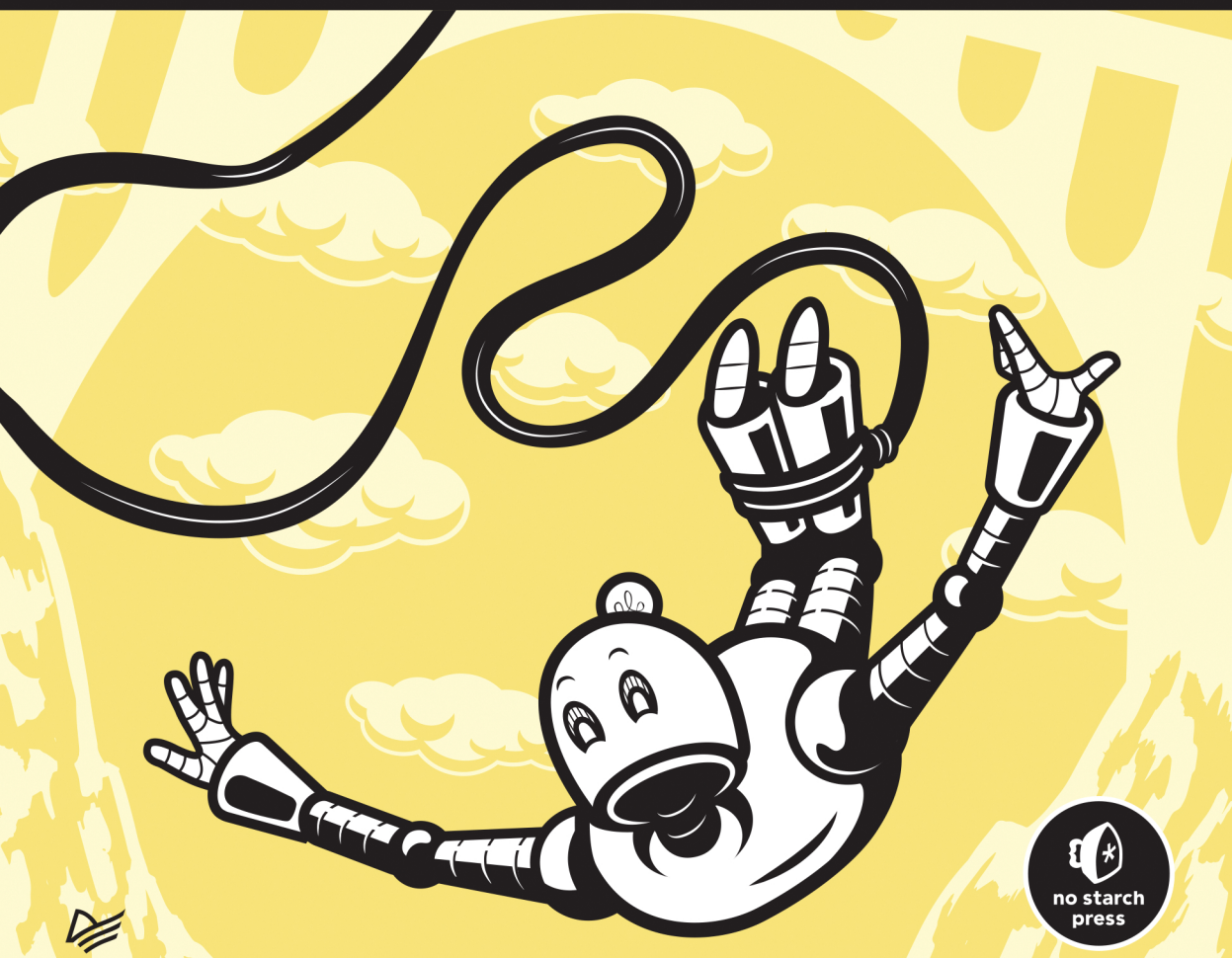
АЛГОРИТМЫ НЕФОРМАЛЬНО

ИНСТРУКЦИЯ ДЛЯ НАЧИНАЮЩИХ ПИТОНИСТОВ

БРЭДФОРД ТАКФИЛД



no starch press

DIVE INTO ALGORITHMS

A Pythonic Adventure for the Intrepid Beginner

Bradford Tuckfield

no starch press

San Francisco

АЛГОРИТМЫ НЕФОРМАЛЬНО

ИНСТРУКЦИЯ ДЛЯ НАЧИНАЮЩИХ ПИТОНИСТОВ

БРЭДФОРД ТАКФИЛД

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018
УДК 004.421
Т15

Такфилд Брэдфорд

Т15 Алгоритмы неформально. Инструкция для начинающих питонистов. — СПб.: Питер, 2022. — 272 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1919-6

Алгоритмы — это не только задачи поиска, сортировки или оптимизации, они помогут вам поймать бейсбольный мяч, проникнуть в «механику» машинного обучения и искусственного интеллекта и выйти за границы возможного.

Вы узнаете нюансы реализации многих самых популярных алгоритмов современности, познакомитесь с их реализацией на Python 3, а также научитесь измерять и оптимизировать их производительность.

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

ББК 32.973.2-018
УДК 004.421

ISBN 978-1718500686 англ.

ISBN 978-5-4461-1919-6

Краткое содержание

Оглавление

Посвящается моим родителям Дэвиду и Бекки Такфилд,
которые когда-то поверили в меня
и научили играть в «Точки и квадраты»

Об авторе

Брэдфорд Такфилд (Bradford Tuckfield) — специалист по теории данных и писатель. Руководит фирмой Kmbara (https://kmbara.com/), занимающейся консультациями в области теории данных, а также ведет литературный сайт Dreamtigers (http://thedreamtigers.com/).

О научном редакторе

Алок Малик (Alok Malik) — специалист по теории данных из Нью-Дели (Индия). Занимается созданием моделей глубокого обучения в областях обработки естественного языка и распознавания изображений на Python. Малик разрабатывал такие решения, как языковые модели, классификаторы изображений и текста, системы перевода, модели преобразования речи в текст, системы распознавания именованных сущностей и детекторы объектов. Кроме того, он участвовал в написании книги о машинном обучении. В свободное время любит читать о финансах, преподает на курсах дистанционного обучения и играет на приставке.

Благодарности

«Одно и то же слово по-разному звучит у разных писателей. Один выдирает его из своего нутра. Другой вынимает его из кармана пальто». Так Шарль Пеги (Charles Peguy) сказал о написании отдельных слов. То же относится к главам и целым книгам. В одних случаях мне казалось, что я вынимал эту книгу из кармана пальто. В других все выглядело так, словно я выдирал ее из своего нутра. Будет уместно поблагодарить всех, кто внес свой вклад в этот долгий процесс — либо одалживая мне пальто, либо помогая мне залечить нутро.

Многие хорошие люди помогали мне на долгом пути, который я прошел, чтобы получить опыт и квалификацию, необходимые для написания книги. Мои родители Дэвид и Бекки Такфилд преподнесли мне множество бесценных даров, начиная с жизни и образования, и продолжали верить и вдохновлять меня, а также помогать мне в других отношениях — слишком многочисленных, чтобы я мог их здесь перечислить. Скотт Робертсон (Scott Robertson) дал мне первую работу по написанию кода, хотя моей квалификации было явно недостаточно. Рэнди Дженсон (Randy Jenson) принял меня на первую работу, связанную с теорией данных, — и снова несмотря на мою неопытность и ограниченный опыт. Кумар Кашьяп (Kumar Kashyap) предоставил мне первую возможность руководить группой разработки для реализации алгоритмов. Дэвид Зу (David Zou) стал первым человеком, заплатившим мне за написание статьи (десять долларов за вычетом процента PayPal за десять коротких обзоров фильмов), и это было настолько прекрасно, что я решил более плотно заняться писательской работой. Адитья Дейт (Aditya Date) был первым, кто предложил мне написать книгу и предоставил первую возможность для этого.

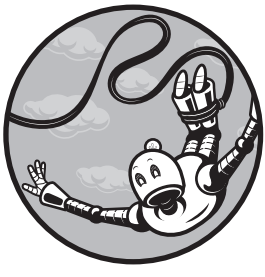
Меня также вдохновляли многие учителя и наставники. Дэвид Кардон (David Cardon) предоставил первую возможность участвовать в академических исследованиях и многому научил в процессе работы. Брайан Скелтон (Bryan Skelton) и Леонард Ву (Leonard Woo) стали для меня примерами тех, на кого я бы хотел быть похожим, когда вырасту. Уэс Хатчинсон (Wes Hutchinson) научил важнейшим

алгоритмам (таким как кластеризация методом k-средних) и помог лучше понять их работу. Чед Эммет (Chad Emmett) научил думать об истории и культуре, я посвятил ему главу 2. Ури Саймонсон (Uri Simonsohn) показал, как следует подходить к анализу данных.

Некоторые люди помогли мне превратить написание книги в приятное занятие. Сешу Эдала (Seshu Edala) помог отрегулировать мой рабочий график, чтобы высвободить время для работы над книгой, и постоянно подбадривал меня. С Алексом Фридом (Alex Freed) было очень приятно работать на протяжении всего процесса редактирования. Дженнифер Игар (Jennifer Eagar) неофициально стала первым человеком, купившим экземпляр книги за несколько месяцев до ее выхода; это помогало мне в трудные времена. Хланг Хланг Тун (Hlaing Hlaing Tun) была всегда готова поддержать и помочь, оставалась милой и доброжелательной на каждом этапе.

Я не могу вернуть весь долг благодарности, но, по крайней мере, могу сказать спасибо всем, кто мне помогал. Спасибо!

Введение



Алгоритмы встречаются повсюду. Вероятно, вы уже выполнили сегодня как минимум несколько алгоритмов. В книге вы прочтете о десятках алгоритмов: простых и сложных, знаменитых и малоизвестных — но все они интересны и заслуживают вашего внимания. Первый алгоритм в книге также оказывается самым вкусным — он описывает процесс приготовления парфе с гранолой и ягодами; данный алгоритм полностью приведен на рис. 1. Возможно, вы привыкли называть алгоритмы такого типа «рецептами», но он соответствует определению *алгоритма* у Дональда Кнута: конечный набор правил, определяющий последовательность операций для решения конкретного типа задач.

Парфе с гранолой и ягодами

Указания

1. Добавьте 1/6 чашки голубики в креманку.
2. Выложите половину чашки турецкого йогурта на голубику.
3. Выложите 1/3 чашки гранолы на йогурт.
4. Выложите половину чашки турецкого йогурта на гранолу.
5. Положите клубнику на содержимое креманки.
6. Украсьте взбитыми сливками.

Рис. 1. Алгоритм: конечный набор правил, определяющий последовательность операций для решения конкретного типа задач

Приготовление десертов — не единственная область жизни, управляемая алгоритмами. Ежегодно правительство США требует, чтобы каждый взрослый гражданин выполнял определенный алгоритм, и старается посадить в тюрьму тех, кто делает

это неправильно. В 2017 году миллионы американцев выполнили свой долг, завершив алгоритм, показанный на рис. 2, который был взят из формы 1040-EZ.

1	Wages, salaries, and tips. This should be shown in box 1 of your Form(s) W-2. Attach your Form(s) W-2.	1
2	Taxable interest. If the total is over $1,500, you cannot use Form 1040EZ.	2
3	Unemployment compensation and Alaska Permanent Fund dividends (see instructions).	3
4	Add lines 1, 2, and 3. This is your **adjusted gross income.**	4
5	If someone can claim you (or your spouse if a joint return) as a dependent, check the applicable box(es) below and enter the amount from the worksheet on back. ☐ **You** ☐ **Spouse** If no one can claim you (or your spouse if a joint return), enter $10,400 if **single;** $20,800 if **married filing jointly.** See back for explanation.	5
6	Subtract line 5 from line 4. If line 5 is larger than line 4, enter -0-. This is your **taxable income.** ▶	6
7	Federal income tax withheld from Form(s) W-2 and 1099.	7
8a	**Earned income credit (EIC)** (see instructions)	8a
b	Nontaxable combat pay election. 8b	
9	Add lines 7 and 8a. These are your **total payments and credits.** ▶	9
10	**Tax.** Use the amount on **line 6 above** to find your tax in the tax table in the instructions. Then, enter the tax from the table on this line.	10
11	Health care: individual responsibility (see instructions) Full-year coverage ☐	11
12	Add lines 10 and 11. This is your **total tax.**	12

Рис. 2. Инструкции по заполнению налоговой декларации соответствуют определению алгоритма

Что общего у налогов и ягодного десерта? Налоги нужно обязательно платить, они выражаются в числовом виде, рассчитываются по сложным формулам, и люди их обычно терпеть не могут. Десерты встречаются не так часто, являются произведением искусства, и их все обожают. Единственное, что у них есть общего, — при подготовке и того и другого используются алгоритмы.

Великий ученый в области вычислительной теории Дональд Кнут замечал, что термин «алгоритмы» почти синонимичен терминам *«рецепт»*, *«процедура»* и *«рутина»*. При декларировании налогов с использованием формы 1040-EZ выполняются 12 шагов (конечный список), определяющих операции (такие как сложение на шаге 4 и вычитание на шаге 6) для решения конкретной задачи: стремления избежать тюремного заключения за уклонение от уплаты налогов. Для приготовления

парфе нужны шесть шагов, определяющих операции (такие как добавление ягод на шаге 1 и выкладывание йогурта на шаге 2) для решения конкретной задачи: желания отведать десерт.

Когда вы больше узнаете об алгоритмах, вы начнете видеть их повсеместно и оцените, насколько впечатляющими они могут быть. В главе 1 мы обсудим выдающуюся человеческую способность ловить мяч и ознакомимся с подробностями алгоритма из области подсознания, который позволяет нам это делать. Позднее рассмотрим алгоритмы отладки кода, максимизации доходов, сортировки списков, планирования задач, правки текста, доставки почты и победы в таких играх, как шахматы или судоку. Попутно вы научитесь оценивать алгоритмы по нескольким атрибутам, которые, по мнению профессионалов, важны для алгоритмов. Со временем вы сможете в совершенстве овладеть *искусством* создания алгоритмов, которое открывает перспективу для творческого подхода и проявления личных качеств в точной и формальной области.

Для кого написана эта книга

В книге доступным языком описываются различные алгоритмы, при этом они сопровождаются кодом на Python. Чтобы извлечь из нее максимум пользы, вам понадобится некоторый опыт в областях, перечисленных ниже.

- **Программирование.** Все значимые примеры в книге поясняются с помощью кода Python. Я постарался предоставить подробный анализ и объяснения для каждого фрагмента кода, чтобы книга была понятной для читателя, не имеющего опыта программирования на Python и значительного опыта программирования вообще. Тем не менее читатель, обладающий хотя бы базовым пониманием основных концепций программирования — присваивания значений переменным, циклов `for`, команд `if/then` и вызовов функций, — будет лучше подготовлен к усвоению материала.
- **Школьный курс математики.** Алгоритмы часто используются для достижения тех же целей, для которых служат и математические конструкции: решение уравнений, оптимизация и вычисление значений. В алгоритмах также применяются многие принципы, связанные с математическим мышлением, например необходимость использования точных определений. Иногда в своих рассуждениях мы заходим на математическую территорию, включая алгебру, теорему Пифагора, число пи и основы математического анализа. Я постарался избежать хитроумных рассуждений и ограничиться рамками школьного курса математики.

Каждый, кто уверенно чувствует себя в указанных областях, сможет легко усвоить весь материал книги. Она была написана для нескольких групп читателей.

- **Учащиеся.** Книга подходит для изучения вводного курса алгоритмов, информатики или программирования уровня средней или высшей школы.
- **Профессионалы.** Практикующие специалисты тоже смогут узнать много полезного из книги. Это и программисты, желающие освоить Python, и разработчики, которые хотят расширить свои знания в области основ информатики и улучшить код за счет алгоритмического мышления.
- **Энтузиасты-любители.** Они составляют настоящую целевую аудиторию книги. Алгоритмы затрагивают практически каждую часть нашей жизни, поэтому каждый читатель сможет найти в издании что-то интересное, расширяющее границы восприятия окружающего мира.

О книге

В книге не рассматриваются все аспекты всех известных алгоритмов; это лишь вводный курс. Прочитав ее, вы будете четко понимать, что такое алгоритм, как писать код для реализации важных алгоритмов, и оценивать и оптимизировать эффективность алгоритмов. Вы также познакомитесь со многими популярными алгоритмами, которыми пользуются профессионалы. Ниже представлена структура издания.

- В **главе 1 «Алгоритмы при решении задач»** мы обсудим, как ловим мяч, поищем доказательства существования подсознательного алгоритма, управляющего человеческим поведением, и выясним, как это может нам помочь в практической реализации алгоритмов и их разработке.
- В **главе 2 «Алгоритмы в истории»** мы совершим путешествие по миру и во времени, чтобы узнать, как древние египтяне и русские крестьяне умножали числа, древние греки находили наибольшие общие делители, а средневековые японские ученые строили магические квадраты.
- В **главе 3 «Максимизация и минимизация»** представлены методы градиентного подъема и градиентного спуска. Эти простые методы поиска максимумов и минимумов функций используются для оптимизации — важной цели многих алгоритмов.
- В **главе 4 «Сортировка и поиск»** представлены фундаментальные алгоритмы сортировки списков и поиска в них элементов. Вы также узнаете, как оценивать эффективность и скорость работы алгоритмов.

- В **главе 5 «Чистая математика»** мы займемся чисто математическими алгоритмами, включая алгоритмы построения непрерывных дробей, вычисления квадратных корней и генерирования псевдослучайных чисел.
- В **главе 6 «Расширенная оптимизация»** рассматривается нетривиальный метод поиска оптимальных решений: имитация отжига. Кроме того, в ней представлена задача о коммивояжере — одна из стандартных задач современной информатики.
- В **главе 7 «Геометрия»** рассматривается генерирование диаграмм Вороного, которые находят применение во многих геометрических областях.
- В **главе 8 «Язык»** речь пойдет об осмысленной расстановке отсутствующих пробелов в тексте и формировании рекомендаций по выбору следующего слова во фразах.
- В **главе 9 «Машинное обучение»** рассматриваются деревья принятия решений — один из фундаментальных методов машинного обучения.
- В **главе 10 «Искусственный интеллект»** мы займемся амбициозным проектом: реализацией алгоритма, который может играть против нас и, возможно, выигрывать. Мы начнем с простой игры «Точки и квадраты» и поговорим о том, как можно улучшить быстродействие программы для этой игры.
- В **главе 11 «Полный вперед»** речь пойдет о том, как перейти к продвинутым задачам, связанным с алгоритмами. Вы узнаете, как построить чат-бот и выиграть миллион долларов, построив алгоритм для головоломки судоку.

Настройка окружения

Алгоритмы, описанные в книге, реализуются на Python. Он распространяется бесплатно, является языком с открытым исходным кодом и работает на всех основных платформах. Ниже описана процедура установки Python для систем Windows, macOS и Linux.

Установка Python в системе Windows

Чтобы установить Python в системе Windows, выполните следующие действия.

1. Откройте страницу с новейшей версией Python для Windows (не забудьте включить завершающую косую черту): https://www.python.org/downloads/windows/.
2. Щелкните на ссылке версии Python, которую хотите скачать. Чтобы скачать новейшую версию, щелкните на ссылке Latest Python 3 Release — 3.X.Y, где

3.X.Y — номер последней версии (например, 3.8.3). Код, приведенный в книге, был протестирован как на Python 3.6, так и на Python 3.8. Если вы захотите скачать более старую версию, то прокрутите страницу до раздела Stable Releases и найдите нужную версию.

3. Ссылка, на которой вы щелкнули на шаге 2, открывает страницу выбранного вами выпуска Python. В разделе Files щелкните на ссылке Windows x86-64 executable installer.
4. Ссылка из шага 3 открывает файл `.exe` на вашем компьютере. Это установочный файл; щелкните на нем дважды, чтобы открыть. Файл автоматически запускает процесс установки. Установите флажок Add Python 3.X to PATH, где X — номер версии для загруженной установочной программы (например, 8). Затем нажмите кнопку Install Now и выберите настройки по умолчанию.
5. Когда появится сообщение Setup was successful, нажмите кнопку Close, чтобы завершить процесс установки.

На вашем компьютере появилось новое приложение Python 3.X, где X — номер установленной версии Python 3. В поле поиска Windows введите `Python`. Когда приложение появится на экране, щелкните на нем, чтобы открыть консоль Python. Вводите на консоли команды Python, и они будут выполнены.

Установка Python в macOS

Чтобы установить Python в системе macOS, выполните следующие действия.

1. Откройте страницу с новейшей версией Python для macOS (не забудьте включить завершающую косую черту): https://www.python.org/downloads/mac-osx/.
2. Щелкните на ссылке версии Python, которую хотите скачать. Чтобы скачать новейшую версию, щелкните на ссылке Latest Python 3 Release — 3.X.Y, где 3.X.Y — номер последней версии (например, 3.8.3). Код, приведенный в книге, был протестирован как на Python 3.6, так и на Python 3.8. Если захотите скачать более старую версию, то прокрутите страницу до раздела Stable Releases и найдите нужную версию.
3. Ссылка, на которой вы щелкнули на шаге 2, открывает страницу выбранного вами выпуска Python. В разделе Files щелкните на ссылке macOS 64-bit installer.
4. Ссылка из шага 3 открывает файл `.pkg` на вашем компьютере. Это установочный файл; дважды щелкните на нем, чтобы открыть. Файл автоматически запускает процесс установки. Выберите настройки по умолчанию.
5. На вашем компьютере создается папка Python 3.X, где X — номер установленной версии Python. В этой папке дважды щелкните на значке IDLE. На экране

появляется приложение 3.X.Y Shell, где 3.X.Y — номер новейшей версии. Это консоль Python, из которой можно выполнять любые команды Python.

Установка Python в системе Linux

Чтобы установить Python в системе Linux, выполните следующие действия.

1. Определите, какой менеджер пакетов используется вашей версией Linux. Два типичных варианта — `yum` и `apt-get`.
2. Откройте консоль Linux (также называемую терминалом) и выполните следующие две команды:

```
> sudo apt-get update
> sudo apt-get install python3.8
```

Если вы используете `yum` или другой менеджер пакетов, то замените оба упоминания `apt-get` в этих двух строках именем `yum` или именем вашего менеджера пакетов. Аналогичным образом, если вы захотите установить более старую версию Python, замените `3.8` (номер последней версии на момент написания книги) другим номером версии (например, `3.6` — одной из версий, которые использовались при тестировании кода книги). Чтобы узнать номер новейшей версии Python, перейдите по адресу https://www.python.org/downloads/source/. Здесь вы найдете ссылку Latest Python 3 Release — 3.X.Y, где 3.X.Y — номер версии; используйте первые две цифры в приведенной выше команде установки.

3. Запустите Python, выполнив следующую команду с консоли Linux:

```
python3
```

Консоль Python открывается в консольном окне Linux. Здесь вы можете вводить команды Python.

Установка сторонних модулей

Часть кода, представленного в книге, зависит от модулей Python, которые не входят в базовое программное обеспечение Python, загруженное с официального сайта Python. Чтобы установить сторонние модули на компьютере, выполните инструкции, изложенные на http://automatetheboringstuff.com/2e/appendixa/.

Резюме

В ходе знакомства с алгоритмами мы посетим много разных мест и окунемся в различные исторические эпохи. Вы познакомитесь с открытиями, сделанными в Древнем Египте, Вавилоне, Афинах времен Перикла, Багдаде, средневековой Европе, Японии периода Эдо и Британской Индии, а также с выдающимися достижениями современности и ее ошеломляющими технологиями. Нам придется искать новые решения задач и преодолевать ограничения, которые на первый взгляд кажутся непреодолимыми. При этом мы установим связь не только первопроходцев древней науки, но и всех, кто пользуется компьютером или ловит бейсбольные мячи, с поколениями пользователей алгоритмов, а также их создателей, которые еще не родились, но в отдаленном будущем будут развивать оставленное нами наследие. Эта книга поможет вам сделать первые шаги в мир алгоритмов.

От издательства

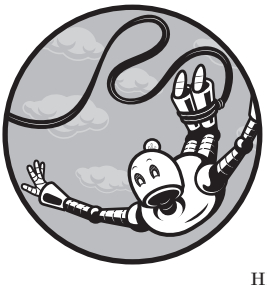
Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

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

1

Алгоритмы при решении задач



Поймать мяч — на редкость нетривиальное дело. В начале полета мяч может находиться настолько далеко, что будет казаться вам крошечным пятнышком на горизонте. Он может находиться в воздухе всего несколько коротких секунд, а то и меньше. На мяч воздействуют сопротивление воздуха, ветер и, конечно, сила тяготения, из-за чего он двигается по траектории, близкой к параболе. И все броски мяча совершаются с разной силой, под разными углами и в разных средах с разными условиями. Как же бейсболист, находящийся в 100 метрах от подающего, в момент удара узнает, куда нужно бежать, чтобы перехватить мяч, пока тот не коснулся земли?

Этот вопрос, называемый *задачей аутфилдера*[1], продолжает обсуждаться в научных журналах и в наши дни. Мы начинаем с задачи аутфилдера, поскольку она имеет два очень разных решения: аналитическое и алгоритмическое. Сравнение этих решений ярко демонстрирует, что такое алгоритм и чем он отличается от других подходов к решению задач. Кроме того, задача аутфилдера помогает наглядно представить область, которая в основном абстрактна, — наверняка вам уже доводилось что-нибудь бросать и ловить, и данный опыт поможет вам понять теорию, лежащую в основе этой практики.

---

[1] Аутфилдер — общий термин, которым в бейсболе обозначается один из трех игроков, занимающих оборонительную позицию на внешнем поле. — *Примеч. пер.*

Чтобы действительно понять, как человек точно определяет, куда упадет мяч, будет полезно понять, как это делает машина. Начнем с аналитического решения задачи аутфилдера. Оно обладает математической точностью, компьютер легко найдет его за долю секунды, и это решение в том или ином виде обычно объясняют во вводном курсе физики. Оно позволит подвижному роботу выполнять функции аутфилдера в бейсбольной команде.

Однако человек не умеет легко решать аналитические уравнения в уме — по крайней мере, не так быстро, как это делает компьютер. Для человеческого мозга лучше подойдет алгоритмическое решение. На его примере мы исследуем, что такое алгоритм и какими преимуществами он обладает по сравнению с другими методами решения задач. Более того, алгоритмическое решение покажет, что алгоритмы естественны для человеческих мыслительных процессов и далеко не всегда выглядят устрашающе. На примере задачи аутфилдера будет представлен новый способ решения задач: алгоритмический подход.

Аналитический подход

Чтобы решить задачу с помощью аналитического метода, необходимо вернуться на несколько столетий назад к ранней модели движения.

Модель Галилея

Уравнения, чаще всего используемые для моделирования перемещения мяча, существуют со времен Галилея. Этот ученый несколько веков назад предложил полиномиальные формулы, связывающие ускорение, скорость и расстояние. Если не учитывать ветер и сопротивление воздуха и предположить, что мяч начинает движение на уровне земли, то, согласно модели Галилея, горизонтальная позиция брошенного мяча в момент времени t определяется формулой:

$$x = v_1 t,$$

где v_1 — начальная скорость мяча по оси x (по горизонтали). Кроме того, высота брошенного мяча (y) в момент времени t по Галилею вычисляется по формуле

$$y = v_2 t + \frac{at^2}{2},$$

где v_2 — начальная скорость мяча по оси y (по вертикали); a — постоянное ускорение свободного падения под воздействием силы тяжести (в метрической системе равно

приблизительно –9,81). Подставив первое уравнение во второе, мы находим, что высота брошенного мяча (y) связана с горизонтальной позицией мяча (x) следующей зависимостью:

$$y = \frac{v_2}{v_1}x + \frac{ax^2}{2xv_1^2}.$$

Формулы Галилея можно использовать для моделирования траектории гипотетического мяча на языке Python; соответствующая функция приведена в листинге 1.1. Полиномиальная формула в данном листинге предназначена для мяча, начальная горизонтальная скорость которого составляет приблизительно 0,99 м/с, а начальная вертикальная скорость — около 9,9 м/с. Вы можете поэкспериментировать с другими значениями v_1 и v_2, чтобы смоделировать бросок с любыми интересующими вас параметрами.

Листинг 1.1. Функция вычисления траектории мяча

```
def ball_trajectory(x):
    location = 10*x - 5*(x**2)
    return(location)
```

Построив график функции из листинга 1.1 на языке Python, мы увидим, как приблизительно должна выглядеть траектория мяча (без учета сопротивления воздуха и других незначительных факторов). Средства построения графиков импортируются из модуля `matplotlib` в первой строке (листинг 1.2). Модуль `matplotlib` — один из многочисленных сторонних модулей, которые будут импортироваться в коде, приводимом в книге. Прежде чем использовать сторонний модуль, его необходимо установить. Инструкции по установке `matplotlib` и любых других сторонних модулей доступны на http://automatetheboringstuff.com/2e/appendixa/.

Листинг 1.2. Построение траектории гипотетического мяча между моментом броска (x = 0) и его соприкосновением с землей (x = 2)

```
import matplotlib.pyplot as plt
xs = [x/100 for x in list(range(201))]
ys = [ball_trajectory(x) for x in xs]
plt.plot(xs,ys)
plt.title('The Trajectory of a Thrown Ball')
plt.xlabel('Horizontal Position of Ball')
plt.ylabel('Vertical Position of Ball')
plt.axhline(y = 0)
plt.show()
```

На выходе (рис. 1.1) вы получаете красивый график с той траекторией, по которой наш гипотетический мяч должен перемещаться в пространстве. Красивая криволинейная траектория будет похожей для всех движущихся брошенных тел, находящихся под воздействием силы тяготения; писатель Томас Пинчон (Thomas Pynchon) поэтично назвал ее *радугой тяготения*.

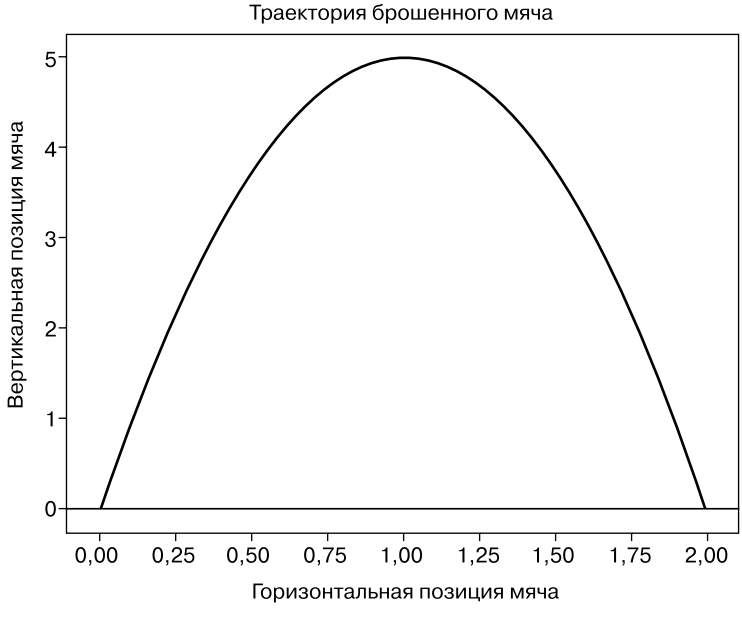


Рис. 1.1. Траектория гипотетического брошенного мяча

Не все мячи будут точно следовать данной траектории, но это один из возможных путей, по которым может перемещаться мяч. Он начинает двигаться в точке 0. Летит вверх, а затем начинает опускаться, перемещаясь от левого края видимой области к правому, как мы не раз видели в жизни.

Стратегия решения для x

Теперь при наличии формулы для вычисления позиции мяча уравнение можно решить для любой интересующей вас точки: когда мяч достигнет наивысшей точки или когда снова опустится до уровня земли — то, что необходимо знать игроку, чтобы поймать его. Студенты, изучающие физику, учатся находить такие решения, и если вы хотите научить робота выполнять функции аутфилдера, то вполне естественно научить его и этим уравнениям. Метод определения итогового положения

мяча сводится к тому, что вы берете функцию `ball_trajectory()`, с которой мы начали, и приравниваете ее к 0:

$$0 = 10x - 5x^2.$$

Далее уравнение решается для x по формуле квадратного уравнения, которую все мы узнали еще в школе:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

В данном случае мы определяем, что решениями являются $x = 0$ и $x = 2$. Первое решение, $x = 0$, относится к начальной точке движения, где мяч был приведен в движение броском питчера или ударом бьющего. Второе решение $x = 2$ относится к точке, в которой мяч снова соприкасается с землей после полета.

Использованная нами стратегия была относительно простой. Назовем ее *стратегией решения для x*. Вы записываете уравнение, описывающее ситуацию, а затем решаете его для интересующей вас переменной. Стратегия решения для x очень часто встречается в естественных науках — как в средней школе, так и в колледже. Студентам предлагается решать уравнения для определения ожидаемой точки падения, идеального уровня экономического производства, пропорции химиката, в которой он должен использоваться в эксперименте, и множества других показателей.

Стратегия решения для x чрезвычайно эффективна. Например, если армия наблюдает, как противник выпускает ракету, то может быстро ввести формулу Галилея в свои вычислительные устройства, почти моментально определить, куда попадет ракета, и вовремя отойти или сбить ее. Это можно легко сделать на обычном ноутбуке, на котором работает Python. Если бы робот играл в защите на бейсбольном поле, то мог бы проделать то же самое и перехватить мяч без малейших усилий.

Стратегия решения для x в данном случае легко реализуется, поскольку мы уже знаем уравнение, которое необходимо решить, и способ его решения. Как упоминалось ранее, уравнение для моделирования траектории брошенного мяча открыл Галилей. Создателем формулы квадратного уравнения был великий Мухаммад ибн Муса аль-Хорезми — первый, кто предложил полное общее решение квадратного уравнения.

Аль-Хорезми — всесторонне образованный ученый, который жил в IX веке и внес свой вклад в астрономию, картографию и тригонометрию. В частности, он ввел термин *«алгебра»* и научную дисциплину, которая им обозначается. Аль-Хорезми — одна из важнейших фигур, благодаря которым эта книга стала возможной. Мы живем

после таких гигантов, как Галилей и аль-Хорезми, поэтому нам не нужно мучиться с выводом уравнений — достаточно просто подставить в них значения и использовать соответствующим образом.

Внутренний физик

Используя уравнения Галилея и аль-Хорезми в сочетании со стратегией решения для *x*, сложная машина может поймать мяч или сбить ракету. Но можно достаточно уверенно предположить, что большинство бейсболистов не начинают писать уравнения при виде летящего мяча. Надежные источники сообщают, что согласно учебным программам для профессиональных бейсболистов игроки в основном бегают по полю и играют, а не стоят за доской в попытках вывести уравнения Навье — Стокса. Разгадка секрета того, где упадет мяч, не дает четкого ответа на задачу аутфилдера — то есть как *человек* инстинктивно определяет, где упадет мяч, без обращения к компьютерной программе...

А может быть, и дает. Самое примитивное из возможных решений задачи аутфилдера — предположить, что если компьютеры решают квадратные уравнения для определения того, где упадает мяч, то и человек делает то же самое. Назовем это решение *теорией внутреннего физика*. Согласно этой теории, «биологический компьютер» нашего мозга способен формулировать и решать квадратные уравнения или рисовать графики и экстраполировать их, причем все это происходит гораздо глубже уровня нашего сознания. Иначе говоря, у каждого из нас где-то глубоко в мозгу обитает «внутренний физик», который может за секунды находить точные уравнения сложных математических задач. Найденное решение передается нашим мускулам, которые приводят в движение наши руки и туловище. Возможно, подсознание позволит сделать это даже тем, кто никогда не посещал уроки физики и не решал уравнения.

У теории внутреннего физика есть свои сторонники. Например, известный математик Кейт Девлин (Keith Devlin) в 2006 году опубликовал книгу, которая называлась «Математический инстинкт: почему вы гениальный математик (наряду с омарами, птицами, кошками и собаками)». На обложке книги изображена собака, которая в прыжке ловит фрисби; стрелками обозначены векторы траектории диска и собаки. Имеется в виду, что собака способна выполнить все сложные вычисления, необходимые для того, чтобы векторы встретились.

Несомненная способность собак ловить фрисби, как и способность людей ловить бейсбольные мячи, вроде бы говорит в пользу теории внутреннего физика. Подсознательное — таинственная и впечатляющая область, глуби́ны которой нам еще только предстоит изучить. Так почему бы ему время от времени не решать

уравнения уровня средней школы? Что еще важнее, теорию внутреннего физика сложно опровергнуть, поскольку трудно предложить альтернативу: если собаки не способны решать дифференциальные уравнения в частных производных, чтобы поймать диск, то как они его ловят? Они прыгают высоко в воздух и без малейших усилий хватают фрисби зубами. Если собаки не решают в мозгу какую-то задачу из области физики, то как они (и мы) узнают, как точно перехватить летающий объект?

Еще в 1967 году ни у кого не было хорошего ответа на данный вопрос. В том году инженер Ванневар Буш (Vannevar Bush) написал книгу, в которой описывал научные аспекты бейсбола (так, как понимал их). Однако автор не смог никак объяснить, как аутфилдеры узнают, куда же им нужно бежать, чтобы перехватить летящий мяч. К счастью, физик Севилл Чепмен (Seville Chapman) прочитал книгу Буша и настолько вдохновился ею, что уже в следующем году выдвинул собственную теорию.

Алгоритмический подход

Чепмена как ученого не устраивало, что все списывали на подсознание, и он захотел получить более конкретное объяснение способностей бейсболистов. И вот что он обнаружил.

Как думать шеей

Чепмен начал решать задачу аутфилдера с анализа информации, необходимой для того, чтобы поймать мяч. Хотя человеку может быть трудно оценить точную скорость мяча или траекторию параболической дуги, Чепмен подумал, что нам проще наблюдать за углами. Если кто-то бросает или пинает мяч от земли, а она ровная и плоская, то игрок увидит, как мяч начинает приближаться к уровню его глаз. Представьте угол, образованный двумя линиями: землей и линией видимости мяча игроком. В момент, когда бьющий ударяет по мячу, данный угол составляет (примерно) 0 градусов. После непродолжительного полета мяч окажется выше земли, так что угол между землей и линией видимости мяча игроком увеличится. Даже если игрок не изучал геометрию, он «чувствует» этот угол — например, ощущая, насколько ему приходится задрать голову, чтобы увидеть мяч.

Если предположить, что игрок стоит там, где в конечном итоге упадет мяч (в точке $x = 2$), то можно получить представление о том, как изменяется угол линии видимости мяча, построив график линии видимости на ранней фазе траектории

мяча. Следующий фрагмент кода создает отрезок для графика, построенного в листинге 1.2; предполагается, что он выполняется в том же сеансе Python. Отрезок представляет линию между глазами игрока и мячом после того, как мяч пролетел 0,1 метра по горизонтали.

```
xs2 = [0.1,2]
ys2 = [ball_trajectory(0.1),0]
```

Эту линию видимости можно нанести на график вместе с другими линиями, чтобы увидеть, как угол продолжает расти на протяжении траектории мяча. Следующие строки кода добавляют другие отрезки на график, построенный в листинге 1.2. Отрезки представляют линию между глазами игрока и мячом для еще двух точек на пути мяча: точек, в которых мяч переместился на 0,1, 0,2 и 0,3 метра по горизонтали. После создания всех этих отрезков они наносятся на график одновременно.

```
xs3 = [0.2,2]
ys3 = [ball_trajectory(0.2),0]
xs4 = [0.3,2]
ys4 = [ball_trajectory(0.3),0]
plt.title('The Trajectory of a Thrown Ball - with Lines of Sight')
plt.xlabel('Horizontal Position of Ball')
plt.ylabel('Vertical Position of Ball')
plt.plot(xs,ys,xs2,ys2,xs3,ys3,xs4,ys4)
plt.show()
```

На полученном графике мы видим несколько линий видимости, которые образуют непрерывно увеличивающиеся углы с уровнем земли (рис. 1.2).

В процессе полета угол линии видимости игрока продолжает расти, и игрок вынужден закидывать голову назад, пока не поймает мяч. Обозначим угол между землей и линией видимости θ. Будем считать, что игрок стоит в точке падения мяча ($x = 2$). Вспомните из школьного курса геометрии, что тангенс угла прямоугольного треугольника равен отношению противолежащего катета к прилежащему. В данном случае тангенс θ равен отношению высоты мяча к его горизонтальному расстоянию от игрока. Следующий фрагмент Python наносит на график стороны, отношение которых определяет тангенс:

```
xs5 = [0.3,0.3]
ys5 = [0,ball_trajectory(0.3)]
xs6 = [0.3,2]
ys6 = [0,0]
plt.title('The Trajectory of a Thrown Ball - Tangent Calculation')
```

```
plt.xlabel('Horizontal Position of Ball')
plt.ylabel('Vertical Position of Ball')
plt.plot(xs,ys,xs4,ys4,xs5,ys5,xs6,ys6)
plt.text(0.31,ball_trajectory(0.3)/2,'A',fontsize = 16)
plt.text((0.3 + 2)/2,0.05,'B',fontsize = 16)
plt.show()
```

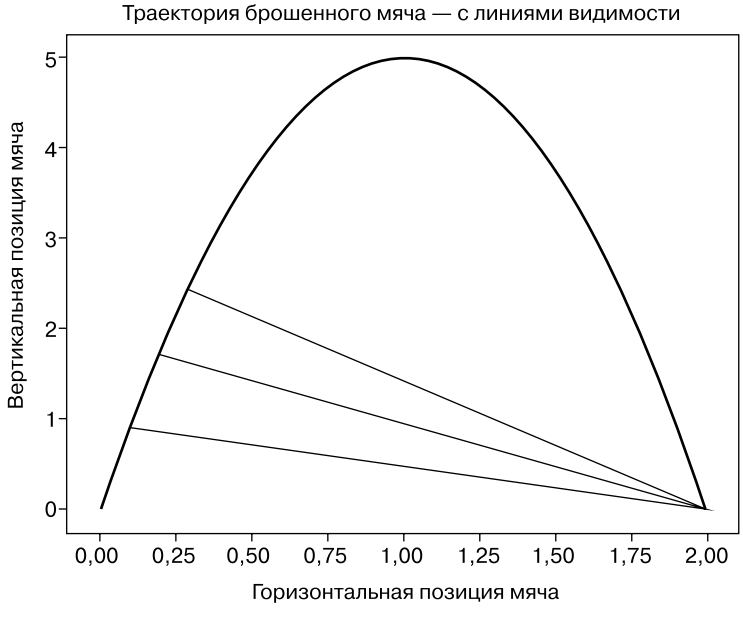


Рис. 1.2. Траектория гипотетического брошенного мяча с отрезками, представляющими линии видимости мяча

Полученный график изображен на рис. 1.3.

Тангенс вычисляется отношением длины стороны A к длине стороны B. Высота A вычисляется по формуле $10x - 5x^2$, а длина B — по формуле $2 - x$. Таким образом, следующее уравнение неявно описывает угол θ для каждой точки полета:

$$\tan(\theta) = \frac{10x - 5x^2}{2 - x} = 5x.$$

Ситуация в целом достаточно сложна: где-то далеко мяч приходит в движение и быстро летит по параболической кривой, конец которой трудно немедленно спрогнозировать. Но в этой сложной ситуации Чепмен нашел простое отношение:

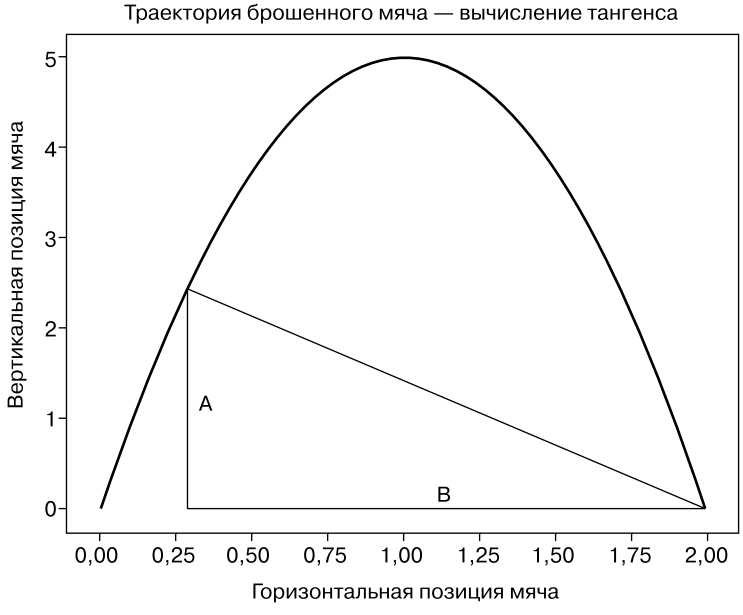


Рис. 1.3. Траектория гипотетического брошенного мяча с отрезками, представляющими линии видимости мяча, и отрезками A и B, отношение длин которых дает интересующий нас тангенс

когда игрок стоит в правильном месте, тангенс θ увеличивается с постоянной скоростью. Вся суть открытия Чепмена заключалась в том, что тангенс θ, угла мяча относительно земли, растет линейно со временем. После того как Чепмен обнаружил эту простую связь, он смог создать элегантное алгоритмическое решение.

Решение опирается на тот факт, что если какая-то характеристика — в данном случае тангенс θ — растет с постоянной скоростью, то ее ускорение равно нулю. Таким образом, стоя точно в том месте, куда падает мяч, вы наблюдаете угол, тангенс которого обладает нулевым ускорением. Если вы стоите слишком близко к исходной позиции мяча, то наблюдаете положительное ускорение, а если слишком далеко — то отрицательное. (При желании вы можете самостоятельно проверить муторные вычисления, на которых основаны эти утверждения.) А это означает, что игрок может понять, в какую сторону ему следует бежать, по тому, насколько равномерно ему приходится задирать голову при наблюдении за подъемом мяча — так сказать, «думать шеей».

Применение алгоритма Чепмена

Не у всех роботов есть шея, так что метод «думать шеей» может оказаться бесполезным для робота-бейсболиста. Не забудьте, что роботы способны моментально решать квадратные уравнения, чтобы понять, где следует ловить мяч, не беспокоясь о скорости роста тангенса θ. Но для людей метод Чепмена может быть в высшей степени полезным. Чтобы оказаться в точке падения мяча, игрок-человек может следовать относительно простому процессу, описанному ниже.

1. Определить ускорение тангенса угла между землей и линией видимости мяча.
2. Если ускорение положительно, то сделать шаг назад.
3. Если ускорение отрицательно, то сделать шаг вперед.
4. Повторять шаги 1–3, пока мяч не окажется прямо перед вами.
5. Поймать его.

Одно серьезное возражение против метода Чепмена из пяти шагов заключается в том, что руководствующиеся им игроки должны на ходу вычислять тангенсы углов, а это означает, что теория внутреннего физика заменяется теорией внутреннего геометра, согласно которой бейсболисты могут мгновенно — и подсознательно — вычислять тангенсы.

Один потенциальный аргумент против этого возражения заключается в том, что для многих углов $\tan(\theta)$ приблизительно равен θ, поэтому вместо отслеживания ускорения тангенса игроки могут наблюдать за ускорением самого угла. Если ускорение угла можно оценить по воспринимаемому ускорению шейных мышц, которые растягиваются при движении шеи для наблюдения за мячом, и если угол может служить разумным приближением для тангенса, то никакие предположения о великих подсознательных математических или геометрических способностях игроков не нужны — важен только физический навык точной реакции на трудноуловимые ощущения органов чувств.

Когда оценка ускорения остается единственной сложной частью процесса, мы получаем потенциальное решение задачи аутфилдера, которое обладает намного большей психологической достоверностью, чем теория внутреннего физика с подсознательной экстраполяцией парабол. Конечно, психологическая привлекательность решения не означает, что оно может использоваться только людьми. Робот-игрок тоже может быть запрограммирован на следование методу Чепмена и даже может лучше справляться с ловлей мяча, поскольку метод Чепмена позволяет динамически реагировать на изменения из-за ветра или рикошета.

Помимо психологической достоверности, у процесса из пяти шагов, предложенного Чепменом, есть еще одна важнейшая особенность: он не полагается на стратегию решения для *x* и вообще какое-либо явное уравнение. Вместо этого он предлагает последовательные итерации с простыми наблюдениями и небольшие, постепенные шаги для достижения четко определенной цели. Другими словами, процесс, следующий из теории Чепмена, является алгоритмом.

Решение задач с применением алгоритмов

Слово *«алгоритм»* также связано с именем великого аль-Хорезми, упоминавшегося ранее. Определить его не так просто — не в последнюю очередь из-за того, что общепринятое определение изменялось со временем. Говоря простым языком, алгоритм представляет собой набор инструкций, который дает четко определенный результат. Это достаточно широкое определение; как было показано во введении, налоговые декларации и рецепты парфе тоже можно с полным основанием считать алгоритмами.

Пожалуй, процесс ловли мяча по Чепмену — или алгоритм Чепмена, как его правильнее называть, — больше похож на алгоритм, чем рецепт парфе, поскольку содержит циклическую структуру, в которой небольшие шаги выполняются многократно до достижения определенного условия. Это типичная алгоритмическая структура, которая неоднократно встречается в данной книге.

Чепмен предложил алгоритмическое решение для задачи аутфилдера, поскольку решение для *x* было неприемлемо (игроки часто не знают необходимые уравнения). В общем случае алгоритмы оказываются наиболее полезными тогда, когда стратегия решения для *x* не дает результата. Иногда нужные уравнения неизвестны, но чаще уравнения, которое бы полностью описывало ситуацию, просто не существует, его невозможно решить, или мы сталкиваемся с ограничениями по времени/памяти.

Бытует мнение, что алгоритмы сложны, непостижимы, загадочны, имеют чисто математическую природу и недоступны пониманию, если вы не изучали их много лет. Современная система образования устроена так, что детей начинают учить стратегии решения для *x* как можно раньше, а алгоритмы в явном виде преподают только на уровне колледжа или магистратуры (если вообще преподают). У многих учащихся на освоение стратегии решения для *x* уходят годы, и этот метод всегда кажется им чем-то искусственным. Люди с подобным опытом уверены, что алгоритмы будут такими же искусственными и еще более трудно понимаемыми, поскольку они являются «продвинутыми».

Однако я вынес из алгоритма Чепмена важный урок: наше обучение поставлено с ног на голову. Во время перерывов в занятиях ученики узнают и совершенствуют свое владение десятками алгоритмов для ловли, бросков, бега и т. д. Вероятно, есть и другие, намного более сложные и еще не изученные алгоритмы, управляющие социальными взаимодействиями: разговоры, стремление добиться положения в компании, сплетни, формирование альянсов и дружеских отношений. А когда перерыв заканчивается и начинается лекция по математике, мы просим учащихся абстрагироваться от реального мира исследования алгоритмов и заставляем изучать неестественный механистический процесс решения для x — процесс, который не является естественной частью развития личности и даже не является самым эффективным методом решения аналитических задач. Только достаточно продвинувшись в области математики и информатики, ученики возвращаются в естественный мир алгоритмов и эффективных процессов, которые бессознательно и с удовольствием практиковали на перерывах.

Эта книга была задумана как интеллектуальный перерыв для любознательных — перерыв в том смысле, в котором его понимают школьники: конец тяжелой монотонной работы, начало действительно важных дел и продолжение занятий с друзьями в хорошем настроении. Если алгоритмы вызывают у вас трепет, то напомните себе, что мы, люди, алгоритмичны по своей природе, и если вы можете поймать мяч или испечь пирог — значит, можете освоить алгоритм.

В книге мы исследуем много разных алгоритмов. Одни сортируют списки или обрабатывают числа. Другие делают возможной обработку естественного языка и искусственный интеллект. Учтите, что алгоритмы не растут на деревьях. Каждый алгоритм до того, как пошел в массы и был представлен для широкого обозрения в этой книге, был кем-то изобретен или найден — как это произошло с Чепменом, который проснулся в мире, где его алгоритм не существовал, а в конце дня лег спать в изменившемся мире. Я постараюсь сделать так, чтобы вы попытались поставить себя на место этих героических изобретателей. Иначе говоря, я рекомендую рассматривать алгоритмы не только как инструменты, но и как трудноразрешимую проблему, которая была успешно решена. Полная карта мира алгоритмов еще не построена — остается еще немало неизученных областей, и я искренне надеюсь, что вы можете стать одним из участников процесса их изучения.

Резюме

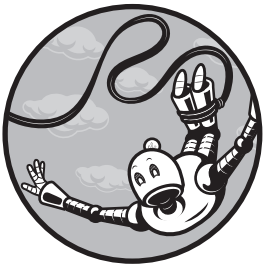
В этой главе были представлены два подхода к решению задач: аналитический и алгоритмический. Рассматривая два способа решения задачи аутфилдера, мы исследовали различия этих подходов и в конечном итоге пришли к алгоритму

Чепмена. Ученый нашел простую закономерность в сложной ситуации (постоянное ускорение тангенса θ) и использовал ее для разработки концепции итеративного процесса, который получает только один простой ввод (поворот головы за мячом) и приводит к определенной цели (пойманный мяч). Если вы хотите разрабатывать и использовать алгоритмы в своей работе, то можете попытаться встать на путь Чепмена.

В следующей главе мы рассмотрим некоторые примеры алгоритмов в истории. Эти примеры должны углубить ваше понимание алгоритмов; в частности, вы узнаете, что они собой представляют и как работают. Речь пойдет об алгоритмах из истории Древнего Египта, Древней Греции и императорской Японии. Каждый новый изучаемый вами алгоритм можно добавить в арсенал алгоритмов, которыми вы сможете пользоваться в своей работе, пока со временем не дойдете до точки, когда сможете разрабатывать и совершенствовать собственные алгоритмы.

2

Алгоритмы в истории



Алгоритмы часто ассоциируются с компьютерами. И в этом есть резон; компьютерные операционные системы могут использовать множество алгоритмов высокой сложности, и программирование хорошо подходит для точной реализации всех видов алгоритмов. Однако алгоритмы имеют более фундаментальную природу, чем компьютерная архитектура, на которой они реализуются. Как упоминалось в главе 1, слово «*алгоритм*» появилось более тысячи лет назад, однако алгоритмы описывались в древних рукописях еще задолго до этого. Даже вне памятников письменности существуют многочисленные данные, свидетельствующие о применении сложных алгоритмов в Древнем мире — например, технологии строительства.

В этой главе представлены несколько алгоритмов древнего происхождения. В них проявляется выдающаяся изобретательность и оригинальность мышления, особенно если учесть, что они создавались и проверялись без помощи компьютеров. Начнем с обсуждения *русского крестьянского умножения* — метода арифметических вычислений, который, несмотря на свое название, может происходить из Египта и не иметь никакого отношения к крестьянам. Затем рассмотрим алгоритм Евклида — важный «классический» алгоритм для нахождения наибольшего общего делителя. Напоследок рассмотрим изобретенный в Японии алгоритм для построения волшебных квадратов.

Русское крестьянское умножение

Изучение таблицы умножения многим запомнилось как особенно трудный этап образования. Дети спрашивают своих родителей, почему так важно учить таблицу умножения, и родители обычно отвечают, что без этого нельзя умножать. Как же они ошибаются! Существует *русское крестьянское умножение* (Russian Peasant Multiplication, RPM) — метод, позволяющий перемножать большие числа, обходясь без знания большей части таблицы умножения.

Происхождение RPM остается неясным. Древнеегипетский свиток, называемый *папирусом Ринда*, содержит разновидность этого алгоритма. Некоторые историки предложили гипотезы (большей частью неубедительные) о том, как метод мог перейти от древнеегипетских ученых к крестьянам необъятной российской глубинки. Как бы то ни было, алгоритм RPM весьма интересен.

RPM вручную

Представьте, что хотите умножить 89 на 18. RPM работает так: сначала нарисуйте два расположенных рядом друг с другом столбца. Первый называется столбцом *деления*, сначала в нем находится число 89. Второй называется столбцом *умножения*, и в исходном состоянии в нем находится число 18 (табл. 2.1).

Таблица 2.1. Таблица деления/умножения, часть 1

Столбец деления	Столбец умножения
89	18

Начнем с заполнения столбца деления. Для каждой его строки берем предыдущее значение и делим его на 2, остаток при этом игнорируется. Например, при делении 89 на 2 мы получаем 44 с остатком 1, поэтому во второй строке столбца деления записывается число 44 (табл. 2.2).

Таблица 2.2. Таблица деления/умножения, часть 2

Столбец деления	Столбец умножения
89	18
44	

Деление на 2 продолжается, пока не будет получен результат 1. При этом каждый раз остаток отбрасывается, а результат записывается в следующую строку. Половина от 44 равна 22, половина от 22 равна 11, половина от 11 (с потерей остатка) равна 5, затем 2, затем 1. Записав эти числа в столбец деления, мы получаем табл. 2.3.

Таблица 2.3. Таблица деления/умножения, часть 3

Столбец деления	Столбец умножения
89	18
44	
22	
11	
5	
2	
1	

Столбец деления готов. Каждый элемент столбца умножения равен удвоенному предыдущему элементу. Так как 18 × 2 = 36, вторая строка столбца умножения содержит 36 (табл. 2.4).

Таблица 2.4. Таблица деления/умножения, часть 4

Столбец деления	Столбец умножения
89	18
44	36
22	
11	
5	
2	
1	

Далее мы продолжаем добавлять элементы в столбец умножения по тому же правилу: предыдущее значение удваивается. Это продолжается до тех пор, пока столбец умножения не сравняется по количеству элементов со столбцом деления (табл. 2.5).

Таблица 2.5. Таблица деления/умножения, часть 5

Столбец деления	Столбец умножения
89	18
44	36
22	72
11	144
5	288
2	576
1	1152

На следующем шаге из таблицы удаляются все строки, у которых столбец деления содержит четное число. Результат показан в табл. 2.6.

Таблица 2.6. Таблица деления/умножения, часть 6

Столбец деления	Столбец умножения
89	18
11	144
5	288
1	1152

Остается сложить все оставшиеся числа в столбце умножения. Результат равен 18 + 144 + 288 + 1152 = 1602. Правильность ответа можно проверить на калькуляторе: 89 × 18 = 1602. Умножение было реализовано с помощью операций деления надвое, удвоения и сложения, и нам не пришлось запоминать большую часть скучной таблицы умножения, которую так не любят дети.

Чтобы понять, почему работает этот метод, попробуйте переписать столбец умножения в виде множителей 18 — умножаемого числа (табл. 2.7).

Таблица 2.7. Таблица деления/умножения, часть 7

Столбец деления	Столбец умножения
89	18×1
44	18×2
22	18×4
11	18×8
5	18×16
2	18×32
1	18×64

В столбце умножения следует серия множителей 1, 2, 4, 8 и т. д. до 64. Все эти числа являются степенями 2, и их также можно записать в виде 2^0, 2^1, 2^2 и т. д. Когда мы вычисляем итоговую сумму (складываем строки столбца умножения, у которых столбец деления содержит нечетное значение), в действительности вычисляется следующая сумма:

$$18 \times 2^0 + 18 \times 2^3 + 18 \times 2^4 + 18 \times 2^6 = 18 \times (2^0 + 2^3 + 2^4 + 2^6) = 18 \times 89.$$

Работа RPM зависит от следующего факта:

$$(2^0 + 2^3 + 2^4 + 2^6) = 89.$$

Внимательно присмотревшись к столбцу деления, можно понять, почему данное уравнение истинно. Этот столбец тоже можно записать в степенях 2 (табл. 2.8).

Таблица 2.8. Таблица деления/умножения, часть 8

Столбец деления	Столбец умножения
$(2^5 + 2^3 + 2^2) \times 2^1 + 2^0 = 2^6 + 2^4 + 2^3 + 2^0$	18×2^0
$(2^5 + 2^3 + 2^2) \times 2^1 + 2^0 = 2^6 + 2^4 + 2^3 + 2^0$	18×2^1
$(2^3 + 2^1 + 2^0) \times 2^1 = 2^4 + 2^2 + 2^1$	18×2^2
$(2^2 + 2^0) \times 2^1 + 2^0 = 2^3 + 2^1 + 2^0$	18×2^3
$2^1 \times 2^1 + 2^0 = 2^2 + 2^0$	18×2^4
$2^0 \times 2^1 = 2^1$	18×2^5
2^0	18×2^6

При этом проще начать с наименьшего числа и двигаться снизу вверх. Стоит напомнить, что $2^0 = 1$, а $2^1 = 2$. В каждой строке значение умножается на 2^1, а в строках, в которых делимое число нечетно, также добавляется 2^0. При продвижении по строкам выражение начинает все больше напоминать наше уравнение. К моменту достижения верхней строки таблицы мы получаем выражение, которое упрощается в точности до $2^6 + 2^4 + 2^3 + 2^0$.

Если мы пронумеруем строки столбца деления (верхняя строка обозначается как строка 0, затем 1, 2 и вплоть до нижней строки 6), то увидим, что нечетные значения в столбце деления содержатся в строках 0, 3, 4 и 6. Теперь заметим важнейшую закономерность: номера этих строк в точности совпадают с показателями степеней в найденном нами выражении для 89: $2^6 + 2^4 + 2^3 + 2^0$. И это совпадение не случайно; способ построения столбца деления означает, что нечетные значения всегда находятся в строках, номера которых равны показателям степени в сумме степеней 2, равной нашему исходному числу. Когда мы вычисляем сумму элементов столбца умножения с этими индексами, мы фактически суммируем произведения 18 на степени 2, дающие в сумме 89, поэтому результат будет равен 89×18.

Почему же эта схема работает? В действительности RPM является алгоритмом внутри алгоритма. Сам столбец деления может считаться реализацией алгоритма, который находит сумму степеней 2, равную числу в первой ячейке столбца. Сумма степеней 2 также называется *двоичным разложением* числа 89. Двоичная система представляет собой альтернативную схему записи чисел с использованием только 0 и 1; она стала играть особенно важную роль в последние десятилетия, поскольку компьютеры хранят информацию в двоичном виде. В двоичной записи число 89 записывается в виде 1011001, с единицами в нулевой, третьей, четвертой и шестой позиции (справа налево); номера позиций соответствуют номерам нечетных строк столбца деления, а также степеням нашего уравнения. 1 и 0 в двоичном представлении можно рассматривать как коэффициенты в сумме степеней 2. Например, двоичное число 100 интерпретируется следующим образом:

$$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

или 4 в обычной (десятичной) записи. Двоичное число 1001 интерпретируется так:

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

или 9 в обычной записи. После выполнения мини-алгоритма для получения двоичного разложения 89 можно легко выполнить полный алгоритм и завершить процесс умножения.

Реализация RPM на Python

Реализация RPM на Python получается относительно простой. Допустим, вы хотите умножить два числа; назовем их n_1 и n_2. Для начала откроем сценарий Python и определим эти переменные:

```
n1 = 89
n2 = 18
```

На следующем шаге начнем строить столбец деления. Как упоминалось выше, он начинается с одного из перемножаемых чисел:

```
halving = [n1]
```

Следующий элемент равен `halving[0]/2` с игнорированием остатка. В Python для округления можно воспользоваться функцией `math.floor()`. Функция просто находит ближайшее целое число, которое меньше заданного. Например, вторая строка столбца деления вычисляется так:

```
import math
print(math.floor(halving[0]/2))
```

Выполнив этот код в Python, вы увидите, что результат равен 44.

Программа перебирает все строки столбца деления, при каждой итерации цикла находит следующее значение в данном столбце и останавливается при достижении 1:

```
while(min(halving) > 1):
    halving.append(math.floor(min(halving)/2))
```

В цикле метод `append()` используется для конкатенации. При каждой итерации цикла `while` вектор деления объединяется с половиной его последнего значения, при этом функция `math.floor()` используется для игнорирования остатка.

Со столбцом умножения делаем то же самое: мы начинаем с 18 и запускаем цикл. При каждой итерации цикла в столбец умножения добавляется удвоенное последнее значение. Цикл останавливается, когда длина этого столбца достигнет длины столбца деления:

```
doubling = [n2]
while(len(doubling) < len(halving)):
    doubling.append(max(doubling) * 2)
```

Наконец, эти два столбца помещаются в кадр данных `half_double`:

```
import pandas as pd
half_double = pd.DataFrame(zip(halving,doubling))
```

Здесь импортируется модуль Python `pandas`. Он упрощает работу с таблицами. В данном случае используется команда `zip`, которая соединяет `halving` с `doubling` подобно тому, как застежка-«молния» соединяет две полы куртки. Два набора чисел `halving` и `doubling` создаются как независимые списки, а после соединения и преобразования в кадр данных `pandas` сохраняются в таблице в виде двух параллельных столбцов, как показано выше в табл. 2.5. Поскольку столбцы выровнены и соединены, мы можем обратиться к любой строке табл. 2.5 (например, третьей) и получить всю строку данных, включающую элементы из `halving` и `doubling` (2 и 72). Возможность обращаться к этим строкам и работать с ними позволяет легко удалить ненужные строки, как было сделано с табл. 2.5 для преобразования ее в табл. 2.6.

