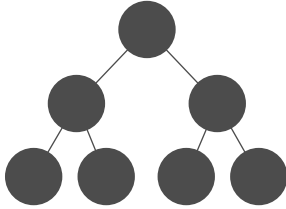


Рис. 15.1. Вы создаете двоичную кучу с помощью двоичного дерева

Есть два вида двоичных куч: максимальные и минимальные. Приоритет родительского узла **максимальной кучи** всегда выше или равен приоритету любого дочернего узла, а узел с наивысшим приоритетом является корнем дерева. Например, на рис. 15.2 показана максимальная куча с целыми числами 1, 2, 3, 4, 6, 8 и 10.

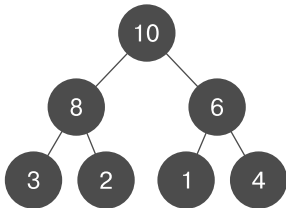


Рис. 15.2. В максимальной куче узел с наивысшим приоритетом является корнем

Приоритет родительского узла **минимальной кучи** всегда ниже или равен приоритету любого дочернего узла, а узел с наименьшим приоритетом является корнем дерева. На рис. 15.3 показана минимальная куча с теми же целыми числами, что и в максимальной куче на рис. 15.2.

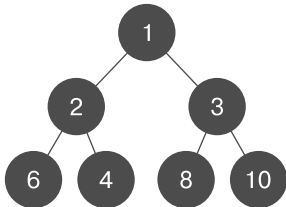


Рис. 15.3. В минимальной куче узел с наименьшим приоритетом является корнем

В двоичной куче упорядочение (по возрастанию или убыванию) применяется только к родительскому узлу и его потомкам. Между братскими узлами нет сортировки. Как вы можете видеть на рис. 15.3, братские узлы не упорядочены (6 и 4).

Создание кучи из такой структуры данных, как массив, программисты называют **построением кучи (heapifying)**. Например, у вас имеется следующий массив неотсортированных ключей:

["R", "C", "T", "H", "E", "D", "L"]

Чтобы построить кучу из этих данных, вы сначала добавляете каждый фрагмент данных как узел в двоичное дерево. Вы начинаете с верха дерева и заполняете дочерние узлы слева направо на каждом последующем уровне, как показано на рис. 15.4.

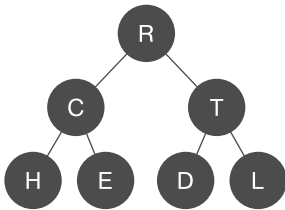


Рис. 15.4. Результат построения кучи из массива

Затем вы балансируете вашу кучу. **Балансировка кучи** означает переупорядочивание ключей, которые расположены не по порядку. В данном примере вы начинаете с последнего родительского узла (Т) и сравниваете его с листьями. Если у какого-то листа значение меньше родительского узла, вы меняете его местами с родительским узлом (рис. 15.5).

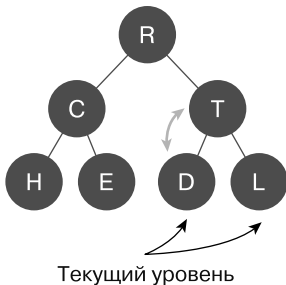


Рис. 15.5. Перестановка значений для балансировки кучи

В данном примере D является наименьшим из трех узлов (T, D и L), поэтому вы меняете его местами с родительским узлом T (рис. 15.6).

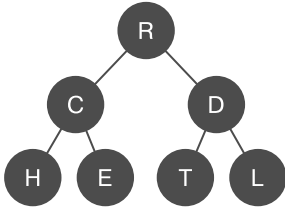


Рис. 15.6. Перестановка D и T — первый шаг балансировки кучи

Далее вы переходите к предпоследнему родительскому узлу и его листьям (C, H и E). C стоит перед H и E, поэтому вы не делаете никаких перестановок в левой части дерева (рис. 15.7).

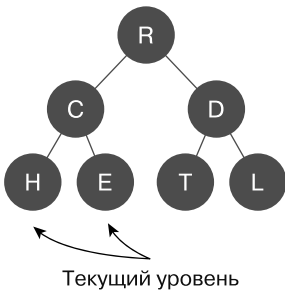


Рис. 15.7. Левая часть кучи уже сбалансирована

Теперь вы переходите на уровень выше и опять сравниваете (рис. 15.8).

У C наименьшее значение из узлов R, C и D, поэтому вы меняете местами C и R. Теперь C — это корень дерева (рис. 15.9).

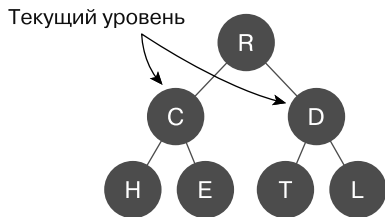


Рис. 15.8. Балансировка дерева на следующем уровне

Корневой узел

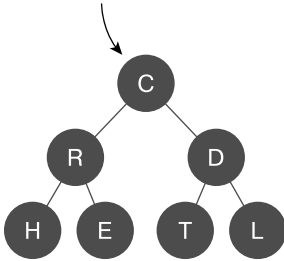


Рис. 15.9. Теперь C — корневой узел двоичной кучи

Далее вы «протискиваете вниз» узел R, сравнивая его значения с его листьями. Если значение узла R больше одного из его листьев, вы меняете их местами и сравниваете значение узла R с его новыми листьями. Процесс продолжается до тех пор, пока значение R больше одного из его листьев или пока вы не достигнете нижнего уровня кучи. E стоит перед R, поэтому вы меняете их местами (рис. 15.10).

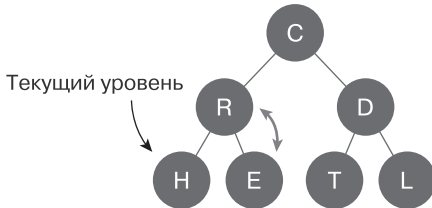


Рис. 15.10. Узел R «протискивается вниз», пока его значение больше, чем у хотя бы одного из его листьев

Теперь ваша куча сбалансирована (рис. 15.11).

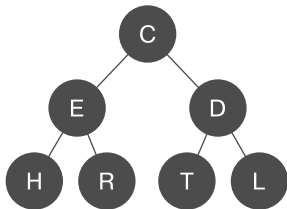


Рис. 15.11. Сбалансированная куча

Программисты часто хранят кучи в массивах. Вы можете сохранить кучу в списке Python, распределив ключи в списке в зависимости от их расположения в дереве (рис. 15.12).

C	E	D	H	R	T	L
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Рис. 15.12. Массив с ключами в индексах, основанных на их местоположении в дереве

Внутри вашего списка у корневого узла кучи индекс 0. У его левого дочернего узла индекс 1, а у его правого дочернего узла индекс 2. Вы можете использовать математическое уравнение, чтобы найти местоположение дочернего элемента узла. Для любого узла k индекс его левого дочернего узла будет равен $2k + 1$, а индекс его правого дочернего узла будет равен $2k + 2$. Для примера, уравнение для нахождения правого дочернего элемента узла C имеет вид $2 * 0 + 2$, что равно 2. Это означает, что правый дочерний узел для узла с индексом 0 находится в индексе 2 (рис. 15.13).

C	E	D	H	R	T	L
[0]		[2]				

Рис. 15.13. Правый дочерний узел корня находится в индексе 2

Когда использовать кучи

Вы можете найти максимальное и минимальное значение в максимальной или минимальной куче за постоянное время, но удаление минимального узла из минимальной кучи или максимального узла из максимальной кучи будет логарифмическим, потому что после удаления элемента вам нужно будет сбалансировать оставшиеся узлы. Вставка данных в кучу тоже логарифмическая, а поиск данных в куче равен $O(n)$.

