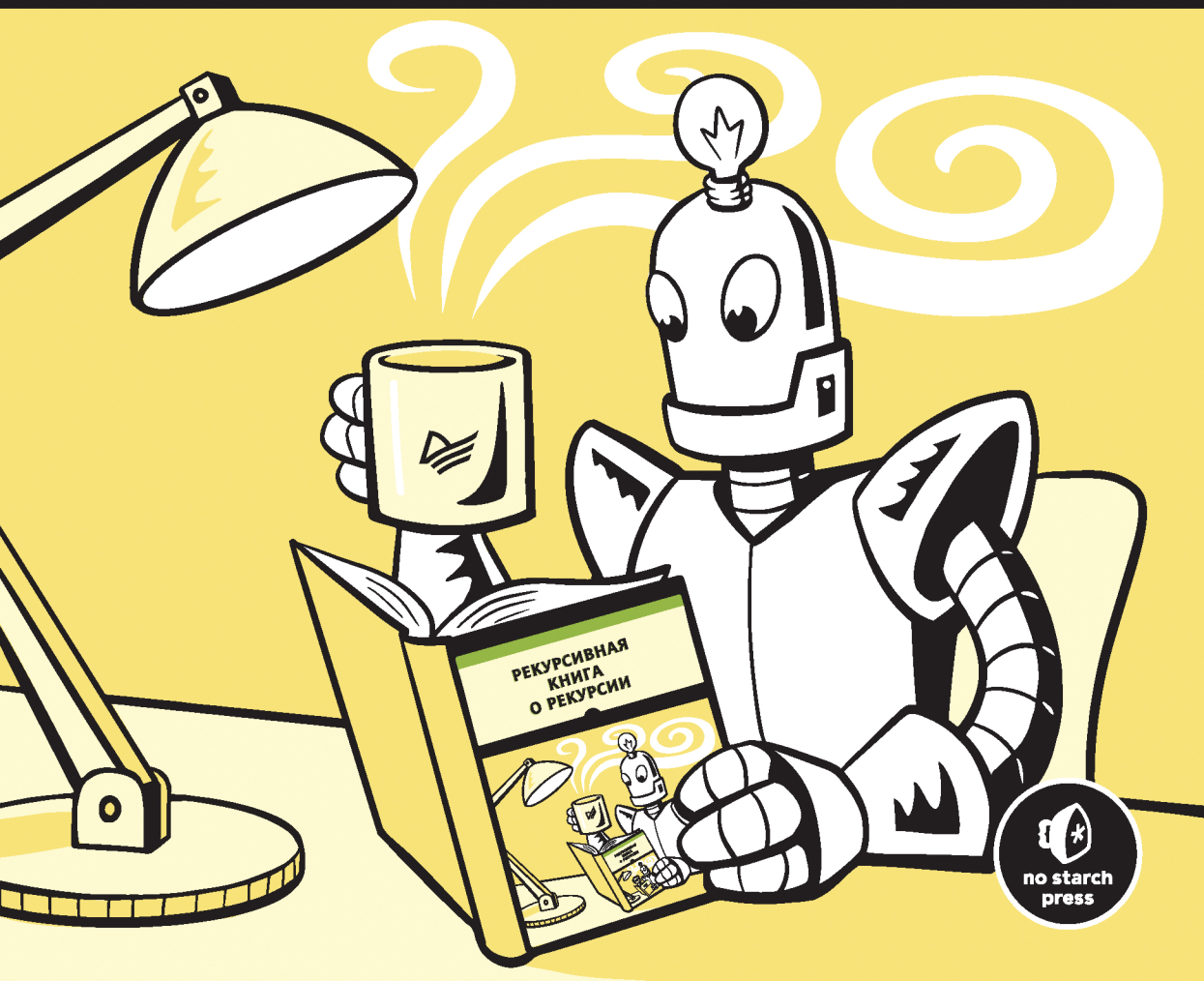
РЕКУРСИВНАЯ КНИГА О РЕКУРСИИ

ЭЛ СВЕЙГАРТ



THE RECURSIVE BOOK OF RECURSION

Ace the Coding Interview with Python and JavaScript

by Al Sweigart

no starch press

San Francisco

РЕКУРСИВНАЯ КНИГА О РЕКУРСИИ

ЭЛ СВЕЙГАРТ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.972.2-018
УДК 004.021
С24

Свейгарт Эл

С24 Рекурсивная книга о рекурсии. — СПб.: Питер, 2023. — 336 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2393-3

Книга «Рекурсивная книга о рекурсии» содержит примеры кода на языке Python и JavaScript, которые иллюстрируют основы рекурсии и проясняют фундаментальные принципы всех рекурсивных алгоритмов. Из книги вы узнаете о том, когда стоит использовать рекурсивные функции (и, главное, когда этого не нужно делать), как реализовывать классические рекурсивные алгоритмы, часто обсуждаемые на собеседованиях, а также о том, как рекурсивные методы помогают решать задачи, связанные с обходом дерева, комбинаторикой и другими сложными темами.

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

ББК 32.972.2-018
УДК 004.021

ISBN 978-1718502024 англ.

ISBN 978-5-4461-2393-3

Краткое содержание

Оглавление

ЧАСТЬ II. ПРОЕКТЫ

Посвящается Джеку, который держал
свое зеркало напротив моего.

Об авторе

Эл Свейгарт (Al Sweigart) — разработчик программного обеспечения, член организации Python Software Foundation и автор нескольких книг по программированию, в том числе «Python. Чистый код для продолжающих» и «Большая книга проектов Python» (издательство «Питер»). Его работы, распространяемые по лицензии Creative Commons, доступны по адресу https://www.inventwithpython.com.

О научном редакторе

Сара Кучински (Sarah Kuchinsky) имеет ученые степени в области менеджмента, инженерии и математики. Проводит корпоративные тренинги, а также занимается моделированием систем здравоохранения, разработкой игр и автоматизацией различных задач, используя для этого Python. Сара является соучредителем конференции North Bay Python, председателем комиссии на конференции PyCon US и ведущим организатором сообщества PyLadies Silicon Valley.

Предисловие

Когда Эл предложил мне написать предисловие к его книге, я буквально загорелся этой идеей. Книга о рекурсии! Не каждый день выпадает такая возможность. Многие считают рекурсию одной из самых сложных и непонятных тем в программировании и нередко обходят ее стороной. И это очень странно, учитывая, насколько часто о ней спрашивают на собеседованиях.

Так или иначе, для изучения рекурсии есть множество практических причин. Рекурсивное мышление напрямую определяет подход к решению задач, когда большие и сложные задачи разбиваются на более мелкие и простые. Такой образ мышления полезен при разработке программного обеспечения даже в тех ситуациях, когда рекурсия не используется. Именно поэтому в рекурсии стоит разобраться всем программистам вне зависимости от их уровня квалификации.

Желая рассказать о рекурсии как можно больше, я изначально написал это предисловие в виде нескольких коротких рассказов о моих друзьях, которые по-разному применяли техники рекурсивного мышления и при этом достигли схожих результатов. Первой я хотел рассказать историю Бена, который, узнав о рекурсии, зашел слишком далеко и каким-то образом умудрился провалиться сквозь землю при загадочных обстоятельствах после того, как запустил следующий код на языке Python:

```
result = [(lambda r: lambda n: 1 if n < 2 else r(r)(n-1) + r(r)(n-2))(
          (lambda r: lambda n: 1 if n < 2 else r(r)(n-1) + r(r)(n-2)))(n)
          for n in range(37)]
```

Затем я собирался поделиться историей Челси, которая добилась такой эффективности в решении реальных проблем, что ее уволили! О, вы не поверите, как не понравились эти истории редакторам издательства No Starch (благослови их Господь). «Нельзя начинать книгу с подобных рассказов. Они распугают всех читателей!» Думаю, в чем-то они правы. Они даже заставили меня переместить более оптимистичный абзац о рекурсии из конца предисловия в начало, чтобы

после прочтения леденящих душу историй о Бене и Челси вы не убежали читать книгу о шаблонах проектирования.

Написание предисловия к книге — дело серьезное, так что, к сожалению, мне придется рассказать реальные истории Бена и Челси в другой раз. Возвращаясь к теме книги, следует отметить, что рекурсия — это не тот метод, который используется при решении множества задач в программировании. Она часто овеяна ореолом таинственности, развеять который и призвана данная книга.

Наконец, отправляясь в свое рекурсивное путешествие, будьте готовы к тому, что ваш мозг вскипит: не беспокойтесь — это нормально! Важно подчеркнуть, что изучение рекурсии должно доставлять удовольствие. Хотя бы чуть-чуть. Так что в добрый путь!

Дэвид Бизли,
автор книги «Python. Исчерпывающее руководство»[1]*,*
https://www.dabeaz.com

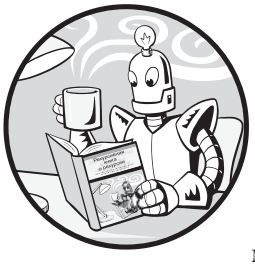
[1] *Бизли Д.* Python. Исчерпывающее руководство. — Питер, 2022.

Благодарности

Хотя на обложке указано только мое имя, было бы несправедливо не отметить тех, кто помогал мне в написании книги. Я хотел бы поблагодарить моего издателя Билла Поллока (Bill Pollock), редактора Фрэнсис Со (Frances Saux), научного редактора Сару Кучински (Sarah Kuchinsky), выпускающего редактора Майлза Бонда (Miles Bond), менеджера по производству Рэйчел Монаган (Rachel Monaghan) и всех остальных сотрудников издательства No Starch Press за их неоценимую помощь.

Наконец, хочу выразить признательность своей семье, друзьям и читателям за ценные предложения и поддержку.

Введение



С помощью такой техники программирования, как рекурсия, можно создавать элегантные кодовые решения. Однако иногда разработчики не понимают, с какой стороны к ней подойти. Это не означает, что программисты могут (или должны) игнорировать рекурсию. Несмотря на свою кажущуюся сложность, рекурсия является важной темой в информатике и позволяет глубже вникнуть в сам процесс программирования. По крайней мере, если вы разбираетесь в ней, вам будет проще пройти собеседование при приеме на работу.

Если вы студент, интересующийся компьютерными науками, рекурсия может стать необходимым препятствием, которое вам придется преодолеть, чтобы разобраться во многих популярных алгоритмах. Если вы программист-самоучка, окончивший курсы по программированию или старающийся обходить теоретические основы информатики стороной, то вы все равно столкнетесь с рекурсией, например, на собеседовании. А если вы уже опытный разработчик, который никогда раньше не использовал рекурсивные алгоритмы, то можете счесть это досадным пробелом в своих знаниях.

Беспокоиться не о чем. Рекурсию гораздо сложнее объяснить, чем понять. Как будет сказано в главе 1, многие не разбираются в рекурсии не потому, что это сложная тема, а потому, что ее плохо объясняют. А поскольку рекурсивные функции обычно не используются в повседневной работе, многие программисты прекрасно обходятся без них.

Между прочим, рекурсия лежит в основе удивительного математического искусства создания *фракталов* — самоподобных фигур, примеры которых представлены на рис. В.1.

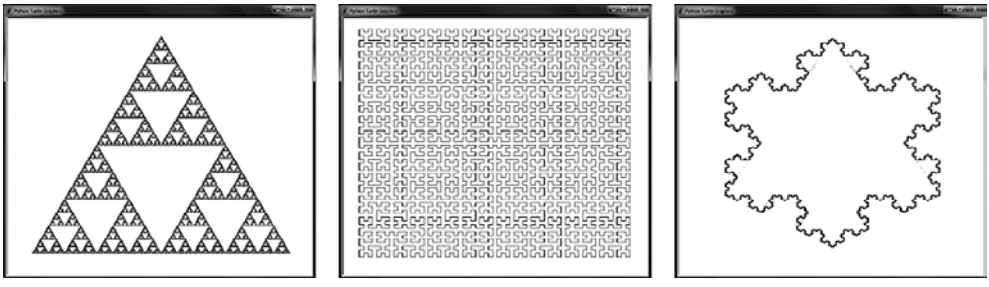


Рис. В.1. Примеры фракталов: треугольник Серпинского *(слева)*, кривая Гильберта *(в центре)* и снежинка Коха *(справа)*

Тем не менее книга посвящена не только восхвалению рекурсии. В ней вы найдете и резкие критические замечания по поводу этой техники. Рекурсией нередко злоупотребляют в тех случаях, когда можно применить более простое решение. Рекурсивные алгоритмы бывают сложными для понимания и отличаются худшей производительностью, а также подвержены ошибкам переполнения стека. Некоторым программистам нравится использовать рекурсию не потому, что это подходящий метод решения конкретной проблемы, а просто потому, что они чувствуют себя умнее, когда пишут код, который другие разработчики понимают с трудом. Доктор технических наук Джон Виландер (John Wilander) однажды сказал: «Когда вы получаете докторскую степень в области computer science, вас отводят в специальную комнату и просят никогда не использовать рекурсию в реальной жизни, поскольку ее главная цель — усложнить жизнь старшекурсникам».

Итак, хотите ли вы получить преимущество на собеседованиях, научиться создавать красивые произведения математического искусства или просто пытаетесь окончательно разобраться в теме — эта книга покажет, насколько глубока кроличья нора (и кроличьи норы внутри нее). Рекурсия — это одна из тех тем информатики, по знанию которой можно профессионала отличить от новичка. Прочитав эту книгу, вы овладеете великим мастерством и узнаете главный секрет рекурсии: она вовсе не так сложна, как многие думают.

Для кого эта книга

Книга предназначена для тех, кого пугают или завораживают рекурсивные алгоритмы. Для большинства новичков понимание рекурсии сродни искусству владения черной магией. В примерах рекурсии далеко не всегда легко разобраться, из-за чего сам предмет вызывает недоумение и даже страх. Я надеюсь, что четкие объяснения и многочисленные примеры, приведенные в книге, помогут читателям наконец освоить эту тему.

Единственным минимальным условием для изучения книги является наличие базового опыта программирования на языке Python или JavaScript, на которых написан код в листингах. Код в книге сведен к самой сути: если вы умеете вызывать и создавать функции, а также различать глобальные и локальные переменные — вы знаете достаточно, чтобы разобраться в этих примерах.

Об этой книге

Книга состоит из 14 глав.

Часть I. О рекурсии

- Глава 1 «Что такое рекурсия?». Здесь объясняется само понятие рекурсии и приводятся доказательства, что в ней нет ничего мистического.
- Глава 2 «Рекурсия и итерация». Эта глава посвящена различиям (и сходствам) между рекурсивными и итеративными методами.
- Глава 3 «Классические рекурсивные алгоритмы». Здесь рассматриваются известные рекурсивные программы, такие как «Ханойская башня», алгоритм заливки и др.
- Глава 4 «Алгоритмы поиска с возвратом и обхода дерева». В главе обсуждается проблема, для решения которой рекурсия подходит особенно хорошо: обход древовидных структур данных, например, при прохождении лабиринтов и навигации по каталогу.
- Глава 5 «Алгоритмы типа “разделяй и властвуй”». Здесь обсуждается применение рекурсии для разделения больших задач на более мелкие подзадачи, а также разбирается несколько распространенных алгоритмов типа «разделяй и властвуй».
- Глава 6 «Перестановки и комбинации». Эта глава охватывает рекурсивные алгоритмы, связанные с упорядочением и сопоставлением, а также общие задачи программирования, для решения которых эти алгоритмы применяются.
- Глава 7 «Мемоизация и динамическое программирование». В ней объясняются некоторые простые приемы повышения эффективности кода при применении рекурсии в реальном мире.
- Глава 8 «Оптимизация хвостовых вызовов». Глава посвящена оптимизации хвостовых вызовов, которые часто используются для повышения производительности рекурсивных алгоритмов.
- Глава 9 «Рисование фракталов». Здесь мы познакомимся с захватывающим способом создания произведений искусства с помощью рекурсивных алгоритмов. Для генерации изображений в этой главе используется так называемая черепашья графика.

Часть II. Проекты

- Глава 10 «Инструмент для поиска файлов». В этой главе описывается программа, которая выполняет поиск файлов на компьютере в соответствии с заданными параметрами.
- Глава 11 «Генератор лабиринтов». В главе представлена программа, которая автоматически генерирует лабиринты любого размера, используя рекурсивный алгоритм поиска с возвратом.
- Глава 12 «Решатель "пятнашек"». Эта глава посвящена созданию программы для решения головоломки под названием «пятнашки».
- Глава 13 «Генератор фракталов». Здесь описана программа, позволяющая создавать фрактальные рисунки в соответствии с вашим собственным дизайном.
- Глава 14 «Создание эффекта Дросте». В главе мы напишем программу, которая создает рекурсивные изображения типа «картинка в картинке» с использованием модуля обработки изображений Pillow.

Практическая экспериментальная информатика

Чтение даже множества книг о рекурсии не научит вас реализовывать ее самостоятельно. В этой книге вы найдете немало примеров рекурсивного кода, написанного на языках программирования Python и JavaScript. Со всеми фрагментами программ вы можете поэкспериментировать. Если вы новичок в программировании, рекомендую прочитать мою книгу *Automate the Boring Stuff with Python* или книгу Эрика Мэтиза «Изучаем Python»[1], чтобы познакомиться с базовыми концепциями программирования на языке Python.

Советую запускать примеры программ с применением *отладчика*, который позволяет выполнять код построчно, следя за состоянием программы и точно определяя места возникновения ошибок. В главе 11 книги *Automate the Boring Stuff with Python* объясняется, как использовать отладчик Python. Ее можно бесплатно прочитать в Интернете по адресу https://automatetheboringstuff.com/2e/chapter11.

Примеры кода на языках Python и JavaScript в книге представлены вместе. Код Python сохраняется в файле с расширением `.py`, а код JavaScript — `.html` (не `.js`). Возьмем, к примеру, файл `hello.py`:

```
print('Hello, world!')
```

[1] *Мэтиз Э.* Изучаем Python: программирование игр, визуализация данных, веб-приложения. 3-е изд. — Питер, 2022.

И файл `hello.html`:

```
<script type="text/javascript">
document.write("Hello, world!<br />");
</script>
```

Эти два листинга описывают на двух разных языках программы, дающие одинаковые результаты.

ПРИМЕЧАНИЕ

HTML-тег
 в файле hello.html обозначает разрыв или перенос строки, позволяющий разделить выводимый результат на несколько строк. Функция print() в Python автоматически добавляет символ разрыва строки в конец текста, в то время как функция document.write() в JavaScript этого не делает.

Рекомендую вам вручную набирать код из листингов, а не просто копировать его в новый файл. Это поможет выработать «мышечную память» и заставит обдумывать каждую вводимую строку.

Файлы с расширением `.html` технически не являются исполняемыми, поскольку в них отсутствуют несколько необходимых HTML-тегов, таких как `<html>` и `<body>`, однако ваш браузер все равно сможет их отобразить. Данные теги специально опущены. При написании приведенных программ-примеров главной целью были простота и удобочитаемость, а не демонстрация лучших практик веб-разработки.

Установка Python

Браузер, позволяющий просматривать файлы `.html`, есть на каждом компьютере, однако если вы намерены запускать код из книги на этом языке, вам придется установить Python отдельно. Можно бесплатно скачать Python для Microsoft Windows, Apple macOS и Ubuntu Linux со страницы https://python.org/downloads. Обязательно загрузите версию Python 3 (например, 3.10), а не Python 2, поскольку в более новую версию было внесено несколько обратно несовместимых изменений, из-за чего на Python 2 представленные здесь примеры могут работать некорректно.

Запуск среды IDLE и примеров кода на языке Python

Для написания кода можно использовать редактор IDLE, который поставляется вместе с Python, или установить бесплатный, например Mu Editor (https://codewith.mu), PyCharm Community Edition (https://www.jetbrains.com/pycharm/download) или Microsoft Visual Studio Code (https://code.visualstudio.com/Download).

Чтобы запустить IDLE в операционной системе (ОС) Windows, откройте меню Пуск в нижнем левом углу экрана, введите IDLE в поле поиска и выберите IDLE (Python 3.10 64-bit).

В macOS откройте окно Finder и выберите Applications ▸ Python 3.10, а затем щелкните на значке IDLE.

В Ubuntu выберите Applications ▸ Accessories ▸ Terminal, а затем введите IDLE 3. Вы также можете нажать кнопку Applications в верхней части экрана, выбрать раздел Programming, а затем нажать IDLE 3.

Среда IDLE предусматривает два вида окон. Окно интерактивной оболочки с подсказкой >>> используется для запуска инструкций на языке Python по одной за раз. Это бывает полезно, когда вы хотите поэкспериментировать с фрагментами кода. В окне редактора можно ввести программу целиком и сохранить ее в виде файла с расширением .py. Именно в нем вы будете писать исходный код приведенных в книге программ на языке Python. Чтобы открыть новое окно редактора файлов, выберите пункт New File в меню File. Для запуска программы необходимо нажать клавишу F5 или выбрать пункт меню Run ▸ Run Module.

Запуск примеров кода JavaScript в браузере

Браузер вашего компьютера позволяет запускать программы на языке JavaScript и выводить на экран результат их выполнения, однако для написания самого кода вам понадобится текстовый редактор. Можно использовать как простые программы вроде Блокнота или TextMate, так и специальные, например IDLE или Sublime Text (https://www.sublimetext.com).

После ввода кода программы на языке JavaScript сохраните его в виде файла с расширением .html, а не .js. Откройте его в любом современном браузере, чтобы просмотреть результат.

От издательства

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

Мы будем рады узнать ваше мнение!

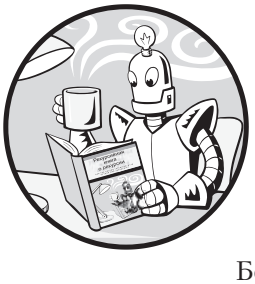
На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

О рекурсии

1

Что такое рекурсия



У рекурсии пугающая репутация. Эта концепция считается сложной для понимания, однако она зависит, по сути, только от двух вещей: от вызовов функций и структуры данных стека.

Большинство начинающих программистов отслеживают логику программы, наблюдая за ее выполнением. Это весьма простой способ чтения кода, при котором вы движетесь, начиная с первой строки и до самого конца. В некоторых случаях вы будете возвращаться назад, а иногда попадать в функцию, из которой затем возвращаться. Так можно легко понять, что именно и в каком порядке делает программа.

Однако для того, чтобы разобраться в рекурсии, вам нужно познакомиться с такой менее очевидной структурой данных, как *стек вызовов*, которая контролирует ход выполнения программы. Большинство новичков не знают о стеках, потому что в руководствах те часто даже не упоминаются при обсуждении вызовов функций. Кроме того, стек вызовов не фигурирует в исходном коде.

Трудно понять то, чего не видишь и о существовании чего даже не подозреваешь! В этой главе мы приоткроем завесу тайны над темой рекурсии, и вы увидите, что эта концепция вовсе не так сложна, как может показаться, и наверняка оцените ее элегантность.

Определение рекурсии

Прежде чем начать, я предлагаю вспомнить самые распространенные шутки про рекурсию, начиная со следующей: «Чтобы понять рекурсию, надо сначала понять рекурсию».

За те несколько месяцев, что я писал книгу, я слышал эту шутку неоднократно и могу вас заверить, что с каждым разом она казалась мне еще смешнее.

А если ввести в поисковую строку Google слово «рекурсия», то на странице с результатами будет написано: «Возможно, вы имели в виду: рекурсия». И перейдя по ссылке, как показано на рис. 1.1, вы попадаете... на страницу с результатами поиска по запросу «рекурсия».

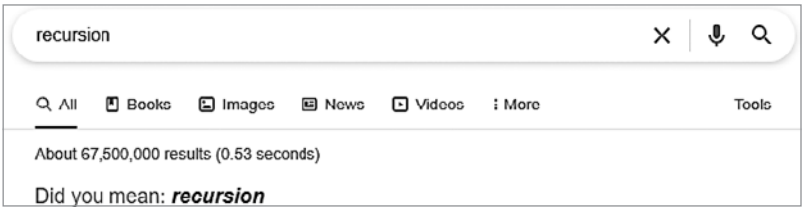


Рис. 1.1. Страница Google с результатами поиска по запросу «рекурсия» ссылается сама на себя

На рис. 1.2 показан пример шуточного рекурсивного акронима из веб-комикса xkcd.

Рис. 1.2. Акроним I'm So Meta, Even This Acronym (I.S. M.E.T.A.) (xkcd.com/917, автор Рэндел Манро)

Большинство шуток о научно-фантастическом боевике «Начало» 2010 года связано с рекурсией. В этом фильме персонажи видят сны, где они видят сны, в которых они видят сны.

И наконец, кто из профессиональных программистов способен забыть такого монстра из греческой мифологии, как рекурсивный кентавр? Как видно на рис. 1.3, это наполовину лошадь, а наполовину рекурсивный кентавр.

Встречая шутки такого рода, вы можете подумать, что рекурсия — это нечто вроде сна внутри сна или отражающегося в зеркале зеркала. Дадим ей конкретное определение: *рекурсивным* называется объект, определение которого включает само себя. То есть этот объект имеет самозамкнутое определение.

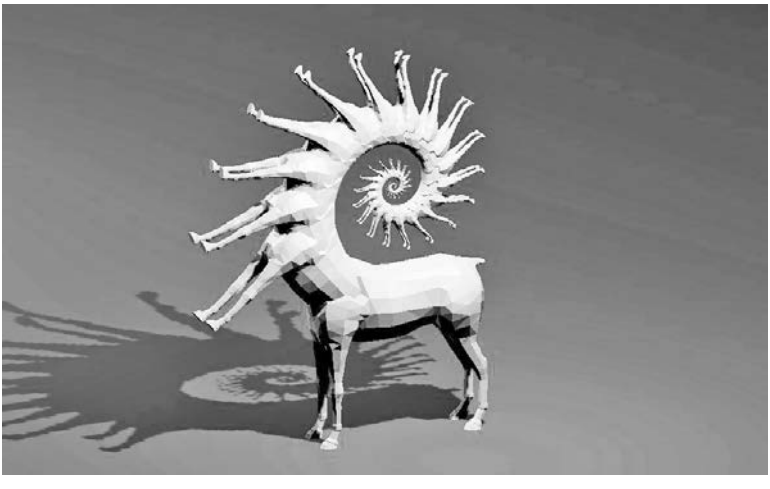


Рис. 1.3. Рекурсивный кентавр. Изображение создано Джозефом Паркером

Треугольник Серпинского, изображенный на рис. 1.4, определяется как равносторонний треугольник с перевернутым треугольником в середине, который образует три новых равносторонних треугольника, каждый из которых содержит треугольник Серпинского. Таким образом, определение треугольника Серпинского включает в себя треугольники Серпинского.

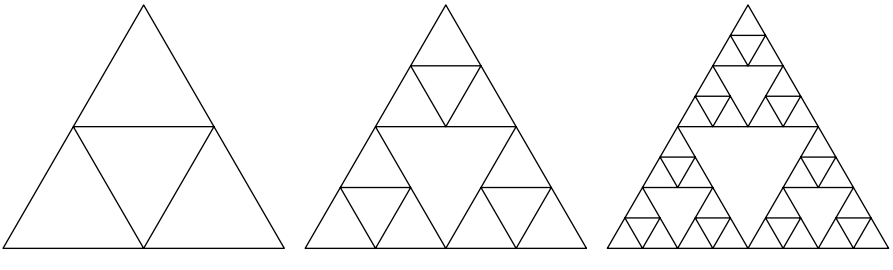


Рис. 1.4. Треугольники Серпинского — это фракталы (рекурсивные формы), которые включают в себя треугольники Серпинского

В контексте программирования *рекурсивной* называется функция, которая вызывает сама себя. Прежде чем приступать к изучению рекурсивных функций, сделаем шаг

назад и разберемся с принципом работы обычных функций. Программисты склонны воспринимать вызовы функций как нечто само собой разумеющееся, однако даже самым опытным из них будет полезно прочесть следующий раздел.

Что такое функции

Функции можно рассматривать как мини-программы внутри основной программы. Они предусмотрены практически во всех языках программирования. Если вам нужно выполнить одни и те же инструкции в трех разных местах программы, то вместо многократного копирования исходного кода достаточно один раз написать функцию и вызвать ее необходимое количество раз. В результате программа получается более короткой и удобочитаемой. Кроме того, ее станет легче изменять: если вам потребуется исправить ошибку в данном фрагменте кода или что-то добавить, достаточно будет внести изменения только в одном месте.

Во всех языках программирования функции имеют четыре характеристики.

1. Функции содержат код, который выполняется при их вызове.
2. В момент вызова функции ей передаются *аргументы* (то есть значения). Это входные данные для функции, количество которых может варьироваться от нуля до бесконечности.
3. Функции возвращают так называемое *возвращаемое значение*, которое представляет собой вывод функции. Правда, некоторые языки программирования позволяют функциям возвращать как нулевые значения, так и ничего вроде `undefined` или `None`.
4. Программа запоминает строку кода, в которой была вызвана функция, и возвращается к ней, когда функция завершает свое выполнение.

В разных языках программирования могут быть предусмотрены дополнительные возможности или варианты вызова функций, но перечисленные выше характеристики являются общими для всех языков. Первые три вы можете непосредственно увидеть, поскольку прописываете их в исходном коде, но как быть с четвертым пунктом?

Чтобы лучше разобраться в этой проблеме, создайте программу `functionCalls.py` (Python) с тремя функциями, где `a()` вызывает `b()`, которая вызывает `c()`:

```
def a():
    print('a() was called.')
    b()
    print('a() is returning.')
def b():
    print('b() was called.')
    c()
    print('b() is returning.')
```

```
def c():
    print('c() was called.')
    print('c() is returning.')

a()
```

Этот код эквивалентен программе `functionCalls.html` (JavaScript):

```
<script type="text/javascript">
function a() {
    document.write("a() was called.<br />");
    b();
    document.write("a() is returning.<br />");
}

function b() {
    document.write("b() was called.<br />");
    c();
    document.write("b() is returning.<br />");
}
function c() {
    document.write("c() was called.<br />");
    document.write("c() is returning.<br />");
}
a();
</script>
```

В итоге мы получим следующее:

```
a() was called.
b() was called.
c() was called.
c() is returning.
b() is returning.
a() is returning.
```

Такой результат показывает начало выполнения функций `a()`, `b()` и `c()`, которые при возвращении отображаются в обратном порядке: `c()`, `b()` и `a()`. Обратите внимание на закономерность вывода текста: каждый раз после выполнения функция возвращается в точку программы, в которой она была вызвана. Когда завершается вызов функции `c()`, программа возвращается к `b()` и отображает строку `b() is returning`. Затем завершается вызов `b()`, и программа возвращается к `a()`, отображая `a() is returning`. Наконец, программа возвращается к исходному вызову `a()` в конце программы. Другими словами, выполнение программы с вызовами функций не похоже на путешествие в один конец.

Но как программа запоминает, что функцию `c()` вызвала именно функция `b()`, а не `a()`? В этом ей помогает стек вызовов. Чтобы понять, как стеки вызовов запоминают точку программы, в которой функция была вызвана, стоит разобраться с самим понятием стека.

Что такое стеки

Ранее я упомянул известную шутку о том, что «для понимания рекурсии надо сначала понять рекурсию». Однако это неверно: чтобы по-настоящему понять рекурсию, необходимо разобраться со стеками.

Стек представляет собой одну из простейших структур данных в информатике. Как и список, он хранит несколько значений, но позволяет добавлять или удалять значения только «сверху». Для стеков, реализованных с помощью списков или массивов, «верхним» является последний элемент в правом конце списка или массива. Добавление значений в стек называется *проталкиванием* (pushing), а извлечение — *выталкиванием* (popping).

Представьте, что в ходе беседы с кем-то вы говорите о своей подруге Алисе, упоминание которой наталкивает вас на мысль о вашем коллеге Бобе, но, чтобы его история имела хоть какой-то смысл, вы сначала должны объяснить кое-что о своей кузине Кэрол. Закончив рассказ о Кэрол, вы приступаете к истории с Бобом, а затем возвращаетесь к разговору об Алисе. Потом вы вспоминаете о своем брате Дэвиде и рассказываете уже о нем. В конце концов, вы завершаете свой рассказ об Алисе.

Ваша беседа имеет стекообразную структуру, изображенную на рис. 1.5, поскольку текущая тема всегда находится на вершине стека.

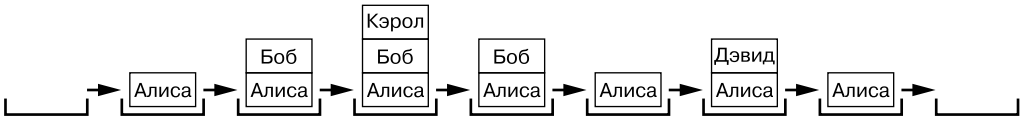


Рис. 1.5. Стекообразная структура беседы

Темы для обсуждения помещаются в стек нашей беседы сверху и удаляются по мере их завершения. Причем предыдущие темы «запоминаются» и хранятся в стеке под текущей темой.

Мы можем использовать в качестве стеков списки Python, если для изменения их содержимого ограничимся методами `append()` и `pop()` для добавления значений в стек и их извлечения. Массивы JavaScript с их методами `push()` и `pop()` также используются в качестве стеков.

ПРИМЕЧАНИЕ

В языке Python используются термины «список» и «элемент», тогда как в JavaScript — «массив» и «элемент», но для наших целей их можно считать идентичными. В книге я использую термины «список» и «элемент» применительно к обоим языкам.

Для наглядности рассмотрим программу `cardStack.py`, которая помещает строковые значения игральных карт в конец списка `cardStack` и извлекает их оттуда:

```
cardStack = ❶ []
❷ cardStack.append('5 of diamonds')
print(','.join(cardStack))
cardStack.append('3 of clubs')
print(','.join(cardStack))
cardStack.append('ace of hearts')
print(','.join(cardStack))
❸ cardStack.pop()
print(','.join(cardStack))
```

Следующая программа `cardStack.html` содержит аналогичный код на языке JavaScript:

```
<script type="text/javascript">
let cardStack = ❶ [];
❷ cardStack.push("5 of diamonds");
document.write(cardStack + "<br />");
cardStack.push("3 of clubs");
document.write(cardStack + "<br />");
cardStack.push("ace of hearts");
document.write(cardStack + "<br />");
❸ cardStack.pop()
document.write(cardStack + "<br />");
</script>
```

Результат выполнения кода выглядит следующим образом:

```
5 of diamonds
5 of diamonds,3 of clubs
5 of diamonds,3 of clubs,ace of hearts
5 of diamonds,3 of clubs
```

Изначально стек пуст ❶. Затем в него помещаются три строки, обозначающие карты ❷. После этого из стека извлекается последняя карта (туз червей) ❸, в результате чего на вершине остается тройка треф. Изменение состояния стека `cardStack` показано на рис. 1.6, *слева направо*.

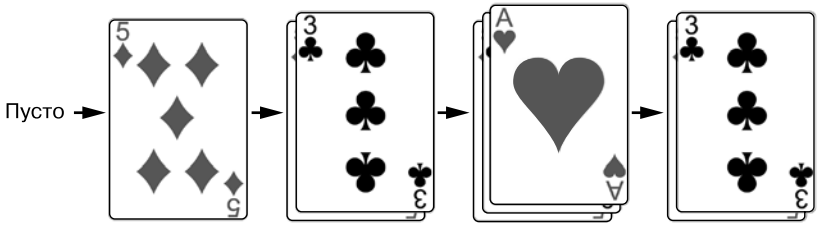


Рис. 1.6. Изначально стек пуст. Затем в него помещаются карты, одна из которых потом извлекается

Вам доступна только самая верхняя карта в стопке или, в случае с нашей программой, самое верхнее значение в стеке. В простейших реализациях стека вы не видите, сколько карт (или значений) находится в нем. Можно лишь узнать, пуст стек или нет.

Стеки — это структура данных типа *LIFO*, что означает *«последним пришел — первым ушел»* (last in, first out), поскольку последнее помещенное в стек значение первым из него извлекается. Такое поведение похоже на результат нажатия кнопки Назад в браузере. История закрытых вкладок в вашем браузере функционирует подобно стеку, содержащему все страницы, которые вы посетили, и именно в том порядке, в котором вы их просматривали. Браузер всегда отображает веб-страницу, находящуюся на вершине стека истории: щелчок на ссылке добавляет в него новую веб-страницу, а нажатие кнопки Назад удаляет ее и показывает ту, что находится «под ней».

Что такое стек вызовов

Программы тоже используют стеки. *Стек вызовов* программы, или просто *стек*, представляет собой объекты, называемые *кадрами*. Они содержат информацию об одном вызове функции, в том числе о строке кода, с которой должно продолжиться выполнение программы после завершения функции.

Кадры создаются и помещаются в стек при вызове функции. После возврата из функции соответствующий объект извлекается из стека. Если мы вызовем функцию, которая вызывает функцию, вызывающую функцию, то в стеке вызовов будет три кадра. После завершения всех этих функций стек вызовов опустеет.

Вам не нужно писать код для работы с кадрами, поскольку язык программирования обрабатывает их автоматически. Каждый язык по-разному реализует эти объекты, но, как правило, кадры содержат следующие элементы:

- адрес возврата или точку, с которой должно продолжиться выполнение программы после возврата из функции;
- аргументы, передаваемые в функцию при ее вызове;
- набор локальных переменных, созданных во время вызова функции.

Например, взгляните на программы `localVariables.py` и `localVariables.html`, которые содержат три функции, как и наши предыдущие программы `functionCalls.py` и `functionCalls.html`:

```
def a():
 ❶ spam = 'Ant'
 ❷ print('spam is ' + spam)
 ❸ b()
   print('spam is ' + spam)
```

```
def b():
 ❹ spam = 'Bobcat'
    print('spam is ' + spam)
 ❺ c()
    print('spam is ' + spam)

def c():
 ❻ spam = 'Coyote'
    print('spam is ' + spam)

❼ a()
```

Аналогичный код, но на языке JavaScript:

```
<script type="text/javascript">
function a() {
 ❶ let spam = "Ant";
 ❷ document.write("spam is " + spam + "<br />");
 ❸ b();
    document.write("spam is " + spam + "<br />");
}
function b() {
 ❹ let spam = "Bobcat";
    document.write("spam is " + spam + "<br />");
 ❺ c();
    document.write("spam is " + spam + "<br />");
}
function c() {
 ❻ let spam = "Coyote";
    document.write("spam is " + spam + "<br />");
}
❼ a();
</script>
```

В результате мы получим следующее:

```
spam is Ant
spam is Bobcat
spam is Coyote
spam is Bobcat
spam is Ant
```

Когда программа вызывает функцию `a()` ❼, создается кадр и помещается на вершину стека вызовов. Этот кадр хранит все аргументы, переданные функции `a()` (в данном случае их нет), а также локальную переменную `spam` ❶ и место, откуда должно продолжиться выполнение программы после завершения `a()`.

Когда вызывается функция `a()`, она отображает содержимое своей локальной переменной `spam` — `Ant` ❷. Когда код в `a()` вызывает `b()` ❸, создается новый кадр, который помещается в стек вызовов поверх кадра для `a()`. Функция `b()` имеет

свою локальную переменную `spam` ❹ и вызывает функцию `c()` ❺, для которой создается новый кадр с локальной переменной `spam` и помещается в стек вызовов ❻. По мере завершения этих функций кадры удаляются из стека вызовов. Программа знает, откуда она должна продолжить свое выполнение, потому что внутри кадра хранится адрес возврата. После завершения выполнения всех функций стек вызовов остается пустым.