Теперь необходимо удалить строки с четными значениями в столбце деления. Для проверки четности можно воспользоваться оператором `%` языка Python, возвращающим остаток от деления. Если число `x` нечетно, то `x%2` будет равно 1. Следующая строка оставляет в таблице только те строки, у которых значение в столбце деления является нечетным:

```
half_double = half_double.loc[half_double[0]%2 == 1,:]
```

В данном случае для отбора только интересующих нас строк используется функциональность `loc` модуля `pandas`. При использовании `loc` отбираемые строки и столбцы заключаются в квадратные скобки (`[]`). В них нужные строки и столбцы перечисляются через запятую: `[`*`строка, столбец`*`]`. Например, если вам нужна строка с индексом 4 и столбец с индексом 1, то можно прибегнуть к записи `half_double.loc[4,1]`. При этом ваши возможности не ограничиваются простым указанием индексов. Можно записать логический шаблон для отбора нужных строк: нас интересуют все строки, где `halving` содержит нечетное значение. В нашей логике столбец `halving` обозначается `half_double[0]`, то есть столбец с индексом 0. Нечетность определяется условием `%2 == 1`. Наконец, чтобы указать, что нам нужны все столбцы, после запятой ставится двоеточие — это сокращение означает, что нам нужны все столбцы.

Остается вычислить сумму оставшихся элементов `doubling`:

```
answer = sum(half_double.loc[:,1])
```

Здесь снова используется `loc`. Квадратные скобки указывают, что нам нужны все строки, для чего снова применяется сокращение с двоеточием. Мы указываем, что

нам нужен вектор `doubling` (столбец с индексом 1 после запятой). Обратите внимание: рассмотренный нами пример 89 × 18 можно было бы реализовать быстрее и проще, если бы вместо этого вычислялось произведение 18 × 89. То есть если бы значение 18 находилось в столбце `halving`, а значение 89 — в столбце `doubling`. Попробуйте самостоятельно реализовать это улучшение. В общем случае RPM работает быстрее, если меньший множитель находится в столбце деления, а больший — в столбце умножения.

Тому, кто уже запомнил таблицу умножения, алгоритм RPM может показаться бесполезным. Но помимо исторической ценности, его стоит изучить по нескольким причинам. Прежде всего, алгоритм показывает, что даже такую сухую операцию, как умножение чисел, можно выполнять по-разному, и в ней есть место для творческого подхода. Даже если вы освоили один алгоритм для какой-то задачи, это не означает, что он является единственным или лучшим алгоритмом для своей цели, — держите свой разум открытым для новых и, возможно, лучших решений.

RPM работает медленно, но требует меньших начальных усилий, поскольку вам не нужно заранее знать бóльшую часть таблицы умножения. Иногда полезно пойти на небольшие потери скорости ради снижения затрат памяти, и этот баланс между скоростью/затратами памяти становится важным фактором во многих ситуациях с проектированием и реализацией алгоритмов.

Как и многие лучшие алгоритмы, RPM также подчеркивает отношения между разрозненными, на первый взгляд, идеями. Может показаться, что двоичное разложение представляет интерес только для создателей транзисторов, но бесполезно для обывателя или профессионального программиста. Однако RPM демонстрирует глубокую связь между двоичным разложением числа и удобным способом умножения, требующим минимальных знаний таблицы умножения. Это еще одна причина, по которой всегда следует продолжать учиться: никогда не знаешь, когда бесполезный, на первый взгляд, факт может оказаться основой для мощного алгоритма.

Алгоритм Евклида

Древние греки преподнесли человечеству много даров. Одним из самых выдающихся их изобретений стала теоретическая геометрия, которая была методично собрана великим Евклидом в серию из 13 книг, названную «Начала». Большая часть математических трудов Евклида написана в стиле «теорема/доказательство», при котором сложное предположение логически выводится из более простых. Некоторые из его работ также были *конструктивными*, то есть в них предоставлялся метод использования простых средств для построения полезных фигур или

линий — например, квадрата с заданной площадью или касательной к кривой. И хотя сам термин «алгоритм» в то время еще не был придуман, конструктивные методы Евклида были алгоритмами, а некоторые из идей, лежащих в основе этих алгоритмов, актуальны и в наши дни.

Алгоритм Евклида вручную

Самый знаменитый алгоритм, описанный Евклидом, известен под названием *алгоритма Евклида*, хотя это всего лишь один из многих алгоритмов, о которых он писал. Алгоритм Евклида предназначен для нахождения наибольшего общего делителя двух чисел. Он прост и элегантен, а его реализация на Python состоит всего из нескольких строк.

Дано два натуральных (целых) числа: назовем их a и b. Допустим, a больше b (а если нет, то просто переименуйте a в b, а b в a, и тогда a будет большим). Если разделить a / b, то вы получите целое частное и целый остаток. Обозначим частное q_1, а остаток c. Это можно записать так:

$$a = q_1 \times b + c.$$

Например, если a = 105 и b = 33, то результат 105 / 33 равен 3, а остаток 6. Обратите внимание: остаток c всегда меньше как a, так и b — по определению остатка. На следующем шаге процесса мы забываем об a и концентрируемся на b и c. Как и прежде, допустим, что b больше c. Далее вычисляется частное и остаток при делении b / c. Обозначаем частное q_2, а остаток d, и записываем результат:

$$b = q_2 \times c + d.$$

И снова d будет меньше b и c, так как это остаток. Присмотревшись к двум уравнениям, можно заметить закономерность: мы продвигаемся по алфавиту, каждый раз сдвигая слагаемые влево. Мы начинали с a, b и c и перешли к b, c и d. Эта закономерность продолжается и на следующем шаге, на котором вычисляется результат c / d с получением частного q_3 и остатка e.

$$c = q_3 \times d + e.$$

Процесс продолжается, пока остаток не окажется равным нулю. Помните, что остатки всегда меньше чисел, которые делились для их получения: c меньше a и b, d меньше b и c, e меньше c и d, и т. д. Это означает, что на каждом шаге мы работаем со все меньшими числами, поэтому со временем доберемся до нуля. При получении нулевого остатка процесс останавливается, и мы знаем, что последний ненулевой остаток является наибольшим общим делителем. Например,

если окажется, что e равно нулю, то d является наибольшим общим делителем двух исходных чисел.

Реализация алгоритма Евклида на Python

Алгоритм Евклида достаточно легко реализуется на Python (листинг 2.1).

Листинг 2.1. Рекурсивная реализация алгоритма на языке Python

```
def gcd(x,y):
    larger = max(x,y)
    smaller = min(x,y)

    remainder = larger % smaller

    if(remainder == 0):
        return(smaller)

    if(remainder != 0):
❶       return(gcd(smaller,remainder))
```

Первое, на что следует обратить внимание, — что частные q_1, q_2, q_3... не нужны. Достаточно остатков, представленных буквами алфавита. Узнать остаток в Python несложно: это делается с помощью оператора %, который упоминался выше. Можно написать функцию, вычисляющую остаток от деления двух чисел. Если остаток равен 0, то наибольшим общим делителем является меньшее из двух входных значений. Если остаток отличен от нуля, то меньшее из двух входных значений и остаток становятся входными значениями для той же функции.

Обратите внимание: эта функция вызывает сама себя, если остаток отличен от нуля ❶. Механизм вызова функцией самой себя называется *рекурсией*. На первый взгляд, рекурсия кажется чем-то непонятным и устрашающим; такой вызов представляется неким парадоксом, наподобие змеи, которая пытается проглотить саму себя, или попыток барона Мюнхгаузена вытянуть себя за волосы из болота. Не бойтесь! Если вы еще не знакомы с рекурсией, то лучше всего начать с конкретного примера: скажем, определить наибольший общий делитель 105 и 33 и выполнить каждый этап кода так, как если бы вы были компьютером. Этот пример покажет, что рекурсия является всего лишь компактным способом выражения действий, описанных в подразделе «Алгоритм Евклида вручную» на с. 48. При рекурсии всегда существует опасность создания бесконечной рекурсии — то есть функция вызывает сама себя, в процессе вызова снова вызывает сама себя и т. д. Ничто не заставляет функцию завершиться, поэтому она пытается вызывать сама себя бесконечно — а это создает проблему, поскольку программа должна завершиться,

чтобы мы получили результат. В данном случае все выглядит безопасно, так как на каждом шаге мы получаем все меньшие остатки, которые в итоге уменьшатся до нуля, что позволит выйти из функции.

Алгоритм Евклида компактен, элегантен и полезен. Предлагаю вам подумать над тем, как создать еще более компактную реализацию этого алгоритма на Python.

Японские магические квадраты

История японской математики особенно увлекательна. В книге «История японской математики», опубликованной в 1914 году, историки Дэвид Юджин Смит (David Eugene Smith) и Ёсио Миками (Yoshio Mikami) написали, что японская математика исторически обладала «гениальной способностью прикладывать титанические усилия» и «изобретательностью в распутывании тысяч мелких узелков». С одной стороны, математики открывают абсолютные истины, которые не должны зависеть от времени и культуры. С другой — те или иные группы обычно склонны концентрироваться на задачах определенного типа и создают свои специфические подходы к ним, не говоря уже о различиях в системах записи и обмена информацией. И этот факт позволяет оценить примечательные культурные различия даже в такой аскетичной области, как математика.

Создание квадрата Ло Шу на Python

Японские математики обожали геометрию, и во многих древних рукописях формулируются и решаются задачи, связанные с нахождением площади разных экзотических фигур — например, эллипсов с вписанными кругами или японских ручных вееров. Другим постоянным направлением исследований для японских математиков на протяжении веков было изучение магических квадратов.

Магический квадрат представляет собой массив, содержащий уникальные последовательные натуральные числа, суммы всех строк, столбцов и двух главных диагоналей которого одинаковы. Магические квадраты могут иметь любой размер. В табл. 2.9 приведен пример магического квадрата 3 × 3.

Таблица 2.9. Квадрат Ло Шу

4	9	2
3	5	7
8	1	6

В этом квадрате суммы всех строк, столбцов и двух главных диагоналей равны 15. Пример выбран не случайно: это знаменитый *квадрат Ло Шу*. Согласно древнекитайской легенде, этот магический квадрат впервые был начертан на панцире волшебной черепахи, вышедшей из реки в ответ на молитвы и жертвоприношения страдающего народа.

Помимо определяющей закономерности с равенством сумм всех строк, столбцов и диагоналей, в квадрате прослеживаются и другие закономерности. Например, во внешнем кольце чисел чередуются четные и нечетные числа, а на главной диагонали располагаются последовательные числа 4, 5 и 6.

Легенда о внезапном появлении этого простого, но очаровательного квадрата как дара богов хорошо подходит для изучения алгоритмов. Алгоритмы часто легко проверять и использовать, но их бывает трудно проектировать «с нуля». Особенно элегантные алгоритмы, если нам вдруг повезет изобрести их, кажутся божественным откровением, словно они появились из ниоткуда на панцире волшебной черепахи. Если вы сомневаетесь в этом, то попробуйте построить магический квадрат 11 × 11 «с нуля» или изобрести алгоритм общего назначения для генерирования волшебных квадратов.

Информация об этом и других магических квадратах перешла из Китая в Японию как минимум в 1673 году, когда математик по имени Санэнобу опубликовал в Японии магический квадрат 20 × 20. На языке Python квадрат Ло Шу создается с помощью следующей программы:

```
luoshu = [[4,9,2],[3,5,7],[8,1,6]]
```

Будет полезно иметь функцию, которая проверяет, является ли заданный квадрат магическим. Для этого следующая функция вычисляет суммы по всем строкам, столбцам и диагоналям, а затем проверяет, что все они одинаковы:

```
def verifysquare(square):
    sums = []
    rowsums = [sum(square[i]) for i in range(0,len(square))]
    sums.append(rowsums)
    colsums = [sum([row[i] for row in square]) for i in range(0,len(square))]
    sums.append(colsums)
    maindiag = sum([square[i][i] for i in range(0,len(square))])
    sums.append([maindiag])
    antidiag = sum([square[i][len(square) - 1 - i] for i in \
    range(0,len(square))])
    sums.append([antidiag])
    flattened = [j for i in sums for j in i]
    return(len(list(set(flattened))) == 1)
```

Реализация алгоритма Курусимы на Python

Выше мы обсуждали, как выполнять интересующие нас алгоритмы «вручную», до того как предоставить подробности реализации. В случае алгоритма Курусимы мы кратко обрисуем основные шаги и одновременно рассмотрим код. Эти изменения объясняются относительной простотой алгоритма и особенно длиной кода, необходимого для его реализации.

Один из самых элегантных алгоритмов генерирования магических квадратов, *алгоритм Курусимы*, назван в честь математика по имени Курусима Ёсита, жившего в период Эдо. Алгоритм Курусимы работает только для магических квадратов нечетного размера, то есть для любого квадрата $n \times n$, где n — нечетное число. Он начинается с заполнения центра квадрата по такой же схеме, как в квадрате Ло Шу. В частности, пять центральных квадратов определяются следующими выражениями, где n — размер квадрата (табл. 2.10).

Таблица 2.10. Центр квадрата Курусимы

	n^2	
n	$(n^2 + 1) / 2$	$n^2 + 1 - n$
	1	

Алгоритм Курусимы для генерирования магического квадрата $n \times n$, где n — нечетное число, описывается с помощью простой схемы, представленной ниже.

1. Заполните пять центральных квадратов по табл. 2.10.
2. Начиная с любого элемента, значение которого уже известно, определите значение неизвестного соседнего элемента по одному из трех правил (см. далее).
3. Повторите шаг 2, пока не будут заполнены все элементы магического квадрата.

Заполнение центральных квадратов

Процесс создания магического квадрата можно начать с создания пустой квадратной матрицы, которая будет заполняться алгоритмом. Например, если вы хотите создать матрицу 7×7, то можно определить `n=7` и создать матрицу с `n` строками и `n` столбцами:

```
n = 7
square = [[float('nan') for i in range(0,n)] for j in range(0,n)]
```

Пока мы не знаем, какие числа должны находиться в квадрате, поэтому он будет заполнен значениями `float('nan')`. Здесь `nan` означает *«не число»* (Not A Number); это заполнитель, который может использоваться в Python для заполнения списков, когда числа не известны заранее. Если затем выполнить команду `print(square)`, то будет видно, что матрица в исходном состоянии заполнена значениями `nan`:

```
[[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan]]
```

В консольном выводе Python квадрат выглядит не слишком красиво, поэтому мы можем написать функцию, которая выводит его в намного более удобочитаемом виде:

```
def printsquare(square):
    labels = ['['+str(x)+']' for x in range(0,len(square))]
    format_row = «{:>6}» * (len(labels) + 1)
    print(format_row.format(«», *labels))
    for label, row in zip(labels, square):
        print(format_row.format(label, *row))
```

На функцию `printsquare()` пока можно не обращать внимания, поскольку она предназначена только для красивого вывода и не является частью нашего алгоритма. Пять центральных квадратов заполняются простыми командами. Сначала получим координаты центральной ячейки:

```
import math
center_i = math.floor(n/2)
center_j = math.floor(n/2)
```

Пять центральных квадратов заполняются в соответствии с выражениями из приведенной выше табл. 2.10:

```
square[center_i][center_j] = int((n**2 +1)/2)
square[center_i + 1][center_j] = 1
square[center_i - 1][center_j] = n**2
square[center_i][center_j + 1] = n**2 + 1 - n
square[center_i][center_j - 1] = n
```

Три правила

Основное содержание алгоритма Курусимы — заполнение остальных квадратов `nan` по простым правилам. Три простых правила позволяют заполнить любой другой

квадрат независимо от размера магического квадрата. Правило 1 объясняется на рис. 2.1.

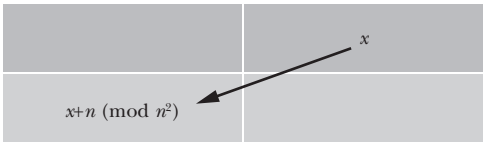


Рис. 2.1. Правило 1 алгоритма Курусимы

Чтобы для любого x в магическом квадрате определить значение, находящееся по диагонали от x, достаточно прибавить к x значение n и вычислить остаток от деления результата на n^2. Конечно, можно пойти и в противоположном направлении: вычесть n и вычислить остаток от деления результата на n^2.

Правило 2 еще проще, оно представлено на рис. 2.2.

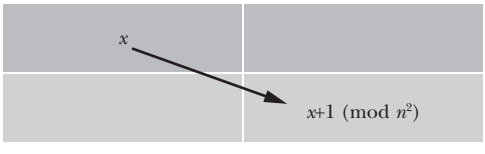


Рис. 2.2. Правило 2 алгоритма Курусимы

Для любого x в магическом квадрате элемент ниже и правее x равен остатку от деления $x + 1$ на n^2. Это простое правило, но у него есть одно важное исключение: оно не действует при переходе из левой верхней половины магического квадрата в правую нижнюю. Можно также сказать, что второе правило не выполняется при пересечении *антидиагонали* магического квадрата — линии, соединяющей левый нижний угол с правым верхним (рис. 2.3).

Здесь видны ячейки, находящиеся на антидиагонали. Линия полностью проходит через них. Имея дело с этими ячейками, можно следовать двум обычным правилам. Правило 3 необходимо только тогда, когда вы начинаете с ячейки, находящейся полностью над антидиагональю, и пересекаете ее, переходя в ячейку, находящуюся полностью под ней (или наоборот). Последнее правило представлено на рис. 2.4, на котором изображены антидиагональ и две ячейки, которые связываются правилом при ее пересечении.

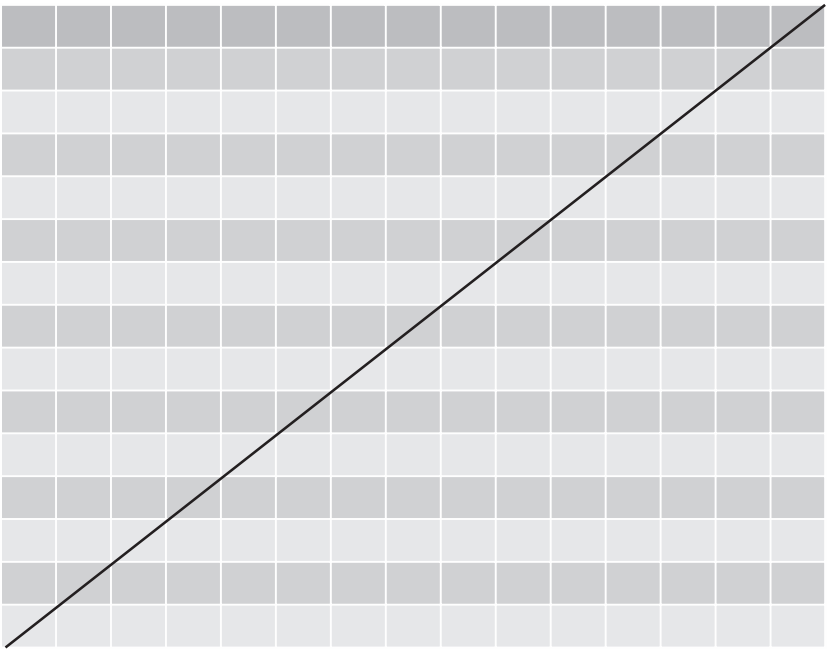


Рис. 2.3. Антидиагональ матрицы квадрата

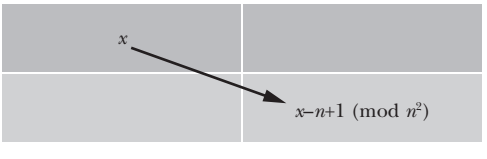


Рис. 2.4. Правило 3 алгоритма Курусимы

Это правило выполняется при пересечении антидиагонали. Если антидиагональ пересекается в направлении от правого нижнего угла к левому верхнему, то выполняется обратное правило — x преобразуется в $x + n - 1 \pmod{n^2}$.

Чтобы написать простую реализацию первого правила на Python, достаточно определить функцию, которая получает `x` и `n` в аргументах и возвращает `(x+n)%n**2`:

```
def rule1(x,n):
    return((x + n)%n**2)
```

Функцию можно опробовать на центральной ячейке квадрата Ло Шу. Квадрат представляет собой матрицу 3 × 3, так что $n = 3$. Центральная ячейка квадрата Ло Шу содержит значение 5. Ячейка ниже и левее ее содержит значение 8, и если функция `rule1()` была реализована правильно, то при выполнении следующей строки будет получено значение 8:

```
print(rule1(5,3))
```

На консоли Python должна появиться строка 8. Похоже, функция `rule1()` работает так, как задумано. Тем не менее ее можно улучшить, чтобы она могла работать и в обратном направлении, определяя значение не только внизу слева от заданной ячейки, но и значение справа наверху (то есть чтобы можно было перейти не только от 5 к 8, но и от 8 к 5). Чтобы внести это улучшение, можно добавить в функцию еще один аргумент `upright`; это индикатор `True`/`False`, который указывает, вычисляется ли значение справа и наверху от `x`. Если он не задан, то по умолчанию вычисляется значение слева внизу от `x`:

```
def rule1(x,n,upright):
    return((x + ((-1)**upright) * n)%n**2)
```

В математическом выражении Python интерпретирует `True` как 1, а `False` как 0. Если аргумент `upright` равен `False`, то наша функция вернет то же значение, что и прежде, так как $(-1)^0 = 1$. Если аргумент `upright` равен `True`, то функция будет вычитать `n` вместо того, чтобы прибавлять, и это позволит нам идти в другом направлении. Проверим, сможет ли функция определить значение справа и выше от 1 в квадрате Ло Шу:

```
print(rule1(1,3,True))
```

Функция должна вывести 7, правильное значение из квадрата Ло Шу.

Аналогичная функция создается и для правила 2. Функция для правила 2 получает аргументы `x` и `n`, как и функция для правила 1. Но функция для правила 2 по умолчанию изменяет ячейка ниже и правее `x`. А значит, мы будем прибавлять аргумент `upleft`, который будет равен `True`, если вы захотите изменить направление правила. Итоговое правило выглядит так:

```
def rule2(x,n,upleft):
    return((x + ((-1)**upleft))%n**2)
```

Функцию можно протестировать на квадрате Ло Шу, хотя в нем всего две пары ячеек, которые не сталкиваются с исключением из правила 2. Для этого исключения можно написать следующую функцию:

```
def rule3(x,n,upleft):
    return((x + ((-1)**upleft * (-n + 1)))%n**2)
```

Правило должно соблюдаться только при пересечении антидиагонали магического квадрата. Позднее вы увидите, как определить, пересекается ли антидиагональ.

Итак, теперь мы знаем, как заполнить пять центральных квадратов, и у нас есть правила для заполнения остальных ячеек по известным центральным ячейкам. Можно переходить к заполнению квадрата.

Заполнение остальных ячеек квадрата

Один из способов заполнить остальные ячейки — обойти его случайным образом, используя известные ячейки для заполнения неизвестных. Сначала определим индексы центральной ячейки:

```
center_i = math.floor(n/2)
center_j = math.floor(n/2)
```

Затем можно случайным образом выбрать направление для «обхода»:

```
import random
entry_i = center_i
entry_j = center_j
where_we_can_go = ['up_left','up_right','down_left','down_right']
where_to_go = random.choice(where_we_can_go)
```

Здесь используется функция Python `random.choice()`, которая выбирает случайный элемент из списка. Она выбирает элемент из заданного нами множества (`where_we_can_go`), но делает это случайно (или настолько случайно, насколько это возможно).

После того как направление для перемещения будет выбрано, выполняется правило, соответствующее направлению перемещения. Если выбрано перемещение вниз и налево (`down_left`) или вверх и направо (`up_right`), то мы следуем правилу 1, выбирая соответствующие аргументы и индексы:

```
if(where_to_go == 'up_right'):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,True)

if(where_to_go == 'down_left'):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,False)
```

Аналогичным образом при выборе направления вверх и налево (`up_left`) или вниз и направо (`down_right`) используется правило 2:

```
if(where_to_go == 'up_left'):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,True)

if(where_to_go == 'down_right'):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,False)
```

Код предназначен для перемещения вверх и налево или вниз и направо, но должен выполняться только в том случае, если при этом не пересекается антидиагональ. Необходимо позаботиться о том, чтобы правило 3 выполнялось только при пересечении антидиагонали. Проверить, находится ли значение рядом с антидиагональю, достаточно просто: индексы значений непосредственно над антидиагональю в сумме дают `n-2`, а индексы значений непосредственно под антидиагональю в сумме дают `n`. Для этих особых случаев нужно реализовать правило 3:

```
if(where_to_go == 'up_left' and (entry_i + entry_j) == (n)):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,True)

if(where_to_go == 'down_right' and (entry_i + entry_j) == (n-2)):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,False)
```

Следует помнить, что магический квадрат конечен, поэтому мы не сможем, например, сместиться вверх/налево в верхней строке или в крайнем левом столбце. Создав список направлений, в которых возможно перемещаться от текущей ячейки, можно добавить простую логику, чтобы проверить, происходит ли перемещение только в разрешенных направлениях:

```
where_we_can_go = []

if(entry_i < (n - 1) and entry_j < (n - 1)):
    where_we_can_go.append('down_right')

if(entry_i < (n - 1) and entry_j > 0):
    where_we_can_go.append('down_left')
```

```
if(entry_i > 0 and entry_j < (n - 1)):
    where_we_can_go.append('up_right')

if(entry_i > 0 and entry_j > 0):
    where_we_can_go.append('up_left')
```

Теперь у нас есть все элементы, которые необходимы для написания кода Python, реализующего алгоритм Курусимы.

Все вместе

Все, о чем говорилось выше, можно объединить в функцию, которая получает начальный квадрат со значениями nan и перемещается по нему, применяя три правила для заполнения. Все функция приведена в листинге 2.2.

Листинг 2.2. Функция, лежащая в основе реализации алгоритма Курусимы

```
import random
def fillsquare(square,entry_i,entry_j,howfull):
     while(sum(math.isnan(i) for row in square for i in row) > howfull):
        where_we_can_go = []

        if(entry_i < (n - 1) and entry_j < (n - 1)):
            where_we_can_go.append('down_right')
        if(entry_i < (n - 1) and entry_j > 0):
            where_we_can_go.append('down_left')
        if(entry_i > 0 and entry_j < (n - 1)):
            where_we_can_go.append('up_right')
        if(entry_i > 0 and entry_j > 0):
            where_we_can_go.append('up_left')

        where_to_go = random.choice(where_we_can_go)
        if(where_to_go == 'up_right'):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] =
                   rule1(square[entry_i][entry_j],n,True)

        if(where_to_go == 'down_left'):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j - 1
            square[new_entry_i][new_entry_j] =
                   rule1(square[entry_i][entry_j],n,False)

        if(where_to_go == 'up_left' and (entry_i + entry_j) != (n)):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j - 1
```

```
            square[new_entry_i][new_entry_j] =
                   rule2(square[entry_i][entry_j],n,True)

        if(where_to_go == 'down_right' and (entry_i + entry_j) != (n-2)):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] =
                   rule2(square[entry_i][entry_j],n,False)

        if(where_to_go == 'up_left' and (entry_i + entry_j) == (n)):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j - 1
            square[new_entry_i][new_entry_j] =
                   rule3(square[entry_i][entry_j],n,True)

        if(where_to_go == 'down_right' and (entry_i + entry_j) == (n-2)):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] =
                   rule3(square[entry_i][entry_j],n,False)

❶   entry_i = new_entry_i
    entry_j = new_entry_j
  return(square)
```

Эта функция получает четыре аргумента: первый — исходный квадрат, содержащий значения `nan`; второй и третий — индексы ячейки, с которой начинается обход; четвертый — степень заполнения квадрата (количество допустимых значений `nan`). Функция состоит из цикла `while`, который при каждой итерации записывает число в квадрат по одному из трех правил. Цикл продолжается, пока не остается количество значений `nan`, заданное четвертым аргументом. После записи значения в ячейку цикл «переходит» в эту ячейку, изменяя индексы (❶), после чего все повторяется.

После того как функция будет написана, остается только правильно вызвать ее.

Использование правильных аргументов

Начнем заполнять магический квадрат с центральной ячейки. В аргументе `howfull` будет передаваться значение `(n**2)/2-4`. Причина для использования такого значения `howfull` станет понятной, когда мы увидим результаты:

```
entry_i = math.floor(n/2)
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,(n**2)/2 - 4)
```

В данном случае функция `fillsquare()` вызывается с передачей существующей переменной `square`, определенной ранее. Напомню, что мы определили переменную `square` и заполнили ее значениями `nan`, кроме пяти центральных элементов. После того как функция `fillsquare()` с этим значением `square` на входе будет выполнена, она заполняет многие оставшиеся незаполненные значения.

Выведем полученный квадрат и посмотрим, как он выглядит после заполнения:

```
printsquare(square)
```

Результат выглядит так:

```
      [0]   [1]   [2]   [3]   [4]   [5]   [6]
[0]    22   nan    16   nan    10   nan     4
[1]   nan    23   nan    17   nan    11   nan
[2]    30   nan    24    49    18   nan    12
[3]   nan    31     7    25    43    19   nan
[4]    38   nan    32     1    26   nan    20
[5]   nan    39   nan    33   nan    27   nan
[6]    46   nan    40   nan    34   nan    28
```

Обратите внимание: `nan` занимают чередующиеся ячейки, образуя шахматный узор. Это объясняется тем, что выбранные нами правила диагонального перемещения дают нам доступ приблизительно к половине ячеек в зависимости от того, какая ячейка была выбрана как начальная.

Допустимы те же перемещения, что и при игре в шашки: фигура, начинающая на черном квадрате, может перемещаться по диагонали на другие черные квадраты, но диагональные движения никогда не позволят ей встать на белую клетку. Если обход начинается с центральной ячейки, то оставшиеся значения `nan` недоступны. В алгоритме `howfull` передается значение `(n**2)/2 - 4` вместо `0`, поскольку мы знаем, что однократный вызов функции не позволит заполнить всю матрицу. Но если начать с одного из соседей центральной ячейки, то можно получить доступ к остальным значениям `nan` «шахматной доски». Снова вызовем функцию `fillsquare()`, однако на этот раз с другой исходной ячейкой и четвертым аргументом, равным 0 (он означает, что в квадрате не должно остаться пустых ячеек):

```
entry_i = math.floor(n/2) + 1
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,0)
```

Если теперь вывести квадрат, то можно увидеть, что он полностью сформирован:

```
>>> printsquare(square)
         [0]   [1]   [2]   [3]   [4]   [5]   [6]
   [0]    22    47    16    41    10    35     4
   [1]     5    23    48    17    42    11    29
   [2]    30     6    24     0    18    36    12
   [3]    13    31     7    25    43    19    37
   [4]    38    14    32     1    26    44    20
   [5]    21    39     8    33     2    27    45
   [6]    46    15    40     9    34     3    28
```

Осталось внести одно последнее изменение. Из-за правил оператора `%` наш квадрат содержит последовательные числа от 0 до 48, а алгоритм Курусимы должен заполнять квадрат целыми числами от 1 до 49. Остается добавить одну строку, которая заменяет в квадрате 0 на 49:

```
square=[[n**2 if x == 0 else x for x in row] for row in square]
```

Построение квадрата завершено. Можно убедиться в том, что это действительно магический квадрат, с помощью созданной ранее функции `verifysquare()`:

```
verifysquare(square)
```

Функция должна вернуть `True` — признак того, что проверка прошла успешно.

Мы только что построили магический квадрат 7 × 7 по алгоритму Курусимы. Протестируем наш код и проверим, сможет ли он построить больший магический квадрат. Если мы заменим `n` числом 11 или другим нечетным значением, то можем выполнить тот же код и получить магический квадрат произвольного размера:

```
n = 11
square=[[float('nan') for i in range(0,n)] for j in range(0,n)]

center_i = math.floor(n/2)
center_j = math.floor(n/2)

square[center_i][center_j] = int((n**2 + 1)/2)
square[center_i + 1][center_j] = 1
square[center_i - 1][center_j] = n**2
square[center_i][center_j + 1] = n**2 + 1 - n
square[center_i][center_j - 1] = n

entry_i = center_i
entry_j = center_j

square = fillsquare(square,entry_i,entry_j,(n**2)/2 - 4)
```

```
entry_i = math.floor(n/2) + 1
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,0)

square = [[n**2 if x == 0 else x for x in row] for row in square]
```

Квадрат 11 × 11 выглядит так:

```
>>> printsquare(square)
         [0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]  [10]
   [0]    56   117    46   107    36    97    26    87    16    77     6
   [1]     7    57   118    47   108    37    98    27    88    17    67
   [2]    68     8    58   119    48   109    38    99    28    78    18
   [3]    19    69     9    59   120    49   110    39    89    29    79
   [4]    80    20    70    10    60   121    50   100    40    90    30
   [5]    31    81    21    71    11    61   111    51   101    41    91
   [6]    92    32    82    22    72     1    62   112    52   102    42
   [7]    43    93    33    83    12    73     2    63   113    53   103
   [8]   104    44    94    23    84    13    74     3    64   114    54
   [9]    55   105    34    95    24    85    14    75     4    65   115
  [10]   116    45   106    35    96    25    86    15    76     5    66
```

Убедитесь (вручную или с помощью функции `verifysquare()`), что это действительно магический квадрат. Все это можно проделать для любого нечетного `n` — и восхититься результатом.

Магические квадраты не имеют большого практического значения, но их закономерности могут быть занимательными. Если вы заинтересовались темой, то предлагаю обдумать следующие вопросы.

- Остается ли в больших магических квадратах, созданных нами, закономерность «чет/нечет» на внешней границе квадрата Ло Шу? Как вы думаете, существует ли эта закономерность во всех возможных магических квадратах? Чем она объясняется (если существует)?
- Не обнаружили ли вы другие закономерности в магических квадратах, созданных нами, которые еще не упоминались?
- Удастся ли вам найти другой набор правил для построения квадратов Курусимы? Например, существуют ли правила для перемещения вверх-вниз по квадрату Курусимы (вместо диагональных перемещений)?
- Существуют ли другие разновидности магических квадратов, которые удовлетворяют определению магического квадрата, но не следуют правилам Курусимы?

- Существует ли более эффективная программная реализация алгоритма Курусимы?

Магические квадраты веками привлекали внимание великих японских математиков, а также занимали значительное место в культурах всего мира. Нам повезло, что великие математики прошлого подарили нам алгоритмы для генерирования и анализа волшебных квадратов, которые легко реализуются на современных мощных компьютерах. В то же время остается только восхищаться терпением и проницательностью великих математиков прошлого, необходимыми для исследования магических квадратов, когда у этих людей не было ничего, кроме пера, бумаги и интеллекта (и появляющихся время от времени волшебных черепах).

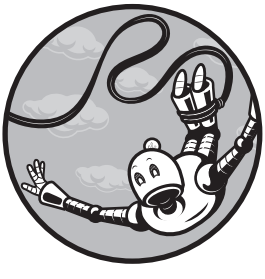
Резюме

В данной главе рассматривались некоторые старинные алгоритмы, возраст которых насчитывает от нескольких столетий до нескольких тысячелетий. Читатели, заинтересовавшиеся такими алгоритмами, найдут немало интересных тем для изучения. Возможно, сегодня эти алгоритмы не представляют особой практической ценности, но их определенно стоит изучить — во-первых, они представляют исторический интерес, а во-вторых, помогают расширить наши горизонты и вдохновляют на разработку собственных инновационных алгоритмов.

Алгоритмы, описанные в следующей главе, позволяют выполнять с математическими функциями некоторые практичные и полезные задачи: определять максимум и минимум. После того как мы обсудили алгоритмы вообще и их место в истории, вы начнете лучше понимать, что они собой представляют и как работают, и будете готовы заняться более серьезными алгоритмами, которые применяются в большинстве высокотехнологичных программных продуктов, разрабатываемых в наши дни.

3

Максимизация и минимизация



В жизни мы, как правило, стараемся избегать крайностей, но в мире алгоритмов интерес обычно представляют экстремальные значения. Некоторые мощные алгоритмы предназначены для поиска максимумов (например, максимальная прибыль, максимальная эффективность, максимальная производительность) и минимумов (минимальные затраты, минимальная вероятность ошибки, минимальный дискомфорт и т. д.). В этой главе мы изучим *градиентный подъем* и *градиентный спуск* — два простых, но действенных метода эффективного нахождения максимумов и минимумов функций. Кроме того, рассмотрим некоторые проблемы, присущие задачам максимизации и минимизации, и способы их решения. Наконец, мы обсудим, как узнать, уместно ли использовать тот или иной алгоритм в конкретной ситуации. Начнем с гипотетического сценария — выбора оптимальной ставки налога для максимизации доходов правительства и посмотрим, как использовать алгоритм для поиска решения.

Выбор ставки налога

Представьте, что вас избрали премьер-министром небольшой страны. Перед вами стоят амбициозные цели, но нет средств для их реализации. А значит, после вступления в должность вы первым делом должны максимизировать поступления от налогов, которые собирает ваше правительство.

Остается понять, какую ставку следует выбрать для максимизации налоговых поступлений. Если ставка налога составляет 0 %, то вы получите нулевые поступления в бюджет. При 100 % налогоплательщики наверняка постараются избежать какой-либо производственной деятельности и начнут искать лазейки настолько усердно, что поступления тоже будут близкими к нулю. Для оптимизации поступлений необходимо найти правильный баланс между ставками настолько высокими, что они подавляют производственную деятельность, и настолько низкими, что сборы оказываются недостаточными. Но чтобы достичь этого баланса, следует больше узнать о том, как налоговая ставка связана с поступлениями.

Шаги в правильном направлении

Допустим, вы обсуждаете проблему со своей командой экономистов. Они понимают, о чем идет речь, и возвращаются в научно-исследовательскую лабораторию. Там они применяют средства, используемые ведущими учеными-экономистами всего мира: колбы, астролябии, колеса с бегающими хомячками и ивовые прутья для лозоходства, чтобы определить точную связь между ставкой налога и поступлениями.

Через какое-то время экономисты сообщают, что нашли функцию, связывающую ставку налога с поступлениями, и написали ее для вас на Python. Можно предположить, что функция выглядит примерно так:

```
import math
def revenue(tax):
    return(100 * (math.log(tax+1) - (tax - 0.2)**2 + 0.04))
```

Эта функция Python получает ставку налога в аргументе и возвращает числовой вывод. Сама функция хранится в переменной `revenue`. Вы запускаете Python для построения простого графика и вводите с консоли приведенную ниже программу. Как и в главе 1, мы воспользуемся модулем `matplotlib` ради функциональности построения графиков.

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue(x) for x in xs]
plt.plot(xs,ys)
plt.title('Tax Rates and Revenue')
plt.xlabel('Tax Rate')
plt.ylabel('Revenue')
plt.show()
```

На графике приводятся налоговые поступления (в миллиардах денежных единиц вашей страны), которые ожидает получить ваша команда для всех ставок налога от 0 до 1 (где 1 соответствует 100 %-ной налоговой ставке). Если в вашей в настоящее время установлена ставка 70 %, то в код можно добавить две строки для нанесения этой точки на кривую:

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue(x) for x in xs]
plt.plot(xs,ys)
current_rate = 0.7
plt.plot(current_rate,revenue(current_rate),'ro')
plt.title('Tax Rates and Revenue')
plt.xlabel('Tax Rate')
plt.ylabel('Revenue')
plt.show()
```

В итоге мы получаем простой график, как на рис. 3.1.

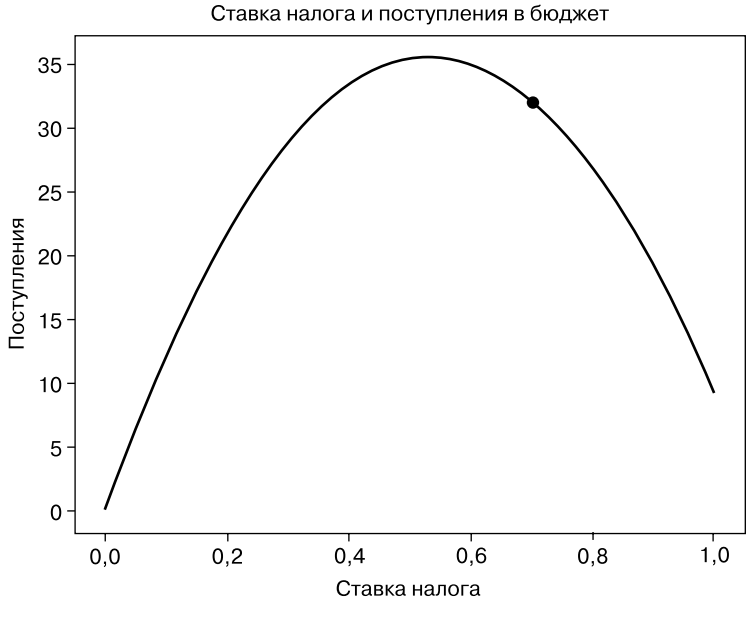


Рис. 3.1. Связь между ставкой налога и поступлениями; точка представляет текущее значение ставки

Согласно формуле экономистов, текущая ставка налога не обеспечивает максимальных поступлений в бюджет. Хотя простое визуальное изучение графика приблизительно показывает, какой уровень соответствует максимальным поступлениям, приближенная оценка вас не устраивает, и вы хотите найти более точное значение для оптимальной ставки налога. По графику видно, что любое повышение ставки от текущих 70 % приведет к снижению поступлений, а некоторое снижение текущей ставки должно повысить поступления, поэтому в данной ситуации максимизация поступлений требует снижения ставки налога.

В правильности этой оценки можно убедиться и более формально: для этого следует вычислить производную формулы поступлений, полученной от экономистов. *Производная* характеризует угол наклона касательной к графику функции; большие значения обозначают крутой рост, а отрицательные значения — убывание функции. Производная наглядно изображена на рис. 3.2; по сути, это всего лишь характеристика скорости роста или убывания функции.

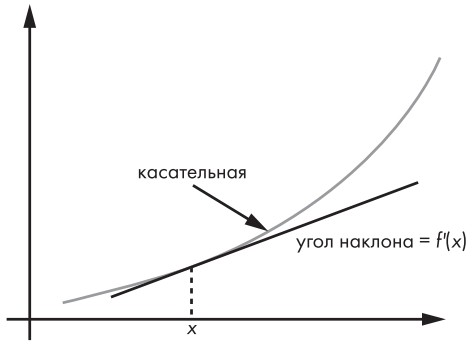


Рис. 3.2. Чтобы вычислить производную, нужно провести касательную к кривой в точке и определить ее угол наклона

Мы можем написать на Python функцию, которая задает производную следующим образом:

```
def revenue_derivative(tax):
    return(100 * (1/(tax + 1) - 2 * (tax - 0.2)))
```

Для вычисления этой функции были использованы четыре правила. Во-первых, мы использовали правило о том, что производная $log(x)$ равна $1 / x$. Отсюда следует, что производная $log(tax + 1)$ равна $1 / (tax + 1)$. Во-вторых, производная x^2 равна $2x$, поэтому производная $(tax - 0{,}2)^2$ равна $2(tax - 0{,}2)$. Два последних

правила — производная постоянного числа всегда равна 0, а производная $100f(x)$ равна 100 × производная $f(x)$. Объединяя все эти правила, мы находим, что функция $100(\log(tax + 1) - (tax - 0{,}2)^2 + 0{,}04)$ имеет следующую производную, которая соответствует приведенной выше функции Python:

$$100\left(\left(\frac{1}{tax+1}\right)-2(tax-0{,}2)\right).$$

Простая проверка показывает, что при текущей ставке налога производная действительно отрицательна:

```
print(revenue_derivative(0.7))
```

Команда выводит результат `-41.17647`.

Отрицательная производная означает, что увеличение ставки налога приведет к снижению поступлений. И наоборот, снижение ставки должно привести к их повышению. Хотя точное значение ставки, обеспечивающее максимум кривой, нам пока не известно, по крайней мере, можно утверждать, что при небольшом смещении в сторону уменьшения ставки налога поступления должны возрасти.

Чтобы сделать шаг в направлении максимальных поступлений, необходимо сначала выбрать его размер. Мы сохраним достаточно малый размер шага в переменной Python:

```
step_size = 0.001
```

Затем продвинемся в направлении максимума и вычислим новую ставку, находящуюся в одном шаге от текущей в направлении максимума:

```
current_rate = current_rate + step_size * revenue_derivative(current_rate)
```

На данный момент процесс поиска максимума выглядит так: мы начинаем с текущей ставки налога и продвигаемся к максимуму на величину, пропорциональную выбранному значению `step_size`, в направлении, определяемом производной функции «ставка налога-поступления» при текущем значении ставки.

Можно убедиться в том, что после этого шага новое значение `current_rate` составит `0.6588235` (ставка налога около 66 %), а поступления, соответствующие этой ставке, равны `33.55896`. Но хотя мы сделали шаг в направлении максимума и увеличили поступления, по сути, оказываемся в той же ситуации, что и прежде: максимум еще не достигнут, но мы знаем производную функции и общее направление для

дальнейшего перемещения. Таким образом, мы просто делаем еще один шаг — точно так же, как и прежде, но со значениями, соответствующими новой ставке. Снова выполняется присваивание:

```
current_rate = current_rate + step_size * revenue_derivative(current_rate)
```

После повторного присваивания новое значение `current_rate` равно `0.6273425`, а поступления, соответствующие новой ставке, составят `34.43267`. Мы сделали еще один шаг в правильном направлении. Тем не менее максимум еще не достигнут, и чтобы приблизиться к нему, нужно сделать еще один шаг.

Преобразование шагов в алгоритм

Нетрудно разглядеть постепенно проявляющуюся закономерность. Мы многократно выполняем действия, которые указаны ниже.

1. Начать с текущей ставки `current_rate` и размера шага `step_size`.
2. Вычислить производную максимизируемой функции в точке `current_rate`.
3. Прибавить к текущей ставке величину `step_size * revenue_derivative(current_rate)` для получения нового значения `current_rate`.
4. Повторять шаги 2 и 3.

В этой схеме не хватает только одного: правила остановки, то есть правила, срабатывающего при достижении максимума. На практике весьма вероятно, что мы будем приближаться к максимуму *асимптотически*, то есть подходить к нему все ближе и ближе, но всегда оставаясь на микроскопическом расстоянии от него. Итак, мы, возможно, никогда не достигнем максимума, но можем подойти достаточно близко к нему с точностью до 3-го, 4-го и даже 20-го знака в дробной части. Мы узнаем, что подошли достаточно близко к асимптоте, если размер приращения очень мал. Соответствующее значение можно задать в программе Python:

```
threshold = 0.0001
```

Процесс должен остановиться, когда ставка изменяется на величину, которая меньше порога `threshold` при каждой итерации процесса. Возможно и то, что процесс пошагового приближения вообще не сойдется к искомому максимуму, поэтому простой цикл может выполняться бесконечно. Чтобы подготовиться к этой возможности, мы зададим максимальное количество итераций, и если количество выполненных приближений достигнет этой величины, то программа просто отказывается от дальнейших попыток и останавливается.

Теперь все фрагменты можно собрать воедино (листинг 3.1).

Листинг 3.1. Реализация градиентного подъема

```
threshold = 0.0001
maximum_iterations = 100000

keep_going = True
iterations = 0
while(keep_going):
    rate_change = step_size * revenue_derivative(current_rate)
    current_rate = current_rate + rate_change

    if(abs(rate_change) < threshold):
        keep_going = False

    if(iterations >= maximum_iterations):
        keep_going = False

    iterations = iterations+1
```

После выполнения кода выясняется, что ставка налога, максимизирующая поступления в бюджет, составляет около `0.528`. Схема, реализованная в листинге 3.1, иногда называется *градиентным подъемом*. Она так названа из-за того, что алгоритм используется для подъема к максимуму, а направление движения определяется вычислением градиента. (В двумерном случае, как в нашей задаче, градиент называется производной.) Мы можем полностью перечислить все действия, которые были выполнены выше, включая описание критерия остановки.

1. Начать с текущей ставки `current_rate` и размера шага `step_size`.
2. Вычислить производную максимизируемой функции в точке `current_rate`.
3. Прибавить к текущей ставке величину `step_size * revenue_derivative(current_rate)` для получения нового значения `current_rate`.
4. Повторять шаги 2 и 3, пока не подойдем к максимуму настолько близко, что изменение ставки налога на каждом шаге будет меньше очень малого порога или пока не будет выполнено достаточно большое количество итераций.

Наш процесс в очень простом виде состоит всего из четырех шагов. Но несмотря на концептуальную простоту, градиентный подъем является алгоритмом, как и алгоритмы из предыдущих глав. Однако в отличие от тех алгоритмов, градиентный подъем часто применяется в наши дни и выступает ключевой частью многих современных методов машинного обучения, используемых профессионалами в повседневной работе.

Аргументы против градиентного подъема

Мы только что воспользовались градиентным подъемом для максимизации доходов гипотетического правительства. У многих людей, освоивших градиентный подъем, возникали практические, если не моральные, возражения против этого метода. Ниже приведены аргументы, которые обычно выдвигаются против градиентного подъема:

- он не нужен, поскольку максимум можно найти визуально на графике;
- он не нужен, так как максимум можно найти с помощью стратегии предположений и проверок;
- он не нужен, поскольку его можно заменить решением условий первого порядка.

Рассмотрим все эти возражения по очереди. Визуальная проверка уже обсуждалась ранее. Для кривой ставки налога/бюджетных поступлений легко получить примерное представление о максимуме, просто взглянув на график. Однако визуальный анализ графика не дает высокой точности. Что еще важнее, наша кривая была чрезвычайно простой: она строилась в двух измерениях и на ней очевидно был только один максимум в интересующем нас диапазоне. Если представить себе более сложные функции, вы начнете видеть, почему визуальный метод нельзя признать удовлетворительным способом поиска максимума функции.

Для примера возьмем многомерный случай. Если экономисты сделали вывод, что бюджетные поступления зависят не только от ставки налога, но и от таможенных пошлин, то кривую пришлось бы строить в трех измерениях, а для комплексной функции (complex function) будет еще сложнее увидеть, где лежит максимум. А если экономисты создадут функцию, которая связывает 10, 20 или 100 факторов для вычисления ожидаемых поступлений, то нарисовать график для всех факторов будет вообще невозможно. Если вы не можете даже нарисовать кривую поступлений, то визуальный метод никак не позволит найти на ней максимум. Визуальный метод хорошо подходит для простых учебных примеров вроде кривой ставки/поступлений, но не для комплексных многомерных задач. Помимо прочего, построение графика кривой требует вычисления значения функции в каждой отдельной точке интересующего диапазона, поэтому всегда занимает больше времени, чем выполнение хорошо написанного алгоритма.

Может показаться, что градиентный метод усложняет задачу, а простой стратегии предположений и проверок будет достаточно для нахождения максимума. Стратегия предположений и проверок строится на попытках угадать потенциальный максимум и проверках того, превышает ли он все ранее предложенные потенциальные

максимумы, пока вы не будете уверены в том, что максимум найден. Один возможный ответ указывает на то, что, как и в случае с визуальным методом, с многомерными функциями высокой сложности метод предположений и проверок становится неприемлемо сложным для успешной реализации на практике. Но лучший ответ на идею поиска максимума с помощью метода предположений и проверок заключается в том, что градиентный подъем *именно это и делает*. Градиентный подъем уже является стратегией предположений и проверок, но стратегией управляемой в том отношении, что предположения определяются направлением градиента, а не случайными догадками. Градиентный подъем — всего лишь более эффективная версия предположений и проверок.

Наконец, рассмотрим идею решения условий первого порядка для нахождения максимума. Этот метод преподают на курсах математического анализа по всему миру. Его можно назвать алгоритмом, и он состоит из шагов, которые перечислены ниже.

1. Найти производную функцию, которую вы пытаетесь максимизировать.
2. Приравнять эту производную нулю.
3. Найти точку, в которой производная равна нулю.
4. Убедиться в том, что эта точка соответствует максимуму, а не минимуму.

(В многомерных задачах можно работать с градиентом вместо производной, а в остальном процесс остается тем же.) Этот оптимизационный алгоритм в принципе не плох, но найти аналитическое решение с нулевой производной (шаг 2) может быть трудно или невозможно, и может оказаться, что найти это решение будет сложнее, чем просто выполнить градиентный подъем. Кроме того, это может потребовать огромных вычислительных ресурсов, включая дисковое пространство, вычислительные мощности или время, и не все программные продукты наделены возможностями символической алгебры. В этом смысле градиентный подъем надежнее, чем данный алгоритм.

Проблема локальных экстремумов

Каждый алгоритм, который пытается найти минимум или максимум, сталкивается с очень серьезной потенциальной проблемой локальных экстремумов (локальных максимумов и минимумов). Вы можете идеально реализовать градиентный подъем, а потом осознать, что найденный пик оказался только «локальным» пиком — он выше любой точки в окрестностях, но не выше некоторого удаленного глобального максимума. Нечто похожее может произойти и в реальной жизни: вы пытаетесь взобраться на гору, находите пик, который выше всего вокруг, а потом понимаете,

что это всего лишь предгорье, а настоящая вершина находится далеко и намного выше. Как ни парадоксально, вам придется немного спуститься, чтобы в итоге добраться до вершины, так что «наивная» стратегия градиентного подъема — постоянный подъем на более высокую точку в непосредственных окрестностях — не позволит добраться до глобального максимума.

Образование и пожизненный доход

Локальные экстремумы создают очень серьезную проблему для градиентного подъема. Например, рассмотрим задачу максимизации пожизненного дохода за счет выбора оптимального уровня образования. В данном случае можно предположить, что пожизненные заработки связываются с продолжительностью образования следующей формулой:

```
import math
def income(edu_yrs):
    return(math.sin((edu_yrs - 10.6) * (2 * math.pi/4)) + (edu_yrs - 11)/2)
```

Здесь переменная `edu_yrs` представляет продолжительность обучения в годах, а `income` — оценка пожизненного дохода. Можно построить график, показанный ниже, с точкой для человека, учившегося в течение 12,5 лет, то есть того, кто окончил старшие классы общеобразовательной школы[1]:

```
import matplotlib.pyplot as plt
xs = [11 + x/100 for x in list(range(901))]
ys = [income(x) for x in xs]
plt.plot(xs,ys)
current_edu = 12.5
plt.plot(current_edu,income(current_edu),'ro')
plt.title('Education and Income')
plt.xlabel('Years of Education')
plt.ylabel('Lifetime Income')
plt.show()
```

Полученный график изображен на рис. 3.3.

Этот график, а также использованная для его генерирования функция, не основан на эмпирических исследованиях, а используется как чисто гипотетический пример. Он отражает интуитивные представления между образованием и доходом. Скорее всего, у человека, который не окончил старшие классы общеобразовательной

[1] Здесь и далее речь об американской системе образования. — *Примеч. ред.*

Рис. 3.3. Отношения между продолжительностью образования и пожизненным доходом

школы (менее 12 лет образования), пожизненный доход будет невысоким. Окончание старших классов — 12 лет — является важной вехой, которая гарантирует людям более высокие заработки по сравнению с доходами тех, кто не окончил школу. Другими словами, это максимум, но, что очень важно, максимум локальный. Тот, кто всего несколько месяцев проучился в колледже, вряд ли получит работу заметно лучшую, чем выпускник общеобразовательной школы. С другой стороны, посещая колледж в течение нескольких месяцев, человек упускает возможность зарабатывать в этот период. Так что его пожизненный заработок будет ниже, чем у тех, кто начинает работать сразу же после окончания школы.

Только через несколько лет обучения в колледже студент приобретает навыки, позволяющие ему заработать за жизнь больше, чем выпускнику школы (с учетом потерянного заработка за годы обучения). Затем выпускники колледжа (16 лет образования) оказываются на более высоком пике заработка, превышающем локальный пик дохода после общеобразовательной школы. Но и этот пик является только локальным. А если вы захотите получить чуть больше знаний, получив степень бакалавра, то оказываетесь в такой же ситуации, как при дополнительном образовании после школы: не получаете достаточно знаний для компенсации времени, в течение которого не зарабатывали. Со временем ситуация меняется,

и после получения степени магистра появляется еще один пик. Дальнейшее развитие событий прогнозировать слишком сложно, но этого упрощенного представления о связи продолжительности образования и заработка будет достаточно для наших целей.

Правильный путь к вершинам образования

Для человека из нашего примера, отмеченного на графике как имеющего 12,5 лет образования, можно выполнить градиентный подъем точно так же, как объяснялось выше. В листинге 3.2 приведена слегка измененная версия кода градиентного подъема из листинга 3.1.

Листинг 3.2. Реализация градиентного подъема для максимизации заработка (вместо поступлений в бюджет)

```
def income_derivative(edu_yrs):
    return(math.cos((edu_yrs - 10.6) * (2 * math.pi/4)) + 1/2)

threshold = 0.0001
maximum_iterations = 100000

current_education = 12.5
step_size = 0.001

keep_going = True
iterations = 0
while(keep_going):
    education_change = step_size * income_derivative(current_education)
    current_education = current_education + education_change
    if(abs(education_change) < threshold):
        keep_going = False
    if(iterations >= maximum_iterations):
        keep_going=False
    iterations = iterations + 1
```

Код в листинге 3.2 реализует точно такой же алгоритм градиентного подъема, как алгоритм максимизации налоговых поступлений, который был реализован ранее. Отличается только кривая, с которой мы работаем. Кривая ставки налога/поступлений имела один глобальный максимум, который к тому же был единственным локальным максимумом. С другой стороны, кривая образования/заработка сложнее: она имеет глобальный максимум, но также несколько локальных максимумов (локальных пиков), которые ниже глобального максимума. Мы должны задать производную кривой (в первых строках листинга 3.2), используется другое исходное значение (12,5 лет вместо 70 % ставки налога, и переменным

присвоены другие имена (`current_education` вместо `current_rate`). Тем не менее все эти различия поверхностны; по сути происходит то же самое: мы делаем несколько шагов в направлении градиента к максимуму, пока не будет достигнута подходящая точка остановки.

В результате процесса градиентного подъема будет сделан вывод, что этот человек излишне образован, и в действительности максимальный доход достигается приблизительно при 12 годах образования. Если действовать слишком наивно и чрезмерно доверять градиентному подъему, то можно порекомендовать первокурснику колледжа бросать учебу и немедленно браться за работу, чтобы обеспечить максимальный заработок в этом локальном максимуме. К этому выводу уже приходили некоторые студенты, когда видели, что их друзья после школы начинали зарабатывать больше них, работавших на свое неопределенное будущее. Очевидно, этот вывод ошибочен; процесс градиентного подъема нашел вершину локального «холма», но не глобальный максимум. Процесс градиентного подъема удручающе локален: он взбирается только на ту вершину, возле которой уже находится, и не способен сделать несколько временных шагов вниз ради того, чтобы подняться на другой «холм» с более высокой вершиной. У этого процесса есть аналогии в реальной жизни: например, некоторые люди не получают университетский диплом, поскольку это помешает им зарабатывать в ближайшем будущем. Они не учитывают, что их долгосрочный заработок только увеличится, если они перейдут от локального максимума к другой вершине (следующей, более высокой ступени).

Локальные экстремумы — сложная задача, и идеального ее решения не существует. В одном из подходов к ее решению алгоритм пытается сделать несколько начальных предположений и проводит процедуру градиентного подъема для каждого из них. Например, если выполнить градиентный подъем для 12,5, 15,5 и 18,5 лет обучения, то мы получим три разных результата. Сравнивая их, можно увидеть, что глобальный максимум достигается при максимальной продолжительности образования (по крайней мере, на этой шкале).

Это разумный подход к решению проблемы локальных экстремумов, но многократное проведение градиентного подъема для получения правильного результата может занять слишком много времени, и правильный ответ не гарантирован даже при сотне попыток. Очевидно, лучший способ предотвратить эту проблему — внести некоторую степень случайности в процесс, чтобы мы могли иногда сделать шаг, который ведет к локально худшему решению, но в долгосрочной перспективе может привести к лучшему максимуму. Улучшенная версия градиентного подъема, называемая *стохастическим градиентным подъемом*, внедряет в этот процесс случайность, другие алгоритмы (такие как метод имитации отжига) делают то же

самое. Метод моделирования отжига и проблемы, связанные с нетривиальной оптимизацией, рассматриваются в главе 6. А пока учтите, что при всей своей эффективности градиентный подъем всегда будет сталкиваться с проблемами, связанными с локальными экстремумами.

От максимизации к минимизации

До настоящего момента мы старались максимизировать некоторую величину, то есть подниматься по «склону» вверх. Разумно задаться вопросом, потребуется ли когда-нибудь пойти в обратную сторону, и что-то минимизировать (например, затраты или ошибку). Можно подумать, что для минимизации потребуется совершенно новый набор приемов или что существующие приемы необходимо перевернуть с ног на голову либо применить в обратном порядке.