Кучи полезны каждый раз, когда требуется выполнять задачи в соответствии с приоритетом. Например, операционная система может использовать кучу для отслеживания различных задач и распределения ресурсов на основании приоритета каждой задачи. Кучу также можно использовать для реализации алгоритма Дейкстры для нахождения кратчайшего пути между двумя узлами в графе. Алгоритм Дейкстры, который вы изучите в главе 16, помогает решать задачи маршрутизации, такие как определение способа добраться из одного города в другой и маршрутизации в компьютерных сетях. Программисты также используют кучи в алгоритме сортировки, который называется *пирамидальной сортировкой*.

Создаем кучу

В Python есть библиотечная функция под названием `heapq`, которая облегчает процесс создания минимальной кучи. Вот программа, использующая функцию `heapify` из `heapq` для построения кучи от списка из семи элементов:

```
from heapq import heapify

a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heapify(a_list)
print(a_list)

>> ['C', 'E', 'D', 'H', 'R', 'T', 'L']
```

Сначала вы импортируете функцию `heapify` из библиотеки `heapq`, а затем передаете список в функцию `heapify`. Как видите, когда вы выводите список после построения из него кучи, он представляет собой минимальную кучу, которую вы храните в списке Python.

Вы можете использовать функцию `heappop` библиотеки `heapq` для извлечения ключа из кучи и повторной ее балансировки. Удалить корневой ключ из кучи и сбалансировать оставшиеся ключи можно следующим образом:

```
from heapq import heapify, heappop

a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heap = heapify(a_list)
print(a_list)
heappop(a_list)
print("After popping")
print(a_list)

>> ['C', 'E', 'D', 'H', 'R', 'T', 'L']
>> After popping
>> ['D', 'E', 'L', 'H', 'R', 'T']
```

Для начала вы импортируете как `heapify`, так и `heappop` из модуля `heapq`:

```
from heapq import heapify, heappop
```

Затем создаете кучу путем передачи вашего списка функции `heapify` и его вывода:

```
a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heap = heapify(a_list)
print(a_list)
```


Далее вы используете функцию `heappop` для выталкивания минимального элемента из кучи и вывода результата:

```
heappop(a_list)
print("After popping")
print(a_list)
```

Для выталкивания элементов из кучи можно использовать цикл `while`. Вот каким образом создается куча и из нее выталкиваются все ключи:

```
from heapq import heapify, heappop

a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
while len(a_list) > 0:
    print(heappop(a_list))
```

Сначала вы создаете кучу:

```
a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
```

Затем используете цикл `while` для выталкивания всех ее ключей:

```
while len(a_list) > 0:
    print(heappop(a_list))
```

В библиотеке `heapq` также есть функция под названием `heappush`, которая вставляет ключ в кучу и повторно балансирует ее. Вот как использовать функцию `heappush` для проталкивания элемента в кучу:

```
from heapq import heapify, heappush

a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
heappush(a_list, "Z")
print(a_list)

>> ['D', 'E', 'L', 'H', 'R', 'T', 'Z']
```

Python предоставляет встроенную функцию только для минимальных куч, но можно легко создать максимальную кучу для числовых значений, умножив каждое значение на -1 . Максимальная куча со строками в качестве ключей более сложна в реализации. Вместо использования библиотеки `heapq` вам надо создать ее с помощью класса или написать код для кучи самостоятельно.

И наконец, вы можете использовать `heapq` для обработки пар «приоритет — значение», сохраняя кортежи, первый элемент которых является приоритетом, а второй — значением (оно может быть любым). Подобный пример вы увидите в следующей главе, когда будете кодировать алгоритм Дейкстры.

Соединяем канаты с минимальными затратами

Вы можете использовать кучи для решения многих задач, с которыми столкнетесь в повседневном программировании, а также для тех, что вам предложат на техническом интервью. Например, на техническом интервью вам могут дать список канатов различной длины и попросить их связать, по два за раз, в порядке, который приведет к наименьшим затратам. Затраты на соединение двух канатов равны их сумме, а общие затраты равны сумме соединения всех канатов. Так, допустим, у вас есть список канатов:

```
[5, 4, 2, 8]
```

Сначала вы можете соединить 8 и 2, затем 4 и 10, а потом 5 и 14. Когда вы просуммируете все затраты, вы получите 43.

```
[5, 4, 2, 8] # 8 + 2 = 10  
[5, 4, 10] # 10 + 4 = 14  
[5, 14] # 5 + 14 = 19  
# 10 + 14 + 19 = 43
```

Однако, если вы соедините узлы в другом порядке, вы получите другой ответ. Чтобы получить верный ответ, вам нужно соединять каждый раз два наименьших каната следующим образом:

```
[5, 4, 2, 8] # 4 + 2 = 6  
[5, 8, 6] # 6 + 5 = 11  
[8, 11] # 8 + 11 = 19  
# 6 + 11 + 19 = 36
```

Общие затраты при подобном подходе к задаче равны 36, что и является верным ответом.

Вы можете использовать минимальную кучу для написания функции, которая решает данную задачу. Сделать это можно следующим образом:

```
from heapq import heappush, heappop, heapify  
  
def find_min_cost(ropes):  
    heapify(ropes)  
    cost = 0  
    while len(ropes) > 1:  
        sum = heappop(ropes) + heappop(ropes)  
        heappush(ropes, sum)  
        cost += sum  
    return cost
```

Для начала вы объявляете функцию `find_min_cost`, принимающую список канатов в качестве параметра:

```
def find_min_cost(ropes):
```

Затем используете функцию `heapify` для превращения `ropes` в минимальную кучу и объявления переменной под названием `cost` для отслеживания общих затрат на добавление всех канатов:

```
    heapify(ropes)
    cost = 0
```

Далее вы создаете цикл `while`, который выполняется до тех пор, пока длина `ropes` больше 1:

```
    while len(ropes) > 1:
```

Внутри цикла вы используете функцию `heappop`, чтобы извлечь два наименьших значения из вашей кучи и суммировать их. Затем используете функцию `heappush` для проталкивания этой суммы обратно в кучу. И наконец, добавляете сумму к затратам:

```
        sum = heappop(ropes) + heappop(ropes)
        heappush(ropes, sum)
        cost += sum
```

Когда цикл завершается, вы возвращаете `cost`, в которой содержится значение наименьших затрат на соединение всех канатов.

```
    return cost
```

Словарь терминов

Очередь с приоритетом — абстрактный тип данных, описывающий структуру данных, в которой у каждого фрагмента данных есть приоритет.

Куча — древовидная структура данных, в которой каждый узел отслеживает два фрагмента данных: сами данные и их приоритет.

Ключ — значение узла в куче.

Двоичная куча — куча, использующая в качестве базовой структуры данных двоичное дерево.

Максимальная куча — куча, в которой приоритет родительского узла всегда выше или равен приоритету любого дочернего узла, а узел с наивысшим приоритетом является корнем дерева.

Минимальная куча — куча, в которой приоритет родительского узла всегда ниже или равен приоритету любого дочернего узла, а узел с наименьшим приоритетом является корнем дерева.

Построение кучи (heapifying) — создание кучи из структуры данных, такой как массив.

Балансировка кучи — переупорядочивание ключей, расположенных не по порядку.

Практикум

1. Напишите функцию, которая принимает двоичное дерево в качестве параметра и возвращает `True`, если это минимальная куча, и `False`, если нет.

16

Графы

Изучение программирования обязательно пригодится вам в будущем, какими бы ни были ваши профессиональные планы. Умение кодировать также сделает вас невероятно крутым!