На рис. 1.7 показано состояние стека вызовов по мере вызова и возврата каждой функции. Обратите внимание, что все локальные переменные имеют одно и то же имя: `spam`. Я сделал так специально, чтобы подчеркнуть, что локальные переменные в разных функциях всегда являются независимыми переменными с всевозможными значениями, даже если их имена совпадают.

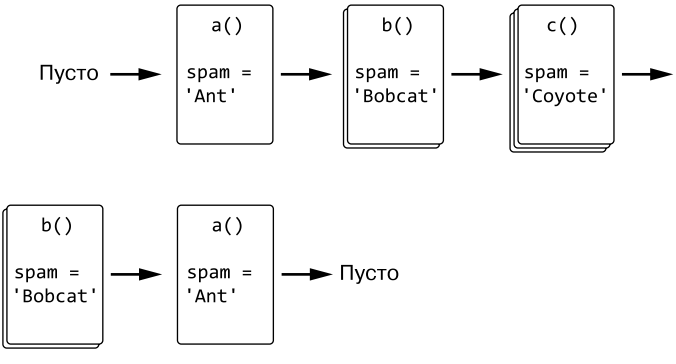


Рис. 1.7. Состояние стека вызовов при выполнении программы localVariables

Как видите, языки программирования допускают использование отдельных локальных переменных с одинаковыми именами (`spam`), поскольку они хранятся в уникальных кадрах. При использовании локальной переменной в исходном коде задействуется одноименная переменная, находящаяся в самом верхнем кадре.

Каждая выполняющаяся программа предусматривает стек вызовов, а многопоточные программы имеют по одному стеку вызовов для каждого потока. Однако при просмотре исходного кода программы вы не увидите в нем даже упоминания о стеке вызовов. Он не хранится в переменной, как другие структуры данных, а автоматически обрабатывается в фоновом режиме.

Отсутствие стека вызовов в исходном коде — основная причина, по которой рекурсия сбивает новичков с толку: рекурсия опирается на то, что разработчик даже не в состоянии увидеть! Понимание принципа работы такой структуры данных, как стек (вызовов), развеивает большую часть ореола таинственности, присущего данной концепции. Функции и стеки — это простые понятия, объединив которые мы сможем разобраться, как работает рекурсия.

Что такое рекурсивные функции и переполнение стека

Рекурсивная функция — это функция, которая вызывает сама себя. Следующая программа, `shortest.py`, представляет собой простейший пример такой функции:

```
def shortest():
    shortest()

shortest()
```

Она же на языке JavaScript (`shortest.html`):

```
<script type="text/javascript">
function shortest() {
    shortest();
}

shortest();
</script>
```

Все, что делает функция `shortest()`, — вызывает функцию `shortest()`. Когда это происходит, она снова вызывает функцию `shortest()`, которая вызывает функцию `shortest()` и т. д., судя по всему, до бесконечности. Это похоже на миф о том, что земля покоится на спине гигантской космической черепахи, которая покоится на спине другой черепахи, под которой находится еще одна черепаха и далее до бесконечности.

Однако подобная «черепаховая» теория не помогает разобраться ни в космологии, ни в рекурсивных функциях. Поскольку стек вызовов использует ограниченную память компьютера, продемонстрированная выше программа не сможет реализовать бесконечный цикл. Единственное, на что она способна, — аварийно завершить работу и вывести сообщение об ошибке.

> **ПРИМЕЧАНИЕ**
>
> Чтобы просмотреть ошибку JavaScript, вам необходимо открыть инструменты разработчика в браузере. Обычно для этого достаточно нажать клавишу F12 и выбрать вкладку Console.

Вывод программы `shortest.py` на языке Python выглядит следующим образом:

```
Traceback (most recent call last):
  File "shortest.py", line 4, in <module>
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
```

```
  File "shortest.py", line 2, in shortest
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

А вот вывод программы `shortest.html` на языке JavaScript в браузере Google Chrome (другие браузеры отображают схожие сообщения об ошибках):

```
Uncaught RangeError: Maximum call stack size exceeded
    at shortest (shortest.html:2)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
```

Такая ошибка называется *переполнением стека* (stack overflow, именно в ее честь был назван популярный сайт https://stackoverflow.com). Многократные вызовы функций без возврата увеличивают стек вызовов до тех пор, пока не будет израсходована вся выделенная для него память компьютера. Чтобы это предотвратить, интерпретаторы Python и JavaScript принудительно завершают работу программы после достижения лимита вызовов невозвратных функций.

Такой предел называется *максимальной глубиной рекурсии* или *максимальным размером стека вызовов*. В Python граничным значением считается 1000 вызовов функций. В случае с JavaScript максимальный размер стека вызовов уже зависит от браузера, в котором выполняется код, но обычно составляет не менее 10 000. Переполнение стека происходит, когда он становится «слишком высоким» (то есть потребляет слишком много памяти компьютера), как показано на рис. 1.8.

Рис. 1.8. Переполнение стека происходит, когда в нем содержится слишком много кадров, занимающих память компьютера

Переполнение стека не вредит компьютеру. При достижении экстремума вызовов подобных функций компьютер просто завершает работу программы. В худшем случае вы рискуете потерять несохраненные данные. Переполнение стека можно предотвратить с помощью так называемого *базового случая*, речь о котором пойдет далее.

Базовые и рекурсивные случаи

Ранее мы рассмотрели проблему переполнения стека на примере функции `shortest()`, которая вызывает функцию `shortest()`, но никогда не возвращает значение. Чтобы избежать сбоя, необходимо предусмотреть набор обстоятельств, при которых функция перестает вызывать саму себя и просто возвратится. Это и есть *базовый случай*. Ситуация, в которой функция рекурсивно вызывает саму себя, называется *рекурсивным случаем*.

Все рекурсивные функции должны предусматривать по крайней мере один базовый и один рекурсивный случай. При отсутствии первого функция никогда не перестанет вызывать саму себя, что в конечном итоге приведет к переполнению стека. При отсутствии второго функция никогда не вызовет саму себя и будет обычной, а не рекурсивной функцией. При написании рекурсивных функций сначала следует подумать о базовом и рекурсивном случаях.

Рассмотрим примеры программ `shortestWithBaseCase.py` и `shortestWithBaseCase.html`, которые определяют самую короткую рекурсивную функцию, неспособную вызвать сбой, обусловленный переполнением стека:

```
def shortestWithBaseCase(makeRecursiveCall):
    print('shortestWithBaseCase(%s) called.' % makeRecursiveCall)
    if not makeRecursiveCall:
        # БАЗОВЫЙ СЛУЧАЙ
        print('Returning from base case.')
     ❶ return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ shortestWithBaseCase(False)
        print('Returning from recursive case.')
        return
print('Calling shortestWithBaseCase(False):')
❸ shortestWithBaseCase(False)
print()
print('Calling shortestWithBaseCase(True):')
❹ shortestWithBaseCase(True)
```

Тот же код на языке JavaScript:

```
<script type="text/javascript">
function shortestWithBaseCase(makeRecursiveCall) {
    document.write("shortestWithBaseCase(" + makeRecursiveCall +
     ") called.<br />");
    if  (makeRecursiveCall === false) {
        // БАЗОВЫЙ СЛУЧАЙ
        document.write("Returning from base case.<br />");
     ❶ return;    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ shortestWithBaseCase(false);
        document.write("Returning from recursive case.<br />");
        return;
    }
}

document.write("Calling shortestWithBaseCase(false):<br />");
❸ shortestWithBaseCase(false);
document.write("<br />");
document.write("Calling shortestWithBaseCase(true):<br />");
❹ shortestWithBaseCase(true);
</script>
```

Результат выполнения этого кода выглядит следующим образом:

```
Calling shortestWithBaseCase(False):
shortestWithBaseCase(False) called.
Returning from base case.

Calling shortestWithBaseCase(True):
shortestWithBaseCase(True) called.
shortestWithBaseCase(False) called.
Returning from base case.
Returning from recursive case.
```

Единственное назначение данной функции — служить коротким примером рекурсии (который можно сделать еще короче, удалив текстовый вывод, однако выводимый текст полезен для объяснения). При вызове `shortestWithBaseCase(False)` ❸ выполняется базовый случай и функция просто возвращается ❶. Однако при вызове `shortestWithBaseCase(True)` ❹ выполняется рекурсивный случай и вызывается `shortestWithBaseCase(False)` ❷.

Важно отметить, что, когда `shortestWithBaseCase(False)` рекурсивно вызывается из точки ❷, а затем возвращается, выполнение программы не сразу переходит в точку

исходного вызова функции ❹. После рекурсивного вызова выполняется остальной код, предусмотренный в рекурсивном случае, поэтому в текстовом выводе появляется фраза `Returning from recursive case` (Возврат из рекурсивного случая). Возврат из базового случая не приводит к немедленному возврату из всех рекурсивных вызовов, которые имели место до него. Мы еще обсудим это при рассмотрении примера с функцией `countDownAndUp()`, приведенного в следующем разделе.

Код, находящийся до и после рекурсивного вызова

Код в рекурсивном случае состоит из двух частей: находящийся до рекурсивного вызова и после. Если в рекурсивном случае содержится два рекурсивных вызова, как в примере с последовательностью Фибоначчи из главы 2, то код будет разделен на «до», «между» и «после». Но пока не будем усложнять.

Важно понимать, что достижение базового случая не обязательно означает достижение конца рекурсивного алгоритма. Это говорит лишь о том, что рекурсивные вызовы больше не будут выполняться.

Например, рассмотрим программу `countDownAndUp.py`, в которой рекурсивная функция производит отсчет от любого числа до нуля, а затем от нуля до этого числа:

```
def countDownAndUp(number):
 ❶ print(number)
    if number == 0:
        # БАЗОВЫЙ СЛУЧАЙ
     ❷ print('Reached the base case.')
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ countDownAndUp(number - 1)
     ❹ print(number, 'returning')
        return

❺ countDownAndUp(3)
```

Этот код эквивалентен программе `countDownAndUp.html`:

```
<script type="text/javascript">
function countDownAndUp(number) {
 ❶ document.write(number + "<br />");
    if (number === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
     ❷ document.write("Reached the base case.<br />");
        return;
    } else {
```

```
        // РЕКУРСИВНЫЙ СЛУЧАЙ
      ❸ countDownAndUp(number - 1);
      ❹ document.write(number + " returning<br />");
        return;
    }
}

❺ countDownAndUp(3);
</script>
```

Результат выполнения этого кода выглядит следующим образом:

```
3
2
1
0
Reached the base case.
1 returning
2 returning
3 returning
```

Помните, что каждый вызов функции сопровождается созданием нового кадра, который помещается в стек вызовов. В этом кадре хранятся все параметры и локальные переменные (например, `number`). Итак, каждый кадр в стеке вызовов содержит отдельную переменную `number`. Эта особенность рекурсии также часто сбивает с толку: при просмотре исходного кода может показаться, что существует только одна переменная `number`, однако, поскольку она является локальной, ее значение будет различаться для каждого вызова функции.

При вызове `countDownAndUp(3)` ❺ создается кадр, а для его локальной переменной `number` задается значение `3`. Функция выводит значение этой переменной на экран ❶. Пока оно не достигнет `0`, функция `countDownAndUp()` рекурсивно вызывается с аргументом `number - 1` ❸. При вызове `countDownAndUp(2)` в стек помещается новый кадр, и для его локальной переменной `number` задается значение `2`. По достижении рекурсивного случая вызывается `countDownAndUp(1)`, после чего рекурсивный случай достигается снова и вызывается `countDownAndUp(0)`.

Подобная последовательность вызовов рекурсивных функций и возврата из них обеспечивает обратный отсчет. При вызове `countDownAndUp(0)` достигается базовый случай ❷, после которого рекурсивные вызовы больше не выполняются. Но на этом работа программы не заканчивается! При достижении базового случая значение локальной переменной `number` равно `0`. Однако после возврата и извлечения кадра из стека вызовов на вершине стека оказывается кадр с локальной переменной `number`, имеющей значение `1`. По мере возвращения к предыдущим кадрам в стеке вызовов выполняется код, находящийся *после* рекурсивного вызова ❹. Именно он отвечает за подсчет чисел. На рис. 1.9 показано состояние стека при каждом вызове рекурсивной функции `countDownAndUp()`.

Когда мы будем вычислять факториал в следующей главе, не забывайте, что выполнение кода не прекращается по достижении базового случая. Учтите, что любой код, находящийся после рекурсивного случая, все равно будет выполнен.

Рекурсивная функция `countDownAndUp()` может показаться слишком сложной. Почему бы не использовать вместо нее итеративное решение для вывода чисел на экран? *Итеративный* подход, использующий циклы для повторения кода вплоть до решения задачи, обычно считается противоположностью рекурсии.

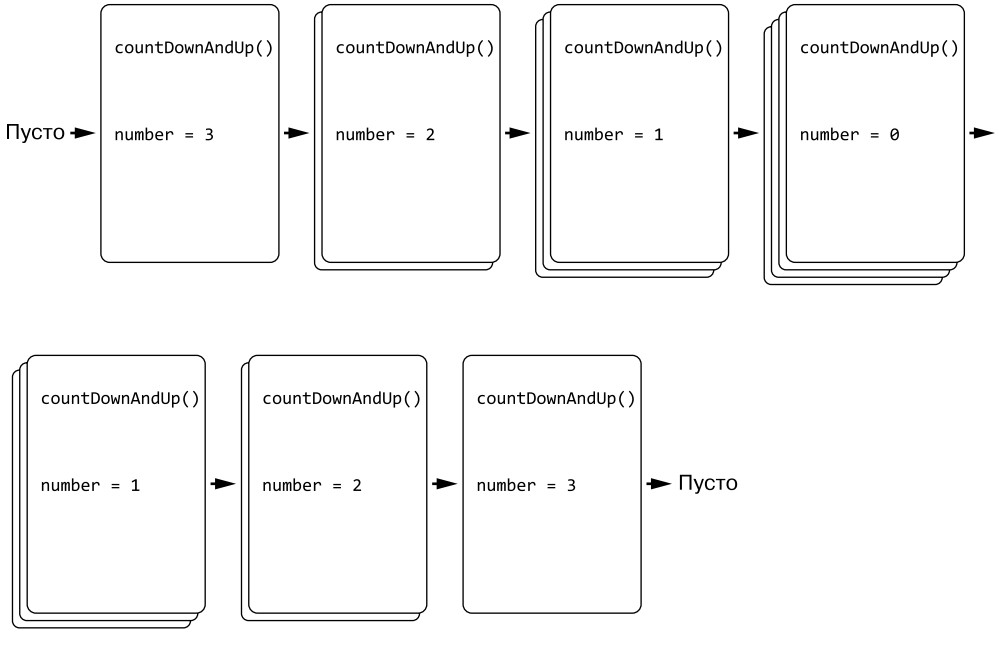


Рис. 1.9. Состояние стека вызовов, отслеживающего значения локальной переменной number при каждом вызове функции

Всякий раз, когда вы задаетесь вопросом «А не проще ли использовать цикл?», ответом почти наверняка будет «Да», и в этих случаях рекурсивного решения следует избегать. Рекурсия может создавать сложности как для начинающих, так и для опытных программистов, а рекурсивный код не всегда более предпочтителен или элегантен по сравнению с итеративным. Удобочитаемость и простота кода важнее свойственной рекурсии элегантности. Однако в некоторых случаях, когда речь идет, например, об алгоритмах поиска с возвратом и обхода дерева, рекурсивный подход особенно хорош. Эти случаи будут подробно рассмотрены в главах 2 и 4.

Резюме

Концепция рекурсии часто сбивает с толку начинающих программистов, однако в ее основе лежит простая идея функции, вызывающей саму себя. При каждом вызове функции в стек вызовов добавляется новый кадр с относящейся к этому вызову информацией (например, локальные переменные и адрес возврата, с которого будет продолжено выполнение программы после завершения функции). Стек вызовов может быть изменен только путем добавления (проталкивания) значения на его «вершину» или удаления (выталкивания) с нее.

Стек вызовов обрабатывается программой неявно, поэтому в коде отсутствует какое-либо упоминание о нем. При вызове функции кадр помещается в стек вызовов, а при возврате из функции извлекается из стека.

Рекурсивные функции предусматривают рекурсивные случаи, в которых выполняется рекурсивный вызов, и базовые случаи, в которых функция просто возвращается. Если базовый случай не предусмотрен или какая-то ошибка препятствует выполнению соответствующего кода, переполнение стека вызывает сбой программы.

Рекурсия — довольно полезная техника, но она не всегда делает код лучше или элегантнее. Об этом мы поговорим подробнее в следующей главе.

Дополнительные источники информации

Узнать больше о рекурсии вы можете, просмотрев мое выступление на конференции North Bay Python 2018 *Recursion for Beginners: A Beginner's Guide to Recursion*, оно доступно по адресу https://youtu.be/AfBqVVKg4GE. Еще одним хорошим введением в тему станет видео *What on Earth is Recursion* на YouTube-канале Computerphile (https://youtu.be/Mv9NEXX1VHc). Антон Спрол (Anton Spraul) рассказывает о рекурсии в своей книге «Думай как программист»[1] и в видео *Recursion (Think Like a Programmer)* (https://youtu.be/oKndim5-G94). Подробное описание рекурсии также можно найти в «Википедии»: https://ru.wikipedia.org/wiki/Рекурсия.

Для Python разработан модуль `ShowCallStack`, добавляющий функцию `showcallstack()`, которую затем можно поместить в любом месте кода, чтобы увидеть состояние стека вызовов в конкретной точке программы. Модуль и документация к нему доступны по адресу https://pypi.org/project/ShowCallStack.

[1] *Спрол А.* Думай как программист. Креативный подход к созданию кода. C++ версия.

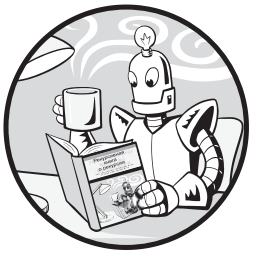
Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое рекурсия?
2. Что такое рекурсивная функция в программировании?
3. Каковы четыре характеристики функций?
4. Что такое стек?
5. Как называются операции добавления и удаления значений из стека?
6. Допустим, вы помещаете в стек букву *J*, затем добавляете в него букву *Q*, потом выталкиваете из стека одно значение, затем помещаете в него букву *K*, а затем снова выталкиваете из стека одно значение. Как в итоге выглядит этот стек?
7. Что помещается в стек вызовов и извлекается из него?
8. Что вызывает переполнение стека?
9. Что такое базовый случай?
10. Что такое рекурсивный случай?
11. Сколько базовых и рекурсивных случаев предусматривают рекурсивные функции?
12. Что будет, если рекурсивная функция не предусматривает базовых случаев?
13. Что будет, если рекурсивная функция не предусматривает рекурсивных случаев?

2

Рекурсия и итерация



В целом ни рекурсия, ни итерация не являются универсальными методами. Фактически любой рекурсивный код возможно написать в виде итеративного с помощью цикла и стека. Равно как и итеративный можно переписать в виде рекурсивной функции. Стоит отметить, что рекурсия не обладает какими-то особыми вычислительными свойствами, которых нет у итеративного алгоритма.

В текущей главе мы сравним две эти техники. Рассмотрим классические способы вычисления факториалов и ряда Фибоначчи, а также выясним, в чем заключаются критические недостатки соответствующих рекурсивных алгоритмов. Кроме того, мы изучим рекурсивный подход в контексте алгоритма экспоненты. В целом глава призвана пролить свет на мнимую элегантность рекурсивных алгоритмов и показать, когда рекурсивное решение бывает полезным, а когда — нет.

Вычисление факториалов

В рамках многих учебных курсов по информатике вычисление факториалов используется в качестве классического примера применения рекурсивной функции. Факториал целого числа (назовем его n) — это произведение всех целых чисел от 1 до n. Например, факториал числа 4 — это $4 \times 3 \times 2 \times 1$, или 24. Восклицательный знак — это математическое обозначение факториала, то есть выражение 4! означает *факториал числа 4*. В табл. 2.1 приведены факториалы нескольких целых чисел.

Факториалы используются во всевозможных видах расчетов, например при определении количества вариантов перестановок чего-либо. Если вас интересует количество способов выстроить очередь из четырех человек — Алисы, Боба,

Кэрол и Дэвида, то ответом будет факториал числа 4. Любой может быть первым в очереди (4), вторыми могут быть трое оставшихся (4×3), два человека могут быть третьими ($4 \times 3 \times 2$), и кто-то один — четвертым ($4 \times 3 \times 2 \times 1$). Таким образом, количество способов упорядочения людей в очереди, то есть совокупность перестановок, является факториалом общего числа людей.

Таблица 2.1. Факториалы нескольких целых чисел

n!		Расширенная форма		Произведение
1!	=	1	=	1
2!	=	1×2	=	2
3!	=	$1 \times 2 \times 3$	=	6
4!	=	$1 \times 2 \times 3 \times 4$	=	24
5!	=	$1 \times 2 \times 3 \times 4 \times 5$	=	120
6!	=	$1 \times 2 \times 3 \times 4 \times 5 \times 6$	=	720
7!	=	$1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$	=	5040
8!	=	$1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$	=	40 320

Теперь рассмотрим итеративный и рекурсивный подходы к вычислению факториалов.

Итеративный алгоритм вычисления факториала

Вычислить факториал с помощью итеративного подхода довольно просто: достаточно перемножить в цикле целые числа от 1 до *n* включительно. *Итеративные* алгоритмы всегда используют цикл. Программа `factorialByIteration.py` на языке Python выглядит так:

```
def factorial(number):
    product = 1
    for i in range(1, number + 1):
        product = product * i
    return product
print(factorial(5))
```

Та же программа `factorialByIteration.html`, но на языке JavaScript:

```
<script type="text/javascript">
function factorial(number) {
    let product = 1;
    for (let i = 1; i <= number; i++) {
        product = product * i;
    }
```

```
    return product;
}
document.write(factorial(5));
</script>
```

После выполнения кода на экране отобразится результат вычисления 5!:

```
120
```

В итеративном способе расчета факториалов нет ничего плохого, он прост и позволяет успешно решить поставленную задачу. Однако давайте рассмотрим рекурсивный алгоритм для более глубокого понимания природы факториалов и самой рекурсии.

Рекурсивный алгоритм вычисления факториала

Обратите внимание, что факториал числа 4 равен $4 \times 3 \times 2 \times 1$, а факториал числа $5 — 5 \times 4 \times 3 \times 2 \times 1$. Таким образом, можно сказать, что $5! = 5 \times 4!$. Это выражение является *рекурсивным*, поскольку определение факториала числа 5 (или любого другого числа n) включает в себя вычисление факториала числа 4 (то есть числа $n - 1$). В свою очередь, $4! = 4 \times 3!$ и т. д., пока вы не дойдете до 1! — базового случая, в котором результат равен 1.

Программа расчета факториала с помощью рекурсивного алгоритма на языке Python `factorialByRecursion.py`:

```
def factorial(number):
    if number == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        return 1
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❶ return number * factorial(number - 1)
print(factorial(5))
```

Эквивалентная программа `factorialByRecursion.html` на языке JavaScript:

```
<script type="text/javascript">
function factorial(number) {
    if (number == 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        return 1;    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❶ return number * factorial(number - 1);
    }
}
document.write(factorial(5));
</script>
```

После выполнения этого кода на экране отобразится результат вычисления 5!, который совпадает с выводом предыдущей программы:

```
120
```

Многим программистам такой рекурсивный код может показаться странным. Вы знаете, что `factorial(5)` вычисляется как $5 \times 4 \times 3 \times 2 \times 1$, но не можете указать строку кода, в которой происходит перемножение.

Путаница возникает из-за того, что в рекурсивном случае есть строка кода ❶, которая разбивается на две части: одна запускается перед рекурсивным вызовом, а другая — после возврата. Идея выполнения лишь половины строки кода за раз кажется непривычной.

Первая половина строки — фрагмент кода `factorial(number - 1)`. Он включает в себя вычисление выражения `number – 1` и создание рекурсивной функции, помещающей в стек вызовов новый кадр. Это происходит до выполнения рекурсивного вызова.

Следующий запуск кода с прежним кадром осуществляется уже после возврата `factorial(number - 1)`. При вызове функции `factorial(5)` выражение `factorial(number - 1)` будет соответствовать выражению `factorial(4)`, которое возвращает значение `24`. Здесь уже выполняется вторая половина строки кода. Теперь выражение `return number * factorial(number - 1)` выглядит как `return 5 * 24`, и именно поэтому вызов `factorial(5)` возвращает значение `120`.

На рис. 2.1 показано изменение состояния стека вызовов по мере добавления кадров (при вызове рекурсивной функции) и их извлечения (при возврате функции). Обратите внимание, что перемножение значений происходит после выполнения рекурсивных вызовов, а не до.

Когда работа исходной функции `factorial()` завершается, возвращается значение вычисленного факториала.

Чем плох рекурсивный алгоритм вычисления факториала

Рекурсивный алгоритм определения факториала, например, числа 5 имеет существенный недостаток: необходимо совершить пять вызовов рекурсивных функций. То есть до достижения базового случая в стек вызовов будет помещено пять кадров. И подобный подход не масштабируется.

Если вы захотите вычислить факториал числа 1001, вам придется вызвать рекурсивную функцию `factorial()` 1001 раз. Однако ваша программа, скорее всего, аварийно завершится в связи с переполнением стека, потому что такое количество вызовов функций без возврата, вероятно, превысит максимальный размер стека

вызовов интерпретатора. Никогда не используйте рекурсивный алгоритм вычисления факториала в реальном коде — это ужасно.

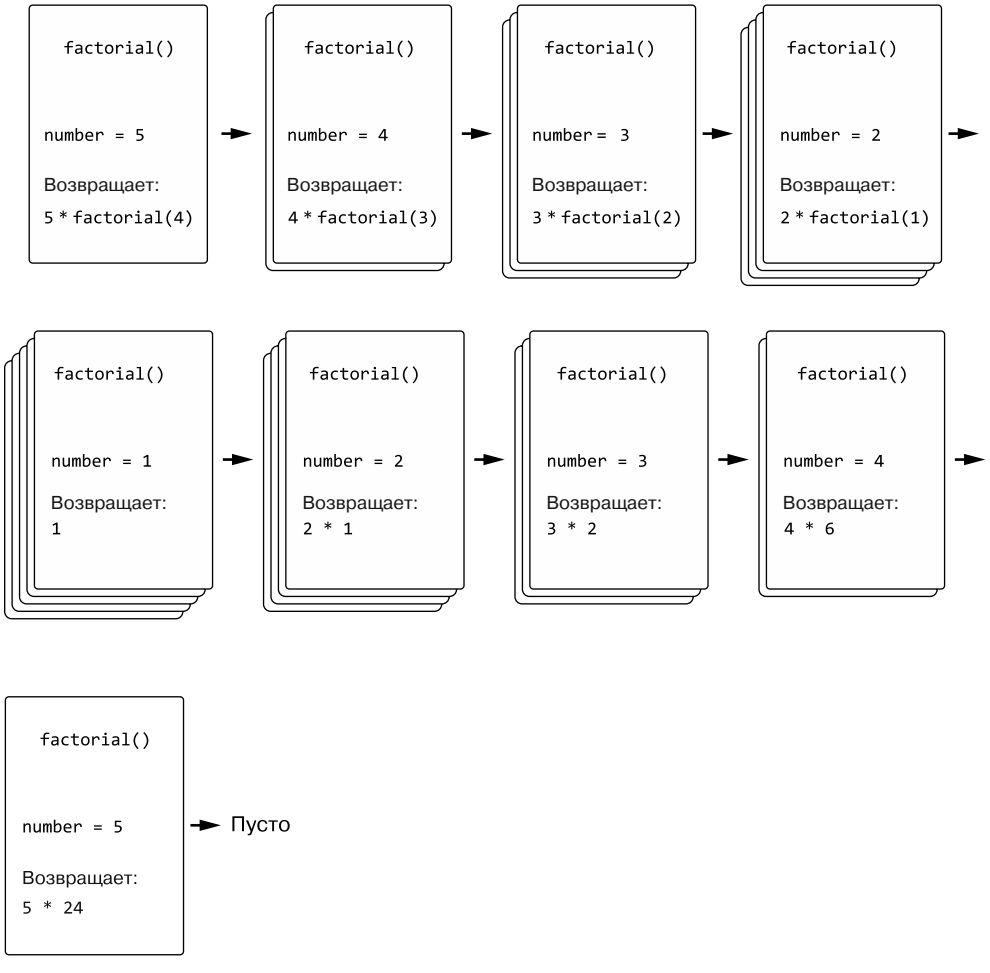


Рис. 2.1. Изменение состояния стека вызовов в результате многократного вызова рекурсивной функции factorial() и последующего возврата

Итеративный алгоритм способен вычислить факториал числа быстро и эффективно. Переполнения стека можно избежать с помощью доступной в некоторых языках программирования техники под названием *«оптимизация хвостовых вызовов»*, о которой мы поговорим в главе 8. Однако она еще больше усложняет реализацию рекурсивной функции. То есть для вычисления факториалов итеративный подход является самым простым и понятным.

Вычисление последовательности Фибоначчи

Еще один классический пример, к которому обычно обращаются при объяснении рекурсии, — *последовательность Фибоначчи*, которая начинается с чисел 1 и 1 (иногда с 0 и 1). Каждое последующее число в этой последовательности представляет собой сумму двух предыдущих. В результате создается последовательность 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 и далее до бесконечности.

Если мы обозначим два последних числа в последовательности буквами *a* и *b*, то сможем представить ее рост как на рис. 2.2.

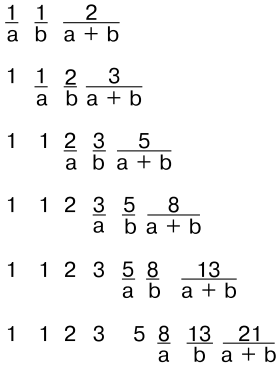


Рис. 2.2. Каждое число в последовательности Фибоначчи является суммой двух предыдущих чисел

Рассмотрим несколько примеров итеративных и рекурсивных алгоритмов вычисления чисел Фибоначчи.

Итеративный алгоритм вычисления чисел Фибоначчи

Итеративный алгоритм расчета последовательности Фибоначчи довольно прост. Он состоит из цикла `for` и двух переменных — `a` и `b`. Данный алгоритм реализован в программе `fibonacciByIteration.py` на языке Python:

```
def fibonacci(nthNumber):
 ❶ a, b = 1, 1
    print('a = %s, b = %s' % (a, b))
    for i in range(1, nthNumber):
     ❷ a, b = b, a + b # Получение следующего числа Фибоначчи
        print('a = %s, b = %s' % (a, b))
    return a

print(fibonacci(10))
```

Аналогичная программа на языке JavaScript (`fibonacciByIteration.html`):

```
<script type="text/javascript">
function fibonacci(nthNumber) {
 ❶ let a = 1, b = 1;
    let nextNum;
    document.write('a = ' + a + ', b = ' + b + '<br />');
    for (let i = 1; i < nthNumber; i++) {
     ❷ nextNum = a + b; // Получение следующего числа Фибоначчи
        a = b;
        b = nextNum;
        document.write('a = ' + a + ', b = ' + b + '<br />');
    }
    return a;
};

document.write(fibonacci(10));
</script>
```

Результат вычисления десятого числа последовательности Фибоначчи с помощью этого алгоритма выглядит следующим образом:

```
a = 1, b = 1
a = 1, b = 2
a = 2, b = 3
--пропущенный фрагмент--
a = 34, b = 55
55
```

Программе приходится отслеживать только два последних числа последовательности одновременно. Поскольку на первом и втором месте в ряде Фибоначчи стоит единица, мы сохраняем значение `1` в переменных `a` и `b` ❶. Внутри цикла `for` вычисляется следующая цифра последовательности путем сложения значений `a` и `b` ❷. Эта величина сохраняется в переменной `b`, предыдущее значение которой сохраняется в переменной `a`. В момент завершения цикла переменная `b` должна содержать *n*-е число последовательности Фибоначчи, которое и будет возвращено.

Рекурсивный алгоритм вычисления чисел Фибоначчи

Процесс вычисления последовательности Фибоначчи имеет свойство рекурсивности. Например, если вы хотите рассчитать десятый элемент такой последовательности, вы складываете девятое и восьмое числа Фибоначчи, для вычисления которых необходимо сложить восьмое и седьмое, а затем седьмое и шестое. То есть имеет место множество повторных вычислений. Обратите внимание, что сложение девятого и восьмого чисел Фибоначчи требует повторного вычисления восьмого числа данной последовательности. Этот рекурсивный процесс продолжается вплоть до достижения базового случая, то есть первого или второго компонента ряда Фибоначчи, значение которого всегда равно 1.

Рекурсивная функция для вычисления последовательности Фибоначчи представлена в программе `fibonacciByRecursion.py` на языке Python:

```
def fibonacci(nthNumber):
    print('fibonacci(%s) called.' % (nthNumber))
    if nthNumber == 1 or nthNumber == 2: ❶
        # БАЗОВЫЙ СЛУЧАЙ
        print('Call to fibonacci(%s) returning 1.' % (nthNumber))
        return 1
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        print('Calling fibonacci(%s) and fibonacci(%s).' %
         (nthNumber - 1, nthNumber - 2))
        result = fibonacci(nthNumber - 1) + fibonacci(nthNumber - 2)
        print('Call to fibonacci(%s) returning %s.' % (nthNumber, result))
        return result

print(fibonacci(10))
```

В файле `fibonacciByRecursion.html` находится эта же программа на языке JavaScript:

```
<script type="text/javascript">
function fibonacci(nthNumber) {
    document.write('fibonacci(' + nthNumber + ') called.<br />');
    if (nthNumber === 1 || nthNumber === 2) { ❶
        // БАЗОВЫЙ СЛУЧАЙ
        document.write('Call to fibonacci(' + nthNumber + ')
          returning 1.<br />');
        return 1;
    }
    else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        document.write('Calling fibonacci(' + (nthNumber - 1) + ') and
          fibonacci(' + (nthNumber - 2) + ').<br />');
        let result = fibonacci(nthNumber - 1) + fibonacci(nthNumber - 2);
        document.write('Call to fibonacci(' + nthNumber + ') returning ' +
          result + '.<br />');
        return result;
    }
}

document.write(fibonacci(10) + '<br />');
</script>
```

Результат вычисления десятого числа последовательности Фибоначчи выглядит следующим образом:

```
fibonacci(10) called.
Calling fibonacci(9) and fibonacci(8).
fibonacci(9) called.
Calling fibonacci(8) and fibonacci(7).
```

```
fibonacci(8) called.
Calling fibonacci(7) and fibonacci(6).
fibonacci(7) called.
--пропущенный фрагмент--
Call to fibonacci(6) returning 8.
Call to fibonacci(8) returning 21.
Call to fibonacci(10) returning 55.
55
```

Большая часть продемонстрированного кода отвечает за отображение приведенного выше вывода, но сама функция `fibonacci()` довольно проста. Базовый случай, то есть обстоятельства, при которых рекурсивные вызовы прекращают выполняться, имеет место, когда значение `nthNumber` равно `1` или `2` ❶. В этом случае функция возвращает значение `1`, поскольку первым и вторым числом ряда Фибоначчи всегда является единица. Любой другой случай — рекурсивный, поэтому возвращаемое значение представляет собой сумму чисел `fibonacci(nthNumber - 1)` и `fibonacci(nthNumber - 2)`. Пока исходный аргумент `nthNumber` — целое число больше нуля, эти рекурсивные вызовы будут совершаться вплоть до достижения базового случая.

Как вы помните, в примере с рекурсивным алгоритмом вычисления факториала одна часть строки кода выполнялась до рекурсивного вызова, а другая — после него. Поскольку рекурсивный алгоритм Фибоначчи предусматривает два рекурсивных вызова в своем рекурсивном случае, необходимо иметь в виду, что код состоит из трех частей: «перед первым рекурсивным вызовом», «после первого рекурсивного вызова, но перед вторым» и «после второго рекурсивного вызова». Однако принцип тот же. И не думайте, что достижение базового случая после любого из рекурсивных вызовов означает окончание выполнения кода. Рекурсивный алгоритм завершает работу только после возврата исходного вызова функции.

Может показаться, что итеративный алгоритм вычисления чисел Фибоначчи использовать гораздо проще, чем рекурсивный. Да, так и есть. Более того, рекурсивный алгоритм имеет критический недостаток, о котором мы поговорим в следующем разделе.

Чем плох рекурсивный алгоритм вычисления чисел Фибоначчи

Он, как и рекурсивный алгоритм вычисления факториала, имеет критический недостаток в виде многократного повторения одних и тех же вычислений. На рис. 2.3 показано, как вызов функции `fibonacci(6)`, отмеченной на древовидной диаграмме как `fib(6)`, вызывает функции `fibonacci(5)` и `fibonacci(4)`.

Это порождает каскад вызовов других функций, которые выполняются вплоть до достижения базовых случаев `fibonacci(2)` и `fibonacci(1)`, возвращающих значение `1`. Однако обратите внимание, что функция `fibonacci(4)` вызывается

дважды, `fibonacci(3)` — трижды и т. д. Эти лишние вычисления замедляют работу алгоритма, а эффективность падает по мере увеличения числа Фибоначчи, которое вы хотите вычислить. Если итеративный алгоритм способен найти число `fibonacci(100)` менее чем за секунду, то рекурсивному алгоритму на это потребуется более миллиона лет.

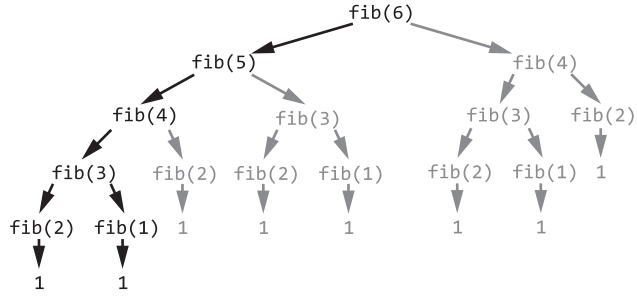


Рис. 2.3. Древовидная диаграмма рекурсивных вызовов функций, начиная с fibonacci(6). Избыточные вызовы выделены серым цветом

Преобразование рекурсивного алгоритма в итеративный

Рекурсивный алгоритм всегда можно преобразовать в итеративный. Вместо многократного вызова рекурсивных функций для повторяющихся вычислений лучше использовать цикл. Стек вызовов, используемый рекурсивными функциями, в итеративном алгоритме можно заменить стековой структурой данных. Таким образом, любой рекурсивный алгоритм может быть преобразован в итеративный с помощью цикла и стека.

Продемонстрируем это на примере программы на языке Python `factorialEmulateRecursion.py`, которая реализует итеративный алгоритм для эмуляции рекурсивного:

```
callStack = [] # Явный стек вызовов, содержащий кадры ❶
callStack.append({'returnAddr': 'start', 'number': 5})
# Вызов функции factorial() ❷
returnValue = None

while len(callStack) > 0:
    # Тело "функции factorial()":
    number = callStack[-1]['number'] # Задание числового параметра
    returnAddr = callStack[-1]['returnAddr']
```