На самом деле переход от максимизации к минимизации достаточно прост. Как вариант, можно «перевернуть» функцию, или, говоря точнее, использовать инвертированную функцию. Возвращаясь к примеру с кривой ставки налога/поступлений, для этого достаточно определить новую инвертированную функцию:

```
def revenue_flipped(tax):
    return(0 - revenue(tax))
```

После этого график инвертированной функции строится так:

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue_flipped(x) for x in xs]
plt.plot(xs,ys)
plt.title('The Tax/Revenue Curve - Flipped')
plt.xlabel('Current Tax Rate')
plt.ylabel('Revenue - Flipped')
plt.show()
```

На рис. 3.4 изображена перевернутая кривая.

Итак, если мы хотим найти максимум кривой ставки налога/поступлений, то в одном из способов минимизируем перевернутую кривую. А если хотим найти минимум перевернутой кривой, то один из вариантов заключается в максимизации перевернутой перевернутой кривой — другими словами, исходной кривой. Каждая задача минимизации является задачей максимизации инвертированной функции, а каждая задача максимизации — задачей минимизации инвертированной функции. Если вы можете решить одну задачу, то сможете решить и другую (после

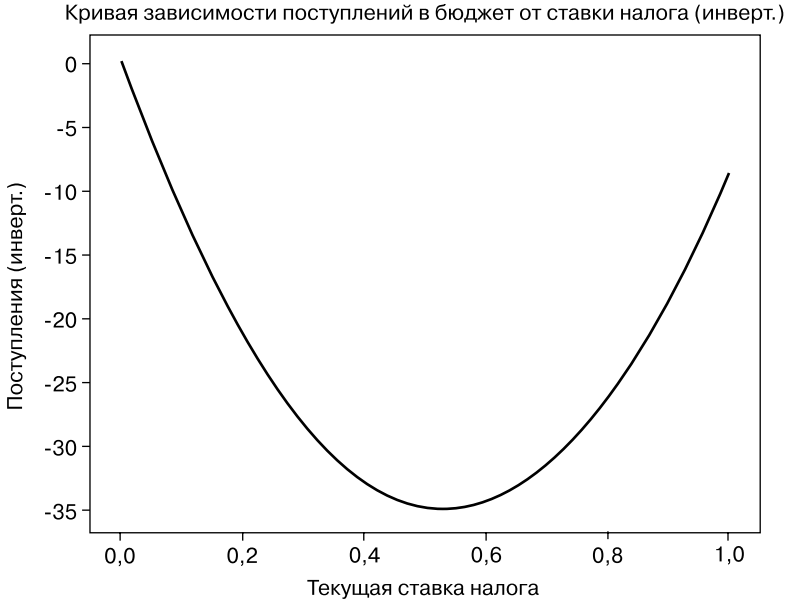


Рис. 3.4. Инвертированная, то есть «перевернутая» версия кривой ставки налога/поступлений

инвертирования). Вместо того чтобы учиться минимизировать функции, достаточно научиться максимизировать их. Тогда каждый раз, когда вам потребуется минимизировать функцию, вы можете максимизировать инвертированную функцию и получить правильный ответ.

Инвертирование — не единственное решение. Процесс поиска минимума очень похож на процесс поиска максимума: вместо *градиентного подъема* используется градиентный спуск. Единственное отличие — направление смещения на каждом шаге; при градиентном спуске мы двигаемся вниз, а не вверх. Напомню: чтобы найти максимум кривой ставки налога/поступлений, мы двигались по направлению градиента. Чтобы найти минимум, следует двигаться в обратном направлении. Это означает, что мы можем изменить исходный код градиентного подъема так, как показано в листинге 3.3.

Листинг 3.3. Реализация градиентного спуска

```
threshold = 0.0001
maximum_iterations = 10000

def revenue_derivative_flipped(tax):
```

```
    return(0-revenue_derivative(tax))

current_rate = 0.7

keep_going = True
iterations = 0
while(keep_going):
    rate_change = step_size * revenue_derivative_flipped(current_rate)
    current_rate = current_rate - rate_change
    if(abs(rate_change) < threshold):
        keep_going = False
    if(iterations >= maximum_iterations):
        keep_going = False
    iterations = iterations + 1
```

Все осталось тем же самым, не считая того, что знак `+` при изменении `current_rate` превратился в `-`. Ограничиваясь очень малыми изменениями, мы преобразовали код градиентного подъема в код градиентного спуска. В каком-то смысле эти два метода можно назвать эквивалентными: оба используют градиент для определения направления, а затем двигаются в данном направлении к определенной цели. В наше время обычно говорят о градиентном спуске, а градиентный подъем рассматривается как его слегка измененная версия — такой подход прямо противоположен тому, как два метода были представлены в текущей главе.

О пользе подъема

Назначение на должность премьер-министра — редкое событие, и выбор ставки налога для максимизации налоговых сборов не входит в повседневные обязанности премьер-министров. (Если вы хотите увидеть реальную версию зависимости поступлений от ставки налога, то рекомендую поискать информацию о кривой Лаффера.) Тем не менее идея максимизации или минимизации чего-либо встречается очень часто. Компании стараются выбирать цены для максимизации прибыли. Производители стремятся выбирать практики, которые максимизируют эффективность и минимизируют количество дефектов. Инженеры стараются выбирать конструктивные особенности, которые максимизируют производительность или минимизируют затраты. Экономика в значительной мере структурирована на задачах максимизации и минимизации, прежде всего максимизации эффективности и денежных сумм (ВВП, налоговые поступления) и минимизации ошибок оценки. Машинное обучение и статистика во многих своих методах опираются на минимизацию; они минимизируют «функцию потерь», или метрику ошибок. Во всех этих задачах заложен потенциал для использования методов поиска экстремума (таких как градиентный подъем или спуск) для поиска оптимального решения.

Даже в повседневной жизни мы выбираем, сколько денег потратить и как распределить доход. Мы стараемся максимизировать счастье, удовольствие и любовь и минимизировать боль, неудобства и огорчения.

Если вам нужен яркий и узнаваемый пример, то представьте, что вы за шведским столом, как и все мы, пытаетесь выбрать оптимальное количество еды. Если съедите слишком мало, то останетесь голодным и будете считать, что слишком дорого заплатили за небольшое количество еды и деньги были потрачены зря. Если съедите слишком много, то вам будет тяжело ходить, а может быть, вы нарушите установленную для себя диету, а то и заболеете. Существует «золотая середина», аналог пика на кривой ставке налога/поступлений, которая определяет оптимальное количество еды для максимизации удовлетворения.

Мы, люди, чувствуем и интерпретируем ощущения в своем животе, который сообщает нам, что мы голодны или сыты; иногда они становятся чем-то вроде физического аналога вычисления градиента кривой. Если мы слишком голодны, то делаем шаг заранее определенного размера (например, один кусочек) в направлении «золотой середины». Если слишком сыты, то просто перестаем есть; отменить что-то уже съеденное не получится. При достаточно малом размере шага можно быть уверенными в том, что нам удастся избежать слишком значительного перекрытия оптимума. Решая, сколько нужно съесть за шведским столом, мы тоже проходим итеративный процесс, в котором многократно проверяется направление и совершаются небольшие шаги в изменяющихся направлениях, — иначе говоря, происходит практически то же, что и в алгоритме градиентного подъема, рассмотренного в этой главе.

Как и в примере с ловлей мяча, мы видим, что такие алгоритмы, как градиентный подъем, естественны для человеческой жизни и принятия решений. Они естественны для нас даже в том случае, если мы никогда не посещали уроки математики и не написали ни единой строки кода. Инструменты, описанные в этой главе, всего лишь формализуют и уточняют уже имеющиеся у нас интуитивные представления.

Когда не следует применять алгоритм

Часто изучение нового алгоритма наполняет нас ощущением силы. Нам начинает казаться, что в любой ситуации, требующей максимизации или минимизации, следует немедленно применить градиентный подъем или спуск и довериться полученному результату. Однако иногда знать алгоритм — не главное. Еще важнее знать, когда не следует его применять, когда это неуместно или недостаточно для текущей задачи либо когда есть другое, более эффективное решение.

Когда использовать градиентный подъем (и спуск), а когда не стоит этого делать? Градиентный подъем хорошо работает в том случае, если у вас с самого начала имеются правильные ингредиенты:

- математическая функция для максимизации;
- информация о текущей точке;
- однозначный критерий максимизации функции;
- возможность изменять текущую точку.

Во многих ситуациях один или несколько из этих ингредиентов отсутствуют. При установлении ставки налога использовалась гипотетическая функция, связывающая ставку налога с поступлениями. Однако среди экономистов не существует единого мнения относительно того, что это за связь и в какой функциональной форме она воплощается. Таким образом, мы можем выполнять градиентный подъем и спуск сколько угодно, но пока все не согласятся с тем, какая функция должна максимизироваться, мы не можем положиться на полученные результаты.

В других ситуациях может оказаться, что градиентный подъем не особенно полезен, поскольку у нас нет возможности выполнить действие для оптимизации ситуации. Например, предположим, что мы вывели уравнение, связывающее рост человека с его личным счастьем. Возможно, эта функция показывает, что слишком высокие люди страдают, так как им неудобно сидеть в самолете, а слишком низкие — потому что не могут добиться успехов в баскетболе, а некий оптимум между слишком высоким и слишком низким ростом максимизирует уровень счастья. Но даже если идеально выразить данную функцию и применить градиентный подъем для нахождения максимума, пользы от результата не будет, поскольку люди не могут менять свой рост.

Можно пойти еще дальше: у вас могут быть все ингредиенты, необходимые для градиентного подъема (или любого другого алгоритма), но при этом вы все равно будете воздерживаться от его применения по более глубоким философским причинам. Допустим, вы точно определяете функцию ставки налога/поступлений и вас назначили премьер-министром с полным контролем над ставкой налогообложения в вашей стране. Прежде чем применять градиентный подъем и искать пик, максимизирующий поступления, стоит спросить себя, является ли максимизация налоговых поступлений правильной целью, с которой следует начинать. Возможно, вас в большей степени интересует личная свобода граждан, экономический рост, справедливость перераспределения или результаты опросов общественного мнения, чем поступления в бюджет. Даже если вы решили, что хотите обеспечить максимальный приток средств, неясно, приведет ли максимизация

доходов в краткосрочной перспективе (например, за год) к максимизации поступлений за долгий срок.

Алгоритмы эффективно работают при решении практических задач, позволяя нам достигать таких целей, как ловля мяча или поиск ставки налога, максимизирующей налоговые поступления. Но несмотря на это, алгоритмы не слишком подходят для более философской задачи выбора целей, к которым вообще стоит стремиться. Алгоритмы делают нас умнее, но сделать нас мудрее они не смогут. Важно помнить, что вся мощь алгоритмов окажется бесполезна, а то и вредна, если они будут применяться не в тех целях.

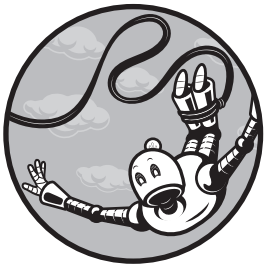
Резюме

В этой главе представлены градиентный подъем и градиентный спуск — простые и мощные алгоритмы, используемые для нахождения максимумов и минимумов функций соответственно. Кроме того, рассматривалась серьезная потенциальная проблема локальных экстремумов и некоторые философские соображения относительно того, когда следует применять алгоритмы, а когда от них лучше воздержаться.

В следующей главе будут рассматриваться различные алгоритмы сортировки и поиска. Сортировка и поиск играют фундаментальную и важную роль в мире алгоритмов. Вдобавок речь пойдет о нотации «О большое» и стандартных средствах оценки эффективности алгоритмов.

4

Сортировка и поиск



Существует ряд рабочих алгоритмов, используемых почти во всех программах. Иногда эти алгоритмы настолько фундаментальны, что мы воспринимаем их как нечто само собой разумеющееся или даже не понимаем, что наш код зависит от них.

Некоторые методы сортировки и поиска принадлежат к числу таких фундаментальных алгоритмов. Их стоит знать, поскольку они часто применяются и пользуются любовью поклонников алгоритмов (и садистов, проводящих собеседования для программистов). Реализация этих алгоритмов может быть простой и короткой, но некоторые теоретики постоянно стараются сделать так, чтобы они проводили сортировку и поиск с головокружительной скоростью. Вследствие этого в данной главе мы рассмотрим скорость выполнения алгоритмов и специальные обозначения, которые используются для сравнения эффективности алгоритмов.

Начнем с сортировки методом вставки — простого и интуитивно понятного алгоритма сортировки. Мы обсудим скорость и эффективность такой сортировки, а также оценим эффективность алгоритмов вообще. Затем рассмотрим сортировку слиянием — более быстрый алгоритм, который в настоящее время является наиболее эффективным в своем классе. Кроме того, изучим спящую сортировку — странный алгоритм, который на практике не применяется, но представляет

интерес как своеобразный курьез. В завершение мы обсудим двоичную сортировку и рассмотрим несколько интересных применений поиска, включая инвертирование математических функций.

Сортировка методом вставки

Представьте, что вас попросили рассортировать все папки в канцелярском шкафу. Каждой папке присвоен номер, и вы должны переставить папки, чтобы папка с наименьшим номером стояла на первом месте, а папка с наибольшим — на последнем. Промежуточные папки должны стоять между ними по порядку.

Какой бы способ вы ни выбрали, его можно описать как алгоритм сортировки. Но еще до того, как вы решили открыть Python с целью написать такой алгоритм, ненадолго остановитесь и подумайте, как бы вы отсортировали такие папки в реальной жизни. Задача может показаться простой, но позвольте вашему внутреннему искателю приключений оценить широкий диапазон возможностей.

В этом разделе представлен очень простой алгоритм сортировки, называемый *сортировкой методом вставки*. Данный метод основан на последовательной проверке каждого элемента списка и вставке его в новый список, который в итоге будет правильно отсортирован. Код нашего алгоритма состоит из двух частей: *вставки*, выполняющей тривиальную операцию вставки папки в список, и *сортировки*, которая многократно выполняет вставку, пока сортировка не будет завершена.

Вставка в сортировке методом вставки

Начнем с самой задачи вставки. Представьте, что вы стоите перед канцелярским шкафом, в котором папки уже идеально отсортированы. Кто-то подает вам новую папку и просит вставить в правильную (в порядке сортировки) позицию на полке. Как вы будете это делать? Задача может показаться настолько простой, что не требует объяснений, и вообще непонятно, возможно ли такое объяснение. (Что значит — как делать? *Просто берете и делаете!*) Но в мире алгоритмов каждую операцию, какой бы тривиальной она ни была, необходимо полностью объяснить.

Ниже описан разумный алгоритм вставки одной папки в отсортированную последовательность папок. Мы можем сравнить две папки и определить, что одна папка «больше» другой. Это может означать, что номер, присвоенный одной папке, больше номера другой либо занимает более высокую позицию по алфавиту или другому критерию.

1. Выбрать наибольшую папку на полке. (Начать с конца полки и двигаться к началу.)
2. Сравнить выбранную папку с той, которую нужно вставить.
3. Если выбранная папка меньше вставляемой, то поместить вставляемую папку в следующую позицию после выбранной.
4. Если выбранная папка больше вставляемой, то выбрать следующую папку на полке (ближе к началу).
5. Повторять шаги 2–4, пока папка не будет вставлена или пока вы не сравните ее с каждой существующей папкой. Если папка еще не была вставлена после сравнения со всеми существующими папками, то вставляется в самое начало полки.

Этот метод более или менее совпадает с интуитивным представлением о том, как вставить запись в отсортированный список. При желании также можно начать с начала списка, а не с конца, и применить аналогичный процесс с теми же результатами. Обратите внимание: мы не просто вставляем запись, а вставляем ее *в правильной позиции*, так что после вставки список остается отсортированным. Мы можем написать сценарий на языке Python, который выполняет этот алгоритм вставки. Сначала определим полку с отсортированными папками. В данном случае она будет представлена списком Python, а папки — обычными числами.

```
cabinet = [1,2,3,3,4,6,8,12]
```

Затем определяется «папка» (в данном случае просто число), которую нужно поставить на полку:

```
to_insert = 5
```

Мы последовательно перебираем все числа в списке (все папки на полке). Определим переменную `check_location`. Как было сказано выше, в ней будет храниться позиция проверяемой папки на полке. Начнем от конца полки:

```
check_location = len(cabinet) - 1
```

Определим также переменную `insert_location`. Цель нашего алгоритма — определить правильное значение `insert_location`, после чего останется только вставить папку в `insert_location`. Будем считать, что изначально переменная `insert_location` равна 0:

```
insert_location = 0
```

Затем простая команда `if` проверяет, что вставляемая папка больше папки в позиции `check_location`. Как только будет обнаружено число, меньшее, чем вставляемое, мы используем его позицию для определения места вставки нового числа. Значение увеличивается на 1, поскольку вставка выполняется в позиции непосредственно за меньшим найденным числом:

```
if to_insert > cabinet[check_location]:
    insert_location = check_location + 1
```

Определив правильную позицию `insert_location` для вставки папки на полку, можно воспользоваться встроенным методом Python `insert` для работы со списками:

```
cabinet.insert(insert_location,to_insert)
```

Впрочем, выполнение этого кода еще не приведет к вставке папки в нужную позицию. Необходимо объединить все действия в одну связную функцию. Она объединяет весь приведенный выше код, а также добавляет цикл `while`. Тот перебирает все папки на полке, начиная с последней и продвигаясь вперед до тех пор, пока не будет найдена правильная позиция `insert_location` или не будет проверена каждая папка. Окончательная версия кода вставки приведена в листинге 4.1.

Листинг 4.1. Вставка папки с заданным номером на полку

```
def insert_cabinet(cabinet,to_insert):
  check_location = len(cabinet) - 1
  insert_location = 0
  while(check_location >= 0):
    if to_insert > cabinet[check_location]:
        insert_location = check_location + 1
        check_location = - 1
    check_location = check_location - 1
  cabinet.insert(insert_location,to_insert)
  return(cabinet)

cabinet = [1,2,3,3,4,6,8,12]
newcabinet = insert_cabinet(cabinet,5)
print(newcabinet)
```

Если выполнить код из листинга 4.1, то он выведет список `newcabinet`, в котором теперь присутствует новая «папка» 5, вставленная в правильной позиции (между 4 и 6).

Стоит задуматься об одном граничном случае вставки: вставке в пустой список. В нашем алгоритме вставки упоминалось, что он последовательно перебирает все

папки на полке. Если на ней нет ни одной папки, то и последовательно перебирать нечего. В таком случае необходимо выполнить только последнюю операцию — вставку новой папки в начало полки. Конечно, это проще сделать, чем сказать, поскольку начало пустой полки также является ее концом и серединой. Все, что нужно сделать в данном случае, — вставить папку без учета позиции. Для этого можно воспользоваться функцией `insert()` в Python и выполнить вставку в позицию 0.

Сортировка методом вставки

Итак, мы формально определили вставку и знаем, как ее выполнить. Все почти готово к тому, чтобы выполнить сортировку методом вставки. Такая сортировка проста: последовательно перебираются элементы несортированного списка и используется алгоритм вставки для правильного размещения его в новом отсортированном списке. В аналогии с папками мы начинаем с несортированной полки, которую будем называть старой полкой, и второй пустой полки — новой. Сначала мы удалим первый элемент со старой несортированной полки и добавим его на новую пустую полку, используя алгоритм вставки. Далее то же самое сделаем со вторым элементом на старой полке, затем с третьим и т. д., пока все элементы со старой полки не окажутся на новой. Тогда мы забываем о старой полке и используем только новую с отсортированными папками. Поскольку при вставке использовался наш алгоритм методом вставки, а он всегда возвращает отсортированный список, мы знаем, что новая полка будет отсортирована в конце процесса.

В языке Python сначала создаются переменные для двух полок: несортированной и пустой новой:

```
cabinet = [8,4,6,1,2,5,3,7]
newcabinet = []
```

Сортировка методом вставки реализуется многократным вызовом функции `insert_cabinet()` из листинга 4.1. При вызове ей необходимо передать папку, которая предварительно снимается с несортированной полки:

```
to_insert = cabinet.pop(0)
newcabinet = insert_cabinet(newcabinet, to_insert)
```

В этом фрагменте используется метод `pop()`. Он удаляет из списка элемент с заданным индексом. В данном случае из `cabinet` удаляется элемент с индексом 0. После использования `pop()` `cabinet` уже не содержит этот элемент, поэтому он сохраняется в переменной `to_insert`, чтобы его можно было поместить в `newcabinet`.

В листинге 4.2 все эти элементы собраны воедино. Мы определяем функцию `insertion_sort()`, которая перебирает все элементы несортированного списка `cabinet` и вставляет их один за другим в `newcabinet`. Наконец, после выполнения списка выводится результат — отсортированная версия `sortedcabinet`.

Листинг 4.2. Реализация сортировки методом вставки

```
cabinet = [8,4,6,1,2,5,3,7]
def insertion_sort(cabinet):
  newcabinet = []
  while len(cabinet) > 0:
    to_insert = cabinet.pop(0)
    newcabinet = insert_cabinet(newcabinet, to_insert)
  return(newcabinet)

sortedcabinet = insertion_sort(cabinet)
print(sortedcabinet)
```

Освоив сортировку методом вставки, вы сможете отсортировать любой список, который вам попадется. Возникает соблазнительное ощущение, что вы знаете о сортировке все, что вам когда-либо понадобится. Однако она так важна и фундаментальна, что нам хотелось бы выполнять ее настолько эффективно, насколько это возможно. Прежде чем рассматривать альтернативы для сортировки методом вставки, разберемся с тем, что понимается под утверждением «один алгоритм лучше другого», а на базовом уровне — с тем, какой алгоритм вообще может считаться хорошим.

Оценка эффективности алгоритма

Можно ли назвать сортировку методом вставки хорошим алгоритмом? На этот вопрос трудно ответить, если не быть уверенным в том, что имеется в виду под «хорошим». Сортировка методом вставки работает — она сортирует списки; поэтому она хороша в том смысле, что достигает своей цели. Другой довод в пользу сортировки методом вставки: она понятна и легко объясняется на аналогиях с конкретными задачами, хорошо знакомыми многим людям. Еще один плюс в пользу этого алгоритма: для его выражения не требуется слишком много строк кода. Пока что сортировка методом вставки выглядит как хороший алгоритм.

Однако у сортировки методом вставки есть один критический недостаток: она слишком долго выполняется. На вашем компьютере код из листинга 4.2 почти наверняка выполнялся меньше секунды, поэтому «слишком долго» для сортировки методом вставки не сравнится с временем, которое понадобится для превращения маленького желудя в могучий дуб, или даже с временем, которое мы проводим

в очереди в автоинспекции. Скорее это слишком долго по сравнению с временем, за которое комар взмахивает крыльями.

Может показаться, что беспокоиться о том, что алгоритм выполняется «слишком долго» по сравнению со взмахом комариных крыльев, — это перебор. Однако есть несколько веских причин для того, чтобы использовать алгоритмы, выполняющиеся как можно быстрее.

Почему так важна эффективность

Первая причина неустанно стремиться к эффективности алгоритмов заключается в том, что они наращивают нашу вычислительную мощность. Если ваш неэффективный алгоритм сортирует список из восьми элементов за минуту, то это вроде бы не создает особых проблем. Но подумайте, что на сортировку списка из тысячи элементов такому неэффективному алгоритму потребуется час, а списка из миллиона элементов — неделя. На сортировку списка из миллиарда элементов может уйти год или столетие (если это вообще получится сделать). Если улучшить алгоритм так, чтобы он быстрее сортировал список из восьми элементов (вроде бы тривиальная задача, которая экономит всего минуту), то это может привести к тому, что список из миллиарда элементов будет сортироваться за час, а не за столетие, что может открыть много новых возможностей. Современные методы машинного обучения — такие как кластеризация методом k-средних или метод k-ближайших соседей — зависят от упорядочения длинных списков. Улучшение производительности таких фундаментальных алгоритмов, как сортировка, позволит нам выполнять такие методы с большими наборами данных, которые иначе бы выходили за пределы наших возможностей.

Даже сортировку коротких списков важно выполнять быстро, если она должна выполняться много раз. Например, поисковые системы в глобальном масштабе получают триллион запросов за несколько месяцев, и каждый набор результатов приходится упорядочивать по убыванию релевантности перед возвращением их пользователю. Если время, необходимое для сортировки простого списка, удастся сократить с секунды до половины секунды, то необходимое время сортировки сократится с миллиона секунд до половины миллиона секунд. В результате экономится время (а экономия тысячи секунд для половины миллиарда людей быстро накапливается!) и сокращаются затраты на обработку данных, а из-за снижения потребления энергии эффективные алгоритмы к тому же оказываются более экологически безопасными.

Последняя причина для создания быстрых алгоритмов — та же, по которой люди стараются добиться наилучших результатов в любом деле. Хотя нет очевидной

необходимости, люди стараются быстрее пробегать стометровку, лучше играть в шахматы и готовить пиццу вкуснее, чем кто-либо прежде. Они делают это по той же причине, которой Джордж Мэллори объяснил свое желание покорить Эверест: «Потому что он есть». Человеческая природа заставляет пытаться выходить за пределы возможного и стараться создать лучшие, более быстрые, мощные и умные алгоритмы, чем создавали другие. Исследователи алгоритмов стараются добиться лучших результатов, поскольку среди прочего пытаются сделать что-то выдающееся независимо от того, приносит оно практическую пользу или нет.

Точное измерение времени

Так как время, необходимое для выполнения алгоритма, очень важно, оценкам типа «слишком долго» или «меньше секунды» не хватает точности. Сколько именно оно занимает? Чтобы получить буквальный ответ, можно воспользоваться модулем `timeit` на Python. Он позволяет создать таймер, который запускается непосредственно в начале сортировки и останавливается сразу же после ее завершения. Вычисляя разность между начальным и конечным временем, можно определить, сколько времени понадобится для выполнения кода.

```
from timeit import default_timer as timer

start = timer()
cabinet = [8,4,6,1,2,5,3,7]
sortedcabinet = insertion_sort(cabinet)
end = timer()
print(end - start)
```

Когда я выполнил этот код на моем обычном ноутбуке, на это потребовалось около 0,0017 секунды. Это дает разумную оценку эффективности алгоритма сортировки методом вставки — он способен полностью отсортировать список из восьми элементов за 0,0017 секунды. Если вы хотите сравнить сортировку методом вставки с любым другим алгоритмом сортировки, то можно сравнить результаты хронометража `timeit` с другим хронометражем, определить, какой работает быстрее, и сказать, что быстрый алгоритм лучше.

Однако сравнение быстродействия алгоритмов по хронометражу создает некоторые проблемы. Например, когда я запустил код хронометража на своем ноутбуке во второй раз, он был выполнен за 0,0008 секунды. В третий раз оказалось, что на другом компьютере он был выполнен за 0,03 секунды. Точный хронометраж, который вы получаете, зависит от скорости и архитектуры оборудования, текущей загруженности операционной системы (ОС), используемой версии Python,

внутреннего планировщика задач ОС, эффективности вашего кода и, вероятно, других случайностей, движения электронов и фаз луны. Так как мы можем получать сильно различающиеся результаты при разных попытках измерения эффективности, трудно полагаться на хронометраж для оценки сравнительной эффективности. Один программист хвастается, что может отсортировать список за Y секунд, другой смеется и говорит, что его алгоритм эффективнее и справляется с задачей за Z секунд. Может оказаться, что они выполняли абсолютно одинаковый код, но на разном оборудовании в разное время, поэтому сравнение дает оценку не эффективности алгоритма, а вычислительной мощности оборудования и везения.

Подсчет шагов

Вместо затрат времени в секундах более надежной метрикой быстродействия алгоритма оказывается количество шагов, необходимых для выполнения алгоритма. Количество шагов в алгоритме — характеристика самого алгоритма, не зависит от архитектуры оборудования, а иногда даже от языка программирования. В листинге 4.3 приведен код сортировки методом вставки из листингов 4.1 и 4.2 с добавлением нескольких строк с командой `stepcounter+=1`. Счетчик шагов увеличивается каждый раз, когда мы берем со старой полки новую папку для вставки, при каждом сравнении этой папки с другой папкой на новой полке и при каждой вставке папки на новой полке.

Листинг 4.3. Код сортировки методом вставки со счетчиком шагов

```
def insert_cabinet(cabinet,to_insert):
  check_location = len(cabinet) - 1
  insert_location = 0
  global stepcounter
  while(check_location >= 0):
    stepcounter += 1
    if to_insert > cabinet[check_location]:
        insert_location = check_location + 1
        check_location = - 1
    check_location = check_location - 1
  stepcounter += 1
  cabinet.insert(insert_location,to_insert)
  return(cabinet)

def insertion_sort(cabinet):
  newcabinet = []
  global stepcounter
  while len(cabinet) > 0:
    stepcounter += 1
    to_insert = cabinet.pop(0)
```

```
    newcabinet = insert_cabinet(newcabinet,to_insert)
  return(newcabinet)

cabinet = [8,4,6,1,2,5,3,7]
stepcounter = 0
sortedcabinet = insertion_sort(cabinet)
print(stepcounter)
```

В данном случае при выполнении этого кода мы видим, что для реализации сортировки методом вставки для списка длиной 8 выполняются 36 шагов. Попробуем выполнить сортировку методом вставки для списков другой длины и посмотрим, сколько шагов для этого потребуется.

Для этого напишем функцию, которая подсчитывает шаги, необходимые для сортировки методом вставки для несортированных списков разной длины. Вместо того чтобы записывать каждый несортированный список вручную, мы воспользуемся простым списковым включением в Python для генерирования случайного списка любой заданной длины. Для упрощения генерирования случайных списков в Python можно импортировать модуль Python `random`. Пример создания случайного несортированного списка длины 10:

```
import random
size_of_cabinet = 10
cabinet = [int(1000 * random.random()) for i in range(size_of_cabinet)]
```

Наша функция просто генерирует список заданной длины, выполняет код сортировки методом вставки и возвращает итоговое найденное значение для `stepcounter`.

```
def check_steps(size_of_cabinet):
  cabinet = [int(1000 * random.random()) for i in range(size_of_cabinet)]
  global stepcounter
  stepcounter = 0
  sortedcabinet = insertion_sort(cabinet)
  return(stepcounter)
```

Создадим список всех чисел от 1 до 100 и проверим, сколько шагов потребуется для сортировки списка каждой длины:

```
random.seed(5040)
xs = list(range(1,100))
ys = [check_steps(x) for x in xs]
print(ys)
```

В этом коде сначала вызывается функция `random.seed()`. Этот вызов не является необходимым, но гарантирует, что при выполнении того же кода вы получите такие

же результаты. Как видите, мы определяем набор значений для `x`, хранящийся в `xs`, и набор значений для `y`, хранящийся в `ys`. Значения `x` просто представляют собой числа от 1 до 100, а значения `y` — количества шагов, необходимые для сортировки случайно сгенерированных списков каждого размера, соответствующего `x`. Взглянув на вывод, вы увидите, сколько шагов потребуется сортировке методом вставки для случайно сгенерированных списков длины 1, 2, 3... 99. График зависимости между длиной списка и количеством шагов сортировки строится фрагментом, приведенным ниже. Для построения графика импортируется пакет `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt
plt.plot(xs,ys)
plt.title('Steps Required for Insertion Sort for Random Cabinets')
plt.xlabel('Number of Files in Random Cabinet')
plt.ylabel('Steps Required to Sort Cabinet by Insertion Sort')
plt.show()
```

Результат показан на рис. 4.1. Как видите, выходная кривая получается неровной — иногда более длинный список сортируется за меньшее количество шагов, чем короткий. Причина в том, что все списки генерируются случайным образом. Время от времени код генерирования случайных списков создает список, который

Рис. 4.1. Количество шагов при сортировке методом вставки

слишком легко упорядочивается и быстро обрабатывается сортировкой методом вставки (поскольку он уже частично отсортирован), а иногда создаются списки, которые труднее обработать быстро — все зависит исключительно от случайности. По той же причине может оказаться, что вывод на вашем компьютере отличается от приведенного здесь, если вы не инициализировали генератор случайных чисел тем же значением, но в целом форма графика останется той же.

Сравнение с известными функциями

Забудем об искусственной неровности рис. 4.1, проанализируем общую форму кривой и попробуем проанализировать скорость ее роста. На интервале между $x = 1$ и $x = 10$ количество шагов растет довольно медленно. После этого кривая постепенно набирает крутизну и становится более неровной. Между $x = 90$ и $x = 100$ скорость роста становится очень заметной.

Можно сказать, что с ростом длины списка кривая постепенно становится более крутой, но и это описание не настолько точное, как нам хотелось бы. Иногда нарастающую скорость роста называют экспоненциальной. Имеем ли мы дело с экспоненциальным ростом в данном случае? Строго говоря, существует *экспоненциальная функция* e^x, где e — число Эйлера, или приблизительно 2,71828. Подчиняется ли количество шагов, необходимых для сортировки методом вставки, экспоненциальной функции, которую можно назвать самым точным из возможных определений экспоненциального роста? Чтобы получить наглядное представление, построим кривую количества шагов вместе с кривой экспоненциального роста, как показано ниже. Для определения максимума и минимума в программе также импортируется модуль `numpy`.

```
import math
import numpy as np
random.seed(5040)
xs = list(range(1,100))
ys = [check_steps(x) for x in xs]
ys_exp = [math.exp(x) for x in xs]
plt.plot(xs,ys)
axes = plt.gca()
axes.set_ylim([np.min(ys),np.max(ys) + 140])
plt.plot(xs,ys_exp)
plt.title('Comparing Insertion Sort to the Exponential Function')
plt.xlabel('Number of Files in Random Cabinet')
plt.ylabel('Steps Required to Sort Cabinet')
plt.show()
```

Как и прежде, мы определяем `xs` как список всех чисел от 1 до 100, а `ys` — как количество шагов, необходимое для сортировки случайно сгенерированных списков

размера, соответствующего каждому `x`. Вдобавок определяется переменная `ys_exp`, которая содержит значение e^x для каждого из значений, хранящихся в `xs`. Затем значения `ys` и `ys_exp` наносятся на один график. В результате мы видим, как количество шагов, требуемых для сортировки списка, соотносится с настоящим экспоненциальным ростом.

При выполнении этого кода будет получен график, как на рис. 4.2.

Рис. 4.2. Сравнение количества шагов при сортировке методом вставки с экспоненциальной функцией

Мы видим, что настоящая кривая экспоненциального роста устремляется к бесконечности в левой части графика. Хотя кривая количества шагов сортировки методом вставки растет в нарастающем темпе, ускорение даже близко не стоит с настоящим экспоненциальным ростом. Если нанести на график другие функции, скорость роста которых тоже может называться экспоненциальной, 2^x или 10^x, то вы увидите, что все эти функции растут намного быстрее нашей кривой количества шагов. Если кривая количества шагов сортировки методом вставки отстает от экспоненциального роста, то какой скорости роста соответствует? Попробуем нанести на тот же график еще несколько функций. В следующем фрагменте вместе с функцией количества шагов сортировки методом вставки строятся графики функций $y = x$, $y = x^{1,5}$, $y = x^2$ и $y = x^3$:

```
random.seed(5040)
xs = list(range(1,100))
ys = [check_steps(x) for x in xs]
xs_exp = [math.exp(x) for x in xs]
xs_squared = [x**2 for x in xs]
xs_threehalves = [x**1.5 for x in xs]
xs_cubed = [x**3 for x in xs]
plt.plot(xs,ys)
axes = plt.gca()
axes.set_ylim([np.min(ys),np.max(ys) + 140])
plt.plot(xs,xs_exp)
plt.plot(xs,xs)
plt.plot(xs,xs_squared)
plt.plot(xs,xs_cubed)
plt.plot(xs,xs_threehalves)
plt.title('Comparing Insertion Sort to Other Growth Rates')
plt.xlabel('Number of Files in Random Cabinet')
plt.ylabel('Steps Required to Sort Cabinet')
plt.show()
```

Результат показан на рис. 4.3.

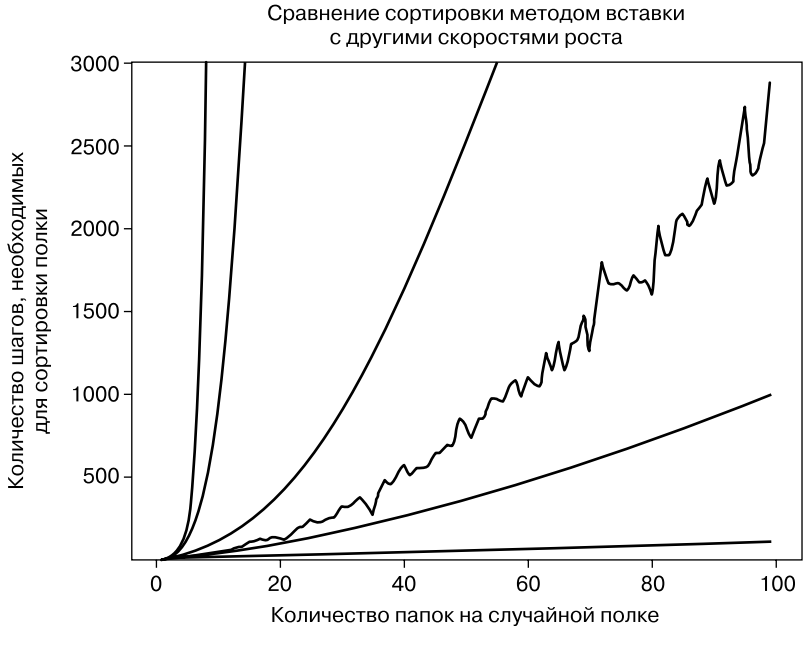


Рис. 4.3. Сравнение количества шагов сортировки методом вставки с другими скоростями роста

На рис. 4.3 представлены три скорости роста помимо зазубренной кривой, характеризующей количество шагов, необходимых для сортировки методом вставки. Как видно из графика, экспоненциальная кривая растет быстрее всех, а следующая за ней кубическая кривая почти не видна на графике, поскольку тоже растет очень быстро. Функция $y = x$ растет очень медленно по сравнению с другими; она видна в самой нижней части графика.

К кривой количества шагов ближе всего функции $y = x^2$ и $y = x^{1,5}$. Пока не очевидно, какая из них ближе к нашей кривой, так что мы не можем что-либо с уверенностью утверждать относительно скорости роста. Но после создания графика можно выдавать утверждения наподобие «сортировка списка из n элементов требует количества шагов приблизительно в диапазоне от $n^{1,5}$ до n^2». Это более точная и обоснованная формулировка, чем «со скоростью взмаха комариного крыла» или «около 0,002 секунды на моем конкретном ноутбуке этим утром».

Повышение теоретической точности

Чтобы добиться еще большей точности, следует тщательно проанализировать количество шагов, необходимых для сортировки методом вставки. Еще раз представим, что у вас имеется новый несортированный список из n элементов. В табл. 4.1 подсчитываются все шаги сортировки методом вставки.

Таблица 4.1. Подсчет шагов при сортировке методом вставки

Описание действий	Количество шагов, необходимых для снятия папки со старой полки	Максимальное количество шагов, необходимых для сравнения с другими папками	Количество шагов, необходимых для вставки папки на новую полку
Взять первую папку со старой полки и поставить ее на (пустую) новую полку	1	0 (нет папок для сравнения)	1
Взять вторую папку со старой полки и поставить ее на новую полку (на которой сейчас стоит одна папка)	1	1 (есть одна папка для сравнения, необходимо выполнить сравнение с ней)	1

Описание действий	Количество шагов, необходимых для снятия папки со старой полки	Максимальное количество шагов, необходимых для сравнения с другими папками	Количество шагов, необходимых для вставки папки на новую полку
Взять третью папку со старой полки и поставить ее на новую полку (на которой сейчас стоят две папки)	1	2 или менее (есть две папки для сравнения, необходимо выполнить сравнение от 1 до всех папок)	1
Взять четвертую папку со старой полки и поставить ее на новую полку (на которой сейчас стоят три папки)	1	3 или менее (есть три папки для сравнения, необходимо выполнить сравнение от 1 до всех папок)	1
...	...	...	...
Взять n-ю папку со старой полки и поставить ее на новую полку (на которой сейчас стоят $n - 1$ папок)	1	$n - 1$ или менее (есть $n - 1$ папок для сравнения, необходимо выполнить сравнение от 1 до всех папок)	1

Если просуммировать все шаги, описанные в таблице, то мы получим следующие значения:

- шаги, необходимые для снятия папок: n (1 шаг на снятие каждой из n папок);
- шаги, необходимые для сравнения: до $1 + 2 + ... + (n - 1)$;
- шаги, необходимые для вставки папок: n (1 шаг на вставку каждой из n папок).

Суммируя эти значения, мы получаем выражение следующего вида:

$$\text{максимальное\_количество\_шагов} = n + (1 + 2 + ... + n).$$

Выражение можно упростить с помощью удобного тождества:

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

Если воспользоваться этим тождеством, просуммировать все значения и упростить, то можно обнаружить, что общее количество необходимых шагов равно

$$\text{максимальное\_количество\_шагов} = \frac{n^2}{2} + \frac{3n}{2}.$$

Наконец-то у нас появилось очень точное выражение для максимального количества шагов, которые могут потребоваться для выполнения сортировки методом вставки. Но как ни странно, это выражение может быть даже слишком точным по нескольким причинам. Одна из них такова: мы вычислили максимальное количество шагов, а минимальное и среднее количество может быть намного меньше, и почти каждый список, который мы захотим отсортировать, потребует меньшего количества шагов. Вспомните неровность кривой, изображенной на рис. 4.1, — всегда существует разброс во времени выполнения алгоритма в зависимости от выбора входных данных.

Другая причина, по которой выражение для максимального количества шагов может быть слишком точным, заключается в том, что знание количества шагов алгоритма наиболее важно для больших значений n, но для очень больших n небольшая часть выражения начинает превосходить все остальные по важности из-за резкого расхождения скоростей роста разных функций.

Возьмем выражение $n^2 + n$. Оно состоит из двух слагаемых: n^2 и n. Для $n = 10$ выражение $n^2 + n$ равно 110, которое на 10 % больше n^2. Для $n = 100$ выражение $n^2 + n$ равно 10 100, которое всего на 1 % больше n^2. С ростом n слагаемое n^2 становится более важным, чем слагаемое n, поскольку квадратичные функции растут намного быстрее линейных. Таким образом, если у нас есть один алгоритм, который состоит из $n^2 + n$ шагов, и другой алгоритм, который выполняется за n^2 шагов, с ростом n разность между ними будет становиться все менее и менее существенной. Оба алгоритма выполняются близко к n^2 шагов.

Нотация «О большое»

Утверждение о том, что алгоритм выполняется более или менее за n^2 шагов, обеспечивает разумный баланс между нужной точностью и компактностью (и случайностью). Подобные отношения «более или менее» более точно выражаются в нотации «О большое» (О — сокращение от Order, то есть порядок). Говорят, что конкретный алгоритм «находится в зависимости “О большое” от n^2» или «имеет сложность $O(n^2)$», если в худшем случае он выполняется более или менее за n^2 шагов при больших n. Техническое определение гласит, что функция $f(x)$ находится

в зависимости «О большое» от функции $g(x)$, если существует некоторая константа M, при которой абсолютное значение $f(x)$ всегда меньше $M \times g(x)$ для всех достаточно больших значений x.

В случае сортировки методом вставки, когда мы рассматриваем выражение для максимального количества шагов, необходимых для выполнения алгоритма, выясняется, что оно является суммой двух слагаемых: одно кратно n^2, а другое — n. Как упоминалось выше, слагаемое, кратное n, становится все менее значимым с ростом n, слагаемое n^2 станет единственным, которое будет на что-то влиять. Таким образом, в худшем случае сортировка методом вставки имеет сложность $O(n^2)$ («находится в зависимости "О большое" от n^2»).

Погоня за эффективностью алгоритмов состоит из поиска алгоритмов, время выполнения которых находится в зависимости «О большое» от все меньших функций. Если вам удастся найти возможность изменить сортировку методом вставки так, что она выполняется со сложностью $O(n^{1,5})$ вместо $O(n^2)$, то это станет огромным достижением, которое очень сильно отразится на времени выполнения при больших значениях n. Нотация «О большое» может использоваться для оценки не только времени, но и затрат памяти. Некоторые алгоритмы могут добиться повышения скорости за счет хранения больших объемов данных в памяти. Их время выполнения может находиться в зависимости «О большое» от малой функции, тогда как затраты памяти будут находиться в зависимости «О большое» от большой функции. В зависимости от обстоятельств может быть разумным увеличить скорость за счет повышения затрат памяти или освободить память за счет некоторого снижения скорости. В этой главе мы сосредоточимся на достижении скорости и проектировании алгоритмов со временем выполнения, находящихся в отношении «О большое» с наименьшими возможными функциями без учета затрат памяти.

После того как вы узнали, что сортировка методом вставки обладает временем выполнения $O(n^2)$, естественно спросить: на какой уровень улучшения можно надеяться? Удастся ли найти какой-нибудь сверхъестественный алгоритм, который сможет отсортировать любой возможный список менее чем за десять шагов? Нет. Любому алгоритму сортировки потребуется не менее n шагов, поскольку он должен будет поочередно рассмотреть каждый элемент списка — каждый из n элементов. Таким образом, любой алгоритм сортировки будет иметь сложность как минимум $O(n)$. Добиться результата лучше $O(n)$ невозможно, но можно ли улучшить сложность $O(n^2)$ сортировки методом вставки? Да, можно. Сейчас мы рассмотрим алгоритм со сложностью $O(n \log(n))$ — значительное улучшение по сравнению с сортировкой методом вставки.

Сортировка слиянием

Алгоритм *сортировки слиянием* намного быстрее сортировки методом вставки. Как и сортировка методом вставки, сортировка слиянием состоит из двух частей: одна объединяет два списка (слияние), а другая многократно использует слияние для выполнения фактической сортировки.

Прежде чем переходить к сортировке, рассмотрим сам процесс слияния. Допустим, имеются две полки с папками, каждая из которых отсортирована, но содержимое одной полки не сравнивалось с содержимым другой. Требуется объединить содержимое полок на одной итоговой полке, которая будет полностью отсортирована. Назовем эту операцию *слиянием* двух отсортированных полок. Как подойти к решению этой задачи?

И снова стоит подумать, как бы вы подошли к решению этой задачи с настоящими полками, прежде чем запускать Python и браться за программирование. Представьте, что вы стоите перед тремя полками: двумя заполненными и отсортированными, содержимое которых требуется объединить, и третьей пустой, на которую будут вставляться папки и которая в конечном счете будет содержать все папки с двух исходных полок. Для удобства будем называть две исходные полки левой и правой (предполагается, что они стоят слева и справа от вас).

Слияние

Чтобы выполнить слияние, возьмем первую папку с каждой из двух исходных полок: с левой полки в левую руку, а с правой полки — в правую. Папка с меньшим номером ставится на новую полку первой. Чтобы определить вторую папку для новой полки, снова возьмем первые папки с левой и правой полок, сравним их и вставим меньшую на последнюю позицию новой полки. Когда левая или правая полка опустеет, мы берем оставшиеся файлы на непустой полке и размещаем их все вместе в конце новой полки. После этого новая полка будет содержать все файлы с левой и правой полки, отсортированные по порядку. В результате содержимое двух исходных полок будет успешно объединено.

В коде Python переменные `left` и `right` будут представлять исходные отсортированные полки. Мы также определим список `newcabinet`, который изначально пуст, а после выполнения будет содержать полный упорядоченный набор всех элементов `left` и `right`.

```
newcabinet = []
```

Определим исходные списки, которым будут присвоены имена `left` и `right`:

```
left = [1,3,4,4,5,7,8,9]
right = [2,4,6,7,8,8,10,12,13,14]
```

Для сравнения соответствующих первых элементов списков `left` и `right` будут использоваться следующие команды `if`, которые можно будет выполнить только после заполнения секций `--...--`:

```
if left[0] > right[0]:
 --...--
elif left[0] <= right[0]:
 --...--
```

Если первый элемент списка `left` меньше первого элемента списка `right`, то нужно извлечь этот элемент из левого списка и вставить его в `newcabinet`, и наоборот. Для выполнения этой операции можно воспользоваться встроенной функцией Python `pop()`, которая вставляется в команду `if` следующим образом:

```
if left[0] > right[0]:
    to_insert = right.pop(0)
    newcabinet.append(to_insert)
elif left[0] <= right[0]:
    to_insert = left.pop(0)
    newcabinet.append(to_insert)
```

Этот процесс — проверка первых элементов списков `left` и `right` и извлечение подходящего элемента в новый список — должен продолжаться, пока на обеих полках стоит хотя бы одна папка. По этой причине команды `if` включаются в цикл `while`, который проверяет минимальную длину списков `left` и `right`. Пока они содержат хотя бы один элемент, процесс продолжается:

```
while(min(len(left),len(right)) > 0):
    if left[0] > right[0]:
        to_insert = right.pop(0)
        newcabinet.append(to_insert)
    elif left[0] <= right[0]:
        to_insert = left.pop(0)
        newcabinet.append(to_insert)
```

Цикл `while` продолжает выполняться, пока в списках `left` или `right` не закончатся элементы для вставки. В этот момент, если список `left` пуст, все файлы из `right`

вставляются в конец новой полки в текущем порядке, и наоборот. Итоговая вставка может выполняться следующим образом:

```
if(len(left) > 0):
    for i in left:
        newcabinet.append(i)

if(len(right) > 0):
    for i in right:
        newcabinet.append(i)
```

Наконец, мы объединяем все эти фрагменты в итоговый алгоритм слияния на Python (листинг 4.4).

Листинг 4.4. Алгоритм слияния двух отсортированных списков

```
def merging(left,right):
    newcabinet = []
    while(min(len(left),len(right)) > 0):
        if left[0] > right[0]:
            to_insert = right.pop(0)
            newcabinet.append(to_insert)
        elif left[0] <= right[0]:
            to_insert = left.pop(0)
            newcabinet.append(to_insert)
    if(len(left) > 0):
        for i in left:
            newcabinet.append(i)
    if(len(right)>0):
        for i in right:
            newcabinet.append(i)
    return(newcabinet)

left = [1,3,4,4,5,7,8,9]
right = [2,4,6,7,8,8,10,12,13,14]

newcab=merging(left,right)
```

Код в листинге 4.4 создает `newcab` — список, содержащий все элементы `left` и `right`, объединенные и упорядоченные. Чтобы убедиться в том, что функция слияния сработала, выполните команду `print(newcab)`.

От слияния к сортировке

Когда вы знаете, как выполнять слияние, сортировка слиянием становится доступной для понимания. Начнем с создания простой функции сортировки слиянием, которая работает только со списками из двух и менее элементов. Одноэлементный список

уже отсортирован, и если он передается на вход функции сортировки слиянием, то должен просто возвращаться в неизменном виде. Если функции сортировки слиянием передается список из двух элементов, то его можно разделить на два одноэлементных списка (которые уже отсортированы по определению) и вызвать функцию слияния для этих одноэлементных списков, чтобы получить итоговый отсортированный список из двух элементов. Следующая функция Python решает эту задачу:

```
import math

def mergesort_two_elements(cabinet):
    newcabinet = []
    if(len(cabinet) == 1):
        newcabinet = cabinet
    else:
        left = cabinet[:math.floor(len(cabinet)/2)]
        right = cabinet[math.floor(len(cabinet)/2):]
        newcabinet = merging(left,right)
    return(newcabinet)
```

В этом коде синтаксис индексирования списков Python используется для разбиения сортируемой полки на две: левую и правую. В строках, определяющих `left` и `right`, выражения `:math.floor(len(cabinet)/2)` и `math.floor(len(cabinet)/2):` обозначают всю первую и всю вторую половину исходной полки соответственно. Эту функцию можно вызвать с любым списком, содержащим один или два элемента, — например, `mergesort_two_elements([3,1])` — и убедиться в том, что она успешно возвращает отсортированный результат.

Теперь напишем функцию, которая сортирует список из четырех элементов. Если разбить список из четырех элементов на два подсписка, то каждый подсписок будет содержать два элемента. Для объединения этих списков можно воспользоваться алгоритмом слияния. Однако вспомните, что он предназначен для объединения двух уже отсортированных списков. Эти два списка могут не быть отсортированными, так что применение алгоритма слияния не обеспечит их успешной сортировки. Но каждый из подсписков содержит всего два элемента, а мы только что написали функцию, которая выполняет сортировку слиянием со списками, содержащими два элемента. Разобьем список из четырех элементов на два подсписка, вызовем функцию сортировки слиянием, которая работает с двумя двухэлементными списками, для каждого из этих подсписков, а затем выполним слияние двух отсортированных списков, получая отсортированный результат из четырех элементов. Эту задачу решает следующая функция Python:

```
def mergesort_four_elements(cabinet):
    newcabinet = []
    if(len(cabinet) == 1):
```

```
        newcabinet = cabinet
    else:
        left = mergesort_two_elements(cabinet[:math.floor(len(cabinet)/2)])
        right = mergesort_two_elements(cabinet[math.floor(len(cabinet)/2):])
        newcabinet = merging(left,right)
    return(newcabinet)

cabinet = [2,6,4,1]
newcabinet = mergesort_four_elements(cabinet)
```

Можно и дальше продолжать писать такие функции, работающие со списками все большего размера. Но настоящий прорыв наступает, когда вы осознаете, что весь процесс можно свернуть с помощью рекурсии. Возьмем функцию из листинга 4.5 и сравним ее с предыдущей функцией `mergesort_four_elements()`.

Листинг 4.5. Реализация функции сортировки слиянием

```
def mergesort(cabinet):
    newcabinet = []
    if(len(cabinet) == 1):
        newcabinet = cabinet
    else:
    ❶ left = mergesort(cabinet[:math.floor(len(cabinet)/2)])
    ❷ right = mergesort(cabinet[math.floor(len(cabinet)/2):])
        newcabinet = merging(left,right)
    return(newcabinet)
```

Как видите, функция почти идентична функции `mergesort_four_elements()`. Принципиальные отличия заключаются в том, что при создании отсортированных списков `left` и `right` она не вызывает другую функцию, работающую с меньшими списками. Вместо этого она вызывает сама себя для меньшего списка ❶ ❷. Сортировка слиянием относится к категории алгоритмов *«разделяй и властвуй»*. Мы начинаем с большого несортированного списка. Последовательно разбиваем его на фрагменты все меньшего размера, пока не получим заведомо отсортированные списки из одного элемента, после чего начинаем выполнять их слияние, пока не вернемся к одному большому отсортированному списку. Вы можете вызвать функцию сортировки слиянием для списка любого размера и убедиться в том, что она работает:

```
cabinet = [4,1,3,2,6,3,18,2,9,7,3,1,2.5,-9]
newcabinet = mergesort(cabinet)
print(newcabinet)
```

В результате объединения всего кода сортировки слиянием будет получен список 4.6.

Листинг 4.6. Полный код сортировки слиянием

```
def merging(left,right):
    newcabinet = []
    while(min(len(left),len(right)) > 0):
        if left[0] > right[0]:
            to_insert = right.pop(0)
            newcabinet.append(to_insert)
        elif left[0] <= right[0]:
            to_insert = left.pop(0)
            newcabinet.append(to_insert)
    if(len(left) > 0):
        for i in left:
            newcabinet.append(i)
    if(len(right) > 0):
        for i in right:
            newcabinet.append(i)
    return(newcabinet)

import math

def mergesort(cabinet):
    newcabinet = []
    if(len(cabinet) == 1):
        newcabinet=cabinet
    else:
        left = mergesort(cabinet[:math.floor(len(cabinet)/2)])
        right = mergesort(cabinet[math.floor(len(cabinet)/2):])
        newcabinet = merging(left,right)
    return(newcabinet)

cabinet = [4,1,3,2,6,3,18,2,9,7,3,1,2.5,-9]
newcabinet=mergesort(cabinet)
```

Можно добавить в код сортировки слиянием счетчик шагов и сравнить результаты с алгоритмом сортировки слиянием. Процесс сортировки слиянием состоит из последовательного разбиения исходного списка на подсписки и последующего слияния подсписков с сохранением порядка сортировки. Каждый раз, когда список разбивается надвое, полученные списки имеют половинную длину. Количество разбиений списка длины n пополам, прежде чем один подсписок будет содержать только один элемент, составляет около $\log(n)$ (логарифм по основанию 2), а количество сравнений при каждом слиянии не превышает n. Таким образом, n или менее сравнений $\log(n)$ разбиений означают, что сортировка слиянием имеет сложность $O(n \times \log(n))$, что на первый взгляд не впечатляет, но в действительности этот алгоритм один из лучших. Собственно, при вызове встроенной функции сортировки Python:

```
print(sorted(cabinet))
```

во внутренней реализации используется гибридная версия сортировки слиянием и сортировки методом вставки. Изучив сортировку слиянием и сортировку методом вставки, вы достигли уровня самого быстрого алгоритма сортировки, который смогли создать теоретики, — алгоритма, ежедневно используемого миллионы раз в самых разнообразных приложениях.

Спящая сортировка

В 2011 году анонимный участник интернет-форума 4chan предложил и предоставил код алгоритма сортировки, который до этого нигде не публиковался. Этот алгоритм получил название *спящей сортировки (sleep sort)*.

Спящая сортировка не разрабатывалась для моделирования каких-либо реальных ситуаций наподобие вставки папок на полку шкафа. Если вам нужна аналогия, то можно рассмотреть задачу распределения мест на спасательных шлюпках тонущего «Титаника». Детям и молодежи отдается приоритет, а люди зрелого возраста пытаются занять оставшиеся места. Если объявить: «Сначала на шлюпки допускаются молодые, а потом старые», возникнет хаос, так как все пассажиры начнут сравнивать свой возраст — они столкнутся с трудной задачей сортировки в неразберихе тонущего корабля.

Подход спящей сортировки к распределению мест выглядел бы примерно так. Вы объявляете: «Пожалуйста, оставайтесь на месте и считайте по порядку: 1, 2, 3... Как только вы досчитаете до своего текущего возраста, выходите вперед и идите занимать место на шлюпке». Восьмилетние пассажиры закончат считать на секунду раньше девятилетних, получат секунду форы и смогут занять место на шлюпке раньше тех, кому исполнилось девять. Восьми- и девятилетние окажутся на лодках раньше десятилетних и т. д. Мы рассчитываем на то, что люди способны выждать время, пропорциональное метрике сортировки, и в дальнейшем занять свое место на шлюпке, после чего сортировка происходит без каких-либо усилий и без прямых сравнений отдельных элементов.

Процесс распределения мест демонстрирует идею спящей сортировки: каждый элемент получает возможность занять свое место, но только после паузы, пропорциональной метрике сортировки. На жаргоне программистов такие паузы называются *сном*, отсюда и название; они могут быть реализованы в большинстве языков.

На Python спящая сортировка может быть реализована так, как показано ниже. Мы импортируем модуль `threading`, который позволяет создавать разные программные потоки (компьютерные процессы), чтобы каждый элемент мог сделать

паузу, а затем вставить себя в итоговый список. Вдобавок импортируется модуль `time.sleep` для перевода программных потоков в «спящее» состояние на соответствующий промежуток времени.

```
import threading
from time import sleep

def sleep_sort(i):
    sleep(i)
    global sortedlist
    sortedlist.append(i)
    return(i)

items = [2, 4, 5, 2, 1, 7]
sortedlist = []
ignore_result = [threading.Thread(target = sleep_sort, args = (i,)).start() \
for i in items]
```

Отсортированный список будет храниться в переменной `sortedlist`, а на список `ignore_result` не обращайте внимания. Вы увидите, что одно из преимуществ спящей сортировки заключается в том, что она компактно записывается на Python. Кроме того, будет интересно вывести переменную `sortedlist` до завершения сортировки (в данном случае около 7 секунд), поскольку в зависимости от того, когда именно будет выполняться команда `print`, вы увидите разные списки. Тем не менее у этой сортировки есть и серьезные недостатки. Один из них заключается в том, что выполнение невозможно приостановить на отрицательное время, так что спящая сортировка не позволяет сортировать списки с отрицательными числами. Другой недостаток — значительная зависимость выполнения сортировки от выбросов; если вставить в список число 1000, то завершения выполнения придется ожидать не менее 1000 секунд. Наконец, числа, близкие друг к другу, могут быть вставлены в неверном порядке, если параллельное выполнение потоков было неидеальным. Наконец, так как спящая сортировка использует многопоточное выполнение, она не сможет (эффективно) работать на оборудовании или в программных средах, которые не поддерживают (эффективно) многопоточность.

Если бы нам потребовалось выразить время выполнения спящей сортировки в нотации «О большое», оно составит $O(max(list))$. В отличие от времени выполнения всех известных алгоритмов сортировки, его время выполнения зависит не только от размера списка, но и от размера элементов списка. Все это делает спящую сортировку слишком ненадежной, поскольку можно быть уверенным в ее быстродействии только с конкретными списками — даже короткий список с очень большими элементами может потребовать слишком много времени.

Никаких причин для практического применения спящей сортировки быть не может, даже на тонущем корабле. Я включил ее в книгу по нескольким причинам. Во-первых, из-за того что она так сильно отличается от других популярных алгоритмов сортировки, это напоминает нам, что даже в самых изученных и статичных областях исследования остается место для творческого подхода и инноваций, и это открывает новую точку зрения на то, что казалось узкой и специализированной темой. Во-вторых, поскольку этот алгоритм был разработан и опубликован анонимно, а его создатель, вероятно, не принадлежит к числу известных исследователей, это напоминает нам, что великие мысли и гении могут встречаться не только в новомодных университетах и ведущих фирмах, но и среди нас, непризнанных и не пользующихся авторитетом. В-третьих, алгоритм представляет замечательное новое поколение алгоритмов, предназначенных для использования на компьютере, — в том смысле, что такие алгоритмы не моделируют нечто такое, что можно сделать руками, как старые алгоритмы, а на фундаментальном уровне зависят от уникальных возможностей компьютеров (в данном случае приостановка выполнения и многопоточность). В-четвертых, компьютерные идеи, на которых базируется алгоритм (приостановка и многопоточность) чрезвычайно полезны и заслуживают место в инструментарии любого специалиста по алгоритмам как возможная основа для разработки других алгоритмов. Наконец, я питаю необъяснимую слабость к этому алгоритму — то ли из-за присущего ему странного проявления творческого подхода, то ли потому, что мне нравится механизм самоорганизующегося упорядочения элементов.

От сортировки к поиску

Поиск, как и сортировка, играет фундаментальную роль во многих задачах из информатики (и в жизни вообще). Допустим, вы хотите найти имя в телефонном справочнике или (раз уж мы живем в XXI веке) обратиться к базе данных и найти интересующую вас запись.

Поиск часто становится всего лишь естественным следствием сортировки. Другими словами, после сортировки списка поиск выполняется тривиально — все сложности часто связаны с сортировкой.

Бинарный поиск

Бинарный поиск — простой и эффективный метод поиска элементов в отсортированном списке. Он немного напоминает игру «угадай число». Допустим, кто-то задумал число от 1 до 100, и вы пытаетесь его отгадать. При первой попытке вы называете 50. Это неправильный ответ, но вам дают подсказку: загаданное

число меньше 50. Вы называете число 49. Снова неправильно — загаданное число меньше 49. Вы называете 48, потом 47 и т. д., пока не придете к правильному ответу. Но это займет слишком много времени — если было загадано число 1, то вам потребуется 50 предположений, а это слишком много, поскольку возможны всего 100 вариантов.

Правильнее будет увеличить расстояние при следующем предположении. Если число 50 слишком велико, то подумайте, что можно узнать, если предположить 40 вместо 49. Если число 40 слишком мало, то мы исключаем 39 возможностей (1–39), и ответ гарантированно можно будет угадать за 9 предположений (41–49). Если число 40 слишком велико, то исключаются 9 возможностей (41–49), и ответ гарантированно можно будет указать за 39 предположений (1–39). Таким образом, выбор числа 40 сокращает количество вариантов с 49 (1–49) до 39 (1–39) в худшем случае. С другой стороны, выбор числа 49 сокращает количество вариантов с 49 (1–49) до 48 (1–48) в худшем случае. Очевидно, выбор числа 40 является более эффективной стратегией поиска, чем выбор 49.

Как выясняется, лучшая стратегия поиска основана на предположении ровно на середине диапазона остающихся возможностей. Если вы поступите подобным образом, а потом проверите, было ли ваше предположение слишком большим или малым, то всегда сможете исключить половину оставшихся вариантов. Исключая половину вариантов при каждом предположении, можно найти нужное значение достаточно быстро ($O(\log(n))$ для тех, кто предпочитает более точные оценки. Например, в списке из 1000 элементов для нахождения любого элемента методом бинарного поиска хватит всего десяти предположений. Если разрешить 20 предположений, то можно правильно определить позицию элемента в списке, содержащем более миллиона элементов. Кстати говоря, именно так пишутся приложения-«угадайки», способные правильно «прочитать ваши мысли» всего за 20 вопросов.

Чтобы реализовать этот алгоритм на Python, для начала следует определить верхнюю и нижнюю границы для позиции, которую может занимать папка на полке. Нижняя граница равна 0, а верхняя определяется длиной полки:

```
sorted_cabinet = [1,2,3,4,5]
upperbound = len(sorted_cabinet)
lowerbound = 0
```

Для начала предположим, что папка находится в середине полки. Мы импортируем *математическую* библиотеку Python для использования функции `floor()`, преобразующей дробные числа в целые. Напомню, что предположение на середине диапазона обеспечивает максимально возможный объем информации:

```
import math
guess = math.floor(len(sorted_cabinet)/2)
```

Затем необходимо проверить, было ли предположение слишком большим или малым. В зависимости от результата проверки будут выполняться разные действия. Для искомого значения будет использоваться переменная `looking_for`:

```
if(sorted_cabinet[guess] > looking_for):
    --...--
if(sorted_cabinet[guess] < looking_for):
    --...--
```

Если номер папки на полке слишком велик, то наше предположение становится новой верхней границей, так как искать выше смысла нет. После этого выдается другое, меньшее предположение — точнее, оно находится на середине между текущим предположением и нижней границей:

```
looking_for = 3
if(sorted_cabinet[guess] > looking_for):
    upperbound = guess
    guess = math.floor((guess + lowerbound)/2)
```

Аналогичный процесс применяется в том случае, если номер папки на полке слишком мал:

```
if(sorted_cabinet[guess] < looking_for):
    lowerbound = guess
    guess = math.floor((guess + upperbound)/2)
```

Наконец, все составляющие можно объединить в функцию `binarysearch()`. Функция содержит цикл `while`, который выполняется до пор, пока не будет найден искомый элемент (листинг 4.7).

Листинг 4.7. Реализация бинарного поиска

```
import math
sortedcabinet = [1,2,3,4,5,6,7,8,9,10]

def binarysearch(sorted_cabinet,looking_for):
    guess = math.floor(len(sorted_cabinet)/2)
    upperbound = len(sorted_cabinet)
    lowerbound = 0
    while(abs(sorted_cabinet[guess] - looking_for) > 0.0001):
        if(sorted_cabinet[guess] > looking_for):
            upperbound = guess
```

```
            guess = math.floor((guess + lowerbound)/2)
        if(sorted_cabinet[guess] < looking_for):
            lowerbound = guess
            guess = math.floor((guess + upperbound)/2)
    return(guess)

print(binarysearch(sortedcabinet,8))
```

Итоговый вывод этого кода сообщает нам, что число 8 находится в позиции 7 списка `sorted_cabinet`. Это правильный результат (помните, что индексы списков Python начинаются с 0). Стратегия предположений, исключающих половину оставшихся вариантов, оказывается полезной во многих областях. Например, она лежит в основе самой эффективной стратегии (в среднем) в некогда популярной настольной игре «угадай кто». К тому же она является лучшим (теоретически) способом поиска слов в большом незнакомом словаре.

Применение бинарного поиска

Помимо игр-угадаек и поиска слов, бинарный поиск используется в ряде других областей. Например, идея бинарного поиска может использоваться при отладке кода. Предположим, вы написали код, который не работает, но не уверены в том, в какой его части кроется ошибка. Для поиска проблемы можно воспользоваться стратегией бинарного поиска. Код делится надвое, и обе половины выполняются независимо. Если какая-то половина выполняется неправильно, значит, проблема кроется именно в ней. Проблемная часть снова разбивается надвое, а каждая половина проверяется отдельно, чтобы сократить количество вариантов, пока не будет найдена строка кода с ошибкой. Аналогичная идея реализована в популярной программной системе управления версиями Git в форме команды `git bisect` (хотя `git bisect` перебирает версии кода, разделенные во времени, вместо строк кода одной версии).

Другое применение бинарного поиска — инвертирование математических функций. Представьте, что вам нужно написать функцию для вычисления арксинуса (или обратного синуса) заданного числа. Всего в нескольких строках можно написать функцию, которая вызывает нашу функцию `binarysearch()` для получения правильного ответа. Для начала необходимо определить *область определения*, то есть значения, по которым будет проводиться поиск для нахождения конкретного значения арксинуса. Функция синуса является периодической; она принимает все возможные значения в диапазоне от `-π/2` до `π/2`, а следовательно, область определения будет состоять из всех чисел между этими крайними точками. Затем для всех значений области определения вычисляются значения синуса. Функция

`binarysearch()` используется для нахождения позиции числа, синус которого равен заданному числу, и возвращает значение предметной области с заданным индексом:

```
def inverse_sin(number):
    domain = [x * math.pi/10000 - math.pi/2 for x in list(range(0,10000))]
    the_range = [math.sin(x) for x in domain]
    result = domain[binarysearch(the_range,number)]
    return(result)
```

Попробуйте выполнить `inverse_sin(0.9)` и убедитесь в том, что функция вернет правильный ответ: около `1.12`.

Это не единственный способ инвертирования функций. Некоторые функции могут инвертироваться посредством алгебраических преобразований. Тем не менее для многих функций алгебраические преобразования оказываются слишком сложными и даже невозможными. С другой стороны, метод бинарного поиска, представленный здесь, будет работать с любой функцией, да еще и с молниеносным временем выполнения $O(\log(n))$.