Макс Левчин

Граф — это абстрактный тип данных, в котором фрагмент данных соединяется с одним или несколькими другими фрагментами данных. Каждый фрагмент данных в графе называется **вершиной** или *узлом*. У вершины есть имя, называемое *ключом*. У вершины могут быть дополнительные данные, такие как **полезная нагрузка**. Соединение между вершинами в графе — **ребро**. Ребра графа могут содержать **вес**: затраты на перемещение между вершинами. Например, если вы создали граф, представляющий данные на карте, каждая вершина может быть городом, а вес между двумя вершинами может быть расстоянием между городами (рис. 16.1).

Существует несколько типов графов, в том числе ориентированные графы, неориентированные графы и полные графы. **Ориентированный граф** — тот, в котором у каждого ребра есть связанное с ним направление и между двумя вершинами можно перемещаться только в этом направлении. Как правило, соединение между двумя вершинами однонаправленное, но вы также можете создать ребро с двунаправленным соединением. Ориентированный граф является великолепным выбором для создания графа, представляющего социальную сеть с подписчиками, например Twitter. С помощью ориентированного графа вы

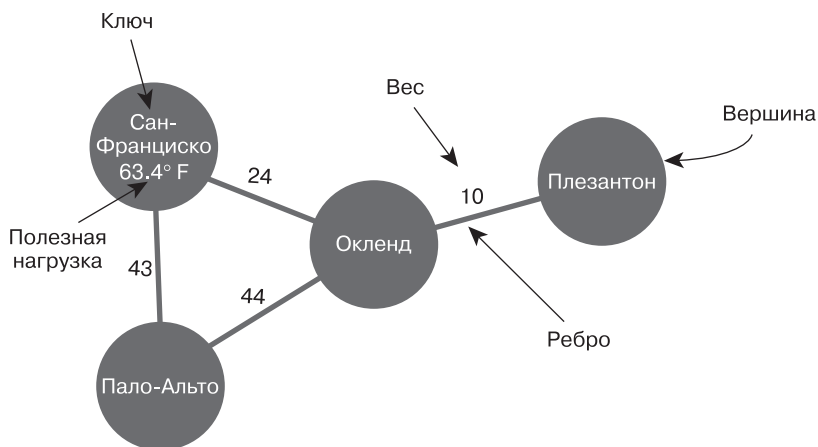


Рис. 16.1. Граф содержит вершины, ребра, полезные нагрузки и вес

можете отобразить, что подписаны в Twitter на Леброна Джеймса, но он не подписался на вас в ответ. Когда вы строите ориентированный граф, вы обычно изображаете ребра стрелками, показывающими направления, в которых можно перемещаться (рис. 16.2).

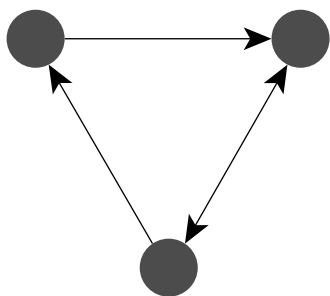


Рис. 16.2. Ориентированный граф перемещается в определенном направлении

Неориентированный граф — граф с двунаправленными ребрами, позволяющий перемещаться вперед и назад в любом направлении между двумя связанными вершинами. Вы можете думать об этом как о двусторонней связи, такой как отношения между друзьями в социальной сети наподобие Facebook. Например, если Логан является другом Хэдли на Facebook, тогда Хэдли является другом Логана. Когда вы строите неориентированный граф, вы обычно изображаете ребра без стрелок (рис. 16.3).

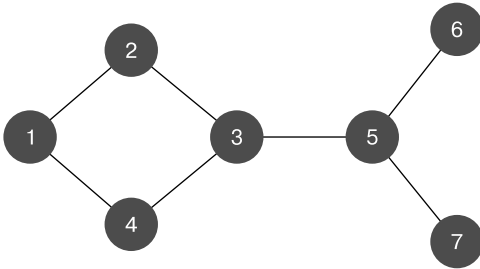


Рис. 16.3. Неориентированный граф позволяет перемещаться в любом направлении

Полный граф — это граф, в котором каждая вершина соединена со всеми другими вершинами (рис. 16.4).

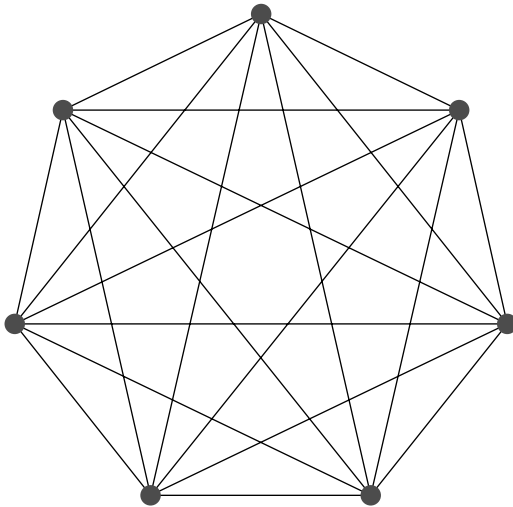


Рис. 16.4. В полном графе соединены все вершины

В **неполном графе** некоторые, но не все, вершины не соединены (рис. 16.5).

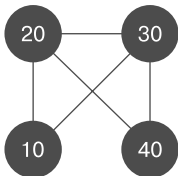


Рис. 16.5. В неполном графе есть несколько несвязанных вершин

Путь графа — это последовательность вершин, соединенных ребрами (рис. 16.6). Например, у вас есть граф, представляющий город. Путь между Лос-Анджелесом и Сан-Франциско будет отображен как серия ребер (то есть дорог), которые вы можете использовать для путешествия из Лос-Анджелеса в Сан-Франциско.

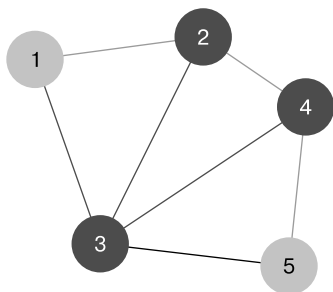


Рис. 16.6. Путь графа соответствует определенной последовательности

Цикл — путь в графе, начинающийся и заканчивающийся в одной и той же вершине. **Ациклический граф** — граф, который не содержит цикла (рис. 16.7).

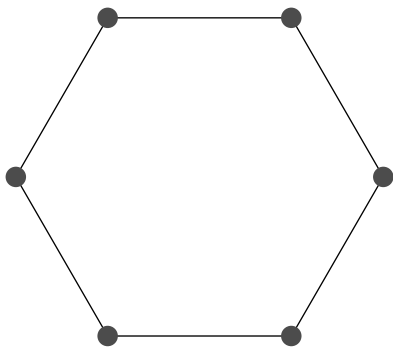


Рис. 16.7. Пример графа, содержащего цикл

Многие из этих понятий вам знакомы, так как вы уже изучили деревья. Дерево — это ограниченная форма графа. У деревьев есть направления (детско-родительские отношения) и нет циклов, что делает их направленными ациклическими графами с ограничением: у дочернего узла может быть лишь один родительский узел.

Существует несколько способов создания графов программным путем. Например, вы можете использовать список ребер, матрицу смежности или список смежности. **Список ребер** — это структура данных, в которой вы представляете

каждое ребро в графе с помощью двух соединенных вершин. Например, на рис. 16.8 изображен граф с четырьмя вершинами.

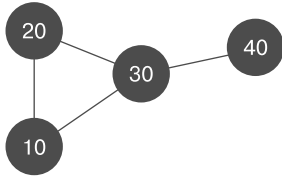


Рис. 16.8. Граф с четырьмя вершинами

С помощью списка ребер его можно представить следующим образом:

```
[  
[10, 20]  
[10, 30]  
[20, 10]  
[20, 30]  
[30, 10]  
[30, 20]  
[30, 40]  
[40, 30]  
]
```

Список ребер является списком списков, где в каждом списке содержатся две соединенные вершины из графа.