```
    if returnAddr == 'start':
        if number == 1:
            # БАЗОВЫЙ СЛУЧАЙ
            returnValue = 1
            callStack.pop() # Возврат из вызова функции ❸
            continue
        else:
            # РЕКУРСИВНЫЙ СЛУЧАЙ
            callStack[-1]['returnAddr'] = 'after recursive call'
            # Вызов функции factorial():
            callStack.append({'returnAddr': 'start', 'number': number - 1}) ❹
            continue
    elif returnAddr == 'after recursive call':
        returnValue = number * returnValue
        callStack.pop() # Возврат из вызова функции ❺
        continue

print(returnValue)
```

Программа `factorialEmulateRecursion.html` содержит эквивалентный код на языке JavaScript:

```
<script type="text/javascript">
let callStack = []; // Явный стек вызовов, содержащий кадры ❶
callStack.push({"returnAddr": "start", "number": 5});
// Вызов функции factorial() ❷
let returnValue;

while (callStack.length > 0) {
// Тело функции factorial():
    let number = callStack[callStack.length - 1]["number"];
    // Задание числового параметра
    let returnAddr = callStack[callStack.length - 1]["returnAddr"];

    if (returnAddr == "start") {
        if (number === 1) {
            // БАЗОВЫЙ СЛУЧАЙ
            returnValue = 1;
            callStack.pop(); // Возврат из вызова функции ❸
            continue;
        } else {
            // РЕКУРСИВНЫЙ СЛУЧАЙ
            callStack[callStack.length - 1]["returnAddr"] =
              "after recursive call";
            // Вызов функции factorial():
            callStack.push({"returnAddr": "start", "number": number - 1}); ❹
            continue;
        }
    } else if (returnAddr == "after recursive call") {
        returnValue = number * returnValue;
        callStack.pop(); // Возврат из вызова функции ❺
```

```
        continue;
    }
}

document.write(returnValue + "<br />");
</script>
```

Обратите внимание, что код не содержит рекурсивной функции — в нем вообще нет никаких функций! Программа эмулирует рекурсивные вызовы, используя в качестве стековой структуры данных список (хранящийся в переменной `callStack` ❶) для имитации стека вызовов. Словарь, хранящий адрес возврата и локальную переменную `nthNumber`, эмулирует кадр ❷. Для эмуляции вызовов функций программа помещает эти кадры в стек вызовов ❹, а для эмуляции возврата из вызова функции — извлекает кадры из этого стека ❸❺.

Таким итеративным способом можно записать любую рекурсивную функцию. Несмотря на то что представленный код невероятно сложен для понимания и вы никогда не стали бы писать таким способом реальный алгоритм для вычисления факториала, он демонстрирует, что рекурсивный код не обладает никакими особыми возможностями, которых не было бы у итеративного.

Преобразование итеративного алгоритма в рекурсивный

Итеративный алгоритм также можно преобразовать в рекурсивный. Итеративный алгоритм — это просто код, использующий цикл. Многократно выполняемый код (тело цикла) может быть прописан внутри рекурсивной функции, которую затем можно неоднократно вызывать из нее самой для выполнения содержащегося в ней кода.

Программа на языке Python в файле `hello.py` пять раз выводит на экран фразу `Hello, world!`, используя для этого цикл, а затем рекурсивную функцию:

```
print('Code in a loop:')
i = 0
while i < 5:
    print(i, 'Hello, world!')
    i = i + 1

print('Code in a function:')
def hello(i=0):
    print(i, 'Hello, world!')
    i = i + 1
    if i < 5:
        hello(i) # РЕКУРСИВНЫЙ СЛУЧАЙ
     else:
        return # БАЗОВЫЙ СЛУЧАЙ
hello()
```

Эквивалентный код на языке JavaScript находится в файле `hello.html`:

```
<script type="text/javascript">
document.write("Code in a loop:<br />");
let i = 0;
while (i < 5) {
    document.write(i + " Hello, world!<br />");
    i = i + 1;
}

document.write("Code in a function:<br />");
function hello(i) {
    if (i === undefined) {
        i = 0; // Если значение i не задано, по умолчанию
               // используется 0
    }

    document.write(i + " Hello, world!<br />");
    i = i + 1;
    if (i < 5) {
        hello(i); // РЕКУРСИВНЫЙ СЛУЧАЙ
    }
    else {
        return; // БАЗОВЫЙ СЛУЧАЙ
    }
}
hello();
</script>
```

Результат выполнения этих программ выглядит следующим образом:

```
Code in a loop:
0 Hello, world!
1 Hello, world!
2 Hello, world!
3 Hello, world!
4 Hello, world!
Code in a function:
0 Hello, world!
1 Hello, world!
2 Hello, world!
3 Hello, world!
4 Hello, world!
```

Цикл `while` содержит условие `i < 5`, которое определяет, до какого момента он будет выполняться. Точно так же рекурсивная функция использует это условие в своем рекурсивном случае, что заставляет данную функцию вызывать саму себя и выполнять код, выводящий на экран фразу `Hello, world!`.

Для наглядности ниже приведены итеративные и рекурсивные функции, которые возвращают индекс подстроки `needle` (игла) в строке `haystack` (стог сена).

Если подстрока не найдена, функции возвращают значение `-1`. Это похоже на строковый метод `find()` в Python и `indexOf()` в JavaScript. Подобная программа на языке Python содержится в файле `findSubstring.py`:

```
def findSubstringIterative(needle, haystack):
    i = 0
    while i < len(haystack):
        if haystack[i:i + len(needle)] == needle:
            return i # Игла найдена
        i = i + 1
    return -1 # Игла не найдена

def findSubstringRecursive(needle, haystack, i=0):
    if i >= len(haystack):
        return -1 # БАЗОВЫЙ СЛУЧАЙ (игла не найдена)

    if haystack[i:i + len(needle)] == needle:
        return i # БАЗОВЫЙ СЛУЧАЙ (игла найдена)
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return findSubstringRecursive(needle, haystack, i + 1)

print(findSubstringIterative('cat', 'My cat Zophie'))
print(findSubstringRecursive('cat', 'My cat Zophie'))
```

Эквивалентная версия программы на языке JavaScript находится в файле `findSubstring.html`:

```
<script type="text/javascript">
function findSubstringIterative(needle, haystack) {
    let i = 0;
    while (i < haystack.length) {
        if (haystack.substring(i, i + needle.length) == needle) {
            return i; // Игла найдена
        }
        i = i + 1
    }
    return -1; // Игла не найдена
}

function findSubstringRecursive(needle, haystack, i) {
    if (i === undefined) {
        i = 0;
    }
    if (i >= haystack.length) {
        return -1; // # БАЗОВЫЙ СЛУЧАЙ (игла не найдена)
    }

    if (haystack.substring(i, i + needle.length) == needle) {
        return i; // # БАЗОВЫЙ СЛУЧАЙ (игла найдена)
    } else {
```

```
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return findSubstringRecursive(needle, haystack, i + 1);
    }}

document.write(findSubstringIterative("cat", "My cat Zophie") + "<br />");
document.write(findSubstringRecursive("cat", "My cat Zophie") + "<br />");
</script>
```

Эти программы вызывают методы `findSubstringIterative()` и `findSubstringRecursive()`, которые возвращают значение `3`, соответствующее индексу первого символа слова `cat` в строке `My cat Zophie`:

```
3
3
```

Приведенный в текущем разделе код демонстрирует возможность преобразования любого цикла в эквивалентную рекурсивную функцию. Однако я не советую этого делать. Рекурсия ради рекурсии только усложнит ваш код, что абсолютно бессмысленно.

Практический пример: вычисление экспоненты

Несмотря на то что рекурсия не всегда позволяет получить более качественный код, ее использование может натолкнуть вас на новые идеи решения стоящей перед вами проблемы. В качестве примера рассмотрим способ расчета экспоненты.

Экспонента вычисляется путем умножения числа самого на себя. Например, три в шестой степени, или 3^6, равно 729, то есть $3 \times 3 \times 3 \times 3 \times 3 \times 3$. Возведение числа в степень — настолько распространенная операция, что в языке Python для нее предусмотрен специальный оператор `**`, а в JavaScript — встроенная функция `Math.pow()`. Для вычисления 3^6 можно использовать код Python `3 ** 6` и код JavaScript `Math.pow(3,6)`.

Напишем собственную программу для вычисления экспоненты: достаточно создать цикл, который многократно умножает число само на себя и возвращает конечный результат. Вот итеративная программа на языке Python, содержащаяся в файле `exponentByIteration.py`:

```
def exponentByIteration(a, n):
    result = 1
    for i in range(n):
        result *= a
    return result

print(exponentByIteration(3, 6))
print(exponentByIteration(10, 3))
print(exponentByIteration(17, 10))
```

А ниже эквивалентная программа `exponentByIteration.html` на языке JavaScript:

```
<script type="text/javascript">
function exponentByIteration(a, n) {
    let result = 1;
    for (let i = 0; i < n; i++) {
         result *= a;
    }
    return result;
}

document.write(exponentByIteration(3, 6) + "<br />");
document.write(exponentByIteration(10, 3) + "<br />");
document.write(exponentByIteration(17, 10) + "<br />");
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
729
1000
2015993900449
```

Это простое вычисление можно легко выполнить с помощью цикла, хотя и он не лишен недостатков. Работа функции замедляется по мере увеличения показателя степени: вычисление 3^{12} занимает в два раза больше времени, чем 3^6, а 3^{600} рассчитывается в сто раз дольше, чем 3^6. В следующем разделе мы попробуем решить эту проблему, используя рекурсивный подход.

Создание рекурсивной функции для вычисления экспоненты

Давайте подумаем, как с помощью рекурсии решить задачу возведения числа в степень, скажем, 3^6. В силу сочетательного свойства операции умножения выражение $3 \times 3 \times 3 \times 3 \times 3 \times 3$ равнозначно $(3 \times 3 \times 3) \times (3 \times 3 \times 3)$ или $(3 \times 3 \times 3)^2$. А поскольку $(3 \times 3 \times 3)$ то же самое, что и 3^3, то 3^6 можно представить как $(3^3)^2$. В математике это называется *возведением степени в степень*: $(a^m)^n = a^{mn}$. При *умножении степеней с одинаковыми основаниями* действует следующее правило: $a^n \times a^m = a^{n+m}$ (в том числе $a^n \times a = a^{n+1}$).

Используем эти математические правила для создания функции `exponentByRecursion()`. Вызов `exponentByRecursion(3, 6)` равнозначен выражению `exponentByRecursion(3, 3) * exponentByRecursion(3, 3)`. Разумеется, нам не нужно дважды вызывать `exponentByRecursion(3, 3)`. Вместо этого мы сохраним возвращенное ею значение в переменной и умножим его само на себя.

Такой подход работает для четных показателей, а как насчет нечетных? Если бы нам потребовалось посчитать 3^7, или $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$, это было бы равнозначно вычислению $(3 \times 3 \times 3 \times 3 \times 3 \times 3) \times 3$ или $(3^6) \times 3$. А для определения 3^6 мы можем использовать тот же рекурсивный вызов.

ПРИМЕЧАНИЕ

Для определения четности или нечетности целого числа в программировании используется оператор деления по модулю (%). Результатом деления любого четного целого числа по модулю 2 является 0, а нечетного — 1.

Мы рассмотрели рекурсивные случаи, а что насчет базовых? В математике n^0 равно единице, а n^1 равно n. Таким образом, после любого вызова функции `exponentByRecursion(a, n)`, если `n` равно `0` или `1`, мы можем возвратить значение `1` или `a` соответственно, потому что a^0 всегда равно `1`, а a^1 всегда равно `a`.

Принимая во внимание вышесказанное, напишем код для функции `exponentByRecursion()`. На языке Python (`exponentByRecursion.py`) он будет выглядеть следующим образом:

```
def exponentByRecursion(a, n):
    if n == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        return a
    elif n % 2 == 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ (когда число n четное)
        result = exponentByRecursion(a, n // 2)
        return result * result
    elif n % 2 == 1:
        # РЕКУРСИВНЫЙ СЛУЧАЙ (когда число n нечетное)
        result = exponentByRecursion(a, n // 2)
        return result * result * a

print(exponentByRecursion(3, 6))
print(exponentByRecursion(10, 3))
print(exponentByRecursion(17, 10))
```

А это эквивалентный код на языке JavaScript (`exponentByRecursion.html`):

```
<script type="text/javascript">
function exponentByRecursion(a, n) {
    if (n === 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        return a;
    } else if (n % 2 === 0) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ (когда число n четное)
        result = exponentByRecursion(a, n / 2);
        return result * result;
    } else if (n % 2 === 1) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ (когда число n нечетное)
        result = exponentByRecursion(a, Math.floor(n / 2));
        return result * result * a;
    }
}
```

```
document.write(exponentByRecursion(3, 6));
document.write(exponentByRecursion(10, 3));
document.write(exponentByRecursion(17, 10));
</script>
```

Результат выполнения данного кода идентичен результату выполнения его итеративной версии:

```
729
1000
2015993900449
```

По сути, каждый рекурсивный вызов сокращает размер проблемы вдвое. Именно поэтому наш рекурсивный алгоритм вычисления экспоненты работает быстрее, чем его итеративная версия. Вычисление 3^{1000} итеративным способом предполагает выполнение 1000 операций умножения, в то время как рекурсивному алгоритму для этого достаточно 23 операций умножения и деления. При запуске кода Python в профилировщике производительности 100 000-кратное итеративное вычисление 3^{1000} занимает 10,633 секунды, а рекурсивное — всего 0,406 секунды. Это весьма впечатляющая разница!

Создание итеративной функции для вычисления экспоненты на основе рекурсивного подхода

Исходная итеративная функция для вычисления экспоненты предполагала использование цикла, чтобы умножить число само на себя необходимое количество раз в соответствии с показателем степени. Однако данный подход не очень хорошо масштабируется. Рекурсивная реализация показала нам, что разбиение задачи на более мелкие подзадачи гораздо эффективнее.

Поскольку у каждого рекурсивного алгоритма есть альтернатива в виде итеративной функции, мы можем создать ее для вычисления экспоненты по тем же правилам возведения степени в степень, что и в рекурсивном алгоритме. Следующая программа, `exponentWithPowerRule.py`, содержит такую функцию:

```
def exponentWithPowerRule(a, n):
    # Этап 1: определение операций, которые предстоит выполнить
    opStack = []
    while n > 1:
        if n % 2 == 0:
            # Число n четное
            opStack.append('square')
            n = n // 2
        elif n % 2 == 1:
            # Число n нечетное
            n -= 1
            opStack.append('multiply')
```

```
    # Этап 2: выполнение операций в обратном порядке
    result = a # Задание для result начального значения 'a'
    while opStack:
        op = opStack.pop()

        if op == 'multiply':
            result *= a
        elif op == 'square':
            result *= result

    return result

print(exponentWithPowerRule(3, 6))
print(exponentWithPowerRule(10, 3))
print(exponentWithPowerRule(17, 10))
```

Эквивалентный код на языке JavaScript содержится в файле `exponentWithPowerRule.html`:

```
<script type="text/javascript">
function exponentWithPowerRule(a, n) {
    // Этап 1: определение операций, которые предстоит выполнить
    let opStack = [];
    while (n > 1) {
        if (n % 2 === 0) {
            // Число n — четное
            opStack.push("square");
            n = Math.floor(n / 2);
        } else if (n % 2 === 1) {
            // Число n — нечетное
            n -= 1;
            opStack.push("multiply");
        }
    }
    // Этап 2: выполнение операций в обратном порядке
    let result = a; // Задание для result начального значения 'a'
    while (opStack.length > 0) {
        let op = opStack.pop();

        if (op === "multiply") {
            result = result * a;
        } else if (op === "square") {
            result = result * result;
        }
    }

    return result;
}

document.write(exponentWithPowerRule(3, 6) + "<br />");
document.write(exponentWithPowerRule(10, 3) + "<br />");
document.write(exponentWithPowerRule(17, 10) + "<br />");
</script>
```

Наш алгоритм последовательно уменьшает n, деля его пополам (если оно четное) или вычитая из него единицу (если оно нечетное), пока оно не станет равно 1, что позволяет нам выполнить операции возведения в квадрат или умножения на a. После завершения этой стадии совершаем указанные операции в обратном порядке. Здесь удобнее всего использовать универсальный стек (отдельный от стека вызовов), представляющий собой структуру данных типа «первым пришел — последним ушел». Сначала операции возведения в квадрат или умножения на a помещаются в стек в переменной opStack. Затем они выполняются в порядке их извлечения из стека.

Например, вызов функции exponentWithPowerRule(6, 5) для вычисления 6^5 задает для a значение 6, а для n — 5. Если число n нечетное, то необходимо вычесть 1 из n, чтобы получить 4, и поместить в opStack операцию умножения на a. Теперь, когда n равно 4 (то есть является четным), мы делим его на 2, чтобы получить 2, и помещаем в opStack операцию возведения в квадрат. Поскольку n четное и равно 2, снова делим его пополам и помещаем в opStack еще одну операцию возведения в квадрат. Получив для n значение 1, первый этап можно считать завершенным.

На втором этапе задаем для переменной result начальное значение a (которое в нашем примере равно 6). Далее мы выталкиваем верхний элемент из стека opStack, которым является операция возведения в квадрат, сообщая тем самым программе о необходимости присвоить переменной result значение result * result (то есть $result^2$), или 36. Затем следует извлечение из opStack еще одной операции возведения в квадрат, поэтому программа изменяет result с 36 на 36 * 36, или 1296. Наконец, мы извлекаем из opStack последнюю операцию — умножение 1296 на a (которое равно 6), получая 7776. Поскольку в opStack больше нет операций, функция завершает свою работу. Перепроверив расчеты, убеждаемся в том, что 6^5 действительно равно 7776.

Изменение состояния стека opStack по мере выполнения вызовов функции WithPowerRule(6, 5) показано на рис. 2.4.

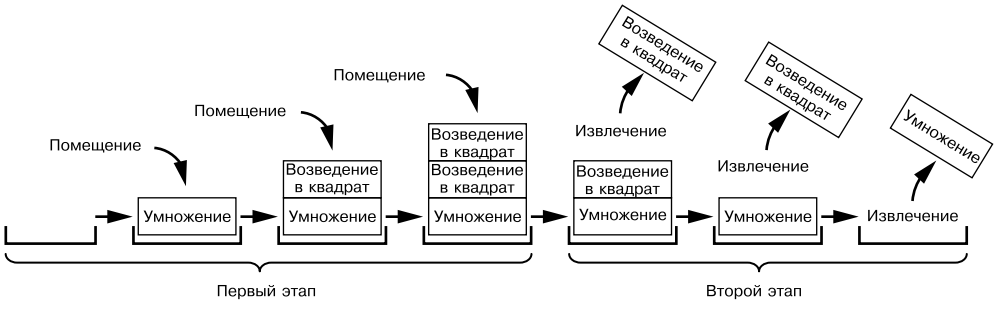


Рис. 2.4. Изменение состояния стека opStack по мере выполнения вызовов функции WithPowerRule(6, 5)

Результат выполнения этого кода идентичен выводу других программ для вычисления экспоненты:

```
729
1000
2015993900449
```

Итеративная функция для вычисления экспоненты, использующая правило возведения степени в степень, по производительности превосходит рекурсивный алгоритм и при этом не переполняет стек. Однако без использования рекурсивного подхода мы, возможно, не сумели бы прийти к подобному итеративному алгоритму.

Когда нужно использовать рекурсию

Вам не *нужно* использовать рекурсию. Ни одна задача программирования не *требует* ее применения. В этой главе я постараюсь продемонстрировать, что рекурсия не обладает магической силой и не позволяет сделать ничего такого, чего не смог бы итеративный код с циклом и стековой структурой данных. На самом деле рекурсивный подход даже избыточен для стоящей перед вами задачи.

На примере функции для вычисления экспоненты из предыдущего раздела мы увидели, что рекурсия позволяет по-новому решать стоящие перед нами задачи программирования. Рекурсивный подход особенно полезен, если задача имеет следующие три особенности:

- у нее древовидная структура;
- она предусматривает поиск с возвратом;
- она не предполагает слишком большую глубину рекурсии, способную спровоцировать переполнение стека.

Структура дерева обладает свойством *самоподобия*: точки ветвления напоминают корень меньшего поддерева. Рекурсия часто имеет дело с самоподобием и задачами, которые можно разделить на более мелкие. Корень дерева аналогичен первому вызову рекурсивной функции, точки ветвления — рекурсивным случаям, а листья — базовым случаям, в которых рекурсивные вызовы перестают выполняться.

Лабиринт — это еще один хороший пример задачи, имеющей древовидную структуру и предусматривающей поиск с возвратом. В лабиринте развилки соответствуют точкам ветвления, а тупик — базовому случаю, по достижении которого вы должны вернуться к предыдущей точке ветвления, чтобы выбрать другой путь.

На рис. 2.5 показан путь в лабиринте, нарисованный так, чтобы визуально напоминать дерево. Несмотря на видимую разницу между путями в лабиринте и «ветвями»

дерева, их точки ветвления связаны друг с другом одинаковым образом. С математической точки зрения эти графики эквивалентны.

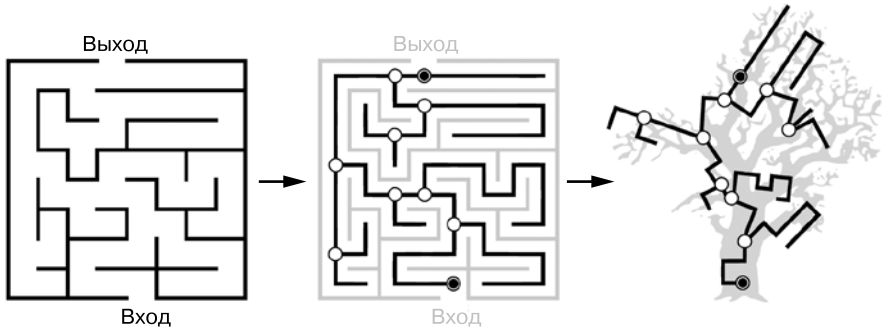


Рис. 2.5. Лабиринт *(слева)* и пути его прохождения *(в центре)*, преобразованные в форму дерева *(справа)*

Многие объекты обладают подобной древовидной структурой. Например, файловая система: вложенные каталоги напоминают корневые папки файловой системы меньшего размера (рис. 2.6).

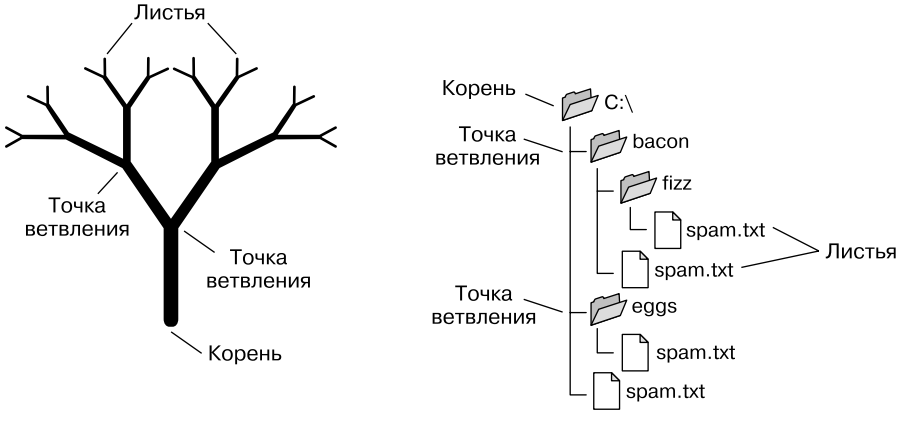


Рис. 2.6. Структура файловой системы напоминает дерево

Поиск определенного имени файла в папке представляет собой рекурсивную задачу: сначала вы ищете в папке, а затем во вложенных каталогах. Если папка не содержит подпапок, это базовый случай, который приводит к остановке рекурсивного поиска.

Если ваш рекурсивный алгоритм не находит нужного файла, он возвращается к предыдущему родительскому каталогу и продолжает поиск оттуда.

Третий момент из перечисленных выше задач, подходящих для рекурсии, — это практичность. Если ваша древовидная структура предусматривает так много уровней ветвей, что рекурсивная функция грозит вызвать переполнение стека до того, как достигнет листьев, то использование рекурсии — неподходящее решение.

С другой стороны, рекурсия отлично подходит для создания компиляторов языков программирования. Проектирование компиляторов — это отдельная обширная тема, обсуждение которой выходит за рамки книги. Отмечу лишь, что языки программирования предусматривают набор правил, позволяющих преобразовать исходный код в древовидную структуру, подобно тому как грамматические правила обычного языка позволяют преобразовать в древовидную диаграмму обыкновенное предложение. Рекурсия идеально подходит для разработки компиляторов.

В книге мы рассмотрим множество рекурсивных алгоритмов, которые часто представляют собой древовидную структуру или функции поиска с возвратом, хорошо сочетающиеся с рекурсией.

Создание собственных рекурсивных алгоритмов

Будем надеяться, что в результате прочтения этой главы у вас сложилось четкое представление, чем рекурсивные функции отличаются от итеративных алгоритмов, с которыми вы, вероятно, знакомы гораздо лучше. Оставшаяся часть издания посвящена детальному рассмотрению различных рекурсивных алгоритмов. Но с чего начать?

Первым делом всегда определяются рекурсивный и базовый случаи. Здесь можно применить нисходящий подход: разбить задачу на похожие, но менее масштабные подзадачи, соответствующие *рекурсивным случаям*. Затем подумайте, при каких условиях подзадачи станут настолько малы, что их решения окажутся тривиальными. Это будут ваши *базовые случаи*. Ваша рекурсивная функция может предусматривать несколько рекурсивных или базовых случаев, однако она неизменно будет иметь по крайней мере по одному из этих случаев.

Наглядным примером служит рекурсивный алгоритм вычисления последовательности Фибоначчи. Каждое число в ней представляет собой сумму двух предыдущих. Допускается разбить задачу определения числа Фибоначчи на подзадачи, связанные с нахождением двух предыдущих значений последовательности. Известно, что

первые два числа Фибоначчи равны 1, и это дает нам ответ для базового случая, соответствующего минимальному масштабу подзадачи.

Иногда полезно применить восходящий подход и сперва рассмотреть базовый случай, а уже затем формулировать и решать более масштабные задачи. Отличный пример — рекурсивный алгоритм вычисления факториала. Факториал 1! равен 1. Это базовый случай. Следующий факториал 2! рассчитывается путем умножения 1! на 2. Факториал 3! — путем умножения 2! на 3 и т. д. Эта общая закономерность позволяет нам понять, каким должен быть рекурсивный случай для рассматриваемого алгоритма.

Резюме

В этой главе мы рассмотрели процесс вычисления факториалов и последовательности Фибоначчи — две классические задачи рекурсивного программирования. Для расчетов мы использовали как итеративные, так и рекурсивные реализации соответствующих алгоритмов. Хотя это и стандартные примеры применения рекурсии, исследованные нами алгоритмы имеют серьезные недостатки: при вычислении факториалов есть риск вызвать переполнение стека, а при попытке найти числа Фибоначчи затрачивается настолько много ресурсов, что функция оказывается слишком медленной и неэффективной для реальных задач.

Мы также изучили процесс преобразования рекурсивных алгоритмов в итеративные и наоборот и выяснили, что любой рекурсивный алгоритм можно преобразовать в итеративный при помощи цикла и стековой структуры данных. Рекурсия часто оказывается слишком сложным решением, однако она хорошо подходит для задач программирования, предусматривающих древовидную структуру и поиск с возвратом.

Написание рекурсивных функций — это навык, который совершенствуется с практикой и опытом. Оставшаяся часть книги посвящена нескольким хорошо известным примерам рекурсии, а также исследованию их сильных сторон и накладываемых ограничений.

Дополнительные источники информации

Дополнительную информацию о разнице между итерацией и рекурсией вы найдете в видео *Programming Loops vs. Recursion* на YouTube-канале Computerphile (https://youtu.be/HXNhEYqFo0o). Если же вы хотите сравнить производительность итеративных и рекурсивных функций, необходимо научиться пользоваться профилировщиком. Информация о профилировщиках для программ на языке Python есть в главе 13 моей книги «Python. Чистый код для продолжающих»[1] (оригинальный

[1] *Свейгарт Э.* Python. Чистый код для продолжающих. — Питер, 2022.

текст доступен по адресу https://inventwithpython.com/beyond/chapter13.html). Кроме того, о профилировщиках можно почитать в официальной документации по языку Python: https://docs.python.org/3/library/profile.html. Профилировщик Firefox для JavaScript описан на сайте Mozilla (https://developer.mozilla.org/en-US/docs/Tools/Performance). Другие браузеры предусматривают аналогичные инструменты.

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое 4! (то есть факториал числа 4)?
2. Как можно использовать факториал числа ($n - 1$) для вычисления факториала n?
3. В чем состоит критический недостаток рекурсивной функции для вычисления факториала?
4. Каковы первые пять чисел последовательности Фибоначчи?
5. Какие два числа необходимо сложить, чтобы получить n-е число Фибоначчи?
6. В чем заключается критический недостаток рекурсивной функции для вычисления последовательности Фибоначчи?
7. Что всегда используется в итеративном алгоритме?
8. Всегда ли можно преобразовать итеративный алгоритм в рекурсивный?
9. Всегда ли можно преобразовать рекурсивный алгоритм в итеративный?
10. С помощью каких двух вещей рекурсивный алгоритм может быть выполнен итеративно?
11. Какими тремя особенностями должны обладать задачи программирования для применения рекурсивных решений?
12. В каких случаях уместно применять рекурсивные методы?

Практика

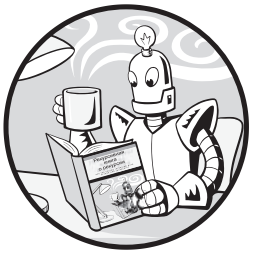
Для того чтобы попрактиковаться, попробуйте решить следующие задачи.

1. Вычислите сумму целых чисел от `1` до `n` с помощью итеративного цикла. Функция для решения данной задачи похожа на функцию `factorial()`, только вместо умножения она выполняет сложение. Например, `sumSeries(1)` возвращает `1`, `sumSeries(2)` возвращает `3` (то есть 1 + 2), `sumSeries(3)` — `6` (то есть 1 + 2 + 3) и т. д. В качестве примера вы можете ориентироваться на программу `factorialByIteration.py`.

2. Напишите рекурсивную версию функции `sumSeries()`. Вместо цикла эта функция должна использовать рекурсивные вызовы. В качестве руководства используйте программу `factorialByRecursion.py`.
3. Итеративно вычислите сумму первых `n` степеней числа 2 с помощью функции с именем `sumPowersOf2()`. Степенями числа 2 являются 2, 4, 8, 16, 32 и т. д. В Python они вычисляются так: `2 ** 1`, `2 ** 2`, `2 ** 3`, `2 ** 4`, `2 ** 5`... соответственно. В JavaScript для этого применяется функция `Math.pow(2, 1)`, `Math.pow(2, 2)` и т. д. Например, `sumPowersOf2(1)` возвращает `2`, `sumPowersOf2(2)` возвращает `6` (то есть 2 + 4), `sumPowersOf2(3)` — `14` (то есть `2 + 4 + 8`) и т. д.
4. Напишите рекурсивную версию функции `sumPowersOf2()`, используя рекурсивные вызовы вместо цикла.

3

Классические рекурсивные алгоритмы



Если вы проходите курс по информатике, то наверняка уже обратили внимание, что модуль, посвященный рекурсии, охватывает некоторые из классических алгоритмов, представленных в этой главе. Их также могут попросить использовать на профильных собеседованиях. В этой главе мы рассмотрим шесть классических рекурсивных задач и их решения.

Начнем с трех простых алгоритмов: суммирования чисел в массиве, обращения текстовой строки и определения строки-палиндрома. Затем рассмотрим алгоритм решения головоломки «Ханойская башня», реализуем метод заливки и обсудим функцию Аккермана.

Вы узнаете о технике «голова — хвост», позволяющей разделять данные в аргументах рекурсивной функции. Кроме того, при поиске рекурсивных решений мы сформулируем три вопроса: что представляет собой базовый случай? Какой аргумент передается рекурсивной функции при ее вызове? И как аргументы, передаваемые рекурсивной функции, приближаются к базовому случаю? Чем опытнее вы становитесь, тем легче будет отвечать на эти вопросы.

Суммирование чисел в массиве

Первая задача довольно проста: у нас есть список (в Python) или массив (в JavaScript) целых чисел, сумму которых нужно возвратить. Например, вызов функции `sum([5, 2, 4, 8])` должен возвратить значение `19`.

Такую задачу легко решить с помощью цикла, в то время как рекурсивный метод требует более тщательного анализа. Если вы внимательно прочитали вторую главу,

то могли заметить, что подобный алгоритм не соответствует критериям, оправдывающим использование рекурсии и связанную с этим дополнительную сложность. Тем не менее суммирование чисел в массиве (или любое другое вычисление, связанное с обработкой данных в линейной структуре) достаточно распространенная рекурсивная задача, предлагаемая в ходе собеседований, поэтому заслуживает нашего внимания.

Чтобы ее решить, рассмотрим метод реализации рекурсивных функций под названием *«голова — хвост»*, который разбивает аргумент рекурсивной функции в виде массива на две части: *голову* (первый элемент массива) и *хвост* (новый массив, включающий все, что следует после первого элемента). Определяем рекурсивную функцию `sum()` для сложения целых чисел в массиве путем прибавления головы к сумме чисел хвоста. Чтобы узнать сумму чисел хвостового массива, мы рекурсивно передаем его в качестве аргумента функции `sum()`.

Поскольку хвостовой массив содержит на один элемент меньше, чем исходный, рано или поздно при вызове рекурсивной функции будет передан пустой массив. Операция сложения элементов пустого массива довольно тривиальна и не требует дополнительных рекурсивных вызовов, а ее результат равен `0`. Исходя из этого, можем дать следующие ответы на три ранее заданных вопроса.

Что представляет собой базовый случай? Пустой массив, сумма элементов которого равна `0`.

Какой аргумент передается рекурсивной функции при ее вызове? Хвост первоначального массива, количество элементов в котором на один меньше, чем в исходном.

Как этот аргумент приближается к базовому случаю? Передаваемый в качестве аргумента массив уменьшается на один элемент при каждом рекурсивном вызове до тех пор, пока не превратится в массив нулевой длины, то есть пока не станет пустым.

Ниже представлена программа на языке Python для суммирования списка чисел, она содержится в файле `sumHeadTail.py`:

```
def sum(numbers):
    if len(numbers) == 0: # БАЗОВЫЙ СЛУЧАЙ
      ❶ return 0
    else: # РЕКУРСИВНЫЙ СЛУЧАЙ
      ❷ head = numbers[0]
      ❸ tail = numbers[1:]
      ❹ return head + sum(tail)

nums = [1, 2, 3, 4, 5]
print('The sum of', nums, 'is', sum(nums))
nums = [5, 2, 4, 8]
print('The sum of', nums, 'is', sum(nums))
nums = [1, 10, 100, 1000]
print('The sum of', nums, 'is', sum(nums))
```

А вот аналогичная программа на языке JavaScript из файла `sumHeadTail.html`:

```
<script type="text/javascript">
function sum(numbers) {
    if (numbers.length === 0) { // БАЗОВЫЙ СЛУЧАЙ
     ❶ return 0;
    } else { // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ let head = numbers[0];
     ❸ let tail = numbers.slice(1, numbers.length);
     ❹ return head + sum(tail);
    }
}
let nums = [1, 2, 3, 4, 5];
document.write('The sum of ' + nums + ' is ' + sum(nums) + "<br />");
nums = [5, 2, 4, 8];
document.write('The sum of ' + nums + ' is ' + sum(nums) + "<br />");
nums = [1, 10, 100, 1000];
document.write('The sum of ' + nums + ' is ' + sum(nums) + "<br />");
</script>
```

Результат запуска этих программ выглядит так:

```
The sum of [1, 2, 3, 4, 5] is 15
The sum of [5, 2, 4, 8] is 19
The sum of [1, 10, 100, 1000] is 1111
```

В базовом случае во время вызова функции и передачи ей в качестве аргумента пустого массива она просто возвращает `0` ❶. В рамках рекурсивного случая мы делим исходный аргумент — массив `numbers` — на голову `head` ❷ и хвост `tail` ❸. Имейте в виду, что `tail` представляет собой массив, как и аргумент `numbers`, тогда как `head` — это всего лишь отдельное числовое значение, а не массив, содержащий один элемент. Значение, возвращаемое функцией `sum()`, также является отдельным числом, а не массивом. Именно поэтому в рекурсивном случае можно сложить `head` и `sum(tail)` ❹.

При каждом последующем рекурсивном вызове функции `sum()` передается массив, содержащий все меньшее количество чисел, что приближает его к базовому случаю, то есть к пустому массиву. Например, на рис. 3.1 показано состояние стека вызовов для функции `sum([5, 2, 4, 8])`.

Каждая карточка в стеке на рис. 3.1 соответствует вызову функции. В верхней части каждой карточки указано имя функции с аргументом, который был передан ей при вызове. Под ним находятся параметр `number` и локальные переменные `head` и `tail`, созданные во время вызова. Внизу — выражение `head + sum(tail)`, результат вычисления которого возвращает функция. При каждом рекурсивном вызове функции в стек помещается новая карточка. Когда функция возвращает значение, верхняя карточка извлекается из стека.

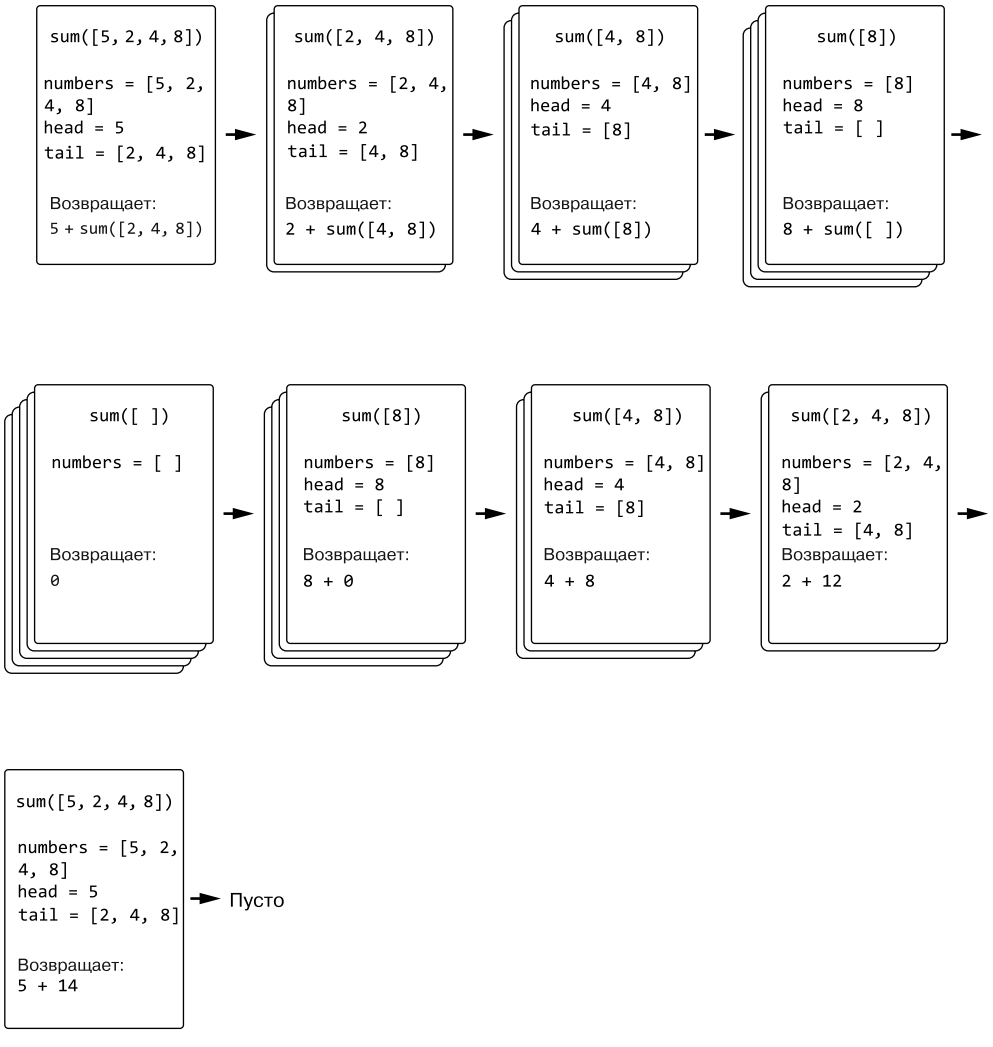


Рис. 3.1. Состояние стека вызовов для функции sum([5, 2, 4, 8])

Допускается использовать функцию `sum()` в качестве шаблона для применения метода «голова — хвост» к другим рекурсивным функциям. Например, мы можем изменить функцию `sum()`, которая складывает содержащиеся в массиве числа, на функцию `concat()`, которая объединяет содержащиеся в массиве строки. В базовом случае, когда аргументом выступает пустой массив, будет возвращена пустая строка. А в рекурсивном случае возвращается результат объединения головной строки со значением, возвращенным рекурсивной функцией, которой в качестве аргумента в момент вызова был передан хвост.

В главе 2 мы говорили, что рекурсия подходит для решения задач, предусматривающих древовидную структуру и поиск с возвратом. Массив, строку или другую линейную структуру данных можно рассматривать как дерево, из каждого узла которого выходит лишь одна ветвь, как показано на рис. 3.2.

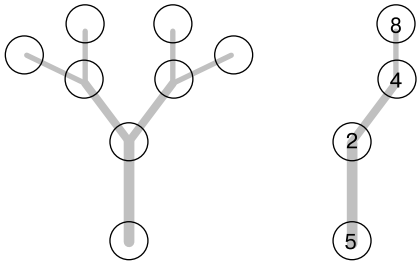


Рис. 3.2. Массив [5, 2, 4, 8] *(справа)* подобен древовидной структуре данных *(слева)*, имеющей только одну ветвь, выходящую из каждого узла

Ключевой момент, говорящий о ненужности такой рекурсивной функции, заключается в том, что при обработке данных она никогда не выполняет поиск с возвратом, а лишь последовательно обрабатывает все элементы массива, что можно сделать с помощью обычного цикла. Кроме того, рекурсивная функция суммирования Python работает примерно в 100 раз медленнее, чем простой итеративный алгоритм. Но даже если нас не волнует производительность, рекурсивная функция `sum()`, обрабатывающая список с десятками тысяч чисел, способна спровоцировать переполнение стека. Хотя рекурсия и представляет собой продвинутую технику, она не всегда является наилучшим решением.

В главе 5 мы рассмотрим рекурсивную функцию суммирования в контексте стратегии «разделяй и властвуй», а в главе 8 — функцию, использующую оптимизацию хвостовых вызовов. Эти альтернативные рекурсивные подходы позволяют решить некоторые проблемы, свойственные функции суммирования, описанной в текущей главе.

Обращение строки

Как и сложение чисел в массиве, обращение (реверс) строки — это еще один часто упоминаемый рекурсивный алгоритм, для которого существует простая итеративная версия. Поскольку строка по сути представляет собой массив отдельных символов, при создании функции `rev()` мы также будем использовать подход «голова — хвост».

Начнем с самых маленьких строк. Пустая и односимвольная строки изначально эквивалентны реверсивным версиям самих себя. Это базовые случаи: если

строковый аргумент — строка типа `''` или `'A'`, то функция должна просто возвратить этот аргумент.

Большие строки мы будем разделять на голову (первый символ) и хвост (все остальные). В случае двухсимвольной строки, такой как `'XY'`, `'X'` — это голова, а `'Y'` — хвост. Для обращения строки нужно поместить головную часть после хвостовой: `'YX'`.

Сработает ли этот алгоритм для более длинных строк? Чтобы обратить строку типа `'CAT'`, мы должны разбить ее на голову `'C'` и хвост `'AT'`. Однако если мы поместим головную часть после хвостовой, то вместо реверса исходной строки получим `'ATC'`. Что на самом деле нужно, так это поместить голову после *обращенного* хвоста. Другими словами, мы должны изменить хвост `'AT'` на `'TA'`, а затем поместить после него голову, чтобы получить ожидаемый результат — `'TAC'`.

Как можно обратить хвостовую часть строки? Необходимо рекурсивно вызывать функцию `rev()`, передавая ей хвост в качестве аргумента. Забудьте на мгновение о реализации этой функции и сосредоточьтесь на ее входных и выходных данных: `rev()` принимает один строковый аргумент и возвращает строку, состоящую из тех же символов, расположенных в обратном порядке.

Искать ответы на вопросы о том, как реализовать рекурсивную функцию наподобие `rev()`, непросто, поскольку этот процесс связан с проблемой курицы и яйца. Чтобы написать рекурсивный случай для `rev()`, нужно вызвать функцию, которая обращает строку, то есть `rev()`. Если у нас есть четкое представление относительно аргументов нашей рекурсивной функции и возвращаемого ею значения, мы можем использовать метод *прыжка веры*, чтобы обойти логический парадокс курицы и яйца, и написать рекурсивный случай, допустив правильность возвращаемого функцией `rev()` значения, несмотря на то что мы еще не завершили ее написание.

Прыжок веры в контексте рекурсии не какая-то волшебная техника, гарантирующая отсутствие ошибок в вашем коде. Это перспектива, которую следует держать в голове для преодоления ментальных блоков, возникающих у программиста при поиске способа реализации рекурсивной функции. Такой «прыжок» требует четкого понимания аргументов рекурсивной функции и возвращаемого ею значения.

Обратите внимание, что подобный метод помогает написать лишь рекурсивный случай. При вызове рекурсивной функции вы должны передать ей аргумент, приближенный к базовому случаю. У вас не получится просто передать тот же самый аргумент, который получила эта рекурсивная функция:

```
def rev(theString):
    return rev(theString) # Это не сработает.
```

Вернемся к примеру со строкой `'CAT'`. Когда мы передаем в функцию `rev()` хвост `'AT'`, в *данном конкретном* вызове головой является `'A'`, а хвостом — `'T'`. Вы уже знаете, что реверс односимвольной строки `'T'` — `'T'`: это наш базовый случай. Таким образом, второй вызов функции `rev()` приведет к изменению `'AT'` на `'TA'`, что нам и нужно. На рис. 3.3 показано состояние стека вызовов во время всех рекурсивных вызовов `rev()`.

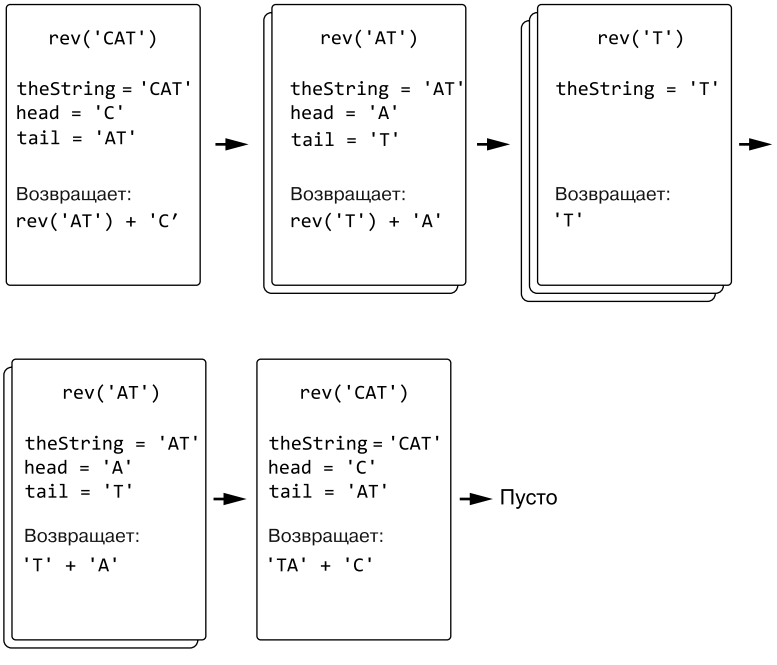


Рис. 3.3. Изменение состояния стека вызовов в процессе обращения строки CAT с помощью функции rev()

Давайте ответим на три уже известных нам вопроса по рекурсивным алгоритмам относительно функции `rev()`.

Что представляет собой базовый случай? Это пустая строка или строка, состоящая из одного символа.

Какой аргумент передается рекурсивной функции при ее вызове? Хвост исходного строкового аргумента, у которого на один символ меньше, чем у первоначального.

Как этот аргумент приближается к базовому случаю? Массив, передаваемый в качестве аргумента, уменьшается на один элемент после каждого рекурсивного вызова до тех пор, пока не станет пустым или односимвольным.

Вот как выглядит программа на языке Python из файла `reverseString.py`:

```
def rev(theString):
 ❶ if len(theString) == 0 or len(theString) == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        return theString
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ head = theString[0]
     ❸ tail = theString[1:]
     ❹ return rev(tail) + head

print(rev('abcdef'))
print(rev('Hello, world!'))
print(rev(''))
print(rev('X'))
```

А это ее эквивалент на языке JavaScript из файла `reverseString.html`:

```
<script type="text/javascript">
function rev(theString) {
 ❶ if (theString.length === 0 || theString.length === 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        return theString;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ var head = theString[0];
     ❸ var tail = theString.substring(1, theString.length);
     ❹ return rev(tail) + head;
    }
}
document.write(rev("abcdef") + "<br />");
document.write(rev("Hello, world!") + "<br />");
document.write(rev("") + "<br />");
document.write(rev("X") + "<br />");
</script>
```

Результат выполнения этих программ выглядит следующим образом:

```
fedcba
!dlrow ,olleH

X
```

Наша рекурсивная функция `rev()` возвращает строку, представляющую собой результат обращения строкового аргумента `theString`. Простейшие строки, то есть пустая и односимвольная, будут эквивалентны результатам «реверса» самих себя. Это два базовых случая, с которых мы начинаем (комбинируя их с помощью логического оператора `or` или `||` ❶). Для рекурсивного случая сформируем голову `head` из первого символа в строке `theString` ❷, а хвост `tail` — из остальных ❸. Затем функция возвращает обращенную версию `tail`, за которой следует символ `head` ❹.

Определение палиндромов

Палиндромом называют слово или словосочетание, одинаково читающееся в обе стороны. *Довод*, *заказ*, *осело колесо*, *а роза упала на лапу Азора* — все это примеры палиндромов. Чтобы определить, является ли строка палиндромом, можно применить рекурсивную функцию `isPalindrome()`.

Базовым случаем будет пустая или односимвольная строка, которая по своей природе зеркальна. Мы будем использовать подход, аналогичный методу «голова — хвост», но на этот раз разделим строковый аргумент на три части: начало, середину и конец. Если первый и последний символы совпадают, а те, что между ними, образуют палиндром, то вся строка считается палиндромом. В данном случае мы рекурсивно передаем функции `isPalindrome()` среднюю часть строки.

Снова ответим на три вопроса, но уже относительно функции `isPalindrome()`.

Что представляет собой базовый случай? Это пустая или односимвольная строка, которая возвращает значение `True`, поскольку всегда является палиндромом.

Какой аргумент передается рекурсивной функции при ее вызове? Символы, составляющие среднюю часть строкового аргумента.

Как этот аргумент приближается к базовому случаю? Строковый аргумент уменьшается на два символа после каждого рекурсивного вызова до тех пор, пока не превратится в пустую или односимвольную строку.

Вот как выглядит программа на языке Python для обнаружения палиндромов из файла `palindrome.py`:

```
def isPalindrome(theString):
    if len(theString) == 0 or len(theString) == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        return True
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
      ❶ head = theString[0]
      ❷ middle = theString[1:-1]
      ❸ last = theString[-1]
      ❹ return head == last and isPalindrome(middle)

text = 'racecar'
print(text + ' is a palindrome: ' + str(isPalindrome(text)))
text = 'amanaplanacanalpanama'
print(text + ' is a palindrome: ' + str(isPalindrome(text)))
text = 'tacocat'
print(text + ' is a palindrome: ' + str(isPalindrome(text)))
text = 'zophie'
print(text + ' is a palindrome: ' + str(isPalindrome(text)))
```

В файле `palindrome.html` содержится эквивалентная версия этой программы на языке JavaScript:

```
<script type="text/javascript">
function isPalindrome(theString) {
    if (theString.length === 0 || theString.length === 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        return true;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❶ var head = theString[0];
     ❷ var middle = theString.substring(1, theString.length -1);
     ❸ var last = theString[theString.length - 1];
     ❹ return head === last && isPalindrome(middle);
    }
}
text = "racecar";
document.write(text + " is a palindrome: " + isPalindrome(text) + "<br />");
text = "amanaplanacanalpanama";
document.write(text + " is a palindrome: " + isPalindrome(text) + "<br />");
text = "tacocat";
document.write(text + " is a palindrome: " + isPalindrome(text) + "<br />");
text = "zophie";
document.write(text + " is a palindrome: " + isPalindrome(text) + "<br />");
</script>
```

Результат выполнения этих программ выглядит так:

```
racecar is a palindrome: True
amanaplanacanalpanama is a palindrome: True
tacocat is a palindrome: True
zophie is a palindrome: False
```

В базовом случае возвращается значение `True`, поскольку пустая или односимвольная строка всегда является палиндромом. В противном случае строковый аргумент разбивается на три части: первый символ ❶, последний символ ❸ и символы между ними ❷.

Оператор `return` в рекурсивном случае ❹ использует один из так называемых *укороченных логических операторов*, предусмотренных практически в любом языке программирования. Если выражение объединено булевым оператором `and` (И) или `&&` и левая его часть имеет значение `False`, то правое значение игнорируется, так как результатом всего выражения будет `False`. Применение укороченных логических операторов — это оптимизация, позволяющая пропустить этап оценки выражения справа от оператора `and` (И), если левая часть принимает значение `False`. Таким образом, в случае с выражением `head == last and isPalindrome(middle)`, если значением `head == last` является `False`, рекурсивный вызов `isPalindrome()` не выполняется. Это означает, что как только начальная и последняя части строки перестают совпадать, рекурсивная функция завершает свою работу и возвращает значение `False`.

Как и описанные в предыдущем разделе функции суммирования и обращения строки, обсуждаемый здесь рекурсивный алгоритм тоже является последовательным, только он обрабатывает данные не от начала до конца, а от обоих концов к середине. Итеративная версия этого алгоритма, использующая простой цикл, является более прямолинейной. В книге мы рассматриваем рекурсивную версию, потому что именно она часто фигурирует на собеседованиях при приеме на работу.

Решение головоломки «Ханойская башня»

«Ханойская башня» — это головоломка, состоящая из трех стержней, на один из которых нанизано несколько колец или дисков разного размера, образующих пирамиду. Деревянная версия данной головоломки показана на рис. 3.4.

Рис. 3.4. Деревянная головоломка «Ханойская башня»

Для ее решения необходимо переместить стопку колец с одного стержня на другой, следуя трем правилам.

- За раз разрешается переносить только одно кольцо.
- Переместить кольцо можно только с вершины на вершину.
- Нельзя класть кольцо большего размера на меньшее.

Встроенный в Python модуль `turtledemo` предусматривает демонстрацию решения «Ханойской башни». Вы можете посмотреть, как это работает, запустив `python -m turtledemo` в Windows или `python3 -m turtledemo` в macOS/Linux и выбрав пункт minimum_hanoi в меню Examples. Видео решения можно также легко найти в Интернете.

Рекурсивный алгоритм решения данной головоломки нельзя назвать интуитивно понятным. В базовом случае, когда башня состоит из одного кольца, решение

тривиально: достаточно переместить кольцо с одного стержня на другой. Если башня состоит из двух колец, задача немного усложняется: для решения головоломки нужно переместить меньшее кольцо на один стержень (назовем его *временным*), а большее — на другой (назовем его *конечным*), а затем переместить меньшее кольцо с временного стержня на конечный. В итоге оба кольца будут располагаться на конечном стержне в правильном порядке.

Как только вы решите головоломку с башней из трех колец, вы заметите закономерность. Чтобы перенести башню из n колец с начального стержня на конечный, необходимо сделать следующее.

1. Переместить башню из $n - 1$ колец с начального стержня на временный.
2. Переставить n-е кольцо с начального стержня на конечный.
3. Переместить башню из $n - 1$ колец с временного стержня на конечный.

Как и в алгоритме для вычисления чисел Фибоначчи, рекурсивный случай «Ханойской башни» предусматривает два рекурсивных вызова вместо одного. Древовидная диаграмма операций для решения головоломки из четырех колец представлена на рис. 3.5. Когда мы перемещаем четвертое кольцо, перед нами снова встает задача решить головоломку из трех колец. Точно так же, как для башни с тремя кольцами необходимо разрешить загадку из двух колец, переместив третье, и т. д. Задача с одним элементом — тривиальный базовый случай, который предполагает перемещение только этого элемента.

Древовидная структура, представленная на рис. 3.5, намекает на то, что идеальным подходом для решения «Ханойской башни» является рекурсия. В данном дереве операции выполняются сверху вниз и слева направо.

Несмотря на то что человек с легкостью может решить головоломку для башни из трех или четырех колец, с увеличением их числа количество необходимых операций возрастает экспоненциально. Для перемещения башни из n колец требуется минимум $2n - 1$ ходов. То есть, чтобы переставить башню из 31 кольца, требуется совершить более миллиарда операций!

Для создания рекурсивного решения данной головоломки ответим на уже привычные три вопроса.

Что представляет собой базовый случай? Решение головоломки для башни из одного кольца.

Какой аргумент передается рекурсивной функции при ее вызове? Решение головоломки для башни, количество колец в которой на одно меньше, чем в текущей.

Как этот аргумент приближается к базовому случаю? Размер башни уменьшается на одно кольцо после каждого рекурсивного вызова до тех пор, пока она не превратится в башню из одного кольца.

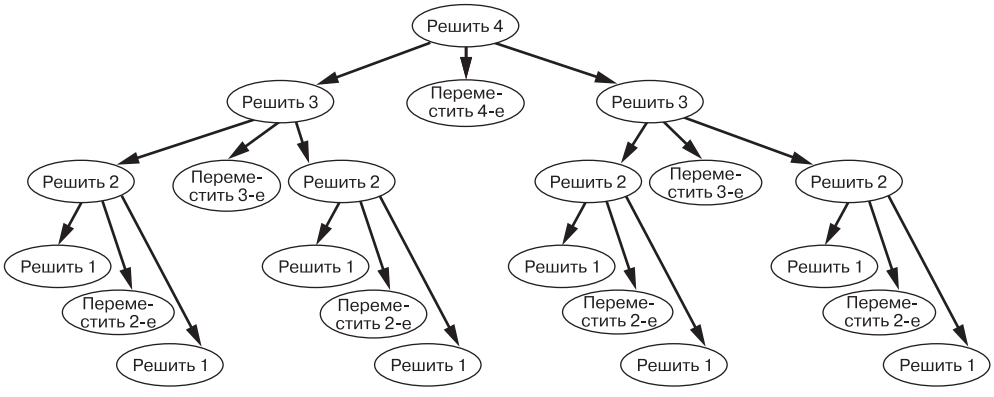


Рис. 3.5. Последовательность операций для решения головоломки «Ханойская башня» из четырех колец

Следующая программа из файла `towerOfHanoiSolver.py` решает головоломку «Ханойская башня», визуализируя каждый шаг этого процесса:

```
import sys

# Создание стержней A, B и C. Конец списка соответствует вершине стержня
  TOTAL_DISKS = 6 ❶

# Заполнение стержня A кольцами:
  TOWERS = {'A': list(reversed(range(1, TOTAL_DISKS + 1))), ❷
            'B': [],
            'C': []}

def printDisk(diskNum):
    # Вывод на экран одного кольца шириной diskNum
    emptySpace = ' ' * (TOTAL_DISKS - diskNum)
    if diskNum == 0:
        # Рисование пустого стержня
        sys.stdout.write(emptySpace + '||' + emptySpace)
    else:
        # Рисование кольца
        diskSpace = '@' * diskNum
        diskNumLabel = str(diskNum).rjust(2, '_')
        sys.stdout.write(emptySpace + diskSpace + diskNumLabel +
                         diskSpace + emptySpace)

def printTowers():
    # Вывод на экран всех трех башен
    for level in range(TOTAL_DISKS, -1, -1):
        for tower in (TOWERS['A'], TOWERS['B'], TOWERS['C']):
            if level >= len(tower):
                printDisk(0)
```

```
            else:
                printDisk(tower[level])
        sys.stdout.write('\n')
    # Вывод на экран меток A, B и C
    emptySpace = ' ' * (TOTAL_DISKS)
    print('%s A%s%s B%s%s C\n' % (emptySpace, emptySpace, emptySpace,
          emptySpace, emptySpace))

def moveOneDisk(startTower, endTower):
    # Перемещение верхнего кольца с начального стержня на конечный
    disk = TOWERS[startTower].pop()
    TOWERS[endTower].append(disk)
def solve(numberOfDisks, startTower, endTower, tempTower):
    # Перемещение верхних numberOfDisks колец с начального стержня на конечный
    if numberOfDisks == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        moveOneDisk(startTower, endTower) ❸
        printTowers()
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        solve(numberOfDisks - 1, startTower, tempTower, endTower) ❹
        moveOneDisk(startTower, endTower) ❺
        printTowers()
        solve(numberOfDisks - 1, tempTower, endTower, startTower) ❻
        return
# Решение:
printTowers()
solve(TOTAL_DISKS, 'A', 'B', 'C')

# Раскомментируйте, чтобы активировать интерактивный режим:
#while True:
#    printTowers()
#    print('Enter letter of start tower and the end tower. (A, B, C)
           Or Q to quit.')
#    move = input().upper()
#    if move == 'Q':
#        sys.exit()
#    elif move[0] in 'ABC' and move[1] in 'ABC' and move[0] != move[1]:
#        moveOneDisk(move[0], move[1])
```

Эквивалентный код на языке JavaScript содержится в файле `towerOfHanoiSolver.html`:

```
<script type="text/javascript">
// Создание стержней A, B и C. Конец массива соответствует вершине стержня.
  var TOTAL_DISKS = 6; ❶
  var TOWERS = {"A": [], ❷
              "B": [],
              "C": []}; // Заполнение стержня A:
for (var i = TOTAL_DISKS; i > 0; i--) {
    TOWERS["A"].push(i);
}
```

```
function printDisk(diskNum) {
    // Вывод на экран одного кольца шириной diskNum
    var emptySpace = " ".repeat(TOTAL_DISKS - diskNum);
    if (diskNum === 0) {
        // Рисование пустого стержня
        document.write(emptySpace + "||" + emptySpace);
    } else {
        // Рисование кольца
        var diskSpace = "@".repeat(diskNum);
        var diskNumLabel = String("___" + diskNum).slice(-2);
        document.write(emptySpace + diskSpace + diskNumLabel +
                       diskSpace + emptySpace);
    }
}
function printTowers() {
    // Вывод на экран всех трех башен
    var towerLetters = "ABC";
    for (var level = TOTAL_DISKS; level >= 0; level--) {
        for (var towerLetterIndex = 0; towerLetterIndex < 3;
          towerLetterIndex++) {
            var tower = TOWERS[towerLetters[towerLetterIndex]];
            if (level >= tower.length) {
                printDisk(0);
            } else {
                printDisk(tower[level]);
            }
        }
        document.write("<br />");
    }
    // Вывод на экран меток A, B и C
    var emptySpace = " ".repeat(TOTAL_DISKS);
    document.write(emptySpace + " A" + emptySpace + emptySpace +
      " B" + emptySpace + emptySpace + " C<br /><br />");
}
function moveOneDisk(startTower, endTower) {
    // Перемещение верхнего кольца с начального стержня на конечный
    var disk = TOWERS[startTower].pop();
    TOWERS[endTower].push(disk);
}

function solve(numberOfDisks, startTower, endTower, tempTower) {
    // Перемещение верхних numberOfDisks колец с начального стержня на конечный
    if (numberOfDisks == 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        moveOneDisk(startTower, endTower); ❸
        printTowers();
        return;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        solve(numberOfDisks - 1, startTower, tempTower, endTower); ❹
        moveOneDisk(startTower, endTower); ❺
        printTowers();
```

```
        solve(numberOfDisks - 1, tempTower, endTower, startTower); ❻
        return;
    }
}

// Решение:
document.write("<pre>");
printTowers();
solve(TOTAL_DISKS, "A", "B", "C");
document.write("</pre>");
</script>
```

Запустив этот код, вы увидите все этапы перемещения колец со стержня A на стержень B:

```
      ||            ||            ||
     @_1@           ||            ||
    @@_2@@          ||            ||
   @@@_3@@@         ||            ||
  @@@@_4@@@@        ||            ||
 @@@@@_5@@@@@       ||            ||
@@@@@@_6@@@@@@      ||            ||
      A             B             C
      ||            ||            ||
      ||            ||            ||
    @@_2@@          ||            ||
   @@@_3@@@         ||            ||
  @@@@_4@@@@        ||            ||
 @@@@@_5@@@@@       ||            ||
@@@@@@_6@@@@@@      ||           @_1@
      A             B             C
--пропущенный фрагмент--

      ||            ||            ||
      ||            ||            ||
      ||            ||            ||
      ||            ||            ||
      ||          @@_2@@          ||
     @_1@        @@@_3@@@         ||
@@@@@@_6@@@@@@  @@@@_4@@@@   @@@@@_5@@@@@
      A             B             C

--пропущенный фрагмент--

      ||            ||            ||
      ||           @_1@           ||
      ||          @@_2@@          ||
      ||         @@@_3@@@         ||
      ||        @@@@_4@@@@        ||
      ||       @@@@@_5@@@@@       ||
      ||      @@@@@@_6@@@@@@      ||
      A             B             C
```

Версия программы на языке Python предусматривает интерактивный режим, позволяющий вам решить головоломку самостоятельно. Для его активации раскомментируйте строки кода в конце программы `TowerOf HanoiSolver.py`.

Для начала можете запустить программу с башнями небольшого размера, задав для константы `TOTAL_DISKS` ❶ значение `1` или `2`. Список целых чисел в коде Python и массив целых чисел в коде JavaScript представляют собой стержень. Целое число олицетворяет кольцо, диаметр которого зависит от величины этого числа. Начало списка или массива соответствует нижней части стержня, а конец — его вершине. Например, список `[6, 5, 4, 3, 2, 1]` воплощает стартовый стержень с шестью кольцами, самое большое из которых находится внизу, а список `[]` представляет пустой стержень. Переменная `TOWERS` содержит все три списка ❷.

В базовом случае кольцо наименьшего размера просто перемещается с начального стержня на конечный ❸. Рекурсивный случай для башни из n колец предполагает выполнение трех шагов: решение головоломки для случая $n - 1$ ❹, перемещение n-го кольца ❺ и повторное решение головоломки для случая $n - 1$ ❻.

Использование заливки

Графические редакторы часто используют *алгоритм заливки* для заполнения области произвольной формы другим цветом. Пример такой фигуры продемонстрирован в левом верхнем углу на рис. 3.6. Далее показаны три различных участка этой фигуры, залитые серым цветом. Заливка начинается с белого пиксела и заполняет замкнутое пространство вплоть до обнаружения небелого пиксела.

Алгоритм заливки — рекурсивный: он сначала изменяет цвет одного пиксела, после чего рекурсивная функция вызывается для всех смежных пикселов того же исходного цвета. Затем алгоритм переходит к соседним и т. д., изменяя цвет каждого пиксела вплоть до заполнения замкнутой фигуры.

Базовым случаем служит край изображения или пиксел, цвет которого отличается от исходного цвета первого пиксела. Поскольку обнаружение базового случая является единственным способом остановки «распространения» рекурсивных вызовов, данному алгоритму свойственно эмерджентное поведение, выражающееся в изменении цвета всех близлежащих пикселов.

По традиции ответим на три вопроса относительно нашей рекурсивной функции `floodFill()`.

Что представляет собой базовый случай? Координаты пиксела, цвет которого отличается от исходного, или координаты края изображения.

Какие аргументы передаются рекурсивной функции при ее вызове? Аргументами для четырех рекурсивных вызовов являются координаты *x* и *y* четырех пикселов, смежных с текущим.

Как эти аргументы приближаются к базовому случаю? Алгоритм обнаруживает пикселы, цвет которых отличается от исходного, или достигает края изображения. Так или иначе, у алгоритма заканчиваются пикселы для проверки.

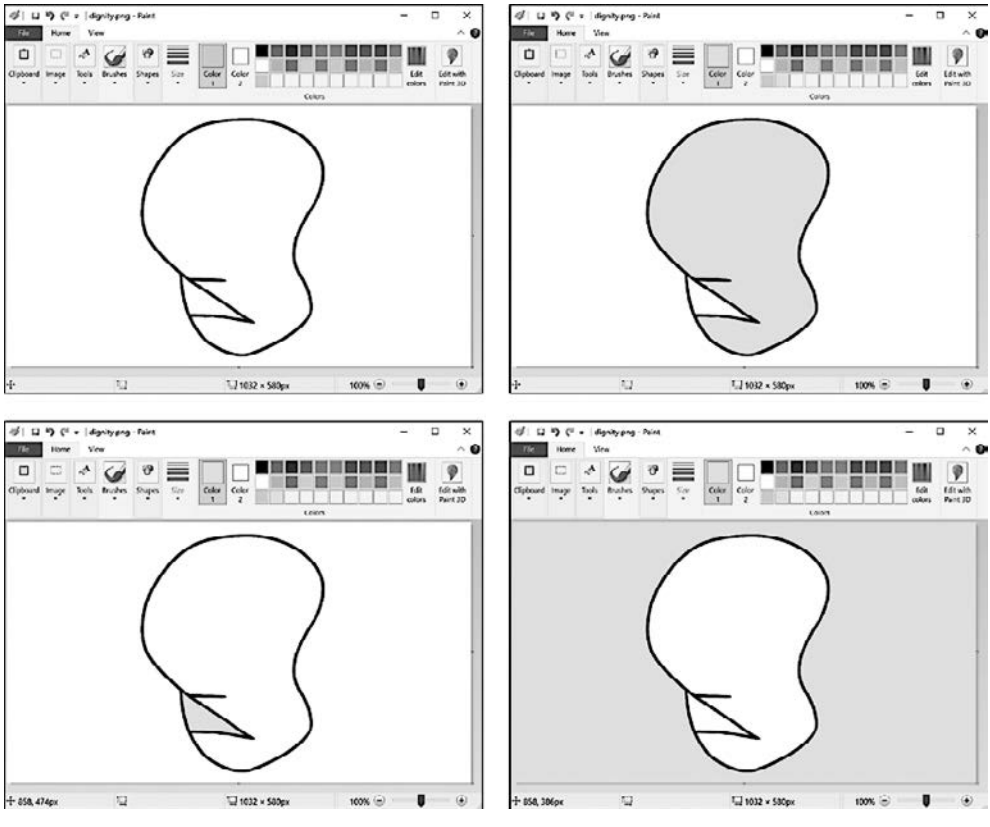


Рис. 3.6. Исходная фигура в графическом редакторе *(вверху слева)* и та же самая фигура с тремя разными областями, залитыми светло-серым цветом

В нашем примере программы вместо изображения используется список односимвольных строк для формирования двумерной сетки из текстовых символов, служащей в качестве «изображения». Каждая строка представляет «пиксел», а конкретный символ — «цвет». В файле `floodfill.py` содержится программа на языке Python, которая реализует алгоритм заливки, данные изображения и функцию для вывода этого изображения на экран:

```
import sys
# Создайте изображение (убедитесь, что оно прямоугольное!)
❶ im = [list('..########################...........'),
        list('..#......................#...#####...'),
        list('..#.........########....#####...#...'),
        list('..#.........#......#............#...'),
        list('..#.........########.........####...'),
        list('..######.....................#......'),
        list('.......#..#####.....###########......'),
        list('.......####...#######................')]

HEIGHT = len(im)
WIDTH = len(im[0])

def floodFill(image, x, y, newChar, oldChar=None):
    if oldChar == None:
        # В oldChar по умолчанию сохраняется символ в точке с координатами x, y
     ❷ oldChar = image[y][x]
    if oldChar == newChar or image[y][x] != oldChar:
        # БАЗОВЫЙ СЛУЧАЙ
        return

    image[y][x] = newChar # Изменение символа

    # Удалите символы комментария, чтобы просмотреть каждый шаг:
    #printImage(image)
    # Изменение соседних символов
    if y + 1 < HEIGHT and image[y + 1][x] == oldChar:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ floodFill(image, x, y + 1, newChar, oldChar)
    if y - 1 >= 0 and image[y - 1][x] == oldChar:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❹ floodFill(image, x, y - 1, newChar, oldChar)
    if x + 1 < WIDTH and image[y][x + 1] == oldChar:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❺ floodFill(image, x + 1, y, newChar, oldChar)
    if x - 1 >= 0 and image[y][x - 1] == oldChar:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❻ floodFill(image, x - 1, y, newChar, oldChar)
 ❼ return # БАЗОВЫЙ СЛУЧАЙ

def printImage(image):
    for y in range(HEIGHT):
        # Вывод каждой строки на экран
        for x in range(WIDTH):
            # Вывод каждого столбца на экран
            sys.stdout.write(image[y][x])
        sys.stdout.write('\n')
    sys.stdout.write('\n')

printImage(im)
floodFill(im, 3, 3, 'o')
printImage(im)
```

Версия программы на языке JavaScript содержится в файле `floodfill.html`:

```
<script type="text/javascript">
// Создайте изображение (убедитесь в том, что оно прямоугольное!)
❶ var im = ["..########################...........".split(""),
           "..#......................#...#####...".split(""),
           "..#.........########....#####...#...".split(""),
           "..#.........#......#...........#...".split(""),
           "..#.........########.........####...".split(""),
           "..######.....................#......".split(""),
           ".......#..#####.....############.....".split(""),
           ".......####...#######..............".split("")];
var HEIGHT = im.length;
var WIDTH = im[0].length;

function floodFill(image, x, y, newChar, oldChar) {
    if (oldChar === undefined) {
        // В oldChar по умолчанию сохраняется символ в точке с координатами x, y
     ❷ oldChar = image[y][x];
    }
    if ((oldChar == newChar) || (image[y][x] != oldChar)) {
        // БАЗОВЫЙ СЛУЧАЙ
        return;
    }

    image[y][x] = newChar; // Изменение символа

    // Удалите символы комментария, чтобы просмотреть каждый шаг:
    // printImage(image);
    // Изменение соседних символов
    if ((y + 1 < HEIGHT) && (image[y + 1][x] == oldChar)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ floodFill(image, x, y + 1, newChar, oldChar);
    }
    if ((y - 1 >= 0) && (image[y - 1][x] == oldChar)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❹ floodFill(image, x, y - 1, newChar, oldChar);
    }
    if ((x + 1 < WIDTH) && (image[y][x + 1] == oldChar)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❺ floodFill(image, x + 1, y, newChar, oldChar);
    }
    if ((x - 1 >= 0) && (image[y][x - 1] == oldChar)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❻ floodFill(image, x - 1, y, newChar, oldChar);
    }
 ❼ return; // БАЗОВЫЙ СЛУЧАЙ
}
function printImage(image) {
    document.write("<pre>");
    for (var y = 0; y < HEIGHT; y++) {
```

```
        // Вывод каждой строки на экран
        for (var x = 0; x < WIDTH; x++) {
            // Вывод каждого столбца на экран
            document.write(image[y][x]);
        }
        document.write("\n");
    }
    document.write("\n</ pre>");
}
printImage(im);
floodFill(im, 3, 3, "o");
printImage(im);
</script>
```

После запуска кода программа заполняет внутреннюю часть фигуры, нарисованной символами `#`, начиная с точки с координатами (3, 3). При этом она заменяет все знаки точки (`.`) буквами `o`. Далее показаны эти изображения до и после выполнения заливки:

```
..########################..........
..#......................#...#####...
..#.........########....#####...#...
..#.........#......#............#...
..#.........########.........####...
..######.......................#.....
.......#..#####.....############.....
.......####...#######..............