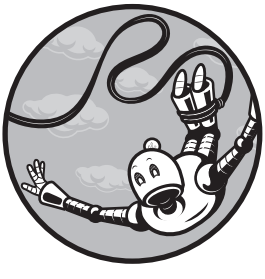
Резюме

Задачи сортировки и поиска могут показаться чем-то обыденным, словно вас оторвали от кругосветного круиза для посещения семинара по методике складывания отглаженного белья. Возможно, это так, но помните, что если вы умеете эффективно складывать белье, то сможете упаковать больше снаряжения для подъема на Килиманджаро. Алгоритмы сортировки и поиска — ваши подручные, которые помогают открывать более новые и интересные возможности. Кроме того, эти алгоритмы заслуживают вашего внимания из-за своей фундаментальной роли и популярности и еще пригодятся на протяжении вашей жизни. В этой главе мы рассмотрели некоторые фундаментальные и интересные алгоритмы сортировки, а также метод бинарного поиска. Помимо этого, мы изучили способы сравнения алгоритмов и нотацию «О большое».

В следующей главе мы обратимся к нескольким способам применения алгоритмов из области чистой математики. Вы научитесь использовать алгоритмы для исследования мира математики и узнаете, как мир математики помогает нам разобраться в нашем собственном мире.

5

Чистая математика



Благодаря своему формальному характеру алгоритмы естественно подходят для применений в области математики. В этой главе мы исследуем некоторые алгоритмы, применяемые в области чистой математики, и посмотрим, как математические идеи помогают улучшать наши алгоритмы. Начнем с обсуждения непрерывных дробей — несложной, на первый взгляд, темы, которая поможет найти порядок в хаосе. Мы рассмотрим квадратные корни — тему более прозаическую, но, возможно, более практичную. Наконец мы перейдем к обсуждению случайности, включая математическое обоснование случайности и некоторые важные алгоритмы для генерирования случайных чисел.

Непрерывные дроби

В 1597 году великий ученый Иоганн Кеплер написал, что он считает двумя «великими сокровищами» геометрии теорему Пифагора и число, которое с тех пор получило название «золотое сечение». Оно часто обозначается греческой буквой φ и равно приблизительно 1,618, и Кеплер был лишь одним из десятков великих мыслителей, которых оно приводило в восторг. Число φ, как и π с другими знаменитыми константами (такими как основание натуральных логарифмов e), нередко появляется в совершенно неожиданных местах. Ученые нередко сталкивались с золотым

сечением, обозначаемым числом φ, в природе и скрупулезно документировали его проявления в искусстве — например, в снабженной аннотациями версии «Венеры с зеркалом» Диего Веласкеса, показанной на рис. 5.1.

На рис. 5.1 энтузиаст добавил разметку, которая показывает, что соотношения некоторых длин (таких как b/a и d/c) равны φ. Подобные проявления золотого сечения встречаются во многих шедеврах живописи.

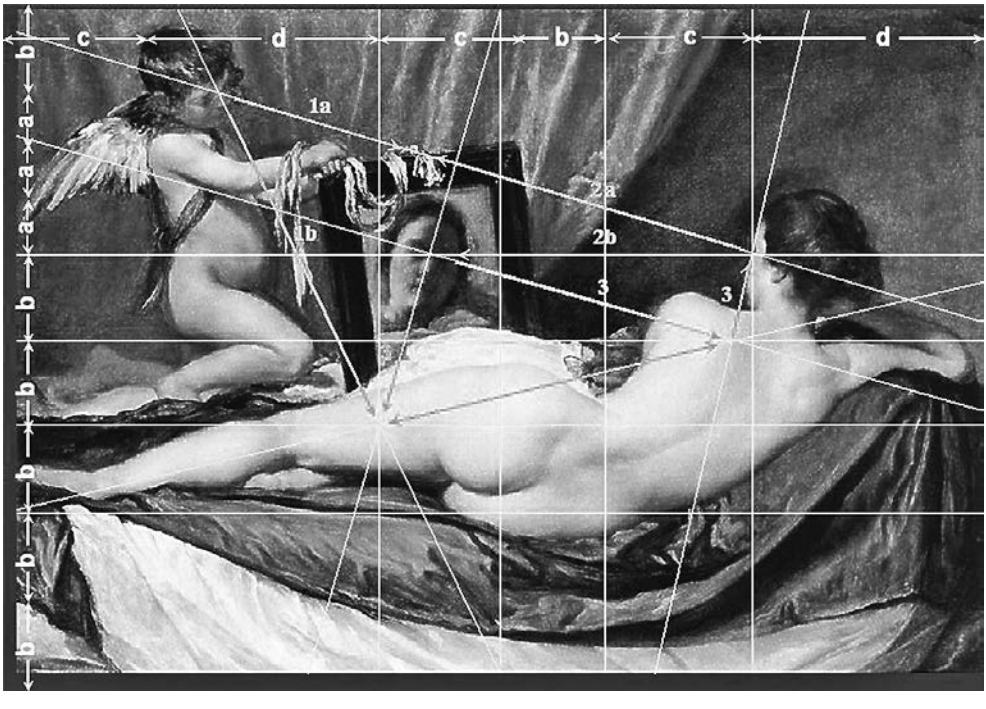


Рис. 5.1. Золотое сечение на картине «Венера с зеркалом» (https://commons.wikimedia.org/wiki/File:DV_The_Toilet_of_Venus_Gr.jpg)

Компактное представление числа φ

Точное значение числа φ на удивление трудно представить. Можно сказать, что оно равно приблизительно 1,61803399... Многоточие можно считать своего рода обманом; оно означает, что далее идут другие цифры (на самом деле бесконечное количество цифр), но я не привожу их, так что вы не знаете точное значение φ.

Для некоторых чисел с бесконечными дробными разложениями возможно точное представление. Например, число 0,11111... равно 1/9 — здесь дробь предоставляет простой способ выражения точного значения дроби, продолжающейся до бесконечности. Даже если точное представление неизвестно, вы можете заметить закономерность с повторяющимися единицами в 0,1111..., а следовательно, понять точное значение. К сожалению, «золотое сечение» относится к так называемым *иррациональным числам*; это означает, что не существует двух целых чисел x и y, для которых φ равно x/y. Более того, никто еще не смог выявить какую-либо закономерность в его последовательности цифр.

Итак, имеется бесконечное десятичное разложение без очевидных закономерностей, которое не имеет дробного представления. Может показаться, что четкое выражение точного значения φ невозможно. Но когда вы больше узнаете об этом числе, появляется возможность четкого и компактного его выражения. Одна из особенностей φ, о которых необходимо знать, — это число является решением следующего уравнения:

$$\varphi^2 - \varphi - 1 = 0$$

Казалось бы, точное значение числа можно было бы выразить как решение уравнения, записанного над этим абзацем. Конечно, такая формулировка компактна и технически точна, но означает, что уравнение придется как-то решить. Кроме того, описание ничего не сообщает о том, какой будет 200-я или 500-я цифра в разложении φ.

Разделив обе части уравнения на φ, получаем следующий результат:

$$\varphi - 1 - \frac{1}{\varphi} = 0.$$

После перестановки получаем:

$$\varphi = 1 + \frac{1}{\varphi}.$$

А теперь представим странную подстановку этого уравнения внутри самого себя:

$$\varphi = 1 + \frac{1}{\varphi} = 1 + \frac{1}{1 + \frac{1}{\varphi}}.$$

Здесь φ в правой стороне переписывается в виде 1 + 1/φ. Ту же подстановку можно выполнить снова; почему нет?

$$\varphi = 1 + \frac{1}{\varphi} = 1 + \frac{1}{1 + \frac{1}{\varphi}} = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\varphi}}}.$$

Подстановку можно выполнять сколько угодно раз без каких-либо ограничений. При этом φ все дальше продвигается на более низкие уровни, в угол растущей дроби. В листинге 5.1 приведено выражение φ с семью уровнями.

Листинг 5.1. Непрерывная дробь с семью уровнями, выражающая значение φ

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\varphi}}}}}}}.$$

Этот процесс можно продолжать до бесконечности. Результат показан в листинге 5.2.

Листинг 5.2. Непрерывная дробь, выражающая значение φ, продолжающаяся до бесконечности

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}}}}}.$$

Теоретически после бесконечной последовательности 1, знаков + и дробных черт, представленной многоточием, можно вставить φ в листинг 5.2 по аналогии с тем, как это сделано в правом нижнем углу листинга 5.1. Но мы никогда не доберемся до конца последовательности единиц (поскольку она продолжается до бесконечности), поэтому о значении φ, вложенном в правой части, можно полностью забыть.

Подробнее о непрерывных дробях

Только что приведенные выражения называются *непрерывными дробями*. Такая дробь состоит из сумм и дробей, вложенных на многих уровнях. Непрерывные дроби могут быть как конечными, как дробь из листинга 5.1, завершавшаяся после семи уровней, так и бесконечными, как в листинге 5.2. Непрерывные дроби особенно полезны для наших целей, поскольку позволяют выразить точное значение φ, не вырубая целые леса для производства достаточного количества бумаги. Математики иногда используют специальную, более компактную запись для выражения непрерывной дроби в одной пустой строке. Вместо того чтобы записывать все дробные черты в непрерывной дроби, можно воспользоваться квадратными скобками (`[ ]`) для обозначения того, что мы работаем с непрерывной дробью, и отделить двоеточием «отдельную» цифру от цифр, стоящих вместе в дроби. Такой записью можно записать непрерывную дробь для φ в следующем виде:

```
phi  = [1; 1,1,1,1 . . .]
```

В этом случае многоточие уже не приводит к потере информации, так как в непрерывной дроби для φ проявляется явная закономерность: она состоит только из 1, поэтому мы точно знаем ее 100-й или 1000-й элемент. Это один из тех случаев, когда математика предоставляет возможность, которая на первый взгляд кажется волшебной: способ компактной записи числа, которое в нашем представлении является бесконечным и не имеет закономерностей, без словесного описания. Однако φ — не единственная непрерывная дробь. Возможен и другой вариант записи непрерывной дроби:

```
mysterynumber = [2; 1,2,1,1,4,1,1,6,1,1,8, . . .]
```

В данном случае после нескольких первых цифр обнаруживается простая закономерность: пары единиц, чередующиеся с возрастающими четными числами. Следующими значениями будут 1, 1, 10, 1, 1, 12 и т. д. Начало этой непрерывной дроби можно записать в более традиционном стиле:

$$\text{mysterynumber} = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}}}}}.$$

На самом деле это загадочное число есть не что иное, как наш старый знакомый — число *e*, основание натуральных логарифмов! Константа *e*, как и φ, и другие иррациональные числа, имеет бесконечное разложение без очевидной закономерности и не может быть представлена конечной дробью. Похоже, точно выразить ее числовое значение невозможно. Но используя новую концепцию непрерывных дробей и новую компактную запись, мы можем записать эти вроде бы непреодолимые числа в одну строку.

Кроме того, существуют несколько интересных способов использования непрерывных дробей для представления числа π. И это можно назвать победой для сжатия данных, а также победой в бесконечной борьбе между порядком и хаосом: если прежде нам казалось, что интересными для нас числами правит только всепроникающий хаос, то сейчас выясняется, что где-то внизу всегда существовал скрытый порядок.

Наша непрерывная дробь для φ происходит из специального уравнения, которое работает только для φ. Но в действительности для любого числа возможно сгенерировать представление в виде непрерывной дроби.

Алгоритм генерирования непрерывных дробей

Ниже описан алгоритм, который строит разложение произвольного числа в непрерывную дробь.

Проще всего найти разложение в непрерывную дробь для чисел, которые уже являются простыми дробями. Для примера рассмотрим задачу нахождения представления 105/33 в виде непрерывной дроби. Наша цель — выразить это число в следующей форме:

$$\frac{105}{33} = a + \cfrac{1}{b + \cfrac{1}{c + \cfrac{1}{d + \cfrac{1}{e + \cfrac{1}{f + \cfrac{1}{g + \cfrac{1}{\ddots}}}}}}}.$$

где многоточием может стать конечная серия (а не бесконечная). Наш алгоритм сначала сгенерирует *a*, потом *b*, потом *c* и т. д., пока не будет достигнуто последнее слагаемое или пока не будет выполнено условие остановки.

Если интерпретировать пример 105/33 как задачу на деление, а не как дробь, то выясняется, что результат 105/33 равен 3 с остатком 6. Перепишем 105/33 в форме 3 + 6/33:

$$3+\frac{6}{33}=a+\cfrac{1}{b+\cfrac{1}{c+\cfrac{1}{d+\cfrac{1}{e+\cfrac{1}{f+\cfrac{1}{g+\cfrac{1}{\ddots}}}}}}}.$$

Левая и правая части этого уравнения состоят из целого числа (3 и a) и дроби (6/33 и остаток правой части). Заключаем, что целые части равны, Так что $a = 3$. После этого необходимо найти подходящие b, c и т. д., чтобы вся дробная часть выражения давала результат 6/33.

Чтобы найти b, c и т. д., посмотрим, какую задачу предстоит решить после вывода о том, что $a = 3$:

$$\frac{6}{33}=a+\cfrac{1}{b+\cfrac{1}{c+\cfrac{1}{d+\cfrac{1}{e+\cfrac{1}{f+\cfrac{1}{g+\cfrac{1}{\ddots}}}}}}}.$$

Если перейти к обратной дроби в обеих частях этого выражения, то получается следующее уравнение:

$$\frac{33}{6}=b+\cfrac{1}{c+\cfrac{1}{d+\cfrac{1}{e+\cfrac{1}{f+\cfrac{1}{g+\cfrac{1}{h+\cfrac{1}{\ddots}}}}}}}.$$

Требуется определить b и c. Деление можно повторить; 33/6 = 5 с остатком 3, поэтому 33/6 записывается в виде 5 + 3/6:

$$5+\frac{3}{6}=b+\cfrac{1}{c+\cfrac{1}{d+\cfrac{1}{e+\cfrac{1}{f+\cfrac{1}{g+\cfrac{1}{h+\frac{1}{\ddots}}}}}}}.$$

Мы видим, что в обеих частях уравнения присутствуют целое число (5 и b) и дробь (3/6 и остаток правой части). Можно заключить, что целые части равны, так что $b = 5$. Мы определили еще одну букву, и теперь для дальнейшего продвижения необходимо упростить 3/6. Если вы не можете немедленно сказать, что 3/6 — это 1/2, то можно повторить тот же процесс, что и для 6/33: 3/6 выражается в виде обратной дроби 1/(6/3), и мы видим, что результат 6/3 равен 2 с остатком 0. Алгоритм, по которому мы действуем, должен завершиться при достижении нулевого остатка, поэтому можно сделать вывод, что процесс завершен, и вся непрерывная дробь записывается так, как показано в листинге 5.3.

Листинг 5.3. Непрерывная дробь для представления 105/33

$$\frac{105}{33}=3+\cfrac{1}{5+\frac{1}{2}}.$$

Процесс многократного деления двух целых чисел для получения частного и остатка кажется вам знакомым? Так и должно быть. Ведь это тот же процесс, который использовался нами в алгоритме Евклида в главе 2! Мы выполняем те же действия, но сохраняем разные ответы: для алгоритма Евклида последний ненулевой остаток сохранялся как окончательный ответ, а в алгоритме генерирования непрерывных дробей сохраняются все частные (буквы алфавита). Как часто бывает в математике, обнаружилась неожиданная связь — в данном случае между генерированием непрерывной дроби и поиском наибольшего общего делителя.

Ниже описана реализация алгоритма генерирования непрерывной дроби на языке Python.

Допустим, процесс начинается с дроби вида x/y. Сначала мы определяем, какое из чисел x и y больше, а какое меньше:

```
x = 105
y = 33
big = max(x,y)
small = min(x,y)
```

Затем мы берем частное от деления большего числа на меньшее, как было сделано с числом 105/33. Когда обнаруживается, что результат равен 3 с остатком 6, мы заключаем, что 3 является первой составляющей (a) в непрерывной дроби. Возьмем частное и сохраним результат следующим образом:

```
import math
output = []
quotient = math.floor(big/small)
output.append(quotient)
```

В данном случае мы собираемся получить результат в виде набора букв (a, b, c и т. д.), поэтому создаем пустой список `output` и присоединяем к нему первый результат.

Наконец, процесс необходимо повторить, как это было сделано для 33/6. Вспомните, что значение 33 ранее хранилось в переменной `small`, но теперь хранится в `big`, и остатком процесса деления становится новая переменная `small`. Так как остаток всегда меньше делителя, `big` и `small` всегда будут содержать правильные значения. На языке Python перестановка реализуется так:

```
new_small = big % small
big = small
small = new_small
```

В этот момент мы завершили один проход алгоритма, теперь нужно повторить его для следующего набора чисел (33 и 6). Чтобы компактно записать этот процесс, мы поместим его в цикл (листинг 5.4).

Листинг 5.4. Алгоритм выражения дробей в виде непрерывных дробей

```
import math
def continued_fraction(x,y,length_tolerance):
    output = []
    big = max(x,y)
    small = min(x,y)
    while small > 0 and len(output) < length_tolerance:
        quotient = math.floor(big/small)
        output.append(quotient)
        new_small = big % small
        big = small
        small = new_small
    return(output)
```

Программа получает x и y на входе и определяет переменную `length_tolerance`. Не забывайте, что одни непрерывные дроби имеют бесконечную длину, а другие просто очень длинные. Включая в функцию переменную `length_tolerance` , мы можем преждевременно прервать процесс, если вывод становится слишком громоздким, избегая тем самым зацикливания.

Вспомните, что при выполнении алгоритма Евклида использовалось рекурсивное решение. На этот раз вместо рекурсии задействован цикл `while`. Рекурсия хорошо подходила для алгоритма Евклида, поскольку для него необходимо только одно число — окончательный ответ, а здесь мы хотим собрать последовательность чисел в список. Цикл лучше подходит для подобного последовательного сбора данных. Новая функция `continued_fraction` вызывается следующим образом:

```
print(continued_fraction(105,33,10))
```

Вы получите следующий простой вывод:

```
[3,5,2]
```

Как видите, числа совпадают с ключевыми значениями в правой части листинга 5.3.

Возможно, вы захотите проверить, что та или иная непрерывная дробь правильно выражает интересующее вас число. Для этого следует определить функцию `get_number()`, которая преобразует непрерывную дробь в десятичное число (листинг 5.5).

Листинг 5.5. Преобразование непрерывной дроби в представление числа

```
def get_number(continued_fraction):
    index = -1
    number = continued_fraction[index]

    while abs(index) < len(continued_fraction):
        next = continued_fraction[index - 1]
        number = 1/number + next
        index -= 1
    return(number)
```

Не обращайте внимания на подробности реализации этой функции, мы просто используем ее для проверки непрерывных дробей. Чтобы удостовериться в том, что функция работает, выполните команду `get_number([3,5,2])` и убедитесь в том, что на выходе получен результат 3,181818... — другой способ записи 105/33 (исходное число).

От десятичных дробей к непрерывным

А если вместо некоторого x/y на вход алгоритма непрерывных дробей подается дробное число — например, 1,4142135623730951? Необходимо внести ряд изменений, но в целом можно следовать тому же процессу, которому мы следовали для дробей. Помните, что мы хотели найти a, b, c и остальные буквы в выражении следующего типа:

$$1{,}4142135623730951 = a + \cfrac{1}{b + \cfrac{1}{c + \cfrac{1}{d + \cfrac{1}{e + \cfrac{1}{f + \cfrac{1}{g + \frac{1}{\ldots}}}}}}}.$$

Найти a проще простого — это просто целая часть числа. Определим `first_term` (a в нашем уравнении) и `leftover` следующим образом:

```
x = 1.4142135623730951
output = []
first_term = int(x)
leftover = x - int(x)
output.append(first_term)
```

Как и прежде, последовательно вычисляемые результаты сохраняются в списке `output`.

После определения a имеется остаток, для которого необходимо найти представление в виде непрерывной дроби:

$$0{,}4142135623730951 = \cfrac{1}{b + \cfrac{1}{c + \cfrac{1}{d + \cfrac{1}{e + \cfrac{1}{f + \cfrac{1}{g + \frac{1}{\ldots}}}}}}}.$$

И здесь также можно перейти к обратным дробям:

$$\frac{1}{0{,}4142135623730951} = 2{,}4142135623730945 = b + \cfrac{1}{c + \cfrac{1}{d + \cfrac{1}{e + \cfrac{1}{f + \cfrac{1}{g + \cfrac{1}{\ldots}}}}}}.$$

Следующий результат b будет целой частью нового числа — в данном случае 2. После этого процесс повторяется: переход к обратной дроби для дробной части, определение целой части и т. д.

На языке Python каждая итерация может быть записана следующим образом:

```
next_term = math.floor(1/leftover)
leftover = 1/leftover - next_term
output.append(next_term)
```

Весь процесс объединяется в одну функцию, приведенную в листинге 5.6.

Листинг 5.6. Вычисление непрерывных дробей для дробных чисел

```
def continued_fraction_decimal(x,error_tolerance,length_tolerance):
    output = []
    first_term = int(x)
    leftover = x - int(x)
    output.append(first_term)
    error = leftover
    while error > error_tolerance and len(output) <length_tolerance:
        next_term = math.floor(1/leftover)
        leftover = 1/leftover - next_term
        output.append(next_term)
        error = abs(get_number(output) - x)
    return(output)
```

Как и в предыдущем случае, в программу включается порог `length_tolerance`. Добавляется и переменная `error_tolerance`, которая позволяет выйти из алгоритма, если приближение «достаточно близко» к точному ответу. Чтобы узнать, подошли ли мы достаточно близко к решению, мы вычисляем разность между `x` (аппроксимируемым числом) и значением непрерывной дроби, вычисленной на текущий

момент. Для получения этого значения можно воспользоваться той же функцией `get_number()` из листинга 5.5.

Опробовать новую функцию достаточно просто:

```
print(continued_fraction_decimal(1.4142135623730951,0.00001,100))
```

Вы получите следующий результат:

```
[1, 2, 2, 2, 2, 2, 2, 2]
```

Эту непрерывную дробь можно записать следующим образом (с использованием знака приближенного равенства, поскольку непрерывная дробь является аппроксимацией в пределах крошечной погрешности, а у нас нет времени для вычисления каждого элемента бесконечной последовательности):

$$1{,}4142135623730951 \approx 1+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2}}}}}}}$$

Обратите внимание: вся диагональ дробей справа заполнена двойками. Мы нашли первые семь результатов для бесконечной непрерывной дроби, бесконечное разложение которой состоит только из двоек. Разложение непрерывной дроби может быть записано в виде [1,2,2,2,2...]. Это разложение в непрерывную дробь $\sqrt{2}$ — другого иррационального числа, которое не может быть представлено в виде простой дроби, а в цифрах его дробной части нет закономерностей. Тем не менее это число имеет легко запоминающееся представление в виде непрерывной дроби.

От дробей к корням

Если вас интересуют непрерывные дроби, то я бы рекомендовал почитать о математике, которого звали Сриниваса Рамануджан, — за свою короткую жизнь он мысленно путешествовал к пределам, ведущим в бесконечность, и вернулся оттуда с сокровищами, которыми поделился с нами. Помимо непрерывных дробей, Рамануджан интересовался *непрерывными квадратными корнями* (также называемыми

вложенными) — приведу примеры трех квадратных корней с бесконечной вложенностью:

$$x = \sqrt{2+\sqrt{2+\sqrt{2+\ldots}}},$$

$$y = \sqrt{1+2\times\sqrt{1+3\times\sqrt{1+4\times\sqrt{1+\ldots}}}},$$

$$z = \sqrt{1+\sqrt{1+\sqrt{1+\ldots}}}.$$

Оказывается, $x = 2$ (давно известный анонимный результат), $y = 3$ (доказано Рамануджаном), а z — не что иное, как φ, золотое сечение! Попробуйте придумать способ генерирования представлений вложенных корней на Python. Квадратные корни, безусловно, интересны в бесконечных формулах, но, как выясняется, интересны и сами по себе!

Квадратные корни

Мы считаем калькуляторы чем-то обыденным и привычным, но если подумать, что делают эти устройства, то они весьма впечатляют. Наверное, вы помните из уроков геометрии, что синус определяется через дли́ны сторон прямоугольных треугольников — как отношение длины противолежащего катета к длине гипотенузы. Но если это определение синуса, то как на калькуляторе может быть кнопка Sin, которая мгновенно выполняет это вычисление? Может, калькулятор где-то внутри рисует прямоугольный треугольник, достает линейку, измеряет длины сторон и делит их? Аналогичный вопрос можно задать и о квадратных корнях: квадратный корень является операцией, обратной по отношению к возведению в квадрат, и не существует никакой прямолинейной аналитической формулы, которую мог бы использовать калькулятор. Наверное, вы уже догадались: существует алгоритм для быстрого вычисления квадратных корней.

Вавилонский алгоритм

Предположим, вы хотите найти квадратный корень числа x. Как и в любой математической задаче, можно воспользоваться стратегией предположений и проверки. Допустим, вы предполагаете, что квадратный корень из x равен y. Далее вы вычисляете y^2, и если результат равен x, то задача решена (редкий случай одношагового «алгоритма удачной догадки»).

Если гипотеза y отличается от квадратного корня x, то нужно представить следующую догадку, и следующее предположение должно привести нас ближе к истинному значению квадратного корня x. Вавилонский алгоритм позволяет последовательно улучшать предположения с постепенным схождением к правильному ответу. Это простой алгоритм, для которого не требуется ничего, кроме деления и вычисления среднего.

1. Предположить, что квадратный корень из x равен y.
2. Вычислить $z = x / y$.
3. Вычислить среднее для z и y. Среднее становится новой оценкой y, то есть новым предположением для квадратного корня из x.
4. Повторять шаги 2 и 3, пока разность $y^2 - x$ не станет достаточно малой.

Наше описание вавилонского алгоритма состоит из четырех шагов. С другой стороны, математик может выразить весь алгоритм одним уравнением:

$$y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2}.$$

В данном случае используется стандартная математическая практика описания бесконечной последовательности сериями индексов: $(y_1, y_2 \ldots y_n \ldots)$. Если вам известен n-й элемент этой бесконечной последовательности, то $(n + 1)$-й элемент можно вычислить по приведенному выше выражению. Последовательность сходится к $\sqrt{x}$ или, другими словами, $y_\infty = \sqrt{x}$. Предпочитаете ли вы ясность описания из четырех шагов, элегантную компактность уравнения или практичность кода? Это дело вкуса, но в любом случае полезно знать все возможные способы описания алгоритма.

Чтобы понять, почему вавилонский алгоритм работает, можно рассмотреть два простых случая:

- **Если** $y < \sqrt{x}$, то $y^2 < x$. Следовательно, $\frac{x}{y^2} > 1$, а $x \times \frac{x}{y^2} > x$.

  Но заметим, что $x \times \frac{x}{y^2} = \frac{x^2}{y^2} = \left(\frac{x}{y}\right)^2 = z^2$. Таким образом, $z^2 > x$. **Отсюда следует, что** $z > \sqrt{x}$.

- **Если** $y > \sqrt{x}$, то $y^2 > x$. Следовательно, $\frac{x}{y^2} < 1$, а $x \times \frac{x}{y^2} < x$.

  Заметим, что $x \times \frac{x}{y^2} = \frac{x^2}{y^2} = \left(\frac{x}{y}\right)^2 = z^2$. Таким образом, $z^2 < x$. **Отсюда следует, что** $z < \sqrt{x}$.

Эти случаи можно записать более компактно с удалением части текста:

- Если $y < \sqrt{x}$, то $z > \sqrt{x}$.
- Если $y > \sqrt{x}$, то $z < \sqrt{x}$.

Если y является заниженной оценкой для истинного значения $\sqrt{x}$, то оценка z будет завышенной. Если же y является завышенной оценкой для истинного значения $\sqrt{x}$, то оценка z будет заниженной. На шаге 3 вавилонского алгоритма вычисляется среднее завышенной и заниженной оценки истинного значения. Среднее будет выше заниженной оценки, но ниже завышенной, поэтому будет ближе к истинному значению, чем худшая из оценок y или z. Со временем после многих итераций постепенного улучшения предположений вы придете к истинному значению $\sqrt{x}$.

Квадратные корни на языке Python

Вавилонский алгоритм достаточно легко реализуется на Python. Определим функцию, которая получает в аргументах `x`, `y` и переменную `error_tolerance`. Мы создаем цикл `while`, который многократно выполняется, пока погрешность не станет достаточно малой. При каждой итерации цикла `while` вычисляется `z`, значение `y` обновляется средним `y` и `z` (в соответствии с шагами 2 и 3 в описании алгоритма), после чего обновляется погрешность, равная $y^2 - x$. Код функции приведен в листинге 5.7.

Листинг 5.7. Функция для вычисления квадратных корней по вавилонскому алгоритму

```
def square_root(x,y,error_tolerance):
    our_error = error_tolerance * 2
    while(our_error > error_tolerance):
        z = x/y
        y = (y + z)/2
        our_error = y**2 - x
    return y
```

Возможно, вы заметили, что вавилонский алгоритм имеет нечто общее с градиентным подъемом и алгоритмом для задачи аутфилдера. Во всех случаях выполняются небольшие итеративные действия, пока вы не приблизитесь на достаточное расстояние к окончательной цели. Такая структура часто встречаются в алгоритмах.

Функцию для вычисления квадратного корня можно проверить следующим образом:

```
print(square_root(5,1,.000000000000001))
```

На консоль выводится число `2.23606797749979`. Вы можете проверить, что это же число будет выведено методом `math.sqrt()`, стандартным для Python:

```
print(math.sqrt(5))
```

Вы получите точно такой же результат:`2.23606797749979`. Вы успешно написали собственную функцию для вычисления квадратных корней. Если вы когда-нибудь окажетесь на необитаемом острове без интернета, не имея возможности загрузить модули Python (в частности, модуль `math`), то всегда сможете самостоятельно написать такие функции, как `math.sqrt()`. Не забудьте поблагодарить жителей Вавилона за придуманный ими алгоритм.

Генераторы случайных чисел

До этого мы пытались в хаосе найти порядок. Математика хорошо подходит для этой цели, но в данном разделе мы займемся прямо противоположным делом: поиском хаоса в порядке. Иначе говоря, вы узнаете, как алгоритмически генерировать случайность.

Существует постоянная потребность в случайных числах. В видеоиграх используются случайно выбранные числа, чтобы игроки сталкивались с неожиданными положениями и перемещениями персонажей. Правильное функционирование некоторых из самых эффективных методов машинного обучения (включая случайные леса и нейронные сети) в значительной мере зависят от случайного выбора. То же можно сказать о некоторых эффективных статистических методах (например, бутстрэп в статистике), которые используют случайность, чтобы небольшие статические наборы данных в большей степени напоминали хаотичный мир. Корпорации и ученые-теоретики применяют тесты A/B, зависящие от случайного распределения элементов по условиям для правильного сравнения воздействия условий. Список можно продолжить; в большинстве технологических областей существует огромный постоянный спрос на случайность.

Возможна ли случайность

Такой спрос на случайные числа чреват одной проблемой: мы не полностью уверены в том, что они реально существуют. Некоторые люди считают, что Вселенная детерминирована: если нечто двигается, то его движение было вызвано другим полностью прослеживаемым движением, которое в свою очередь было вызвано другим движением, и т. д. — как при столкновении бильярдных шаров. Если Вселенная ведет себя как бильярдный стол, то, зная текущее состояние каждой частицы во Вселенной,

мы могли бы однозначно определить все прошлое и будущее. В таком случае любое событие — выигрыш в лотерею, встреча со старым знакомым на другом конце света, падение метеорита на голову — в действительности не случайно, как нам хотелось бы думать, а всего лишь является полностью предопределенным следствием конфигурации Вселенной с десяток миллиардов лет назад. Это означало бы, что никакой случайности не существует, что мы застряли в мелодии механического пианино, а события кажутся случайными только потому, что мы о них недостаточно знаем.

Математические правила физики в нашем понимании соответствуют концепции детерминированной Вселенной, но также соответствуют и недетерминированной Вселенной, в которой случайность действительно существует — как выражаются некоторые, «Бог играет в кости». Они соответствуют и сценарию «множественных миров», в котором происходят все возможные версии события, но происходят они в разных Вселенных, недоступных друг для друга. Все эти интерпретации законов физики только усложняются, когда мы пытаемся найти в космосе место для свободы воли. Интерпретация математической физики, которую мы принимаем, зависит не от нашего математического понимания, а скорее от наших философских склонностей — с математической точки зрения приемлема любая позиция.

Независимо от того, существует во Вселенной случайность или нет, в вашем компьютере ее нет — или, по крайней мере, не должно быть. Предполагается, что компьютеры — наши идеально послушные слуги, которые делают только то, что мы им явно приказываем, в точности, когда и как приказываем. Когда вы приказываете компьютеру запустить видеоигру, выполнить машинное обучение на базе случайного леса или провести эксперимент с элементами случайности, вы приказываете изначально детерминированной машине сгенерировать нечто недетерминированное: случайное число. Выполнить такой приказ невозможно.

Так как компьютер не может обеспечить полноценную случайность, были разработаны алгоритмы, которые обеспечивают лучшее, что возможно в такой ситуации: *псевдослучайность*. Алгоритмы генерирования псевдослучайных чисел важны по всем тем же причинам, по которым важны случайные числа. Поскольку истинная случайность на компьютере невозможна (и не исключено, что невозможна во Вселенной в целом), алгоритмы генерирования псевдослучайных чисел должны проектироваться чрезвычайно внимательно, чтобы их вывод был по возможности близок к полноценной случайности. Как можно судить о том, в какой степени алгоритм генерирования псевдослучайных чисел напоминает настоящую случайность? Это зависит от математических определений и теории, которые вскоре будут представлены ниже.

Начнем с простого алгоритма генерирования случайных чисел и посмотрим, в какой степени его вывод напоминает случайные числа.

Линейные конгруэнтные генераторы

Один из простейших примеров генератора псевдослучайных чисел (ГПСЧ) — *линейный конгруэнтный генератор* (ЛКГ). Для реализации этого алгоритма необходимо выбрать три числа, которые мы назовем n_1, n_2 и n_3. ЛКГ начинает с натурального числа (например, 1), после чего применяет следующее уравнение для получения следующего числа:

$$следующее = (предыдущее \times n_1 + n_2) \; mod \; n_3.$$

Это весь алгоритм, который, по сути, состоит из одного шага. На языке Python вместо *mod* будет использоваться оператор `%`, а вся реализация функции ЛКГ приведена в листинге 5.8.

Листинг 5.8. Линейный конгруэнтный генератор

```
def next_random(previous,n1,n2,n3):
    the_next = (previous * n1 + n2) % n3
    return(the_next)
```

Обратите внимание: функция `next_random()` является детерминированной, то есть при получении одного ввода вы всегда будете получать один и тот же вывод. Стоит еще раз пояснить, что наш ГПСЧ должен быть таким из-за детерминированности самих компьютеров. ЛКГ генерирует не настоящие случайные числа, а числа, которые выглядят как случайные — то есть *псевдослучайные*.

Чтобы оценить способность этого алгоритма генерировать псевдослучайные числа, будет полезно просмотреть набор сгенерированных чисел. Вместо того чтобы получать случайные числа по одному, можно построить целый список значений с помощью функции, многократно вызывающей только что созданную функцию `next_random()`:

```
def list_random(n1,n2,n3):
    output = [1]
    while len(output) <=n3:
        output.append(next_random(output[len(output) - 1],n1,n2,n3))
    return(output)
```

При вызове функции `list_random(29,23,32)` будет получен следующий список:

```
[1, 20, 27, 6, 5, 8, 31, 26, 9, 28, 3, 14, 13, 16, 7, 2, 17, 4, 11, 22, 21,
24, 15, 10, 25, 12, 19, 30, 29, 0, 23, 18, 1]
```

В этом списке сложно обнаружить какую-то закономерность — что, собственно, нам и нужно. Прежде всего можно заметить, что он содержит только числа от 0

до 32. Кроме того, можно заметить, что последний элемент этого списка равен 1, то есть совпадает с первым элементом. Если вам понадобится больше случайных чисел, то список можно было бы продолжить вызовом функции `next_random()` для его последнего элемента 1. Однако следует помнить, что функция `next_random()` является детерминированной. Продолжив список, вы получите повторение его начала, так как следующим «случайным» числом после 1 всегда будет 20, следующим случайным числом после 20 всегда будет 27, и т. д. Со временем вы снова вернетесь к числу 1, и весь цикл будет повторяться до бесконечности. Количество уникальных значений, которые можно получить до повторения, называется *периодом* ГПСЧ. В данном случае период нашего ЛКГ равен 32.

Оценка ГПСЧ

Тот факт, что наш генератор случайных чисел со временем начнет повторяться, является потенциальной слабостью, поскольку он позволяет людям предсказать, что будет дальше — а именно этого не должно быть в ситуациях, требующих случайности. Предположим, ЛКГ будет использоваться для управления интернет-казино с рулеткой на 32 сектора. Умный игрок, достаточно долго наблюдающий за рулеткой, заметит, что выпадающие числа регулярно повторяются через каждые 32 запуска. Он сможет выиграть все ваши деньги, делая ставки на числа, которые, как он заранее знает, будут выигрывать в каждом раунде.

Идея умного игрока, пытающегося выиграть в рулетку, будет полезной для оценки любого ГПСЧ. Если рулеткой будет управлять настоящая случайность, то ни один игрок не сможет уверенно выиграть в нее. Но любую незначительную слабость (или отклонение от истинной случайности) нашего ГПСЧ достаточно умный игрок сможет обратить в свою пользу. Даже если вы создаете ГПСЧ для цели, не имеющей никакого отношения к рулетке, спросите себя: «А если я использую ГПСЧ для управления рулеткой, не проиграю ли я все свои деньги?» Интуитивно понятный тест рулетки позволяет судить о том, насколько хорош или плох любой конкретный ГПСЧ. Наш ЛКГ пройдет проверку, если количество запусков не превысит 32, но после этого игрок сможет заметить повторяющуюся последовательность и начнет делать ставки с идеальной точностью. ЛКГ не проходит тест рулетки из-за своей малой периодичности. А значит, будет полезно позаботиться о том, чтобы ГПСЧ имел большой период. Но в случае с рулеткой, имеющей всего 32 сектора, период детерминированного алгоритма не может быть более 32. Вот почему ГПСЧ часто оценивается не по большому, а по *полному* периоду. Возьмем ГПСЧ, который будет получен в результате вызова `list_random(1,2,24)`:

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21,
23, 1]
```

В этом случае период равен 12. Этого может быть достаточно только для очень простых целей, но данный период не является полным, поскольку не включает все возможные значения в диапазоне. И снова умный игрок заметит, что на рулетке не выпадают четные числа (не говоря уже о том, что нечетные числа следуют очень простой закономерности), и повысит свой выигрыш за ваш счет.

С большим полным периодом связана идея *равномерного распределения*, при котором все числа в диапазоне ГПСЧ генерируются с равной вероятностью. Выполнение команды `list_random(1,18,36)` дает следующий результат:

```
[1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1,
19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1, 19, 1]
```

Здесь 1 и 19 выпадают с 50%-ной вероятностью в результатах ГПСЧ, а у всех остальных чисел вероятность составляет 0 %. С таким неравномерным ГПСЧ задача игрока предельно упрощается. С другой стороны, в случае `list_random(29,23,32)` все числа генерируются с вероятностью около 3,1 %.

Как видите, математические критерии оценки ГПСЧ отчасти связаны друг с другом: отсутствие большого или полного периода может быть причиной неравномерности распределения. С более практической точки зрения этим математические свойства важны только из-за того, что наша интернет-рулетка будет терять деньги. Если же формулировать в более общем виде, то единственный действительно важный критерий для ГПСЧ — возможность выявления закономерностей в результатах.

К сожалению, возможность обнаружения закономерностей трудно компактно сформулировать на математическом или научном языке. Таким образом, большой полный период и равномерное распределение интересуют нас как признаки, предоставляющие информацию об обнаружении закономерностей.

Разумеется, это не все признаки, по которым можно судить о наличии закономерностей. Возьмем ЛКГ, определяемый вызовом `list_random(1,1,37)`. Команда выдает следующий список:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 0, 1]
```

Результат характеризуется большим периодом (37), полным периодом (37) и равномерным распределением (каждое число появляется в выводе с вероятностью 1/37). Тем не менее в нем легко обнаруживается закономерность (число каждый раз увеличивается на 1, пока не достигнет 36, после чего повторяется от 0). Генератор соответствует математическим критериям, которые мы приняли, но определенно не проходит тест рулетки.

Тесты Diehard

Не существует одного универсального теста, который бы показывал, существует ли в ГПСЧ закономерность, которая может быть использована для взлома. Ученые разработали много нетривиальных тестов для оценки устойчивости набора случайных чисел к поиску закономерностей (другими словами, к его способности пройти тест рулетки). Одна из таких подборок — так называемые *тесты Diehard* — состоит из 12 тестов, которые оценивают набор случайных чисел с определенных позиций. Если набор чисел проходит все тесты Diehard, то считается, что эти числа очень близки к истинной случайности. Один из тестов Diehard, называемый *тестом пересекающихся сумм*, берет весь список случайных чисел и вычисляет суммы последовательных элементов списка. Набор таких сумм должен следовать математической закономерности, которая называется *колоколообразной*, или *гауссовой кривой*. Реализация функции, генерирующей список пересекающихся сумм на языке Python, выглядит так:

```
def overlapping_sums(the_list,sum_length):
    length_of_list = len(the_list)
    the_list.extend(the_list)
    output = []
    for n in range(0,length_of_list):
        output.append(sum(the_list[n:(n + sum_length)]))
    return(output)
```

Применение этого теста к новому случайному списку может выглядеть следующим образом:

```
import matplotlib.pyplot as plt
overlap = overlapping_sums(list_random(211111,111112,300007),12)
plt.hist(overlap, 20, facecolor = 'blue', alpha = 0.5)
plt.title('Results of the Overlapping Sums Test')
plt.xlabel('Sum of Elements of Overlapping Consecutive Sections of List')
plt.ylabel('Frequency of Sum')
plt.show()
```

Новый случайный список создается с помощью вызова `list_random(211111,111112,300007)`. Новый случайный список имеет достаточную длину для надежной работы теста пересекающихся сумм. Результатом выполнения кода является гистограмма, на которой представлены частоты наблюдаемых сумм. Если список действительно похож на случайный набор, то можно ожидать, что одни суммы будут большими, другие — маленькими, но большинство будет лежать поблизости от середины диапазона возможных значений. Именно это мы видим на гистограмме (рис. 5.2).

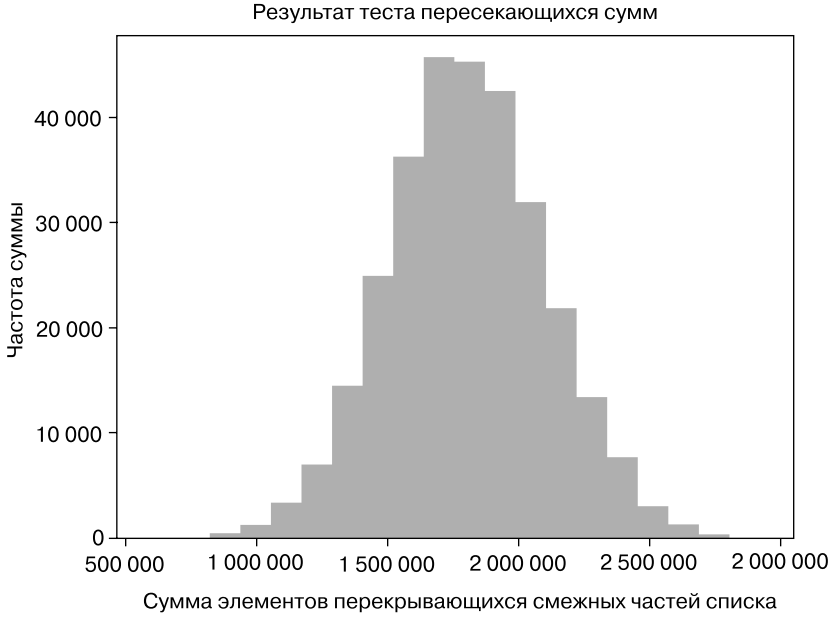


Рис. 5.2. Результат теста пересекающихся сумм для ЛКГ

Присмотревшись, можно заметить, что гистограмма имеет форму колокола. Как говорилось выше, список проходит тест пересекающихся сумм, если распределение набора напоминает колоколообразную кривую — конкретную кривую, важную с математической точки зрения (рис. 5.3).

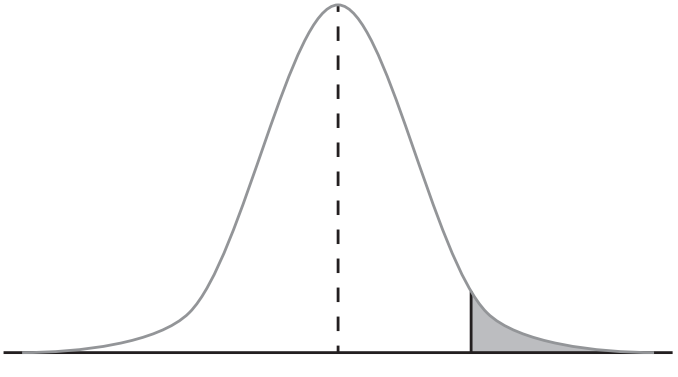


Рис. 5.3. Колоколообразная кривая, или гауссова кривая нормального распределения (источник: Wikimedia Commons)

Колоколообразная кривая, как и золотое сечение, нередко встречается в совершенно неожиданных местах — как в математике, так и в реальном мире. В данном случае близкое сходство между результатами теста перекрывающихся сумм и колоколообразной кривой интерпретируется как доказательство того, что ГПСЧ близок к истинной случайности.

Знание математической теории случайности пригодится вам при разработке генераторов случайных чисел. Тем не менее почти того же результата можно добиться, просто придерживаясь житейских представлений о том, как выигрывать в рулетку.

Регистры сдвига с линейной обратной связью

ЛКГ легко реализуются, но их возможностей недостаточно для всех вариантов применения ГПСЧ; опытный игрок в рулетку моментально взломает ЛКГ. Рассмотрим более совершенную и надежную разновидность алгоритмов — *регистры сдвига с линейной обратной связью* (РСЛС), служащие отправной точкой для углубленного изучения алгоритмов ГПСЧ.

РСЛС проектировались с учетом особенностей компьютерной архитектуры. На самом низком уровне данные в компьютерах хранятся в виде последовательностей 0 и 1 *(битов)*. Одна из возможных строк из десяти битов изображена на рис. 5.4.

Рис. 5.4. Строка из десяти битов

Начнем с этих битов и перейдем к простому алгоритму РСЛС. Начнем с вычисления простой суммы подмножества битов — например, четвертого, шестого, восьмого и десятого бита (также можно выбрать другие подмножества). В данном случае эта сумма равна 3. В архитектуре компьютеров могут храниться только 0 и 1, поэтому мы вычисляем результат sum mod 2 и получаем итоговую сумму 1. Затем правый крайний бит удаляется, а все оставшиеся сдвигаются на 1 позицию вправо (рис. 5.5).

Рис. 5.5. Биты после удаления и сдвига

После удаления бита и сдвига всех остальных битов появляется пустая позиция, в которую нужно вставить новый бит. Им будет сумма, вычисленная ранее. После вставки появляется новое состояние битов (рис. 5.6).

Рис. 5.6. Биты после вставки вычисленной суммы

Бит, удаленный в правой части, берется как результат работы алгоритма — псевдослучайное число, которое должно генерироваться алгоритмом. А поскольку у нас появился новый набор из десяти битов, то можно снова выполнить алгоритм и получить новый псевдослучайный бит, как и прежде. Процесс можно повторять сколько угодно раз.

На языке Python регистр сдвига с линейной обратной связью реализуется достаточно просто. Вместо того чтобы напрямую перезаписывать отдельные биты на жестком диске, мы просто создадим список битов:

```
bits = [1,1,1]
```

Сумма битов в заданных позициях вычисляется в одной строке. Результат сохраняется в переменной `xor_result`, поскольку операция sum mod 2 также называется *исключающим ИЛИ*, или XOR (eXclusive OR). Если вы уже изучали формальную логику, то, вероятно, операция XOR вам уже встречалась — у нее есть как логическое определение, так и эквивалентное математическое определение; мы будем использовать математическое. Мы работаем с короткой битовой строкой, поэтому не суммируем четвертый, шестой, восьмой и десятый биты (так как они не существуют), а вместо этого суммируем второй и третий:

```
xor_result = (bits[1] + bits[2]) % 2
```

Затем извлекаем правый элемент `bits` с помощью удобной функции Python `pop()` и сохраняем результат в переменной `output`:

```
output = bits.pop()
```

Затем сумма вставляется в список функцией `insert()`. При этом указывается позиция 0, так как новый элемент должен находиться у левого края списка:

```
bits.insert(0,xor_result)
```

Объединим все части в одну функцию, которая возвращает два результата: псевдослучайный бит и новое состояние последовательности `bits` (листинг 5.9).

Листинг 5.9. Функция, реализующая РСЛС

```
def feedback_shift(bits):
    xor_result = (bits[1] + bits[2]) % 2
    output = bits.pop()
    bits.insert(0,xor_result)
    return(bits,output)
```

Как и в случае с ЛКГ, мы создадим функцию, которая генерирует список выходных битов:

```
def feedback_shift_list(bits_this):
    bits_output = [bits_this.copy()]
    random_output = []
    bits_next = bits_this.copy()
    while(len(bits_output) < 2**len(bits_this)):
        bits_next,next = feedback_shift(bits_next)
        bits_output.append(bits_next.copy())
        random_output.append(next)
    return(bits_output,random_output)
```

В данном случае цикл `while` выполняется, пока серия битов должна генерироваться. Так как список битов имеет всего $2^3 = 8$ возможных состояний, можно ожидать, что период не превысит 8. На самом деле РСЛС обычно не может выводить полный набор нулей, поэтому на практике период не превышает $2^3 - 1 = 7$. Можно выполнить следующий код, чтобы найти все возможные варианты вывода и проверить период:

```
bitslist = feedback_shift_list([1,1,1])[0]
```

В `bitslist` будет сохранен следующий вывод:

```
[[1, 1, 1], [0, 1, 1], [0, 0, 1], [1, 0, 0], [0, 1, 0], [1, 0, 1], [1, 1, 0],
[1, 1, 1]]
```

Мы видим, что РСЛС выводит все семь возможных битовых строк, которые не состоят из одних нулей. РСЛС имеет полную периодичность, а также демонстрирует равномерное распределение выходных значений. Если использовать больше входных битов, то максимальный возможный период растет с экспоненциальной скоростью: с 10 битами он будет равен $2^{10} - 1 = 1023$, а с 20 битами — $2^{20} - 1 = 1\,048\,575$.

Для проверки списка псевдослучайных битов, генерируемых простым РСЛС, можно воспользоваться следующей командой:

```
pseudorandom_bits = feedback_shift_list([1,1,1])[1]
```

Вывод, хранящийся в `pseudorandom_bits`, выглядит достаточно случайным — особенно если учесть, насколько простым был РСЛС и его входные данные:

```
[1, 1, 1, 0, 0, 1, 0]
```

РСЛС используются для генерирования псевдослучайных чисел в различных областях, включая генерирование белого шума. Мы рассмотрели его здесь, чтобы вы получили представление о современных ГПСЧ. На практике чаще всего используется ГПСЧ *вихрь Мерсенна*, который представляет собой видоизмененный обобщенный регистр сдвига с обратной связью — фактически это намного более изощренная версия РСЛС, представленного в данном разделе. Если вы продолжите изучать ГПСЧ, то столкнетесь с большим количеством запутанных рассуждений и нетривиальной математики, но все они будут строиться на идеях, представленных здесь: детерминированных математических формулах, результаты которых достаточно близки к случайным по строгим математическим критериям.

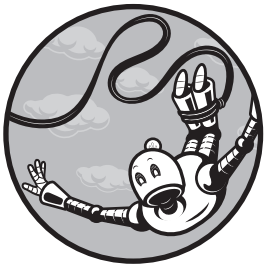
Резюме

Математика и алгоритмы всегда были тесно связаны. Чем сильнее вы углубляетесь в одну область, тем более будете готовы к восприятию нетривиальных идей из другой. Математические обоснования могут показаться непрактичными и запутанными, но здесь идет игра с дальним прицелом: теоретические достижения в математике иногда приводят к появлению практических технологий только через несколько веков. В этой главе мы изучили непрерывные дроби и алгоритм, генерирующий представление любого числа в виде непрерывной дроби. Кроме того, обсудили квадратные корни и алгоритм, который калькуляторы используют для их вычисления. Наконец, мы обсудили тему случайности, включающую два алгоритма для генерирования псевдослучайных чисел, и математические принципы, которые могут использоваться для оценки списков, претендующих на случайность.

В следующей главе мы рассмотрим оптимизацию и эффективный метод, с помощью которого вы можете путешествовать по миру или выковать меч.

6

Расширенная оптимизация



Тема оптимизации вам уже знакома. В главе 3 рассматривался метод градиентного подъема/спуска, предназначенный для поиска максимумов и минимумов функций. Любую задачу оптимизации можно рассматривать как разновидность поиска экстремумов: мы стараемся найти лучший возможный результат из огромного диапазона возможностей. Метод градиентного подъема прост и элегантен, но имеет свою ахиллесову пяту: он может найти пик, который оптимален только локально, но не является глобальным оптимумом. Если провести аналогию с альпинизмом, то данный метод может привести вас на вершину предгорья; если вы немного спуститесь, то сможете подняться на огромную гору, на которую действительно хотите взобраться. Решение этой проблемы — самый сложный и важный аспект расширенной оптимизации.

В этой главе мы рассмотрим более сложный алгоритм расширенной оптимизации, использующий ситуационный анализ. Мы обсудим задачу коммивояжера, а также некоторые ее возможные решения и их недостатки. Наконец, мы рассмотрим *имитацию отжига* — алгоритм оптимизации, который справляется с этими недостатками и может выполнять глобальную оптимизацию, не ограничиваясь локальной.

Жизнь коммивояжера

Задача коммивояжера играет чрезвычайно важную роль в информатике и комбинаторике. Представьте, что коммивояжер хочет объехать несколько городов, чтобы продать свои товары. Поездки между городами обходятся дорого (потерянное время, стоимость бензина, головная боль после долгой поездки) (рис. 6.1).

Рис. 6.1. Коммивояжер в Неаполе

В задаче коммивояжера требуется определить порядок перемещений между городами, минимизирующий затраты на поездки. Как и все лучшие научные задачи, она просто формулируется и в высшей степени сложно решается.

Постановка задачи

Запустим Python и начнем эксперименты. Сначала сгенерируем случайную карту, по которой будет перемещаться коммивояжер. Начнем с выбора числа N, представляющего количество городов на карте. Допустим, $N = 40$. Затем выберем 40 пар координат: одно значение x и одно значение y для каждого города. Для генерирования случайных координат будет использоваться модуль `numpy`:

```
import numpy as np
random_seed = 1729
np.random.seed(random_seed)
N = 40
x = np.random.rand(N)
y = np.random.rand(N)
```

В этом фрагменте используется метод `random.seed()` модуля `numpy`. Этот метод получает произвольное число и использует его для инициализации алгоритма генерирования псевдослучайных чисел (о генерировании псевдослучайных чисел см. в главе 5). Это означает, что если вы используете для инициализации такое же значение, как в предыдущем фрагменте, то будут сгенерированы те же случайные числа, что и в нашем примере. Так вам будет проще следить за выполнением кода, а ваши диаграммы и результаты будут идентичны приведенным в тексте. Затем значения x и y объединяются для создания `cities` — списка с парами координат для каждого из 40 случайно сгенерированных городов.

```
points = zip(x,y)
cities = list(points)
```

Если вы выполните команду `print(cities)` с консоли Python, то получите список случайно сгенерированных точек. Каждая точка представляет город. Мы не станем присваивать городам названия, а будем обозначать первый город `cities[0]`, второй — `cities[1]` и т. д.

У нас уже есть все необходимое для того, чтобы опробовать возможные решения задачи коммивояжера. Первое решение — простой обход всех городов в порядке их следования в списке `cities`. Определим переменную `itinerary`, в которой этот порядок будет храниться в списке:

```
itinerary = list(range(0,N))
```

Этот способ записи эквивалентен следующему:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,\
26,27,28,29,30,31,32,33,34,35,36,37,38,39]
```

Числа следуют в том порядке, в котором мы собираемся обходить города: сначала город 0, затем город 1 и т. д.

Далее необходимо оценить этот маршрут и решить, представляет ли он хорошее или, по крайней мере, приемлемое решение для задачи коммивояжера. Вспомните, что суть задачи коммивояжера — минимизация затрат на перемещение между городами. Что же считать такими затратами? Вы можете задать любую функцию затрат: возможно, на некоторых дорогах более интенсивное движение, или вам приходится пересекать реки, или на север ездить труднее, чем на восток, — или наоборот.

Но начнем с простого: предположим, что перемещение на расстояние 1 обходится в 1 доллар независимо от направления и от того, через сколько городов вы проезжаете. В этой главе единицы расстояния не указываются, поскольку алгоритмы будут работать одинаково независимо от того, перемещается ли коммивояжер на мили, километры или световые годы. В этом случае минимизация затрат эквивалентна минимизации проходимого расстояния.

Чтобы определить расстояние для конкретного маршрута, мы определим две новые функции. Прежде всего понадобится функция, которая генерирует набор отрезков, соединяющих все точки. После этого необходимо просуммировать расстояния, представленные этими отрезками. Начнем с определения пустого списка, в котором будет храниться информация об отрезках:

```
lines = []
```

Затем перебираем все города и на каждом шаге добавляем в коллекцию `lines` новый отрезок, связывающий текущий город со следующим.

```
for j in range(0,len(itinerary) - 1):
    lines.append([cities[itinerary[j]],cities[itinerary[j + 1]]])
```

При выполнении команды `print(lines)` вы увидите, как хранится информация об отрезках в Python. Каждый отрезок хранится в виде списка с координатами двух городов. Например, команда `print(lines[0])` выводит первый отрезок, а ее результат выглядит так:

```
[(0.21215859519373315, 0.1421890509660515), (0.25901824052776146,
0.4415438502354807)]
```

Эти элементы можно поместить в одну функцию `genlines`, которая получает в аргументах `cities` и `itinerary` и возвращает набор отрезков, соединяющих все города в списке в порядке, заданном в маршруте:

```
def genlines(cities,itinerary):
    lines = []
    for j in range(0,len(itinerary) - 1):
        lines.append([cities[itinerary[j]],cities[itinerary[j + 1]]])
    return(lines)
```

Итак, теперь мы можем сгенерировать набор отрезков между каждыми двумя городами в маршруте, можем написать функцию для вычисления общего расстояния по этим отрезкам. Сначала общее расстояние определяется равным 0, после чего для каждого элемента в списке `lines` к переменной `distance` прибавляется длина этого отрезка. Для вычисления длин отрезков используется теорема Пифагора.

ПРИМЕЧАНИЕ

Вычисление расстояний на поверхности Земли по теореме Пифагора не совсем правильно; поверхность Земли изогнута, поэтому для вычисления истинных расстояний потребуется более изощренная геометрия. Мы игнорируем этот нюанс и считаем, что коммивояжер умеет прокладывать туннели под землей или же живет в некой двумерной геометрической утопии, в которой расстояния легко вычисляются по древнегреческим формулам. Теорема Пифагора дает неплохое приближение для истинных расстояний (особенно небольших).