Вы также можете представить граф с помощью матрицы смежности. **Матрица смежности** — двумерный массив строк и столбцов, содержащий вершины графа. В матрице смежности пересечение строки и столбца используется для представления ребра. Традиционно соединенные вершины представляются как 1, а несоединенные — как 0. Когда две вершины соединены, они **смежные**. На рис. 16.9 показано представление этого же графа с помощью матрицы смежности.

	10	20	30	40
10	0	1	1	0
20	1	0	1	0
30	1	1	0	1
40	0	0	1	0

Рис. 16.9. Матрица смежности графа с рис. 16.8

Одной из проблем матрицы смежности является разреженность, или пустые ячейки: в данном примере имеется восемь пустых ячеек. Матрицы смежности — не самый эффективный способ хранения данных, потому что в итоге в них может оказаться много пустых ячеек, а это приводит к неэффективному использованию памяти компьютера.

И наконец, вы можете отобразить граф с помощью списка смежности. **Список смежности** — это набор неупорядоченных списков, каждый из которых представляет соединения для одной вершины. Вот как выглядит тот же граф с рис. 16.8, отображенный с помощью списка смежных вершин:

```
{  
10: [20, 30],  
20: [10, 30],  
30: [10, 20, 40],  
40: [30]  
}
```

Как видите, узел 10 соединен с узлами 20 и 30, а узел 20 соединен с узлами 10 и 30 и т. д.

Когда использовать графы

Как вы уже знаете, существует множество различных реализаций графов. Добавление вершины и ребра в граф, как правило, равно $O(1)$. Время выполнения поиска, удаления и других алгоритмов в графе зависит от реализации графа и от того, какая структура данных используется для этой реализации: массивы, связанные списки, хеш-таблицы или что-то еще. В целом производительность основных операций в графе зависит либо от количества вершин в нем, либо от количества ребер, либо от некоторого сочетания этих двух чисел, поскольку, по сути, графы имеют дело с двумя вещами: элементами в графе (вершинами) и соединениями (ребрами) между этими элементами.

Графы полезны во многих ситуациях. Например, разработчики программного обеспечения, работающие на такие социальные сети, как Twitter, используют вершины в графах для представления людей и ребра для представления связей между ними. Программисты также применяют графы для построения сетей, часто отображая устройства в них в виде вершин с ребрами, представляющими беспроводные или проводные соединения между этими устройствами. Графы можно использовать для создания карт, где вершины представляют города и другие пункты назначения, а ребра — дороги, автобусные или авиамаршруты между этими пунктами. Программисты обращаются к графам для нахождения самого быстрого пути между двумя пунктами назначения. Графы полезны и для

компьютерной графики: их вершины и ребра могут быть точками, линиями и плоскостями 2D- и 3D-фигур (рис. 16.10).

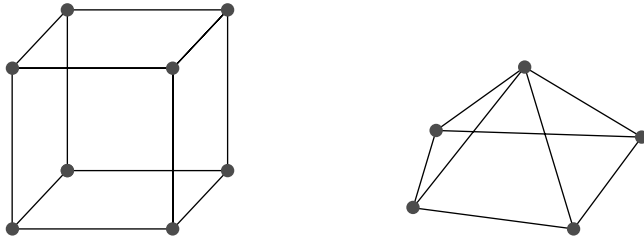


Рис. 16.10. Графы могут представлять 3D-фигуры

Алгоритмы поисковых машин часто используют графы для определения рейтинга поиска на основе связанности поиска и результатов. Операционные системы и системы языков программирования также используют графы при управлении памятью.

Создаем граф

Вот каким образом можно создать список смежности на языке Python:

```
class Vertex:
    def __init__(self, key):
        self.key = key
        self.connections = {}

    def add_adj(self, vertex, weight=0):
        self.connections[vertex] = weight

    def get_connections(self):
        return self.connections.keys()

    def get_weight(self, vertex):
        return self.connections[vertex]

class Graph:
    def __init__(self):
        self.vertex_dict = {}

    def add_vertex(self, key):
        new_vertex = Vertex(key)
        self.vertex_dict[key] = new_vertex
```

```

def get_vertex(self, key):
    if key in self.vertex_dict:
        return self.vertex_dict[key]
    return None

def add_edge(self, f, t, weight=0):
    if f not in self.vertex_dict:
        self.add_vertex(f)
    if t not in self.vertex_dict:
        self.add_vertex(t)
    self.vertex_dict[f].add_adj(self.vertex_dict[t], weight)

```

Сначала вы объявляете класс вершины, как делали ранее с узлами, когда создавали связанные списки:

```

class Vertex:
    def __init__(self, key):
        self.key = key
        self.connections = {}

    def add_adj(self, vertex, weight=0):
        self.connections[vertex] = weight

```

У класса `Vertex` есть две переменные экземпляра: `self.key` и `self.connections`. Первая переменная, `key`, — ключ вершины, а вторая переменная, `connections`, — словарь, в котором вы будете хранить вершины, с которыми смежна каждая из вершин.

```

def __init__(self, key):
    self.key = key
    self.connections = {}

```

У класса `Vertex` есть метод под названием `add_adj`, принимающий вершину в качестве параметра и делающий ее смежной с вершиной, для которой вы вызвали метод, добавляя соединение в `self.connections`. Метод также принимает вес в качестве параметра, если вы хотите добавить вес в связь.

```

def add_adj(self, vertex, weight=0):
    self.connections[vertex] = weight

```

Далее вы объявляете класс под названием `Graph`. `Graph` имеет переменную экземпляра `self.vertex_dict`, хранящую вершины каждого графа.

```

def __init__(self):
    self.vertex_dict = {}

```

Метод класса `add_vertex` добавляет новую вершину в граф, сначала создавая новую вершину, а затем соотнося переданный пользователем в качестве параметра ключ с новой вершиной внутри `self.vertex_dict`.

```
def add_vertex(self, key):
    new_vertex = Vertex(key)
    self.vertex_dict[key] = new_vertex
```

Метод `get_vertex` принимает ключ в качестве параметра и проверяет `self.vertex_dict`, чтобы узнать, является ли вершина частью вашего графа, производя попытку вернуть ее.

```
def get_vertex(self, key):
    if key in self.vertex_dict:
        return self.vertex_dict[key]
    return None
```

Наконец, у класса графа есть метод под названием `add_edge`, который добавляет ребро между двумя вершинами в вашем графе.

```
def add_edge(self, f, t, weight=0):
    if f not in self.vertex_dict:
        self.add_vertex(f)
    if t not in self.vertex_dict:
        self.add_vertex(t)
    self.vertex_dict[f].add_adj(self.vertex_dict[t], weight)
```

Теперь вы можете создать граф и добавить вершины в него следующим образом:

```
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B", 1)
graph.add_edge("B", "C", 10)
vertex_a = graph.get_vertex("A")
vertex_b = graph.get_vertex("B")
```

В данном примере для простоты у двух разных вершин не может быть одного и того же ключа.

Алгоритм Дейкстры

Когда вы работаете с графами, вам часто требуется найти кратчайший путь между двумя вершинами. Один из наиболее известных алгоритмов в программировании — **алгоритм Дейкстры**: его можно использовать для нахождения кратчайшего пути от одной вершины в графе до любой другой вершины. Алгоритм изобрел известный в сфере информатики ученый Эдсгер Виле Дейкстра — всего за 20 минут, в своей голове, без ручки и бумаги.

Вот каким образом работает алгоритм Дейкстры. Сначала вы выбираете начальную вершину. Начальная вершина — это вершина, из которой вы найдете кратчайший путь к любой другой вершине вашего графа. Допустим, у вас есть такой граф, как на рис. 16.11.