..########################..........
..#ooooooooooooooooooooo#...#####...
..#ooooooooo########oooo#####ooo#...
..#ooooooooo#......#ooooooooooooo#...
..#ooooooooo########ooooooooo####...
..######ooooooooooooooooooooo#......
.......#oo#####ooooo############.....
.......####...#######..............
```

Если хотите увидеть пошаговое выполнение процесса заливки, раскомментируйте строку `printImage(image)` ❶ в функции `floodFill()` и снова запустите программу.

Изображение представлено двумерным строковым массивом. Мы можем передать функции `floodFill()` эту структуру данных `image`, координаты `x` и `y`, а также новый символ. Функция проверяет текущий знак в точке с координатами `x` и `y` и сохраняет его в переменной `oldChar` ❷.

Если текущий символ с координатами `x` и `y` не совпадает с `oldChar` — это базовый случай, в котором функция просто возвращается. В противном случае функция продолжает работу и реализует свои четыре рекурсивных случая, предполагающих

передачу *x*- и *y*-координат соседнего пиксела снизу ❸, сверху ❹, справа ❺ и слева ❻. После выполнения этих четырех потенциальных рекурсивных вызовов работа функции завершается неявным базовым случаем, который в нашей программе явно выражен оператором `return` ❼.

Алгоритм заполнения не обязательно должен быть рекурсивным. В случае больших изображений применение рекурсивной функции приводит к переполнению стека. Если бы мы решили реализовать алгоритм заливки с помощью цикла и стека, то этот стек начинался бы с *x*- и *y*-координат начального пиксела. Код в цикле удалял бы координаты с вершины стека, а в случае совпадения пиксела, находящегося в точке с указанными координатами, со значением `oldChar` помещал бы в стек расположение четырех соседних пикселов. Цикл будет считаться завершенным при достижении базового случая, при котором стек пустеет, поскольку в него больше не помещаются координаты соседних пикселов.

Однако для реализации алгоритма заливки использование стека не является необходимым условием. Помещение и извлечение значений из стека типа «первым пришел — последним ушел» эффективно для поиска с возвратом, а порядок, в котором пикселы обрабатываются алгоритмом, может быть произвольным. То есть вместо стека мы могли бы прибегнуть к такой структуре данных, которая удаляла бы элементы в случайном порядке. Реализация этих итеративных алгоритмов заливки содержится в файлах `floodFillIterative.py` и `floodFillIterative.html`, которые вы можете загрузить со страницы https://nostarch.com/recursive-book-recursion.

Функция Аккермана

Функция Аккермана была представлена в 1928 году и названа в честь ее первооткрывателя Вильгельма Аккермана, ученика математика Давида Гильберта (его фрактальную кривую мы обсудим в главе 9). Позднее математики Ружа Петер и Рафаэль Робинсон разработали версию данной функции, о которой пойдет речь в текущем разделе.

Хотя функция Аккермана применяется в высшей математике, в основном она известна в качестве примера очень быстро растущей рекурсивной функции. Даже незначительное увеличение двух ее целочисленных аргументов приводит к существенному возрастанию количества рекурсивных вызовов, которые она совершает.

Функция Аккермана принимает два аргумента — `m` и `n` — и в базовом случае возвращает `n + 1`, когда `m` равно `0`. Она также предусматривает два рекурсивных случая: когда `n` равно `0`, функция возвращает `ackermann(m - 1, 1)`, а когда `n` больше `0` — `ackermann(m - 1, ackermann(m, n - 1))`. Вероятно, это ни о чем вам не говорит. Просто имейте в виду, что количество рекурсивных вызовов, совершаемых

функцией Аккермана, растет очень быстро. Вызов функции `ackermann(1, 1)` приводит к трем рекурсивным вызовам, `ackermann(2, 3)` — к 43, `ackermann(3, 5)` — к 42 437, а `ackermann(5, 7)` — к... На самом деле я не знаю, к скольким, потому что для вычисления потребовалось бы время, в несколько раз превышающее возраст Вселенной.

Давайте ответим на три вопроса, которые мы задаем при реализации рекурсивных алгоритмов.

Что представляет собой базовый случай? Случай, когда `m` равно `0`.

Какие аргументы передаются рекурсивной функции при ее вызове? В качестве следующего параметра `m` передается либо `m`, либо `m - 1`, а в качестве `n` передается `1`, `n - 1` или значение, возвращаемое функцией `ackermann(m, n - 1)`.

Как эти аргументы приближаются к базовому случаю? Значение аргумента `m` всегда либо уменьшается, либо остается прежним, поэтому в конечном итоге оно достигает `0`.

Вот как выглядит программа из файла `ackermann.py` на языке Python:

```
def ackermann(m, n, indentation=None):
    if indentation is None:
        indentation = 0
    print('%sackermann(%s, %s)' % (' ' * indentation, m, n))

    if m == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return n + 1
    elif m > 0 and n == 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return ackermann(m - 1, 1, indentation + 1)
    elif m > 0 and n > 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return ackermann(m - 1, ackermann(m, n - 1, indentation + 1), indentation + 1)

print('Starting with m = 1, n = 1:')
print(ackermann(1, 1))
print('Starting with m = 2, n = 3:')
print(ackermann(2, 3))
```

А вот эквивалентная программа (см. файл `ackermann.html`) на языке JavaScript:

```
<script type="text/javascript">
function ackermann(m, n, indentation) {
    if (indentation === undefined) {
        indentation = 0;
    }
    document.write(" ".repeat(indentation) + "ackermann(" + m + ", " + n + ")\n");
```

```
    if (m === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
        return n + 1;
    } else if ((m > 0) && (n === 0)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return ackermann(m - 1, 1, indentation + 1);
    } else if ((m > 0) && (n > 0)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return ackermann(m - 1, ackermann(m, n - 1, indentation + 1), indentation + 1);
    }
}
document.write("<pre>");
document.write("Starting with m = 1, n = 1:<br />");
document.write(ackermann(1, 1) + "<br />");
document.write("Starting with m = 2, n = 3:<br />");
document.write(ackermann(2, 3) + "<br />");
document.write("</pre>");
</script>
```

Отступ в выводе программы (заданный аргументом `indentation`) сообщает о том, насколько глубоко в стеке находится данный вызов рекурсивной функции:

```
Starting with m = 1, n = 1:
ackermann(1, 1)
 ackermann(1, 0)
  ackermann(0, 1)
 ackermann(0, 2)
3
Starting with m = 2, n = 3:
ackermann(2, 3)
 ackermann(2, 2)
  ackermann(2, 1)
   ackermann(2, 0)
--пропущенный фрагмент--
    ackermann(0, 6)
   ackermann(0, 7)
  ackermann(0, 8)
9
```

Вы также можете попробовать вызвать функцию `ackermann(3, 3)`, однако будьте готовы, что вычисления займут слишком много времени. Чтобы ускорить процесс, попробуйте закомментировать все вызовы `print()` и `document.write()`, кроме тех, которые выводят на экран окончательное значение, возвращаемое функцией `ackermann()`.

Помните, что даже такой рекурсивный алгоритм, как функция Аккермана, может быть реализован в виде итеративной функции. Итеративные алгоритмы Аккермана находятся в файлах `ackermannIterative.py` и `ackermannIterative.html`, которые расположены по адресу https://nostarch.com/recursive-book-recursion.

Резюме

В этой главе мы рассмотрели несколько классических рекурсивных алгоритмов. Перед реализацией каждого из них мы отвечали на три вопроса, которые всегда следует задавать при разработке собственных рекурсивных функций: что представляет собой базовый случай? Какие аргументы передаются рекурсивной функции при ее вызове? Как эти аргументы приближаются к базовому случаю? Если этого не делать, то ваша функция будет рекурсивно вызываться до тех пор, пока не спровоцирует переполнение стека.

Рекурсивные функции суммирования, обращения (реверса) строки и определения палиндрома можно было бы легко реализовать с помощью простого цикла. Ключевым моментом является то, что все они совершают один проход по предоставленным им данным без поиска с возвратом. Как говорилось в главе 2, рекурсивные алгоритмы особенно хороши для задач, предполагающих древовидную структуру и поиск с возвратом.

Древовидные структуры в решении головоломки «Ханойская башня» предусматривают поиск с возвратом, поскольку выполнение программы идет по дереву сверху вниз, слева направо. Это делает данную задачу идеальным кандидатом для воплощения рекурсии в жизнь, особенно учитывая, что решение требует выполнения двух рекурсивных вызовов для башен меньшего размера.

Алгоритм заливки применяется для графики и рисования, а также для определения формы непрерывных областей. Если вы пользовались инструментами заливки в графическом редакторе, то, вероятно, имели дело с какой-то разновидностью данного алгоритма.

Функция Аккермана — отличный пример того, как быстро может расти рекурсивная функция по мере увеличения значения ее аргументов. Хотя в повседневном программировании у такой функции не так много практических применений, без нее тема рекурсии была бы неполной. Тем не менее, как и все остальные рекурсивные функции, функцию Аккермана возможно реализовать итеративно с помощью цикла и стека.

Дополнительные источники информации

Дополнительную информацию о головоломке «Ханойская башня» можно найти в «Википедии»: https://ru.wikipedia.org/wiki/Ханойская_башня. А в видео *Recursion 'Super Power' (in Python)* на канале Computerphile по адресу https://youtu.be/8lhxIOAfDss рассказывается о решении данной головоломки на языке Python. В двухсерийной видеолекции *Binary, Hanoi and Sierpiński* на канале 3Blue1Brown подробно описывается взаимосвязь между «Ханойской башней», двоичными числами

и фрактальным треугольником Серпинского. Первая часть доступна по адресу https://youtu.be/2SUvWfNJSsM.

В «Википедии» есть анимированная демонстрация процесса работы алгоритма заливки небольшой формы: https://ru.wikipedia.org/wiki/Заливка.

Функция Аккермана обсуждается в видео *The Most Difficult Program to Compute?* на канале Computerphile по адресу https://youtu.be/i7sm9dzFtEI. Если хотите узнать больше о месте данной функции в теории вычислимости, можете посмотреть серию из пяти видеолекций о примитивно-рекурсивных и частично-рекурсивных функциях на канале Hackers at Cambridge по адресу https://youtu.be/yaDQrOUK-KY.

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое голова массива или строки?
2. Что такое хвост массива или строки?
3. Какие три вопроса необходимо задать при реализации рекурсивного алгоритма?
4. Что такое «прыжок веры» в контексте рекурсии?
5. Что вы должны понять о создаваемой вами рекурсивной функции, прежде чем совершать «прыжок веры»?
6. Чем такая линейная структура данных, как массив или строка, напоминает древовидную структуру?
7. Предполагает ли рекурсивная функция `sum()` поиск с возвратом по отношению к данным, которые она обрабатывает?
8. В программе с алгоритмом заливки попробуйте изменить строки в переменной `im`, создав незамкнутую фигуру в форме буквы C. Что произойдет при попытке залить эту фигуру, начиная с ее середины?
9. Ответьте на следующие три вопроса относительно каждого из рекурсивных алгоритмов, представленных в главе.

   А. Что представляет собой базовый случай?

   Б. Какой аргумент передается рекурсивной функции при ее вызове?

   В. Как этот аргумент приближается к базовому случаю?

   Затем воссоздайте рекурсивные алгоритмы из пройденной главы, не подглядывая в исходный код.

Практика

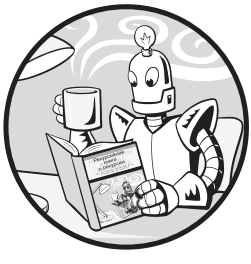
Решите каждую из следующих задач.

1. Используя метод «голова — хвост», создайте рекурсивную функцию `concat()`, которая получает массив строк и возвращает результат их объединения в одну строку. Например, функция `concat(['Hello', 'World'])` должна возвратить строку `HelloWorld`.
2. Используя метод «голова — хвост», создайте рекурсивную функцию `product()`, которая получает массив целых чисел и возвращает результат их перемножения. Этот код будет практически идентичен коду функции `sum()`, представленной в пройденной главе. Однако обратите внимание, что в базовом случае с массивом, содержащим одно целое число, возвратится целое число, а в базовом случае с пустым массивом — `1`.
3. Используя алгоритм заливки, подсчитайте количество «комнат» или замкнутых фигур в двумерной сетке. Вы можете сделать это с помощью вложенных циклов `for`, которые вызывают функцию заливки для каждого символа в сетке, если он является точкой, чтобы заменить ее символом решетки. Например, следующие данные приведут к тому, что программа обнаружит в сетке шесть фигур, заполненных точками, — пять «комнат» и окружающее их пространство.

```
...##########.................................
...#........#....####..................##########
...#........#....#..#...#############...#........#
...##########....#..#...#..........#...##.......#
.......#....#....####...#..........#....##......#
.......#....#....#......#############.....##.....#
.......######....#........................##....#
.................####........####..........######
```

4

Алгоритмы поиска с возвратом и обхода дерева



Из предыдущих глав вы узнали, что рекурсия хорошо подходит для решения задач, предусматривающих использование древовидной структуры и поиска с возвратом, например, для разработки алгоритма прохождения лабиринтов. Чтобы понять почему, представьте, что ствол дерева разделяется на несколько ветвей, которые, в свою очередь, разделяются на другие ветви. Таким образом, дерево обладает рекурсивной, самоподобной формой.

Лабиринт может быть представлен в виде древовидной структуры данных, поскольку ходы в нем разветвляются на разные пути. Когда вы заходите в тупик, приходится возвращаться к более ранней точке ветвления.

Задача обхода древовидных графов тесно связана со многими рекурсивными алгоритмами, в том числе с алгоритмом прохождения лабиринта, описанным в текущей главе, а также с программой генерации лабиринта из главы 11. Мы рассмотрим алгоритмы обхода дерева и применим их для нахождения определенных имен в древовидной структуре данных. С помощью обхода дерева мы также реализуем алгоритм нахождения самого глубокого узла. Наконец, визуализируем лабиринт в виде древовидной структуры данных и применим методы обхода дерева и поиска с возвратом для поиска пути от начала лабиринта до выхода.

Использование метода обхода дерева

Если вы программируете на Python и JavaScript, то уже наверняка привыкли работать со списками, массивами и словарями. С древовидными структурами данных вы сталкиваетесь только при работе с низкоуровневыми деталями некоторых

алгоритмов. Обсуждение таких структур, как абстрактные синтаксические деревья, приоритетные очереди и деревья Адельсона — Вельского — Лэндиса (АВЛ-деревья), выходит за рамки книги. Однако сами по себе деревья представляют собой достаточно простую концепцию.

Древовидная структура данных состоит из *узлов* (или *вершин*), соединенных ребрами. *Ребра* определяют взаимосвязи между *узлами*, которые, в свою очередь, содержат данные. Начальный узел дерева называется *корнем*, а конечные — *листьями*. У деревьев может быть только один корень.

Родительские узлы соединены ребрами с *дочерними* и не являются листьями. Таким образом, листья — это вершины, у которых нет дочерних узлов. Дочерними считаются любые некорневые узлы. У вершин дерева бывает несколько дочерних узлов. Родительские узлы, расположенные между дочерним узлом и корнем, также называются *предками* этого дочернего узла. Дочерние узлы, находящиеся между родительским узлом и листом, называются *потомками* данного родительского узла. У родительских узлов может быть несколько дочерних, но у каждого дочернего есть только один родитель (исключением служит корневой узел, у которого нет родителя). Между любыми двумя вершинами в деревьях может существовать только один путь.

На рис. 4.1 показаны примеры дерева и трех структур, которые не являются деревьями.

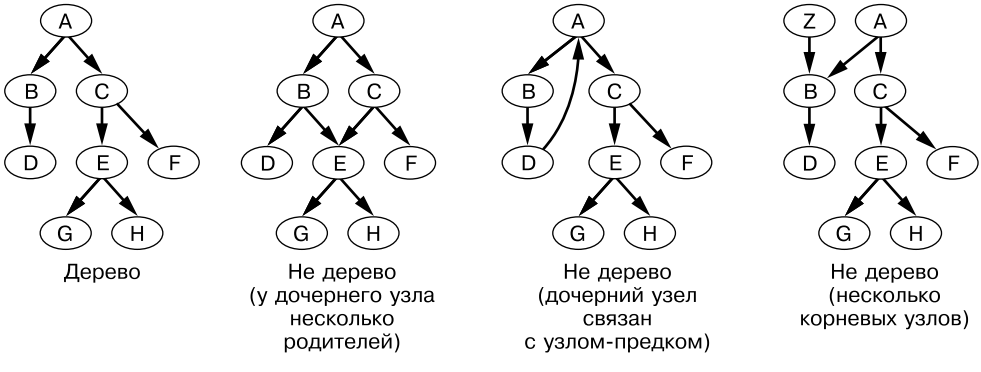


Рис. 4.1. Дерево *(слева)* и три структуры, не являющиеся деревьями

Итак, дочерние узлы должны иметь только одного родителя и не иметь ребра, создающего петлю, в противном случае структура не может считаться деревом. Рекурсивные алгоритмы, которые будут рассмотрены в этой главе, применимы только к древовидным структурам данных.

Древовидная структура данных в Python и JavaScript

Древовидные структуры данных часто изображаются растущими вниз, с корнем вверху. На рис. 4.2 показано дерево, созданное с помощью следующего кода Python (который также является допустимым кодом JavaScript):

```
root  = {'data': 'A', 'children': []}
node2 = {'data': 'B', 'children': []}
node3 = {'data': 'C', 'children': []}
node4 = {'data': 'D', 'children': []}
node5 = {'data': 'E', 'children': []}
node6 = {'data': 'F', 'children': []}
node7 = {'data': 'G', 'children': []}
node8 = {'data': 'H', 'children': []}
root['children'] = [node2, node3]
node2['children'] = [node4]
node3['children'] = [node5, node6]
node5['children'] = [node7, node8]
```

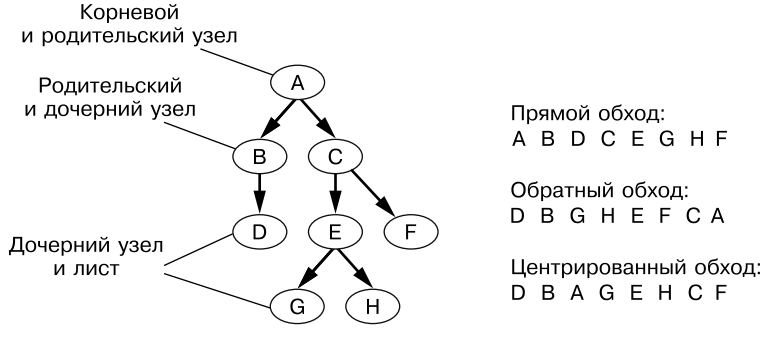


Рис. 4.2. Дерево с корнем A и листьями D, G, H и F, а также способы его обхода

Каждый узел дерева содержит фрагмент данных (строку, содержащую букву от `A` до `H`) и список своих дочерних узлов. Способы обхода дерева, перечисленные на рис. 4.2, описаны в следующих разделах.

В коде этого дерева каждая вершина представлена словарем Python (или объектом JavaScript) с ключом `data`, в котором хранятся данные узла, и ключом `children`, содержащим список других узлов. Для повышения удобочитаемости кода я использую переменные от `node2` до `node8` и `root` для хранения каждого узла, но повторять за мной вовсе не обязательно. Следующий код Python/JavaScript эквивалентен предыдущему листингу, но чуть более сложен для восприятия:

```
root = {'data': 'A', 'children': [{'data': 'B', 'children':
[{'data': 'D', 'children': []}]}, {'data': 'C', 'children':
[{'data': 'E', 'children': [{'data': 'G', 'children': []},
{'data': 'H', 'children': []}]}, {'data': 'F', 'children': []}]}]}
```

Дерево, изображенное на рис. 4.2, представляет собой особую структуру данных, известную как *направленный* или *ориентированный ациклический граф* (directed acyclic graph, DAG). В математике и информатике под *графом* подразумевается коллекция узлов и ребер, поэтому дерево считается разновидностью графа. Граф является *направленным*, поскольку его ребра имеют одно направление: от родительского узла к дочернему. Ребра DAG не являются ненаправленными или двунаправленными (деревья, как правило, не имеют подобного ограничения, так что их ребра могут быть направлены в обе стороны, в том числе от дочернего узла к родительскому). Граф называют *ациклическим*, поскольку он не содержит петель или *циклов*, связывающих дочерние вершины с их узлами-предками; ветви дерева должны расти в одном и том же направлении.

Списки, массивы и строки можно воспринимать в качестве линейных деревьев; первый элемент — это корень, а у вершин есть только один прямой потомок. Данное линейное дерево заканчивается единственным листовым узлом. Такие линейные деревья называются *связанными списками*, поскольку каждый узел в списке имеет только один, следующий за ним узел. На рис. 4.3 показан связанный список, в котором хранятся буквы, составляющие слово *HELLO*.

Рис. 4.3. Связанный список, в котором хранятся буквы, составляющие слово HELLO. Связанные списки можно рассматривать в качестве разновидности древовидной структуры данных

В примерах, представленных в текущей главе, используется код дерева, изображенного на рис. 4.2. Алгоритм обхода дерева будет обрабатывать каждый его узел, начиная с корня и следуя вдоль ребер.

Обход дерева

Мы можем написать код для получения доступа к данным, хранящимся в любом узле, начав с корня `root`. Например, после ввода кода дерева в интерактивной командной строке Python или JavaScript выполните следующие команды:

```
>>> root['children'][1]['data']
'C'
>>> root['children'][1]['children'][0]['data']
'E'
```

Код для обхода дерева можно написать в виде рекурсивной функции, потому что древовидные структуры данных обладают свойством самоподобия: у родительского узла есть дочерние, которые являются родителем уже для собственных дочерних узлов. Алгоритмы обхода дерева гарантируют, что ваши программы смогут получить

доступ или изменить данные в каждом из узлов дерева вне зависимости от его формы или размера.

Ответим на три вопроса из предыдущей главы относительно нашего кода для обхода дерева.

Что представляет собой базовый случай? Листовой узел, который не имеет дочерних узлов и не требует дополнительных рекурсивных вызовов, что вынуждает алгоритм вернуться к предыдущему родительскому узлу.

Какой аргумент передается рекурсивной функции при ее вызове? Узел, который требуется посетить и дочерние элементы которого будут следующими узлами для обхода.

Как этот аргумент приближается к базовому случаю? В DAG нет циклов, поэтому следование по узлам-потомкам в конечном итоге приводит к достижению листового узла.

Имейте в виду, что обход особенно глубоких древовидных структур рискует спровоцировать переполнение стека. Это связано с тем, что каждый дополнительный уровень дерева требует еще одного вызова функции, а слишком большое количество вызовов без возврата приводит к переполнению стека. Однако широкие, хорошо сбалансированные деревья крайне редко бывают настолько глубокими. Если каждый узел дерева глубиной 1000 уровней имеет две дочерние вершины, то количество узлов в этом дереве составляет около 2^{1000}. Это больше, чем атомов во Вселенной, поэтому крайне маловероятно, что ваша древовидная структура данных будет настолько велика.

Существует три алгоритма обхода дерева: прямой, обратный и центрированный. Мы обсудим каждый из них в следующих трех разделах.

Прямой обход дерева

Прямой обход дерева рекомендуется использовать, если вашему алгоритму необходимо получить доступ к данным в родительских узлах перед получением данных, хранящихся в их дочерних узлах. Например, прямой обход применяется при создании копии древовидной структуры данных, поскольку в дубликате дерева нужно создать сначала родительские узлы, а потом дочерние.

В файле `preorderTraversal.py` находится функция `preorderTraverse()`, которая сначала обрабатывает каждый дочерний узел, а затем обращается к данным узла для вывода их на экран:

```
root = {'data': 'A', 'children': [{'data': 'B', 'children':
[{'data': 'D', 'children': []}]}, {'data': 'C', 'children':
[{'data': 'E', 'children': [{'data': 'G', 'children': []},
{'data': 'H', 'children': []}]}, {'data': 'F', 'children': []}]}]}
```

```
def preorderTraverse(node):
    print(node['data'], end=' ') # Получение доступа к данным узла
 ❶ if len(node['children']) > 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        for child in node['children']:
            preorderTraverse(child) # Обход дочерних узлов
    # БАЗОВЫЙ СЛУЧАЙ
 ❷ return

preorderTraverse(root)
```

Эквивалентная программа на языке JavaScript находится в файле `preorderTraversal.html`:

```
<script type="text/javascript">
root = {"data": "A", "children": [{"data": "B", "children":
[{"data": "D", "children": []}]}, {"data": "C", "children":
[{"data": "E", "children": [{"data": "G", "children": []},
{"data": "H", "children": []}]}, {"data": "F", "children": []}]}]};

function preorderTraverse(node) {
    document.write(node["data"] + " "); // Получение доступа к данным узла
 ❶ if (node["children"].length > 0) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        for (let i = 0; i < node["children"].length; i++) {
            preorderTraverse(node["children"][i]); // Обход дочерних узлов
        }
    }
    // БАЗОВЫЙ СЛУЧАЙ
 ❷ return;
}

preorderTraverse(root);
</script>
```

В результате запуска этих программ на экран выводятся данные узлов в следующем порядке:

```
A B D C E G H F
```

Как видно на рис. 4.1, прямой обход дерева отображает данные в левых узлах раньше, чем в правых, а в нижних — раньше, чем в верхних.

Все обходы дерева начинаются с передачи корневого узла рекурсивной функции, которая выполняет рекурсивный вызов и последовательно передает в качестве аргумента все дочерние вершины корня. Поскольку у этих узлов есть собственные дочерние узлы, процесс обхода продолжается вплоть до достижения листа. В этот момент функция просто завершает свою работу.

Рекурсивный случай возникает, если у узла есть потомки ❶, каждый из которых передается в качестве аргумента в процессе выполнения рекурсивных вызовов.

Вне зависимости от наличия потомков у того или иного узла базовый случай всегда происходит на последнем этапе работы функции, когда она возвращается ❷.

Обратный обход дерева

Обратный обход дерева предполагает обход потомков перед получением доступа к данным узла. Этот алгоритм используется, например, при удалении дерева и гарантирует, что ни одна дочерняя вершина не «осиротеет» из-за удаления предков, став недоступной для корневого узла. Код программы в файле `postorderTraversal.py` аналогичен коду программы для прямого обхода дерева, представленной в предыдущем разделе. Отличие лишь в том, что вызов рекурсивной функции предшествует вызову функции `print()`:

```
root = {'data': 'A', 'children': [{'data': 'B', 'children':
[{'data': 'D', 'children': []}]}, {'data': 'C', 'children':
[{'data': 'E', 'children': [{'data': 'G', 'children': []},
{'data': 'H', 'children': []}]}, {'data': 'F', 'children': []}]}]}

def postorderTraverse(node):
    for child in node['children']:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        postorderTraverse(child) # Обход дочерних узлов
    print(node['data'], end=' ') # Получение доступа к данным узла
    # БАЗОВЫЙ СЛУЧАЙ
    return

postorderTraverse(root)
```

Эквивалентная программа на языке JavaScript содержится в файле `postorderTraversal.html`:

```
<script type="text/javascript">
root = {"data": "A", "children": [{"data": "B", "children":
[{"data": "D", "children": []}]}, {"data": "C", "children":
[{"data": "E", "children": [{"data": "G", "children": []},
{"data": "H", "children": []}]}, {"data": "F", "children": []}]}]};

function postorderTraverse(node) {
    for (let i = 0; i < node["children"].length; i++) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        postorderTraverse(node["children"][i]); // Обход дочерних узлов
    }
    document.write(node["data"] + " "); // Получение доступа к данным узла
    // БАЗОВЫЙ СЛУЧАЙ
    return;
}

postorderTraverse(root);
</script>
```

В результате запуска этих программ на экран выводятся данные узлов в следующем порядке:

```
D B G H E F C A
```

Обратный обход дерева отображает данные в левых узлах раньше, чем в правых, а в нижних — раньше, чем в верхних. Если посмотреть на функции `postorderTraverse()` и `preorderTraverse()`, то выбор их имен может показаться не очень удачным: в данном случае приставки *pre* (до) и *post* (после) не относятся к порядку, в котором обрабатываются узлы. Они всегда обходятся по одному и тому же правилу: мы перебираем дочерние узлы, спускаясь вглубь, насколько это возможно (то есть осуществляем *поиск в глубину*), а не делаем это по уровням (как при *поиске в ширину*). Приставки *pre* и *post* обозначают момент обращения к данным узла, то есть либо до, либо после обхода его потомков.

Центрированный обход дерева

Двоичные деревья представляют собой древовидные структуры данных, в которых у каждой вершины есть не более двух дочерних узлов, часто называемых *левым* и *правым* потомками. При *центрированном обходе дерева* сначала осуществляется обработка левого потомка и получение доступа к данным узла, а затем правого потомка. Такой способ обхода используется в алгоритмах, работающих с двоичными деревьями поиска (обсуждение которых выходит за рамки книги). Программа на языке Python, выполняющая такой обход, содержится в файле `inorderTraversal.py`:

```
root = {'data': 'A', 'children': [{'data': 'B', 'children':
[{'data': 'D', 'children': []}]}, {'data': 'C', 'children':
[{'data': 'E', 'children': [{'data': 'G', 'children': []},
{'data': 'H', 'children': []}]}, {'data': 'F', 'children': []}]}]}

def inorderTraverse(node):
    if len(node['children']) >= 1:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
         inorderTraverse(node['children'][0]) # Посещение левого потомка
    print(node['data'], end=' ') # Получение доступа к данным узла
    if len(node['children']) >= 2:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        inorderTraverse(node['children'][1]) # Посещение правого потомка
    # БАЗОВЫЙ СЛУЧАЙ
    return

inorderTraverse(root)
```

Эквивалентная программа на языке JavaScript содержится в файле `inorderTraversal.html`:

```
<script type="text/javascript">
root = {"data": "A", "children": [{"data": "B", "children":
[{"data": "D", "children": []}]}, {"data": "C", "children":
```

```
[{"data": "E", "children": [{"data": "G", "children": []},
{"data": "H", "children": []}]}, {"data": "F", "children": []}]}];

function inorderTraverse(node) {
    if (node["children"].length >= 1) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        inorderTraverse(node["children"][0]); // Посещение левого потомка
    }
    document.write(node["data"] + " "); // Получение доступа к данным узла
    if (node["children"].length >= 2) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        inorderTraverse(node["children"][1]); // Посещение правого потомка
    }
    // БАЗОВЫЙ СЛУЧАЙ
    return;
}

inorderTraverse(root);
</script>
```

Результат выполнения этих программ выглядит так:

```
D B A G E H C F
```

Обычно центрированный обход применяется к двоичным деревьям, однако получение данных узла после обработки первой вершины и перед последней будет считаться центрированным обходом вне зависимости от размера дерева.

Поиск восьмибуквенных имен в дереве

Вместо того чтобы выводить на экран данные каждого узла по мере их обхода, мы можем осуществить так называемый *поиск в глубину*, чтобы найти конкретный элемент в древовидной структуре данных. Давайте напишем алгоритм, который ищет имена из восьми букв в дереве, изображенном на рис. 4.4. Это довольно примитивный пример, но он наглядно демонстрирует, как алгоритм может использовать обход дерева для извлечения из него необходимых сведений.

Зададим три вопроса относительно нашего кода для обхода дерева. Ответы на них аналогичны тем, что мы давали при обсуждении предыдущего алгоритма обхода дерева.

Что представляет собой базовый случай? Либо листовой узел, который вынуждает алгоритм вернуться к предыдущему родительскому узлу, либо узел, содержащий восьмибуквенное имя.

Какой аргумент передается рекурсивной функции при ее вызове? Узел, который требуется обработать и дочерние элементы которого будут следующими узлами для обхода.

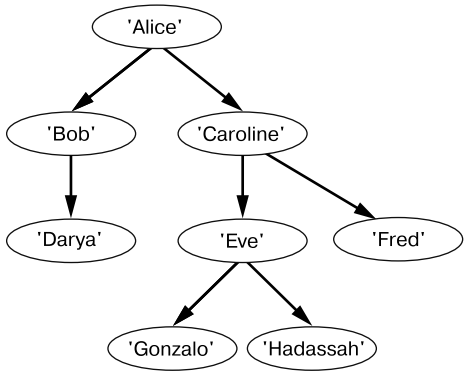


Рис. 4.4. Дерево, в котором хранятся имена, используемое в программах depthFirstSearch.py и depthFirstSearch.html

Как этот аргумент приближается к базовому случаю? Ориентированный ациклический граф не имеет циклов, поэтому последовательный обход узлов-потомков в конечном итоге приводит к достижению листового узла.

Файл `depthFirstSearch.py` содержит код программы на языке Python, которая осуществляет поиск в глубину с помощью прямого обхода дерева:

```
root = {'name': 'Alice', 'children': [{'name': 'Bob', 'children':
[{'name': 'Darya', 'children': []}]}, {'name': 'Caroline',
'children': [{'name': 'Eve', 'children': [{'name': 'Gonzalo',
'children': []}, {'name': 'Hadassah', 'children': []}]}, {'name': 'Fred',
'children': []}]}]}

def find8LetterName(node):
    print(' Visiting node ' + node['name'] + '...')

    # Поиск в глубину с помощью прямого обхода:
    print('Checking if ' + node['name'] + ' is 8 letters...')
 ❶ if len(node['name']) == 8: return node['name'] # БАЗОВЫЙ СЛУЧАЙ

    if len(node['children']) > 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        for child in node['children']:
            returnValue = find8LetterName(child)
            if returnValue != None:
                return returnValue
    # Поиск в глубину с помощью обратного обхода:
    #print('Checking if ' + node['name'] + ' is 8 letters...')
 ❷ #if len(node['name']) == 8: return node['name'] # БАЗОВЫЙ СЛУЧАЙ

    # Значение не найдено, или отсутствуют дочерние узлы
    return None # БАЗОВЫЙ СЛУЧАЙ

print('Found an 8-letter name: ' + str(find8LetterName(root)))
```

Эквивалентная программа на языке JavaScript содержится в файле `depthFirstSearch.html`:

```
<script type="text/javascript">
root = {'name': 'Alice', 'children': [{'name': 'Bob', 'children':
[{'name': 'Darya', 'children': []}]}, {'name': 'Caroline',
'children': [{'name': 'Eve', 'children': [{'name': 'Gonzalo',
'children': []}, {'name': 'Hadassah', 'children': []}]}, {'name': 'Fred',
'children': []}]}]};

function find8LetterName(node, value) {
    document.write("Visiting node " + node.name + "...<br />");

    // Поиск в глубину с помощью прямого обхода:
    document.write("Checking if " + node.name + " is 8 letters...<br />");
 ❶ if (node.name.length === 8) return node.name; // БАЗОВЫЙ СЛУЧАЙ

    if (node.children.length > 0) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        for (let child of node.children) {
            let returnValue = find8LetterName(child);
            if (returnValue != null) {
                return returnValue;
            }
        }
    }
    // Поиск в глубину с помощью обратного обхода:
    document.write("Checking if " + node.name + " is 8 letters...<br />");
 ❷ //if (node.name.length === 8) return node.name; // БАЗОВЫЙ СЛУЧАЙ

    // Значение не найдено, или отсутствуют дочерние узлы
    return null; // БАЗОВЫЙ СЛУЧАЙ
}

document.write("Found an 8-letter name: " + find8LetterName(root));
</script>
```

Результат выполнения этих программ выглядит так:

```
Visiting node Alice...
Checking if Alice is 8 letters...
Visiting node Bob...
Checking if Bob is 8 letters...
Visiting node Darya...
Checking if Darya is 8 letters...
Visiting node Caroline...
Checking if Caroline is 8 letters...
Found an 8-letter name: Caroline
```

Функция `find8LetterName()` работает так же, как и наши предыдущие функции обхода дерева, только вместо вывода на экран данных, хранящихся в узле, эта

функция проверяет их и возвращает первое обнаруженное восьмибуквенное имя. Допускается изменять способ обхода с прямого на обратный, закомментировав первую проверку длины имени и строку `Checking if` ❶ и удалив символы комментария на вторую проверку длины имени и строку `Checking if` ❷. После подобного преобразования первым обнаруживаемым восьмибуквенным именем будет `Hadassah`:

```
Visiting node Alice...
Visiting node Bob...
Visiting node Darya...
Checking if Darya is 8 letters...
Checking if Bob is 8 letters...
Visiting node Caroline...
Visiting node Eve...
Visiting node Gonzalo...
Checking if Gonzalo is 8 letters...
Visiting node Hadassah...
Checking if Hadassah is 8 letters...
Found an 8-letter name: Hadassah
```

Хотя оба подхода позволяют обнаружить восьмибуквенное значение, изменение способа обхода дерева может повлиять на поведение программы.

Определение максимальной глубины дерева

Алгоритм способен определить самую глубокую ветвь в дереве, рекурсивно запрашивая глубину дочерних вершин. *Глубина* узла определяется количеством ребер, находящихся между ним и корнем. Сам корень имеет глубину 0, а его непосредственный дочерний узел — 1 и т. д. Эта информация используется в рамках более масштабного алгоритма или для выяснения общего размера древовидной структуры данных.

Мы можем создать функцию `getDepth()`, принимающую в качестве аргумента узел и возвращающую число, равное глубине его самого глубокого дочернего узла. По достижении листа (базовый случай) эта функция будет возвращать `0`.

Например, в случае с деревом, изображенным на рис. 4.1, мы можем вызвать функцию `getDepth()`, передав ей корневой узел (`A`). В результате она возвратит значение глубины его потомков — `B` и `C`, — увеличенное на единицу. Для получения этих данных функции придется рекурсивно вызывать `getDepth()`. Рано или поздно узел `A` вызовет `getDepth()` для `C`, который, в свою очередь, вызовет ее для вершины `E`. Когда `E` вызовет `getDepth()` для двух своих дочерних элементов — `G` и `H`, — они оба возвратят значение `0`, поэтому функция `getDepth()`, вызванная для `E`, возвратит `1`, что вынудит функцию, вызванную для `C`, возвратить `2`, а функцию для `A` (корень) — `3`. Таким образом, максимальная глубина нашего дерева составит три уровня.

Зададим наши три вопроса относительно рекурсивной функции `getDepth()`.

Что представляет собой базовый случай? Листовой узел, не имеющий потомков, глубина которого по определению составляет один уровень.

Какой аргумент передается рекурсивной функции при ее вызове? Узел, максимальную глубину которого мы хотим определить.

Как этот аргумент приближается к базовому случаю? Ориентированный ациклический граф не имеет циклов, поэтому последовательный обход вершин-потомков в конечном итоге приводит к достижению листового узла.

Следующая программа из файла `getDepth.py` содержит рекурсивную функцию `getDepth()`, которая возвращает количество уровней в самой глубокой ветви дерева:

```
root = {'data': 'A', 'children': [{'data': 'B', 'children':
[{'data': 'D', 'children': []}]}, {'data': 'C', 'children':
[{'data': 'E', 'children': [{'data': 'G', 'children': []},
{'data': 'H', 'children': []}]}, {'data': 'F', 'children': []}]}]}

def getDepth(node):
    if len(node['children']) == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return 0
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        maxChildDepth = 0
        for child in node['children']:
            # Определение глубины каждого из дочерних узлов:
            childDepth = getDepth(child)
            if childDepth > maxChildDepth:
                # Этот дочерний узел самый глубокий из найденных до сих пор:
                maxChildDepth = childDepth
        return maxChildDepth + 1

print('Depth of tree is ' + str(getDepth(root)))
```

Эквивалентная программа на языке JavaScript содержится в файле `getDepth.html`:

```
<script type="text/javascript">
root = {"data": "A", "children": [{"data": "B", "children":
[{"data": "D", "children": []}]}, {"data": "C", "children":
[{"data": "E", "children": [{"data": "G", "children": []},
{"data": "H", "children": []}]}, {"data": "F", "children": []}]}]};

function getDepth(node) {
    if (node.children.length === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
        return 0;
    } else {
```

```
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        let maxChildDepth = 0;
        for (let child of node.children) {
            // Определение глубины каждого из дочерних узлов:
            let childDepth = getDepth(child);
            if (childDepth > maxChildDepth) {
                // Этот дочерний узел самый глубокий из найденных до сих пор:
                maxChildDepth = childDepth;
            }
        }
        return maxChildDepth + 1;
    }
}

document.write("Depth of tree is " + getDepth(root) + "<br />");
</script>
```

Результат выполнения этих программ выглядит следующим образом:

```
Depth of tree is 3
```

Он соответствует тому, что мы видим на рис. 4.2: количество уровней от корня `A` до самых нижних узлов `G` и `H` равно трем.

Прохождение лабиринтов

Хотя лабиринты бывают разных форм и размеров, *односвязные лабиринты*, также называемые *идеальными*, не содержат замкнутых маршрутов. В этом лабиринте между любыми двумя точками (например, между входом и выходом) есть только один путь. Такие лабиринты могут быть представлены ориентированным ациклическим графом (DAG).

Например, на рис. 4.5 показан лабиринт, который проходит наша программа, а на рис. 4.6 он представлен в виде DAG. Заглавной буквой *S* на рис. 4.6 обозначен вход в лабиринт, а заглавной буквой *E* — выход из него. Несколько развилок в лабиринте, отмеченных строчными буквами, соответствуют узлам графа.

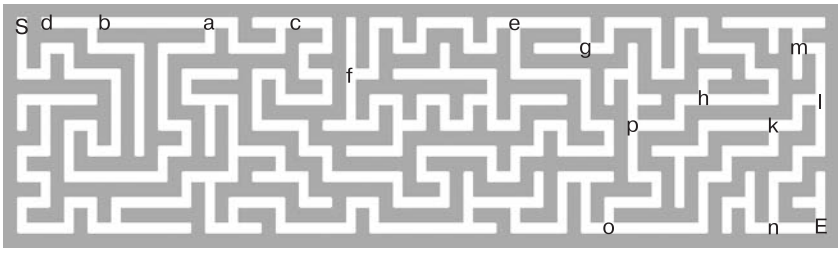


Рис. 4.5. Лабиринт, который проходит программа, описанная в этой главе. Развилки, обозначенные строчными буквами, соответствуют узлам графа на рис. 4.6

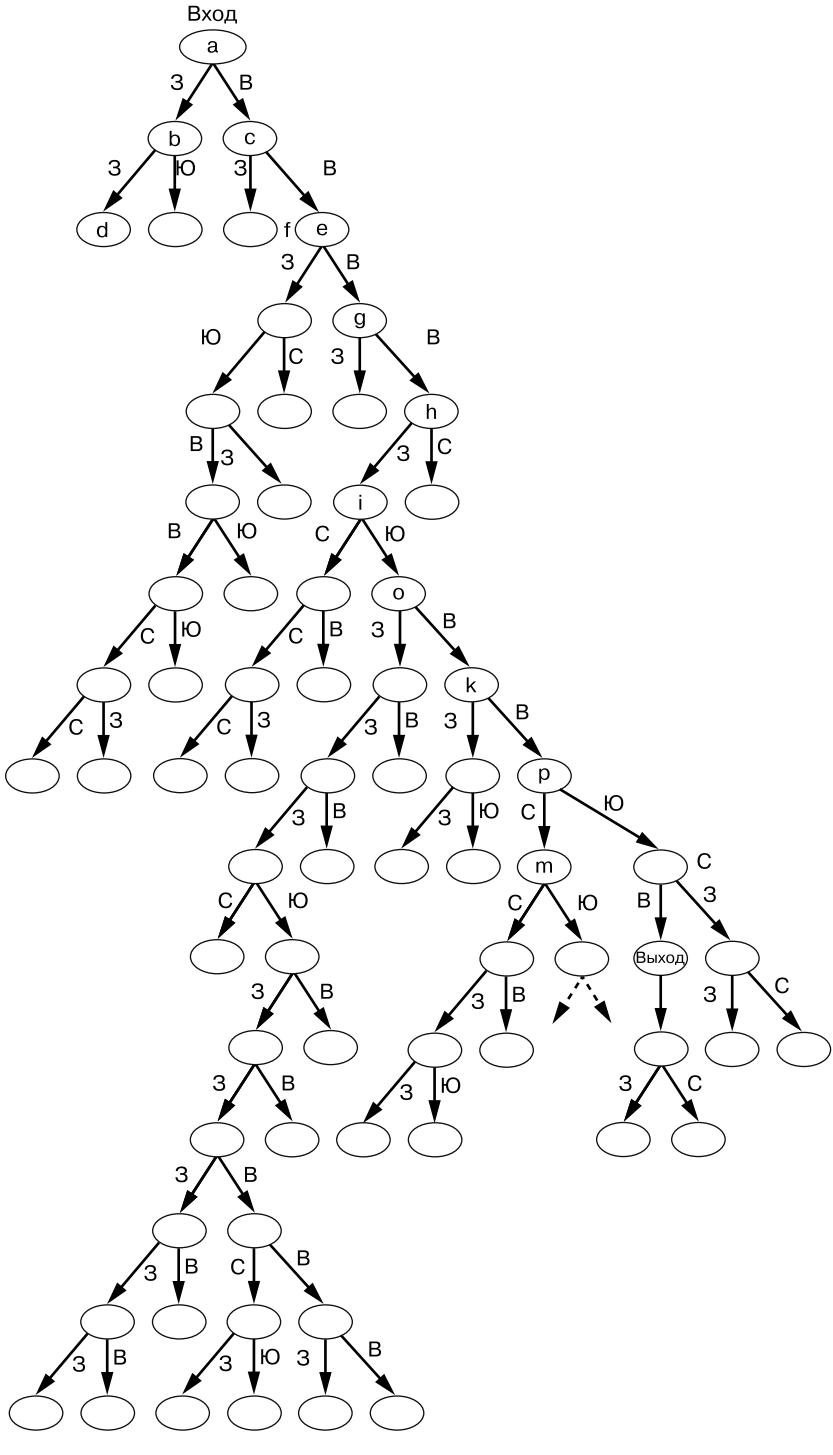


Рис. 4.6. В данном графе узлы обозначают развилки, а ребра — северное, южное, восточное или западное направления движения. Узлы, обозначенные строчными буквами, соответствуют развилкам на рис. 4.5

Из-за сходства в структуре допускается использовать алгоритм обхода дерева для прохождения лабиринта. Узлы этого древовидного графа представляют собой развилки, в которых можно выбрать северное, южное, восточное или западное направление дальнейшего следования. Корневой узел олицетворяет вход в лабиринт, а листья — тупики.

Рекурсивный случай наступает, когда алгоритм обхода дерева переходит от одного узла к другому. Если алгоритм натыкается на листовой узел (тупик), он достигает базового случая и возвращается к более раннему узлу, чтобы выбрать другой путь. Как только алгоритм достигает узла, соответствующего выходу из лабиринта, маршрут от корня до этого узла представляет собой путь его прохождения.

Зададим наши три вопроса относительно рекурсивного алгоритма для прохождения лабиринта.

Что представляет собой базовый случай? Достижение тупика или выхода из лабиринта.

Какой аргумент передается рекурсивной функции при ее вызове? Координаты x, y, а также данные лабиринта и список уже посещенных координат x, y.

Как этот аргумент приближается к базовому случаю? Как и в случае с заливкой, алгоритм продолжает перебирать координаты соседних точек до тех пор, пока не достигнет тупика или выхода из лабиринта.

Файл `mazeSolver.py` содержит код программы на языке Python для прохождения лабиринта, хранящегося в переменной `MAZE`:

```
# Создаем структуру данных лабиринта:
# вы можете скопировать ее со страницы inventwithpython.com/examplemaze.txt
MAZE = """
##########################################################################
#S#                 #       # #   #     #         #     #   #         #
# ##### ######### # ### ### # # # # ### # # ##### # ### # # ##### # ###
# #   #     #     #     #   # # #   # #   # #       # # # #     # #   #
# # # ##### # ########### ### # ##### ##### ######### # # ##### ### # #
#   #     # # #     #   #   #   #         #       #   #   #   #   # # #
######### # # # ##### # ### # ########### ####### # # ##### ##### ### #
#       # # # #     # #     # #   #   #   #     # # #   #         #   #
# # ##### # # ### # # ####### # # # # # # # ##### ### ### ######### # #
# # #   # # #   # # #     #     #   #   #   #   #   #     #         # #
### # # # # ### # # ##### ####### ########### # ### # ##### ##### ### #
#   # #   # #   # #     #   #     #       #   #     # #     #     #   #
# ### ####### ##### ### ### ####### ##### # ######### ### ### ##### ###
#   #         #     #     #       #   # #   # #     #   # #   # #   # #
### ########### # ####### ####### ### # ##### # # ##### # # ### # ### #
#   #   #       # #     #   #   #     #       # # #     # # #   # #   #
# ### # # ####### # ### ##### # ####### ### ### # # ####### # # # ### #
#     #         #     #       #           #     #           # #      E#
##########################################################################
""".split('\n')
```

```
# Константы, используемые в этой программе:
EMPTY = ' '
START = 'S'
EXIT = 'E'
PATH = '.'

# Получаем значение высоты и ширины лабиринта:
HEIGHT = len(MAZE)
WIDTH = 0
for row in MAZE: # Задаем для WIDTH значение ширины самой широкой строки
    if len(row) > WIDTH:
        WIDTH = len(row)
# Превращаем каждую строку в лабиринте в список шириной WIDTH:
for i in range(len(MAZE)):
    MAZE[i] = list(MAZE[i])
    if len(MAZE[i]) != WIDTH:
        MAZE[i] = [EMPTY] * WIDTH # Делаем эту строку пустой

def printMaze(maze):
    for y in range(HEIGHT):
        # Выводим на экран каждую строку.
        for x in range(WIDTH):
            # Выводим на экран каждый столбец в этой строке
            print(maze[y][x], end='')
        print() # Добавляем в конце строки символ перехода на новую строку
    print()
def findStart(maze):
    for x in range(WIDTH):
        for y in range(HEIGHT):
            if maze[y][x] == START:
                return (x, y) # Возвращаем координаты входа в лабиринт

def solveMaze(maze, x=None, y=None, visited=None):
    if x == None or y == None:
        x, y = findStart(maze)
        maze[y][x] = EMPTY # Избавляемся от буквы "S" в лабиринте
    if visited == None:
     ❶ visited = [] # Создаем новый список посещенных точек

    if maze[y][x] == EXIT:
         return True # Выход найден, возвращаем значение True

    maze[y][x] = PATH # Отмечаем путь в лабиринте.
 ❷ visited.append(str(x) + ',' + str(y))
 ❸ #printMaze(maze) # Раскомментируем, чтобы просмотреть каждый шаг вперед

    # Проверяем северную соседнюю точку:
    if y + 1 < HEIGHT and maze[y + 1][x] in (EMPTY, EXIT) and \
    str(x) + ',' + str(y + 1) not in visited:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        if solveMaze(maze, x, y + 1, visited):
            return True # БАЗОВЫЙ СЛУЧАЙ
```

```
    # Проверяем южную соседнюю точку:
    if y - 1 >= 0 and maze[y - 1][x] in (EMPTY, EXIT) and \
    str(x) + ',' + str(y - 1) not in visited:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        if solveMaze(maze, x, y - 1, visited):
            return True # БАЗОВЫЙ СЛУЧАЙ
    # Проверяем восточную соседнюю точку:
    if x + 1 < WIDTH and maze[y][x + 1] in (EMPTY, EXIT) and \
    str(x + 1) + ',' + str(y) not in visited:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        if solveMaze(maze, x + 1, y, visited):
            return True # БАЗОВЫЙ СЛУЧАЙ
    # Проверяем западную соседнюю точку:
    if x - 1 >= 0 and maze[y][x - 1] in (EMPTY, EXIT) and \
    str(x - 1) + ',' + str(y) not in visited:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        if solveMaze(maze, x - 1, y, visited):
            return True # БАЗОВЫЙ СЛУЧАЙ

    maze[y][x] = EMPTY # Заменяем пробелы точками
❹ #printMaze(maze) # Раскомментируйте, чтобы просмотреть каждый шаг назад

    return False # БАЗОВЫЙ СЛУЧАЙ

printMaze(MAZE)
solveMaze(MAZE)
printMaze(MAZE)
```

Эквивалентная программа на языке JavaScript содержится в файле `mazeSolver.html`:

```
<script type="text/javascript">
// Создаем структуру данных лабиринта:
// можете скопировать ее со страницы inventwithpython.com/examplemaze.txt
let MAZE = `
##############################################################################
#S#                 #       # #   #     #         #     #   #         #
# ##### ######### # ### ### # # # # ### # # ##### # ### # # ##### # ###
# #   #     #     #     #   # # #   # #   # #       # # # #     # #   #
# # # ##### # ########### ### # ##### ##### ######### # # ##### ### # #
#   #     # # #     #   #   #   #         #       #   #   #   #   # # #
######### # # # ##### # ### # ########### ####### # # ##### ##### ### #
#       # # # #     # #     # #   #   #   #     # # #   #         #   #
# # ##### # # ### # # ####### # # # # # # ##### ### ### ######### # #
# # #   # # #   # # #     #     #   #   #   #   #   #     #         # #
### # # # # ### # # ##### ####### ########### # ### # ##### ##### ### #
#   # #   # #   # #     #   #     #       #   #     # #     #     #   #
# ### ######## ##### ### ### ####### ##### # ######### ### ### ##### ###
#   #         #     #     #       #   # #   # #     #   # #   # #   # #
### ########### # ####### ####### ### # ##### # # ##### # # ### # ### #
#   #   #       # #     #   #   #     #       # # #     # # #   # #   #
# ### # # ####### # ### ##### # ####### ### ### # # ####### # # # ### #
#     #         #     #       #           #     #           # #      E#
##############################################################################
`.split("\n");
```

```
// Константы, используемые в этой программе:
const EMPTY = " ";
const START = "S";
const EXIT = "E";
const PATH = ".";

// Получаем значение высоты и ширины лабиринта:
const HEIGHT = MAZE.length;
let maxWidthSoFar = MAZE[0].length;
for (let row of MAZE) { // Задаем для WIDTH значение ширины самой широкой строки
    if (row.length > maxWidthSoFar) {
        maxWidthSoFar = row.length;
    }
}
const WIDTH = maxWidthSoFar;
// Превращаем каждую строку в лабиринте в список шириной WIDTH:
for (let i = 0; i < MAZE.length; i++) {
    MAZE[i] = MAZE[i].split("");
    if (MAZE[i].length !== WIDTH) {
        MAZE[i] = EMPTY.repeat(WIDTH).split(""); // Делаем эту строку пустой
    }
}

function printMaze(maze) {
    document.write("<pre>");
    for (let y = 0; y < HEIGHT; y++) {
        // Выводим на экран каждую строку
        for (let x = 0; x < WIDTH; x++) {
            // Выводим на экран каждый столбец в этой строке
            document.write(maze[y][x]);
        }
        document.write("\n"); // Добавляем в конце строки символ перехода
                              // на новую строку
    }
    document.write("\n</ pre>");
}

function findStart(maze) {
    for (let x = 0; x < WIDTH; x++) {
        for (let y = 0; y < HEIGHT; y++) {
            if (maze[y][x] === START) {
                return [x, y]; // Возвращаем координаты входа в лабиринт
            }
        }
    }
}

function solveMaze(maze, x, y, visited) {
    if (x === undefined || y === undefined) {
        [x, y] = findStart(maze);
        maze[y][x] = EMPTY; // Избавляемся от буквы S в лабиринте
    }
```

```
    if (visited === undefined) {
     ❶ visited = []; // Создаем новый список посещенных точек
    }

    if (maze[y][x] == EXIT) {
         return true; // Выход найден, возвращаем значение True
    }

    maze[y][x] = PATH; // Отмечаем путь в лабиринте
❷ visited.push(String(x) + "," + String(y));
❸ //printMaze(maze) // Раскомментируйте, чтобы просмотреть каждый шаг вперед

    // Проверяем северную соседнюю точку:
    if ((y + 1 < HEIGHT) && ((maze[y + 1][x] == EMPTY) ||
    (maze[y + 1][x] == EXIT)) &&
    (visited.indexOf(String(x) + "," + String(y + 1)) === -1)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        if (solveMaze(maze, x, y + 1, visited)) {
            return true; // БАЗОВЫЙ СЛУЧАЙ
        }
    }
    // Проверяем южную соседнюю точку:
    if ((y - 1 >= 0) && ((maze[y - 1][x] == EMPTY) ||
    (maze[y - 1][x] == EXIT)) &&
    (visited.indexOf(String(x) + "," + String(y - 1)) === -1)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        if (solveMaze(maze, x, y - 1, visited)) {
            return true; // БАЗОВЫЙ СЛУЧАЙ
        }
    }
    // Проверяем восточную соседнюю точку:
    if ((x + 1 < WIDTH) && ((maze[y][x + 1] == EMPTY) ||
    (maze[y][x + 1] == EXIT)) &&
    (visited.indexOf(String(x + 1) + "," + String(y)) === -1)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        if (solveMaze(maze, x + 1, y, visited)) {
            return true; // БАЗОВЫЙ СЛУЧАЙ
        }
    }
    // Проверяем западную соседнюю точку:
    if ((x - 1 >= 0) && ((maze[y][x - 1] == EMPTY) ||
    (maze[y][x - 1] == EXIT)) &&
    (visited.indexOf(String(x - 1) + "," + String(y)) === -1)) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        if (solveMaze(maze, x - 1, y, visited)) {
            return true; // БАЗОВЫЙ СЛУЧАЙ
        }
    }

    maze[y][x] = EMPTY; // Заменяем пробелы точками
❹ //printMaze(maze); // Раскомментируем, чтобы просмотреть каждый шаг назад
    return false; // БАЗОВЫЙ СЛУЧАЙ
}
```

```
printMaze(MAZE);
solveMaze(MAZE);
printMaze(MAZE);
</script>
```

Большая часть кода из листинга выше не имеет непосредственного отношения к рекурсивному алгоритму прохождения лабиринта. В переменной `MAZE` хранятся данные лабиринта в виде многострочного набора символов (так сказать, многострочной строки). Символы `#` обозначают его стены, а буквы `S` и `E` — вход и выход соответственно. Эта строка преобразуется в список, содержащий перечень строк, каждая из которых представляет один символ в лабиринте. Благодаря чему мы получаем доступ к `MAZE[y][x]` (обратите внимание, что координата `y` идет первой), чтобы определить символ, находящийся в точке с координатами x и y в исходной строке `MAZE`. Функция `printMaze()` может принять данный список списков и отобразить лабиринт на экране. Функция `findStart()` принимает эту структуру данных и возвращает координаты x и y начальной точки `S`. Вы можете отредактировать строку лабиринта самостоятельно, однако помните, что алгоритм прохождения лабиринта сработает только при отсутствии в нем замкнутых маршрутов.

Рекурсивный алгоритм находится в функции `solveMaze()`, аргументами которой выступают структура данных лабиринта, текущие координаты x и y, а также список `visited` (создается, если ни один не был предоставлен) ❶. Список `visited` содержит координаты всех ранее посещенных точек, поэтому при возвращении из тупика к более ранней развилке алгоритм «помнит», какие маршруты он уже проходил, и может попробовать другой путь. Маршрут от входа до выхода помечается заменой пробелов (соответствующих константе `EMPTY`) в структуре данных лабиринта точками (из константы `PATH`).

Алгоритм прохождения лабиринта похож на алгоритм заливки из главы 3 в том смысле, что он «распространяется» на соседние точки и, достигая тупика, возвращается к предыдущей развилке. Функция `solveMaze()` получает координаты x и y, указывающие фактическое местоположение обрабатываемой алгоритмом в данный момент точки. Если это выход, функция возвращает `True`, в результате чего все рекурсивные вызовы также возвращают `True`. А в структуре данных лабиринта сохраняется разметка пути его прохождения.

В противном случае алгоритм помечает текущие координаты x и y в структуре данных лабиринта точкой и добавляет их в список `visited` ❷. Затем он проверяет координаты соседней позиции, находящейся к северу от актуальной, чтобы выяснить, не находится ли она за границей карты, соответствует ли она пробелу или выходу из лабиринта и не посещалась ли она раньше. Если условия выполняются, алгоритм совершает рекурсивный вызов функции `solveMaze()`, передавая ей координаты северной точки. Если же условия не выполняются или рекурсивный вызов

`solveMaze()` возвращает `False`, алгоритм проверяет координаты точек, находящихся к югу, востоку и западу. Как и в случае с алгоритмом заливки, рекурсивные вызовы выполняются с использованием соседних координат.

ИЗМЕНЕНИЕ СПИСКА ИЛИ МАССИВА НА МЕСТЕ

В момент вызова функции язык Python/JavaScript передает не копии списков/массивов, а ссылку на них. Поэтому любые изменения, сделанные в списке или массиве (вроде `maze` и `visited`), остаются в силе даже после возврата из функции. Это называется изменением списка *на месте*. В случае с рекурсивными функциями структуру данных лабиринта и множество посещенных точек можно рассматривать как единую копию, совместно используемую всеми рекурсивными вызовами, в отличие от аргументов *x* и *y*. Вот почему структура данных, хранящаяся в переменной `MAZE`, продолжает меняться после завершения первого вызова `solveMaze()`.

Чтобы лучше понять принцип работы алгоритма, раскомментируйте два вызова `printMaze(MAZE)` ❸ ❹ внутри функции `solveMaze()`. Они будут отображать изменения в структуре данных лабиринта по мере того, как алгоритм пытается найти новые пути, достигает тупиков, возвращается назад и пробует другие маршруты.

Резюме

Мы рассмотрели несколько алгоритмов, использующих древовидные структуры данных и поиск с возвратом. Мы обсудили древовидные структуры данных, состоящие из узлов, содержащих данные, и ребер, связывающих родительские узлы с дочерними. В частности, мы изучили особый тип дерева, называемый ориентированным ациклическим графом (DAG), который часто задействуется в рекурсивных алгоритмах. Вызов рекурсивной функции схож с переходом к дочернему узлу в дереве, а возврат из этого вызова аналогичен возвращению к предыдущему родительскому узлу.

Рекурсией нередко злоупотребляют при решении простых задач программирования, тогда как ее применение лучше всего подходит для проблем, предполагающих использование древовидных структур данных и поиска с возвратом. Опираясь на эти идеи, мы написали несколько алгоритмов обхода, поиска и определения глубины древовидных структур. Было также показано, что односвязный лабиринт имеет древовидную структуру и то, как рекурсия и поиск с возвратом помогают его пройти.

Дополнительные источники информации

Информация о деревьях и способах их обхода не ограничивается кратким описанием DAG, представленным в пройденной главе. Дополнительные сведения вы можете найти в «Википедии» по адресам https://ru.wikipedia.org/wiki/Дерево_(структура_данных) и https://ru.wikipedia.org/wiki/Обход_дерева.

Кроме того, рассмотренные концепции обсуждаются в видео *Maze Solving* на YouTube-канале Computerphile (https://youtu.be/rop0W4QDOUI). Антон Спрол, автор книги «Думай как программист», также опубликовал видео о прохождении лабиринтов под названием *Backtracking* (https://youtu.be/gBC_Fd8EE8A). А еще организация freeCodeCamp (https://freeCodeCamp.org) выпустила целую серию видеороликов, посвященных алгоритмам поиска с возвратом, которые доступны по адресу https://youtu.be/A80YzvNwqXA.

Алгоритм рекурсивного поиска с возвратом позволяет не только проходить, но и создавать лабиринты. Подробнее об этом и многом другом вы можете узнать в «Википедии»: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_backtracker.

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое узлы и ребра?
2. Что такое корневой узел и листья?
3. Каковы три способа обхода дерева?
4. Что такое DAG?
5. Что такое цикл и есть ли циклы в DAG?
6. Что такое двоичное дерево?
7. Как называются дочерние узлы в двоичном дереве?
8. Если родительский узел соединен ребром с дочерним узлом, а этот дочерний узел имеет ребро, ведущее обратно к родительскому узлу, считается ли данный граф ориентированным и ациклическим?
9. Что такое поиск с возвратом в контексте обхода дерева?

   При решении следующих задач, связанных с обходом дерева, можете использовать код Python/JavaScript из раздела «Древовидная структура данных в Python и JavaScript» в качестве дерева и многострочную строку `MAZE` из программ `mazeSolver.py` и `mazeSolver.html` в качестве данных лабиринта.

10. Ответьте на следующие три вопроса относительно каждого из рекурсивных алгоритмов, представленных в пройденной главе.

    А. Что представляет собой базовый случай?

    Б. Какой аргумент передается рекурсивной функции при ее вызове?

    В. Как этот аргумент приближается к базовому случаю?

    Затем воссоздайте рекурсивные алгоритмы, описанные в главе, не заглядывая в исходный код.

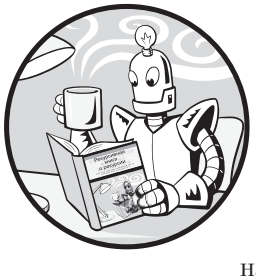
Практика

Решите каждую из следующих задач.

1. Напишите функцию поиска, которая использует центрированный обход дерева, но посещает правый дочерний узел перед левым.
2. Создайте функцию, которая, получив в качестве аргумента корневой узел, делает дерево на один уровень глубже, добавляя один дочерний узел к каждому его листу. Эта функция должна выполнить обход дерева, обнаружить лист, а затем добавить к нему одну-единственную дочернюю вершину. Удостоверьтесь в том, что потомки не будут добавлены к новому листу, так как это может привести к переполнению стека.

5

Алгоритмы типа «разделяй и властвуй»



Алгоритмы «*разделяй и властвуй*» разбивают крупные задачи на более мелкие вплоть до таких, которые предполагают тривиальное решение. Данный подход идеально сочетается с рекурсией: рекурсивный случай разделяет задачу на самоподобные подзадачи, а базовый случай соответствует самой маленькой из них, решение которой является тривиальным. Одно из преимуществ описываемого подхода состоит в том, что получившиеся фрагментарные задачи можно решать параллельно, используя несколько ядер центрального процессора (ЦП) или компьютеров.

В текущей главе мы рассмотрим несколько распространенных рекурсивных алгоритмов, использующих подход «разделяй и властвуй», в том числе двоичный поиск, быструю сортировку и сортировку слиянием. Мы также применим данный подход к задаче суммирования массива целых чисел. Наконец, познакомимся с алгоритмом Карацубы, разработанным в 1960 году, который сделал возможным быстрое перемножение целых чисел с помощью компьютерного оборудования.

Двоичный поиск: поиск среди книг, упорядоченных по алфавиту

Представьте, что у вас есть полка с сотней томов. Вы не помните, какие именно книги на ней стоят, но знаете, что они отсортированы по названиям в алфавитном порядке. К поиску произведения «Язык цветов» вы, скорее всего, приступите не с первой полки, где стоит, например, «Алиса в Зазеркалье», а с конечной. Возможно, роман «Язык цветов» окажется не последним, если у вас есть и другие, начинающиеся

с буквы «я», однако сам факт того, что книги расположены в алфавитном порядке, а «я» является последней буквой алфавита, подсказывает вам, с какого места следует начать поиск.

Двоичный (или бинарный) поиск — это тип поискового алгоритма, который последовательно делит пополам заранее отсортированный массив (список), чтобы обнаружить нужный элемент. Самый непредвзятый способ поиска из нашего примера заключается в том, чтобы начать с книги в середине, а затем выяснить, с какой стороны от нее находится искомая книга.

Затем вы можете повторить данный процесс, как показано на рис. 5.1: посмотрите на том в середине выбранной вами половины, после чего определите, в какой части от него находится искомая книга — в левой или в правой. Алгоритм выполняется до тех пор, пока не будут найдены нужная книга или место, в котором она должна была бы находиться, если книги на полке нет.

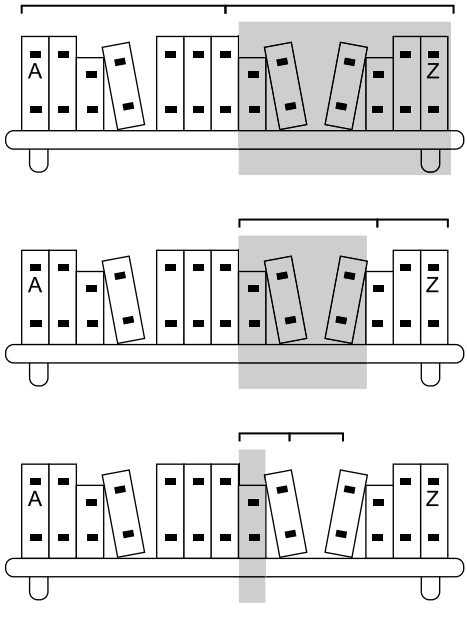


Рис. 5.1. Двоичный поиск в отсортированном массиве элементов предполагает многократное дробление массива пополам и определение, в какой половине содержится нужный элемент

Подобный подход довольно эффективно масштабируется: удвоение количества книг добавляет всего один этап в процесс поиска. В случае с линейным поиском процедура нахождения нужного произведения среди 50 книг состоит из 50 этапов,

а среди 100 — из 100. В случае с двоичным поиском алгоритм нахождения целевой книги среди 50 томов состоит из шести этапов, а среди 100 — всего из семи.

Зададим наши три вопроса относительно реализации двоичного поиска.

Что представляет собой базовый случай? Поиск в диапазоне длиной в один элемент.

Какой аргумент передается рекурсивной функции при ее вызове? Индексы левого и правого концов диапазона в списке, в котором мы осуществляем поиск.

Как этот аргумент приближается к базовому случаю? При каждом рекурсивном вызове диапазон уменьшается вдвое, в конечном итоге стремясь к единице.

Изучите следующую функцию `binarySearch()` в файле `binarySearch.py`, которая находит значение `needle` (игла) в отсортированном списке значений `haystack` (стог сена):

```
def binarySearch(needle, haystack, left=None, right=None):
    # По умолчанию значения 'left' и 'right' охватывают весь список 'haystack':
    if left is None:
        left = 0 # По умолчанию 'left' соответствует индексу нулевого элемента
    if right is None:
        right = len(haystack) - 1 # По умолчанию 'right' соответствует
                                  # индексу последнего

    print('Searching:', haystack[left:right + 1])

    if left > right: # БАЗОВЫЙ СЛУЧАЙ
         return None # Значение 'needle' отсутствует в списке 'haystack'

    mid = (left + right) // 2
    if needle == haystack[mid]: # БАЗОВЫЙ СЛУЧАЙ
         return mid # Значение 'needle' найдено в списке 'haystack'
    elif needle < haystack[mid]: # РЕКУРСИВНЫЙ СЛУЧАЙ
         return binarySearch(needle, haystack, left, mid - 1)
    elif needle > haystack[mid]: # РЕКУРСИВНЫЙ СЛУЧАЙ
         return binarySearch(needle, haystack, mid + 1, right)

print(binarySearch(13, [1, 4, 8, 11, 13, 16, 19, 19]))
```

Эквивалентный код на языке JavaScript содержится в файле `binarySearch.html`:

```
<script type="text/javascript">
function binarySearch(needle, haystack, left, right) {
    // По умолчанию значения 'left' и 'right' охватывают весь список 'haystack':
    if (left === undefined) {
        left = 0; // По умолчанию 'left' соответствует индексу нулевого элемента
    }
```

```
    if (right === undefined) {
        right = haystack.length - 1; // По умолчанию 'right' соответствует
                                     // индексу последнего
    }    document.write("Searching: [" +
    haystack.slice(left, right + 1).join(", ") + "]<br />");

    if (left > right) { // БАЗОВЫЙ СЛУЧАЙ
         return null; // Значение 'needle' отсутствует в списке 'haystack'
    }

    let mid = Math.floor((left + right) / 2);
    if (needle == haystack[mid]) { // БАЗОВЫЙ СЛУЧАЙ
         return mid; // Значение 'needle' найдено в списке 'haystack'
    } else if (needle < haystack[mid]) { // РЕКУРСИВНЫЙ СЛУЧАЙ
         return binarySearch(needle, haystack, left, mid - 1);
    } else if (needle > haystack[mid]) { // РЕКУРСИВНЫЙ СЛУЧАЙ
         return binarySearch(needle, haystack, mid + 1, right);
    }
}

document.write(binarySearch(13, [1, 4, 8, 11, 13, 16, 19, 19]));
</script>
```

После запуска эти программы осуществляют поиск значения `13` в списке `[1, 4, 8, 11, 13, 16, 19, 19]` и выводят на экран следующий результат.

```
Searching: [1, 4, 8, 11, 13, 16, 19, 19]
Searching: [13, 16, 19, 19]
Searching: [13]
4
```

Искомое значение `13` действительно четвертый элемент списка.

Код вычисляет индекс среднего элемента диапазона, ограниченного левым (`left`) и правым (`right`) индексами, и сохраняет его в переменной `mid`. На первом этапе длина этого диапазона соответствует длине всего списка элементов. Если значение индекса `mid` совпадает с `needle`, то возвращается `mid`. В противном случае нам нужно выяснить, в какой половине диапазона находится наше целевое значение: если в левой, то диапазон для поиска ограничивается индексами от `left` до `mid - 1`, а если в правой, то от `mid + 1` до `end`.

Для осуществления поиска в этом новом диапазоне мы можем воспользоваться уже имеющейся у нас функцией `binarySearch()`, выполнив ее рекурсивный вызов. Если вдруг правый конец диапазона поиска оказался перед левым, это говорит о том, что его длина сократилась до нуля, а искомое значение в нем отсутствует.

Обратите внимание, что код не выполняет никаких действий после завершения рекурсивного вызова, а сразу же возвращает значение. То есть мы можем реализовать

для рассматриваемого рекурсивного алгоритма оптимизацию хвостовых вызовов, о чем поговорим в главе 8. Это также означает, что двоичный поиск легко реализуется в виде итеративного алгоритма, который не использует рекурсивные вызовы функций. Загружаемые ресурсы для этой книги, доступные по ссылке https://nostarch.com/recursive-book-recursion, содержат исходный код итеративного алгоритма двоичного поиска, который вы можете сравнить с описанным выше рекурсивным алгоритмом.

ОЦЕНКА АЛГОРИТМА С ПОМОЩЬЮ НОТАЦИИ «О БОЛЬШОЕ»

Если ваши данные уже отсортированы, двоичный поиск занимает гораздо меньше времени, чем линейный, в ходе которого алгоритм проверяет каждое значение в массиве методом грубой силы. Мы можем сравнить эффективность подобных алгоритмов в терминах так называемой нотации «О большое». В разделе «Дополнительные источники информации» в конце главы вы найдете ссылки на ресурсы, посвященные этой теме.

Если же ваши данные еще не отсортированы, то их предварительное ранжирование (с помощью алгоритма быстрой сортировки или сортировки слиянием) и двоичный поиск займут больше времени, чем простой линейный поиск. Тем не менее, если вы собираетесь искать данные многократно, выигрыш в производительности за счет использования двоичного поиска окупит временные затраты на сортировку. Это все равно что потратить час на заточку топора, перед тем как рубить деревья: увеличение скорости рубки острым топором компенсирует час, потраченный на его заточку.

Быстрая сортировка: разделение несортированной стопки книг на отсортированные стопки

Помните, что преимущество функции `binarySearch()` в скорости обусловлено тем, что значения в списках уже отсортированы. Если они никак не упорядочены, описанный алгоритм не будет работать. В такой ситуации может помочь *быстрая сортировка* — рекурсивный алгоритм сортировки, разработанный информатиком Тони Хоаром в 1959 году.

Сортировка Хоара использует метод «разделяй и властвуй», называемый *разбиением*. Представьте, что у вас есть куча иностранной литературы. Если вы будете брать по одной книге и ставить их в нужное место на полке, то по мере ее заполнения

вам придется тратить больше времени на их переупорядочение. Проще было бы сначала разделить всю кипу на две поменьше: от *A* до *M* и от *N* до *Z*, где *M* будет нашим *опорным элементом*.

При этом мы занимаемся не сортировкой, а *разбиением*, которое не составляет никакого труда: книгу не обязательно помещать в правильное место в одной из двух стопок, достаточно положить ее в нужную. Затем мы разделяем эти две кучки на четыре: от *A* до *G*, от *H* до *M*, от *N* до *T* и от *U* до *Z*, как показано на рис. 5.2. Если продолжить разбиение, то в конечном итоге получим отсортированные стопки, состоящие из одной книги (базовый случай). Это означает, что книги тоже будут ранжированы. Именно такое многократное разбиение лежит в основе метода быстрой сортировки.

Чтобы разбить группу *A — Z*, мы выбираем в качестве опорного элемента букву *M*, потому что она находится посередине. Однако если бы наша коллекция включала одну книгу на букву *A* и 99 книг на букву *Z*, то получившиеся в результате разбиения две стопки оказались бы несбалансированными. Первая — от *A* до *M* — состояла бы всего из одной книги, а вторая — от *M* до *Z* — из всех остальных. Алгоритм быстрой сортировки работает быстрее всего, когда получающиеся в результате разбиения наборы оказываются сбалансированными, поэтому на каждом этапе важно выбирать подходящий опорный элемент.

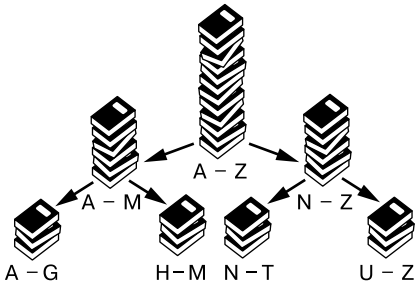


Рис. 5.2. Быстрая сортировка предполагает многократное разбиение коллекции элементов на два набора

Однако, если вам ничего не известно о сортируемых данных, вы не сможете выбрать идеальный опорный элемент. Поэтому вместо него универсальный алгоритм быстрой сортировки использует последнее значение в диапазоне.

В нашей реализации при каждом вызове `quicksort()` передается массив элементов для сортировки. Как и в случае с функцией `binarySearch()`, ей также передаются аргументы `left` и `right`, определяющие диапазон индексов элементов, подлежащих сортировке. Алгоритм выбирает опорный элемент для сравнения с другими

значениями в диапазоне, а затем помещает их либо в левую часть диапазона (если они меньше опорного значения), либо в правую (если они превышают опорное значение). Так выглядит этап разбиения. Затем функция `quicksort()` рекурсивно вызывается для двух меньших диапазонов до тех пор, пока длина диапазона не сократится до нуля. По мере выполнения рекурсивных вызовов список становится все более отсортированным, и в конце концов все его элементы оказываются расположенными в правильном порядке.

Обратите внимание, что алгоритм модифицирует массив на месте. Для получения дополнительных сведений об этом см. врезку «Изменение списка или массива на месте» в главе 4. Таким образом, `quicksort()` не возвращает отсортированный массив. Достижение базового случая означает всего лишь прекращение выполнения рекурсивных вызовов.

Ответим на три извечных вопроса, но в контексте реализации быстрой сортировки.

Что представляет собой базовый случай? Получение диапазона для сортировки, который пуст или содержит всего один элемент, а значит, по определению отсортирован.

Какой аргумент передается рекурсивной функции при ее вызове? Индексы левого и правого концов диапазона сортируемого списка.

Как этот аргумент приближается к базовому случаю? Диапазон уменьшается вдвое в результате каждого рекурсивного вызова и в конечном итоге становится пустым.

Следующая функция `quicksort()` в программе `quicksort.py` на языке Python ранжирует значения элементов списка в порядке возрастания:

```
def quicksort(items, left=None, right=None):
    # По умолчанию значения 'left' и 'right' охватывают весь список 'items':
    if left is None:
        left = 0 # По умолчанию 'left' соответствует индексу нулевого элемента
    if right is None:
        right = len(items) - 1 # По умолчанию 'right' соответствует индексу последнего
    print('\nquicksort() called on this range:', items[left:right + 1])
    print('.............The full list is:', items)

    if right <= left: ❶
        # Если диапазон пустой или содержит только один элемент, значит,
        # список 'items' уже отсортирован
        return  # БАЗОВЫЙ СЛУЧАЙ

    # НАЧАЛО ПРОЦЕССА РАЗБИЕНИЯ
    i = left # Исходное значение i соответствует левому концу диапазона ❷
    pivotValue = items[right] # Выбираем последнее значение в качестве опорного
```

```
    print('.....................The pivot is:', pivotValue)

    # Производим итерации вплоть до достижения опорного элемента, не включая его:
    for j in range(left, right):
        # Если значение меньше опорного, помещаем его в левую часть списка 'items':
        if items[j] <= pivotValue:
            # Меняем местами эти два значения:
            items[i], items[j] = items[j], items[i] 3
            i += 1
    # Помещаем опорный элемент в левую часть списка 'items':
    items[i], items[right] = items[right], items[i]
    # ОКОНЧАНИЕ ПРОЦЕССА РАЗБИЕНИЯ

    print('....After swapping, the range is:', items[left:right + 1])
    print('Recursively calling quicksort on:', items[left:i], 'and',
    items[i + 1:right + 1])