```
import math
def howfar(lines):
    distance = 0
    for j in range(0,len(lines)):
        distance += math.sqrt(abs(lines[j][1][0] - lines[j][0][0])**2 + \
        abs(lines[j][1][1] - lines[j][0][1])**2)
    return(distance)
```

Функция получает на входе список отрезков и выводит сумму их длин. Имея такие функции, мы можем вызвать их вместе с маршрутом для определения общего расстояния, которое преодолеет наш коммивояжер:

```
totaldistance = howfar(genlines(cities,itinerary))
print(totaldistance)
```

При выполнении этого кода оказалось, что значение `totaldistance` равно приблизительно `16.81`. Вы получите те же результаты, если будете использовать то же значение инициализации. Если выбрать другое значение инициализации или набор городов, то результат будет слегка отличаться от приведенного.

Чтобы получить представление о том, что означает этот результат, полезно построить диаграмму маршрута. Для этого мы создадим функцию `plotitinerary()`:

```
import matplotlib.collections as mc
import matplotlib.pylab as pl
def plotitinerary(cities,itin,plottitle,thename):
    lc = mc.LineCollection(genlines(cities,itin), linewidths=2)
    fig, ax = pl.subplots()
```

```
    ax.add_collection(lc)
    ax.autoscale()
    ax.margins(0.1)
    pl.scatter(x, y)
    pl.title(plottitle)
    pl.xlabel('X Coordinate')
    pl.ylabel('Y Coordinate')
    pl.savefig(str(thename) + '.png')
    pl.close()
```

Функция `plotitinerary()` получает аргументы `cities`, `itin`, `plottitle` и `thename`, где `cities` — список городов, `itin` — маршрут, который вы хотите нанести на диаграмму, `plottitle` — заголовок, который будет выводиться в верхней части диаграммы, а `thename` — имя, которое присваивается выходному файлу в формате `png`. Функция использует модуль `pylab` для построения диаграммы и модуль `collections` из `matplotlib` для создания набора отрезков. Затем она выводит точки маршрута и соединяющие их отрезки.

При построении диаграммы маршрута с помощью вызова `plotitinerary(cities,itinerary,'TSP - Random Itinerary','figure2')` вы получите диаграмму, как на рис. 6.2.

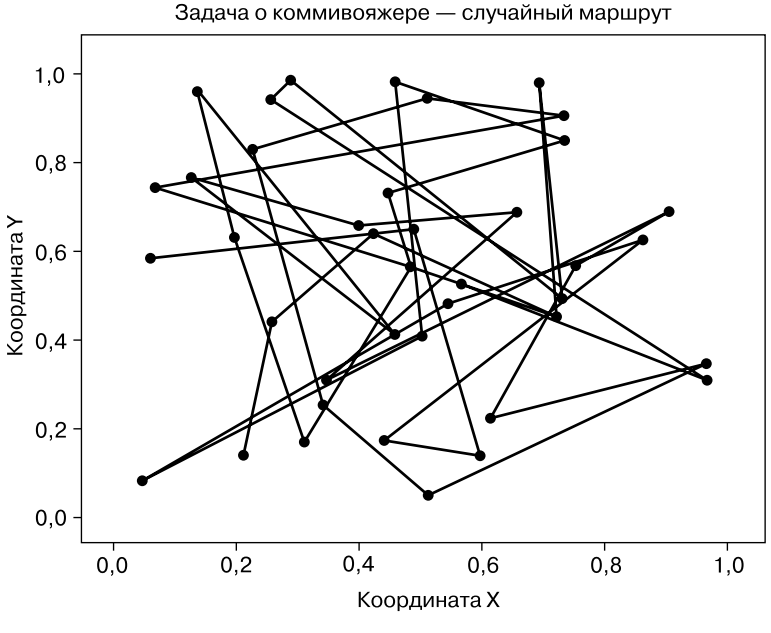


Рис. 6.2. Маршрут, полученный при посещении городов в случайном порядке, в котором они были созданы

По рис. 6.2 нетрудно догадаться, что мы еще не нашли лучшего решения задачи коммивояжера. Бедному коммивояжеру приходится несколько раз метаться по всей стране в отдаленные города; совершенно очевидно, что он мог бы действовать намного эффективнее, если бы остановился в других городах на этом пути. В оставшейся части данной главы мы будем использовать алгоритмы для нахождения маршрута с минимальным расстоянием.

Первое возможное решение, которое мы обсудим, будет самым простым, но оно же оказывается наименее эффективным. После этого рассмотрим решения, которые за счет некоторого усложнения значительно улучшают быстродействие.

Ум против грубой силы

Казалось бы, можно составить список всех возможных маршрутов, соединяющих города, проверить их все и посмотреть, какой из них лучше. Если вы хотите посетить три города, нетрудно составить полный список всех возможных порядков их посещения:

- 1, 2, 3;
- 1, 3, 2;
- 2, 3, 1;
- 2, 1, 3;
- 3, 1, 2;
- 3, 2, 1.

Сравнение длин всех маршрутов и определение лучшего из них не займет много времени. Такое решение называется *методом грубой силы*. Имеется в виду не физическая сила, а проверка полного списка, основанная на вычислительной мощности процессора вместо интеллекта разработчика алгоритма, который мог бы найти более элегантное решение с меньшим временем выполнения.

Иногда метод грубой силы оказывается именно тем, чем нужно. Такие решения проще программируются и надежно работают. Их главная слабость — время выполнения, которое никогда не бывает лучше, а обычно оказывается намного хуже, чем у алгоритмических решений.

В задаче коммивояжера время выполнения растет слишком быстро, чтобы метод грубой силы можно было реально применить более чем для 20 городов. Чтобы убедиться в этом, подумайте, сколько возможных маршрутов пришлось бы проверить, если вы пытаетесь найти все возможные порядки посещения четырех городов.

1. При выборе первого города возможны четыре варианта, так как ни один из четырех городов еще не был посещен. Таким образом, общее количество вариантов выбора первого города равно 4.
2. При выборе второго города возможны три варианта, так как городов всего четыре и один из них уже был посещен. Таким образом, общее количество вариантов выбора первых двух городов равно $4 \times 3 = 12$.
3. При выборе третьего города остаются два варианта, так как городов всего четыре и мы уже посетили первые два. Следовательно, общее количество способов выбора первых трех городов равно $4 \times 3 \times 2 = 24$.
4. При выборе четвертого города остается всего один вариант, так как из четырех городов три уже были посещены. Таким образом, общее количество способов выбора всех четырех городов составит $4 \times 3 \times 2 \times 1 = 24$.

Нетрудно заметить закономерность: для N городов общее количество возможных маршрутов составит $N \times (N - 1) \times (N - 2) \times ... \times 3 \times 2 \times 1$, или N факториал (обозначается $N!$). Факториал растет невероятно быстро: если значение 3! равно всего 6 (что можно легко перебрать даже без компьютера), то для 10! оно превышает 3 миллиона (достаточно просто перебирается методом грубой силы на современном компьютере). Далее значение 18! превышает 6 квадриллионов, 25! превышает 15 септиллионов, а значения 35! и выше выходят за границы того, что можно сделать методом грубой силы при современном уровне технологий и предполагаемого срока жизни Вселенной.

Это явление называется *комбинаторным взрывом*. У него нет четкого математического определения, но этим термином обозначаются подобные ситуации, в которых комбинации и перестановки небольших наборов данных порождает огромное количество вариантов, далеко превышающее размер исходного набора и вообще любые размеры, с которыми можно работать методом грубой силы.

Например, количество возможных маршрутов, соединяющих 90 населенных пунктов Род-Айленда, намного больше количества атомов во Вселенной, хотя сам Род-Айленд гораздо меньше Вселенной. Аналогичным образом количество возможных партий на шахматной доске больше количества атомов во Вселенной, хотя шахматная доска меньше даже Род-Айленда. Из-за таких парадоксальных ситуаций, в которых из безусловно ограниченного набора возникает практически бесконечное количество комбинаций, разработка хороших алгоритмов играет еще более важную роль, поскольку метод грубой силы никогда не сможет исследовать все возможные решения самых сложных задач. Комбинаторный взрыв означает, что для задачи коммивояжера придется искать алгоритмические решения, так как во всем мире не найдется достаточных вычислительных мощностей для нахождения решения методом грубой силы.

Алгоритм ближайшего соседа

Теперь рассмотрим простой и интуитивно понятный метод, называемый *алгоритмом ближайшего соседа*. Начнем с первого города в списке. Затем найдем ближайший к нему город, который мы еще не посещали ранее, и посетим его вторым. На каждом шаге мы просто определяем текущее местоположение и выбираем ближайший город, который еще не посещали, в качестве следующей точки маршрута.

Такой подход минимизирует расстояние на каждом шаге, хотя необязательно минимизирует общее расстояние. Обратите внимание: вместо того чтобы перебирать все возможные маршруты, как бы это делалось методом грубой силы, на каждом шаге мы находим только ближайшего соседа. При таком подходе обеспечивается время выполнения, очень быстрое даже для очень больших N.

Реализация поиска ближайшего соседа

Начнем с написания функции, которая находит ближайшего соседа любого конкретного города. Предположим, имеется точка `point` и список городов `cities`. Расстояние между `point` и `j`-м элементом `cities` вычисляется по следующей формуле в стиле теоремы Пифагора:

```
point = [0.5,0.5]
j = 10
distance = math.sqrt((point[0] - cities[j][0])**2 + (point[1] - cities[j][1])**2)
```

Чтобы узнать, какой элемент `cities` находится ближе всего к точке `point` (то есть найти ближайшего соседа `point`), необходимо перебрать все элементы `cities` и проверить расстояние между `point` и каждым элементом `cities` (листинг 6.1).

Листинг 6.1. Функция findnearest() для поиска города, ближайшего к заданному

```
def findnearest(cities,idx,nnitinerary):
    point = cities[idx]
    mindistance = float('inf')
    minidx = - 1
    for j in range(0,len(cities)):
        distance = math.sqrt((point[0] - cities[j][0])**2 + (point[1] -
                                        cities[j][1])**2)
        if distance < mindistance and distance > 0 and j not in nnitinerary:
            mindistance = distance
            minidx = j
    return(minidx)
```

С появлением функции `findnearest()` можно переходить к реализации алгоритма ближайшего соседа. Наша цель — построить маршрут, который будет называться `nnitinerary`. Для начала выберем первый город, с которого начнется перемещение коммивояжера:

```
nnitinerary = [0]
```

Если маршрут должен состоять из *N* городов, мы должны перебрать все числа от 0 до *N* – 1, для каждого из этих чисел найти соседа, ближайшего к последнему посещенному городу, и присоединить этот город к маршруту. Эта задача будет решаться функцией `donn()` (сокращение от DO Nearest Neighbour) из листинга 6.2. Функция начинает с первого города из `cities` и на каждом шаге добавляет в маршрут город, ближайший к последнему посещенному, пока все города не будут включены в маршрут.

Листинг 6.2. Функция, которая последовательно находит ближайший город и возвращает полученный маршрут

```
def donn(cities,N):
    nnitinerary = [0]
    for j in range(0,N - 1):
        next = findnearest(cities,nnitinerary[len(nnitinerary) - 1],nnitinerary)
        nnitinerary.append(next)
    return(nnitinerary)
```

У нас есть все необходимое для проверки эффективности алгоритма ближайшего соседа. Начнем с вывода графического представления маршрута ближайшего соседа:

```
plotitinerary(cities,donn(cities,N),'TSP - Nearest Neighbor','figure3')
```

Результат показан на рис. 6.3.

Можно также проверить, какое расстояние коммивояжеру придется преодолеть по новому маршруту:

```
print(howfar(genlines(cities,donn(cities,N))))
```

В данном случае мы видим, что наш алгоритм сократил расстояние до 6,29 (вместо 16,81 для случайного пути). Напомню, что конкретные единицы не указываются, так что расстояние можно интерпретировать как 6,29 миль (или километров, или парсеков). Здесь важно то, что значение меньше 16,81 мили/километра/парсека для случайного маршрута. Это значительное улучшение было достигнуто исключительно благодаря очень простому интуитивному алгоритму. На рис. 6.3 повышение эффективности очевидно; количество перемещений на другой конец карты сократилось, и стало больше коротких перемещений между городами, расположенными вблизи друг от друга.

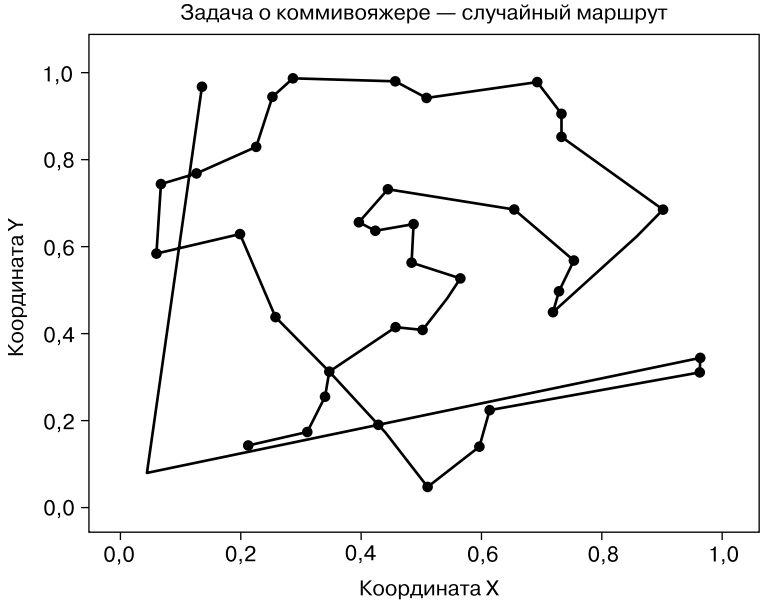


Рис. 6.3. Маршрут, сгенерированный алгоритмом ближайшего соседа

Проверка дальнейших улучшений

Внимательно присмотревшись к рис. 6.2 или 6.3, можно представить себе другие возможные усовершенствования. Вы можете попробовать внести эти усовершенствования самостоятельно, а потом проверить их с помощью функции `howfar()`. Допустим, вы смотрите на исходный случайный маршрут:

```
initial_itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,\
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39]
```

Вы думаете, что маршрут коммивояжера можно улучшить, изменив позиции городов 6 и 30 в порядке маршрута. Для этого можно определить новый маршрут, в котором эти два числа (выделенные жирным шрифтом) меняются местами:

```
new_itinerary = [0,1,2,3,4,5,30,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,\
24,25,26,27,28,29,6,31,32,33,34,35,36,37,38,39]
```

Простое сравнение покажет, привела ли замена к снижению общего расстояния:

```
print(howfar(genlines(cities,initial_itinerary)))
print(howfar(genlines(cities,new_itinerary)))
```

Если new_itinerary лучше initial_itinerary, то можно отбросить initial_itinerary и оставить новый маршрут. В данном случае мы видим, что у new_itinerary суммарное расстояние равно 16,79 — небольшое улучшение по сравнению с исходным. Обнаружив одно небольшое улучшение, можно повторить процесс: выбрать два города, поменять их местами в маршруте и проверить, уменьшилось ли расстояние. Этот процесс можно продолжать бесконечно и на каждом шаге с разумной вероятностью ожидать, что нам удастся найти возможность снижения расстояния. После многократного повторения этого процесса можно получить маршрут с очень малым общим расстоянием (или, по крайней мере, надеяться на это).

Функция, которая выполняет процесс замены и проверки автоматически, пишется достаточно просто (листинг 6.3).

Листинг 6.3. Функция, которая вносит небольшое изменение в маршрут, сравнивает измененный маршрут с исходным и возвращает более короткий маршрут

```
def perturb(cities,itinerary):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    itinerary2[neighborids1] = itinerary[neighborids2]
    itinerary2[neighborids2] = itinerary[neighborids1]

    distance1 = howfar(genlines(cities,itinerary))
    distance2 = howfar(genlines(cities,itinerary2))

    itinerarytoreturn = itinerary.copy()

    if(distance1 > distance2):
        itinerarytoreturn = itinerary2.copy()

    return(itinerarytoreturn.copy())
```

Функция perturb() получает в аргументах произвольный список городов и маршрут. Затем она определяет две переменные: neighborids1 и neihborids2, случайно выбранные целые числа в диапазоне от 0 до длины маршрута. Затем создается новый маршрут itinerary2, который идентичен исходному, не считая того, что города neighborids1 и neighborids2 меняются местами. Затем функция вычисляет distance1 — общее расстояние исходного маршрута, и distance2 — общее расстояние маршрута itinerary2. Если distance2 меньше distance1, то возвращается новый маршрут (с заменой городов). В противном случае возвращается исходный маршрут. Таким образом, этой функции передается маршрут, а она всегда возвращает маршрут как минимум не хуже переданного.

Итак, теперь у нас есть функция `perturb()`, и мы можем многократно вызывать ее для случайного маршрута. Вместо одного раза мы вызовем ее 2 миллиона раз, пытаясь получить наименьшее общее расстояние:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,\
26,27,28,29,30,31,32,33,34,35,36,37,38,39]

np.random.seed(random_seed)
itinerary_ps = itinerary.copy()
for n in range(0,len(itinerary) * 50000):
    itinerary_ps = perturb(cities,itinerary_ps)

print(howfar(genlines(cities,itinerary_ps)))
```

Только что реализованную схему можно было бы назвать алгоритмом *поиска с возмущениями (perturb search algorithm)*. Алгоритм проверяет многие тысячи возможных маршрутов в попытке найти лучший маршрут по аналогии с поиском методом грубой силы. Однако этот алгоритм лучше, поскольку поиск методом грубой силы рассматривает все возможные маршруты без разбора, а новый алгоритм представляет собой *управляемый поиск*, который рассматривает набор маршрутов, монотонно убывающих по общему расстоянию перемещения, поэтому он должен прибыть к хорошему решению быстрее метода грубой силы. Остается внести несколько небольших изменений, чтобы реализовать *имитацию отжига* — краеугольный алгоритм данной главы.

Прежде чем переходить к коду имитации отжига, мы разберемся, какие улучшения он предлагает по сравнению с алгоритмами, рассматривавшимися до настоящего момента. Будет также представлена *температурная функция*, которая позволяет реализовать функциональность имитации отжига на языке Python.

Жадные алгоритмы

Алгоритмы ближайшего соседа и поиска с возмущением, рассматривавшиеся выше, принадлежат к классу так называемых *жадных алгоритмов*. Они работают по шагам, и на каждом из них принимаются решения локально оптимальные, но они могут не быть глобально оптимальными при рассмотрении ситуации в целом. В случае алгоритма ближайшего соседа на каждом шаге ищется ближайший город, к которому переместится коммивояжер на этом шаге; остальные города при этом игнорируются. Посещение ближайшего города является локально оптимальным, так как он минимизирует расстояние перемещения на текущем шаге. Но поскольку алгоритм не учитывает все остальные города, он может не быть глобально оптимальным — он может создавать странные перемещения по карте, из-за которых общее

расстояние становится очень длинным и затратным, хотя каждый отдельный шаг выглядит разумно.

Под жадностью подразумевается недальновидность процесса принятия решений с локальной оптимизацией. Чтобы понять суть жадного подхода к задачам оптимизации в контексте задачи поиска наивысшей точки сложной, холмистой местности, «высокие» точки сравниваются с лучшими, оптимальными решениями (малыми расстояниями в задаче коммивояжера), а «низкие» — с худшими, субоптимальными решениями (большими расстояниями в задаче коммивояжера). Жадный подход к поиску наивысшей точки на местности всегда будет заставлять нас подниматься, но иногда будет заводить на вершину небольшого предгорья вместо вершины самой высокой горы. Иногда бывает нужно спуститься к основанию холма, чтобы начать более важный подъем на более высокую гору. Так как жадные алгоритмы ищут только локальные улучшения, они никогда не позволяют спускаться вниз, из-за чего могут застрять на локальных экстремумах. Именно эта проблема рассматривалась в главе 3.

Понимая этот факт, вы готовы к идее, которая позволит решить проблему локальной оптимизации, присущую жадным алгоритмам. Идея заключается в том, чтобы отказаться от наивного стремления только вверх. В задаче коммивояжера иногда приходится переключаться на худшие маршруты, чтобы позднее добраться до лучших возможных маршрутов, подобно тому как мы спускаемся по предгорью, чтобы в конечном итоге подняться на гору. Иначе говоря, чтобы добиться лучшего результата в будущем, необходимо пойти на некоторые потери на начальной стадии.

Температурная функция

Пойти на определенные потери, чтобы добиться выигрыша в будущем, — нетривиальная задача. Если переусердствовать с готовностью к временным потерям, то можно спускаться вниз на каждом шаге и в итоге оказаться на низкой точке вместо высокой. Необходимо найти способ отступать на небольшую величину, нечасто и только чтобы понять, как в конечном итоге добиться лучшего результата.

Снова представьте, что вы стоите на сложной холмистой местности. Вы начинаете свое путешествие ближе к вечеру и знаете, что на поиск самой высокой точки местности у вас не более двух часов. У вас нет хронометра, чтобы следить за временем, но воздух вечером постепенно остывает, и вы решаете использовать температуру для примерной оценки времени, оставшегося на поиск наивысшей точки.

В начале пути еще достаточно тепло, так что вы не ограничиваете себя в исследованиях. Времени еще много, поэтому вы не особо рискуете, спускаясь немного вниз, чтобы лучше узнать местность и увидеть новые места. Но постепенно вокруг

становится холоднее, конец двухчасового интервала приближается, и возможности для свободного исследования понемногу теряются. Вы стараетесь улучшить свой результат и в меньшей степени склонны спускаться вниз.

Поразмыслите над этой стратегией и постарайтесь понять, почему она повышает ваши шансы на достижение наивысшей точки. Мы уже обсуждали, почему время от времени приходится спускаться вниз; это делается для ухода от локальных оптимумов, то есть вершины предгорья рядом с огромной горой. Но когда следует спускаться? Возьмем последние десять секунд двухчасового периода времени. Где бы вы ни находились, в это время нужно идти только вверх. Спускаться вниз, чтобы искать новые горы и предгорья, в эти десять секунд уже бессмысленно. Даже если вы найдете перспективную гору, у вас все равно не будет времени на то, чтобы подняться на нее, и если за последние десять секунд вы совершите ошибку и спуститесь вниз, то у вас не будет времени на ее исправление. Таким образом, в последние десять секунд следует идти только вверх, даже не помышляя о спуске.

И наоборот, возьмем первые десять секунд двухчасового периода. В это время нет необходимости стремиться вверх. Сначала стоит извлечь максимум пользы от небольшого спуска для проведения исследований. Если вы совершите ошибку в первые десять секунд, то у вас будет достаточно времени на ее исправление. У вас есть все возможности для того, чтобы воспользоваться полученной информацией. В первые десять секунд можно спокойно относиться к временному спуску и не стремиться к постоянному подъему.

Аналогичным образом можно подойти к остальному времени. Если рассматривать промежуток за десять минут до конца, то вы можете руководствоваться более умеренным вариантом рассуждений о десяти секундах. Время уже на исходе, нужно идти наверх. Но десять минут все-таки больше, чем десять секунд, и вы скорее рискнете небольшим понижающим исследованием на случай, если вдруг обнаружится что-нибудь перспективное. По тому же принципу через десять минут от начала будут действовать более умеренные рассуждения о десяти первых секундах. На протяжении двухчасового периода времени будет действовать градиент намерений: готовность иногда спускаться на начальном этапе, за которой следует постепенно растущее стремление двигаться только вверх.

Чтобы смоделировать этот сценарий на языке Python, можно определить функцию. Начнем с высокой температуры, готовности исследовать и спускаться вниз и постепенно придем к низкой температуре и отказу от спуска. Температурная функция относительно проста. При вызове ей передается аргумент `t` (прошедшее время):

```
temperature = lambda t: 1/(t + 1)
```

Чтобы увидеть простой график температурной функции, выполните приведенную ниже программу на консоли Python. Сначала программа импортирует функциональность `matplotlib`, а затем определяет `ts` — переменную, содержащую диапазон значений `t` от 1 до 100. Наконец, она строит график температуры, связанной с каждым значением `t`. Напомню, что нас не интересуют конкретные единицы измерения или значения величин, поскольку это гипотетическая ситуация, которая должна продемонстрировать форму температурной функции. По этой причине мы используем 1 для представления максимальной температуры, 0 для представления минимальной температуры, 0 для представления минимального времени и 99 для представления максимального времени без указания единиц.

```
import matplotlib.pyplot as plt
ts = list(range(0,100))
plt.plot(ts, [temperature(t) for t in ts])
plt.title('The Temperature Function')
plt.xlabel('Time')
plt.ylabel('Temperature')
plt.show()
```

На рис. 6.4 изображен полученный график.

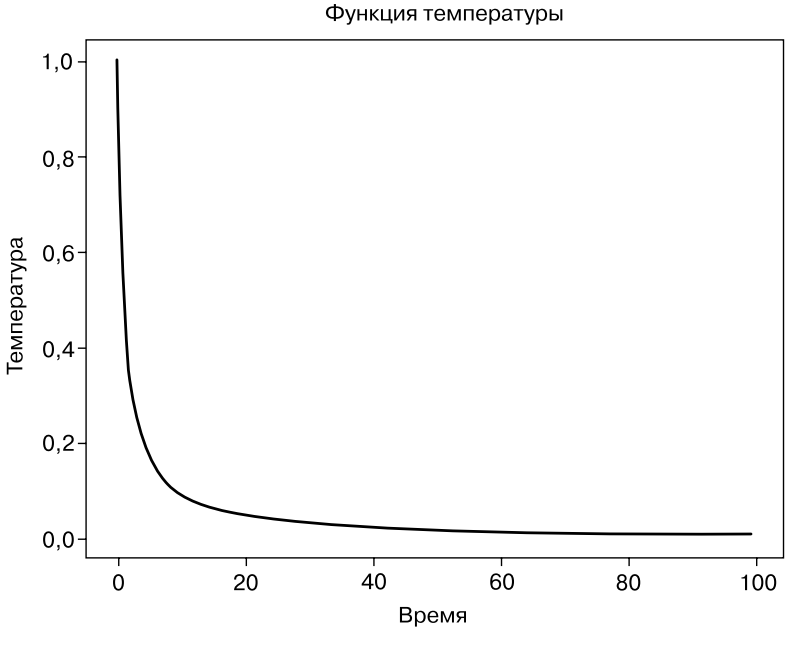


Рис. 6.4. Температура убывает с течением времени

На графике показана температура, наблюдаемая во время нашей гипотетической оптимизации. Температура используется как фактор, управляющий оптимизацией; наша готовность двигаться вниз в любой момент времени пропорциональна температуре.

Теперь у нас есть все необходимое для полноценной реализации имитации отжига. Вперед, довольно теоретических рассуждений!

Имитация отжига

Объединим все представленные идеи: функцию температуры, задачу поиска на холмистой местности, алгоритм поиска с возмущением и задачу коммивояжера. В контексте задачи коммивояжера сложная холмистая местность состоит из всех возможных решений задачи. Можно представить, что хорошее решение соответствует более высокой точке, а плохое — более низкой. Применяя функцию `perturb()`, мы переходим к другой точке местности в надежде, что она окажется по возможности высокой.

Для управления исследованиями этой местности будет использоваться температурная функция. В начале процесса высокая температура будет диктовать большую готовность к выбору ухудшенного маршрута. Ближе к концу мы будем менее открыты для выбора плохих маршрутов и будем склоняться к жадной оптимизации.

Реализуемый алгоритм — *имитация отжига* (simulated annealing) — представляет собой измененную форму алгоритма поиска с возмущением. Принципиальное отличие заключается в том, что при имитации отжига мы иногда соглашаемся принять изменения маршрута, увеличивающие преодолеваемое расстояние, поскольку это позволяет обойти проблему локальной оптимизации. Наша готовность к принятию худших маршрутов зависит от текущей температуры.

Внесем эти изменения в функцию `perturb()`. У нее появится новый аргумент: время `time`. Он определяет, насколько мы продвинулись в процессе имитации отжига; первый вызов `perturb()` начинается с времени `1`, после чего время будет равно `2`, `3` и т. д., в зависимости от того, сколько раз вызывается функция `perturb()`. Мы добавим строку, которая задает температурную функцию, и еще одну, в которой выбирается случайное число. Если случайное число меньше температуры, то мы будем готовы принять ухудшенный маршрут. Если случайное число больше температуры, то ухудшенный маршрут приниматься не будет. При таком подходе в отдельных случаях (хотя и не всегда) будут приниматься ухудшенные маршруты, а вероятность принятия ухудшенного маршрута будет убывать со временем

по мере снижения температуры. Назовем эту новую функцию `perturb_sa1()`, где `sa` — сокращение от Simulated Annealing (то есть имитация отжига). В листинге 6.4 приведена новая функция `perturb_sa1()` с этими изменениями.

Листинг 6.4. Обновленная версия функции perturb(), которая учитывает температуру и случайный фактор

```
def perturb_sa1(cities,itinerary,time):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    itinerary2[neighborids1] = itinerary[neighborids2]
    itinerary2[neighborids2] = itinerary[neighborids1]

    distance1 = howfar(genlines(cities,itinerary))
    distance2 = howfar(genlines(cities,itinerary2))

    itinerarytoreturn = itinerary.copy()

    randomdraw = np.random.rand()
    temperature = 1/((time/1000) + 1)

    if((distance2 > distance1 and (randomdraw) < (temperature))
      or (distance1 > distance2)):
        itinerarytoreturn=itinerary2.copy()

    return(itinerarytoreturn.copy())
```

Просто добавив эти две короткие строки, новый аргумент и новое условие `if` (выделены жирным шрифтом в листинге 6.4), мы уже получаем очень простую функцию имитации отжига. Была немного изменена и температурная функция; так как она будет вызываться с очень большими значениями `time`, мы используем `time/1000` вместо `time` в составе делителя в температурной функции. Для сравнения эффективности имитации отжига с алгоритмом поиска с возмущением и алгоритмом ближайшего соседа можно воспользоваться следующим кодом:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,\
26,27,28,29,30,31,32,33,34,35,36,37,38,39]
np.random.seed(random_seed)

itinerary_sa = itinerary.copy()
for n in range(0,len(itinerary) * 50000):
    itinerary_sa = perturb_sa1(cities,itinerary_sa,n)

print(howfar(genlines(cities,itinerary)))      # Случайный маршрут
print(howfar(genlines(cities,itinerary_ps)))   # Поиск с возмущением
print(howfar(genlines(cities,itinerary_sa)))   # Имитация отжига
print(howfar(genlines(cities,donn(cities,N)))) # Ближайший сосед
```

Поздравляю! Вы научились выполнять имитацию отжига. Мы видим, что случайный маршрут обеспечивает расстояние 16,81, тогда как маршрут по алгоритму ближайшего соседа обеспечивает расстояние 6,29, как наблюдалось ранее. У маршрута путем поиска с возмущением расстояние равно 7,38, а у маршрута методом имитации отжига оно равно 5,92. В данном случае оказалось, что поиск с возмущением обеспечивает лучшие результаты, чем случайный маршрут, алгоритм ближайшего соседа превосходит поиск с возмущением и случайный маршрут, а имитация отжига превосходит все остальные алгоритмы. При других значениях инициализации генератора можно получить другие результаты, включая случаи, в которых имитация отжига отстает от метода ближайшего соседа. Дело в том, что процесс имитации отжига весьма чувствителен, и некоторые его аспекты нуждаются в точной настройке, чтобы алгоритм работал эффективно и надежно. После настройки он будет стабильно обеспечивать более высокую эффективность, чем более простые жадные алгоритмы оптимизации. Оставшаяся часть главы посвящена тонкостям имитации отжига, включая настройку алгоритма для обеспечения наилучшей эффективности.

МЕТАФОРИЧЕСКАЯ МЕТАЭВРИСТИКА

Особенности имитации отжига будет проще понять, если знать происхождение названия. Отжиг — процесс из металлургии, при котором металлы сначала нагреваются, а затем постепенно охлаждаются. Нагревание металла приводит к разрушению многих связей между частицами. В процессе остывания формируются новые связи, с которыми металл приобретает новые, более желательные свойства. Имитация отжига напоминает отжиг в том смысле, что при высокой температуре пространство решений «разрушается»: мы принимаем ухудшенные решения в надежде на то, что при охлаждении связи можно будет привести в состояние, в котором они станут лучше, чем прежде.

Метафора получается немного надуманной и вряд ли будет интуитивно понятной человеку, не сведущему в металлургии. Имитация отжига относится к категории так называемых метафорических метаэвристик. Существует много других метафорических метаэвристик, которые берут процесс, существующий в природе или человеческом обществе, и находят возможность адаптировать его для решения имеющейся задачи оптимизации. Им присваиваются такие названия, как «алгоритм муравьиной колонии», «алгоритм кукушки», «оптимизация каракатицы», «оптимизация кошачьей стаи», «лягушачьи прыжки», «колония императорских пингвинов», «гармонический поиск» (основанный на импровизациях джазовых музыкантов) и «дождевой алгоритм». Некоторые аналогии неестественны и не особенно полезны, но иногда позволяют по-настоящему понять суть серьезной задачи или вдохновить на это. В любом случае такие алгоритмы почти всегда интересно изучать и программировать.

Настройка алгоритма

Как упоминалось ранее, имитация отжига — чувствительный процесс. Приведенный выше код показывает, как выполнять ее на простейшем уровне, но желательно настроить некоторые детали для повышения эффективности работы алгоритма. Изменение мелких подробностей или параметров алгоритма в целях обеспечения улучшенной эффективности без изменения основной схемы обычно называется *настройкой* и способно существенно повлиять на результат в сложных случаях, к которым относится наша задача.

Функция `perturb()` вносит небольшое изменение в маршрут: меняет местами два города. Но это не единственный способ внесения возмущений в маршрут. Трудно заранее определить, какие методы внесения возмущений эффективнее других, но вы всегда можете опробовать несколько вариантов и сравнить результаты.

Другой естественный способ внесения возмущений в маршрут основан на обратном обходе его части: берем подмножество городов и обходим их в другом порядке. На языке Python обратную перестановку можно выполнить в одну строку. Если выбрать в маршруте два города с индексами `small` и `big`, то следующий фрагмент переставляет все города между ними в обратном порядке:

```
small = 10
big = 20
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,\
26,27,28,29,30,31,32,33,34,35,36,37,38,39]
itinerary[small:big] = itinerary[small:big][::-1]
print(itinerary)
```

При выполнении того фрагмента в выводе приводится маршрут, в котором города с 10 по 19 переставлены в обратном порядке:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

Другой способ внесения возмущений в маршрут — перемещение части маршрута с одного места на другое. Например, можно взять следующий маршрут:

```
itinerary = [0,1,2,3,4,5,6,7,8,9]
```

и переместить всю часть `[1,2,3,4]` на более позднюю позицию в маршруте, в результате чего маршрут принимает следующий вид:

```
itinerary = [0,5,6,7,8,1,2,3,4,9]
```

Подобные перемещения могут выполняться с помощью следующего фрагмента Python, который перемещает выбранную часть в случайную позицию:

```
small = 1
big = 5
itinerary = [0,1,2,3,4,5,6,7,8,9]
tempitin = itinerary[small:big]
del(itinerary[small:big])
np.random.seed(random_seed + 1)
neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
for j in range(0,len(tempitin)):
    itinerary.insert(neighborids3 + j,tempitin[j])
```

Можно обновить функцию `perturb()`, чтобы она случайным образом переключалась между этими методами внесения возмущений. Для этого мы выберем еще одно случайное число в диапазоне от 0 до 1. Если новое случайное число принадлежит некоему диапазону (допустим, 0–0,45), то в маршруте подмножество городов переставляется в обратном порядке, но если оно входит в другой диапазон (допустим, 0,45–0,55), то два города меняются местами. Если оно принадлежит последнему диапазону (0,55–1), то в маршруте перемещается подмножество городов. Таким образом, функция `perturb()` может случайным образом переключаться между разными типами возмущений. Случайный выбор и эти типы возмущений помещаются в новую функцию `perturb_sa2()`, приведенную в листинге 6.5.

Листинг 6.5. Использование нескольких методов внесения возмущений в маршрут

```
def perturb_sa2(cities,itinerary,time):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    randomdraw2 = np.random.rand()
    small = min(neighborids1,neighborids2)
    big = max(neighborids1,neighborids2)
    if(randomdraw2 >= 0.55):
        itinerary2[small:big] = itinerary2[small:big][:: - 1]
    elif(randomdraw2 < 0.45):
        tempitin = itinerary[small:big]
        del(itinerary2[small:big])
        neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
        for j in range(0,len(tempitin)):
            itinerary2.insert(neighborids3 + j,tempitin[j])
    else:
        itinerary2[neighborids1] = itinerary[neighborids2]
        itinerary2[neighborids2] = itinerary[neighborids1]

    distance1 = howfar(genlines(cities,itinerary))
```

```
    distance2 = howfar(genlines(cities,itinerary2))

    itinerarytoreturn = itinerary.copy()

    randomdraw = np.random.rand()
    temperature = 1/((time/1000) + 1)

    if((distance2 > distance1 and (randomdraw) < (temperature))
      or (distance1 > distance2)):
        itinerarytoreturn = itinerary2.copy()

    return(itinerarytoreturn.copy())
```

Наша функция `perturb()` стала намного более сложной и гибкой; она может вносить разные типы изменений в маршруты в зависимости от случайных проверок. Гибкость не всегда становится целью, к которой следует стремиться просто ради нее самой, а сложность таковой безусловно не является. Чтобы судить о том, стоит ли наращивать сложность и гибкость в данном случае (и во всех остальных), нужно проверить, улучшают ли они быстродействие. Такова природа настройки: как и при настройке музыкального инструмента, вы не знаете заранее, до какой степени нужно натянуть струны — вы усиливаете или ослабляете натяжение, прислушиваетесь к тому, как звучит инструмент, и продолжаете настройку. При проверке изменений (в листинге 6.5 они выделены жирным шрифтом) вы видите, повышают ли они эффективность по сравнению с кодом, который выполнялся до этого.

Предотвращение крупных потерь

Вся суть имитации отжига заключается в том, что нам приходится идти на временные потери, чтобы получить лучший результат в будущем. Однако при этом нам хотелось бы избежать внесения изменений, которые *слишком сильно* ухудшают текущий результат. В текущей реализации функция `perturb()` принимает ухудшенный маршрут каждый раз, когда случайное значение меньше температуры. Для этого используется следующая условная команда:

```
if((distance2 > distance1 and randomdraw < temperature) or (distance1 > distance2)):
```

Мы хотим изменить это условие, чтобы готовность принять худший маршрут зависела не только от температуры, но и от того, насколько гипотетическое изменение ухудшает маршрут. Если ухудшение незначительно, то мы скорее примем его, чем значительную потерю. Для этого в условную команду будет включена метрика ухудшения маршрута. Следующая команда эффективно решает эту задачу:

```
scale = 3.5
if((distance2 > distance1 and (randomdraw) < (math.exp(scale*(distance1-distance2)) *
temperature)) or (distance1 > distance2)):
```

Включая условную команду в код, мы получаем функцию из листинга 6.6, в котором приведен только самый конец функции `perturb()`.

```
--...--
# Начало функции perturb

    scale = 3.5
    if((distance2 > distance1 and (randomdraw) < (math.exp(scale * (distance1 -
        distance2)) * temperature)) or (distance1 > distance2)):
        itinerarytoreturn = itinerary2.copy()

    return(itinerarytoreturn.copy())
```

Поддержка отмены

В процессе имитации отжига мы можем неосмотрительно принять однозначно плохое изменение маршрута. На такой случай будет полезно запомнить лучший маршрут, который был обнаружен на данный момент, и разрешить алгоритму вернуться к нему при определенных условиях. В листинге 6.6 приведен код реализации отмены, выделенный жирным шрифтом в новой, полной функции внесения возмущений при имитации отжига.

Листинг 6.6. Функция, выполняющая полный процесс имитации отжига и возвращающая оптимальный маршрут

```
def perturb_sa3(cities,itinerary,time,maxitin):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))
    global mindistance
    global minitinerary
    global minidx
    itinerary2 = itinerary.copy()
    randomdraw = np.random.rand()

    randomdraw2 = np.random.rand()
    small = min(neighborids1,neighborids2)
    big = max(neighborids1,neighborids2)
    if(randomdraw2>=0.55):
        itinerary2[small:big] = itinerary2[small:big][::- 1 ]
    elif(randomdraw2 < 0.45):
        tempitin = itinerary[small:big]
        del(itinerary2[small:big])
        neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
        for j in range(0,len(tempitin)):
            itinerary2.insert(neighborids3 + j,tempitin[j])
    else:
        itinerary2[neighborids1] = itinerary[neighborids2]
```

```
        itinerary2[neighborids2] = itinerary[neighborids1]

    temperature=1/(time/(maxitin/10)+1)

    distance1 = howfar(genlines(cities,itinerary))
    distance2 = howfar(genlines(cities,itinerary2))

    itinerarytoreturn = itinerary.copy()

    scale = 3.5
    if((distance2 > distance1 and (randomdraw) < (math.exp(scale*(distance1 -
      distance2)) * \ temperature)) or (distance1 > distance2)):
        itinerarytoreturn = itinerary2.copy()

    reset = True
    resetthresh = 0.04
    if(reset and (time - minidx) > (maxitin * resetthresh)):
        itinerarytoreturn = minitinerary
        minidx = time

    if(howfar(genlines(cities,itinerarytoreturn)) < mindistance):
        mindistance = howfar(genlines(cities,itinerary2))
        minitinerary = itinerarytoreturn
        minidx = time

    if(abs(time - maxitin) <= 1):
        itinerarytoreturn = minitinerary.copy()

    return(itinerarytoreturn.copy())
```

В этой функции мы определяем глобальные переменные для минимального расстояния, найденного на данный момент маршрута, при котором оно было достигнуто, и времени, за которое оно было достигнуто. Если пройдет слишком много времени и при этом не будет найдено ничего лучше маршрута с наименьшим расстоянием, то можно сделать вывод, что все изменения, внесенные после этой точки, были ошибочными и стоит вернуться к этому лучшему маршруту. Отмена выполняется только в том случае, если мы опробовали много возмущений, но не добились улучшения по сравнению с предыдущим самым хорошим результатом, а переменная `resetthresh` определит, как долго следует ожидать перед отменой. Наконец, в функцию добавляется новый аргумент `maxitin`, который сообщает функции, сколько раз мы собираемся вызывать эту функцию, чтобы можно было точно определить текущую точку процесса. Аргумент `maxitin` также используется в температурной функции, чтобы температурная кривая могла гибко адаптироваться к различным возмущениям, которые мы собираемся вносить. Когда время истечет, возвращается маршрут, который обеспечил наилучшие результаты на протяжении поиска.

Проверка эффективности

После всех правок и усовершенствований можно написать функцию `siman()` (сокращение от SIMulated ANnealing, то есть имитация отжига), которая создаст глобальные переменные, а затем многократно вызывает новейшую функцию `perturb()`, в конечном итоге приходя к маршруту с очень малым расстоянием перемещения (листинг 6.7).

Листинг 6.7. Функция выполняет полный процесс имитации отжига и возвращает оптимизированный маршрут

```
def siman(itinerary,cities):
    newitinerary = itinerary.copy()
    global mindistance
    global minitinerary
    global minidx
    mindistance = howfar(genlines(cities,itinerary))
    minitinerary = itinerary
    minidx = 0

    maxitin = len(itinerary) * 50000
    for t in range(0,maxitin):
        newitinerary = perturb_sa3(cities,newitinerary,t,maxitin)

    return(newitinerary.copy())
```

Затем мы вызываем функцию `siman()` и сравниваем ее результаты с результатами алгоритма ближайшего соседа:

```
np.random.seed(random_seed)
itinerary = list(range(N))
nnitin = donn(cities,N)
nnresult = howfar(genlines(cities,nnitin))
simanitinerary = siman(itinerary,cities)
simanresult = howfar(genlines(cities,simanitinerary))
print(nnresult)
print(simanresult)
print(simanresult/nnresult)
```

При выполнении этого кода выясняется, что итоговая функция имитации отжига обеспечивает маршрут с расстоянием 5,32. По сравнению с маршрутом алгоритма ближайшего соседа с расстоянием 6,29 обеспечивается улучшение более чем на 15 %. На первый взгляд, результат обескураживает: мы потратили несколько десятков страниц на изучение сложных концепций, а общее расстояние уменьшилось всего на 15 %. Претензия разумная — вполне возможно, что вам вообще никогда не понадобится эффективность, превышающая эффективность алгоритма ближайшего

соседа. Но представьте, что вы предлагаете исполнительному директору глобальной логистической компании — такой как UPS или DHL — возможность сократить затраты на перевозки на 15 %; вы увидите, как заблестят его глаза при мысли о миллиардах долларов, которые принесет такая экономия. Логистика остается основным определяющим фактором высоких затрат и загрязнения окружающей среды в большинстве коммерческих отраслей, и эффективное решение задачи коммивояжера всегда приносит большие изменения на практике. Кроме того, задача коммивояжера чрезвычайно важна с академической точки зрения как основа для сравнения методов оптимизации и отправная точка для исследования современных теоретических идей.

Чтобы вывести маршрут, полученный в результате имитации отжига, выполните команду `plotitinerary(cities,simanitinerary,'Traveling Salesman Itinerary - Simulated Annealing','figure5')`. Маршрут изображен на рис. 6.5.

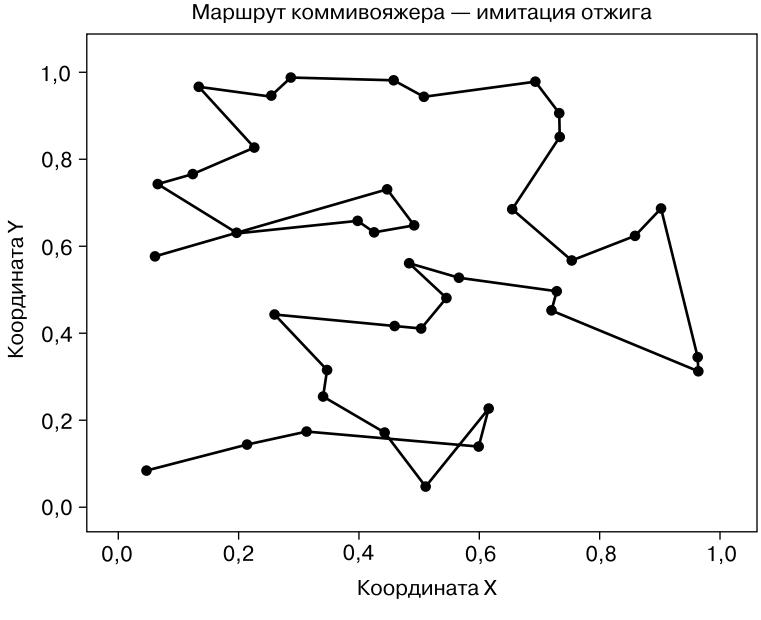


Рис. 6.5. Маршрут, сгенерированный алгоритмом имитации отжига

С одной стороны, это всего лишь набор случайно сгенерированных точек, соединенных отрезками. С другой — это результат процесса оптимизации, выполняемого за сотни тысяч итераций, с непрестанным поиском идеала среди почти бесконечных вариантов — и в этом отношении он прекрасен.

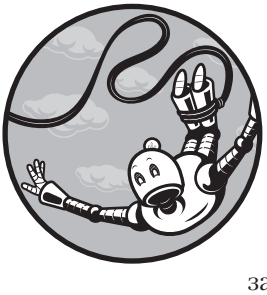
Резюме

В этой главе расширенная оптимизация рассматривалась на примере задачи коммивояжера. Мы обсудили несколько подходов к решению задачи, включая поиск методом грубой силы, поиск методом ближайшего соседа и, наконец, имитацию отжига — эффективный алгоритм, который позволяет пойти на временное ухудшение ради достижения лучшего результата в будущем. Надеюсь, что при изучении такого сложного случая, как задача коммивояжера, вы приобрели навыки, которые могут применяться в других задачах оптимизации.

В следующей главе мы обратимся к геометрии и займемся изучением действенных алгоритмов, лежащих в основе многих геометрических операций и конструкций. Приключения продолжаются!

7

Геометрия



Людям присуще интуитивное восприятие геометрии. Каждый раз, когда вы толкаете диван по узкому коридору, рисуете картину в Pictionary или оцениваете, на каком расстоянии от вас находится другая машина на дороге, вы занимаетесь сложным геометрическим осмыслением, которое часто зависит от подсознательно усвоенных вами алгоритмов. К настоящему моменту вас уже не удивит тот факт, что геометрия естественно подходит для алгоритмических рассуждений.

В этой главе мы воспользуемся геометрическим алгоритмом для решения задачи почтмейстера. Начнем с формулировки задачи и посмотрим, как она решается с использованием диаграмм Вороного. Оставшаяся часть главы объясняет, как сгенерировать это решение алгоритмическим методом.

Задача почтмейстера

Предположим, в одном городе среди жилых домов размещаются четыре почтовых отделения (рис. 7.1).

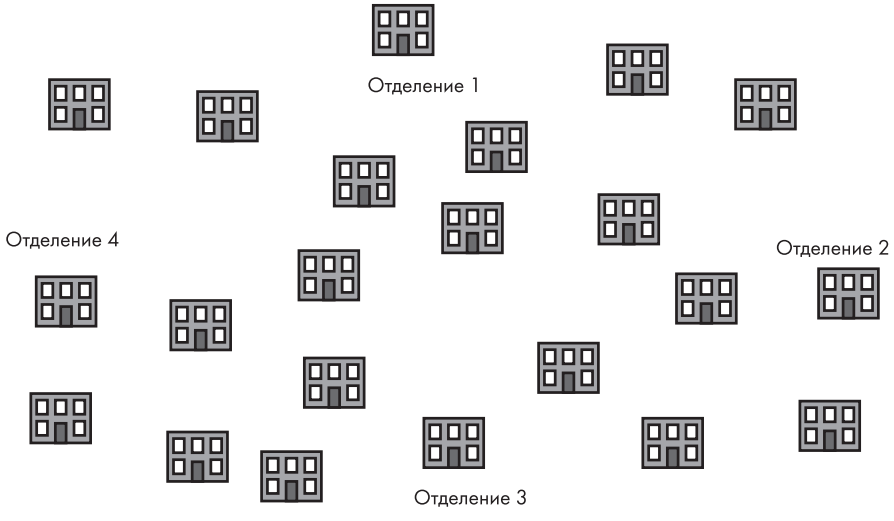


Рис. 7.1. Город с почтовыми отделениями

Система доставки почты между отделениями явно нуждается в оптимизации. Может оказаться, что почтовое отделение 4 должно доставлять почту в дом, который находится ближе к отделениям 2 и 3; в то же время почтовое отделение 2 должно доставлять почту в дом, который находится ближе к отделению 4 (рис. 7.2).

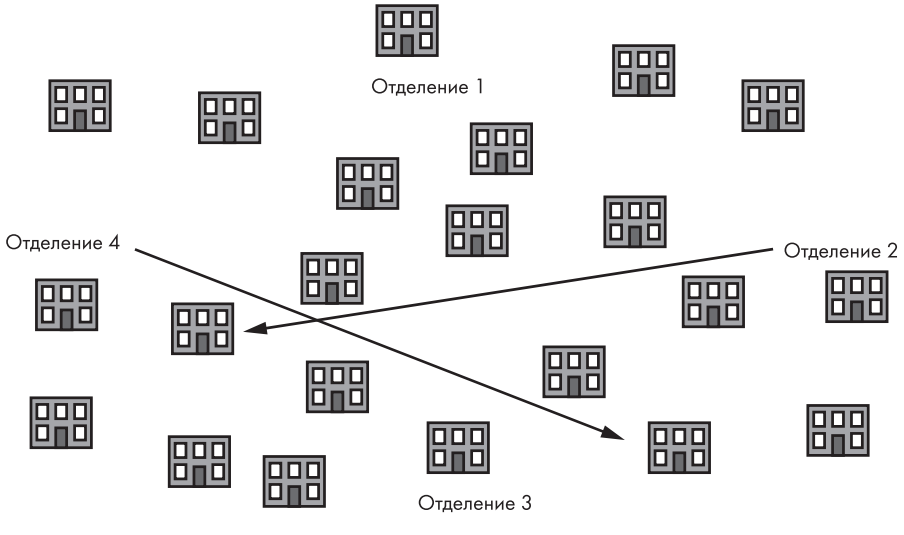


Рис. 7.2. Неэффективная доставка почты из почтовых отделений 2 и 4

Систему доставки нужно реорганизовать так, чтобы каждый дом получал почтовые отправления из наиболее подходящего почтового отделения. Таковым может быть то, в котором больше всего свободного персонала, или то, в котором имеется подходящий транспорт для доставки, или то, которое располагает необходимой информацией для нахождения всех адресов в окрестностях. Но, скорее всего, идеальным почтовым отделением будет попросту ближайшее. Возможно, вы заметили, что эта задача отчасти напоминает задачу коммивояжера — по крайней мере в том смысле, что объекты перемещаются и мы хотим сократить расстояние перемещения. Однако задача коммивояжера является задачей оптимизации порядка обхода заданного маршрута одним коммивояжером, а в новой задаче оптимизируется назначение маршрутов для нескольких почтальонов. На самом деле эту задачу и задачу коммивояжера можно решать последовательно, чтобы достичь максимальной эффективности: после того как вы решите, какое почтовое отделение должно осуществлять доставку в те или иные дома, отдельные почтальоны могут воспользоваться задачей коммивояжера для определения порядка обхода этих домов.

Простейший подход к решению данной задачи, которую можно назвать *задачей почтмейстера*, основан на поочередном рассмотрении каждого дома, вычислении расстояния между домом и каждым из четырех почтовых отделений и назначении ближайшего отделения для доставки почты в дом.

У такого подхода есть ряд недостатков. Во-первых, он не предоставляет простой возможности назначения при появлении новых домов; каждый вновь построенный дом должен пройти через тот же трудоемкий процесс сравнения со всеми существующими почтовыми отделениями. Во-вторых, вычисления на уровне отдельных домов не позволяют собрать информацию о регионе в целом. Например, может оказаться, что весь район находится в шаге от одного почтового отделения, но отдален на много миль от других почтовых отделений. Было бы лучше сделать вывод, что весь район должен обслуживаться одним и тем же ближайшим почтовым отделением. К сожалению, наш метод требует повторения вычислений для каждого дома в районе только для того, чтобы каждый раз получать один и тот же результат. Вычисляя расстояния для каждого дома по отдельности, мы повторяем работу, которую не пришлось бы повторять, если бы мы могли сделать какие-то обобщающие выводы о целых регионах. И эта работа будет только накапливаться. В мегаполисах с десятками миллионов жителей, множеством почтовых отделений и быстрыми темпами строительства такой подход будет излишне медленным и затратным по вычислительным ресурсам.

Другое, более элегантное решение рассматривает всю карту в целом и разбивает ее на регионы, каждый из которых представляет зону обслуживания одного почтового

отделения. В нашем гипотетическом городе для этого достаточно провести всего две линии (рис. 7.3).

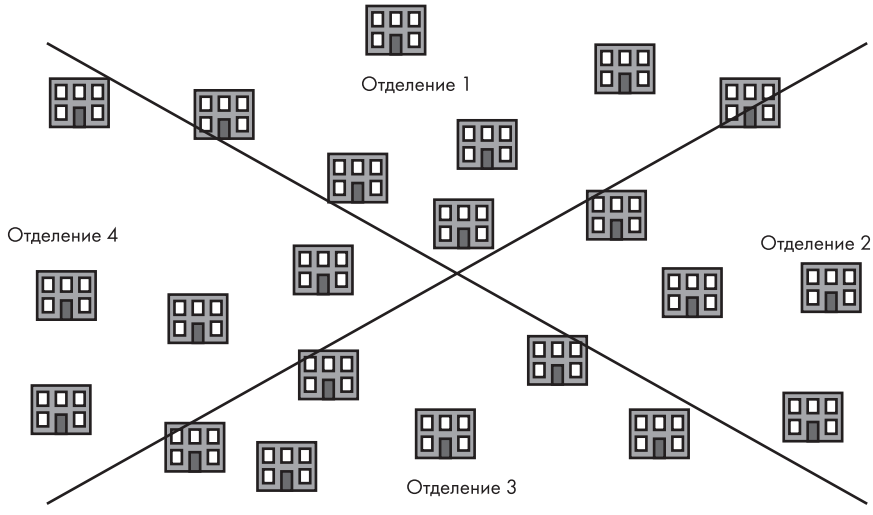


Рис. 7.3. Диаграмма Вороного, разбивающая город на регионы оптимальной доставки

Регионы обозначают ближайшие области, в которых для каждого отдельного дома ближайшим почтовым отделением является то, которое находится в данном регионе. Теперь, когда вся карта была разбита на регионы, можно легко закрепить любое вновь построенное здание за ближайшим почтовым отделением. Для этого достаточно проверить, какому региону оно принадлежит.

Диаграмма, которая разбивает карту на регионы, приближенные к одному почтовому отделению, называется *диаграммой Вороного*. У этих диаграмм долгая история, которая начинается еще со времен Рене Декарта. Они использовались для анализа размещения водных колонок в Лондоне и сбора доказательств относительно распространения холеры. В наши дни эти диаграммы продолжают использоваться в физике и математике для представления кристаллических структур. В данной главе будет описан алгоритм генерирования диаграммы Вороного для произвольного набора точек, позволяющий решить задачу почтмейстера.

Треугольники: краткий курс

Сделаем шаг назад и начнем с простейших элементов исследуемых алгоритмов. В геометрии простейшим элементом анализа является точка. Мы будем представ-

лять точки в виде списков с двумя элементами: координатой x и координатой y, как в следующем примере:

```
point = [0.2,0.8]
```

На следующем уровне сложности точки соединяются, образуя треугольники. Треугольник представляется списком из трех точек:

```
triangle = [[0.2,0.8],[0.5,0.2],[0.8,0.7]]
```

Определим также вспомогательную функцию, которая преобразует набор из трех разрозненных точек в треугольник. Эта маленькая функция просто включает три точки в список и возвращает результат:

```
def points_to_triangle(point1,point2,point3):
    triangle = [list(point1),list(point2),list(point3)]
    return(triangle)
```

Будет полезно наглядно представить треугольники, с которыми мы работаем. Создадим простую функцию, которая получает произвольный треугольник и рисует его. Для начала воспользуемся функцией `genlines()`, определенной в главе 6. Напомню, что функция получает набор точек и преобразует их в линии. Эта функция тоже очень проста, она всего лишь присоединяет точки к списку `lines`:

```
def genlines(listpoints,itinerary):
    lines = []
    for j in range(len(itinerary)-1):
        lines.append([listpoints[itinerary[j]],listpoints[itinerary[j+1]]])
    return(lines)
```

Далее напишем простую функцию графического вывода. Она получает переданный треугольник, разбивает его на значения x и y, вызывает `genlines()` для создания набора отрезков на базе этих значений, рисует точки и линии и, наконец, сохраняет изображение в файле `.png`. Для рисования используется модуль `pylab`, а для создания коллекции отрезков — код модуля `matplotlib`. Эта функция приведена в листинге 7.1.

Листинг 7.1. Функция рисования треугольников

```
import pylab as pl
from matplotlib import collections as mc
def plot_triangle_simple(triangle,thename):
    fig, ax = pl.subplots()

    xs = [triangle[0][0],triangle[1][0],triangle[2][0]]
```

```
    ys = [triangle[0][1],triangle[1][1],triangle[2][1]]

    itin=[0,1,2,0]

    thelines = genlines(triangle,itin)

    lc = mc.LineCollection(genlines(triangle,itin), linewidths=2)

    ax.add_collection(lc)

    ax.margins(0.1)
    pl.scatter(xs, ys)
    pl.savefig(str(thename) + '.png')
    pl.close()
```

Теперь можно выбрать три точки, преобразовать их в треугольник и нарисовать этот треугольник всего одной строкой:

```
plot_triangle_simple(points_to_triangle((0.2,0.8),(0.5,0.2),(0.8,0.7)),'tri')
```

Результат показан на рис. 7.4.

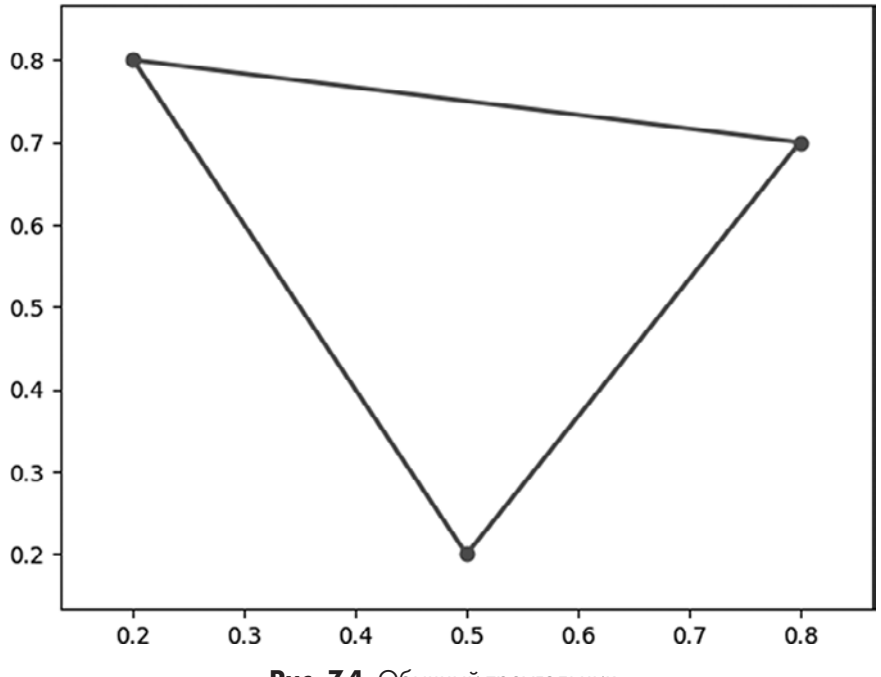


Рис. 7.4. Обычный треугольник

Будет полезна также функция, которая позволяет вычислить расстояние между любыми двумя точками по теореме Пифагора:

```
def get_distance(point1,point2):
    distance = math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
    return(distance)
```

Остается напомнить некоторые распространенные термины из области геометрии:

- *равносторонний* — термин для описания фигуры, у которой все стороны имеют равную длину;
- *перпендикулярный* — термин для описания двух линий, расположенных под углом 90 градусов;
- *вершина* — точка, в которой встречаются две стороны геометрической фигуры.

Продвинутая теория треугольников

Ученый и философ Готфрид Вильгельм Лейбниц считал, что наш мир является лучшим из всех возможных миров, поскольку имеет «максимально простые законы, из которых вытекает наибольшее богатство явлений». Лейбниц считал, что научные законы могут быть сведены к нескольким простым правилам, но эти правила ведут к нескончаемому разнообразию и красоте мира, которые мы наблюдаем. Возможно, это и не относится к Вселенной, но безусловно истинно для треугольника. Начиная с чего-то настолько концептуально простого, как треугольник (фигура с тремя сторонами), мы входим в мир, чрезвычайно богатый явлениями.

Поиск центра описанной окружности

Чтобы вы начали понимать все богатство мира треугольников, рассмотрим простой алгоритм, который можно опробовать с любым треугольником.

1. Найти среднюю точку каждой стороны треугольника.
2. Провести линию от каждой вершины к середине стороны, противоположной от вершины.

Выполнив этот алгоритм, вы получите результат, сходный с тем, что показан на рис. 7.5.

Интересно, что все проведенные вами линии сходятся в одной точке, которая становится своего рода центром треугольника. Все три линии сходятся в одной точке

независимо от того, с какого треугольника вы начинаете. Эта точка иногда называется *центроидом* треугольника, и она всегда находится внутри, в месте, которое можно назвать центром треугольника.

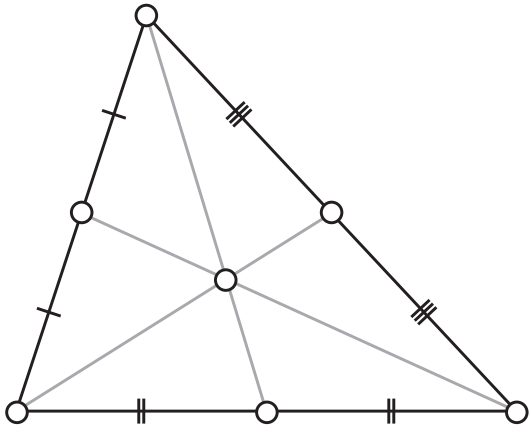


Рис. 7.5. Центроид треугольника (источник: Wikimedia Commons)

Некоторые фигуры — например, окружности — всегда имеют одну точку, которую можно однозначно назвать центром фигуры. Но с треугольниками дело обстоит иначе: центроид — всего лишь одна центроподобная точка, но есть и другие, которые тоже можно считать центрами. Рассмотрим новый алгоритм для произвольного треугольника.

1. Найти среднюю точку каждой стороны треугольника.
2. Провести линию, перпендикулярную к каждой стороне, через среднюю точку этой стороны.

В данном случае линии обычно не проходят через вершины, как это было при нахождении центроида. Сравните рис. 7.5 с рис. 7.6.

Обратите внимание: все линии снова сходятся в одной точке, и она не является центроидом, но часто находится внутри треугольника. У этой точки есть еще одно интересное свойство: она является центром уникальной окружности, которая проходит через все три вершины треугольника. Здесь также проявляется богатство явлений применительно к треугольникам: у каждого треугольника существует одна уникальная окружность, проходящая через все три его вершины. Эта окружность называется окружностью, *описанной вокруг треугольника*. Только что описанный алгоритм находит центр этой окружности.

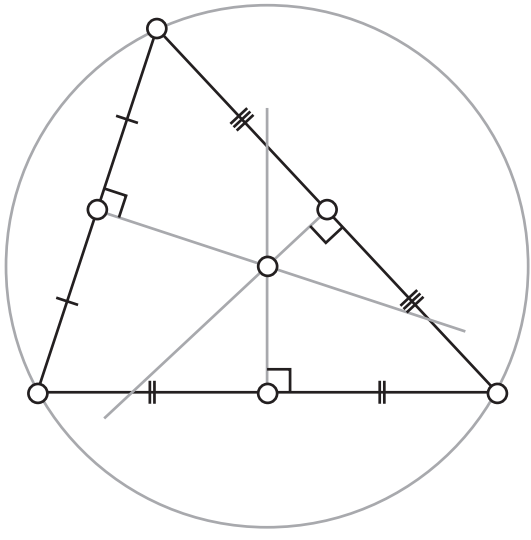


Рис. 7.6. Центр описанной окружности (источник: Wikimedia Commons)

Центр описанной окружности, как и центроид, является точкой, которую можно назвать центром треугольника, но это не единственные кандидаты — в энциклопедии по адресу https://faculty.evansville.edu/ck6/encyclopedia/ETC.html приведен список 40 000 (на данный момент) точек, которые по тем или иным причинам могут называться центрами треугольника. Как сказано в самой энциклопедии, определение центра треугольника «удовлетворяется бесконечным множеством объектов, из которых когда-либо будет опубликовано только конечное подмножество». Интересно, что, начав с трех простых точек и трех прямых отрезков, мы получаем потенциально бесконечную энциклопедию уникальных центров — Лейбниц был бы в восторге.

Мы можем написать функцию, которая находит центр описанной окружности и ее радиус для любого заданного треугольника. Эта функция использует преобразование в комплексные числа. На входе она получает треугольник, а на выходе возвращает координаты центра и радиус:

```
def triangle_to_circumcenter(triangle):
    x,y,z = complex(triangle[0][0],triangle[0][1]),
            complex(triangle[1][0],triangle[1][1]), \
    complex(triangle[2][0],triangle[2][1])
    w = z - x
    w /= y - x
    c = (x-y) * (w-abs(w)**2)/2j/w.imag - x
    radius = abs(c + x)
    return((0 - c.real,0 - c.imag),radius)
```

Подробности того, как эта функция вычисляет центр и радиус, весьма сложны. Я не стану на них задерживаться — при желании вы можете проанализировать код самостоятельно.

Расширение графического вывода

Теперь, когда вы научились находить центр описанной окружности и ее радиус для любого треугольника, улучшим функцию `plot_triangle()`, чтобы она могла вывести всю информацию. Новая функция приведена в листинге 7.2.

Листинг 7.2. Улучшенная функция plot_triangle(), которая рисует описанную окружность и ее центр

```
def plot_triangle(triangles,centers,radii,thename):
    fig, ax = pl.subplots()
    ax.set_xlim([0,1])
    ax.set_ylim([0,1])
    for i in range(0,len(triangles)):
        triangle = triangles[i]
        center = centers[i]
        radius = radii[i]
        itin = [0,1,2,0]
        thelines = genlines(triangle,itin)
        xs = [triangle[0][0],triangle[1][0],triangle[2][0]]
        ys = [triangle[0][1],triangle[1][1],triangle[2][1]]

        lc = mc.LineCollection(genlines(triangle,itin), linewidths = 2)

        ax.add_collection(lc)
        ax.margins(0.1)
        pl.scatter(xs, ys)
        pl.scatter(center[0],center[1])

        circle = pl.Circle(center, radius, color = 'b', fill = False)

        ax.add_artist(circle)
    pl.savefig(str(thename) + '.png')
    pl.close()
```

Начнем с добавления двух новых аргументов: переменной `centers`, которая содержит список соответствующих центров описанных окружностей всех треугольников, и переменной `radii`, которая содержит список радиусов описанных окружностей всех треугольников. Обратите внимание: передаваемые аргументы состоят из списков, так как функция предназначена для рисования нескольких треугольников вместо одного. Для рисования кругов используется

функциональность модуля `pylab`. Позднее мы будем работать с несколькими треугольниками одновременно, и тогда нам пригодится функция, которая может рисовать сразу несколько треугольников вместо одного. Мы включим в функцию цикл, который перебирает все треугольники и центры и последовательно рисует их.

Вызовем эту функцию со списком треугольников, который определим:

```
triangle1 = points_to_triangle((0.1,0.1),(0.3,0.6),(0.5,0.2))
center1,radius1 = triangle_to_circumcenter(triangle1)
triangle2 = points_to_triangle((0.8,0.1),(0.7,0.5),(0.8,0.9))
center2,radius2 = triangle_to_circumcenter(triangle2)
plot_triangle([triangle1,triangle2],[center1,center2],[radius1,radius2],'two')
```

Вывод показан на рис. 7.7.

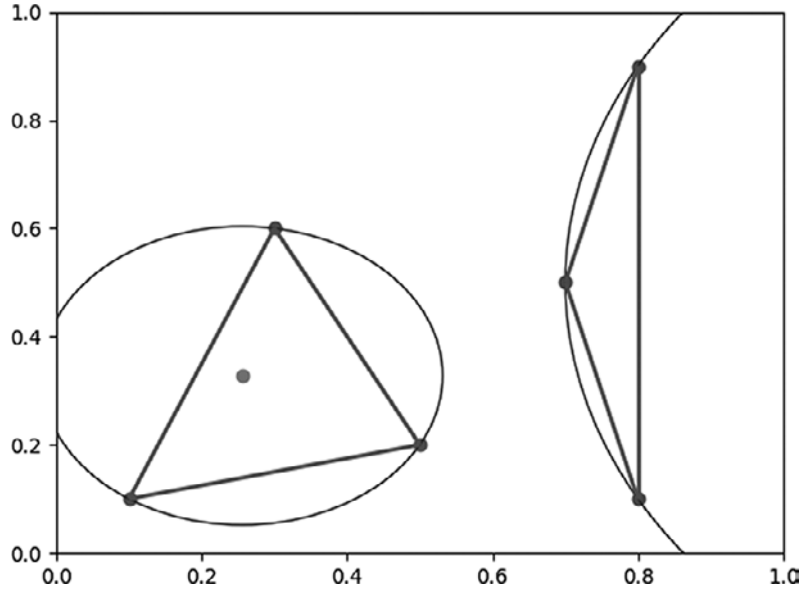


Рис. 7.7. Два треугольника с описанными окружностями и центрами описанных окружностей

Обратите внимание: первый треугольник почти равносторонний. Его описанная окружность мала, а ее центр лежит внутри. Второй треугольник — узкий и вытянутый. Его описанная окружность велика, а ее центр находится далеко за границами данного изображения. Каждый треугольник имеет уникальную описанную

окружность, и для разных треугольников создаются разные описанные окружности. Вы можете самостоятельно исследовать разные треугольники и описанные окружности, которые им соответствуют. Позднее различия между описанными окружностями треугольников начнут играть важную роль.

Триангуляция Делоне

Вы готовы к первому серьезному алгоритму данной главы. Он получает множество точек на входе и возвращает множество треугольников на выходе. В этом контексте преобразование множества точек во множество треугольников называется *триангуляцией*.

Функция `points_to_triangle()`, определенная ближе к началу главы, является простейшим алгоритмом триангуляции из всех возможных. Тем не менее этот алгоритм весьма ограничен, поскольку работает только в том случае, если ему передаются ровно три входные точки. Если вы хотите выполнить триангуляцию более чем для трех точек, то способов триангуляции неизбежно будет несколько. Например, рассмотрим два разных способа триангуляции для тех же семи точек, изображенных на рис. 7.8.

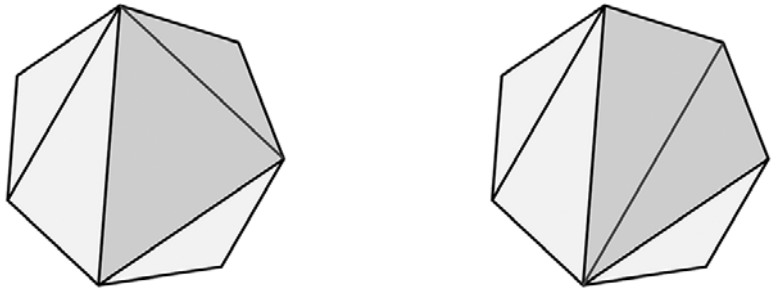


Рис. 7.8. Два разных способа триангуляции семи точек (Wikimedia Commons)

Более того, для этого правильного семиугольника существуют 42 возможных способа триангуляции (рис. 7.9).

Если у вас более семи точек и они не образуют правильную фигуру, то количество возможных триангуляций может достичь невероятных высот. Триангуляцию можно выполнить вручную, соединяя точки на бумаге с карандашом. Но, конечно, то же самое можно быстрее и лучше сделать с помощью алгоритма.

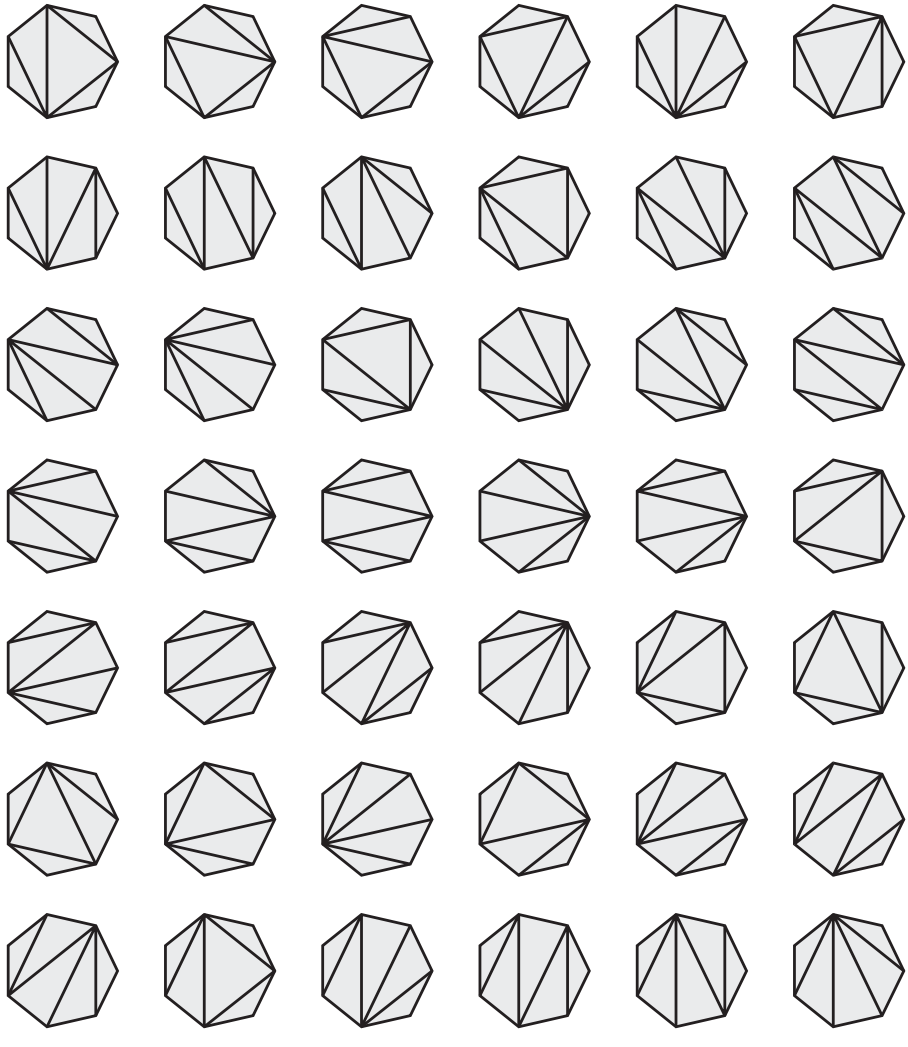


Рис. 7.9. Все 42 возможных способа триангуляции семи точек (источник: Википедия)

Существует несколько разных алгоритмов триангуляции. Для одних основным критерием была высокая скорость выполнения, для других — простота, третьи должны были порождать триангуляции, обладающие конкретными желательными свойствами. Рассматриваемый здесь алгоритм называется *алгоритмом Бауэра — Уотсона*; он получает на входе набор точек и выводит триангуляцию Делоне.

Триангуляция Делоне (ТД) отличается тем, что старается избегать узких вытянутых треугольников. Обычно выводятся треугольники, относительно близкие к равносторонним. Вспомните, что равносторонние треугольники имеют относительно небольшие описанные окружности, а у вытянутых треугольников эти окружности относительно велики. Учитывая этот факт, рассмотрим техническое определение триангуляции Делоне: для набора точек она выдает набор треугольников, соединяющих все точки, в котором ни одна точка не находится в описанной окружности каких-либо треугольников. Большие описанные окружности вытянутых треугольников с большой вероятностью будут охватывать одну или несколько других точек множества, поэтому правило, гласящее, что точки не могут располагаться внутри описанных окружностей, приводит к относительно небольшому количеству вытянутых треугольников. Если этот момент вам показался непонятным, то не переживайте — он будет наглядно продемонстрирован ниже.

Инкрементное генерирование триангуляций Делоне

Наша конечная цель — написать функцию, которая получает произвольный набор точек и выводит полную триангуляцию Делоне. Однако мы начнем с чего-то простого: напишем функцию, которая получает существующую ТД из n точек и еще одну точку, которая добавляется к ней и выводит ТД из $n + 1$ точек. С функцией «расширения Делоне» мы очень близко подойдем к написанию полной функции ТД.

ПРИМЕЧАНИЕ

Пример и изображения в этом подразделе любезно предоставлены LeatherBee (https://leatherbee.org/index.php/2018/10/06/terrain-generation-3-voronoi-diagrams/).

Сначала предположим, что у вас уже имеется ТД из девяти точек (рис. 7.10).

Теперь допустим, что в ТД добавляется десятая точка (рис. 7.11).

У ТД существует только одно правило: никакая точка не может лежать внутри описанной окружности любого из ее треугольников. Таким образом, мы проверяем описанную окружность каждого треугольника в существующей ТД, чтобы узнать, лежит ли точка 10 внутри какого-либо из них. Как выясняется, точка 10 лежит внутри описанных окружностей трех треугольников (рис. 7.12).

Эти треугольники уже не принадлежат ТД, поэтому мы исключаем их. В результате будет получена ТД, изображенная на рис. 7.13.

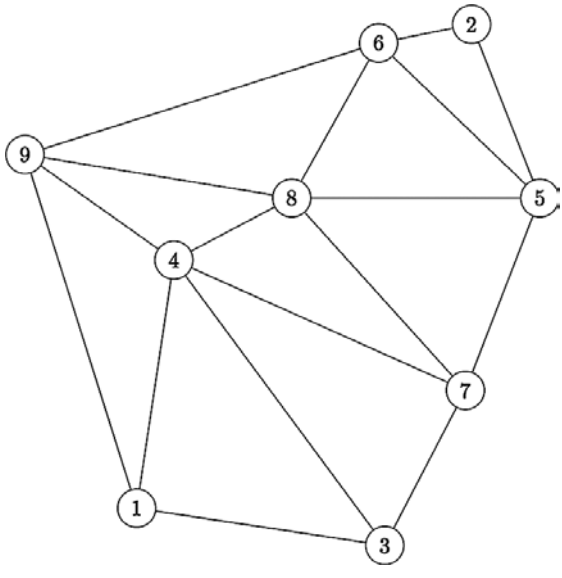


Рис. 7.10. Триангуляция Делоне с девятью точками

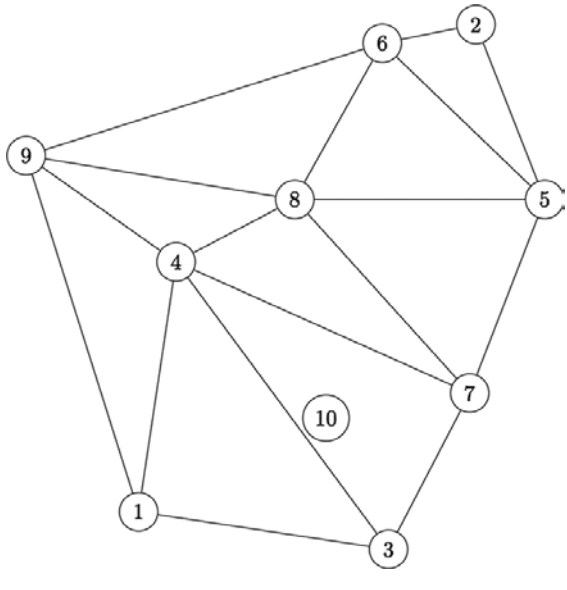


Рис. 7.11. Девятиточечная триангуляция Делоне с десятой добавленной точкой

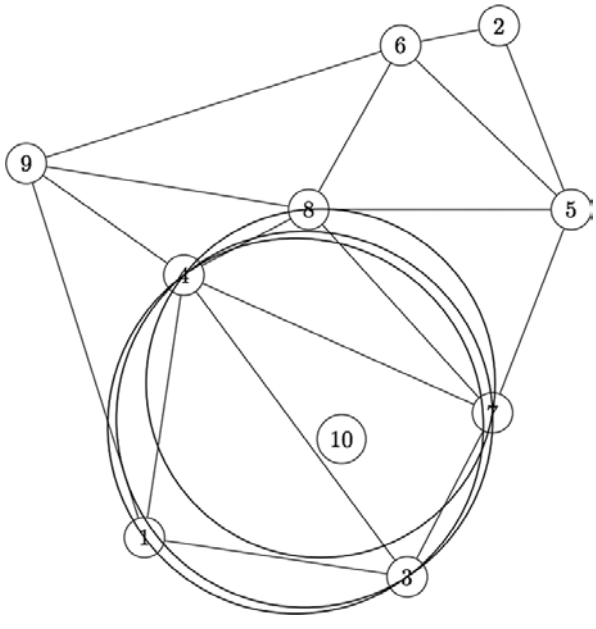


Рис. 7.12. Точка 10 находится внутри описанных окружностей трех треугольников в ТД

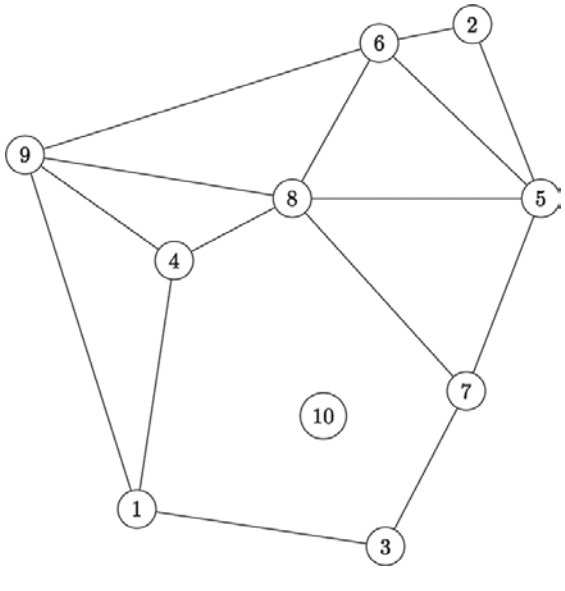


Рис. 7.13. Недействительные треугольники удалены

Работа еще не закончена. Необходимо заполнить появившуюся дыру и позаботиться о том, чтобы точка 10 была правильно соединена с другими точками. Если этого не сделать, то у нас не будет набора треугольников, а будут только точки и отрезки. Способ соединения точки 10 можно описать просто: мы добавляем ребро, соединяющее точку 10 с каждой вершиной самого большого пустого многоугольника, в границах которого лежит точка 10 (рис. 7.14).

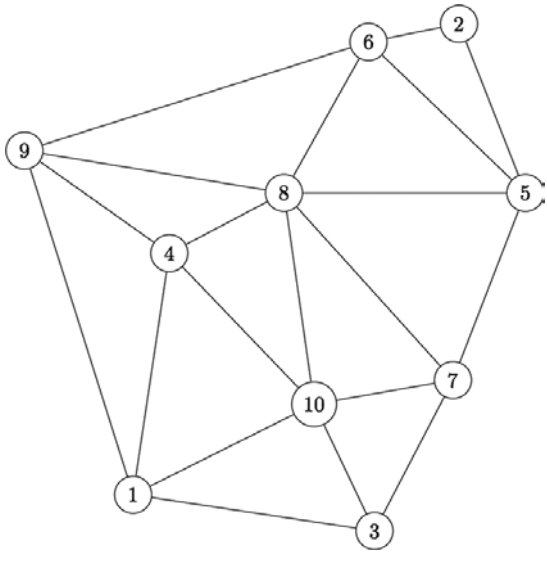


Рис. 7.14. Завершение десятиточечной ТД восстановлением связей действительных треугольников

Вуаля! Мы начали с девятиточечной ТД, добавили новую точку и получили десятиточечную ТД. Процесс может показаться прямолинейным. К сожалению, как это часто бывает с геометрическими алгоритмами, для того, что кажется ясным и интуитивным для человеческого глаза, может быть достаточно трудно написать код. Но это не должно страшить вас, храбрых искателей приключений.

Реализация триангуляций Делоне

Для начала предположим, что у нас уже есть ТД, которую мы назовем `delaunay`. Она представляет собой не что иное, как обычный список треугольников. Начать можно даже с одного треугольника:

```
delaunay = [points_to_triangle((0.2,0.8),(0.5,0.2),(0.8,0.7))]
```

Затем определим точку, которая добавляется в ТД; этой точке присваивается имя `point_to_add`:

```
point_to_add = [0.5,0.5]
```

Сначала необходимо определить, какие треугольники в существующей ТД (если они есть) стали недействительными, поскольку их описанная окружность содержит точку `point_to_add`. Мы будем действовать по схеме, которая представлена ниже.

1. Перебрать все треугольники в существующей ТД в цикле.
2. Для каждого треугольника найти центр и радиус описанной окружности.
3. Вычислить расстояние между `point_to_add` и центром окружности.
4. Если расстояние меньше радиуса описанной окружности, то новая точка находится внутри описанной окружности треугольника. Тогда можно сделать вывод, что треугольник является недействительным и его нужно исключить из ТД.

Эти действия реализуются следующим фрагментом кода:

```
import math
invalid_triangles = []
delaunay_index = 0
while delaunay_index < len(delaunay):
    circumcenter,radius = triangle_to_circumcenter(delaunay[delaunay_index])
    new_distance = get_distance(circumcenter,point_to_add)
    if(new_distance < radius):
        invalid_triangles.append(delaunay[delaunay_index])
    delaunay_index += 1
```

Фрагмент создает пустой список `invalid_triangles`, перебирает все треугольники в существующей ТД и проверяет, является ли текущий треугольник недействительным. Для этого он проверяет, что расстояние между `point_to_add` и центром описанной окружности меньше радиуса описанной окружности. Если треугольник недействителен, то он присоединяется к списку `invalid_triangles`.

Мы получаем список недействительных треугольников. Поскольку они недействительны, их нужно удалить. Позднее в ТД также потребуется добавить новые треугольники. Для этого будет удобно иметь список всех точек, присутствующих в одном из недействительных треугольников, так как эти точки будут входить в новые, действительные треугольники.

Следующий фрагмент удаляет из ТД все недействительные треугольники, а мы получаем набор образующих их точек.

```
    points_in_invalid = []

    for i in range(len(invalid_triangles)):
        delaunay.remove(invalid_triangles[i])
        for j in range(0,len(invalid_triangles[i])):
            points_in_invalid.append(invalid_triangles[i][j])
❶   points_in_invalid = [list(x) for x in set(tuple(x) for x in points_in_invalid)]
```

Сначала создается пустой список `points_in_invalid`. Затем перебирается содержимое списка `invalid_triangles`, при этом метод Python `remove()` используется для извлечения всех недействительных треугольников из существующей ТД. Затем все точки треугольника перебираются для добавления в список `points_in_invalid`. Наконец, так как в список `points_in_invalid` были добавлены повторяющиеся точки, мы используем списковое включение ❶ для воссоздания в `points_in_invalid` только точек с уникальными значениями.

Последний шаг алгоритма оказывается самым сложным. Необходимо добавить новые треугольники для замены недействительных. У каждого нового треугольника одной из точек является `point_to_add`, а две другие принадлежат существующей ТД. Но, конечно, мы не можем добавить все возможные комбинации `point_to_add` и двух существующих точек.

На рис. 7.13 и 7.14 можно заметить, что у всех новых треугольников, которые нужно было добавить, одной из точек была точка 10, а стороны выбирались из пустого многоугольника, содержащего ее. На визуальном уровне это выглядит достаточно просто, но написать код будет не так легко.

Необходимо найти простое геометрическое правило, которое легко объясняется в буквальном стиле интерпретации языка Python. Представьте себе правила, которые могут использоваться для генерирования новых треугольников на рис. 7.14. Как это часто бывает в математике, можно найти несколько эквивалентных наборов правил. Правила могут относиться к точкам, поскольку одно из определений треугольника — множество трех точек. Другие правила могут относиться к отрезкам, потому что в другом эквивалентном определении треугольник определяется как множество трех отрезков. Можно использовать любой набор правил, просто следует выбрать правила, которые проще всего понять и реализовать в коде. Как вариант можно рассмотреть все возможные комбинации точек недействительных треугольников с `point_to_add`, но один из таких треугольников должен добавляться только в том случае, если ребро, не содержащее `point_to_add`, встречается в списке недействительных треугольников ровно один раз. Это правило работает, поскольку ребрами, встречающимися ровно один раз, будут ребра внешнего многоугольника, окружающего новую точку (на рис. 7.13 такими ребрами будут ребра многоугольника, соединяющего точки 1, 4, 8, 7 и 3).

Правило реализуется с помощью данного кода:

```
for i in range(len(points_in_invalid)):
    for j in range(i + 1,len(points_in_invalid)):
        #count the number of times both of these are in the bad triangles
        count_occurrences = 0
        for k in range(len(invalid_triangles)):
            count_occurrences += 1 * (points_in_invalid[i] in
              invalid_triangles[k]) * \ (points_in_invalid[j] in
              invalid_triangles[k])
        if(count_occurrences == 1):
            delaunay.append(points_to_triangle(points_in_invalid[i],
              points_in_invalid[j], \ point_to_add))
```

Здесь перебираются все точки из `points_in_invalid`. Для каждой из них перебираются все последующие точки из `points_in_invalid`. Двойной цикл позволяет рассмотреть все комбинации из двух точек, входивших в недействительный треугольник. Для каждой комбинации мы перебираем все недействительные треугольники и считаем, сколько раз эти две точки встречаются в таком треугольнике. Если они обе присутствуют ровно в одном недействительном треугольнике, то мы заключаем, что они должны быть вместе в одном из новых треугольников; в ТД добавляется новый треугольник, состоящий из этих двух точек и новой точки.

Мы выполнили действия, необходимые для добавления новой точки в существующую ТД. Таким образом, мы можем взять ТД из n точек, добавить новую точку и получить ТД из $n + 1$ точки. Теперь необходимо пользоваться этой возможностью, чтобы взять набор из n точек и построить ТД «на пустом месте», от 0 до n точек. После начала построения ТД все происходит достаточно просто: необходимо лишь в цикле повторять процесс, переходящий от n точек к $n + 1$ точке снова и снова, пока не будут добавлены все точки.

Остается только одно усложнение. По причинам, которые будут рассмотрены позднее, мы хотим добавить еще три точки в набор точек, для которых генерируется ТД. Эти точки будут лежать далеко снаружи от выбранных точек; чтобы выполнить данное требование, мы найдем левые верхние точки и добавим новую точку выше и левее от них, затем проделаем то же самое для нижних правых и нижних левых точек. Эти точки станут первым треугольником ТД. Мы начнем с ТД, соединяющей три точки: это точки только что упомянутого нового треугольника. Затем уже продемонстрированная выше логика превратит ТД из трех точек в ТД из четырех точек, затем из пяти и т. д., пока не будут добавлены все точки.

В листинге 7.3 код, написанный ранее, объединяется для создания функции `gen_delaunay()`, которая получает на входе набор точек и выводит полную ТД.

Листинг 7.3. Функция, которая получает набор точек и возвращает триангуляцию Делоне

```
def gen_delaunay(points):
    delaunay = [points_to_triangle([-5,-5],[-5,10],[10,-5])]
    number_of_points = 0


    while number_of_points < len(points): ❶
        point_to_add = points[number_of_points]

        delaunay_index = 0

        invalid_triangles = [] ❷
        while delaunay_index < len(delaunay):
            circumcenter,radius = triangle_to_circumcenter(delaunay[
                                  delaunay_index])
            new_distance = get_distance(circumcenter,point_to_add)
            if(new_distance < radius):
                invalid_triangles.append(delaunay[delaunay_index])
            delaunay_index += 1

        points_in_invalid = [] ❸
        for i in range(0,len(invalid_triangles)):
            delaunay.remove(invalid_triangles[i])
            for j in range(0,len(invalid_triangles[i])):
                points_in_invalid.append(invalid_triangles[i][j])
        points_in_invalid = [list(x) for x in set(tuple(x) for x
                             in points_in_invalid)]

        for i in range(0,len(points_in_invalid)): ❹
            for j in range(i + 1,len(points_in_invalid)):
                # Подсчитать количество вхождений обеих точек
                # в недействительные треугольники
                count_occurrences = 0
                for k in range(0,len(invalid_triangles)):
                    count_occurrences += 1 * (points_in_invalid[i]
                                              in invalid_triangles[k]) * \
                    (points_in_invalid[j] in invalid_triangles[k])
                if(count_occurrences == 1):
                    delaunay.append(points_to_triangle(points_in_invalid[i], \
                      points_in_invalid[j], point_to_add))

        number_of_points += 1

    return(delaunay)
```

Полная функция генерирования ТД начинает работу с добавления нового треугольника, о чем говорилось ранее. Затем перебирает все точки в коллекции

точек ❶. Для каждой точки создается список недействительных треугольников: всех треугольников из ТД, описанная окружность которых включает текущую точку ❷. Недействительные треугольники удаляются из ТД, после чего создается коллекция точек с использованием всех точек, входивших в недействительные треугольники ❸. Затем по этим точкам создаются новые треугольники, следующие правилам триангуляций Делоне ❹. Процедура выполняется поэтапно, задействуя приведенный выше код. Наконец, функция возвращает `delaunay` — список, содержащий коллекцию треугольников, образующих ТД.

Эта функция может вызываться для генерирования ТД для произвольного набора точек. Следующий код задает значение `N` и генерирует `N` случайных точек (значения x и y). Затем значения x и y объединяются вызовом `zip()`, помещаются в список и передаются функции `gen_delaunay()`. Функция возвращает полную действительную ТД, которая сохраняется в переменной `the_delaunay`:

```
N=15
import numpy as np
np.random.seed(5201314)
xs = np.random.rand(N)
ys = np.random.rand(N)
points = zip(xs,ys)
listpoints = list(points)
the_delaunay = gen_delaunay(listpoints)
```

Переменная `the_delaunay` будет использоваться в следующем разделе для генерирования диаграммы Вороного.

От триангуляции Делоне к диаграмме Вороного

Итак, алгоритм генерирования ТД завершен, и до алгоритма генерирования диаграммы Вороного рукой подать. Набор точек можно преобразовать в диаграмму Вороного по алгоритму, представленному ниже.

1. Найти ТД для набора точек.
2. Найти центр описанной окружности для каждого треугольника в ТД.
3. Провести линии, соединяющие центры описанных окружностей для всех треугольников ТД, имеющих общую сторону.

Мы уже знаем, как выполнить шаг 1 (поскольку проделали это в предыдущем разделе), а шаг 2 можно выполнить с помощью функции `triangle_to_circumcenter()`. Таким образом, остается написать фрагмент кода для реализации шага 3.

Код, который мы напишем для шага 3, будет размещаться в функции графического вывода. Напомню, что на вход этой функции подается набор треугольников и центры описанных окружностей. Наш код должен создать набор линий, соединяющих центры описанных окружностей. Но он будет соединять не все центры, а только центры описанных окружностей для треугольников, имеющих общую сторону.

Треугольники хранятся в виде набора точек, а не сторон. Но мы все равно можем проверить, имеют ли два треугольника общую сторону; мы только проверяем, имеют ли они ровно две общие точки. Если они имеют только одну общую точку, то соприкасаются вершинами, но не имеют общей стороны. Если у них три общих точки, значит, это одинаковые треугольники, у которых центры описанных окружностей совпадают. Наш код перебирает все треугольники, и для каждого из них снова перебирает все треугольники и проверяет количество точек, общих для двух треугольников. Если общих точек ровно две, то добавляется линия между центрами описанных окружностей этих треугольников. Линии между центрами станут границами диаграммы Вороного. Следующий фрагмент кода показывает, как перебирать треугольники, но он является лишь частью большой функции, поэтому пока не пытайтесь его выполнять:

```
------
for j in range(len(triangles)):
    commonpoints = 0
    for k in range(len(triangles[i])):
        for n in range(len(triangles[j])):
            if triangles[i][k] == triangles[j][n]:
               commonpoints += 1
    if commonpoints == 2:
        lines.append([list(centers[i][0]),list(centers[j][0])])
```

Код добавляется в функцию графического вывода, так как конечной целью функции является рисование диаграммы Вороного.

Заодно можно внести ряд полезных дополнений в функцию. Новый код графической функции приведен в листинге 7.4, изменения выделены жирным шрифтом.

Листинг 7.4. Функция для рисования треугольников, описанных окружностей, их центров, точек Вороного и регионов Вороного

```
def plot_triangle_circum(triangles,centers,plotcircles,plotpoints, \
plottriangles,plotvoronoi,plotvpoints,thename):
    fig, ax = pl.subplots()
    ax.set_xlim([-0.1,1.1])
    ax.set_ylim([-0.1,1.1])

    lines=[]
```

```
for i in range(0,len(triangles)):
    triangle = triangles[i]
    center = centers[i][0]
    radius = centers[i][1]
    itin = [0,1,2,0]
    thelines = genlines(triangle,itin)
    xs = [triangle[0][0],triangle[1][0],triangle[2][0]]
    ys = [triangle[0][1],triangle[1][1],triangle[2][1]]

    lc = mc.LineCollection(genlines(triangle,itin), linewidths=2)
    if(plottriangles):
        ax.add_collection(lc)
    if(plotpoints):
        pl.scatter(xs, ys)

    ax.margins(0.1)

  ❶ if(plotvpoints):
        pl.scatter(center[0],center[1])

    circle = pl.Circle(center, radius, color = 'b', fill = False)
    if(plotcircles):
        ax.add_artist(circle)

  ❷ if(plotvoronoi):
        for j in range(0,len(triangles)):
            commonpoints = 0
            for k in range(0,len(triangles[i])):
                for n in range(0,len(triangles[j])):
                    if triangles[i][k] == triangles[j][n]:
                        commonpoints += 1
            if commonpoints == 2:
                lines.append([list(centers[i][0]),list(centers[j][0])])

    lc = mc.LineCollection(lines, linewidths = 1)

    ax.add_collection(lc)

pl.savefig(str(thename) + '.png')
pl.close()
```

Сначала мы добавляем новые аргументы, которые точно определяют, что именно нужно нарисовать. Вспомните: в этой главе мы работали с точками, сторонами, треугольниками, описанными окружностями, центрами описанных окружностей, ТД и регионами Вороного. Вывод сразу всех элементов может оказаться избыточным, поэтому мы добавим флаги, управляющие выводом информации: аргумент `plotcircles` для кругов, `plotpoints` для точек, `plottriangles` для ТД, `plotvoronoi` для сторон диаграммы Вороного и `plotvpoints` для центров описанных окружностей (которые являются вершинами ребер диаграммы Вороного). Новые фрагменты

выделены жирным шрифтом. Одно дополнение выводит вершины диаграммы Вороного (центры описанных окружностей), если аргументы указывают, что они должны выводиться ❶. Другое, более длинное, дополнение выводит стороны диаграммы Вороного ❷. Были также добавлены команды `if`, которые позволяют управлять тем, должна ли программа рисовать треугольники, вершины и центры описанных окружностей на усмотрение пользователя.

Почти все готово к тому, чтобы вызвать функцию графического вывода и увидеть итоговую диаграмму Вороного. Тем не менее сначала необходимо получить центры описанных окружностей для каждого треугольника ТД. К счастью, это делается очень просто. Мы можем создать пустой список `circumcenters` и присоединить к нему центр описанной окружности каждого треугольника в нашей ТД:

```
circumcenters = []
for i in range(0,len(the_delaunay)):
    circumcenters.append(triangle_to_circumcenter(the_delaunay[i]))
```

Наконец, мы вызовем функцию графического вывода и укажем, что она должна выводить границы регионов Вороного:

```
plot_triangle_circum(the_delaunay,circumcenters,False,True,False,True,False,'final')
```

Результат показан на рис. 7.15.

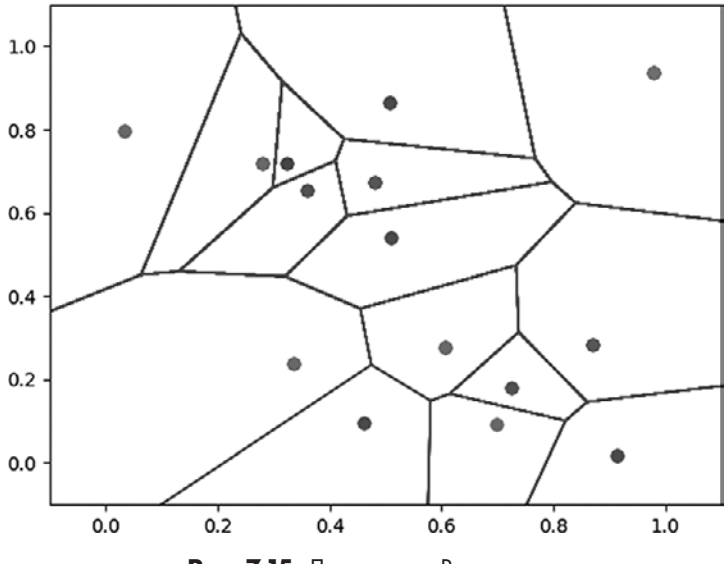


Рис. 7.15. Диаграмма Вороного

Набор точек был преобразован в диаграмму Вороного за считаные секунды. Мы видим, что границы в ней проходят прямо по краю диаграммы. Если увеличить размер изображения, то границы пройдут далее. Напомню, что стороны регионов Вороного соединяют центры описанных окружностей для треугольников, входящих в ТД. Но наша ТД может соединять небольшое количество точек, близких к центру диаграммы, так что все центры описанных окружностей будут лежать в пределах небольшой области в середине диаграммы. В таком случае границы на диаграмме Вороного не будут простираться до границ диаграммы. Именно из-за этого мы добавили новый внешний треугольник в первой строке функции `gen_delaunay()`; с треугольником, точки которого лежат далеко за пределами диаграммы, можно быть уверенным в том, что на диаграмме всегда будут стороны регионов, которые доходят до края карты, поэтому (например) мы будем знать, за каким почтовым отделением закрепить новые пригороды, построенные за городской чертой.

Возможно, вам захочется поэкспериментировать с новой функцией. Например, если присвоить всем аргументам значение `True`, то будет построен громоздкий, но красивый график со всеми элементами, рассмотренными в этой главе:

```
plot_triangle_circum(the_delaunay,circumcenters,True,True,True,True,True,'everything')
```

Вывод представлен на рис. 7.16.

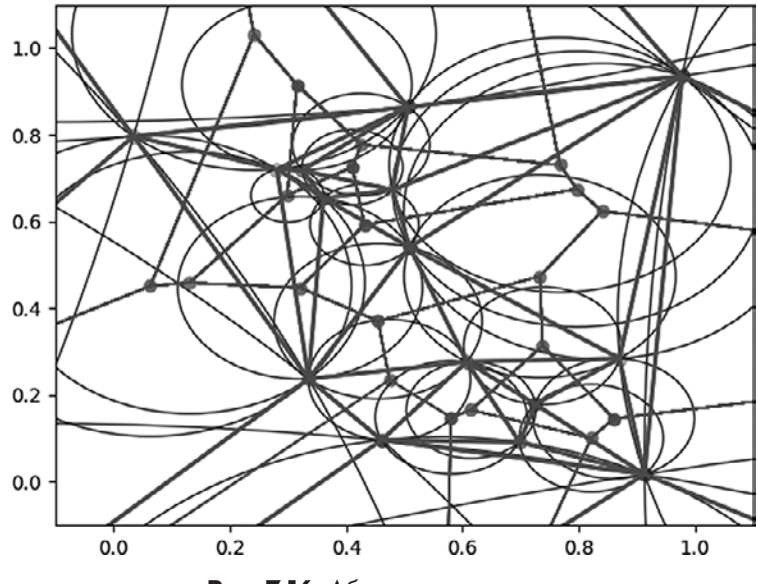


Рис. 7.16. Абстрактная картина

С помощью этого изображения вы можете убедить своих соседей и членов семьи, что участвуете в сверхсекретном проекте CERN по анализу столкновений частиц или же подать заявку на стипендию для одаренных художников как духовный наследник Пита Мондриана. При взгляде на диаграмму Вороного с триангуляцией Делоне и описанными окружностями можно представить себе почтовые отделения, водные колонки, кристаллические структуры или другие возможные применения диаграммы Вороного. А можно представить точки, треугольники и отрезки и наслаждаться чистой радостью геометрии.

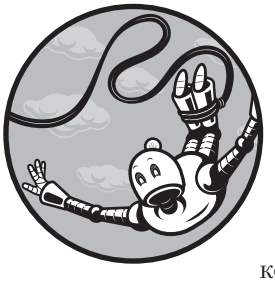
Резюме

В этой главе были описаны методы программирования геометрических выкладок. Мы начали с простых точек, линий и треугольников. Затем перешли к разным способам нахождения центра треугольников и возможностям генерирования триангуляции Делоне для произвольного набора точек. Наконец, рассмотрели простую процедуру применения триангуляции Делоне для генерирования диаграммы Вороного, которая может использоваться для решения задачи почтмейстера или в других областях. У нее есть свои нюансы, но в конечном итоге все сводится к элементарным манипуляциям с точками, линиями и треугольниками.

В следующей главе рассматриваются алгоритмы, которые могут использоваться для работы с языками. В частности, вы узнаете, как алгоритм может исправить текст с отсутствующими пробелами и как написать программу, которая способна спрогнозировать следующее слово в предложении.

8

Язык



В этой главе мы шагнем в неформальный мир естественного языка. Сначала обсудим различия между языками и математикой, усложняющие построение языковых алгоритмов. Затем построим алгоритм вставки пробелов, который может взять текст на любом языке и вставить любые недостающие пробелы. После этого построим алгоритм завершения фраз, способный имитировать стиль писателя и найти наиболее подходящее следующее слово.

Алгоритмы этой главы сильно зависят от двух инструментов, которыми ранее мы еще не пользовались: *списковых включений* (list comprehensions) и *корпусов текстов*. Списковые включения позволяют быстро генерировать списки в логике циклов и итераций. Они оптимизированы для очень быстрого выполнения в Python, легко и компактно записываются, но их трудно читать, а к их синтаксису нужно привыкнуть. Корпус представляет собой массив текста, по которому наш алгоритм «обучается» языку и стилю.

Почему языковые алгоритмы сложны

Применение алгоритмического мышления в естественных языках восходит еще ко временам Декарта, который заметил, что, хотя количество существующих чисел

бесконечно, любой человек с базовым пониманием арифметики знает, как создать или интерпретировать никогда не встречавшееся ему ранее число. Допустим, вы никогда не сталкивались с числом 14 326 — никогда не считали до него, никогда не читали финансовые отчеты с такими суммами, никогда не нажимали именно такие клавиши на клавиатуре. Но я уверен, что вы легко сможете понять, много это или мало, какие числа больше, а какие меньше этого числа и как им оперировать в уравнениях.

Алгоритм, который позволяет легко понять до сих пор неизвестные числа, состоит из комбинации десяти цифр (0–9), запоминаемых по порядку, и системы позиционирования. Мы знаем, что число 14 326 на 1 больше 14 325, поскольку цифра 6 по порядку следует за цифрой 5, они занимают одинаковую позицию в соответствующих числах, а цифры во всех остальных позициях совпадают. Знание цифр и система позиционирования позволяет нам мгновенно получить представление о том, насколько число 14 326 близко к 14 325, что оба числа больше 12, но меньше 1 000 000. Можно также мгновенно понять, что число 14 326 в некоторых отношениях похоже на 4326, но сильно отличается от него по величине.

С языком дело обстоит иначе. Если вы изучаете английский и впервые встречаете слово *stage*, то не сможете сделать какие-то выводы о его смысле, просто заметив его сходство со словами *stale*, *stake*, *state*, *stave*, *stade* или *sage*, хотя эти слова отличаются от *stage* приблизительно в такой же степени, в какой 14 326 отличается от 14 325. Нельзя сделать и вывод о том, что бактерия больше лося, просто на основании количества слогов и букв в словах. Даже надежные на первый взгляд правила — например, добавление *s* для образования множественного числа в английском — может сильно подвести, если вы сделаете вывод, что *princess* является одиночным числом от *princess*.

Чтобы использовать алгоритмы применительно к языку, необходимо либо упростить его, чтобы короткие математические алгоритмы могли с ним надежно работать, либо усовершенствовать алгоритмы, чтобы они могли справиться с хаотичной сложностью человеческого языка, появившейся в ходе его естественного развития. Мы займемся этим позднее.

Расстановка пробелов

Представьте, что работаете старшим специалистом по алгоритмам в большой и известной компании, у которой целый склад забит документами, написанными от руки. Специалист по оцифровке ведет долгосрочный проект по сканированию этих документов и переводу в графические с последующим применением технологий распознавания текста для преобразования графики в текст, который удобно

хранить в базах данных компании. Тем не менее некоторые бумаги написаны ужасным почерком, а технология распознавания текста неидеальна, так что итоговый текст, который извлекается из бумажных оригиналов, иногда несовершенен. Вам передают оцифрованный текст. Ваша задача — найти ошибки, не обращаясь к бумажным оригиналам.

Допустим, вы читаете первое оцифрованное предложение в программе Python и обнаруживаете, что это цитата из Г. К. Честертона. Неидеально оцифрованный текст сохраняется в переменной `text`:

```
text = "The oneperfectly divine thing, the oneglimpse of God's paradisegiven
on earth, is to fight a losingbattle - and notlose it."
```

Текст написан на английском языке, и хотя каждое слово распознано правильно, между ними отсутствуют некоторые пробелы: вместо `oneperfectly` должно быть `one perfectly`, вместо `paradisegiven` — `paradise given` и т. д. (Для человека пропускать пробелы нетипично, но технология распознавания текста часто совершает подобные ошибки.) Чтобы справиться с задачей, необходимо вставить пробелы в нужных местах текста. Для человека, хорошо владеющего английским, эта задача не составляет особых проблем. Однако представьте, что вам необходимо быстро расставить пробелы на миллионах отсканированных страниц — очевидно, необходимо написать алгоритм, который сделает это за вас.

Определение списка слов и поиск слов

Первым делом необходимо обучить алгоритм английским словам. Сделать это несложно: определим список `word_list` и заполним его словами. Начнем с нескольких слов:

```
word_list = ['The','one','perfectly','divine']
```

В этой главе мы будем создавать списки и работать с ними с помощью списковых включений. Скорее всего, списковые включения понравятся вам после того, как вы привыкнете к ним. Ниже приведено очень простое списковое включение для создания копии нашего списка `word_list`:

```
word_list_copy = [word for word in word_list]
```

Как видите, синтаксис `for word in word_list` очень похож на синтаксис цикла `for`. Тем не менее здесь не нужны ни двоеточие, ни лишние строки. В данном случае списковое включение сделано по возможности простым: оно всего лишь указывает, что каждое слово из `word_list` должно находиться в нашем новом списке

word_list_copy. Возможно, вам это покажется тривиальным, но вы можете добавить логику, чтобы сделать его более полезным. Например, если вы хотите включить каждое слово из списка, содержащее букву *n*, то для этого достаточно добавить простую команду if:

```
has_n = [word for word in word_list if 'n' in word]
```

Выполните команду print(has_n), чтобы увидеть, что включение выдает ожидаемый результат:

```
['one', 'divine']
```

Позднее в этой главе будут приведены более сложные списковые включения, в том числе и с вложенными циклами. Однако все они строятся по одной схеме: цикл for определяет перебор, а необязательная команда if описывает логику выбора элементов для итогового списка.

Для обращения к функциональности работы с текстом мы будем использовать модуль Python re. Одна из полезных функций re — finditer() — может использоваться для поиска в тексте позиции любого слова из списка word_list. Пример использования finditer() в списковом включении выглядит так:

```
import re
locs = list(set([(m.start(),m.end()) for word in word_list for m in
re.finditer(word, text)]))
```

Строка получилась слишком насыщенной; уделим ей немного времени, чтобы вы лучше поняли ее. Мы определяем переменную locs (сокращение от locations); она содержит позиции каждого слова из списка в тексте. Для получения этого списка будет использоваться списковое включение.

Списковое включение размещается в квадратных скобках ([]). Конструкция for word in word_list перебирает все слова из списка word_list. Для каждого слова вызывается функция re.finditer(), которая находит выбранное слово в тексте и возвращает список всех позиций, в которых оно встречается. Эти позиции последовательно перебираются, и каждая отдельная позиция сохраняется в m. При обращении к m.start() и m.end() мы получаем соответственно позицию начала и конца слова в тексте. Обратите внимание на порядок циклов for — и привыкните к нему, поскольку некоторые люди считают его обратным тому, что они ожидали.

Все списковое включение заключается в list(set()). Это удобный способ получить список, который содержит только уникальные значения без дубликатов. Наше

списковое включение может состоять из нескольких идентичных элементов, но при преобразовании его во множество автоматически удаляются дубликаты, после чего оно снова преобразуется в список, чтобы данные были сохранены в нужном формате: списке уникальных позиций слов.

Выполните команду `print(locs)`, чтобы просмотреть результат всей операции:

```
[(17, 23), (7, 16), (0, 3), (35, 38), (4, 7)]
```

В Python такие упорядоченные пары называются *кортежами*. Они показывают позиции всех слов из `word_list` в тексте. Например, при выполнении выражения `text[17:23]` (с числами из третьего кортежа в предыдущем списке) выясняется, что это слово `divine`. Буква `d` является 17-м символом текста, буква `i` — 18-м символом и т. д. до буквы `e`, 22-го символа текста, поэтому кортеж закрывается значением 23. Вы можете убедиться в том, что другие кортежи также обозначают позиции слов в списке `word_list`.

Выражение `text[4:7]` определяет слово `one`, а `text[7:16]` — слово `perfectly`. Конец слова `one` стыкуется с началом слова `perfectly` без разделительного пробела. Если вы не заметили этого при чтении текста, то это можно обнаружить по кортежам `(4, 7)` и `(7, 16)` в переменной `locs`: так как 7 является вторым элементом `(4, 7)`, а также первым элементом `(7, 16)`, мы знаем, что одно слово завершается по индексу, с которого начинается другое слово. Чтобы найти места, в которых нужно вставить пробелы, мы будем искать такие случаи: когда конец одного действительного слова находится в одной позиции с началом другого действительного слова.

Составные слова

К сожалению, два допустимых слова, следующих друг за другом без пробела, не являются однозначным доказательством отсутствия пробела. Возьмем слово *butterly*. И *butter*, и *fly* являются допустимыми словами, но из этого не следует, что слово *butterly* написано с ошибкой, поскольку *butterly* тоже является допустимым словом. Следовательно, необходимо проверить не только допустимые слова, следующие друг за другом без пробела, но и убедиться в том, что допустимые слова, не разделенные пробелом, не образуют другое допустимые слово. Это означает, что в тексте нужно проверить, не является ли словом `oneperfectly`, не является ли словом `paradisegiven` и т. д.

Чтобы проверить это условие, необходимо найти все пробелы в тексте. Рассмотрим все подстроки между двумя последовательными пробелами и назовем их потенциальными словами. Если потенциальное слово не входит в список слов,

то можно сделать вывод, что его не существует. Далее проверим каждое недействительное слово и определим, не является ли оно комбинацией двух меньших слов; если является, то делаем вывод, что здесь отсутствует пробел, и добавляем его между двумя допустимыми словами, которые были соединены с образованием недопустимого.

Проверка потенциальных слов между существующими пробелами

Снова воспользуемся функцией `re.finditer()` для нахождения всех пробелов в тексте, которые будут сохранены в переменной `spacestarts`. Добавим в переменную `spacestarts` и еще два элемента: один представляет позицию начала текста, а другой — позицию конца текста. Это гарантирует, что будут найдены все потенциальные слова, так как слова в начале и конце будут единственными словами, не заключенными между пробелами. Кроме того, мы добавим строку, которая сортирует список `spacestarts`:

```
spacestarts = [m.start() for m in re.finditer(' ', text)]
spacestarts.append(-1)
spacestarts.append(len(text))
spacestarts.sort()
```

Список `spacestarts` сохраняет позиции пробелов в тексте. Для получения этих позиций используются списковое включение и функция `re.finditer()`. В этом случае `re.finditer()` находит позицию каждого пробела в тексте и сохраняет ее в списке, при этом каждый отдельный элемент обозначается `m`. Для каждого из этих элементов `m`, которые являются пробелами, мы получаем начальную позицию пробела с помощью функции `start()`. Мы ищем потенциальные слова между пробелами. Будет полезно иметь еще один список, в котором сохраняются позиции символов, следующих после пробела; это будут позиции первого символа каждого потенциального слова. Назовем этот список `spacestarts_affine`, поскольку в технической терминологии этот новый список является аффинным преобразованием списка `spacestarts`. Термином *«аффинный»* часто обозначаются линейные преобразования — например, прибавление 1 к каждой позиции, как это делается здесь. Вдобавок отсортируем этот список:

```
spacestarts_affine = [ss+1 for ss in spacestarts]
spacestarts_affine.sort()
```

Затем получим все подстроки, заключенные между двумя пробелами:

```
between_spaces = [(spacestarts[k] + 1,spacestarts[k + 1]) for k in
range(0,len(spacestarts) - 1 )]
```

Создаваемая здесь переменная называется `between_spaces`; она содержит список кортежей в форме (`<позиция начала подстроки>, <позиция конца подстроки>`) — например, `(17, 23)`. Для получения этих кортежей используется списковое включение, выполняющее перебор по `k`. В данном случае `k` принимает значения целых чисел от 0 до длины списка `spacestarts`, уменьшенной на 1. Для каждого `k` генерируется один кортеж. Первый элемент кортежа содержит `spacestarts[k]+1` позицию, следующую за позицией каждого пробела. Вторым элементом кортежа является `spacestarts[k+1]` — позиция следующего пробела в тексте. Таким образом, результат содержит кортежи, обозначающие начало и конец каждой подстроки между пробелами.

Теперь рассмотрим все потенциальные слова между пробелами и определим, какие из них являются недействительными (то есть не входят в список слов):

```
between_spaces_notvalid = [loc for loc in between_spaces if \
text[loc[0]:loc[1]] not in word_list]
```

Обратившись к `between_spaces_notvalid`, мы видим, что это список позиций всех недействительных потенциальных слов в тексте:

```
[(4, 16), (24, 30), (31, 34), (35, 45), (46, 48), (49, 54), (55, 68), (69,
71), (72, 78), (79, 81), (82, 84), (85, 90), (91, 92), (93, 105), (106, 107),
(108, 111), (112, 119), (120, 123)]
```

Наш код полагает, что все эти позиции относятся к недействительным словам. Однако если взглянуть на некоторые из упоминаемых слов, то они выглядят вполне нормальными. Например, выражение `text[103:106]` выводит действительное слово `and`. Почему же наш код считает это слово недействительным? Потому что оно не входит в список слов. Конечно, можно добавить его в список слов вручную и продолжить действовать так и далее. Но помните, что наш алгоритм вставки пробелов должен работать для миллионов страниц отсканированного текста, и они могут содержать много тысяч уникальных слов. Было бы полезно импортировать список слов, который уже содержит обширный набор действительных слов английского языка. Такой набор слов называется *корпусом*.

Использование импортированного корпуса для проверки действительных слов

К счастью, существуют модули Python, которые позволяют импортировать полный корпус всего в нескольких строках. Сначала необходимо загрузить корпус:

```
import nltk
nltk.download('brown')
```

Мы загрузили корпус `brown` из модуля `nltk`. Затем корпус импортируется в программу:

```
from nltk.corpus import brown
wordlist = set(brown.words())
word_list = list(wordlist)
```

Мы импортировали корпус и преобразовали его набор слов в список Python. Но прежде чем использовать новый список `word_list`, следует выполнить проверку, чтобы удалить знаки препинания, которые ошибочно считаются словами:

```
word_list = [word.replace('*','') for word in word_list]
word_list = [word.replace('[','') for word in word_list]
word_list = [word.replace(']','') for word in word_list]
word_list = [word.replace('?','') for word in word_list]
word_list = [word.replace('.','') for word in word_list]
word_list = [word.replace('+','') for word in word_list]
word_list = [word.replace('/','') for word in word_list]
word_list = [word.replace(';','') for word in word_list]
word_list = [word.replace(':','') for word in word_list]
word_list = [word.replace(',','') for word in word_list]
word_list = [word.replace(')','') for word in word_list]
word_list = [word.replace('(','') for word in word_list]
word_list.remove('')
```

Эти строки используют функции `remove()` и `replace()` для замены знаков препинания пустыми строками, после чего эти строки удаляются. Теперь, когда у нас имеется подходящий список слов, мы сможем точнее распознавать недействительные слова. Можно снова провести проверку недействительных слов по новому списку `word_list` и получить улучшенный результат:

```
between_spaces_notvalid = [loc for loc in between_spaces if \
text[loc[0]:loc[1]] not in word_list]
```

При выводе `between_spaces_notvalid` будет получен более короткий и более точный список:

```
[(4, 16), (24, 30), (35, 45), (55, 68), (72, 78), (93, 105), (112, 119), (120, 123)]
```

Итак, мы нашли недействительные потенциальные слова в тексте. Теперь можно проверить список на наличие слов, которые могут объединяться для формирования этих недействительных слов. Начать стоит с поиска слов, начинающихся после пробела; они могут быть первой половиной недействительного слова.

```
partial_words = [loc for loc in locs if loc[0] in spacestarts_affine and \
loc[1] not in spacestarts]
```

Списковое включение перебирает все элементы переменной `locs`, содержащей позиции всех слов в списке. Оно проверяет, присутствует ли `locs[0]` (начало слова) в `spacestarts_affine` — списке, содержащем символы, следующие после пробела. Затем она проверяет, что `loc[1]` не присутствует в `spacestarts` — то есть проверяет, завершается ли слово там, где начинается пробел. Если слово начинается после пробела и не завершается в позиции с пробелом, то помещается в переменную `partial_words`, поскольку может являться словом, после которого следует вставить пробел.

Теперь рассмотрим слова, завершающиеся пробелом. Они могут быть второй половиной недействительного слова. Чтобы найти их, внесем небольшие изменения в предыдущую логику:

```
partial_words_end = [loc for loc in locs if loc[0] not in spacestarts_affine \
and loc[1] in spacestarts]
```

Теперь можно переходить к вставке пробелов.

Поиск первой и второй половин потенциальных слов

Начнем с вставки пробела в `oneperfectly`. Определим переменную `loc`, в которой хранится позиция `oneperfectly` в тексте:

```
loc = between_spaces_notvalid[0]
```

Теперь необходимо проверить, могут ли какие-либо из слов в `partial_words` быть первой половиной `oneperfectly`. Чтобы действительное слово было первой половиной `oneperfectly`, оно должно иметь ту же начальную позицию в тексте, что и `oneperfectly`, но другую конечную позицию. Мы напишем списковое включение, которое находит конечную позицию каждого действительного слова, начинающегося в той же позиции, что и `oneperfectly`:

```
endsofbeginnings = [loc2[1] for loc2 in partial_words if loc2[0] == loc[0] \
and (loc2[1] - loc[0]) > 1]
```

Присваивание `loc2[0] == loc[0]` означает, что действительное слово должно начинаться в той же позиции, что и `oneperfectly`. Кроме того, указано условие `(loc2[1]-loc[0])>1`, что гарантирует, что длина найденного действительного слова превышает один символ. Это не является абсолютно необходимым, но помогает избежать ложных положительных срабатываний. Для примера можно привести такие слова, как *avoid*, *aside*, *along*, *irate* и *iconic*, в которых первую букву можно считать самостоятельным словом — но, наверное, не стоит.

Наш список `endsofbeginnings` должен включать конечную позицию каждого действительного слова, начинающегося в той же позиции, что и `oneperfectly`. Воспользуемся списковым включением для создания похожей переменной `beginningsofends`, которая будет включать начальные позиции всех действительных слов, завершающихся в одной позиции с `oneperfectly`:

```
beginningsofends = [loc2[0] for loc2 in partial_words_end if loc2[1] == loc[1] and \
(loc2[1] - loc[0]) > 1]
```

Условие `loc2[1] == loc[1]` означает, что действительное слово должно заканчиваться в той же позиции, что и `oneperfectly`. Вдобавок задано условие `(loc2[1]-loc[0])>1`, которое гарантирует, что найденное действительное слово имеет длину более одного символа, как и прежде.

Работа почти закончена. Остается определить, присутствуют ли какие-либо позиции как в `endsofbeginnings`, так и `beginningsofends`. Если это так, то, значит, недействительное слово действительно является комбинацией двух действительных слов без пробела. Мы можем воспользоваться функцией `intersection()` для нахождения всех элементов, общих для двух списков:

```
pivot = list(set(endsofbeginnings).intersection(beginningsofends))
```

Здесь снова используется синтаксис `list(set())`; как и прежде, это делается для того, чтобы список содержал только уникальные значения без дубликатов. Назовем результат `pivot`. Может оказаться, что `pivot` содержит более одного элемента. Это будет означать, что существует несколько комбинаций действительных слов, составляющих недействительное. В таком случае придется решить, какую комбинацию имел в виду автор. Сделать это с полной уверенностью невозможно. Для примера возьмем недействительное слово *choosespain*. Может, это слово взято из рекламной брошюры туристического агентства (Choose Spain!, то есть «Выбирайте Испанию!»), а может, из описания мазохиста (chooses pain, то есть «выбирает боль»). Из-за большого количества слова в нашем языке и огромного количества возможных комбинаций иногда бывает трудно с уверенностью сказать, какая из комбинаций правильна. Более эффективное решение будет учитывать контекст — то есть ассоциируются ли слова рядом с *choosespain* с оливками и корридой или же с плетками и ненужными визитами к стоматологу. Такое решение трудно реализовать хорошо и невозможно реализовать идеально, что снова демонстрирует сложность языковых алгоритмов вообще. В нашем случае берется наименьший элемент `pivot` — не потому, что это мы уверены в его правильности, а потому, что какой-то элемент нужно выбрать:

```
import numpy as np
pivot = np.min(pivot)
```

Наконец, мы напишем одну строку, которая заменяет недействительное слово двумя действительными составляющими словами и пробелом:

```
textnew = text
textnew = textnew.replace(text[loc[0]:loc[1]],text[loc[0]:pivot]+'
'+text[pivot:loc[1]])
```

При выводе нового текста можно убедиться в том, что программа правильно вставила пробел в ошибочное слово `oneperfectly`, хотя в остальных местах пробелы еще не были вставлены.

```
The one perfectly divine thing, the oneglimpse of God's paradisegiven on
earth, is to fight a losingbattle - and notlose it.
```

Все фрагменты объединяются в одну красивую функцию, приведенную в листинге 8.1. Она использует цикл `for` для вставки пробелов во все комбинации двух действительных слов, образующих недействительное слово из-за отсутствующего пробела.

Листинг 8.1. Функция для вставки пробелов в текст объединяет многие приведенные фрагменты кода

```
def insertspaces(text,word_list):
    locs = list(set([(m.start(),m.end()) for word in word_list for m \
           in re.finditer(word, text)]))
    spacestarts = [m.start() for m in re.finditer(' ', text)]
    spacestarts.append(-1)
    spacestarts.append(len(text))
    spacestarts.sort()
    spacestarts_affine = [ss + 1 for ss in spacestarts]
    spacestarts_affine.sort()
    partial_words = [loc for loc in locs if loc[0] in spacestarts_affine \
                     and loc[1] not in spacestarts]
    partial_words_end = [loc for loc in locs if loc[0] not in spacestarts_affine \
                         and loc[1] in spacestarts]
    between_spaces = [(spacestarts[k] + 1,spacestarts[k+1]) for k \
    in range(0,len(spacestarts) - 1)]
    between_spaces_notvalid = [loc for loc in between_spaces if \
                               text[loc[0]:loc[1]] not in word_list]
    textnew = text
    for loc in between_spaces_notvalid:
        endsofbeginnings = [loc2[1] for loc2 in partial_words if loc2[0] == \
                            loc[0] and (loc2[1] - loc[0]) > 1]
        beginningsofends = [loc2[0] for loc2 in partial_words_end if loc2[1] == \
                            loc[1] and (loc2[1] - loc[0]) > 1]
        pivot = list(set(endsofbeginnings).intersection(beginningsofends))
```

```
        if(len(pivot) > 0):
            pivot = np.min(pivot)
            textnew = textnew.replace(text[loc[0]:loc[1]],text[loc[0]:pivot]+' \
            '+text[pivot:loc[1]])
    textnew = textnew.replace('  ',' ')
    return(textnew)
```

Затем можно определить произвольный текст и вызвать функцию следующим образом:

```
text = "The oneperfectly divine thing, the oneglimpse of God's paradisegiven \
on earth, is to fight a losingbattle - and notlose it."
print(insertspaces(text,word_list))
```

Функция выдает ожидаемый результат с идеально расставленными пробелами:

```
The one perfectly divine thing, the one glimpse of God's paradise given on earth,
is to fight a losing battle - and not lose it.
```

Мы создали алгоритм, который может правильно вставлять пробелы в английский текст. Необходимо также понять, возможно ли сделать то же самое в других языках? Да, можно: если вы найдете хороший, подходящий корпус для языка, с которым работаете, для определения списка слов, то функция, которая определяется и вызывается в этом примере, сможет правильно вставить пробелы в текст на любом языке. Она даже может исправить текст на языке, который вы никогда не изучали и даже не слышали. Опробуйте разные корпусы, разные языки и разные тексты, чтобы понять, какие результаты можете получить, — и получите некоторое представление об эффективности языковых алгоритмов.

Завершение фраз

Представьте, что проводите консультации по алгоритмам для начинающей фирмы, которая старается расширить функциональность своей поисковой системы. Заказчик хочет добавить возможность завершения фраз, чтобы они могли предоставлять поисковые рекомендации пользователям. Например, если пользователь вводит *«торт»*, поисковая система предлагает вариант *«наполеон»*.

Реализовать такую возможность несложно. Мы начнем с корпуса, как и в программе вставки пробелов. В данном случае нас интересуют не только отдельные слова корпуса, но и их сочетания, поэтому на базе корпуса необходимо скомпилировать списки *n*-грамм. Термином *«n-грамма»* называется обычная коллекция из *n* слов, которые располагаются вместе. Например, фраза *Reality is not always probable,*

or likely состоит из семи слов, некогда произнесенных великим Хорхе Луисом Борхесом. 1-грамма представляет собой отдельное слово, так что 1-граммы этой фразы включают *reality*, *is*, *not*, *always*, *probable*, *or* и *likely*. В список 2-грамм включаются все строки из двух слов, следующих друг за другом, включая *reality is*, *is not*, *not always*, *always probable* и т. д. 3-граммы — *reality is not*, *is not always* и т. д.

Разбиение на лексемы и получение n-грамм

Мы воспользуемся модулем Python `nltk`, чтобы упростить построение коллекции *n*-грамм. Сначала текст разбивается на лексемы. Этим термином обозначается простое разделение строки на составляющие ее слова, включая знаки препинания. Пример:

```
from nltk.tokenize import sent_tokenize, word_tokenize
text = "Time forks perpetually toward innumerable futures"
print(word_tokenize(text))
```

Результат выглядит так:

```
['Time', 'forks', 'perpetually', 'toward', 'innumerable', 'futures']
```

Разбиение на лексемы и получение *n*-грамм для нашего текста выполняется следующим образом:

```
import nltk
from nltk.util import ngrams
token = nltk.word_tokenize(text)
bigrams = ngrams(token,2)
trigrams = ngrams(token,3)
fourgrams = ngrams(token,4)
fivegrams = ngrams(token,5)
```

Кроме того, можно поместить все *n*-граммы в список `grams`:

```
grams = [ngrams(token,2),ngrams(token,3),ngrams(token,4),ngrams(token,5)]
```

В этом случае мы получили разбиение и список *n*-грамм для короткого текста из одного предложения. Но чтобы иметь в своем распоряжении универсальный инструмент завершения фраз, нам понадобится существенно больший корпус. Корпус `brown`, который использовался для вставки пробелов, не подойдет, поскольку состоит из одиночных слов, поэтому для него невозможно построить *n*-граммы.

Одним из корпусов, которым можно было бы воспользоваться, является подборка литературных текстов, предложенная Питером Норвигом (Peter Norvig)

из компании Google (http://norvig.com/big.txt). Для примеров этой главы я загрузил файл с полным текстом работ Шекспира, находящийся в свободном доступе по адресу http://www.gutenberg.org/files/100/100-0.txt, после чего удалил заголовок проекта «Гутенберг» в начале файла. Вы также можете воспользоваться полным сборником работ Марка Твена по адресу http://www.gutenberg.org/cache/epub/3200/pg3200.txt. Прочитайте корпус в программе Python:

```
import requests
file = requests.get('http://www.bradfordtuckfield.com/shakespeare.txt')
file = file.text
text = file.replace('\n', '')
```

Здесь мы используем модуль `requests` для прямой загрузки текстового файла со сборником работ Шекспира с сайта, после чего читаем его в сеансе Python в переменную `text`.

После чтения выбранного корпуса снова выполните код, создавший переменную `grams`. Приведу фрагмент с новым определением переменной `text`:

```
token = nltk.word_tokenize(text)
bigrams = ngrams(token,2)
trigrams = ngrams(token,3)
fourgrams = ngrams(token,4)
fivegrams = ngrams(token,5)
grams = [ngrams(token,2),ngrams(token,3),ngrams(token,4),ngrams(token,5)]
```

Наша стратегия

Наша стратегия генерирования поисковых рекомендаций проста. Когда пользователь вводит условие поиска, мы проверяем количество слов во введенном условии. Иначе говоря, пользователь вводит n-грамму, и мы определяем, чему равно n. Когда пользователь ищет n-грамму, нужно помочь ему с вводом, то есть предложить ему $(n + 1)$-грамму. Мы проводим поиск по корпусу и находим все $(n + 1)$-граммы, первые n элементов которой совпадают с n-граммой. Например, пользователь может ввести слово *crane* (1-грамма), а корпус может содержать 2-граммы *crane feather*, *crane operator* и *crane neck*. Каждая 2-грамма является потенциальной рекомендацией, которую можно было бы предложить пользователю.

Здесь можно было бы остановиться и просто выдать все $(n + 1)$-граммы, первые n элементов которых соответствуют n-грамме, введенной пользователем. Однако не все рекомендации одинаково хороши. Например, если мы работаем со специализированным ядром, которое ищет рекомендации в руководствах по оборудованию для промышленного строительства, то вариант *crane operator* («оператор крана»)

будет релевантной, то есть актуальной, рекомендацией скорее, чем *crane feather* («журавлиное перо»). Чтобы определить, какая из (n + 1)-грамм будет лучшей рекомендацией, проще всего предложить ту, которая чаще встречается в нашем корпусе.

Таким образом, полный алгоритм выглядит так: пользователь ищет n-грамму; мы находим все (n + 1)-граммы, первые n элементов которых совпадают с n-граммой пользователя, и рекомендуем совпавшие (n + 1)-граммы, наиболее часто встречающиеся в корпусе.

Поиск подходящих (n + 1)-грамм

Чтобы найти (n + 1)-граммы, которые будут составлять поисковые рекомендации, необходимо знать длину поискового условия, введенного пользователем. Предположим, пользователь ввел строку *life is a*; таким образом, мы ищем рекомендации для завершения фразы *life is a ...* . Для получения длины поискового условия можно воспользоваться следующими простыми строками:

```
from nltk.tokenize import sent_tokenize, word_tokenize
search_term = 'life is a'
split_term = tuple(search_term.split(' '))
search_term_length = len(search_term.split(' '))
```

Теперь, когда мы знаем длину поискового условия, мы знаем n: значение равно 3. Напомню, что пользователю будут возвращаться самые частые (n + 1)-граммы (4-граммы). Следовательно, необходимо учесть частоты разных (n + 1)-грамм. Мы воспользуемся функцией `Counter()` для подсчета количества вхождений каждой (n + 1)-граммы в коллекции.