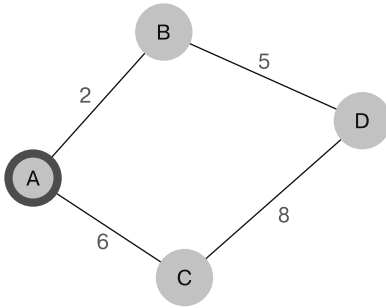


Рис. 16.11. Граф с четырьмя вершинами

Если A — ваша начальная вершина, то в конце программы у вас будет словарь, содержащий все вершины в графе и кратчайший путь от начальной вершины (A) до каждой вершины.

```
{  
"A": 0,  
"B": 2,  
"C": 6,  
"D": 7,  
}
```

Как вы можете видеть на рис. 16.11, кратчайший путь из A в D равен 7, потому что путь от вершины A до B, а потом до D равен 7 ($2 + 5$), а путь от вершины A до C и потом до D равен 14 ($6 + 8$).

В начале алгоритма вы устанавливаете путь от начальной вершины к самой себе равным 0, а все остальные пути устанавливаете равными бесконечности (рис. 16.12).

Расстояние:

A: 0
B: ∞
C: ∞
D: ∞

Рис. 16.12. Установите путь до начальной вершины равным нулю, а остальные пути — равными бесконечности

Ключом к алгоритму является очередь с приоритетом. Она используется для поиска в ширину по графу с отслеживанием вершин и расстояний от них до начальной вершины. Посмотрим, как это работает с графом из предыдущего примера.

Алгоритм начинается с начальной вершины A в вашей очереди с приоритетом. Вы отслеживаете все кратчайшие пути в словаре. В словаре вы устанавливаете расстояние от вершины A к ней самой равным 0 , а все остальные расстояния — равными бесконечности (рис. 16.13). На данном этапе вы пока не посетили ни одной вершины. Посетить вершину — значит вытолкнуть ее из очереди с приоритетом и, не найдя к ней кратчайший путь от начальной вершины, просмотреть все смежные вершины в поисках кратчайшего пути от начальной вершины. Если вы нашли кратчайший путь, вы помещаете смежную вершину в очередь с приоритетом.

```
Unvisited Vertices {A, B, C, D}
Priority Queue [(0, A)]
Distances {
A: 0,
B: ∞,
C: ∞,
D: ∞,
}
```

Рис. 16.13. Вид структур данных в алгоритме при первом запуске

На данном этапе у вашей очереди с приоритетом есть только одна вершина, поэтому вы выталкиваете вершину A и ее расстояние от начальной вершины (0) из очереди с приоритетом и проверяете, нашли ли вы уже кратчайший путь до этой вершины. Когда вы просматриваете новую вершину в вашей очереди с приоритетом и уже нашли кратчайший путь от начальной вершины до этой вершины, делать ничего не нужно. В нашем примере вы еще не нашли кратчайший путь до вершины A , поэтому вы перебираете все смежные с A вершины.

Далее вы вычисляете расстояния от смежных вершин до начальной вершины и делаете это путем добавления расстояния, полученного из очереди с приоритетом, к весу смежной вершины (ее расстояния от вершины, которую вы вытолкнули из очереди с приоритетом). Очередь с приоритетом содержит два фрагмента информации для каждой вершины в ней — вершину и ее расстояние от начальной вершины, поэтому независимо от того, как далеко вы отошли от начальной вершины, вы можете легко вычислить расстояние между новой смежной вершиной и начальной, добавив расстояние, полученное из очереди с приоритетом, к весу смежной вершины.

Если расстояние от смежной вершины до начальной короче, чем любой другой путь, который вы до этого нашли, вы добавляете новый путь в ваш словарь и помещаете смежную вершину в очередь с приоритетом. В данном примере вы помещаете обе смежные с A вершины (B и C) в очередь с приоритетом и добавляете их пути в словарь (рис. 16.14).

```
Unvisited Vertices {A, B, C, D}
Priority Queue [(2, B), (6, C)]
Distances {
A: 0,
B: 2,
C: 6,
D: ∞,
}
```

Рис. 16.14. Структуры данных после посещения вершины A

Теперь вы выталкиваете вершину B из очереди с приоритетом, потому что она имеет наивысший приоритет (кратчайший путь от начальной вершины). Вы еще не нашли более короткий путь от B до начальной вершины, поэтому продолжаете посещать эту вершину. Вы проверяете все смежные с ней вершины в поисках более короткого пути, добавляете в словарь любой найденный более короткий путь и обновляете очередь с приоритетом. В данном примере у B есть только одна смежная вершина с более коротким путем — D, поэтому вы обновляете D в словаре до 7 и добавляете D и ее расстояние от начальной вершины в очередь с приоритетом (рис. 16.15).

```
Unvisited Vertices {A, B, C, D}
Priority Queue [(6, C) (7, D)]
Distances {
A: 0,
B: 2,
C: 6,
D: 7,
}
```

Рис. 16.15. Структуры данных после посещения вершины B

Теперь вы выталкиваете вершину C из очереди с приоритетом, потому что она имеет кратчайший путь в очереди с приоритетом. C также является смежной с D, но, как вы видели ранее, ее расстояние от начальной вершины равно 14 и вы уже нашли более короткий путь для D, поэтому вы не добавляете вершину D снова в очередь с приоритетом (рис. 16.16). Игнорирование более длинных путей (непосещение их повторно) — это то, что делает ваш алгоритм таким эффективным.


```

Unvisited Vertices {A, B, C, D}
Priority Queue [(7, D)]
Distances {
A: 0,
B: 2,
C: 6,
D: 7,
}

```

Рис. 16.16. Структуры данных после посещения вершины C

Вершина D не смежна ни с какими другими вершинами, поэтому после того, как вы ее вытолкнули, ваш алгоритм завершается (рис. 16.17).

```

Unvisited Vertices {A, B, C, D}
Priority Queue [ ]
Distances {
A: 0,
B: 2,
C: 6,
D: 7,
}

```

Рис. 16.17. Структуры данных после посещения вершины D

Ниже приведен код для реализации алгоритма Дейкстры на языке Python. В данном примере ваш алгоритм ожидает, что граф скорее будет словарем словарей, а не классом `Graph`, код для которого вы писали ранее в этой главе.

```

import heapq

def dijkstra(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0
    pq = [(0, starting_vertex)]

    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

```

```
graph = {
    'A': {'B': 2, 'C': 6},
    'B': {'D': 5},
    'C': {'D': 8},
    'D': {},
}

dijkstra(graph, 'A')
print(dijkstra(graph, 'A'))
```

Сначала вы импортируете `heapq`, потому что алгоритм использует кучу в качестве очереди с приоритетом. Функция `dijkstra` возвращает словарь, содержащий кратчайшие пути от начальной вершины. Функция принимает два параметра: граф и вершину, из которой вы хотите найти кратчайший путь.

```
import heapq

def dijkstra(graph, starting_vertex):
```

В этой реализации вы передаете список смежности так:

```
graph = {
    'A': {'B': 2, 'C': 6},
    'B': {'D': 5},
    'C': {'D': 8},
    'D': {},
}
```

Когда вы вызываете функцию `dijkstra`, вы передаете граф и строку для представления начальной вершины следующим образом:

```
dijkstra(graph, 'A')
```

Начальная вершина должна быть вершиной в графе.