    # Вызываем функцию quicksort() для двух полученных разделов:
    quicksort(items, left, i - 1)   # РЕКУРСИВНЫЙ СЛУЧАЙ
    quicksort(items, i + 1, right)  # РЕКУРСИВНЫЙ СЛУЧАЙ

myList = [0, 7, 6, 3, 1, 2, 5, 4]
quicksort(myList)
print(myList)
```

Эквивалентный код на языке JavaScript содержится в файле `quicksort.html`:

```
<script type="text/javascript">
function quicksort(items, left, right) {
    // По умолчанию значения 'left' и 'right' охватывают весь список 'items':
    if (left === undefined) {
        left = 0; // По умолчанию 'left' соответствует индексу нулевого элемента
    }
    if (right === undefined) {
        right = items.length - 1; // По умолчанию 'right' соответствует
                                  // индексу последнего
    }

    document.write("<br /><pre>quicksort() called on this range: [" +
    items.slice(left, right + 1).join(", ") + "]</pre>");
    document.write("<pre>.....................The full list is: [" + items.join(", ") +
    "]</pre>");

    if (right <= left) { ❶
        // Если диапазон пуст или содержит только один элемент, значит,
        // список 'items' уже отсортирован
        return; // БАЗОВЫЙ СЛУЧАЙ
    }
```

```
    // НАЧАЛО ПРОЦЕССА РАЗБИЕНИЯ
    let i = left; ❷ // Исходное значение i соответствует левому концу диапазона
    let pivotValue = items[right]; // Выберите последнее значение в качестве опорного

    document.write("<pre>...................The pivot is: " + pivotValue.toString() +
    "</pre>");

    // Производите итерации вплоть до достижения опорного элемента, не включая его:
    for (let j = left; j < right; j++) {
        // Если значение меньше опорного, поместите его
        // в левую часть списка 'items':
        if (items[j] <= pivotValue) {
            // Поменяйте местами эти два значения:
            [items[i], items[j]] = [items[j], items[i]]; 3
            i++;
        }
    }
    // Поместите опорный элемент в левую часть списка 'items':
    [items[i], items[right]] = [items[right], items[i]];
    // ОКОНЧАНИЕ ПРОЦЕССА РАЗБИЕНИЯ

    document.write("<pre>....After swapping, the range is: [" + items.slice(left,
    right + 1).join(", ") + "]</pre>");
    document.write("<pre>Recursively calling quicksort on: [" + items.slice(left,
    i).join(", ") + "] and [" + items.slice(i + 1, right + 1).join(", ") + "]
    </pre>");

    // Вызовите функцию quicksort() для двух полученных разделов:
    quicksort(items, left, i - 1); // РЕКУРСИВНЫЙ СЛУЧАЙ
    quicksort(items, i + 1, right); // РЕКУРСИВНЫЙ СЛУЧАЙ
}

let myList = [0, 7, 6, 3, 1, 2, 5, 4];
quicksort(myList);
document.write("<pre>[" + myList.join(", ") + "]</pre>");
</script>
```

Этот код аналогичен коду алгоритма двоичного поиска. По умолчанию левая (`left`) и правая (`right`) границы диапазона соответствуют началу и концу всего массива `items`. Если алгоритм достигает базового случая, при котором значение `right` оказывается меньше или равно `left` (то есть диапазон является пустым или состоящим из одного элемента), процесс сортировки завершается ❶.

При каждом вызове функции `quicksort()` мы разделяем элементы текущего диапазона (ограниченного индексами `left` и `right`), а затем меняем их местами так, чтобы элементы, которые меньше опорного, оказывались в левой части диапазона, а те, что больше, — в правой. Например, если опорным элементом в массиве `[81, 48, 94, 87, 83, 14, 6, 42]` будет число `42`, то результат разбиения такого массива будет выглядеть так: `[14, 6, 42, 81, 48, 94, 87, 83]`. Обратите внимание, что данный массив

не является отсортированным: несмотря на его разбиение, два элемента слева от числа 42 и пять элементов справа от него остались неупорядоченными.

Большая часть кода функции `quicksort()` отвечает за процесс разбиения. Чтобы получить представление о том, как оно происходит, представьте индекс `j`, который начинается с левого конца диапазона и перемещается к правому ❷. Мы сравниваем элемент с индексом `j` с опорным, а затем движемся вправо, чтобы сравнить с ним следующий элемент. В качестве опорного элемента можно выбрать любое значение в диапазоне, но мы всегда используем для этого крайнее правое значение.

Вообразите второй индекс `i`, исходное значение которого также соответствует левому концу диапазона. Если элемент с индексом `j` меньше опорного или равен ему, то элементы с индексами `i` и `j` меняются местами ❸ и значение индекса `i` прирастает на единицу. Таким образом, `j` возрастает (перемещается вправо) всегда, то есть после каждого сравнения с опорным элементом, а `i` увеличивается только в том случае, если элемент с индексом `j` оказывается меньше опорного или равен ему.

Буквы `i` и `j` исторически сложившиеся имена переменных, которые содержат индексы массива. Однако в другой реализации `quicksort()` вместо них могут использоваться совершенно иные. Важно помнить о том, что эти две переменные содержат индексы и ведут себя так, как описано здесь.

В качестве примера давайте разберем первый этап разбиения массива `[0, 7, 6, 3, 1, 2, 5, 4]`, задав для аргумента `left` значение `0`, а для `right` — `7`, чтобы охватить весь массив. Опорным элементом (`pivot`) будет крайнее правое значение — `4`. Начальным значением индексов `i` и `j` будет `0`, оно соответствует левому концу диапазона. Индекс `j` перемещается вправо на каждом шаге, а `i` — только в том случае, если значение элемента с индексом `j` меньше опорного или равно ему. Изначально массив `items` и индексы `i` и `j` выглядят следующим образом:

```
items:   [0, 7, 6, 3, 1, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
          ^
i = 0     i
j = 0     j
```

Значение элемента с индексом `j` (оно равно `0`) меньше опорного — `4`, поэтому `i` и `j` меняются местами. В данном случае фактических изменений не происходит, поскольку индексы `i` и `j` совпадают. Мы увеличиваем значение `i`, смещая его вправо. Индекс `j` возрастает при каждом сравнении с опорным элементом. Теперь индексы `i` и `j` выглядят так:

```
items:   [0, 7, 6, 3, 1, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
             ^
i = 1        i
j = 1        j
```

На этот раз значение элемента с индексом `j` равно 7, что больше 4, поэтому мы не меняем значения местами. Помните, значение `j` увеличивается всегда, а `i` — только после выполнения перестановки, поэтому `i` обычно находится либо в том же месте, что и `j`, либо слева от него. Теперь состояние этих переменных таково:

```
items:   [0, 7, 6, 3, 1, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
             ^
i = 1        i  ^
j = 2           j
```

Элемент с индексом `j` (6) снова больше опорного (4), поэтому мы не меняем значения местами. На текущем этапе состояние переменных таково:

```
items:   [0, 7, 6, 3, 1, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
             ^
i = 1        i     ^
j = 3              j
```

Теперь элемент с индексом `j` (3) меньше 4, поэтому мы меняем местами значения элементов с индексами `i` и `j`, то есть 7 и 3, а также увеличиваем `i` на единицу, чтобы сместить индекс вправо. В результате получится следующее:

```
items:   [0, 3, 6, 7, 1, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
                ^
i = 2           i     ^
j = 4                 j
```

Так как индекс `j` (1) меньше 4, мы меняем местами значения элементов с индексами `i` (6) и `j` (1) и увеличиваем значение `i` на единицу, чтобы сместить индекс вправо:

```
items:   [0, 3, 1, 7, 6, 2, 5, 4]
indices:  0  1  2  3  4  5  6  7
                   ^
i = 3              i     ^
j = 5                    j
```

И снова элемент с индексом `j` (2) меньше опорного (4), поэтому меняем местами значения элементов с индексами `i` (7) и `j` (2), а также увеличиваем `i` на единицу для смещения индекса вправо. Теперь переменные выглядят так:

```
items:   [0, 3, 1, 2, 6, 7, 5, 4]
indices:  0  1  2  3  4  5  6  7
                      ^
i = 4                 i     ^
j = 6                       j
```

Значение элемента с индексом `j` (`6`) больше опорного (`4`), поэтому мы не меняем значения местами. На этом этапе состояние переменных таково:

```
items:   [0, 3, 1, 2, 6, 7, 5, 4]
indices:  0  1  2  3  4  5  6  7
                      ^
i = 4                 i        ^
j = 7                          j
```

Процесс разбиения завершен. Индекс `j` указывает на опорный элемент (который всегда является крайним правым значением в диапазоне), поэтому давайте в последний раз поменяем `i` и `j` местами, чтобы опорное значение не осталось в правой половине раздела. После перестановки значений `6` и `4` переменные выглядят так:

```
items:   [0, 3, 1, 2, 4, 7, 5, 6]
indices:  0  1  2  3  4  5  6  7
                      ^
i = 4                 i        ^
j = 7                          j
```

Обратите внимание, что происходит с индексом `i`: в результате перестановки он всегда получает значения меньше опорного, затем `i` перемещается вправо для получения аналогичных значений. В конечном итоге слева от него оказываются все значения меньше опорного или равные ему, а справа — больше опорного.

Данный процесс повторяется в ходе рекурсивных вызовов функции `quicksort()` для левого и правого разделов. Дальнейшее их последовательное разбиение с помощью рекурсивных вызовов `quicksort()` позволяет отсортировать весь массив.

После запуска этих программ на экране отображается процесс сортировки списка `[0, 7, 6, 3, 1, 2, 5, 4]`. Ряды точек предназначены для выравнивания вывода:

```
quicksort() called on this range: [0, 7, 6, 3, 1, 2, 5, 4]
................The full list is: [0, 7, 6, 3, 1, 2, 5, 4]
.....................The pivot is: 4
....After swapping, the range is: [0, 3, 1, 2, 4, 7, 5, 6]
Recursively calling quicksort on: [0, 3, 1, 2] and [7, 5, 6]

quicksort() called on this range: [0, 3, 1, 2]
................The full list is: [0, 3, 1, 2, 4, 7, 5, 6]
.....................The pivot is: 2
....After swapping, the range is: [0, 1, 2, 3]
Recursively calling quicksort on: [0, 1] and [3]
quicksort() called on this range: [0, 1]
................The full list is: [0, 1, 2, 3, 4, 7, 5, 6]
.....................The pivot is: 1
....After swapping, the range is: [0, 1]
Recursively calling quicksort on: [0] and []
```

```
quicksort() called on this range: [0]
...............The full list is: [0, 1, 2, 3, 4, 7, 5, 6]

quicksort() called on this range: []
...............The full list is: [0, 1, 2, 3, 4, 7, 5, 6]

quicksort() called on this range: [3]
...............The full list is: [0, 1, 2, 3, 4, 7, 5, 6]

quicksort() called on this range: [7, 5, 6]
...............The full list is: [0, 1, 2, 3, 4, 7, 5, 6]
.................The pivot is: 6
....After swapping, the range is: [5, 6, 7]
Recursively calling quicksort on: [5] and [7]
quicksort() called on this range: [5]
...............The full list is: [0, 1, 2, 3, 4, 5, 6, 7]

quicksort() called on this range: [7]
...............The full list is: [0, 1, 2, 3, 4, 5, 6, 7]

Sorted: [0, 1, 2, 3, 4, 5, 6, 7]
```

Быстрая сортировка используется довольно широко благодаря своей простоте и скорости работы. Другой часто применяемый алгоритм под названием «сортировка слиянием» также работает достаточно быстро и использует рекурсию. Рассмотрим его далее.

Сортировка слиянием: объединение небольших стопок игральных карт в более крупные и отсортированные

Математик Джон фон Нейман разработал алгоритм *сортировки слиянием* в 1945 году. Данный алгоритм использует подход «разделение — слияние»: каждый рекурсивный вызов функции `mergeSort()` делит неотсортированный список пополам до тех пор, пока длина списков не достигает нуля или единицы. Затем, по мере возврата из рекурсивных вызовов, эти маленькие списки объединяются в один большой и отсортированный. После возврата из последнего рекурсивного вызова весь список оказывается отсортированным.

Например, на этапе разделения список `[2, 9, 8, 5, 3, 4, 7, 6]` разбивается на два: `[2, 9, 8, 5]` и `[3, 4, 7, 6]`, которые передаются функции в ходе ее рекурсивных вызовов. Базовым случаем будет считаться список, состоящий из одного элемента или пустой. После возврата из рекурсивных вызовов программа объединяет списки в более крупные, получая в итоге один упорядоченный. На рис. 5.3 показан пример применения сортировки слиянием к стопке игральных карт.

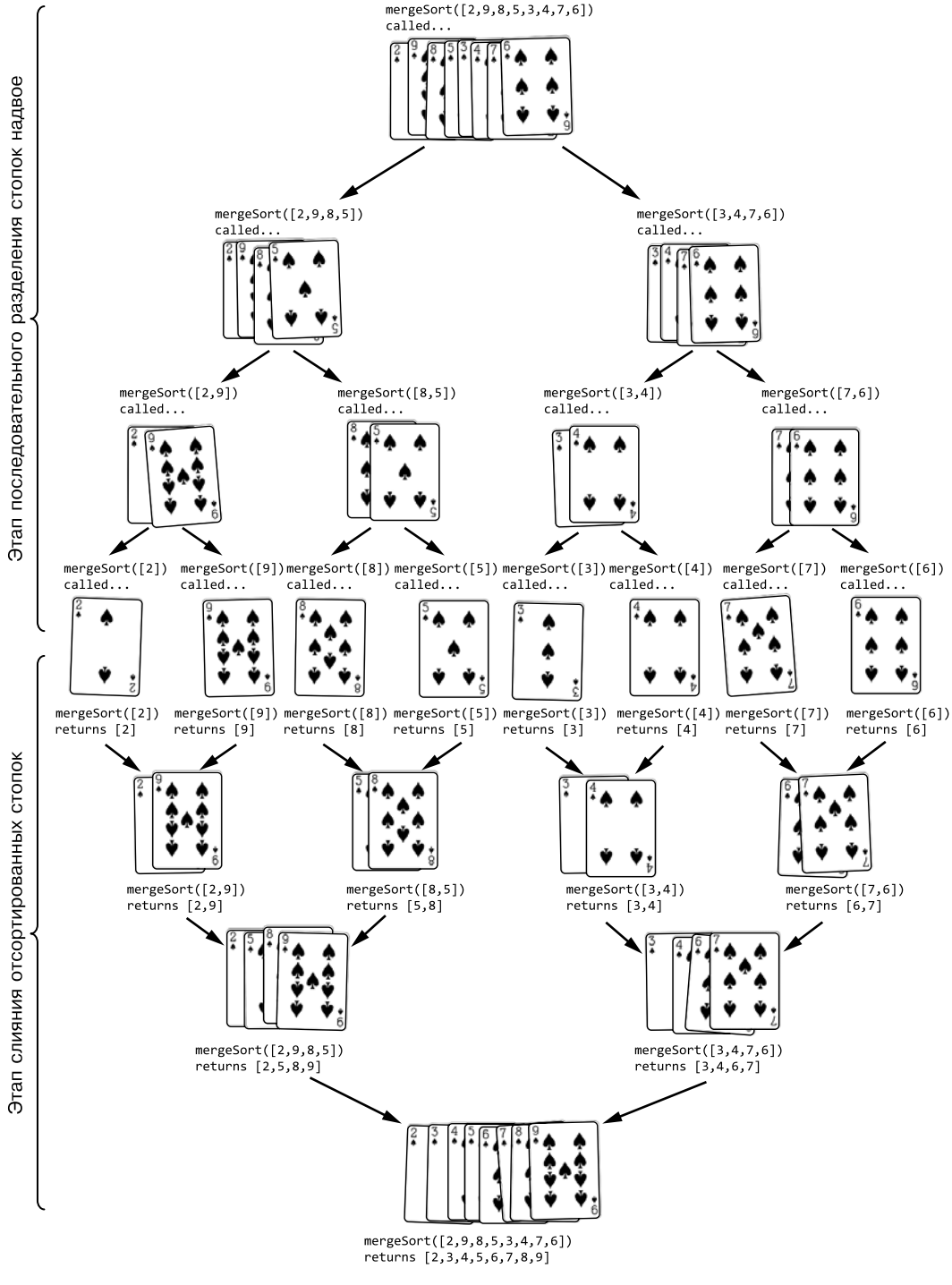


Рис. 5.3. Этапы процесса сортировки слиянием

Например, по окончании этапа разделения мы получаем восемь списков, состоящих из одного числа: `[2]`, `[9]`, `[8]`, `[5]`, `[3]`, `[4]`, `[7]`, `[6]`. Такие списки по определению являются отсортированными. Слияние двух отсортированных списков в более крупный предполагает сравнение их начальных значений и добавление меньшего из них в большой список. На рис. 5.4 показан пример слияния `[2, 9]` и `[5, 8]`. Данный процесс повторяется многократно до тех пор, пока исходный вызов `mergeSort()` не возвратит один отсортированный список.

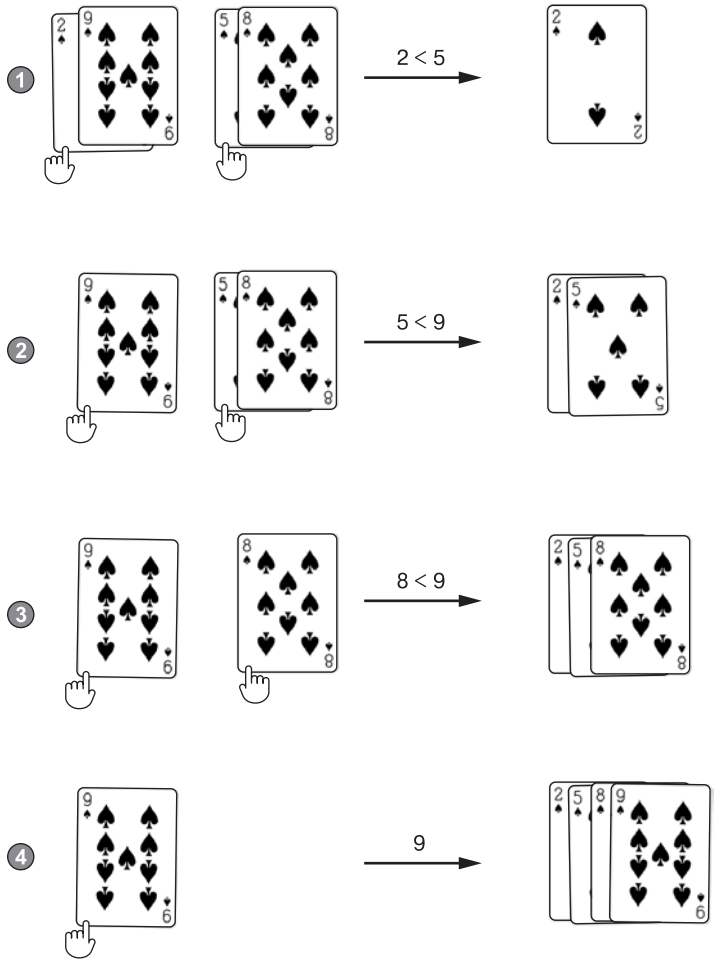


Рис. 5.4. На этапе слияния производится сравнение начальных значений небольших отсортированных списков и добавление меньшего из них в большой список. Для слияния списков, содержащих в общей сложности четыре карты, требуется всего четыре шага

Зададим наши три вопроса относительно рекурсивного алгоритма сортировки слиянием.

Что представляет собой базовый случай? Получение списка для сортировки, который является пустым или содержит всего один элемент, а значит, по определению отсортированный.

Какой аргумент передается рекурсивной функции при ее вызове? Списки, составленные из левой и правой половин исходного сортируемого списка.

Как этот аргумент приближается к базовому случаю? Списки, передаваемые рекурсивной функции в момент ее вызова, каждый раз уменьшаются вдвое, поэтому рано или поздно их длина сокращается до нуля или единицы.

Следующая функция `mergeSort()` в программе Python `mergeSort.py` сортирует значения в списке `items` в порядке возрастания:

```
import math

def mergeSort(items):
    print('.....mergeSort() called on:', items)

    # БАЗОВЫЙ СЛУЧАЙ – пустой список или список, содержащий один элемент,
    # по определению является отсортированным:
    if len(items) == 0 or len(items) == 1:
        return items ❶

    # РЕКУРСИВНЫЙ СЛУЧАЙ – передаем левую и правую половины списка функции mergeSort():
    # округляем в меньшую сторону, если число элементов в списке не делится пополам
    # без остатка:
    iMiddle = math.floor(len(items) / 2) ❷

    print('................Split into:', items[:iMiddle], 'and', items[iMiddle:])

    left = mergeSort(items[:iMiddle]) ❸
    right = mergeSort(items[iMiddle:])
    # БАЗОВЫЙ СЛУЧАЙ – возврат объединенных, отсортированных данных:
    # на этом этапе левая и правая половины должны быть отсортированы.
    # Мы можем объединить их в один отсортированный список.
    sortedResult = []
    iLeft = 0
    iRight = 0
    while (len(sortedResult) < len(items)):
        # Добавляем меньшее значение в список sortedResult
        if left[iLeft] < right[iRight]: ❹
            sortedResult.append(left[iLeft])
            iLeft += 1
        else:
            sortedResult.append(right[iRight])
            iRight += 1
```

```
        # Если один из указателей достиг конца своего списка, помещаем
        # остальную часть другого списка в sortedResult
        if iLeft == len(left):
            sortedResult.extend(right[iRight:])
            break
        elif iRight == len(right):
            sortedResult.extend(left[iLeft:])
            break
    print('The two halves merged into:', sortedResult)

    return sortedResult # Возврат отсортированной версии списка items

myList = [2, 9, 8, 5, 3, 4, 7, 6]
myList = mergeSort(myList)
print(myList)
```

Эквивалентная программа на языке JavaScript содержится в файле `mergeSort.html`:

```
<script type="text/javascript">
function mergeSort(items) {
    document.write("<pre>" + ".....mergeSort() called on: [" +
    items.join(", ") + "]</pre>");

    // БАЗОВЫЙ СЛУЧАЙ – пустой список или список, содержащий один элемент,
    // по определению является отсортированным:
    if (items.length === 0 || items.length === 1) { // БАЗОВЫЙ СЛУЧАЙ
        return items; ❶
    }

    // РЕКУРСИВНЫЙ СЛУЧАЙ – передаем левую и правую половины списка
    // функции mergeSort():
    // округляем в меньшую сторону, если число элементов в списке не делится
    // пополам без остатка:
    let iMiddle = Math.floor(items.length / 2); ❷

    document.write("<pre>..............Split into: [" + items.slice(0,
    iMiddle).join(", ") + "] and [" + items.slice(iMiddle).join(", ") + "]</pre>");

    let left = mergeSort(items.slice(0, iMiddle)); ❸
    let right = mergeSort(items.slice(iMiddle));
    // БАЗОВЫЙ СЛУЧАЙ – возврат объединенных, отсортированных данных:
    // на этом этапе левая и правая половины должны быть отсортированы.
    // Мы можем объединить их в один отсортированный список.
    let sortedResult = [];
    let iLeft = 0;
    let iRight = 0;
    while (sortedResult.length < items.length) {
        // Добавляем меньшее значение в список sortedResult
        if (left[iLeft] < right[iRight]) { ❹
            sortedResult.push(left[iLeft]);
            iLeft++;
        } else {
```

```
            sortedResult.push(right[iRight]);
            iRight++;
        }
        // Если один из указателей достиг конца своего списка, помещаем
        // остальную часть другого списка в sortedResult
        if (iLeft == left.length) {
            Array.prototype.push.apply(sortedResult, right.slice(iRight));
            break;
        } else if (iRight == right.length) {
            Array.prototype.push.apply(sortedResult, left.slice(iLeft));
            break;
        }
    }

    document.write("<pre>The two halves merged into: [" +
      sortedResult.join(", ") + "]</pre>");

    return sortedResult; // Возврат отсортированной версии списка items
}
let myList = [2, 9, 8, 5, 3, 4, 7, 6];
myList = mergeSort(myList);
document.write("<pre>[" + myList.join(", ") + "]</pre>");
</script>
```

Функция `mergeSort()` (и все ее рекурсивные вызовы) принимает неотсортированный список, а возвращает отсортированный. На первом этапе функция выполняет проверку на соответствие условиям базового случая ❶. Пустой список или список из одного элемента уже является упорядоченным, поэтому функция возвращает его как есть.

В противном случае функция определяет индекс среднего элемента списка ❷, позволяющего разделить его на левую и правую половины для их передачи рекурсивной функции ❸. Эти рекурсивные вызовы возвращают отсортированные списки, которые мы сохраняем в переменных `left` и `right`.

На следующем этапе мы объединяем два отсортированных списка в один с именем `sortedResult`. Для левого (`left`) и правого (`right`) списков используем два индекса с именами `iLeft` и `iRight`. Внутри цикла меньшее из двух значений ❹ прибавляется к списку `sortedResult`, а значение соответствующего индекса (`iLeft` или `iRight`) прирастает на единицу. Когда `iLeft` или `iRight` достигают конца своего списка, оставшиеся элементы другого добавляются в `sortedResult`.

Проследим этап слияния списков `[2, 9]` и `[5, 8]`, возвращенных рекурсивной функцией в результате ее вызова для списков `left` и `right` соответственно. Поскольку эти списки возвращены функцией `mergeSort()`, мы всегда можем предположить, что они являются отсортированными. Нам необходимо объединить их в один упорядоченный список `sortedResult`, который будет возвращен по завершении текущего вызова `mergeSort()`.

Начальное значение индексов `iLeft` и `iRight` равно `0`. Мы сравниваем значение `left[iLeft]` (`2`) и `right[iRight]` (`5`), чтобы определить наименьшее из них:

```
sortedResult = []
      left: [2, 9]    right: [5, 8]
   indices:  0  1             0  1
 iLeft = 0   ^
iRight = 0                    ^
```

Поскольку наименьшим значением в данном случае выступает `2`, добавляем его в список `sortedResult` и увеличиваем `iLeft` с `0` до `1`. Теперь состояние переменных таково:

```
sortedResult = [2]
      left: [2, 9]    right: [5, 8]
   indices:  0  1             0  1
 iLeft = 1      ^
iRight = 0                    ^
```

Снова сравнив значения `left[iLeft]` (`9`) и `right[iRight]` (`5`), мы определяем меньшее из них (`5`), добавляем его в `sortedResult` и прибавляем к `iRight` единицу. Теперь состояние переменных таково:

```
sortedResult = [2, 5]
      left: [2, 9]    right: [5, 8]
   indices:  0  1             0  1
 iLeft = 1      ^
iRight = 1                       ^
```

В очередной раз сопоставив значения `left[iLeft]` (`9`) и `right[iRight]` (`8`), определяем меньшее из них (`8`), добавляем его в `sortedResult` и увеличиваем значение `iRight` с `1` до `2`. Теперь состояние переменных таково:

```
sortedResult = [2, 5, 8]
      left: [2, 9]    right: [5, 8]
   indices:  0  1             0  1
 iLeft = 1      ^
iRight = 2                          ^
```

Поскольку `iRight` теперь равно `2`, то есть длине списка `right`, оставшиеся элементы списка `left`, находящиеся между индексом `iLeft` и концом этого списка, добавляются в `sortedResult`, так как их больше не с чем сравнивать. Таким образом, возвращаемый отсортированный список `sortedResult` выглядит так: `[2, 5, 8, 9]`. Такой процесс слияния выполняется в ходе каждого вызова `mergeSort()` для создания итогового ранжированного списка.

После запуска программ `mergeSort.py` и `mergeSort.html` на экране отображается процесс сортировки списка `[2, 9, 8, 5, 3, 4, 7, 6]`:

```
.....mergeSort() called on: [2, 9, 8, 5, 3, 4, 7, 6]
.............Split into: [2, 9, 8, 5] and [3, 4, 7, 6]
.....mergeSort() called on: [2, 9, 8, 5]
.............Split into: [2, 9] and [8, 5]
.....mergeSort() called on: [2, 9]
.............Split into: [2] and [9]
.....mergeSort() called on: [2]
.....mergeSort() called on: [9]
The two halves merged into: [2, 9]
.....mergeSort() called on: [8, 5]
.............Split into: [8] and [5]
.....mergeSort() called on: [8]
.....mergeSort() called on: [5]
The two halves merged into: [5, 8]
The two halves merged into: [2, 5, 8, 9]
.....mergeSort() called on: [3, 4, 7, 6]
.............Split into: [3, 4] and [7, 6]
.....mergeSort() called on: [3, 4]
.............Split into: [3] and [4].....mergeSort() called on: [3]
.....mergeSort() called on: [4]
The two halves merged into: [3, 4]
.....mergeSort() called on: [7, 6]
.............Split into: [7] and [6]
.....mergeSort() called on: [7]
.....mergeSort() called on: [6]
The two halves merged into: [6, 7]
The two halves merged into: [3, 4, 6, 7]
The two halves merged into: [2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7, 8, 9]
```

Как видите, функция делит список `[2, 9, 8, 5, 3, 4, 7, 6]` на `[2, 9, 8, 5]` и `[3, 4, 7, 6]` и передает их функции `mergeSort()` в ходе ее рекурсивных вызовов. Затем первый разбивается на `[2, 9]` и `[8, 5]`. Список `[2, 9]` разбивается на `[2]` и `[9]`. Они содержат только одно значение, а значит, их нельзя разделить и мы достигли базового случая: они объединяются в отсортированный список `[2, 9]`. Список `[8, 5]` делится на `[8]` и `[5]`, достигает базового случая, а затем объединяется в `[5, 8]`.

Списки `[2, 9]` и `[5, 8]` сами по себе являются ранжированными. Помните, что функция `mergeSort()` не *конкатенирует* их, результатом чего был бы неотсортированный список `[2, 9, 5, 8]`, а осуществляет их *слияние* в упорядоченный список `[2, 5, 8, 9]`. К моменту возврата из исходного вызова функции `mergeSort()` возвращаемый ею список будет полностью отсортирован.

Суммирование массива целых чисел

В главе 3 мы уже обсуждали процесс суммирования массива целых чисел с помощью метода «голова — хвост». А сейчас попробуем использовать стратегию «разделяй и властвуй». В силу сочетательного свойства сложения выражение 1 + 2 + 3 + 4

равносильно (1 + 2) + (3 + 4), поэтому допускается разделить большой массив чисел, подлежащих суммированию, на два, но поменьше.

Преимущество такого подхода в том, что при необходимости обработки больших наборов данных можно решать подзадачи параллельно, используя разные компьютеры. При этом компьютеру, который складывает вторую половину массива, нет необходимости ждать, пока другой завершит свою задачу. Большое преимущество метода «разделяй и властвуй» заключается именно в возможности одновременного применения нескольких процессоров.

Зададим наши три вопроса о рекурсивной функции суммирования.

Что представляет собой базовый случай? Пустой массив (возвращаем `0`) или массив, содержащий одно число (возвращаем это число).

Какой аргумент передается рекурсивной функции при ее вызове? Левая или правая половина массива чисел.

Как этот аргумент приближается к базовому случаю? Массив чисел уменьшается вдвое при каждом рекурсивном вызове и в конечном итоге превращается в массив нулевой или единичной длины.

Программа `sumDivConq.py` на языке Python реализует стратегию «разделяй и властвуй» для суммирования чисел с помощью функции `sumDivConq()`:

```
def sumDivConq(numbers):
    if len(numbers) == 0: # БАЗОВЫЙ СЛУЧАЙ
     ❶ return 0
    elif len(numbers) == 1: # БАЗОВЫЙ СЛУЧАЙ
     ❷ return numbers[0]
    else: # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ mid = len(numbers) // 2
        leftHalfSum = sumDivConq(numbers[0:mid])
        rightHalfSum = sumDivConq(numbers[mid:len(numbers) + 1])
     ❹ return leftHalfSum + rightHalfSum

nums = [1, 2, 3, 4, 5]
print('The sum of', nums, 'is', sumDivConq(nums))
nums = [5, 2, 4, 8]
print('The sum of', nums, 'is', sumDivConq(nums))
nums = [1, 10, 100, 1000]
print('The sum of', nums, 'is', sumDivConq(nums))
```

Эквивалентная программа на языке JavaScript содержится в файле `sumDivConq.html`:

```
<script type="text/javascript">
function sumDivConq(numbers) {
    if (numbers.length === 0) { // БАЗОВЫЙ СЛУЧАЙ
     ❶ return 0;
```

```
    } else if (numbers.length === 1) { // БАЗОВЫЙ СЛУЧАЙ
     ❷ return numbers[0];
    } else { // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ let mid = Math.floor(numbers.length / 2);
        let leftHalfSum = sumDivConq(numbers.slice(0, mid));
        let rightHalfSum = sumDivConq(numbers.slice(mid, numbers.length + 1));
     ❹ return leftHalfSum + rightHalfSum;
    }
}

let nums = [1, 2, 3, 4, 5];
document.write('The sum of ' + nums + ' is ' + sumDivConq(nums) + "<br />");
nums = [5, 2, 4, 8];
document.write('The sum of ' + nums + ' is ' + sumDivConq(nums) + "<br />");
nums = [1, 10, 100, 1000];
document.write('The sum of ' + nums + ' is ' + sumDivConq(nums) + "<br />");
</script>
```

Результат запуска этих программ выглядит так:

```
The sum of [1, 2, 3, 4, 5] is 15
The sum of [5, 2, 4, 8] is 19
The sum of [1, 10, 100, 1000] is 1111
```

Сначала функция `sumDivConq()` проверяет массив `numbers` на соответствие условиям базового случая. Если они соблюдаются, ничего суммировать не нужно: функция возвращает либо `0` ❶, либо единственное содержащееся в массиве число ❷. В рекурсивном случае вычисляется индекс среднего элемента массива ❸ и выполняются рекурсивные вызовы функции для левой и правой его половин. Сумма двух возвращаемых ими значений становится возвращаемым значением текущего вызова `sumDivConq()` ❹.

Благодаря сочетательному свойству сложения суммирование массива чисел не обязательно выполнять последовательно с помощью одного компьютера. Все операции нашей программы выполняются на одном и том же компьютере, но в случае больших массивов или более сложных вычислений их возможно распределить на несколько машин. Задачу можно разделить на похожие подзадачи и использовать для их решения рекурсивный подход.

Алгоритм умножения Карацубы

Оператор * упрощает умножение при работе с такими высокоуровневыми языками программирования, как Python и JavaScript. Однако для его непосредственного выполнения низкоуровневое аппаратное обеспечение нуждается в более примитивных (элементарных) операциях. Мы можем перемножить два целых числа,

например `5678 * 1234`, используя только операцию сложения и цикл, как показано в следующем коде на языке Python:

```
>>> x = 5678
>>> y = 1234
>>> product = 0
>>> for i in range(x):
...     product += y
...
>>> product
7006652
```

Однако подобный код малоэффективен в случае с большими целыми числами. *Умножение Карацубы* представляет собой быстрый рекурсивный алгоритм, разработанный в 1960 году Анатолием Карацубой. Данный алгоритм способен перемножать целые числа, используя сложение, вычитание и таблицу умножения, содержащую произведения всех однозначных чисел. Эта таблица умножения, показанная на рис. 5.5, называется *таблицей поиска*.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Рис. 5.5. Таблица поиска избавляет нашу программу от необходимости в повторных вычислениях, поскольку компьютер сохраняет заранее вычисленные значения в памяти для их последующего применения

Алгоритму не нужно перемножать однозначные числа, потому что он может просто найти их произведение в таблице. Сохраняя заранее вычисленные значения, мы увеличиваем объем используемой памяти, чтобы сократить время работы ЦП.

Реализуем алгоритм умножения Карацубы на таком высокоуровневом языке, как Python и JavaScript, представив, что оператора * не существует. Наша функция `karatsuba()` принимает два целочисленных аргумента, `x` и `y`, которые требуется

перемножить. Алгоритм Карацубы предусматривает пять шагов. На первых трех выполняются рекурсивные вызовы `karatsuba()` с аргументами, представляющими собой целые числа, полученные в результате разделения `x` и `y`. Базовый случай возникает, когда аргументы `x` и `y` являются однозначными числами, чье произведение можно найти в таблице поиска.

Мы также определяем еще четыре переменные: `a` и `b` содержат по половине цифр, составляющих число `x`, а `c` и `d` — по половине цифр, составляющих число `y`, как показано на рис. 5.6. Например, если `x` и `y` равны `5678` и `1234` соответственно, то `a` равно `56`, `b` — `78`, `c` — `12`, а `d` — `34`.

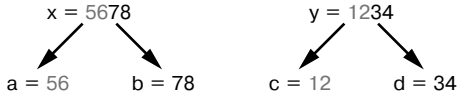


Рис. 5.6. Подлежащие перемножению целые числа x и y разделяются на половины a, b, c и d

Вот пять шагов алгоритма Карацубы.

1. Перемножаем значения `a` и `c` с помощью таблицы поиска или рекурсивного вызова функции `karatsuba()`.
2. Перемножаем значения `b` и `d` тем же способом.
3. Точно так же перемножаем значения `a + c` и `b + d`.
4. Вычисляем разность: результат шага 3 – результат шага 2 – результат шага 1.
5. Дополните нулями строки результатов шагов 1 и 4, затем добавьте эти результаты к результату шага 2.

Результатом шага 5 будет произведение `x` и `y`. Подробнее о том, как дополнить результаты шагов 1 и 4 нулями, мы поговорим чуть позже в этом разделе.

Ответим на три вопроса о рекурсивной функции `karatsuba()`.

Что представляет собой базовый случай? Перемножение однозначных чисел, которое можно выполнить с помощью предварительно вычисленной таблицы поиска.

Какой аргумент передается рекурсивной функции при ее вызове? Значения `a`, `b`, `c` и `d`, полученные из аргументов `x` и `y`.

Как этот аргумент приближается к базовому случаю? Поскольку каждое из значений `a`, `b`, `c` и `d` представляет собой половину цифр, составляющих числа `x` и `y`, и сами используются в качестве аргументов для следующего рекурсивного вызова, аргументы рекурсивного вызова постоянно приближаются к однозначным числам, соответствующим базовому случаю.

Наша реализация алгоритма умножения Карацубы на языке Python находится в файле `karatsubaMultiplication.py`:

```
import math

# Создаем таблицу поиска, содержащую произведения всех однозначных чисел:
MULT_TABLE = {} ❶
for i in range(10):
    for j in range(10):
        MULT_TABLE[(i, j)] = i * j

def padZeros(numberString, numZeros, insertSide):
    """Возвращаем строку, дополненную нулями слева или справа."""
    if insertSide == 'left':
        return '0' * numZeros + numberString
    elif insertSide == 'right':
        return numberString + '0' * numZeros

def karatsuba(x, y):
    """Перемножаем два целых числа с помощью алгоритма Карацубы.
    Обратите внимание, что оператор * не используется."""
    assert isinstance(x, int), 'x must be an integer'
    assert isinstance(y, int), 'y must be an integer'
    x = str(x)
    y = str(y)

    # При получении однозначных чисел находим их произведения в таблице поиска:
    if len(x) == 1 and len(y) == 1: # БАЗОВЫЙ СЛУЧАЙ
        print('Lookup', x, '*', y, '=', MULT_TABLE[(int(x), int(y))])
        return MULT_TABLE[(int(x), int(y))]