```
from collections import Counter
counted_grams = Counter(grams[search_term_length - 1])
```

Эта строка выбрала для переменной `grams` только (n + 1)-граммы. В результате применения функции `Counter()` создается список кортежей. В каждом кортеже первым элементом является (n + 1)-грамма, а вторым — частота вхождения этой (n + 1)-граммы в корпусе. Для примера выведем первый элемент `counted_grams`:

```
print(list(counted_grams.items())[0])
```

В выводе приводится первая (n + 1)-грамма в корпусе, а также сообщается, что во всем корпусе она встречается только один раз:

```
(('From', 'fairest', 'creatures', 'we'), 1)
```

Эта *n*-грамма является началом шекспировского сонета № 1. Интересно взглянуть на некоторые интересные 4-граммы, случайным образом найденные в работах Шекспира. Например, если выполнить команду `print(list(counted_grams)[10])`, то можно увидеть, что десятой 4-граммой в работах Шекспира является фраза rose might never die. Команда `run print(list(counted_grams)[240000])` показывает, что 240 000-й *n*-граммой является фраза *I shall command all*, 323 002-й — *far more glorious star*, а 328 004-й — *crack my arms asunder*. Но мы хотим реализовать функциональность завершения фраз, а не просто просмотр (*n* + 1)-грамм. Необходимо найти подмножество (*n* + 1)-грамм, первые *n* элементов которого совпадают с поисковым условием. Это можно сделать так:

```
matching_terms = [element for element in list(counted_grams.items()) if \
element[0][:-1] == tuple(split_term)]
```

Списковое включение перебирает все (*n* + 1)-граммы и для каждого элемента проверяет, выполняется ли условие `element[0][:-1]==tuple(split_term)`. Левая сторона этого условия, `element[0][:-1]`, просто берет первые *n* элементов каждой (*n* + 1)-граммы: `[:-1]` — удобный способ отбросить последний элемент списка. Правая сторона, `tuple(split_term)`, определяет искомую *n*-грамму (*life is a*). Таким образом, мы отбираем (*n* + 1)-граммы, первые *n* элементов которых совпадают с интересующей нас *n*-граммой. Найденные результаты сохраняются в выходном списке `matching_terms`.

Выбор фразы на основании частоты

Список `matching_terms` содержит все необходимое для завершения работы; он состоит из (*n* + 1)-грамм, первые *n* элементов которых совпадают с поисковым условием, а также включаются частоты их вхождения в корпус. Если список содержит хотя бы один элемент, то мы можем найти элемент, который встречается чаще всего в корпусе, и предложить его пользователю для завершения фразы. Следующий фрагмент решает эту задачу:

```
if(len(matching_terms)>0):
    frequencies = [item[1] for item in matching_terms]
    maximum_frequency = np.max(frequencies)
    highest_frequency_term = [item[0] for item in matching_terms if item[1] == \
    maximum_frequency][0]
    combined_term = ' '.join(highest_frequency_term)
```

В этом фрагменте сначала определяется `frequencies` — список, содержащий частоты всех (*n* + 1)-грамм в корпусе, соответствующих поисковому условию. Затем

с помощью функции `max()` модуля `numpy` находится наибольшая из этих частот. Другое списковое включение используется для получения первой (n + 1)-граммы, встречающейся в корпусе с наибольшей частотой. Наконец, мы создаем `combined_term` — строку, которая собирает все слова поискового условия, разделяя их пробелами.

И, наконец, объединим весь код в функции, приведенной в листинге 8.2.

Листинг 8.2. Функция, предоставляющая поисковые рекомендации; для этого она берет n-грамму и возвращает наиболее вероятную (n + 1)-грамму, начинающуюся с заданной n-граммы

```
def search_suggestion(search_term, text):
    token = nltk.word_tokenize(text)
    bigrams = ngrams(token,2)
    trigrams = ngrams(token,3)
    fourgrams = ngrams(token,4)
    fivegrams = ngrams(token,5)
    grams = [ngrams(token,2),ngrams(token,3),ngrams(token,4),ngrams(token,5)]
    split_term = tuple(search_term.split(' '))
    search_term_length = len(search_term.split(' '))
    counted_grams = Counter(grams[search_term_length-1])
    combined_term = 'No suggested searches'
    matching_terms = [element for element in list(counted_grams.items()) if \
    element[0][:-1] == tuple(split_term)]
    if(len(matching_terms) > 0):
        frequencies = [item[1] for item in matching_terms]
        maximum_frequency = np.max(frequencies)
        highest_frequency_term = [item[0] for item in matching_terms if item[1] == \
        maximum_frequency][0]
        combined_term = ' '.join(highest_frequency_term)
    return(combined_term)
```

При вызове функции n-грамма передается в аргументе, функция возвращает (n + 1)-грамму. Пример ее вызова выглядит следующим образом:

```
file = requests.get('http://www.bradfordtuckfield.com/shakespeare.txt')
file = file=file.text
text = file.replace('\n', '')
print(search_suggestion('life is a', text))
```

Функция выдает рекомендацию *life is a tedious* — самую частую 4-грамму, начинающуюся со слов *life is a* в трудах Шекспира. По частоте она совпадает с двумя другими 4-граммами. Шекспир использует эту 4-грамму всего один раз в «Цимбелине», когда Имогена говорит: *I see a man's life is a tedious one*. В «Короле Лире» Эдгар говорит Глочестеру: *Thy life is a miracle*, так что эта 4-грамма тоже могла бы стать допустимым завершением фразы.

Для интереса попробуем взять другой корпус и сравнить результаты. Воспользуемся корпусом собрания работ Марка Твена:

```
file = requests.get('http://www.bradfordtuckfield.com/marktwain.txt')
file = file=file.text
text = file.replace('\n', '')
```

Повторим проверку рекомендаций с новым корпусом:

```
print(search_suggestion('life is a',text))
```

На этот раз завершенная фраза имеет вид *life is a failure*, что указывает на различия между двумя корпусами — и, вероятно, на различия между стилем и мировоззрением Шекспира и Марка Твена. Можно опробовать и другие поисковые условия. Например, фраза *I love* завершается словом *you*, если взять корпус Марка Твена, и словом *thee* при использовании корпуса Шекспира. Это указывает на разницу в стилях авторов, разделенных веками и океаном, если не разницу в идеях. Попробуйте использовать другой корпус с другими фразами и посмотрите, как будут завершаться эти фразы. Если взять корпус, написанный на другом языке, то написанная нами функция может использоваться для завершения фраз даже на тех языках, на которых вы не говорите.

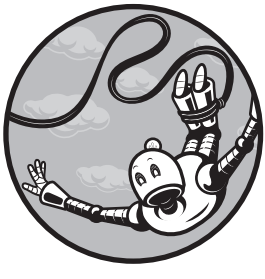
Резюме

В этой главе рассматривались алгоритмы, которые могут использоваться для работы с естественным языком. Мы начали с алгоритма вставки пробелов, который способен исправлять неправильно отсканированные тексты, а затем перешли к алгоритму завершения фраз, который может добавлять слова во входные фразы в соответствии с содержанием и стилем текстового корпуса. Подход, который применялся в этих алгоритмах, сходен с подходами, работающими для других видов языковых алгоритмов, включая системы проверки орфографии и анализа намерений.

В следующей главе рассматривается машинное обучение — перспективная развивающаяся область, с которой должен быть знаком каждый специалист по алгоритмам. Основное внимание будет уделено такому алгоритму машинного обучения, как *деревья принятия решений*, — простым, гибким, точным и интерпретируемым моделям, которые могут далеко завести вас в вашем путешествии по алгоритмам и жизни.

9

Машинное обучение



Итак, вы понимаете идеи, лежащие в основе многих фундаментальных алгоритмов, и мы можем обратиться к более сложным темам. Текущая глава посвящена *машинному обучению*. Этим термином обозначается широкий диапазон методов, но все они преследуют одну и ту же цель: выявление закономерностей в данных и использование их для прогнозирования. Мы обсудим метод, называемый *деревьями принятия решений*, а затем построим такое дерево для прогнозирования уровня человеческого счастья на основании некоторых персональных характеристик.

Деревья принятия решений

Деревья принятия решений представляют собой диаграммы с иерархической структурой, напоминающей дерево. Они используются примерно так же, как и блок-схемы — ответы на вопросы «да/нет» управляют нашим перемещением по пути, ведущему к итоговому решению, прогнозу или рекомендации. Процесс создания дерева решений, ведущего к оптимальным решениям, относится к числу хрестоматийных примеров алгоритмов машинного обучения.

Рассмотрим реальный сценарий, в котором могут использоваться деревья принятия решений. В отделениях неотложной помощи лицо, принимающее решение, должно

выполнить медицинскую *сортировку*, то есть установить очередность оказания помощи каждому поступающему пациенту. Попросту говоря, речь идет о назначении приоритета: тот, кто находится в двух шагах от смерти, но может быть спасен с помощью своевременной операции, направляется на лечение немедленно, тогда как пациенту с неглубоким порезом или насморком могут предложить подождать, пока врачи разберутся с более неотложными делами.

Такая медицинская сортировка сложна тем, что вы должны поставить достаточно точный диагноз, имея минимум информации или времени. Если 50-летняя женщина приходит в пункт оказания неотложной помощи и жалуется на сильную боль в груди, специалист должен решить, что стало причиной — инфаркт или изжога. Мыслительный процесс человека, принимающего решения при медицинской сортировке, неизбежно сложен. Ему приходится принимать во внимание целый ряд факторов: возраст и пол пациента, избыточный вес или курение, симптомы, о которых сообщает пациент, и то, как он о них говорит, выражение лица, загруженность больницы, наличие других пациентов, ожидающих лечения, а также факторы, которых врач даже может не сознавать. Чтобы эффективно выполнять медицинскую сортировку, он должен изучить многих пациентов.

Понять, как профессионал принимает решение, непросто. На рис. 9.1 изображен гипотетический, полностью вымышленный процесс принятия решений (он не предназначен для медицинских целей — не пытайтесь повторить дома!)

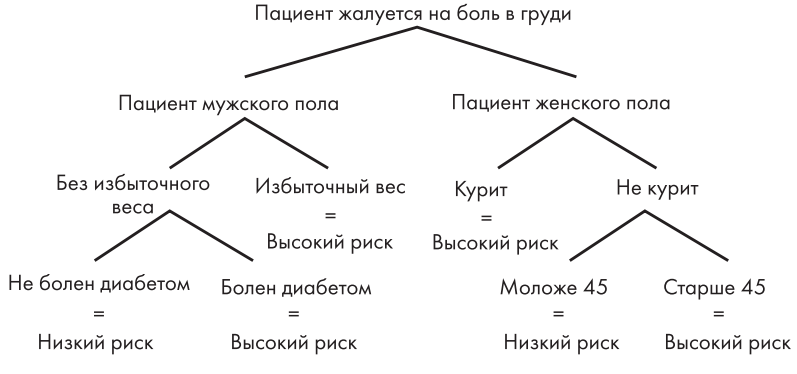


Рис. 9.1. Упрощенное дерево принятия решений для оценки рисков при инфаркте

Диаграмма читается сверху вниз. Наверху мы видим, что процесс диагностики инфаркта начинается с жалобы пациента на боль в груди. После этого процесс расходится в зависимости от пола пациента. Если пациент мужчина, то процесс диагностики переходит на левую ветвь, и мы проверяем наличие лишнего веса. Если

пациент женщина, то процесс переходит на правую ветвь, и мы проверяем, курит ли она. В каждой точке процесса выбирается соответствующая ветвь, пока не будет достигнут низ дерева, где находится классификация риска инфаркта у пациента (высокий/низкий). Процесс бинарного ветвления напоминает дерево, у которого ствол делится на меньшие побеги, пока не будут достигнуты концы самых дальних ветвей. Соответственно, процесс, показанный на рис. 9.1, называется *деревом принятия решений*.

Каждая позиция с текстом на рис. 9.1 является *узлом* в дереве принятия решений. Такой узел, как «без избыточного веса», называется *узлом ветвления*, поскольку за ним следует как минимум одна ветвь, прежде чем мы сможем принять решение. Узел «не болен диабетом = низкий риск» называется *терминальным узлом*, так как при прибытии к нему ветвления больше не нужны, и мы знаем итоговую классификацию дерева решений («низкий риск»).

Если бы мы могли создать полное, хорошо исследованное дерево принятия решений, которое всегда приводит к хорошим решениям, даже человек без медицинского образования смог бы выполнять медицинскую сортировку пациентов с инфарктом. Это сэкономило бы всем больницам в мире огромное количество денег, поскольку им уже не пришлось бы нанимать и обучать рациональных, образованных профессионалов. Достаточно хорошее дерево принятия решений даже позволило бы заменить профессионалов-людей роботами, хотя вопрос о том, насколько это достойная цель, остается спорным. Хорошее дерево принятия решений даже может привести к лучшим решениям, чем средний человек-специалист, поскольку теоретически это позволит устранить подсознательные искажения, к которым склонны мы, неидеальные люди. (Собственно, это уже произошло: в 1996 и 2002 годах независимые группы исследователей опубликовали статьи об успешном улучшении результатов диагностики для пациентов, жалующихся на боль в груди, с применением деревьев принятия решений.)

Действия с ветвлением, описанные в дереве принятия решений, образуют алгоритм. Выполнить его очень просто: решите, по какой из двух ветвей вы пойдете в каждом узле, и следуйте по ветвям до конца. Но не стоит слепо подчиняться рекомендациям каждого дерева принятия решений, которое вам встретится. Помните, что любой может создать дерево принятия решений, предписывающее любой мыслимый процесс принятия решений, даже если он приводит к неправильным решениям. Самая трудная часть деревьев принятия решений — не выполнение алгоритма дерева, а построение дерева, приводящего к лучшим возможным решениям. Создание оптимального дерева принятия решений является задачей машинного обучения, тогда как простое следование по дереву принятия решений этого не требует. Рассмотрим алгоритм, который создает оптимальное дерево

принятия решений — алгоритм для генерирования алгоритмов, — и перейдем к основным шагам построения дерева принятия решений.

Построение дерева принятия решений

Построим дерево принятия решений, использующее информацию о человеке для прогнозирования уровня его счастья. Поиски секрета счастья веками занимали умы миллионов людей, и социологи в наши дни тратят ведра чернил (и осваивают множество исследовательских грантов) в поисках ответов. Если у вас есть дерево принятия решений, которое по нескольким фрагментам информации может надежно предсказать, насколько счастлив человек, то это даст важные сведения о том, что определяет ощущение счастья, а то и, возможно, пару-тройку идей о том, как его достичь. К концу текущей главы вы будете знать, как строятся такие деревья.

Загрузка набора данных

Алгоритмы машинного обучения ищут полезные закономерности в данных, поэтому для них необходим хороший набор данных. Мы воспользуемся данными Европейского социального обследования (European Social Survey, ESS) для построения дерева. (Мы получили файлы на https://www.kaggle.com/pascalbliem/european-social-survey-ess-8-ed21-201617, где они были доступны для бесплатного распространения.) ESS — крупномасштабное социологическое исследование взрослых жителей Европы, проводимое каждые два года. Респонденту задается множество личных вопросов, в том числе о религиозных взглядах, состоянии здоровья, общественной жизни и уровне счастья. Рассматриваемые файлы хранятся в формате `csv`. Название формата (и расширение файла `.csv`) является сокращением от Comma-Separated Values («значения, разделенные запятыми»); это очень распространенный и простой способ хранения наборов данных, которые можно открывать из Microsoft Excel, LibreOffice Calc, текстовых редакторов и некоторых модулей Python.

Файл `variables.csv` содержит подробное описание каждого вопроса, включенного в опрос. Например, в строке 103 файла `variables.csv` содержится описание переменной `happy`. В ней хранится ответ респондента на вопрос: «В общем и целом насколько вы чувствуете себя счастливым?» Ответы на него лежат в диапазоне от 1 (полностью несчастен) до 10 (в высшей мере счастлив). Просматривая другие переменные в `variables.csv`, можно составить представление о доступной информации. Например, переменная `sclmeet` содержит информацию о том, насколько часто респондент встречается с друзьями, родственниками и коллегами. В переменной `health` хранится информация о субъективной оценке здоровья. Переменная `rlgdgr` содержит субъективную оценку религиозности респондента и т. д.

После знакомства с данными можно начать думать о гипотезах, относящихся к прогнозам счастья. Можно обоснованно предположить, что люди с активной социальной жизнью и хорошим здоровьем счастливее других. По поводу других переменных — пол, уровень благосостояния, возраст — что-то предположить уже сложнее.

Изучение данных

Начнем с чтения данных. Скачайте их по ссылке и сохраните локально в файле `ess.csv`. После этого можно воспользоваться модулем `pandas` для работы с ними, сохранив его в переменной `ess` сеанса Python:

```
import pandas as pd
ess = pd.read_csv('ess.csv')
```

Помните: чтобы вы могли прочитать файл CSV, он должен храниться в каталоге, из которого запускается Python. Если он хранится в другом каталоге, то вам придется изменить строку `'ess.csv'` из предыдущего фрагмента, чтобы она отражала правильный путь хранения файла CSV. Чтобы узнать количество строк и столбцов в данных, можно воспользоваться атрибутом `shape` кадра данных `pandas`:

```
print(ess.shape)
```

Вывод `(44387, 534)` показывает, что набор данных состоит из 44 387 строк (по одной для каждого респондента) и 534 столбцов (по одному для каждого вопроса в исследовании). Чтобы более внимательно рассмотреть некоторые столбцы, представляющие интерес, можно воспользоваться средствами сегментации модуля `pandas`. Например, следующий фрагмент просматривает первые пять ответов на вопрос о счастье (`happy`):

```
print(ess.loc[:,'happy'].head())
```

Набор данных `ess` состоит из 534 столбцов, по одному для каждого вопроса в исследовании. Возможно, для каких-то целей нам понадобится работать со всеми 534 столбцами сразу, но здесь мы хотим рассматривать только столбец `happy` без остальных 533. Именно для этой цели использовалась функция `loc()`. Здесь она извлекает из кадра данных `pandas` переменную `happy`. Другими словами, выделяет только этот столбец и игнорирует остальные 533. Затем функция `head()` выводит первые пять строк этот столбца. Мы видим, что первые пять ответов были равны `5`, `5`, `8`, `8` и `5`. То же самое можно сделать с переменной `sclmeet`:

```
print(ess.loc[:,'sclmeet'].head())
```

Результат должен быть равен 6, 4, 4, 4 и 6. Ответы happy и ответы sclmeet сохраняют порядок опроса. Например, 134-й элемент sclmeet был дан тем же человеком, который дал ответ в 134-м элементе happy.

Персонал ESS старается получить от каждого респондента полный набор ответов. Тем не менее в отдельных случаях ответы на некоторые вопросы отсутствуют — например, потому что респондент отказывается отвечать или не знает, что ответить. Отсутствующим ответам в наборе данных ESS присваиваются коды, намного превышающие допустимый диапазон реальных ответов. Например, если респондент отказывается ответить на вопрос, который предлагает респонденту выбрать число от 1 до 10, то ESS регистрирует ответ 77. Для нашего анализа будут рассматриваться только полные ответы, без отсутствующих значений для интересующих нас переменных. Если вы хотите ограничить набор ess, чтобы он содержал только полные ответы для нужных переменных, то это можно сделать так:

```
ess = ess.loc[ess['sclmeet'] <= 10,:].copy()
ess = ess.loc[ess['rlgdgr'] <= 10,:].copy()
ess = ess.loc[ess['hhmmb'] <= 50,:].copy()
ess = ess.loc[ess['netusoft'] <= 5,:].copy()
ess = ess.loc[ess['agea'] <= 200,:].copy()
ess = ess.loc[ess['health'] <= 5,:].copy()
ess = ess.loc[ess['happy'] <= 10,:].copy()
ess = ess.loc[ess['eduyrs'] <= 100,:].copy().reset_index(drop=True)
```

Разбиение данных

Существует много способов использования этих данных для исследования отношений между социальной жизнью человека и его уровнем счастья. Один из простейших подходов — бинарное разбиение: мы сравниваем уровни счастья людей с активной и менее активной социальной жизнью (листинг 9.1).

Листинг 9.1. Вычисление средних уровней счастья людей с активной и неактивной социальной жизнью

```
import numpy as np
social = list(ess.loc[:,'sclmeet'])
happy = list(ess.loc[:,'happy'])
low_social_happiness = [hap for soc,hap in zip(social,happy) if soc <= 5]
high_social_happiness = [hap for soc,hap in zip(social,happy) if soc > 5]

meanlower = np.mean(low_social_happiness)
meanhigher = np.mean(high_social_happiness)
```

В листинге 9.1 модуль numpy импортируется для вычисления среднего значения. Мы определяем две новые переменные social и happy, отделяя их от кадра данных ess.

Затем используем списковые включения, чтобы найти уровни счастья всех людей с более низкими оценками социальной активности (сохраняются в переменной `low_social_happiness`) и более высокими оценками социальной активности (сохраненными в переменной `high_social_happiness`). Наконец, мы вычисляем средние оценки счастья асоциальных людей и социально активных людей (`meanlower` и `meanhigher` соответственно). Если вы выполните команды `print(meanlower)` и `print(meanhigher)`, то увидите, что люди, оценившие себя как социально активных, также оценили себя как чуть более счастливых по сравнению с менее социально активными лицами: для социально активных средний уровень счастья составлял около `7.8`, а для социально неактивных — около `7.2`.

Нарисуем простую диаграмму того, что только что было сделано (рис. 9.2).

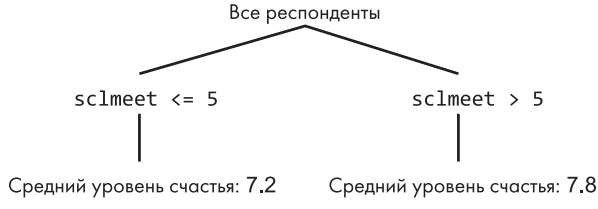


Рис. 9.2. Простое дерево принятия решений, прогнозирующее уровень счастья по частоте социальных взаимодействий

Диаграмма простого бинарного разбиения уже начала напоминать дерево принятия решений. И это не случайность: создание бинарного разбиения в наборе данных и сравнение результатов по каждой половине — процесс, занимающий центральное место в алгоритме генерирования дерева принятия решений. Собственно, рис. 9.2 можно назвать деревом принятия решений, хотя и содержащим всего один узел ветвления. Мы можем использовать рис. 9.2 как очень простую систему прогнозирования счастья: определяем, насколько часто респондент общается с другими людьми. Если значение `sclmeet` равно `5` и менее, то мы прогнозируем, что уровень счастья составляет `7.2`. Если оно выше `5`, то прогнозируем, что уровень счастья составляет `7.8`. Прогноз будет не идеальным, но это лишь начало, и оно лучше случайных догадок.

Можно попытаться использовать дерево принятия решений для выводов о влиянии различных характеристик и решений, связанных с образом жизни. Например, мы видим, что различия между низким и высоким социальным счастьем составляют около 0,6, и делаем вывод, что повышение уровня социальной активности с низкого до высокого может привести к прогнозируемому повышению уровня счастья около 0,6 по десятибалльной шкале. Конечно, попытки формирования подобных

заключений сопряжены с определенным риском. Может оказаться, что социальная активность не повышает уровень счастья, а наоборот — счастье порождает социальную активность; возможно, счастливые люди чаще находятся в приподнятом настроении, из-за чего звонят своим друзьям и устраивает встречи. Отделение корреляции от причинно-следственной связи выходит за уровень подачи материала в этой главе, но независимо от направления причинно-следственных связей наше простое дерево принятия решений, по крайней мере, указало нам на факт связи, который при желании можно исследовать и дальше. Как объяснил карикатурист Рэндал Мунро (Randall Munroe), «корреляция необязательно подразумевает причинно-следственную связь, но выразительно нам подмигивает и яростно жестикулирует, призывая нас: “Посмотри сюда!”».

Вы уже знаете, как построить простое дерево принятия решений с двумя ветвями. Теперь необходимо отточить процесс создания ветвей, а затем создать побольше ветвей для улучшенного, более полного дерева принятия решений.

Умное разбиение

Сравнивая уровни счастья людей с активной и неактивной социальной жизнью, мы использовали `5` как *точку разбиения*: было решено, что респонденты с оценкой выше `5` имеют активную социальную жизнь, а респонденты с оценкой `5` и ниже — неактивную. Мы выбрали `5`, поскольку это естественная средняя точка для оценок в диапазоне от 1 до 10. Однако следует помнить, что наша цель — построение системы для надежного прогнозирования счастья. Вместо того чтобы выполнять разбиение по нашим интуитивным представлениям о том, что именно естественно или что считать активной социальной жизнью, возможно, лучше выполнить бинарное разбиение в каком-то месте, которое приводит к наивысшей возможной точности.

В задачах машинного обучения возможно несколько разных способов измерения точности. Наиболее естественный — вычисление суммы ошибок. В нашем случае ошибка, интересующая нас, равна разности между прогнозируемой и фактической оценкой уровня счастья. Если дерево принятия решений прогнозирует, что уровень счастья равен `6`, а на самом деле он равен `8`, то ошибка дерева по отношению к оценке равна `2`. Суммировав ошибки прогнозирования для каждого респондента в некой группе, можно получить суммарную ошибку, которая оценивает точность дерева принятия решений по прогнозированию счастья для участников группы. Чем ближе суммарная ошибка к нулю, тем лучше дерево (но обратите внимание на важное предупреждение в подразделе «Проблема переобучения» на с. 231). Следующий фрагмент демонстрирует простой способ вычисления суммарной ошибки.

```
lowererrors = [abs(lowhappy - meanlower) for lowhappy in low_social_happiness]
highererrors = [abs(highhappy - meanhigher) for highhappy in high_social_happiness]

total_error = sum(lowererrors) + sum(highererrors)
```

Этот код вычисляет сумму всех ошибок прогнозирования для всех респондентов. Он определяет `lowererrors` — список, содержащий ошибки прогнозирования для всех менее социально активных респондентов, и `highererrors` — список, содержащий ошибки прогнозирования для социально активных респондентов. Обратите внимание на получение абсолютного значения (модуля) — это делается для того, чтобы для вычисления суммы ошибки складывались только неотрицательные числа. При выполнении этого кода оказывается, что суммарная ошибка составляет около `60224`. Данное число намного выше нуля, но если учесть, что это сумма ошибок для более чем 40 000 респондентов, уровень счастья которых прогнозировался по дереву всего с двумя ветвями, то внезапно результат начинает казаться уже не таким плохим.

Можно опробовать разные точки разбиения и посмотреть, уменьшит ли это ошибку. Например, можно классифицировать всех респондентов с оценкой выше `4` как социально активных, а всех респондентов с оценкой `4` и ниже — как социально неактивных, и сравнить полученные ошибки. А можно использовать в качестве точки разбиения значение `6`. Чтобы добиться самой высокой точности из возможных, нужно последовательно проверить все возможные точки разбиения и выбрать вариант, ведущий к самой низкой ошибке из всех возможных. Функция, которая решает эту задачу, приведена в листинге 9.2.

Листинг 9.2. Функция, находящая лучшую точку разбиения переменной для ветвления в дереве принятия решений

```
def get_splitpoint(allvalues,predictedvalues):
    lowest_error = float('inf')
    best_split = None
    best_lowermean = np.mean(predictedvalues)
    best_highermean = np.mean(predictedvalues)
    for pctl in range(0,100):
        split_candidate = np.percentile(allvalues, pctl)

        loweroutcomes = [outcome for value,outcome \
                      in zip(allvalues,predictedvalues) if value <= split_candidate]
        higheroutcomes = [outcome for value,outcome \
                       in zip(allvalues,predictedvalues) if value > split_candidate]

        if np.min([len(loweroutcomes),len(higheroutcomes)]) > 0:
            meanlower = np.mean(loweroutcomes)
            meanhigher = np.mean(higheroutcomes)
```

```
            lowererrors = [abs(outcome - meanlower) for outcome in loweroutcomes]
            highererrors = [abs(outcome - meanhigher) for outcome
                            in higheroutcomes]

            total_error = sum(lowererrors) + sum(highererrors)

            if total_error < lowest_error:
                best_split = split_candidate
                lowest_error = total_error
                best_lowermean = meanlower
                best_highermean = meanhigher
    return(best_split,lowest_error,best_lowermean,best_highermean)
```

В этой функции переменная `pctl` (сокращение от percentile) используется для перебора всех чисел от 0 до 100. В первой строке цикла определяется новая переменная `split_candidate`, которая определяет `pctl`-й процентиль данных. Далее повторяется тот же процесс, который использовался в листинге 9.2. Мы создаем список уровней счастья людей, у которых значения `sclmeet` меньше либо равны потенциальной точке разбиения, и список уровней счастья людей, у которых значения `sclmeet` больше потенциальной точки разбиения, после чего проверяем величину ошибки при использовании этой потенциальной точки разбиения. Если суммарная ошибка для этой точки разбиения меньше всех суммарных ошибок при использовании всех предыдущих кандидатов, то мы переопределяем переменную `best_split` равной `split_candidate`. После завершения цикла переменная `best_split` содержит точку разбиения, которая обеспечила наивысшую точность.

Эту функцию можно выполнить для любой переменной, как делается в следующем примере для `hhmmb` — переменной, в которой хранится количество членов семьи респондента.

```
allvalues = list(ess.loc[:,'hhmmb'])
predictedvalues = list(ess.loc[:,'happy'])
print(get_splitpoint(allvalues,predictedvalues))
```

Приведенный вывод демонстрирует как правильную точку разбиения, так и прогнозируемые уровни счастья для групп, определяемых этой точкой разбиения:

```
(1.0, 60860.029867951016, 6.839403436723225, 7.620055170794695)
```

Этот вывод можно интерпретировать так: для разбиения переменной `hhmmb` лучше всего использовать уровень `1.0`; респонденты делятся на людей, живущих в одиночку (один член семьи), и живущих с другими (несколько членов семьи). Также мы видим средние уровни счастья для этих двух групп: около `6.84` и `7.62` соответственно.

Выбор переменных разбиения

Для любой переменной, выбранной в наших данных, можно найти оптимальную точку разбиения. Однако следует помнить, что в таком дереве принятия решений, как на рис. 9.1, мы не ограничиваемся поиском точек разбиения для только одной переменной. Мужчины отделяются от женщин, курящие от некурящих, имеющие лишний вес от стройных и т. д. Естественный вопрос: как узнать, какую переменную использовать для разбиения на каждом узле ветвления? Узлы на рис. 9.1 можно переупорядочить так, чтобы сначала осуществлялось разбиение по весу, а затем по полу, или только по полу на левой ветви, или вообще никак. Выбор переменной для разбиения на каждом узле ветвления является важнейшей точкой генерирования оптимального дерева принятия решений, чтобы мы могли написать код для этой части процесса.

Лучшая переменная для разбиения будет определяться с помощью того же принципа, который использовался для получения оптимальных точек разбиения: лучшим способом разбиения считается тот, который приводит к наименьшей ошибке. Чтобы определить его, необходимо перебрать все доступные переменные и проверить, обеспечивает ли разбиение по этой переменной минимальную величину ошибки. Затем нужно определить, какая переменная приводит к разбиению с наименьшей ошибкой. Для решения этой задачи используется код в листинге 9.3.

Листинг 9.3. Функция перебирает все переменные и находит лучшую переменную для разбиения

```
def getsplit(data,variables,outcome_variable):
    best_var = ''
    lowest_error = float('inf')
    best_split = None
    predictedvalues = list(data.loc[:,outcome_variable])
    best_lowermean = -1
    best_highermean = -1
    for var in variables:
        allvalues = list(data.loc[:,var])
        splitted = get_splitpoint(allvalues,predictedvalues)

        if(splitted[1] < lowest_error):
            best_split = splitted[0]
            lowest_error = splitted[1]
            best_var = var
            best_lowermean = splitted[2]
            best_highermean = splitted[3]

    generated_tree = [[best_var,float('-inf'),best_split,best_lowermean], \
                      [best_var,best_split, float('inf'),best_highermean]]

    return(generated_tree)
```

В листинге 9.3 определяется функция с циклом `for`, перебирающим все переменные в списке переменных. Для каждой из них функция `get_splitpoint()` используется для нахождения лучшей точки разбиения. Каждая переменная при разбиении по оптимальной точке приводит к некоторой суммарной ошибке в наших прогнозах. Если некая переменная имеет более низкую сумму ошибки, чем все переменные, рассматриваемые ранее, то ее имя сохраняется в переменной `best_var`. После перебора всех имен переменных будет найдена переменная с наименьшей суммой ошибки, хранящаяся в `best_var`. Этот код можно выполнить и для других переменных, кроме `sclmeet`:

```
variables = ['rlgdgr','hhmmb','netusoft','agea','eduyrs']
outcome_variable = 'happy'
print(getsplit(ess,variables,outcome_variable))
```

В этом случае результат выглядит так:

```
[['netusoft', -inf, 4.0, 7.041597337770383], ['netusoft', 4.0, inf,
7.73042471042471]]
```

Функция `getsplit()` выводит очень простое «дерево» в формате вложенного списка. Оно состоит всего из двух ветвей. Первая представлена первым вложенным списком, а вторая — вторым. Каждый элемент обоих вложенных списков сообщает что-то о соответствующих ветвях. Первый список сообщает, что перед нами ветвь, основанная на значении `netusoft` (частоте использования интернета респондентом). А если говорить конкретнее, то первая ветвь соответствует людям, у которых значение `netusoft` лежит в диапазоне от `-inf` до `4.0`, где `inf` соответствует бесконечности. Другими словами, люди в этой ветви оценивают свою частоту использования интернета значением 4 и менее по 5-балльной шкале. Последний элемент каждого списка содержит оценку уровня счастья: около `7.0` для людей, не являющихся особо активными пользователями интернета. Это простое дерево изображено на рис. 9.3.

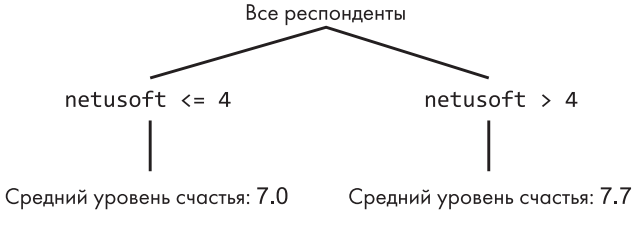


Рис. 9.3. Дерево, генерируемое первым вызовом функции getsplit()

Пока что функция сообщает нам, что люди, относительно редко пользующиеся интернетом, считают себя менее счастливыми (средняя оценка уровня счастья около `7.0`), тогда как люди, часто пользующиеся интернетом, сообщают о среднем уровне счастья `7.7`. И снова не стоит торопиться с выводами на основе этого простого факта: использование интернета может не являться настоящим определяющим фактором счастья, но может быть связано с уровнями счастья из-за сильных корреляций с возрастом, богатством, здоровьем, образованием и другими характеристиками. Машинное обучение само по себе обычно не позволяет уверенно определять сложные причинно-следственные связи, но, как и в случае с простым деревом на рис. 9.3, позволяет делать точные прогнозы.

Добавление глубины

Мы сделали все необходимое, чтобы выполнить наилучшее возможное разбиение в каждой точке ветвления и сгенерировать дерево с двумя ветвями. Далее нужно увеличить дерево за пределы одного узла ветвления и двух терминальных узлов. Взгляните на рис. 9.1: это дерево имеет более двух ветвей. Его *глубина* равна 3, поскольку для определения окончательного диагноза необходимо пройти три последовательные ветви. На последнем шаге генерирования дерева принятия решений указывается глубина, которую вы хотите достичь, и новые ветви строятся, пока она не будет достигнута. Для этого мы внесем некоторые изменения в функцию `getsplit()`, показанную в листинге 9.4.

Листинг 9.4. Функция, генерирующая дерево заданной глубины

```
maxdepth = 3
def getsplit(depth,data,variables,outcome_variable):
    --...--
    generated_tree = [[best_var,float('-inf'),best_split,[]],[best_var,\
best_split,float('inf'),[]]]

    if depth < maxdepth:
        splitdata1=data.loc[data[best_var] <= best_split,:]
        splitdata2=data.loc[data[best_var] > best_split,:]
        if len(splitdata1.index) > 10 and len(splitdata2.index) > 10:
            generated_tree[0][3] =
                      getsplit(depth + 1,splitdata1,variables,outcome_variable)
            generated_tree[1][3] =
                      getsplit(depth + 1,splitdata2,variables,outcome_variable)
        else:
            depth = maxdepth + 1
            generated_tree[0][3] = best_lowermean
            generated_tree[1][3] = best_highermean
  else:
```

```
    generated_tree[0][3] = best_lowermean
    generated_tree[1][3] = best_highermean
  return(generated_tree)
```

В этой обновленной функции при определении переменной `generated_tree` в нее добавляются пустые списки вместо средних значений. Средние значения вставляются только в терминальных узлах, но если вам нужно дерево с большей глубиной, то необходимо вставить в каждую ветвь другие ветви (они будут содержаться в пустых списках). Кроме того, в конце функции добавляется команда `if` с длинным блоком кода. Если глубина текущей ветви меньше максимальной глубины, необходимой для дерева, то данная часть будет рекурсивно вызывать функцию `get_split()` для заполнения другой ветви внутри нее. Процесс продолжается до достижения максимальной глубины.

Выполните следующий код, чтобы найти дерево принятия решений, обеспечивающее минимальную ошибку в прогнозах уровня счастья для нашего набора данных:

```
variables = ['rlgdgr','hhmmb','netusoft','agea','eduyrs']
outcome_variable = 'happy'
maxdepth = 2
print(getsplit(0,ess,variables,outcome_variable))
```

В результате мы получаем следующий вывод, представляющий дерево с глубиной 2 (листинг 9.5).

Листинг 9.5. Представление дерева принятия решений с вложенными списками

```
[['netusoft', -inf, 4.0, [['hhmmb', -inf, 4.0, [['agea', -inf, 15.0,
8.035714285714286], ['agea', 15.0, inf, 6.997666564322997]]], ['hhmmb',
4.0, inf, [['eduyrs', -inf, 11.0, 7.263969171483622], ['eduyrs', 11.0, inf,
8.0]]]]], ['netusoft', 4.0, inf, [['hhmmb', -inf, 1.0, [['agea', -inf, 66.0,
7.135361428970136], ['agea', 66.0, inf, 7.621993127147766]]], ['hhmmb',
1.0, inf, [['rlgdgr', -inf, 5.0, 7.743893678160919], ['rlgdgr', 5.0, inf,
7.9873320537428025]]]]]]
```

Как видите, это коллекция списков, вложенных друг в друга. Вложенные списки представляют полное дерево принятия решений, хотя эта форма читается не так просто, как рис. 9.1. На каждом уровне вложенности находится имя переменной и ее диапазон, как и с простым деревом на рис. 9.3. На первом уровне вложенности видна ветвь, показанная на рис. 9.3: она представляет респондентов со значением `netusoft`, меньшим либо равным `4.0`. Следующий список, вложенный в первый, начинается с `hhmmb, -inf, 4.0`. Это еще одна ветвь дерева принятия решений, которая отходит от только что исследованной ветви и состоит из людей, указавших размер семьи `4` и менее. Если нарисовать часть дерева принятия решений, которая была

рассмотрена во вложенном списке к настоящему моменту, то это будет выглядеть так, как показано на рис. 9.4.

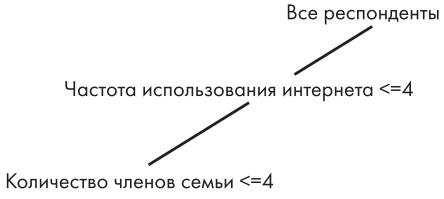


Рис. 9.4. Выбор ветвей в дереве принятия решений

Можно продолжить рассматривать вложенные списки, чтобы заполнить новые ветви дерева принятия решений. Списки, вложенные в другие списки, соответствуют ветвям, расположенным на более низких уровнях дерева. Вложенный список ответвляется от списка, в котором он содержится. Терминальные узлы вместо новых вложенных списков содержат оценку уровня счастья.

Мы успешно создали дерево принятия решений, которое позволяет прогнозировать уровни счастья с относительно низкой ошибкой. Проанализируйте вывод, обращая внимание на относительные детерминанты счастья и уровни счастья, связанные с каждой ветвью.

С деревьями принятия решений и набором данных можно провести еще много интересных исследований. Например, вы можете попробовать выполнить тот же код с другим или большим набором переменных. Можно также создать дерево с другой максимальной длиной. Пример выполнения кода с другим списком переменных и глубиной:

```
variables = ['sclmeet','rlgdgr','hhmmb','netusoft','agea','eduyrs','health']
outcome_variable = 'happy'
maxdepth = 3
print(getsplit(0,ess,variables,outcome_variable))
```

С этими параметрами вы получите совершенно другое дерево принятия решений. Результат выглядит так:

```
[['health', -inf, 2.0, [['sclmeet', -inf, 4.0, [['health', -inf, 1.0, [['rlgdgr',
-inf, 9.0, 7.9919636617749825], ['rlgdgr', 9.0, inf, 8.713414634146341]]],
['health', 1.0, inf, [['netusoft', -inf, 4.0, 7.195121951219512], ['netusoft',
4.0, inf, 7.565659008464329]]]]], ['sclmeet', 4.0, inf, [['eduyrs',
-inf, 25.0, [['eduyrs', -inf, 8.0, 7.9411764705882355], ['eduyrs', 8.0,
inf, 7.999169779991698]]], ['eduyrs', 25.0, inf, [['hhmmb', -inf, 1.0,
```

```
7.297872340425532], ['hhmmb', 1.0, inf, 7.9603174603174605]]]]]]], ['health',
2.0, inf, [['sclmeet', -inf, 3.0, [['health', -inf, 3.0, [['sclmeet', -inf,
2.0, 6.049427365883062], ['sclmeet', 2.0, inf, 6.70435393258427]]], ['health',
3.0, inf, [['sclmeet', -inf, 1.0, 4.135036496350365], ['sclmeet', 1.0, inf,
5.407051282051282]]]]], ['sclmeet', 3.0, inf, [['health', -inf, 4.0, [['rlgdgr',
-inf, 9.0, 6.992227707173616], ['rlgdgr', 9.0, inf, 7.434662998624484]]],
['health', 4.0, inf, [['hhmmb', -inf, 1.0, 4.948717948717949], ['hhmmb', 1.0, inf,
6.132075471698113]]]]]]]]
```

В частности, обратите внимание на то, что первая ветвь разбивается по переменной `health` вместо `netusoft`. Другие ветви на более низких уровнях разбиваются в других точках и по разным переменным. Гибкость метода деревьев принятия решений означает, что два исследователя, начинающие с одного набора данных и одной цели, могут прийти к совершенно разным заключениям в зависимости от параметров и решений, относящихся к работе с данными. Эта особенность типична для методов машинного обучения и является одной из причин, по которым их так трудно применять.

Оценка дерева принятия решений

Чтобы сгенерировать дерево принятия решений, мы сравнивали ошибки для каждой потенциальной точки разбиения и каждой потенциальной переменной разбиения и всегда выбирали переменную и точку разбиения, обеспечивавшие наименьшую величину ошибок для конкретной ветви. Теперь, когда дерево принятия решений успешно сгенерировано, стоит провести аналогичные вычисления по оценке ошибок не только для конкретной ветви, но и для всего дерева. Оценка уровня ошибки для всего дерева может дать нам представление о том, насколько хорошо мы справились с задачей прогнозирования и каких результатов стоит ожидать от будущих задач (например, диагностики будущих пациентов, жалующихся на боль в груди).

Взглянув на дерево принятия решений, сгенерированное к настоящему моменту, можно заметить, что все эти вложенные списки плохо читаются и не существует естественного способа спрогнозировать уровень счастья, обойдясь без мучительного прохождения всех вложенных ветвей в поисках нужного терминального узла. Будет полезно написать код, который может определить прогнозируемый уровень счастья исходя из того, что нам известно о человеке на основании его ответов ESS. Задача решается с помощью функции `get_prediction()`:

```
def get_prediction(observation,tree):
    j = 0
    keepgoing = True
    prediction = - 1
    while(keepgoing):
```

```
        j = j + 1
        variable_tocheck = tree[0][0]
        bound1 = tree[0][1]
        bound2 = tree[0][2]
        bound3 = tree[1][2]
        if observation.loc[variable_tocheck] < bound2:
            tree = tree[0][3]
        else:
            tree = tree[1][3]
        if isinstance(tree,float):
            keepgoing = False
            prediction = tree
    return(prediction)
```

Далее создается цикл, который перебирает любую часть набора данных и получает для нее прогноз уровня счастья. В данном случае опробуем дерево с максимальной глубиной 4:

```
predictions=[]
outcome_variable = 'happy'
maxdepth = 4
thetree = getsplit(0,ess,variables,outcome_variable)
for k in range(0,30):
    observation = ess.loc[k,:]
    predictions.append(get_prediction(observation,thetree))

print(predictions)
```

Этот код многократно вызывает функцию `get_prediction()` и присоединяет результат к списку прогнозов. В данном случае прогнозы делаются только для первых 30 наблюдений.

Наконец, мы проверяем, как прогнозы соотносятся с реальными оценками счастья, чтобы узнать общий уровень ошибок. Мы сделаем прогнозы для всего набора данных и вычислим абсолютное расхождение между прогнозами и зарегистрированными уровнями счастья:

```
predictions = []

for k in range(0,len(ess.index)):
    observation = ess.loc[k,:]
    predictions.append(get_prediction(observation,thetree))

ess.loc[:,'predicted'] = predictions
errors = abs(ess.loc[:,'predicted'] - ess.loc[:,'happy'])

print(np.mean(errors))
```

При выполнении этого фрагмента мы видим, что средняя ошибка для прогнозов в дереве принятия решений составляет `1.369`. Величина больше нуля, но меньше того, что могло бы быть при худшем методе прогнозирования. Похоже, наше дерево принятия решений неплохо справляется с прогнозированием.

Проблема переобучения

Возможно, вы заметили одно очень важное отличие нашего способа оценки дерева принятия решений от того, как работают прогнозы в реальной жизни. Вспомните, что было сделано: мы задействовали полный набор респондентов, чтобы сгенерировать дерево принятия решений, после чего использовали тот же набор для оценки точности прогнозов нашего дерева. Но прогнозы уровней счастья для респондентов, уже прошедших опрос, избыточны — ведь они уже ответили на вопросы, поэтому мы знаем их уровни счастья, и прогнозировать их вообще не нужно! Все выглядит так, словно вы получили набор данных пациентов с инфарктом, тщательно проанализировали все симптомы, предшествовавшие лечению, и построили модель, которая сообщает, перенес ли кто-то инфаркт на прошлой неделе. Но к этому моменту уже хорошо известно, был ли инфаркт у данного человека, и для получения нужной информации необязательно заглядывать в исходные диагностические данные. Прошлое прогнозировать легко, но помните, что настоящие прогнозы всегда направлены в будущее. Как говорит профессор Джозеф Симмонс (Joseph Simmons), «история — это о том, что произошло. Наука — это о том, что произойдет *дальше*».

Может показаться, что проблема не столь серьезна. В конце концов, если мы можем создать дерево принятия решений, которое хорошо работает для пациентов прошлой недели, то разумно предположить, что оно будет хорошо работать и для пациентов следующей недели. И это до определенной степени так. Тем не менее существует опасность того, что мы можем столкнуться с распространенной и очень коварной опасностью *переобучения* (overfitting) — склонности моделей машинного обучения достигать очень низких уровней ошибок для наборов данных, которые использовались для их создания (данные из прошлого), а затем демонстрировать неожиданно высокие уровни ошибок для других данных (тех, которые для вас действительно важны, то есть данных из будущего).

Для примера возьмем прогнозирование инфаркта. Если вы понаблюдаете за работой врача скорой помощи в течение нескольких дней, то может оказаться, что по стечению обстоятельств все пациенты в синих рубашках страдали от инфаркта, а все пациенты в зеленых рубашках были здоровы. Модель дерева принятия решений, включающая цвет рубашки в прогностические переменные, обнаружит закономерность и использует ее как переменную ветвления, поскольку она обладает столь высокой диагностической точностью в наших наблюдениях. Но если

потом это дерево принятия решений будет использоваться для прогнозирования инфаркта в другой больнице или в другой день, то окажется, что прогнозы часто оказываются ошибочными: многие люди в зеленых рубашках также страдают от инфаркта, а многие люди в синих рубашках — нет. Наблюдения, использованные для построения дерева принятия решений, называются *внутривыборочными*, а наблюдения, используемые для тестирования модели — *вневыборочными*. Переобучение означает, что фанатичное стремление к низким уровням ошибок в прогнозах по внутривыборочным наблюдениям приводит к тому, что дерево принятия решений будет иметь аномально высокие уровни ошибок при прогнозировании вневыборочных наблюдений.

Переобучение является серьезной проблемой во всех приложениях машинного обучения, о которую спотыкаются даже лучшие специалисты-практики в области машинного обучения. Чтобы избежать ее, мы предпримем важный шаг, который сделает наш процесс создания дерева принятия решений более похожим на прогностические сценарии из реальной жизни.

Помните, что прогнозирование в реальной жизни направлено на будущее, но при построении дерева принятия решений мы неизбежно располагаем данными только из прошлого. Получить данные из будущего невозможно, поэтому набор данных разбивается на два поднабора: *обучающий*, который будет использоваться только для построения дерева принятия решений, и *тестовый*, который будет служить только для проверки точности дерева принятия решений. Тестовый набор получен из прошлого, как и остальные данные, но мы работаем с ним так, будто он из будущего. Он не задействован в создании дерева принятия решений (так, словно дерево еще не построено), но используется только после полного построения дерева для проверки точности дерева принятия решений (словно был получен позднее).

Благодаря простому разбиению данных на обучающий и тестовый наборы мы добились того, что процесс генерирования дерева напоминает реальную задачу прогнозирования неизвестного будущего; тестовый набор напоминает моделируемое будущее. Частота ошибок, наблюдаемая для тестового набора, определяет разумное ожидание уровня ошибок, который будет наблюдаться в реальном будущем. Если уровень ошибок в обучающем наборе очень низок, а в тестовом наборе очень высок, значит, произошло переобучение. Обучающий и тестовый наборы определяются следующим образом:

```
import numpy as np
np.random.seed(518)
ess_shuffled = ess.reindex(np.random.permutation(ess.index)).reset_index(drop = True)
training_data = ess_shuffled.loc[0:37000,:]
test_data = ess_shuffled.loc[37001:,:].reset_index(drop = True)
```

В этом фрагменте модуль `numpy` используется для случайной перестановки данных — другими словами, все данные остаются, но строки переставляются в случайном порядке. Перестановка выполняется с помощью метода `reindex()` модуля `pandas`, который меняет номера строк в случайном порядке. После перестановки набора данных первые 37 000 переставленных строк выбираются для обучающего набора данных, а остальные строки — для тестового. Вообще говоря, выполнять команду `np.random.seed(518)` необязательно, но если вы выполните ее, то получите те же псевдослучайные результаты, приведенные в тексте.

Когда обучающие и тестовые данные будут определены, генерируется дерево принятия решений на основании только обучающих данных:

```
thetree = getsplit(0,training_data,variables,outcome_variable)
```

Наконец, мы проверяем среднюю величину ошибки для тестовых данных, которые не использовались для обучения дерева принятия решений:

```
predictions = []
for k in range(0,len(test_data.index)):
    observation = test_data.loc[k,:]
    predictions.append(get_prediction(observation,thetree))

test_data.loc[:,'predicted'] = predictions
errors = abs(test_data.loc[:,'predicted'] - test_data.loc[:,'happy'])
print(np.mean(errors))
```

Как выясняется, средняя величина ошибки для тестовых данных составляет `1.371`. Это лишь немногим выше ошибки `1.369`, вычисленной при использовании всего набора данных для обучения и тестирования. Это означает, что наша модель не страдает от переобучения: она хорошо справляется с прогнозированием как прошлого, так и будущего. Достаточно часто вместо хороших новостей мы получаем плохие — модель ведет себя хуже, чем предполагалось. Тем не менее и эти новости полезны, поскольку исправления лучше вносить до того, как модель начнет использоваться в реальной ситуации. В таких случаях необходимо доработать модель, чтобы величина ошибки *для тестового набора* была минимизирована.

Улучшения и доработка

Может оказаться, что созданное вами дерево принятия решений имеет более низкую точность, чем вам хотелось бы. Например, более низкая точность может объясняться последствиями переобучения. Многие стратегии для решения проблем переобучения сводятся к тем или иным упрощениям, поскольку в простых моделях машинного обучения она проявляется реже, чем в сложных.

Первый и самый простой способ упрощения моделей дерева принятия решений — ограничение максимальной глубины; сделать это несложно, так как она задается переменной, которая переопределяется в одной короткой строке. Чтобы определить правильную глубину, необходимо проверить величину ошибки для вневыборочных данных с разными глубинами. Если глубина слишком велика, то это с большой вероятностью приведет к высокой ошибке из-за переобучения. Если глубина слишком мала, то высокая ошибка произойдет из-за недообучения. *Недообучение* (underfitting) может рассматриваться как явление, обратное переобучению. Последнее возникает из-за попыток обучения модели на произвольных или неактуальных закономерностях — другими словами, модель узнает «слишком много» из шума в обучающих данных (например, носит ли пациент зеленую рубашку). Недообучение же возникает из-за того, что при обучении было получено недостаточно информации — созданные модели упускают критические закономерности в данных (например, избыточный вес или курение).

Переобучение обычно возникает из-за того, что модель использует слишком много переменных или имеет слишком большую глубину, тогда как недообучение характерно для моделей с нехваткой переменных или недостаточной глубиной. Как и во многих ситуациях с проектированием алгоритмов, правильный выбор лежит где-то посередине между слишком высоким и слишком низким значениями. Выбор правильных параметров для модели машинного обучения, включая глубину дерева принятия решений, часто называется *настройкой* (tuning) по аналогии с тем, как изменение натяжения струны на гитаре или скрипке помогает достичь золотой середины между слишком высоким и слишком низким звуком.

Другой способ упрощения модели принятия решений связан с тем, что называется *отсечением* (pruning). Для этого дерево принятия решений наращивается до полной длины, после чего в нем находятся ветви, которые можно удалить, не вызывая значительное увеличение ошибок.

Еще одно улучшение, о котором также стоит упомянуть, — использование разных метрик для выбора правильной точки разбиения и правильной переменной разбиения. В этой главе была представлена идея использования суммарной ошибки классификации для выбора точки разбиения; оптимальной точкой разбиения считается та, которая минимизирует суммарную ошибку. Однако существуют и другие способы выбора точек разбиения для деревьев принятия решений, включая примеси Джини, энтропию, прирост информации и понижение дисперсии. На практике эти метрики, особенно примеси Джини и прирост информации, почти всегда используются вместо величины ошибки классификации, поскольку некоторые математические свойства делают их предпочтительными во многих случаях. Поэкспериментируйте с разными способами выбора точки разбиения и переменной

разбиения и найдите тот вариант, который показывает лучшие результаты для ваших данных и вашей задачи принятия решений.

Все, что мы делаем в области машинного обучения, направлено на формирование точных прогнозов с новыми данными. Пытаясь улучшить модель машинного обучения, вы всегда можете судить об эффективности своих действий; для этого достаточно проверить, насколько они улучшают величину ошибки для тестовых данных. И не бойтесь действовать творчески — все, что улучшает ошибку для тестовых данных, заслуживает вашего внимания.

Случайные леса

Деревья принятия решений безусловно полезны, но профессионалы не считают их лучшим методом машинного обучения. Отчасти это связано с репутацией самих деревьев (переобучение, относительно высокая величина ошибки), а отчасти с изобретением метода *случайных лесов*. Этот метод, завоевавший популярность в последнее время, обеспечивает бесспорный выигрыш по эффективности по сравнению с деревьями принятия решений.

Как подсказывает название, модель случайного леса состоит из набора моделей деревьев принятия решений. Каждое дерево принятия решений в случайном лесу зависит от рандомизации (внесения элемента случайности). Благодаря использованию случайного фактора мы получаем разнообразный лес с множеством деревьев вместо одного леса, в котором одно дерево повторяется снова и снова. Рандомизация происходит в двух местах. Во-первых, рандомизируется обучающий набор данных: каждое дерево строится на подмножестве обучающего набора, который выбирается случайным образом, а следовательно, будет отличаться для разных деревьев. (Тестовый набор выбирается случайным образом в начале процесса, но не проходит повторной рандомизации или повторного выбора для каждого дерева.) Во-вторых, переменные, служащие для построения дерева, также рандомизируются: для каждого дерева используется только некоторое подмножество полного набора переменных, и эти подмножества тоже могут быть разными.

Когда набор рандомизированных деревьев будет построен, в вашем распоряжении появляется целый случайный лес. Чтобы сделать прогноз относительно конкретного наблюдения, необходимо определить, что прогнозирует каждое из разных деревьев принятия решений, а затем усреднить прогнозы всех отдельных деревьев. Поскольку деревья принятия решений рандомизируются как по данным, так и по переменным, усреднение помогает обойти проблему переобучения и часто приводит к более точным прогнозам.

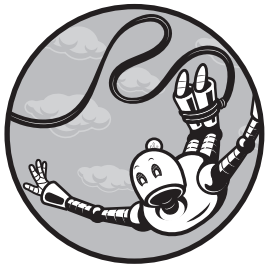
Наш код в этой главе строит деревья принятия решений с нуля за счет непосредственных манипуляций с наборами данных списками и циклами. Когда вы будете работать с деревьями принятия решений и случайными лесами в будущем, вы сможете воспользоваться модулями Python, выполняющими большую часть черной работы. Однако не рассчитывайте, что модули сделают все за вас: если вы понимаете каждый шаг этих важных алгоритмов достаточно хорошо, чтобы запрограммировать их самостоятельно, то ваша работа в области машинного обучения станет намного более эффективной.

Резюме

В этой главе были представлены методы машинного обучения, а также рассмотрены деревья принятия решений — фундаментальный, простой и полезный метод машинного обучения. Деревья принятия решений являются алгоритмом, а генерирование дерева принятия решений также представляет собой алгоритм, так что в текущей главе был описан алгоритм для генерирования алгоритма. Ознакомившись с деревьями принятия решений и фундаментальными идеями случайных лесов, вы сделали большой шаг к тому, чтобы стать экспертом в области машинного обучения. Знания, которые вы получили в этой главе, закладывают прочную основу для изучения других алгоритмов машинного обучения, в том числе работы нейронных сетей. Все методы машинного обучения пытаются делать то, чем мы занимались в этой главе: прогнозировать на основе закономерностей в наборе данных. В следующей главе обсудим искусственный интеллект — одно из самых современных и сложных направлений в рассматриваемой нами области.

10

Искусственный интеллект



В книге мы неоднократно отмечали способность человеческого разума делать разные замечательные вещи: ловить бейсбольные мячи, исправлять ошибки в текстах или диагностировать инфаркт. Мы рассмотрели возможности преобразования этих способностей в алгоритмы и некоторые проблемы, связанные с этим. В данной главе мы вернемся к этим проблемам и построим алгоритм для искусственного интеллекта (ИИ). Рассматриваемый ИИ-алгоритм будет применим не только для одной узкой задачи (например, ловли мяча), но и к широкому диапазону сценариев. Именно широкая применимость искусственного интеллекта и привлекает людей — подобно тому как человек изучает новые навыки в своей жизни, лучший ИИ сможет применяться в совершенно неизвестных ему областях с минимальным изменением конфигурации.

Термин *«искусственный интеллект»* окружен аурой, из-за которой люди думают, что это нечто таинственное и непостижимое. Некоторые полагают, что ИИ позволяет компьютерам думать, чувствовать и сознательно мыслить так же, как это делают люди; смогут ли компьютеры когда-нибудь делать это — трудный вопрос, выходящий далеко за рамки данной главы. Искусственный интеллект, который мы напишем, будет намного проще. Он хорошо справляется с игрой, но не сможет искренне писать любовную лирику, чувствовать уныние или вожделение (насколько я могу судить!).

Наш искусственный интеллект сможет играть в «Точки и квадраты» — простую, но нетривиальную игру, известную во всем мире. Начнем с рисования игрового поля, а затем построим функции для ведения счета по ходу игры. Далее займемся генерированием деревьев игры, представляющих все возможные комбинации ходов для заданной игры. В завершение представим минимаксный алгоритм — элегантный способ реализации искусственного интеллекта всего в нескольких строках.

Точки и квадраты

Игру «Точки и квадраты» придумал французский математик Эдуард Лукас. Игра начинается с *сетки* из точек, похожей на ту, которая изображена на рис. 10.1.

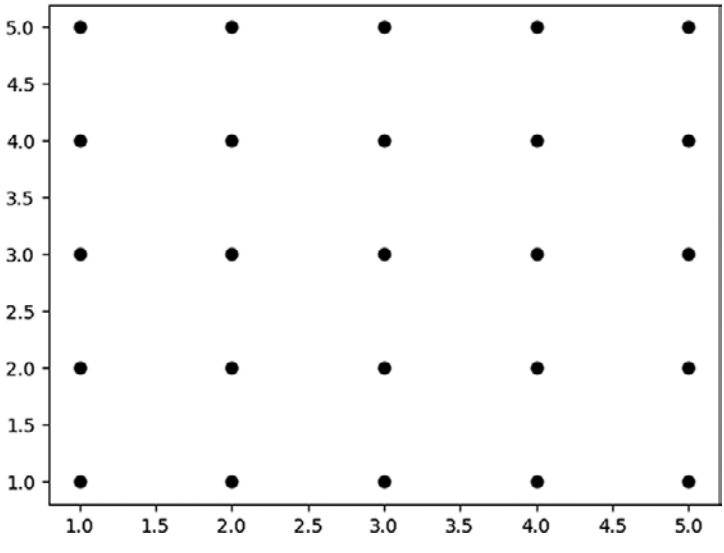


Рис. 10.1. Сетка для игры «Точки и квадраты»

Обычно используется прямоугольная сетка, но она может иметь любую форму. Два игрока совершают ходы по очереди. При каждом ходе игрок проводит линию, соединяющую две соседние точки на сетке. Если игроки рисуют линии разными цветами, то вы сразу видите, кто какую линию провел, но это необязательно. В ходе игры линии заполняют сетку, пока не будут нарисованы все возможные линии, соединяющие соседние точки. Пример незаконченной партии показан на рис. 10.2.

Цель игрока в «Точках и квадратах» — рисование линий, завершающих квадраты. На рис. 10.2 вы видите, что в левом нижнем углу поля находится один достроенный

квадрат. Игрок, нарисовавший последний линию, достроившую квадрат, получает за это одно очко.

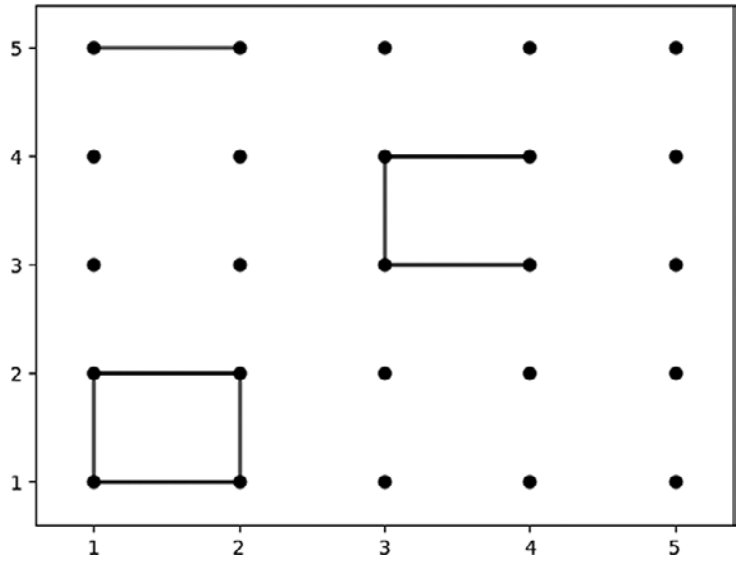


Рис. 10.2. Незавершенная партия в игре «Точки и квадраты»

В правой верхней части поля были нарисованы три стороны квадрата. Если сейчас ход первого игрока и он использует свой ход для рисования линии между точками (4,4) и (4,3), то получит за это одно очко. Если вместо этого он проведет другую линию (например, из точки (4,1) в (5,1)), то это даст возможность второму игроку достроить квадрат и получить очко. Игроки получают очки только за завершение наименьших возможных квадратов на поле, то есть квадратов со стороной 1. В игре побеждает игрок, получивший больше очков при заполнении всего поля. У игры есть ряд вариаций, включая разную форму полей и более сложные правила, но простой искусственный интеллект, который мы построим в этой главе, будет работать с правилами, описанными выше.

Рисование игрового поля

Хоть это и не является строго обязательным для алгоритмических целей, рисование игрового поля упростит визуализацию рассматриваемых идей. Очень простая функция вывода создает сетку $n \times n$, для чего она перебирает координаты x и y и использует функцию `plot()` в модуле Python `matplotlib`:

```
import matplotlib.pyplot as plt
from matplotlib import collections as mc
def drawlattice(n,name):
    for i in range(1,n + 1):
        for j in range(1,n + 1):
            plt.plot(i,j,'o',c = 'black')
    plt.savefig(name)
```

В этом коде *n* представляет размер каждой стороны сетки, а аргумент `name` определяет путь к файлу, в котором будет сохраняться вывод. Аргумент `c = 'black'` задает цвет точек сетки. Черная сетка 5×5 создается и сохраняется в файле с помощью следующей команды:

```
drawlattice(5,'lattice.png')
```

Именно эта команда была использована для создания рис. 10.1.

Представление партии

Игра «Точки и квадраты» состоит из последовательного рисования линий, поэтому партию можно записать в виде списка упорядоченных линий. По аналогии с тем, как это делалось в предыдущих главах, линия (один ход) может быть представлена списком из двух упорядоченных пар (координаты конечных точек линий). Например, линия между точками (1,2) и (1,1) представляется следующим списком:

```
[(1,2),(1,1)]
```

Партия представляется упорядоченным списком таких линий, как в следующем примере:

```
game = [[(1,2),(1,1)],[(3,3),(4,3)],[(1,5),(2,5)],[(1,2),(2,2)],[(2,2),(2,1)],\
[(1,1),(2,1)],[(3,4),(3,3)],[(3,4),(4,4)]]
```

Эта партия представлена выше на рис. 10.2. Мы видим, что партия не закончена, поскольку сетка еще не заполнена всеми возможными линиями.

Функцию `drawlattice()` можно расширить для создания функции `drawgame()`. Она должна рисовать точки игрового поля, а также все линии, проведенные между ними к текущему моменту партии. Эту задачу решает функция из листинга 10.1.

Листинг 10.1. Функция, рисующая игровое поле для игры «Точки и квадраты»

```
def drawgame(n,name,game):
    colors2 = []
    for k in range(0,len(game)):
```

```
        if k%2 == 0:
            colors2.append('red')
        else:
            colors2.append('blue')
    lc = mc.LineCollection(game, colors = colors2, linewidths = 2)
    fig, ax = plt.subplots()
    for i in range(1,n + 1):
        for j in range(1,n + 1):
            plt.plot(i,j,'o',c = 'black')
    ax.add_collection(lc)
    ax.autoscale()
    ax.margins(0.1)
    plt.savefig(name)
```

Функция получает аргументы `n` и `name`, как это делала функция `drawlattice()`. Вдобавок она включает точно такие же вложенные циклы, как и те, которые использовались для рисования точек в `drawlattice()`. Первое дополнение — список `colors2`, который изначально пуст, но заполняется цветами, которые назначаются выводимым линиям. В «Точках и квадратах» ходы выполняются игроками поочередно, поэтому мы будем чередовать цвета линий — в данном случае красный для первого игрока, синий для второго. Цикл `for` после определения списка `colors2` заполняет его чередующимися экземплярами `'red'` и `'blue'`, пока количество назначенных цветов не сравняется с количеством сделанных ходов. Другие добавленные строки кода создают коллекцию линий и рисуют их так же, как выводились коллекции линий в предыдущих главах.

ПРИМЕЧАНИЕ

Эта книга печатается в черно-белом варианте, но я все равно включаю код цветов в примеры, чтобы вы видели цвета при выполнении кода на своем компьютере.

Функцию `drawgame()` можно вызвать в одной строке:

```
drawgame(5,'gameinprogress.png',game)
```

Именно так был построен рис. 10.2.

Ведение счета

Затем мы создадим функцию, которая ведет счет в игре «Точки и квадраты». Начнем с функции, которая получает произвольную партию и находит завершенные квадраты. Затем будет создана функция для вычисления счета. Наша функция подсчитывает завершенные квадраты, перебирая все линии в игре. Если линия

горизонтальная, то мы определяем, является ли она верхней стороной завершенного квадрата; для этого она проверяет, чтобы в партии была проведена параллельная линия внизу, а также линии левой и правой сторон квадрата. Эту задачу решает функция в листинге 10.2.

Листинг 10.2. Функция для подсчета квадратов в игре «Точки и квадраты»

```
def squarefinder(game):
    countofsquares = 0
    for line in game:
        parallel = False
        left=False
        right=False
        if line[0][1]==line[1][1]:
            if [(line[0][0],line[0][1]-1),(line[1][0],line[1][1] - 1)] in game:
                parallel=True
            if [(line[0][0],line[0][1]),(line[1][0]-1,line[1][1] - 1)] in game:
                left=True
            if [(line[0][0]+1,line[0][1]),(line[1][0],line[1][1] - 1)] in game:
                right=True
            if parallel and left and right:
                countofsquares += 1
    return(countofsquares)
```

Как видите, функция возвращает значение переменной `countofsquares`, которая была инициализирована `0` в начале функции. Цикл `for` в функции перебирает все линии в игре. Сначала мы предполагаем, что в игре еще не была проведена ни параллельная линия внизу, ни левая и правая линии, соединяющие эти параллельные линии. Если заданная линия является горизонтальной, проверяем существование параллельных, левой и правой линий. Если все четыре линии проверяемого квадрата проведены, то переменная `countofsquares` увеличивается на 1. В результате в `countofsquares` будет храниться общее количество квадратов, полностью нарисованных в игре к настоящему моменту.

Теперь можно написать короткую функцию для вычисления счета игры. Он будет сохраняться в списке из двух элементов — например, `[2,1]`. Первый элемент списка представляет счет первого игрока, а второй элемент — счет второго. В листинге 10.3 приведена функция вычисления счета.

Листинг 10.3. Функция для вычисления текущего счета игры «Точки и квадраты»

```
def score(game):
    score = [0,0]
    progress = []
    squares = 0
    for line in game:
```

```
        progress.append(line)
        newsquares = squarefinder(progress)
        if newsquares > squares:
            if len(progress)%2 == 0:
                score[1] = score[1] + 1
            else:
                score[0] = score[0] + 1
        squares=newsquares
    return(score)
```

Функция вычисления счета последовательно обрабатывает каждую линию в игре. Она рассматривает незавершенную партию, которая состоит из всех линий, нарисованных до текущего хода. Если общее количество квадратов, завершенных в партии, превышает количество квадратов на предыдущий ход, то мы знаем, что сделавший ход игрок получил очко и его счет увеличится на 1. Чтобы просмотреть счет партии, изображенной выше на рис. 10.2, выполните команду `print(score(game))`.

Деревья игры и как победить

Итак, теперь вы знаете, как выводить состояние игры и как определять текущий счет. Остается понять, как победить в «Точках и квадратах». Возможно, эта игра не представляет особого интереса с точки зрения игровой теории, но подход к достижению победы в ней приблизительно такой же, как в шахматах, шашках и крестиках-ноликах, а алгоритм победы во всех этих играх может дать вам новую точку зрения на любые конфликтные ситуации, с которыми вы столкнетесь в жизни. Суть выигрышной стратегии заключается в систематическом анализе будущих последствий текущих действий и выборе действия, которое приводит к лучшему исходу в будущем. На первый взгляд, такая формулировка кажется тавтологией, но достижение цели базируется на тщательном систематическом анализе; его можно представить в виде дерева наподобие тех, которые мы строили в главе 9.

Рассмотрим некоторые варианты исходов, показанные на рис. 10.3.

Анализ начинается с вершины дерева и рассмотрения текущей ситуации: мы отстаем со счетом 0–1, сейчас наш ход. Первым рассматривается ход на левой ветви: линия, соединяющая точку (4,4) с точкой (4,3). Этот ход достраивает квадрат и дает одно очко. Неважно, что сделает противник (варианты перечислены в двух ветвях внизу слева на рис. 10.3), — после его следующего хода счет будет ничейным. С другой стороны, если мы используем текущий ход для соединения точек (1,3) и (2,3), как показано в правой ветви на рис. 10.3, у противника появляется выбор между соединением точек (4,4) и (4,3) с завершением квадрата и получением очка и соединением точек (3,1) и (4,1) с сохранением счета 0–1.

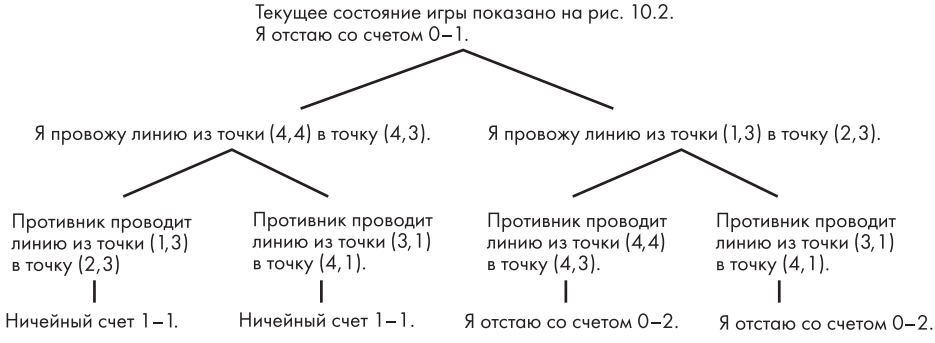


Рис. 10.3. Дерево возможных продолжений игры

Рассматривая эти возможности, мы видим, что в пределах двух ходов игра может перейти к одному из трех вариантов счета: 1–1, 0–2 и 0–1. В представленном дереве ясно, что выбирать нужно левую ветвь, поскольку каждая возможность, происходящая из этой ветви, ведет к лучшему для нас счету, чем возможности, происходящие из правой ветви. Подобные рассуждения составляют суть того, как наш искусственный интеллект будет выбирать следующий ход. Он строит дерево игры, проверяет результаты для всех терминальных узлов дерева игры, а потом использует простые рекурсивные рассуждения для выбора хода в свете возможных исходов, которые открывает это решение.

Вероятно, вы заметили, что дерево на рис. 10.3 неполно. На нем рассматриваются всего два возможных хода (левая и правая ветви), а после каждого из этих возможных ходов рассматриваются всего два возможных хода противника. Конечно, это неправильно; каждому из игроков доступно множество вариантов. Помните, что ход может соединять любые две соседние точки сетки. Полноценное дерево, представляющее текущее состояние игры, будет состоять из множества ветвей, по одной для каждого возможного хода игрока. И это относится к каждому уровню дерева: у меня есть выбор из множества ходов, он есть и у моего противника, и каждый из этих ходов имеет собственную ветвь в каждой точке дерева, где он может быть сделан. Только в самом конце игры, когда почти все линии уже были проведены, количество возможных ходов сокращается до двух и одного. Мы не стали рисовать каждую ветвь дерева игры на рис. 10.3, поскольку на странице не хватило бы места — мы смогли включить только пару ходов просто для того, чтобы продемонстрировать идею дерева игры и суть рассуждений.

Дерево игры может простираться на любую глубину — рассматривать следует не только наш ход и ответ противника, но и наш ответ на его ответ, потом ответ противника на наш ответ и т. д., пока мы готовы продолжать построение дерева.

Построение дерева

Деревья игры, которые мы будем строить, во многих важных отношениях отличаются от деревьев решений из главы 9. Самое важное отличие — цель: деревья принятия решений обеспечивают возможность классификации и прогнозов, основанных на характеристиках, тогда как деревья игры просто описывают все возможные исходы в будущем. Так как цели различаются, будут различаться и способы построения деревьев. Напомню, что в главе 9 для определения каждой ветви дерева нам приходилось выбирать переменную и точку разбиения. Тем не менее определить следующую ветвь несложно, поскольку существует ровно одна ветвь для каждого возможного хода. Все, что от нас потребуется, — сгенерировать список для всех возможных ходов игры. Для этого можно воспользоваться парой вложенных циклов, перебирающих все возможные соединения между точками сетки:

```
allpossible = []

gamesize = 5

for i in range(1,gamesize + 1):
    for j in range(2,gamesize + 1):
        allpossible.append([(i,j),(i,j - 1)])

for i in range(1,gamesize):
    for j in range(1,gamesize + 1):
        allpossible.append([(i,j),(i + 1,j)])
```

Фрагмент начинается с определения пустого списка `allpossible` и переменной `gamesize`, определяющей длину каждой стороны сетки. Далее следуют два цикла. Первый добавляет вертикальные ходы в список возможных ходов. Для всех возможных значений `i` и `j` первый цикл присоединяет ход, представленный списком `[(i,j),(i,j - 1)]`, в наш список возможных ходов. Этот ход всегда будет вертикальной линией. Второй цикл похож на первый, но для каждой возможной комбинации `i` и `j` к списку возможных ходов присоединяется горизонтальный ход `[(i,j),(i + 1,j)]`. В конце список `allpossible` будет заполнен всеми возможными ходами.

В текущей партии (наподобие той, что изображена выше на рис. 10.2) возможны не все ходы. Если игрок сделал в игре какой-то ход, то никто из игроков уже не сможет сделать этот ход в оставшейся части игры. Необходимо исключить из списка всех возможных ходов все те, что уже были сделаны; в результате мы получим список всех возможных ходов, оставшихся в конкретной незавершенной партии. Сделать это несложно:

```
for move in allpossible:
    if move in game:
        allpossible.remove(move)
```

Как видите, мы перебираем все ходы в списке возможных ходов, и если ход уже был сделан, то исключается из списка. В результате мы получаем список только тех ходов, которые возможны в текущем состоянии игры. Чтобы просмотреть все эти ходы и убедиться в их правильности, выполните команду `print(allpossible)`.

Получив список всех возможных ходов, мы можем построить дерево игры. Оно сохраняется в виде вложенного списка ходов. Напомню, что каждый ход может быть сохранен в виде списка упорядоченных пар вида `[(4,4),(4,3)]` (первый ход на левой ветви, см. рис. 10.3). Чтобы представить дерево, состоящее только из двух верхних ходов на рис. 10.3, можно использовать следующую запись:

```
simple_tree = [[(4,4),(4,3)],[(1,3),(2,3)]]
```

Дерево содержит только два хода — те, которые рассматривались в текущем состоянии игры на рис. 10.3. Если хотите включить потенциальные ответы противника, то придется добавить еще один уровень вложенности. Для этого каждый ход помещается в список вместе с его потомками (ходами, ответвляющимися от него). Начнем с добавления пустых списков, представляющих потомков:

```
simple_tree_with_children = [[[(4,4),(4,3)],[]],[[(1,3),(2,3)],[]]]
```

Ненадолго задержитесь и убедитесь в том, что понимаете смысл всех вложений. Каждый ход представлен списком и к тому же является первым элементом списка, который также содержит потомков списка. Затем все списки сохраняются в главном списке, представляющем полное дерево.

Все дерево игры на представленном выше рис. 10.3, включая ответы противника, можно выразить следующей структурой вложенных списков:

```
full_tree = [[[(4,4),(4,3)],[[(1,3),(2,3)],[(3,1),(4,1)]]],[[(1,3),(2,3)],\
[[(4,4),(4,3)],[(3,1),(4,1)]]]]
```

Квадратные скобки быстро становятся слишком громоздкими и неудобными, но вложенная структура необходима, чтобы мы могли правильно хранить информацию о связях между родителями и потомками.

Вместо того чтобы записывать дерево игры вручную, мы построим функцию, которая будет создавать его за нас. На входе она получает список возможных ходов, а затем присоединяет каждый ход к дереву (листинг 10.4).

Листинг 10.4. Функция, создающая дерево игры с заданной глубиной

```
def generate_tree(possible_moves,depth,maxdepth):
    tree = []
    for move in possible_moves:
        move_profile = [move]
```

```
        if depth < maxdepth:
            possible_moves2 = possible_moves.copy()
            possible_moves2.remove(move)
            move_profile.append(generate_tree(possible_moves2,depth + 1,maxdepth))
        tree.append(move_profile)
    return(tree)
```

Функция `generate_tree()` начинается с определения пустого списка `tree`. Затем она перебирает все возможные ходы и для каждого хода создает список `move_profile`. Сначала `move_profile` состоит только из самого хода. Но для ветвей, которые еще не находятся на самом нижнем уровне дерева, необходимо добавить потомков этих ходов. Потомки добавляются рекурсивно: мы снова вызываем функцию `generate_tree()`, однако на сей раз из списка `possible_moves` исключается один ход. Наконец, список `move_profile` присоединяется к дереву.

Для вызова этой функции достаточно пары строк:

```
allpossible = [[(4,4),(4,3)],[(4,1),(5,1)]]
thetree = generate_tree(allpossible,0,1)
print(thetree)
```

В результате выполнения этого фрагмента будет получено следующее дерево:

```
[[[(4, 4), (4, 3)], [[[(4, 1), (5, 1)]]]], [[(4, 1), (5, 1)], [[[(4, 4), (4, 3)]]]]]
```

Пара дополнений сделает наше дерево более полезным: первое сохраняет игровой счет вместе с ходами, а второе присоединяет пустой список, чтобы зарезервировать место для потомков (листинг 10.5).

Листинг 10.5. Функция, генерирующая дерево игры с ходами-потомками и текущим счетом

```
def generate_tree(possible_moves,depth,maxdepth,game_so_far):
    tree = []
    for move in possible_moves:
        move_profile = [move]
        game2 = game_so_far.copy()
        game2.append(move)
        move_profile.append(score(game2))
        if depth < maxdepth:
            possible_moves2 = possible_moves.copy()
            possible_moves2.remove(move)
            move_profile.append(generate_tree(possible_moves2,depth +
                                1,maxdepth,game2))
      else:
          move_profile.append([])
      tree.append(move_profile)
  return(tree)
```

Вызовем обновленную функцию:

```
allpossible = [[(4,4),(4,3)],[(4,1),(5,1)]]
thetree = generate_tree(allpossible,0,1,[])
print(thetree)
```

Результат выглядит так:

```
[[[(4, 4), (4, 3)], [0, 0], [[[(4, 1), (5, 1)], [0, 0], []]]], [[(4, 1), (5, 1)],
[0, 0], [[[(4, 4), (4, 3)], [0, 0], []]]]
```

Мы видим, что каждый элемент дерева представляет собой полный профиль хода, состоящий из хода (например, `[(4,4),(4,3)]`), счета (например, `[0,0]`) и списка потомков (который может быть пустым).

Выигрыш

Наконец, все готово для создания функции, умеющей хорошо играть в «Точки и квадраты». Прежде чем писать код, рассмотрим принципы, на которых он базируется. А именно как мы, люди, играем в эту игру? Или на более общем уровне — как победить в любой стратегической игре (например, в шахматах или крестиках-ноликах)? В каждой игре существуют уникальные правила и возможности, но есть обобщенный механизм выбора выигрышной стратегии на основании дерева игры.

Алгоритм, который мы используем для выбора выигрышной стратегии, называется *минимаксным* (сочетание слов «минимум» и «максимум»). Выбор названия связан с тем, что мы пытаемся максимизировать свой счет в игре, тогда как наш противник пытается его минимизировать. Постоянный конфликт между максимизацией нашего счета и минимизацией счета противником как раз и составляет суть того, что мы должны стратегически учитывать при выборе правильного хода.

Повнимательнее присмотритесь к простому дереву игры на рис. 10.3. Теоретически оно может быть гигантским, с огромной глубиной и множеством ветвей на каждой глубине. Но любое дерево игры, большое или малое, состоит из одних и тех же компонентов: множества вложенных ветвей.

В точке, представленной на рис. 10.3, существуют два варианта (рис. 10.4).

Наша цель — максимизация счета. Чтобы выбрать между двумя ходами, необходимо знать, к какому исходу они приведут. Чтобы это знать, необходимо переместиться далее по дереву игры и рассмотреть все возможные последствия. Начнем с правого хода (рис. 10.5).

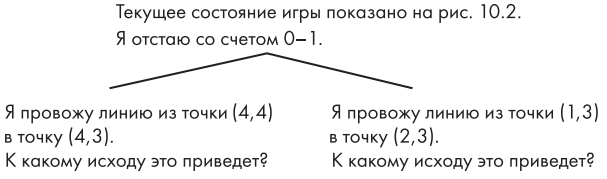


Рис. 10.4. Выбор одного из двух ходов

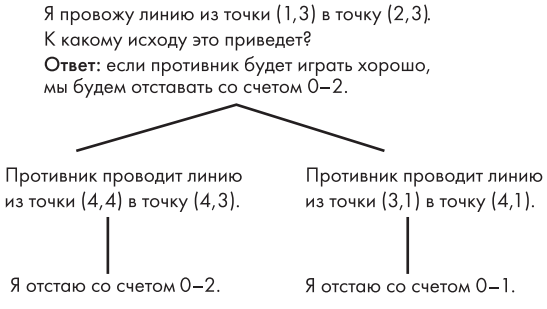


Рис. 10.5. Предполагая, что противник будет пытаться минимизировать ваш счет, можно узнать, к какому ожидаемому исходу приведет ход

Этот ход может привести к одному из двух возможных исходов: к концу дерева мы можем отставать со счетом 0–1 или отставать со счетом 0–2. Если противник играет хорошо, то будет стремиться максимизировать свой счет, что эквивалентно минимизации нашего счета. Если противник хочет минимизировать свой счет, то выберет ход, при котором мы будем отставать 0–2. С другой стороны, рассмотрим иной вариант: левую ветвь на рис. 10.5, возможные исходы которой представлены на рис. 10.6.

Рис. 10.6. Какой бы вариант ни выбрал противник, ожидается один и тот же исход

В этом случае оба варианта противника ведут к счету 1–1. Снова предполагая, что противник будет стремиться к минимизации нашего счета, мы говорим, что этот ход ведет к ничейному исходу 1–1.

Теперь мы знаем, к какому исходу приведет каждый из двух ходов. На рис. 10.7 эти исходы показаны в обновленной версии рис. 10.4.

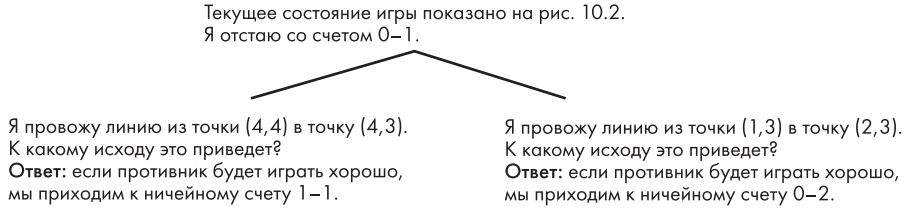


Рис. 10.7. По рис. 10.5 и 10.6 можно сделать вывод о том, к какому исходу приведет каждый ход, и сравнить их

Поскольку мы точно знаем, какого исхода можно ожидать от каждого из двух ходов, можно выполнить максимизацию: к максимуму (лучшему счету) ведет левый ход, поэтому мы выбираем его.

Этот процесс рассуждений называется минимаксным алгоритмом. Наше решение в текущий момент направлено на максимизацию счета. Но чтобы максимизировать наш счет, необходимо рассмотреть все возможности, которыми противник будет пытаться его минимизировать. Таким образом, лучшим вариантом становится максимум среди минимумов.

Следует учитывать, что минимаксный поиск происходит в обратном порядке. Игра двигается по времени вперед, от настоящего к будущему. Но в некотором смысле для минимаксного алгоритма время идет в обратном направлении, поскольку мы сначала рассматриваем счета разных исходов в далеком будущем, а затем возвращаемся к настоящему, чтобы выбрать текущий ход, приводящий к лучшему будущему. В контексте дерева игры минимаксный код начинает с начала дерева. Он рекурсивно вызывает себя для каждой из дочерних ветвей. В свою очередь, дочерние ветви рекурсивно вызывают минимаксный код для каждой из своих дочерних ветвей. Эти рекурсивные вызовы продолжаются на всем пути к терминальным узлам, где вместо нового вызова минимаксной функции вычисляется счет для каждого узла. Из-за этого мы сначала вычисляем счет для терминальных узлов; вычисление игрового счета начинается с далекого будущего. Счет передается родительским узлам, чтобы родительские узлы могли вычислить лучшие ходы и соответствующий счет для своей

части игры. Эти счета и ходы продолжают передаваться вверх по дереву игры, пока не достигнут самого верха — родительского узла, представляющего текущее состояние.

В листинге 10.6 приведена функция, реализующая минимаксный алгоритм.

Листинг 10.6. Функция, использующая минимаксный алгоритм для нахождения лучшего хода в дереве игры

```
import numpy as np
def minimax(max_or_min,tree):
    allscores = []
    for move_profile in tree:
        if move_profile[2] == []:
            allscores.append(move_profile[1][0] - move_profile[1][1])
        else:
            move,score=minimax((-1) * max_or_min,move_profile[2])
            allscores.append(score)
    newlist = [score * max_or_min for score in allscores]
    bestscore = max(newlist)
    bestmove = np.argmax(newlist)
    return(bestmove,max_or_min * bestscore)
```

Функция `minimax()` получилась относительно короткой. Большую ее часть занимает цикл `for`, который перебирает все профили ходов в нашем дереве. Если профиль хода не имеет дочерних ходов, то мы вычисляем счет, связанный с этим ходом, как разность между количествами квадратов у нас и у противника. Если профиль хода не имеет дочерних ходов, то `minimax()` вызывается для каждого потомка для получения счета, связанного с каждым ходом. Тогда остается только найти ход, связанный с максимальным счетом.

Эта функция вызывается для нахождения лучшего хода в любой точке любой незавершенной партии. Прежде чем вызывать `minimax()`, необходимо выполнить всю необходимую подготовку. Начнем с определения игры и получения всех возможных ходов, задействовав точно такой же код, который уже использовался ранее:

```
allpossible = []

game = [[(1,2),(1,1)],[(3,3),(4,3)],[(1,5),(2,5)],[(1,2),(2,2)],[(2,2),(2,1)],\
[(1,1),(2,1)],[(3,4),(3,3)],[(3,4),(4,4)]]

gamesize = 5

for i in range(1,gamesize + 1):
    for j in range(2,gamesize + 1):
        allpossible.append([(i,j),(i,j - 1)])
```

```
for i in range(1,gamesize):
    for j in range(1,gamesize + 1):
        allpossible.append([(i,j),(i + 1,j)])

for move in allpossible:
    if move in game:
        allpossible.remove(move)
```

Затем мы генерируем полное дерево игры, простирающееся на глубину трех уровней:

```
thetree = generate_tree(allpossible,0,3,game)
```

После того как дерево игры будет получено, вызывается функция `minimax()`:

```
move,score = minimax(1,thetree)
```

Далее остается проверить лучший ход:

```
print(thetree[move][0])
```

Мы видим, что лучшим ходом оказывается `[(4, 4), (4, 3)]` — ход, который завершает квадрат и дает нам очко. Наш искусственный интеллект умеет играть в «Точки и квадраты» и выбирать лучший ход! Вы можете опробовать другие размеры поля, другие сценарии игры или другую глубину дерева и убедиться в том, что реализация минимаксного алгоритма дает хороший результат. А в следующей книге я расскажу, как сделать так, чтобы искусственный интеллект не обрел самосознание и не решил уничтожить человечество.

Улучшения

Теперь вы умеете выполнять минимаксный поиск и сможете применить его в любой игре, в которую вам доведется играть. А можете использовать его в жизненных ситуациях, продумывая будущее и максимизируя все минимальные возможности. (Структура минимаксного алгоритма остается одинаковой во всех конкурентных сценариях, но для использования минимаксного кода в другой игре вам потребуется написать новый код для генерирования игрового дерева, перебора всех возможных ходов и вычисления игрового счета.)

Искусственный интеллект, построенный нами, обладает весьма умеренными возможностями. Он умеет играть только в одну игру с одной простой версией правил. В зависимости от того, на каком процессоре выполняется этот код, вероятно,

ИИ сможет заглянуть всего на несколько ходов вперед, пока каждое решение не будет требовать неразумного времени (нескольких минут и более). Возникает естественное желание усовершенствовать наш искусственный интеллект, чтобы сделать его более эффективным.

Первое, что определенно нуждается в усовершенствовании, — скорость нашего искусственного интеллекта. Он работает медленно из-за огромного размера деревьев игры, с которыми ему приходится работать. Один из основных способов улучшения быстродействия минимаксного алгоритма основан на *отсечении* ветвей дерева. Как вы, вероятно, помните из главы 9, отсечение основано на удалении ветвей дерева, если мы считаем их бесперспективными или они являются дубликатами другой ветви. Отсечение реализуется нетривиально, а для его эффективного выполнения придется изучать другие алгоритмы. Один из примеров — алгоритм *альфа-бета отсечения*, который перестает проверять подветви, если те однозначно хуже других подветвей в другой позиции дерева.

Другое естественное улучшение нашего искусственного интеллекта — поддержка разных правил или разных игр. Например, по одному из часто используемых правил в «Точках и квадратах» после получения очка игрок рисует новую линию. Иногда это приводит к каскадам, когда один игрок завершает сразу несколько квадратов за один ход. Это простое изменение изменяет стратегические аспекты игры, а их реализация требует внесения определенных изменений в код. Вы также можете попытаться реализовать искусственный интеллект, который играет в «Точки и квадраты» на сетке в форме креста или другой экзотической фигуры, влияющей на стратегию. Элегантность минимаксного алгоритма проявляется в том, что он не требует тонкого стратегического понимания игры; нужна только возможность заглядывать вперед, поэтому программист, плохо играющий в шахматы, может написать реализацию минимаксного алгоритма, которая обыграет его в шахматы.

Есть много эффективных алгоритмов повышения быстродействия компьютерных искусственных интеллектов, которые выходят за рамки этой главы. К числу таких методов относится обучение с подкреплением (когда, например, шахматная программа играет сама с собой, чтобы стать сильнее), методы Монте-Карло (когда программа для игры в сёги генерирует случайные партии, чтобы лучше понять возможности) и нейронные сети (когда программа для игры «Крестики-нолики» прогнозирует действия противника с помощью метода машинного обучения, аналогичного рассмотренному в главе 9). Эти методы эффективны и интересны, но в основном лишь повышают эффективность поиска по дереву и минимаксных алгоритмов; поиск по дереву и минимаксные алгоритмы остаются основной рабочей силой стратегического искусственного интеллекта.

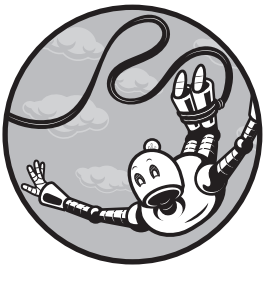
Резюме

Эта глава посвящена искусственному интеллекту. Вокруг этого термина много шумихи, но когда вы видите, что для написания функции `minimax()` требуется всего несколько десятков строк, ИИ перестает казаться чем-то таинственным и устрашающим. Но, конечно, чтобы вы могли написать эти строки, сначала необходимо изучить правила игры, нарисовать игровое поле, построить деревья игры и настроить функцию `minimax()` для правильного вычисления исхода игры.

В следующей главе приводятся рекомендации для целеустремленных алгоритмистов, которые захотят продолжить свое путешествие к границам мира алгоритмов и выйти к новым, неизведанным территориям.

11

Полный вперед



Вы преодолели темный лес поиска и сортировки, заледеневшую реку таинственных математических выкладок, коварные горные перевалы градиентного подъема, топи геометрической безнадежности и сразили дракона медленного выполнения. Примите мои поздравления.

В принципе, ничто не мешает вам вернуться в свой уютный дом на земле, свободной от алгоритмов. Эта глава написана для тех, кто захочет продолжить путешествие после того, как закроет книгу.

Ни в одной книге нельзя рассказать все об алгоритмах. О них известно слишком много, и мы постоянно узнаем что-то новое. В этой главе рассматриваются три основных вопроса: как сделать больше с помощью алгоритмов, как пользоваться ими лучше и быстрее и как справиться с их самыми сокровенными тайнами.

В этой главе мы построим простой чат-бот, который будет поддерживать беседу о предыдущих главах книги. Затем обсудим некоторые из самых сложных задач и пути к построению алгоритмов для их решения. В завершение рассмотрим некоторые из самых сложных задач мира алгоритмов, включая подробные инструкции о том, как выиграть миллион долларов с использованием продвинутой алгоритмической теории.

Как сделать больше с помощью алгоритмов

В предыдущих десяти главах книги рассматривались алгоритмы, которые могут решать разные задачи во многих областях. Но алгоритмы могут сделать еще больше, чем вы видели здесь. Если вы захотите продолжить свои странствия в мире алгоритмов, то займитесь исследованием других областей и важных алгоритмов, связанных с ними.

Например, многие алгоритмы сжатия информации могут сохранить объемную книгу в закодированной форме, которая занимает лишь малую часть от размера оригинала, и могут сжать фотографию или фильм в высоком разрешении до небольшого размера с минимальной потерей качества.

Возможность безопасной передачи данных в интернете, включая удобную передачу данных кредитных карт третьим сторонам, основана на криптографических алгоритмах. Криптография — чрезвычайно интересная область, которая связана с захватывающими историями авантюризма, шпионажа, предательства и торжествующих ученых, которые взламывали коды, чтобы победить в войне.

Недавно были разработаны инновационные алгоритмы для выполнения параллельных распределенных вычислений. Вместо того чтобы миллионы раз выполнять по одной операции, алгоритмы распределенных вычислений разбивают набор данных на множество небольших частей, а затем передают их разным компьютерам. Эти компьютеры одновременно выполняют требуемую операцию и возвращают результаты, которые собираются воедино и представляются как окончательный результат. Параллельная обработка всех частей данных (вместо последовательной обработки) обеспечивает колоссальную экономию времени. Это в высшей степени полезно в области машинного обучения, где возникает необходимость обработки слишком больших наборов данных или одновременного выполнения множества простых вычислений.

На протяжении последних десятилетий люди много говорят о потенциале квантовых вычислений. Квантовые компьютеры, если нам удастся заставить их правильно работать, обладают потенциалом для выполнения в высшей степени сложных вычислений (включая методы взлома самых надежных криптографических алгоритмов) за малую долю времени, которое потребуется для этого на современных (но не квантовых) суперкомпьютерах. Квантовые компьютеры строятся на базе архитектуры, отличной от архитектуры стандартных компьютеров, поэтому появляется возможность разработки новых алгоритмов, использующих физические свойства квантовых компьютеров для быстрого решения сложнейших задач. Пока это остается скорее теоретическим вопросом, поскольку квантовые компьютеры еще не достигли состояния, в котором могут использоваться для практических целей.

Но если эта технология когда-либо воплотится в жизнь, то квантовые алгоритмы начнут играть в высшей степени важную роль.

Когда вы будете изучать алгоритмы в этих и многих других областях, вам не придется начинать с нуля. Освоив алгоритмы из данной книги, вы станете понимать, что они собой представляют, как работают и как реализуются в программном коде. Вероятно, изучать первый алгоритм было достаточно трудно, но изучать 50-й или 200-й алгоритм будет гораздо проще, поскольку ваш мозг привыкнет к общим схемам построения алгоритмов и к тому, с какой точки зрения о них стоит думать.

Чтобы доказать, что вы уже можете понимать и программировать алгоритмы, мы исследуем алгоритмы, которые используются совместно для реализации функциональности чат-бота. Если по этому краткому введению вы сможете понять, как работают эти алгоритмы и как написать для них код, значит, вы прошли немалую часть пути к тому, чтобы разобраться в работе любого алгоритма в любой области.

Построение чат-бота

Построим простой чат-бот, который может отвечать на вопросы о содержании этой книги. Начнем с импортирования модулей, которые сыграют важную роль позднее:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy import spatial
import numpy as np
import nltk, string
```

Следующим шагом при создании чат-бота становится *нормализация текста* — процесс преобразования текста на естественном языке в стандартизированные подстроки; это позволяет легко сравнивать вроде бы разные тексты. Мы хотим, чтобы наш бот понимал, что слова *America* и *america* означают одно и то же, что *regeneration* выражает ту же идею, что и *regenerate* (хотя и другой частью речи), что *centuries* является множественным числом от *century*, а *hello;* по сути не отличается от *hello*. Мы хотим, чтобы чат-бот одинаково рассматривал слова, происходящие от одного корня, если только нет веских причин действовать иначе.

Допустим, имеется следующий запрос:

```
query = 'I want to learn about geometry algorithms.'
```

Первое, что необходимо сделать, — преобразовать все символы в нижний регистр. Задача решается встроенным методом Python `lower()`:

```
print(query.lower())
```

Команда выводит текст `i want to learn about geometry algorithms.`. Второй шаг — удаление знаков препинания. Для этого мы сначала создадим объект Python, называемый *словарем*:

```
remove_punctuation_map = dict((ord(char), None) for char in string.punctuation)
```

Этот фрагмент создает словарь, который связывает каждый стандартный знак препинания с объектом Python `None` и сохраняет словарь в переменной `remove_punctuation_map`. Словарь используется для удаления знаков препинания следующим образом:

```
print(query.lower().translate(remove_punctuation_map))
```

Здесь метод `translate()` находит все знаки препинания в запросе и не заменяет их ничем — то есть фактически удаляет их. Полученный вывод почти не отличается от того, что мы видели — `i want to learn about geometry algorithms`, в нем только нет завершающей точки. Затем проводится разбиение на лексемы, которое преобразует строку текста в список осмысленных подстрок:

```
print(nltk.word_tokenize(query.lower().translate(remove_punctuation_map)))
```

Для выполнения этой операции используется готовая функция из модуля `nltk`, которая выдает следующий результат:

```
['i', 'want', 'to', 'learn', 'about', 'geometry', 'algorithms']
```

Затем можно выполнить процедуру *выделения основы* (stemming). В английском языке слова *jump*, *jumps*, *jumping*, *jumped* и другие производные формы отличаются друг от друга, но обладают общим корнем *jump*. Наш чат-бот не должен отвлекаться на мелкие различия производных форм; предложения о *jumping* и *jumper* должны считаться приблизительно равными, хотя формально это разные слова. Выделение корня удаляет окончания производных слов, превращая их в стандартизированные корни. Функция выделения корней доступна в модуле Python `nltk` и может использоваться со списковым включением:

```
stemmer = nltk.stem.porter.PorterStemmer()
def stem_tokens(tokens):
    return [stemmer.stem(item) for item in tokens]
```

В этом фрагменте определяется функция `stem_tokens()`. Она получает список лексем и вызывает функцию `stemmer.stem()` модуля `nltk` для преобразования их в корни:

```
print(stem_tokens(nltk.word_tokenize(query.lower().translate(remove_punctuation_map))))
```

Результат вызова — `['i', 'want', 'to', 'learn', 'about', 'geometri', 'algorithm']`. Наша функция преобразовала *algorithms* в *algorithm*, а *geometry* — в *geometri*. Она заменила слово тем, что рассматривается как его корень: одно слово или часть слова, упрощающая сравнение текста. Наконец, все действия по нормализации объединяются в одну функцию `normalize()`:

```
def normalize(text):
    return stem_tokens(nltk.word_tokenize(text.lower().translate(remove_
                       punctuation_map)))
```

Векторизация текста

Перейдем к преобразованию текста в числовые векторы. Количественные сравнения проще выполнять с числами и векторами, чем со словами, а для работы нашего чат-бота придется выполнять количественные сравнения.

Мы воспользуемся простым методом *TFIDF* (Term Frequency-Inverse Document Frequency, «частотность термина — обратная частота документов») для преобразования документов в числовые векторы. Каждый вектор документа содержит один элемент для каждого термина в корпусе. Каждый элемент вычисляется как произведение частоты встречаемости термина (простое количество вхождений термина в конкретном документе) и обратной частоты документа для заданного термина (логарифм обратной пропорции документов, в которых присутствует термин).

Представьте, что вы создаете векторы TFIDF для биографий президентов США. В контексте создания векторов TFIDF каждая биография будет рассматриваться как документ. В биографии Авраама Линкольна слово *«представитель»*, вероятно, встретится хотя бы один раз, поскольку он служил в Палате представителей Иллинойса и в Палате представителей США. Если слово *«представитель»* встречается в биографии три раза, то считается, что для него частота термина равна 3. В Палате представителей США служили более дюжины президентов, поэтому, скорее всего, термин *«представитель»* будет встречаться приблизительно в 20 из 44 биографиях президентов. Тогда обратная частота документа вычисляется следующим образом:

$$\log\left(\frac{44}{20}\right) = 0{,}788.$$

Искомое значение равно произведению частоты термина и обратной частоты документов: 3 × 0,788 = 2,365. Теперь возьмем термин *«Геттисберг»*. Он может встретиться дважды в биографии Линкольна, но в других биографиях не встретится ни разу, поэтому частота термина будет равна 2, а обратная частота документов будет вычисляться по формуле:

$$\log\left(\frac{44}{1}\right) = 3{,}784.$$

Элемент вектора, связанный с термином *«Геттисберг»*, будет равен произведению частоты термина и обратной частоты документов, то есть 2 × 3,784 = 7,568. Значение TFIDF для каждого термина должно отражать его важность в документе. Вскоре этот показатель сыграет важную роль в способности нашего чат-бота определять намерения пользователя. Вычислять TFIDF вручную необязательно, можно воспользоваться готовой функцией модуля `scikit-learn`:

```
vctrz = TfidfVectorizer(ngram_range = (1, 1),tokenizer = normalize, stop_words =
                        'english')
```

В этой строке используется функция `TfidfVectorizer()`, создающая векторы TFIDF для наборов документов. Чтобы создать векторизатор, необходимо задать значение `ngram_range`. Оно сообщает векторизатору, что должно считаться термином. Мы указали значение `(1, 1)`, которое означает, что векторизатор будет считать терминами только 1-граммы (отдельные слова). Если бы мы указали значение `(1, 3)`, то 1-граммы (отдельные слова), 2-граммы (фразы из двух слов) и 3-граммы (фразы из трех слов) считались бы терминами и для каждого из них был бы создан элемент TFIDF. Кроме того, передается аргумент `tokenizer`, которому присваивается функция `normalize()`, созданная ранее.

Наконец, необходимо передать аргумент `stop_words` со списком стоп-слов, которые должны быть отфильтрованы, поскольку не являются информативными. В английском языке к стоп-словам относятся *the*, *and*, *of* и другие чрезвычайно распространенные слова. Задавая значение `stop_words = 'english'`, мы приказываем векторизатору отфильтровать встроенный список английских стоп-слов и применять векторизацию только к менее распространенным и более информативным словам.

Теперь подготовим темы, на которые сможет беседовать наш чат-бот. Здесь речь пойдет о главах нашей книги, поэтому мы создадим список с очень простыми описаниями каждой главы. В данном контексте каждая строка представляет один из наших *документов*.

```
alldocuments = ['Chapter 1. The algorithmic approach to problem solving, including
                 Galileo and baseball.',
                'Chapter 2. Algorithms in history, including magic squares, Russian
                 peasant multiplication, and Egyptian methods.',
                'Chapter 3. Optimization, including maximization, minimization,
                 and the gradient ascent algorithm.',
                'Chapter 4. Sorting and searching, including merge sort, and
                 algorithm runtime.',
                'Chapter 5. Pure math, including algorithms for continued fractions
                 and random numbers and other mathematical ideas.',
                'Chapter 6. More advanced optimization, including simulated
                 annealing and how to use it to solve the traveling salesman
                 problem.',
                'Chapter 7. Geometry, the postmaster problem, and Voronoi
                 triangulations.',
                'Chapter 8. Language, including how to insert spaces and predict
                 phrase completions.',
                'Chapter 9. Machine learning, focused on decision trees and how
                 to predict happiness and heart attacks.',
                'Chapter 10. Artificial intelligence, and using the minimax
                 algorithm to win at dots and boxes.',
                'Chapter 11. Where to go and what to study next, and how to build
                 a chatbot.']
```

Применим векторизатор TFIDF к этим описаниям глав в целях обработки документов, чтобы мы могли создать векторы TFIDF в любой нужный момент. Делать это вручную необязательно, так как в модуле `scikit-learn` определен метод `fit()`:

```
vctrz.fit(alldocuments)
```

Теперь создадим векторы TFIDF для описаний глав и для нового запроса главы, посвященной сортировке и поиску:

```
query = 'I want to read about how to search for items.'
tfidf_reports = vctrz.transform(alldocuments).todense()
tfidf_question = vctrz.transform([query]).todense()
```

Новый запрос содержит текст на английском языке. В первых двух строках встроенные методы `translate()` и `todense()` создают векторы TFIDF для описаний глав и запроса.

Таким образом, описания глав и запрос были преобразованы в числовую форму. Работа нашего простого чат-бота основана на сравнении вектора TFIDF запроса с векторами описаний глав TFIDF. Предполагается, что пользователю нужно порекомендовать главу, вектор описания которой обладает наибольшим сходством с вектором запроса.

Сходство векторов

Для определения сходства между двумя векторами мы будем использовать метод *косинусного сходства*. Если вы изучали геометрию, то знаете, что для любых двух числовых векторов можно вычислить угол между ними. Принципы геометрии позволяют вычислять углы между векторами не только в двух или трех измерениях, но и в четырех или пяти либо в пространстве с произвольным количеством измерений. При большом сходстве векторов угол между ними будет достаточно малым. Если же они сильно различаются, то угол будет большим. Идея о том, что тексты на английском языке можно сравнивать вычислением «угла» между ними, звучит странно, но именно поэтому мы создали свои числовые векторы TFIDF — чтобы можно было применять такие числовые средства, как сравнение углов, для данных, которые не находились в числовой форме.

На практике бывает проще вычислить косинус угла между двумя векторами, чем сам угол. Это не создает проблем, поскольку при большом косинусе угла между двумя векторами сам угол мал, и наоборот. В языке Python модуль `scipy` содержит подмодуль `spatial`, который содержит функцию для вычисления косинусов углов между векторами. Функциональность `spatial` позволяет вычислить косинусы между векторами описаний каждой главы и запроса с помощью спискового включения:

```
row_similarities = [1 - spatial.distance.cosine(tfidf_reports[x],tfidf_question) \
for x in range(len(tfidf_reports)) ]
```

При выводе переменной `row_similarities` будет получен следующий результат:

```
[0.0, 0.0, 0.0, 0.3393118510377361, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

В данном случае только четвертый элемент больше нуля; это означает, что только вектор описания четвертой главы находится в какой-то угловой близости к нашему вектору запроса. В общем случае можно автоматически определить, какая строка обладает наибольшей косинусной близостью:

```
print(alldocuments[np.argmax(row_similarities)])
```

Так мы узнаем главу, которую, по мнению чат-бота, ищет пользователь: это глава 4. В листинге 11.1 простая функциональность чат-бота объединена в функцию.

Листинг 11.1. Простой чат-бот получает запрос и возвращает документ, обладающий наибольшим сходством с запросом

```
def chatbot(query,allreports):
    clf = TfidfVectorizer(ngram_range = (1, 1),tokenizer = normalize, stop_words =
          'english')
```

```
    clf.fit(allreports)
    tfidf_reports = clf.transform(allreports).todense()
    tfidf_question = clf.transform([query]).todense()
    row_similarities = [1 - spatial.distance.cosine(tfidf_reports[x],
                        tfidf_question) for x in \ range(len(tfidf_reports)) ]
    return(allreports[np.argmax(row_similarities)])
```

В листинге 11.1 нет ничего нового; весь его код уже встречался нам ранее. Теперь можно вызвать чат-бот с запросом о том, в какой главе можно найти некую информацию:

```
print(chatbot('Please tell me which chapter I can go to if I want to read about
mathematics algorithms.',alldocuments))
```

Вывод предлагает обратиться к главе 5:

```
Chapter 5. Pure math, including algorithms for continued fractions and random
numbers and other
mathematical ideas.
```

Итак, вы видели, как работает чат-бот, и понимаете, почему необходимо выполнить нормализацию и векторизацию. Благодаря нормализации и выделению корней термин *mathematics* указывает боту вернуть описание главы 5, хотя точная форма слова в нем не встречается. Векторизация делает возможным применение метрики косинусного сходства, которая сообщает, описание какой главы обеспечивает наилучшее совпадение.

Мы завершили работу над чат-ботом, для чего потребовалось связать воедино несколько меньших алгоритмов (алгоритмы нормализации, выделения корней и числовой векторизации текста; алгоритм вычисления косинусов углов между векторами; объединяющий алгоритм формирования ответов чат-бота на основании сходства векторов запроса/документа). Возможно, вы заметили, что нам не пришлось выполнять вычисления вручную — все вычисления TFIFD и косинусов выполнялись импортированными модулями. На практике вам часто не требуется понимать внутренние механизмы алгоритма, чтобы импортировать его и использовать в своих программах. Это может быть благом, поскольку такой подход ускоряет работу и предоставляет в наше распоряжение невероятно эффективные инструменты в тот момент, когда это потребуется. С другой стороны, это может быть и проклятием, так как люди могут неправильно использовать алгоритмы, суть которых они не понимают. Например, статья в журнале Wired утверждает, что некорректное применение финансового алгоритма (метод использования гауссовской копулы для прогнозирования рисков) привело к «крушению на Уолл-Стрит» и «потерям триллионов долларов», а также стало главной причиной Великой рецессии (https://

www.wired.com/2009/02/wp-quant/). Я рекомендую изучить теорию алгоритмов даже при том, что из-за простоты импортирования модулей в Python такое изучение может показаться излишним; знание теории всегда сделает вас более квалифицированным теоретиком или практиком.

Конечно, это всего лишь простейший чат-бот, и он отвечает только на вопросы, относящиеся к главам нашей книги. Для него можно предложить множество потенциальных улучшений: сделать описания глав более конкретными, чтобы они соответствовали более широкому диапазону запросов; найти метод векторизации, превосходящий TFIDF по эффективности; добавить другие документы, чтобы чат-бот отвечал на большее количество запросов. Пусть он получился не самым мощным, но им все равно можно гордиться, поскольку мы построили его самостоятельно. Если вы способны без особого труда построить чат-бот, то можете по праву считать себя компетентным разработчиком и создателем реализаций алгоритмов. Поздравляю, это достойное завершение вашего путешествия по этой книге!

Лучше и быстрее

Алгоритмы позволят вам сделать больше, чем вы могли сделать до чтения этой книги. Но каждый серьезный разработчик также стремится к тому, чтобы его задачи решались быстрее и лучше.

Если вы хотите научиться лучше разрабатывать и реализовывать алгоритмы, есть много вариантов. Задумайтесь над тем, что каждый алгоритм, реализованный в книге, базируется на понимании неалгоритмической темы. Наш алгоритм ловли мяча требует понимания физики и даже до определенной степени психологии. Крестьянское умножение основано на понимании степеней и нетривиальных свойствах арифметических операций, включая двоичную запись. Геометрические алгоритмы главы 7 основаны на понимании связей и взаимодействий между точками, линиями и треугольниками. Чем глубже вы понимаете область, для которой пытаетесь писать алгоритмы, тем проще вам будет разрабатывать и реализовывать алгоритмы. Таким образом, стать сильнее в алгоритмах несложно: для этого достаточно идеально разбираться во всех областях.

Следующий естественный шаг для будущего специалиста по алгоритмам — отработка и доведение до совершенства навыков программирования. Напомню, что в главе 8 были представлены списковые включения — средство Python для компактной записи и эффективного выполнения языковых алгоритмов. По мере того как вы будете изучать новые языки программирования и осваивать их функциональность, вы научитесь писать лучше организованный, более компактный

и более эффективный код. Даже опытному программисту бывает полезно вернуться к основам и повторять их, пока они не станут вашей второй натурой. Многие талантливые программисты пишут плохо организованный, плохо документированный или неэффективный код, полагая, что «и так сойдет», поскольку код «работает». Но помните: код обычно не бывает успешным сам по себе — он почти всегда является частью более крупной программы, какой-либо коллективной работы или серьезного бизнес-проекта, основанного на сотрудничестве между людьми во времени. Из-за этого даже такие коммуникативные навыки, как планирование, устное и письменное общение, переговоры и коллективное управление, повышают ваш шанс на успех в мире алгоритмов.

Если вам нравится создавать оптимальные алгоритмы и доводить их до предела эффективности, то вам повезло. Для очень многих задач из области вычислительной теории не существует эффективных алгоритмов, работающих намного быстрее перебора методом грубой силы. В следующем разделе мы рассмотрим некоторые из этих задач и обсудим, чем же они так сложны. Если вы, дорогой читатель, создадите алгоритм, который сможет быстро решить какую-либо из этих задач, то вас ждет слава, богатство и всемирное признание на протяжении всей жизни.

Алгоритмы для смелых

Рассмотрим относительно простую задачу, связанную с шахматами. В шахматы играют на доске 8×8, два игрока ходят по очереди, перемещая разные фигуры. Одна фигура — ферзь — может перемещаться на любое количество полей по вертикали, горизонтали или диагонали относительно занимаемого поля. Обычно у игрока только один ферзь, но в стандартной шахматной партии может быть и до девяти ферзей. Если у игрока несколько ферзей, то может оказаться, что два и более ферзя находятся под боем друг друга — иначе говоря, находятся на одной вертикали, горизонтали или диагонали. В *задаче о восьми ферзях* требуется расставить восемь ферзей на стандартной шахматной доске так, чтобы никакие два ферзя не находились на одной вертикали, горизонтали или диагонали. На рис. 11.1 показано одно из решений этой задачи.

Ни один ферзь на этом поле не находится под боем с другим ферзем. Самый простой способ решить задачу о восьми ферзях — просто запомнить решение (например, показанное на рис. 11.1) и приводить его каждый раз, когда потребуется решить задачу. Однако у задачи есть пара модификаций, из-за которых запоминание оказывается неприемлемым. Первая модификация — увеличение количества ферзей и размера доски. В *задаче о n ферзях* требуется разместить n ферзей на шахматной доске $n \times n$ так, чтобы ни один ферзь не находился под боем; n может быть любым натуральным числом, сколь угодно большим. Вторая модификация — *задача*

о завершении позиции с n ферзями; ваш противник сначала расставляет несколько ферзей (возможно, в местах, из-за которых вам будет трудно расставить остальные фигуры), и вы должны разместить остальных ферзей до *n*, чтобы ни один не находился под боем других. Получится ли у вас разработать алгоритм для решения этой задачи, который будет выполняться очень быстро? Если получится, то вы можете заработать миллион долларов (см. раздел «Решение самых сложных задач», с. 268).

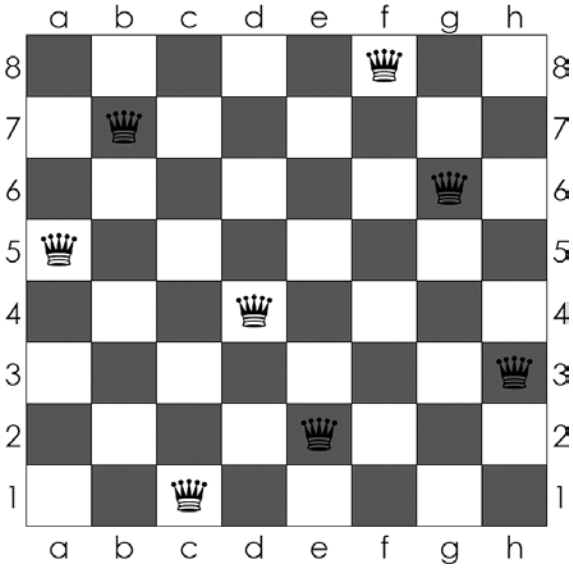


Рис. 11.1. Решение задачи о восьми ферзях (источник: Wikimedia Commons)

Рисунок 11.1 может напоминать головоломку судоку, поскольку в нем требуется проверять уникальность обозначений в строках и столбцах. В судоку требуется заполнить сетку цифрами от 1 до 9 так, чтобы в каждой строке, столбце и блоке 3 × 3 каждая цифра встречалась ровно один раз (рис. 11.2). Головоломка изначально завоевала популярность в Японии, причем отчасти напоминает японские магические квадраты, рассмотренные в главе 2.

Интересно задуматься над тем, как написать алгоритм для решения судоку. Самый простой и самый медленный алгоритм основан на переборе методом грубой силы: вы просто перебираете все возможные комбинации цифр и проверяете, образуют ли они правильное решение, пока ответ не будет найден. Такое решение работает, но ему не хватает элегантности, а на решение может потребоваться очень много времени. Интуитивно понятно, что если размещение 81 цифры на сетке по правилам, которые легко понятны любому, выходит за пределы вычислительных возможностей нашего

мира, то здесь что-то не так. Более сложные решения должны использовать логику для сокращения затрат времени.

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

Рис. 11.2. Незаполненная сетка судоку (источник: Wikimedia Commons)

У задачи о *n* ферзях и судоку есть одна важная общая особенность: решения очень легко проверяются. Если я покажу вам шахматную доску с ферзями, то, вероятно, вы за считаные секунды определите, является ли представленная позиция решением задачи о *n* ферзях. А если вам покажут сетку с 81 цифрой, то вы легко определите, является ли она правильным решением судоку. Как ни печально, простота проверки решений не соответствует простоте генерирования решений — на решение сложной головоломки судоку может потребоваться несколько часов, а полученное решение проверяется за секунды. Несоответствие затрат за генерирование/проверку встречается во многих областях жизни: я могу с минимальными усилиями сказать, хорошо ли приготовлено блюдо, однако на создание хорошего блюда уйдет намного больше времени и ресурсов. Или на то, чтобы понять, что картина нарисована красиво, мне потребуется гораздо меньше времени, чем на рисование красивой картины; или убедиться в том, что самолет может летать, я смогу гораздо быстрее, чем построить самолет, который может летать...

Проблемы, которые трудно решать алгоритмически, но решения которых легко проверяются, играют в высшей степени важную роль в информатике. Пожалуй, это самая глубокая и самая насущная тайна в данной сфере. Самые бесстрашные искатели приключений могут попробовать ее разгадать, но должны остерегаться опасностей, которые их поджидают.

Решение самых сложных задач

Когда мы говорим, что решения судоку легко проверяются, но долго генерируются, в более формальном выражении это означает, что решения могут проверяться за *полиномиальное время*; иначе говоря, количество шагов, необходимых для проверки решения, является некоторой полиномиальной функцией от размера игрового поля судоку. Если вы вернетесь к главе 4 и обсуждению времени выполнения, то вспомните, что хотя полиномиальные функции — такие, как x^2 и x^3, — растут достаточно быстро, они достаточно медленны по сравнению с экспоненциальными функциями вроде e^x. Если алгоритмическое решение задачи можно проверить за полиномиальное время, то такая проверка считается простой, но если генерирование решения занимает экспоненциальное время, то оно считается сложным.

Для класса задач, решения которых могут быть проверены за полиномиальное время, существует формальное назначение: *класс сложности NP*. (NP — сокращение от Nondeterministic Polynomial time, то есть «недетерминированное полиномиальное время»; чтобы объяснить происхождение термина, нам пришлось бы надолго отвлечься на вычислительную теорию, что здесь было бы лишним.) NP — один из двух фундаментальных классов сложности в вычислительной теории. Второй класс называется P (от Polynomial time, то есть «полиномиальное время»). К классу сложности P относятся все задачи, решения которых могут быть найдены с помощью алгоритма, выполняемого за полиномиальное время. Для задач класса P полные решения могут быть *найдены* за полиномиальное время, тогда как для задач класса NP решения *проверяются* за полиномиальное время, но поиск этих решений может потребовать экспоненциального времени.

Мы знаем, что судоку относится к задачам класса NP — предложенное решение легко проверяется за полиномиальное время. Является ли судоку также задачей класса P? Иначе говоря, существует ли алгоритм, способный решить любую задачу судоку за полиномиальное время? Такой алгоритм еще не нашли, и, похоже, никто в ближайшее время не найдет, но мы не можем с уверенностью утверждать, что это невозможно.

Список задач, которые, как мы знаем, принадлежат классу NP, в высшей степени велик. Некоторые разновидности задачи коммивояжера относятся к классу NP. К нему же относится оптимальное решение кубика Рубика и такие важные математические задачи, как целочисленное линейное программирование. Как в случае с судоку, возникает вопрос, принадлежат ли эти задачи также классу P — можно ли найти их решения за полиномиальное время? Или в одной из возможных формулировок этого вопроса — совпадают ли эти классы, или P = NP?

В 2000 году математический институт Клэя опубликовал список, названный «Задачи тысячелетия». Было объявлено, что каждый, кто опубликует подтвержденное

решение одной из этих задач, получит миллион долларов. В список были включены семь важнейших задач из области математики, и вопрос об истинности P = NP был одной из них; никто еще не получил премию. Возможно, кто-то из отважных читателей со временем разрубит гордиев узел и решит эту важнейшую алгоритмическую задачу? Я на это искренне надеюсь и желаю каждому из вас удачи, силы и позитива в данном путешествии.

Если решение когда-либо будет найдено, то оно докажет одно из двух утверждений: либо P = NP, либо P ≠ NP. Доказательство того, что P = NP, может быть относительно простым, так как все, что для этого потребуется, — алгоритмическое решение с полиномиальным временем для NP-полной задачи. *NP-полные задачи* составляют особую разновидность задач NP, определяемую одним свойством: каждая задача NP может быть быстро сведена к NP-полной задаче; другими словами, если вы можете решить одну NP-полную задачу, то сможете решить все задачи класса NP. Если любая отдельная NP-полная задача может быть решена за полиномиальное время, то любая задача класса NP может быть решена за полиномиальное время, а это докажет, что P = NP. Оказывается, судоку и задача о завершении позиции с *n* ферзями также являются NP-полными. Это означает, что нахождение алгоритмического решения с полиномиальным временем любой из этих задач не только позволит решить каждую существующую задачу NP, но и принесет вам миллион долларов и всемирную славу (не говоря уже о том, что вы сможете победить любого в соревнованиях по решению головоломок судоку).

Доказательство того, что P ≠ NP, вероятно, не будет таким прямолинейным, как решение судоку. Утверждение P ≠ NP означает, то существуют задачи класса NP, которые не могут быть решены никаким алгоритмом с полиномиальным временем. По сути, речь идет о доказательстве негативного утверждения, а на концептуальном уровне доказать несуществование чего-то намного сложнее, чем привести пример. Чтобы продвинуться в доказательстве того, что P ≠ NP, необходимы расширенные исследования в вычислительной теории, выходящие за рамки книги. И хотя этот путь сложнее, теоретики сходятся на том, что P ≠ NP, и если вопрос о тождественности P и NP когда-нибудь будет разрешен, то, скорее всего, будет доказано, что P ≠ NP.

Вопрос о тождественности P и NP является не единственной глубокой тайной, связанной с алгоритмами, хотя и принесет наибольшую немедленную пользу. В каждом аспекте разработки алгоритмов остаются широкие поля областей, которые могут стать темой для исследований. Помимо вопросов чисто теоретических и академических, существуют и практические вопросы, относящиеся к реализации алгоритмически содержательных практик в контексте бизнеса. Не теряйте времени даром; не забывайте то, что узнали в книге, и двигайтесь вперед, вооружившись приобретенными знаниями, до самых дальних границ знаний и практики. Друзья, до встречи!

Брэдфорд Такфилд

Алгоритмы неформально. Инструкция для начинающих питонистов

Перевел с английского *Е. Матвеев*

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Сидорова, Г. Шкатова*
Верстка	*Л. Егорова*

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

Дата изготовления: 08.2022. 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 10.06.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 1200. Заказ 0000.

А. Бхаргава

ГРОКАЕМ АЛГОРИТМЫ. ИЛЛЮСТРИРОВАННОЕ ПОСОБИЕ ДЛЯ ПРОГРАММИСТОВ И ЛЮБОПЫТСТВУЮЩИХ

Алгоритмы — это всего лишь пошаговые инструкции решения задач, и большинство таких задач уже были кем-то решены, протестированы и проверены. Можно, конечно, погрузиться в глубокую философию гениального Кнута, изучить многостраничные фолианты с доказательствами и обоснованиями, но хотите ли вы тратить на это свое время? Откройте великолепно иллюстрированную книгу и вы сразу поймете, что алгоритмы — это просто. А грокать алгоритмы — веселое и увлекательное занятие.

Эрик Мэтиз

ИЗУЧАЕМ PYTHON: ПРОГРАММИРОВАНИЕ ИГР, ВИЗУАЛИЗАЦИЯ ДАННЫХ, ВЕБ-ПРИЛОЖЕНИЯ

3-е издание

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете не только максимально быстро его освоить, но и научитесь писать программы, устранять ошибки и создавать работающие приложения.

В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения приучат вас к шаблонам чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.