Внутри функции вы создаете словарь под названием `distances` для хранения путей от начальной вершины до каждой имеющейся в графе иной вершины. В конце алгоритма этот словарь будет содержать кратчайшие пути от начальной вершины до всех других вершин. Вы создаете словарь с помощью словарного включения, аналогичного списковому включению. Ваше словарное включение соотносит каждую вершину с `float('infinity')` — представлением бесконечности в Python. Так как ваш алгоритм сравнивает длины путей, а пути начинаются как неизвестные, для представления этого вы используете бесконечность.

```
distances = {vertex: float('infinity') for vertex in graph}
```

Когда вы передаете словарь (представляющий граф, показанный выше в описании алгоритма Дейкстры) в `dijkstra`, предыдущий код создает такой словарь:

```
{'A': inf, 'B': inf, 'C': inf, 'D': inf}
```

Затем вы устанавливаете расстояние от начальной вершины (вершины, от которой вы ищете все кратчайшие пути) к самой себе равным 0, поскольку расстояние от вершины к самой себе равно 0.

```
distances[starting_vertex] = 0
```

Далее вы создаете список (его вы будете использовать как очередь с приоритетом), содержащий начальную вершину и ее расстояние от начальной вершины (0):

```
pq = [(0, starting_vertex)]
```

Затем следует код для посещения вершин в очереди с приоритетом. Цикл `while` выполняется до тех пор, пока в очереди с приоритетом все еще есть одна или более вершин. Вы используете цикл `while` для посещения всех вершин в вашем графе.

```
while len(pq) > 0:
```

Внутри цикла `while` вы выталкиваете расстояние от начальной вершины и текущую вершину из очереди с приоритетом и сохраняете их в переменных `current_distance` и `current_vertex`. Текущая вершина — это вершина в очереди с приоритетом, имеющая наименьшее расстояние от начальной вершины. Ваша очередь с приоритетом автоматически предоставляет вам вершину с наименьшим расстоянием, когда вы выталкиваете из нее новую вершину (потому что очередь с приоритетом является минимальной кучей).

```
current_distance, current_vertex = heapq.heappop(pq)
```

Вы обрабатываете вершину, только если еще не нашли более короткий путь от этой вершины до начальной. Вот почему далее вы проверяете, является ли текущее расстояние от начальной вершины больше, чем расстояние, которое вы уже внесли в словарь `distances`. Когда оно больше, вы не рассматриваете этот путь, так как уже записали более короткий, и используете ключевое слово `continue`, чтобы вернуться к началу цикла `while` и проверить другую вершину (при ее наличии).

```
if current_distance > distances[current_vertex]:
    continue
```

Если переменная `current_distance` не больше, чем `distances[current_vertex]` (другими словами, короче или равна), вы перебираете все вершины, смежные с текущей.

```
for neighbor, weight in graph[current_vertex].items():
```

Для каждой смежной вершины вы вычисляете ее расстояние от начальной вершины путем добавления значения `current_distance` к ее весу. Это вычисление

работает, потому что `current_distance` отображает, насколько далеко от начальной вершины находится текущая вершина, а переменная `weight` отображает, насколько далеко смежная вершина находится от текущей. Поэтому, когда вы складываете обе переменные, вы получаете расстояние от начальной вершины.

```
distance = current_distance + weight
```

Далее вы проверяете, является ли вновь найденный вами путь для смежной вершины более коротким, чем путь, уже записанный для этой вершины в словаре `distances`. Если да, вы обновляете словарь, добавляя в него этот новый путь. Затем вы проталкиваете новое расстояние и вершину в вашу очередь с приоритетом, чтобы алгоритм мог их посетить.

```
if distance < distances[neighbor]:
    distances[neighbor] = distance
    heapq.heappush(pq, (distance, neighbor))
```

Когда вы выходите из цикла `while`, это означает, что вы изучили все вершины и словарь `distances` теперь содержит кратчайший путь от начальной вершины до какой-либо иной вершины в вашем графе. Все, что теперь остается сделать, это вернуть `distances`.

```
return distances
```

Словарь терминов

Граф — абстрактный тип данных, в котором фрагмент данных соединяется с одним или несколькими другими фрагментами данных.

Вершина — фрагмент данных в графе.

Полезная нагрузка — дополнительные данные в вершине графа.

Ребро — соединение между двумя вершинами в графе.

Вес — затраты на перемещение между вершинами.

Ориентированный граф — граф, в котором у каждого ребра есть связанное с ним направление, причем перемещаться между двумя вершинами можно только в этом направлении.

Неориентированный граф — тип графа с двунаправленными ребрами, позволяющий перемещаться вперед и назад в любом направлении между двумя связанными вершинами.

Полный граф — граф, в котором каждая вершина соединяется со всеми другими вершинами.

Неполный граф — граф, в котором некоторые, но не все, вершины не соединены.

Путь — последовательность вершин, соединенных ребрами.

Цикл — путь в графе, начинающийся и заканчивающийся в одной и той же вершине.

Ациклический граф — граф, не имеющий цикла.

Список ребер — структура данных, в которой вы представляете каждое ребро в графе с помощью двух соединенных вершин.

Матрица смежности — двумерный массив строк и столбцов, содержащий вершины графа.

Смежные вершины — две или более соединенные вершины в графе.

Список смежности — набор неупорядоченных списков, каждый из которых представляет соединения для одной вершины.

Алгоритм Дейкстры — алгоритм, который можно использовать для нахождения кратчайшего пути от одной вершины в графе к любой другой вершине.

Практикум

1. Измените алгоритм Дейкстры таким образом, чтобы он просто возвращал путь от начальной вершины к другой вершине, которую вы передаете.

17

Вдохновение самоучки: Илон Маск

Сегодня Илон Маск наиболее известен такими проектами, как Tesla, SpaceX и PayPal, которые произвели революцию в своих отраслях. Но задолго до того, как Маск стал предпринимателем и одним из богатейших людей на планете, у него была одна более простая идея: он хотел создавать видеоигры. Как же программист-самоучка Илон Маск прошел путь от ребенка, играющего в игры, до миллиардера? В этой главе вы узнаете об образовании Маска и о том, как интерес к играм привел его к изучению программирования.

Илон Маск родился и вырос в Южной Африке и заинтересовался компьютерами, когда ему было десять лет. Он был увлекающимся ребенком и мог иногда проводить по десять часов в день за чтением книг. Он был также одержим видеоиграми. Маск объяснял, что любовь к видеоиграм дала ему мотивацию для изучения программирования. «Я думал, что смогу создавать свои собственные игры. Я хотел увидеть, как игры работают, — объяснял он. — Это привело меня к изучению программирования компьютеров». Маск начал с книги о языке программирования BASIC, популярном в 1960-е годы: в 1980-х многие компьютеры его все еще использовали. Книга предлагала шестимесячную программу обучения кодированию, но Маск освоил всю программу за три дня. Это было незадолго до того, как он запрограммировал свою первую видеоигру. В 1984 году, когда ему было 12 лет, Илон Маск создал *Blastar*. Космический шутер черпал свое вдохновение из видеоигры *Alien Invaders*. В игре Маска игрок сбивал космические корабли, перевозившие водородные бомбы, уклоняясь от смертоносных «статусных лучей».

Маск продал свою игру компании *PC and Office Technology*, которая предложила за *Blastar* 500 долларов. Таким образом, уже в своем первом дерзком начинании Маск научился извлекать прибыль, что стало важным поворотным моментом

в его образовании. История с *Blastar* преподавала Маску несколько жизненно важных уроков. Во-первых, после прочтения книги и попыток кодирования он понял, что может создавать свои собственные видеоигры; во-вторых, реализация полученных знаний в виде готового продукта также принесла результат — уже в 12 лет благодаря своим навыкам программирования Илон Маск заработал деньги.

Однако образование Маска не завершилось на этом этапе. Внутренний импульс, подтолкнувший заняться программированием, остался с ним и в подростковом возрасте. В 17 лет Илон Маск переехал из Южной Африки в Канаду, планируя жить у своего двоюродного деда в Монреале. Но была одна проблема: дед уже переехал в Миннесоту, о чем Маск узнал, только когда прилетел в Канаду. Но он не сдался. У него были другие дальние родственники в Канаде, поэтому он купил билет на автобус и начал их искать. Ему понадобилось проехать на автобусе почти 2 тысячи миль, чтобы найти троюродного кузена, который предложил Маску пожить у него. Еще подростком Маск работал на ферме в Саскачеване, рубил дрова в Ванкувере и чистил котлы.

В своей книге 2017 года *Elon Musk: Tesla, SpaceX and the Quest for a Fantastic Future* («Илон Маск: Tesla, SpaceX и поиски фантастического будущего») он так описывал чистку котлов: «Вам надо надеть защитный костюм и потом пробраться сквозь маленький тоннель, в который вы едва помещаетесь. Затем лопатой вы должны выгребать песок, лишнюю противную массу и другие отходы, которые все еще дымятся, обратно в ту же дыру, через которую сюда попали. И выхода нет. Кто-то другой на противоположной стороне собирает это в тачку. Если вы останетесь там больше, чем на 30 минут, вы перегреетесь и умрете».

В 1989 году Маск поступил в Королевский университет Онтарио. В колледже он как-то сказал другу: «Если бы я мог не есть, чтобы больше работать, я бы не ел. Хорошо бы был способ получать питательные вещества, не садясь за стол с едой!» Неуспокоенность Маска продолжала толкать его дальше. Он собирал и продавал компьютеры в своей комнате в общежитии. «Я мог создать именно то, что было нужно покупателям, например навороченный игровой автомат или простой текстовый процессор, которые были дешевле, чем в магазине», — рассказывал Маск.

Он часами играл в такие игры, как *Civilization*, и размышлял о карьере в области игровой индустрии. После перехода в Университет Пенсильвании Маск начал склоняться к бизнесу и технологиям. Несмотря на то что компьютерные игры были его страстью еще с детства, Маск хотел стать влиятельным в другой сфере. «Я и вправду люблю компьютерные игры, но если бы я всего лишь делал очень классные игры, как сильно это повлияло бы на мир? — размышлял Маск. — Это не произвело бы большого эффекта. Даже несмотря на то, что

у меня врожденная любовь к видеоиграм, я не смог бы заставить себя сделать это своей карьерой».

Во время учебы в колледже Маск понимал, что способен быстро учиться. Он уже тогда интересовался солнечной энергией, космосом, интернетом и электрокарами. После получения диплома бакалавра в области экономики и физики Маск переехал в Калифорнию, чтобы получить докторскую степень в области энергетической физики в Стэнфорде. Кремниевая долина быстро отвлекла внимание Маска, и он бросил докторскую программу уже через два дня.

Вместо этого Илон Маск запустил корпорацию Zip2, которая создавала онлайн-гиды городов: эту компанию в 1999 году он продал более чем за 300 миллионов долларов. С тех пор Маск сотрудничал со многими успешными компаниями, включая PayPal, SpaceX, Tesla Motors и The Boring Company. Импульс, подтолкнувший Маска изучать программирование самостоятельно, помог ему стать одним из самых успешных предпринимателей всех времен.

18

Дальнейшие шаги

Для большинства людей на Земле цифровая революция еще даже не началась. В течение следующих 10 лет все это изменится. Давайте заставим весь мир кодировать!

Эрик Шмидт

Отлично! Вы справились со всеми техническими частями этой книги. Ваша усердная работа уже окупается, и вы на верном пути, чтобы стать инженером-программистом. Я хочу поблагодарить вас за то, что вы решили прочесть мою книгу и что вы являетесь членом сообщества самоучек. Я не могу поверить в то, каким большим стало наше сообщество. Было потрясающе получить возможность встретить столько вдохновляющих людей, и я не могу дождаться, чтобы прочитать вашу историю успеха следующей. В заключительной главе я расскажу, что вам нужно сделать для своего дальнейшего продвижения, и укажу некоторые источники, которые вам в этом помогут.

Что дальше?

Для начала давайте на секундочку задержимся и отпразднуем то, насколько далеко вы продвинулись в качестве программиста-самоучки. Вы не только умеете программировать, но также разбираетесь во многих фундаментальных понятиях компьютерной науки. Вы знаете, как писать алгоритмы для решения различных задач, и вы можете посмотреть на два алгоритма и быстро решить, какой из них вы хотите использовать. Вы можете писать рекурсивные алгоритмы для элегантного решения задач, поиска и сортировки данных различными

способами. Вы знакомы с разными структурами данных и знаете не только то, что они собой представляют, но и когда их использовать. В целом вы стали более осведомленным программистом с большим количеством новых инструментов в вашем арсенале.

Вы не только значительно расширили свои знания в области программирования, но и, немного попрактиковавшись, сможете пройти техническое собеседование. А это означает, что вы на верном пути к получению должности инженера-программиста. Итак, что же вам делать дальше? Это зависит от того, сколько у вас профессионального опыта в программировании. Если опыт уже есть, можете пропустить следующий раздел. Если у вас его нет и вы хотите увеличить свои шансы на получение работы, читайте дальше.

Карьера фрилансера

Одна из трудных вещей, с которой сталкиваются программисты-самоучки, — это поиск работы без какого-либо опыта. Классическая проблема: нужен опыт, чтобы получить работу, и нужна работа, чтобы получить опыт. Я нашел решение для данной проблемы, и его с большим успехом используют мои студенты. Я называю свой метод *подъемом по лестнице фриланса* — я сам его использовал, чтобы получить работу инженера-программиста в eBaу без опыта работы в традиционной компании.

До получения работы в eBaу я был программистом-фрилансером. Ключом к моему успеху стал тот факт, что я не пытался начинать с крупного проекта, а взялся за небольшие задачи на платформе Elance (сейчас Upwork). Если вы не знакомы с Upwork — это сайт для фрилансеров. Работодатели, ищущие работников по контракту, размещают проект, а исполнители предлагают свою цену за выполнение работы. Есть и другие сайты, подобные Upwork, такие как [freelancer.com](https://www.freelancer.com) и [Fiverr](https://www.fiverr.com). Моя первая работа стоила около 25 долларов и заняла у меня несколько часов: не слишком большая почасовая оплата. К счастью, это дало мне опыт, к тому же клиент оказался доволен тем, что я сделал, и поставил мне оценку в пять звезд; благодаря этому получить следующий проект мне было немного легче. И снова я усердно работал и заслужил оценку в пять звезд. Я прокладывал себе дорогу к все более крупным проектам и наконец добился проектов стоимостью в тысячу долларов.

Когда я проходил собеседование в eBaу, у меня не было опыта программирования в традиционной компании. Однако во время собеседования я сконцентрировался на рассказе о своих фриланс-проектах. Мои интервьюеры в eBaу почувствовали, что опыт работы фрилансером позволяет мне считаться достойным кандидатом, и в результате предложили мне работу. Если бы я пошел на это собеседование без опыта работы в традиционной компании и без опыта

работы фрилансером, сомневаюсь, что я получил бы ту самую работу, с которой началась моя карьера.

Если вы хотите работать инженером-программистом, но у вас нет никакого профессионального опыта, не начинайте ходить на собеседования. Зарегистрируйтесь на платформе типа Upwork и попробуйте получить любую работу, даже если она стоит всего 25 долларов. Затем продвигайтесь по карьерной лестнице фрилансера, зарабатывая оценки в пять звезд и делая ставки на проекты, которые платят больше. Накопив внушительный опыт, вы будете готовы подать заявку на работу своей мечты в качестве инженера-программиста в избранной вами компании.

Как попасть на собеседование

Свою первую работу инженера-программиста я получил через LinkedIn. LinkedIn — все еще фантастически эффективный ресурс для поиска работы. Как только вы получите немного опыта, взбираясь по лестнице фриланса, я рекомендую потратить время на обновление профиля на LinkedIn. Обязательно внесите в резюме опыт фриланса в качестве последней работы и укажите должность инженера-программиста. Подумайте о том, чтобы попросить какие-нибудь компании, с которыми вы сотрудничали, подтвердить для LinkedIn ваши навыки программирования.

Как только вы обновите профиль на LinkedIn (а также резюме), придет время налаживать связи. Я рекомендую выбрать от пяти до десяти интересных вам компаний и обратиться к их рекрутерам или другим членам команды. Компании часто испытывают нехватку инженеров и предлагают реферальные бонусы тем сотрудникам, которые направляют инженеров. Поэтому, если вы достаточно квалифицированы для должности, они наверняка будут вам рады.

Можно использовать такой ресурс, как Meetup.com, для поиска групп, которые собираются вместе для общения и знакомства с новыми людьми. Можно также напрямую подать заявку на работу через такие сайты, как Angel.co или Indeed.com.

Как подготовиться к техническому интервью

После отправки заявлений для устройства на работу вам в конце концов придется пройти техническое интервью. Подготовка к нему потребует много времени: нет какого-либо неизменного правила, но я рекомендую выделить на это два или три месяца. Значение имеет и конкурентоспособность компаний, в которые вы отправляете резюме. Если вы обращаетесь в одну из компаний Великой

пятерки FAANG (Facebook, Amazon, Apple, Netflix, Google/Alphabet), нет ничего удивительного в том, что подающие надежды инженеры тратят шесть и более месяцев для подготовки к оценке своих технических навыков этими компаниями. С другой стороны, если вы подаете заявку в стартап, на подготовку может хватить и нескольких недель.

Я рекомендую как минимум несколько часов в день посвящать решению задач на LeetCode, одном из моих любимых ресурсов для подготовки к техническому интервью: на нем есть сотни практических задач по алгоритмам и структурам данных, а также решения этих задач.

Одна из особенностей технического интервью — это неестественная обстановка, в которой вы оказываетесь. Обычно у вас никто не стоит над душой и не оценивает вас, когда вы программируете. Программисты также не привыкли решать задачи за короткий промежуток времени. Тем не менее эти искусственные ограничения — как раз то, с чем вы столкнетесь лицом к лицу на техническом интервью. Соревновательное программирование, представляющее кодинг как спорт, — лучшее решение, которое я нашел для подготовки к интервью в подобной среде. Вы соревнуетесь с другими программистами в скорости и качестве решения задач. Это лучший способ подготовиться к техническому интервью, потому что он готовит вас к уникальным условиям, с которыми вы столкнетесь: решению задач за ограниченное время. Когда я соревновался по программированию для подготовки к ряду технических интервью, я значительно лучше показал себя на них, чем когда готовился, просто решая задачи наедине с собой. Если вы готовы принять участие в соревновании, попробуйте такой сайт, как Codeforces.

После участия в соревновательном программировании с целью обретения навыка быстро решать сложные технические задачи вам стоит попробовать симулировать несколько интервью с другим инженером-программистом — в идеале с тем, кто уже проводил интервью раньше. Если вы не можете найти друга, способного вам помочь, попробуйте нанять опытного инженера-программиста на таких фриланс-площадках, как Upwork или Codementor. Это будет вам стоить примерно 30–60 долларов, но несколько часов фиктивного интервью станут отличным вложением.

Дополнительные источники

Как вы уже знаете, компьютерная наука — обширная тема. Моей целью было написать настолько интересную книгу, чтобы вы дочитали ее до конца, и осветить в ней темы, которые с наибольшей вероятностью помогут вам в вашей карьере программиста. К сожалению, это означает, что многое осталось за пределами данной книги. В этом разделе я коротко освещу некоторые невошедшие темы и укажу источники, из которых вы можете о них узнать.

Хотя мы и говорили о двоичных деревьях, я не слишком углублялся в тему. Возможно, вы захотите посвятить больше времени изучению различных типов деревьев, таких как двоичные деревья поиска, AVL-деревья или деревья разбора. Мы также не рассмотрели все общепринятые алгоритмы сортировки. Если вы готовитесь к техническому интервью и хотите узнать больше об алгоритмах сортировки, вам следует изучить пирамидальную сортировку, сортировку выбором, быструю сортировку, сортировку подсчетом и поразрядную сортировку.

Обо всем этом и о многом другом вы можете узнать в книге Томаса Кормена *Introduction to Algorithms*¹. Однако предупреждаю, что читать ее нелегко. Теперь, когда вы знаете основы программирования, вам будет значительно легче понимать смысл написанного. Книга *Computer Science Illuminated* Нелла Дейла и Джона Льюиса — великолепный выбор для более подробного изучения тем компьютерной науки за пределами структур данных и алгоритмов.

Заключительные размышления

Спасибо за то, что выбрали мою книгу. Я надеюсь, вам понравилось ее читать так же, как мне понравилось ее писать. Если у вас есть какие-то вопросы или комментарии, не стесняйтесь писать мне в группе программистов-самоучек на Facebook по адресу <https://facebook.com/groups/selftaughtprogrammers>. Вы можете также подписаться на мою новостную рассылку, чтобы быть в курсе событий сообщества самоучек, расположенного по адресу <https://selftaught.blog>. И наконец, вы можете поддерживать со мной связь в социальных сетях. Мой идентификатор в социальных сетях Instagram, Twitter и Facebook — @coryalhoff. Если у вас есть минутка, чтобы оставить отзыв на Amazon, я буду бесконечно благодарен. Каждый отзыв очень помогает продажам книги, что позволяет мне продолжать создавать новый образовательный материал для программистов-самоучек.

Берегите себя!

¹ Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ.

Кори Альтхофф

**Computer Science для программиста-самоучки.
Все, что нужно знать о структурах данных и алгоритмах**

Перевел с английского С. Черников

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>А. Котов</i>
Литературные редакторы	<i>Н. Кудрейко, Н. Хлебина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректор	<i>Е. Рафалюк-Бузовская</i>
Верстка	<i>Г. Блинов</i>

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: 12.2022. 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 19.10.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 19,350. Тираж 1200. Заказ 0000.

Роберт Седжвик, Кевин Уэйн

COMPUTER SCIENCE: ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA, ООП, АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ



Преподаватели Принстонского университета Роберт Седжвик и Кевин Уэйн создали универсальное введение в Computer Science на языке Java, которое идеально подходит как студентам, так и профессионалам. Вы начнете с основ, освоите современный курс объектно-ориентированного программирования и перейдете к концепциям более высокого уровня: алгоритмам и структурам данных, теории вычислений и архитектуре компьютеров.

И главное — вся теория рассматривается на практических и ярких примерах: прикладная математика, физика и биология, числовые методы, визуализация данных, синтез звука, обработка графики, финансовое моделирование и многое другое.

КУПИТЬ

Алекс Петров

РАСПРЕДЕЛЕННЫЕ ДАННЫЕ. АЛГОРИТМЫ РАБОТЫ СОВРЕМЕННЫХ СИСТЕМ ХРАНЕНИЯ ИНФОРМАЦИИ



Когда дело доходит до выбора, использования и обслуживания базы данных, важно понимать ее внутреннее устройство. Как разобраться в огромном море доступных сегодня распределенных баз данных и инструментов? На что они способны? Чем различаются?

Алекс Петров знакомит нас с концепциями, лежащими в основе внутренних механизмов современных баз данных и хранилищ. Для этого ему пришлось обобщить и систематизировать разрозненную информацию из многочисленных книг, статей, постов и даже из нескольких баз данных с открытым исходным кодом.

Вы узнаете о принципах и концепциях, используемых во всех типах СУБД, с акцентом на подсистеме хранения данных и компонентах, отвечающих за распределение. Эти алгоритмы используются в базах данных, очередях сообщений, планировщиках и в другом важном инфраструктурном программном обеспечении. Вы разберетесь, как работают современные системы хранения информации, и это поможет взвешенно выбирать необходимое программное обеспечение и выявлять потенциальные проблемы.

КУПИТЬ