    # РЕКУРСИВНЫЙ СЛУЧАЙ
    print('Multiplying', x, '*', y)

    # Дополняем строки x и y начальными нулями, чтобы они были одинаковой длины:
    if len(x) < len(y): ❷
        # Если x короче y, дополняем нулями x:
        x = padZeros(x, len(y) - len(x), 'left')
    elif len(y) < len(x):
        # Если y короче x, дополняем нулями y:
        y = padZeros(y, len(x) - len(y), 'left')
    # На данном этапе x и y имеют одинаковую длину

    halfOfDigits = math.floor(len(x) / 2) ❸

    # Делим x на половины a и b, а y — на половины c и d:
    a = int(x[:halfOfDigits])
    b = int(x[halfOfDigits:])
    c = int(y[:halfOfDigits])
    d = int(y[halfOfDigits:])
    # Выполняем рекурсивные вызовы, передавая эти половины в качестве аргументов:
    step1Result = karatsuba(a, c) ❹ # Шаг 1: перемножаем значения a и c
```

```
    step2Result = karatsuba(b, d) # Шаг 2: перемножаем значения b и d
    step3Result = karatsuba(a + b, c + d) # Шаг 3: перемножаем суммы a + b и c + d

    # Шаг 4: вычисляем разность: результат шага 3 – результат
    # шага 2 – результат шага 1:
    step4Result = step3Result - step2Result - step1Result ❺

    # Шаг 5: дополняем эти числа нулями и складываем их,
    # чтобы получить возвращаемое значение:
    step1Padding = (len(x) - halfOfDigits) + (len(x) - halfOfDigits)
    step1PaddedNum = int(padZeros(str(step1Result), step1Padding, 'right'))

    step4Padding = (len(x) - halfOfDigits)
    step4PaddedNum = int(padZeros(str(step4Result), step4Padding, 'right'))

    print('Solved', x, 'x', y, '=', step1PaddedNum + step2Result +
      step4PaddedNum)

    return step1PaddedNum + step2Result + step4PaddedNum ❻

# Пример: 1357 x 2468 = 3349076
print('1357 * 2468 =', karatsuba(1357, 2468))
```

Эквивалентная программа на языке JavaScript содержится в файле `karatsubaMultiplication.html`:

```
<script type="text/javascript">

// Создаем таблицу поиска, содержащую произведения всех однозначных чисел:
let MULT_TABLE = {}; ❶
for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
        MULT_TABLE[[i, j]] = i * j;
    }
}

function padZeros(numberString, numZeros, insertSide) {
    // Возвращаем строку, дополненную нулями слева или справа
    if (insertSide === "left") {
        return "0".repeat(numZeros) + numberString;
    } else if (insertSide === "right") {
        return numberString + "0".repeat(numZeros);
    }
}
function karatsuba(x, y) {
    // Перемножаем два целых числа с помощью алгоритма Карацубы.
    // Обратите внимание, что оператор * не используется.
    console.assert(Number.isInteger(x), "x must be an integer");
    console.assert(Number.isInteger(y), "y must be an integer");
    x = x.toString();
    y = y.toString();
```

```
// При получении однозначных чисел находим их произведения в таблице поиска:
if ((x.length === 1) && (y.length === 1)) { // БАЗОВЫЙ СЛУЧАЙ
    document.write("Lookup " + x.toString() + " * " + y.toString() + " = " +
    MULT_TABLE[[parseInt(x), parseInt(y)]] + "<br />");
    return MULT_TABLE[[parseInt(x), parseInt(y)]];
}

// РЕКУРСИВНЫЙ СЛУЧАЙ
document.write("Multiplying " + x.toString() + " * " + y.toString() + "<br />");

// Дополняем строки x и y начальными нулями, чтобы они были одинаковой длины:
if (x.length < y.length) { ❷
    // Если x короче y, дополняем нулями x:
    x = padZeros(x, y.length - x.length, "left");
} else if (y.length < x.length) {
    // Если y короче x, дополняем нулями y:
    y = padZeros(y, x.length - y.length, "left");
}
// На данном этапе x и y имеют одинаковую длину

let halfOfDigits = Math.floor(x.length / 2); ❸

// Делим x на половины a и b, а y — на половины c и d:
let a = parseInt(x.substring(0, halfOfDigits));
let b = parseInt(x.substring(halfOfDigits));
let c = parseInt(y.substring(0, halfOfDigits));
let d = parseInt(y.substring(halfOfDigits));

// Выполняем рекурсивные вызовы, передавая эти половины в качестве аргументов:
let step1Result = karatsuba(a, c); ❹ // Шаг 1: перемножаем значения a и c
let step2Result = karatsuba(b, d); // Шаг 2: перемножаем значения b и d
let step3Result = karatsuba(a + b, c + d); // Шаг 3: перемножаем
                                           // суммы a + b и c + d
// Шаг 4: вычисляем разность: результат шага 3 – результат
// шага 2 – результат шага 1:
let step4Result = step3Result - step2Result - step1Result; ❺

// Шаг 5: дополняем эти числа нулями и складываем их,
// чтобы получить возвращаемое значение:
let step1Padding = (x.length - halfOfDigits) + (x.length –
  halfOfDigits);
let step1PaddedNum = parseInt(padZeros(step1Result.toString(),
  step1Padding, "right"));

let step4Padding = (x.length - halfOfDigits);
let step4PaddedNum = parseInt(padZeros((step4Result).toString(),
  step4Padding, "right"));

document.write("Solved " + x + " x " + y + " = " +
(step1PaddedNum + step2Result + step4PaddedNum).toString() + "<br />");
```

```
    return step1PaddedNum + step2Result + step4PaddedNum; ❻
}

// Пример: 1357 x 2468 = 3349076
document.write("1357 * 2468 = " + karatsuba(1357, 2468).toString() + "<br />");
</script>
```

Результат запуска этих программ выглядит так:

```
Multiplying 1357 * 2468
Multiplying 13 * 24
Lookup 1 * 2 = 2
Lookup 3 * 4 = 12
Lookup 4 * 6 = 24
Solved 13 * 24 = 312
Multiplying 57 * 68
Lookup 5 * 6 = 30
Lookup 7 * 8 = 56
Multiplying 12 * 14
Lookup 1 * 1 = 1
Lookup 2 * 4 = 8
Lookup 3 * 5 = 15
Solved 12 * 14 = 168
Solved 57 * 68 = 3876
Multiplying 70 * 92
Lookup 7 * 9 = 63
Lookup 0 * 2 = 0
Multiplying 7 * 11
Lookup 0 * 1 = 0
Lookup 7 * 1 = 7
Lookup 7 * 2 = 14
Solved 07 * 11 = 77
Solved 70 * 92 = 6440
Solved 1357 * 2468 = 3349076
1357 * 2468 = 3349076
```

Первая часть программы выполняется до вызова функции `karatsuba()`. Сначала программа должна создать таблицу поиска в переменной `MULT_TABLE` ❶. Обычно такие таблицы жестко запрограммированы непосредственно в исходном коде, начиная с `MULT_TABLE[[0, 0]] = 0` и заканчивая `MULT_TABLE[[9, 9]] = 81`. Но в целях уменьшения объема вводимого кода мы будем использовать вложенные циклы `for` для вычисления каждого произведения. Обращение к `MULT_TABLE[[m, n]]` позволяет получить результат произведения целых чисел `m` и `n`.

Наша функция `karatsuba()` также полагается на вспомогательную функцию `padZeros()`, которая на пятом шаге алгоритма Карацубы дополняет строку цифр нулями (слева или справа от нее). Например, вызов `padZeros("42", 3, "left")` возвращает строку `00042`, а вызов `padZeros("99", 1, "right")` — строку `990`.

Сама функция `karatsuba()` сначала проверяет аргументы на соответствие условиям базового случая, при котором `x` и `y` являются однозначными числами. Их можно перемножить с помощью таблицы поиска и сразу же возвратить полученное произведение. Если условия базового случая не соблюдаются, имеет место рекурсивный случай.

Нам нужно преобразовать целые числа `x` и `y` в строки и сделать так, чтобы они содержали одинаковое количество цифр. Если одно из этих чисел короче другого, то оно дополняется нулями слева. Например, если `x` равно `13`, а `y` — `2468`, наша функция вызывает `padZeros()`, чтобы заменить значение `x` на `0013`. Это необходимо, поскольку в дальнейшем мы создадим переменные `a`, `b`, `c` и `d`, каждая из которых будет содержать половину цифр, составляющих числа `x` и `y` ❷. Чтобы алгоритм Карацубы сработал, переменные `a`, `c`, `b` и `d` должны состоять из одинакового количества цифр.

Обратите внимание, что для разбиения числа `x` на две группы цифр, из которых оно состоит, мы используем деление и округление в меньшую сторону ❸. Данные математические операции столь же сложны, как и умножение, поэтому могут быть недоступны для низкоуровневого оборудования, для которого мы создаем алгоритм Карацубы. В настоящей реализации для нахождения этих значений мы могли бы использовать еще одну таблицу поиска: `HALF_TABLE = [0, 0, 1, 1, 2, 2, 3, 3...]` и т. д. Обращение к `HALF_TABLE[n]` позволило бы получить половину числа `n`, округленную в меньшую сторону. Достаточно массива из 100 элементов для вычисления всех чисел, кроме поистине астрономических, и избавления программы от необходимости выполнять операции деления и округления. Однако наши программы предназначены для демонстрации принципа работы, поэтому мы просто используем оператор `/` и встроенные функции округления.

После создания этих переменных мы можем приступить к выполнению рекурсивных вызовов нашей функции ❹. Первые три шага предполагают выполнение рекурсивных вызовов с использованием аргументов `a` и `b`, `c` и `d` и, наконец, `a + b` и `c + d`. Четвертый шаг сводится к подсчету разности результатов первых трех шагов ❺. Пятый предполагает дополнение результатов первого и четвертого шагов нулями с правой стороны и их добавление к результатам второго шага ❻.

Алгебра, лежащая в основе алгоритма Карацубы

Чтобы работа этого алгоритма не казалась какой-то магией, давайте рассмотрим алгебру, лежащую в ее основе. Допустим, нам нужно умножить целое число x, равное 1357, на целое число y, равное 2468. Создадим переменную n для количества цифр, составляющих x или y. Поскольку a равно 13, а b — 57, мы можем вычислить исходное значение x с помощью выражения $10^{n/2} \times a + b$: $10^2 \times 13 + 57 = 1300 + 57 = 1357$. Точно так же значение y можно вычислить с помощью выражения $10^{n/2} \times c + d$.

Это означает, что произведение $x \times y = (10^{n/2} \times a + b) \times (10^{n/2} \times c + d)$. Уравнение можно переписать в виде $x \times y = 10^n \times ac + 10^{n/2} \times (ad + bc) + bd$. Подставив числа из нашего примера, получим $1357 \times 2468 = 10\,000 \times (13 \times 24) + 100 \times (13 \times 68 + 57 \times \times 24) + (57 \times 68)$. В результате вычислений в обеих частях этого уравнения получается значение 3 349 076.

Мы разбили операцию умножения *xy* на операции умножения *ac*, *ad*, *bc* и *bd*, что позволило нам сформировать основу для нашей рекурсивной функции: мы определили умножение *x* и *y* через перемножение меньших чисел (как вы помните, *a*, *b*, *c* и *d* соответствуют половинам цифр, составляющих числа *x* или *y*), которые приближаются к базовому случаю, предполагающему перемножение однозначных чисел, которое можно выполнить с помощью таблицы поиска.

Итак, нам необходимо рекурсивно вычислить произведение *ac* (первый шаг алгоритма Карацубы) и *bd* (второй шаг). На третьем шаге нужно посчитать выражение $(a + b)(c + d)$, которое можно представить в виде $ac + ad + bc + bd$. У нас уже есть результаты вычисления *ac* и *bd*, полученные на первых двух шагах, поэтому за их вычетом остается $ad + bc$. То есть для определения суммы $ad + bc$ вместо двух операций нам достаточно одной операции умножения (и одного рекурсивного вызова) для вычисления $(a + b)(c + d)$. А сумма $ad + bc$ требуется для части $10^{n/2} \times (ad + bc)$ нашего исходного уравнения.

Умножение на 10^n и $10^{n/2}$ можно выполнить путем добавления нулей: например, $10\,000 \times 123$ равно 1 230 000. Таким образом, эти операции умножения не требуют выполнения рекурсивных вызовов. В конце концов, перемножение *x* и *y* можно осуществить с помощью трех рекурсивных вызовов: `karatsuba(a, c)`, `karatsuba(b, d)` и `karatsuba((a + b), (c + d))`.

Как видите, алгебра, лежащая в основе алгоритма, вполне объяснима. Однако лично мне сложно понять, как Анатолий Карацуба сумел разработать его менее чем за неделю, будучи 23-летним студентом.

Резюме

В основе рекурсии лежит разделение задач на более мелкие самоподобные подзадачи, из-за чего алгоритмы типа «разделяй и властвуй» особенно хороши для решения рекурсивных задач. Мы создали версию программы для суммирования чисел в массиве из главы 3 на основе такого алгоритма. Одним из ее преимуществ является то, что после дробления задачи ее производные можно решать параллельно с помощью нескольких компьютеров.

Алгоритм двоичного поиска ищет значение в отсортированном массиве, многократно уменьшая диапазон поиска вдвое. Алгоритм линейного поиска последовательно обрабатывает весь массив с самого начала, а алгоритм двоичного поиска находит

искомый элемент, пользуясь тем, что значения в массиве отсортированы. Разница в их производительности достаточно велика для того, чтобы потратить некоторое время на ранжирование массива перед применением алгоритма двоичного поиска.

Мы также рассмотрели два популярных алгоритма сортировки: быструю сортировку и сортировку слиянием. Первый разбивает массив на две части, используя опорный элемент, а затем рекурсивно повторяет этот процесс вплоть до получения массивов единичной длины. Получившиеся массивы и находящиеся в них элементы оказываются упорядоченными. При сортировке слиянием применяется противоположный подход. Сначала алгоритм разбивает исходный массив на массивы поменьше, которые затем объединяет в один большой отсортированный массив.

Наконец, был изучен рекурсивный алгоритм умножения Карацубы, позволяющий перемножать целые числа без использования оператора *. Это бывает актуально при низкоуровневом аппаратном программировании, которое не предусматривает встроенной инструкции умножения. Алгоритм Карацубы разбивает операцию перемножения двух целых чисел на три операции перемножения меньших чисел. Для перемножения однозначных чисел в базовом случае алгоритм сохраняет произведения от 0×0 до 9×9 в таблице поиска.

Алгоритмы, описанные в этой главе, часто изучаются студентами в рамках курсов по структурам данных и алгоритмам. А в следующей мы продолжим работу с алгоритмами, лежащими в основе вычислений, в частности связанных с перестановками и сочетаниями.

Дополнительные источники информации

На YouTube-канале Computerphile вы можете найти видеоролики, посвященные быстрой сортировке (https://youtu.be/XE4VP_8Y0BU) и сортировке слиянием (https://youtu.be/kgBjXUE_Nwc). Для получения более полной информации рекомендую пройти бесплатный онлайн-курс *Algorithmic Toolbox*, который охватывает многие вопросы по структурам данных и алгоритмам, включая двоичный поиск, быструю сортировку и сортировку слиянием. Записаться на курс можно на сайте Coursera по ссылке https://www.coursera.org/learn/algorithmic-toolbox.

Алгоритмы сортировки часто сравниваются друг с другом в терминах нотации «О большое», о которой рассказывается в главе 13 моей книги «Python. Чистый код для продолжающих», доступной по адресу https://inventwithpython.com/beyond. Python-разработчик Нед Бэтчелдер (Ned Batchelder) подробно рассказал об этой концепции и о том, как замедляется код по мере увеличения объема данных, в своем выступлении на PyCon в 2018 году (*Big-O: How Code Slows as Data Grows*, https://youtu.be/duvZ-2UK0fc).

Алгоритмы типа «разделяй и властвуй» полезны тем, что они могут выполняться параллельно на нескольких компьютерах. Этой теме было посвящено выступление

Гая Стила-младшего (Guy Steele Jr.) под названием *Four Solutions to a Trivial Problem* на конференции Google TechTalk (https://youtu.be/ftcIcn8AmSY).

Видеолекция об алгоритме умножения Карацубы, подготовленная профессором Тимом Рафгарденом (Tim Roughgarden) для Стэнфордского университета, доступна по адресу https://youtu.be/JCbZayFr9RE.

Чтобы лучше разобраться в принципе работы алгоритма быстрой сортировки и сортировки слиянием, возьмите колоду игральных карт или просто напишите числа на карточках и потренируйтесь сортировать их вручную в соответствии с правилами этих двух алгоритмов. Данный подход поможет вам запомнить, что быстрая сортировка осуществляется с использованием опорного элемента, а сортировка слиянием выполняется по принципу «разделение — объединение».

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. В чем заключается преимущество суммирования по принципу «разделяй и властвуй» по сравнению с алгоритмом суммирования методом «голова — хвост»?
2. Если двоичный поиск среди 50 книг предполагает шесть шагов, сколько шагов потребуется, чтобы найти нужный том среди 100 книг?
3. Можно ли использовать алгоритм двоичного поиска для нахождения нужного элемента в неотсортированном массиве?
4. Разбиение и сортировка — это одно и то же?
5. Что происходит на этапе разбиения при использовании быстрой сортировки?
6. Что такое опорный элемент в контексте быстрой сортировки?
7. Что является базовым случаем при быстрой сортировке?
8. Сколько рекурсивных вызовов предусматривает функция `quicksort()`?
9. Почему при использовании опорного значения `4` массив `[0, 3, 1, 2, 5, 4, 7, 6]` не разбивается должным образом?
10. Что является базовым случаем при сортировке слиянием?
11. Сколько рекурсивных вызовов предусматривает функция `mergeSort()`?
12. Какой массив получится в результате применения алгоритма сортировки слиянием к массивам `[12, 37, 38, 41, 99]` и `[2, 4, 14, 42]`?
13. Что такое таблица поиска?
14. Что хранят переменные a, b, c и d в алгоритме Карацубы, перемножающем целые числа x и y?

15. Ответьте на следующие три вопроса относительно каждого из рекурсивных алгоритмов, представленных в этой главе.

    А. Что представляет собой базовый случай?

    Б. Какой аргумент передается рекурсивной функции при ее вызове?

    В. Как этот аргумент приближается к базовому случаю?

    Затем воссоздайте рекурсивные алгоритмы, описанные в пройденной главе, не заглядывая в исходный код.

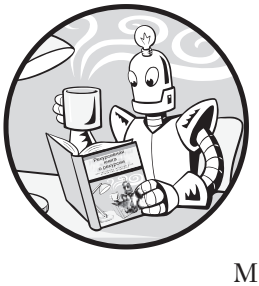
Практика

Решите каждую из следующих задач.

1. Напишите функцию `karatsuba()`, которая предусматривает таблицу поиска, содержащую результаты произведения от 0×0 до 999×999. Приблизительно оцените время, необходимое для 10 000-кратного вычисления `karatsuba(12345678, 87654321)` в цикле с помощью этой более крупной таблицы поиска. Если функция работает слишком быстро, чтобы эту разницу можно было измерить, увеличьте количество итераций до 100 000 или более (совет: для проведения подобного теста вам следует удалить или закомментировать вызовы `print()` и `document.write()` внутри функции `karatsuba()`).
2. Напишите функцию, которая 10 000 раз выполняет линейный поиск в большом массиве целых чисел. Приблизительно оцените необходимое для этого время (если программа выполняется слишком быстро, увеличьте количество итераций до 100 000 или 1 000 000). Сравните полученный результат с тем, сколько времени требуется второй функции, чтобы отсортировать массив перед тем, как выполнить двоичный поиск такое же количество раз.

6

Перестановки и сочетания



Рекурсия особенно хорошо подходит для решения задач, связанных с перестановками и сочетаниями, распространенными в *теории множеств* — разделе математической логики, посвященном вопросам выбора, упорядочения и манипулирования объектами.

Мы можем с легкостью удерживать в кратковременной памяти небольшие множества, состоящие из трех или четырех объектов, вместе со всеми возможными вариантами их упорядочения (то есть *перестановками*) или *сочетания*. Упорядочение и комбинирование объектов в более крупном множестве требует тех же усилий, но быстро становится невыполнимой задачей для человеческого мозга. Справиться с комбинаторным взрывом, обусловленным ростом числа элементов множества, помогают компьютеры.

По сути, вычисление перестановок и сочетаний больших групп объектов предусматривает то же самое для меньших групп, что делает эти расчеты пригодными для применения рекурсии. В данной главе мы рассмотрим рекурсивные алгоритмы генерации всех возможных перестановок и сочетаний символов строки. Затем на их основе разработаем алгоритм, генерирующий все потенциальные комбинации сбалансированных скобок (упорядоченных наборов открывающих и соответствующих им закрывающих скобок). И наконец, мы вычислим булеан множества, то есть множество всех его возможных подмножеств.

Многие рекурсивные функции, описанные в этой главе, предусматривают аргумент с именем `indent`. Он предназначен не для самих рекурсивных алгоритмов, а для оформления выходных данных отладки, чтобы вы могли увидеть, какому уровню рекурсии соответствуют те или иные выходные данные. Отступ увеличивается на один пробел в результате каждого рекурсивного вызова и отображается на экране в виде точек, которые легко подсчитать.

Терминология теории множеств

В текущей главе теория множеств рассматривается не так полно, как в учебнике по математике или информатике. Однако охват материала довольно большой, и, чтобы облегчить его понимание, мы познакомимся с основными терминами данной дисциплины. *Множество* — это набор уникальных объектов, называемых *элементами* или *членами*. Например, буквы *A*, *B* и *C* образуют множество из трех элементов. В математике (и синтаксисе кода Python) множества заключаются в фигурные скобки, а составляющие их элементы разделяются запятыми: {*A*, *B*, *C*}.

Порядок не имеет значения; множество {*A*, *B*, *C*} эквивалентно {*C*, *B*, *A*}. Множества состоят из различающихся между собой уникальных элементов, что означает отсутствие их дубликатов: {*A*, *C*, *A*, *B*} содержит две буквы *A* и, следовательно, не является множеством.

Множество является *подмножеством* другого, если оно состоит только из элементов этого множества. Например, {*A*, *C*} и {*B*, *C*} — подмножества {*A*, *B*, *C*}, чего нельзя сказать об {*A*, *C*, *D*}. В то время как {*A*, *B*, *C*} — *надмножество* для {*A*, *C*} и {*B*, *C*}, поскольку оно содержит в себе все составляющие их элементы. *Пустым множеством* { } называется набор, не содержащий ни одного элемента. Такая математическая модель считается подмножеством всех возможных множеств.

Подмножество также может включать в себя все элементы некоторого множества. Например, {*A*, *B*, *C*} — подмножество {*A*, *B*, *C*}. Однако *правильное*, или *строгое*, *подмножество* не содержит всех элементов множества. Никакое множество не является правильным подмножеством для самого себя, поэтому {*A*, *B*, *C*} — подмножество, но не правильное подмножество {*A*, *B*, *C*}. Все остальные подмножества правильные. На рис. 6.1 показано графическое представление множества {*A*, *B*, *C*} и некоторых его подмножеств.

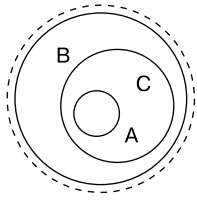


Рис. 6.1. Множество {A, B, C}, обведенное пунктирной линией, и его подмножества {A, B, C}, {A, C} и { }, обведенные сплошными линиями. Круги обозначают множества, а буквы — их элементы

Перестановка — это определенный способ упорядочения всех элементов множества. Например, множество {*A*, *B*, *C*} предусматривает шесть возможных перестановок: *ABC*, *ACB*, *BAC*, *BCA*, *CAB* и *CBA*. Мы называем их *перестановками без повторений*

или *перестановками без замены*, потому что каждый элемент в них встречается только один раз.

Сочетание — это набор элементов, выбранных из множества. Если более формально, то *k-элементное сочетание* — это подмножество, состоящее из *k* элементов множества. В отличие от перестановок, сочетания не предусматривают никакого упорядочения. Например, 2-элементными сочетаниями множества {*A*, *B*, *C*} являются {*A*, *B*}, {*A*, *C*} и {*B*, *C*}, а 3-элементным сочетанием множества {*A*, *B*, *C*} является {*A*, *B*, *C*}.

Число сочетаний из n по k (*n choose k*) — это количество возможных комбинаций (без повторений) *k* элементов, выбранных из множества, состоящего из *n* элементов (некоторые математики используют термин «*число сочетаний из n по r*»). Данное понятие относится не к самим элементам, а только к их количеству. Скажем, число сочетаний из 4 по 2 равно 6, потому что существует шесть способов выбрать два элемента из множества, состоящего из четырех элементов, например {*A*, *B*, *C*, *D*}: {*A*, *B*}, {*A*, *C*}, {*A*, *D*}, {*B*, *C*}, {*B*, *D*} и {*C*, *D*}. Между тем число сочетаний из 3 по 3 равно 1, потому что множество из трех элементов, вроде {*A*, *B*, *C*}, предусматривает только одно 3-элементное сочетание — {*A*, *B*, *C*}. Формула для расчета числа сочетаний из *n* по *k* выглядит так: $(n!) / (k! \times (n - k)!)$. Как вы помните, $n!$ — это обозначение факториала.

Числом сочетаний из n по k с повторениями (n multichoose k) называется количество возможных комбинаций *с повторениями k* элементов, выбранных из множества, состоящего из *n* элементов. Подобные сочетания не содержат дублирующихся элементов и не предусматривают повторений. Когда мы используем *k*-элементные сочетания с повторяющимися элементами, то специально называем их *k-элементными сочетаниями с повторениями*.

Вне зависимости от наличия повторений перестановку можно рассматривать как определенный вариант упорядочения всех элементов множества, а сочетание — как неупорядоченную выборку элементов множества. Перестановки подразумевают порядок и задействуют все содержимое набора, в то время как сочетания не подразумевают никакого упорядочения и задействуют лишь некоторое количество его элементов. Чтобы лучше понять смысл этих терминов, обратитесь к табл. 6.1, в которой показаны различия между перестановками и сочетаниями элементов множества {*A*, *B*, *C*}, с повторениями и без них.

Таблица 6.1. Все возможные перестановки и сочетания элементов множества {A, B, C}, с повторениями и без них

	Перестановки	**Сочетания**
Без повторений	ABC, ACB, BAC, BCA, CAB	(Нет), A, B, C, AB, AC, BC, ABC
С повторениями	AAA, AAB, AAC, ABA, ABB, ABC, ACA, ACB, ACC, BAA, BAB, BAC, BBA, BBB, BBC, BCA, BCB, BCC, CAA, CAB, CAC, CBA, CBB, CBC, CCA, CCB, CCC	(Нет), A, B, C, AA, AB, AC, BB, BC, CC, AAA, AAB, AAC, ABB, ABC, ACC, BBB, BBC, BCC, CCC

Удивительно, насколько быстро растет количество перестановок и сочетаний по мере увеличения числа элементов множества. Этот *комбинаторный взрыв* описывается формулами, приведенными в табл. 6.2. Например, множество из 10 элементов предусматривает 10!, или 3 628 800, возможных перестановок, а множество из 20 элементов — 20!, или 2 432 902 008 176 640 000, перестановок.

Таблица 6.2. Вычисление количества возможных перестановок и сочетаний множества из n элементов, с повторениями и без них

	Перестановки	**Сочетания**
Без повторений	n!	2^n
С повторениями	n^n	Число сочетаний из 2n по n, или $(2n)! / (n!)^2$

Обратите внимание, что перестановки без повторений всегда имеют тот же размер, что и множество. Например, перестановки {*A*, *B*, *C*} всегда состоят из трех букв: *ABC*, *ACB*, *BAC* и т. д. Однако перестановки с повторениями могут быть любой длины. В табл. 6.1 показаны трехбуквенные перестановки {*A*, *B*, *C*} в диапазоне от *AAA* до *CCC*, однако также могут существовать, например, пятибуквенные перестановки с повторениями в диапазоне от *AAAAA* до *CCCCC*. Для набора из *n* элементов количество перестановок с повторениями длиной *k* элементов составляет *nk*. В табл. 6.2 указано количество перестановок с повторениями длиной *n* элементов и составляет *nn*.

Порядок имеет значение для перестановок, но не для сочетаний. Несмотря на то что AAB, ABA и BAA считаются одним и тем же сочетанием с повторениями, эти комбинации представляют собой три разные перестановки с повторениями.

Поиск всех перестановок без повторения: схема рассадки гостей на свадьбе

Представьте, что вам нужно составить схему рассадки гостей на свадебном банкете, учитывая деликатные социальные соображения. Некоторые из гостей ненавидят друг друга, а другие хотят сидеть рядом с влиятельным гостем. Столы не круглые, а прямоугольные, поэтому сиденья выстроены в один длинный ряд. При планировании вам было бы полезно учесть все возможные варианты рассадки гостей, то есть все перестановки без повторений. Дублирование отсутствует, потому что каждый гость учитывается в схеме рассадки только один раз.

Рассмотрим простое множество гостей, состоящее из трех человек — Алисы, Боба и Кэрол, или {*A*, *B*, *C*}. На рис. 6.2 показаны все шесть возможных вариантов их рассадки.

Один из способов вычислить количество перестановок без повторений — использовать рекурсивный метод «голова — хвост». Мы выбираем один элемент из

множества в качестве головного. Затем берем все комбинации оставшихся элементов (которые составляют хвост) и помещаем головной элемент в каждую из них.

В нашем примере с множеством гостей *ABC* голова — Алиса *(A)*, а хвост — Боб и Кэрол *(BC)*. Перестановками {*B, C*} являются *BC* и *CB* (процесс получения перестановок *BC* и *CB* объясняется в следующем абзаце, поэтому пока не думайте об этом). Мы помещаем элемент *A* во все возможные положения перестановки *BC*, то есть сажаем Алису перед Бобом *(ABC)*, между Бобом и Кэрол *(BAC)* и после Кэрол *(BCA)*. В результате получаем перестановки *ABC*, *BAC* и *BCA*. Мы также располагаем элемент *A* в перестановке *CB*, получая *ACB*, *CAB* и *CBA*. Итак, нам удалось получить все шесть вариантов рассадки Алисы, Боба и Кэрол за столом. Теперь можно выбрать тот вариант, который обещает наименьшее количество скандалов (или наибольшее, если вы хотите сделать свадебный банкет незабываемым).

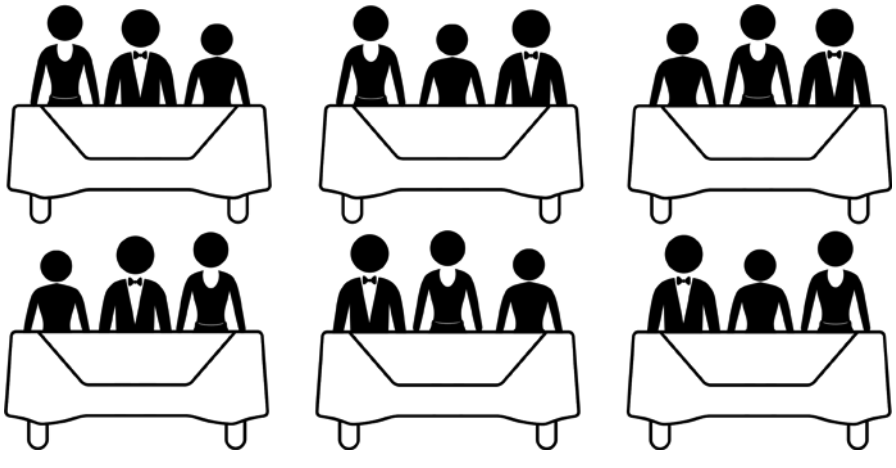


Рис. 6.2. Все шесть возможных вариантов рассадки трех гостей за столом

Чтобы получить каждую из возможных перестановок {*B*, *C*}, мы рекурсивно повторяем процесс, используя *B* в качестве головы, а *C* в качестве хвоста. Перестановка множества, состоящего из одного символа, представляет саму себя — это наш базовый случай. Помещая головной элемент *B* во все возможные положения относительно *C*, получаем перестановки *BC* и *CB*, упомянутые в предыдущем абзаце. Помните, что для множеств порядок не имеет значения ({*B*, *C*} эквивалентно {*C*, *B*}), однако он важен для перестановок (*BC* — это не то же самое, что *CB*).

Рекурсивная функция нахождения перестановок принимает в качестве аргумента строку символов и возвращает массив строк, содержащий все возможные комбинации этих символов.

Ответим на наши три вопроса относительно изученной рекурсивной функции.

Что представляет собой базовый случай? Аргумент, состоящий из односимвольной или пустой строки. В данном случае возвращается массив, содержащий только эту строку.

Какой аргумент передается рекурсивной функции при ее вызове? Строковый аргумент, в котором отсутствует один символ. Для каждого отсутствующего символа выполняется отдельный рекурсивный вызов.

Как этот аргумент приближается к базовому случаю? Размер строки постепенно уменьшается, и в конечном итоге она становится односимвольной.

Рекурсивный алгоритм поиска перестановок реализован в программе `permutations.py`:

```
def getPerms(chars, indent=0):
    print('.' * indent + 'Start of getPerms("' + chars + '")')
    if len(chars) == 1: ❶
        # БАЗОВЫЙ СЛУЧАЙ
        print('.' * indent + 'When chars = "' + chars + '" base case returns', chars)
        return [chars]

    # РЕКУРСИВНЫЙ СЛУЧАЙ
    permutations = []
    head = chars[0] ❷
    tail = chars[1:]    tailPermutations = getPerms(tail, indent + 1)
    for tailPerm in tailPermutations: ❸
        print('.' * indent + 'When chars =', chars, 'putting head', head, 'in all
        places in', tailPerm)
        for i in range(len(tailPerm) + 1): ❹
            newPerm = tailPerm[0:i] + head + tailPerm[i:]
            print('.' * indent + 'New permutation:', newPerm)
            permutations.append(newPerm)
    print('.' * indent + 'When chars =', chars, 'results are', permutations)
    return permutations

print('Permutations of "ABCD":')
print('Results:', ','.join(getPerms('ABCD')))
```

Эквивалентная программа на языке JavaScript находится в файле `permutations.html`:

```
<script type="text/javascript">
function getPerms(chars, indent) {
    if (indent === undefined) {
        indent = 0;
    }
    document.write('.'.repeat(indent) + 'Start of getPerms("' + chars + '")<br />');
    if (chars.length === 1) { ❶
        // БАЗОВЫЙ СЛУЧАЙ
        document.write('.'.repeat(indent) + "When chars = \"" + chars +
```

```
        "\" base case returns " + chars + "<br />");
        return [chars];
    }
    // РЕКУРСИВНЫЙ СЛУЧАЙ
    let permutations = [];
    let head = chars[0]; ❷
    let tail = chars.substring(1);
    let tailPermutations = getPerms(tail, indent + 1);
    for (tailPerm of tailPermutations) { ❸
        document.write('.'.repeat(indent) + "When chars = " + chars +
        " putting head " + head + " in all places in " + tailPerm + "<br />");
        for (let i = 0; i < tailPerm.length + 1; i++) { ❹
            let newPerm = tailPerm.slice(0, i) + head + tailPerm.slice(i);
            document.write('.'.repeat(indent) + "New permutation: " +
              newPerm + "<br />");
            permutations.push(newPerm);        }
    }
    document.write('.'.repeat(indent) + "When chars = " + chars +
    " results are " + permutations + "<br />");
    return permutations;
}

document.write("<pre>Permutations of \"ABCD\":<br />");
document.write("Results: " + getPerms("ABCD") + "</pre>");
</script>
```

Результат выполнения этих программ выглядит следующим образом:

```
Permutations of "ABCD":
Start of getPerms("ABCD")
.Start of getPerms("BCD")
..Start of getPerms("CD")
...Start of getPerms("D")
...When chars = "D" base case returns D
..When chars = CD putting head C in all places in D
..New permutation: CD
..New permutation: DC
..When chars = CD results are ['CD', 'DC']
.When chars = BCD putting head B in all places in CD
.New permutation: BCD
.New permutation: CBD
.New permutation: CDB
.When chars = BCD putting head B in all places in DC
.New permutation: BDC
.New permutation: DBC
.New permutation: DCB
.When chars = BCD results are ['BCD', 'CBD', 'CDB', 'BDC', 'DBC', 'DCB']
--пропущенный фрагмент--
When chars = ABCD putting head A in all places in DCB
New permutation: ADCB
New permutation: DACB
```

```
New permutation: DCAB
New permutation: DCBA
When chars = ABCD results are ['ABCD', 'BACD', 'BCAD', 'BCDA', 'ACBD', 'CABD',
'CBAD', 'CBDA', 'ACDB','CADB', 'CDAB', 'CDBA', 'ABDC', 'BADC', 'BDAC', 'BDCA',
'ADBC', 'DABC', 'DBAC', 'DBCA', 'ADCB', 'DACB', 'DCAB', 'DCBA']
Results: ABCD,BACD,BCAD,BCDA,ACBD,CABD,CBAD,CBDA,ACDB,CADB,CDAB,CDBA,ABDC,
BADC,BDAC,BDCA,ADBC,DABC,DBAC,DBCA,ADCB,DACB,DCAB,DCBA
```

После вызова функция `getPerms()` первым делом выполняет проверку на соответствие условиям базового случая ❶. Если строка `chars` состоит из одного символа, то она предусматривает только одну перестановку, эквивалентную самой строке `chars`. Функция возвращает эту строку в виде массива.

Если данное условие не выполняется, то имеет место рекурсивный случай, в рамках которого функция превращает первый символ аргумента `chars` в переменную `head`, а остальные символы — в переменную `tail` ❷. Затем функция выполняет рекурсивный вызов `getPerms()`, чтобы получить все перестановки строки, сохраненной в переменной `tail`. Первый цикл `for` ❸ перебирает эти перестановки, а второй ❹ создает новую перестановку, помещая символ `head` во все возможные положения обрабатываемой строки.

Например, если при вызове функции `getPerms()` в качестве аргумента `chars` передается `ABCD`, то значением переменной `head` становится `A`, а значением переменной `tail` — `BCD`. Вызов `getPerms('BCD')` возвращает массив перестановок хвоста `['BCD', 'CBD', 'CDB', 'BDC', 'DBC', 'DCB']`. Первый цикл `for` начинает обработку с перестановки `BCD`, а второй помещает строку `A`, хранящуюся в переменной `head`, во все возможные положения, создавая перестановки `ABCD`, `BACD`, `BCAD`, `BCDA`. Этот процесс повторяется для остальных перестановок хвоста, после чего `getPerms()` возвращает весь список.

Поиск перестановок с помощью вложенных циклов: далеко не идеальный подход

Допустим, у нас есть четырехзначный кодовый велосипедный замок как на рис. 6.3. Он поддерживает 10 000 возможных комбинаций цифр (от 0000 до 9999), но лишь одна разблокирует его (в этом контексте слово *«комбинация»* правильнее было бы заменить на *перестановку с повторениями*, поскольку порядок цифр в данном случае имеет значение).

Теперь предположим, что у нас есть гораздо более простой четырехзначный замок, который поддерживает всего пять букв от A до E. Количество их возможных сочетаний составляет $5^4 = 5 \times 5 \times 5 \times 5 = 625$. Кодовый замок, поддерживающий k символов, выбираемых из множества, включающего n элементов, предусматривает n^k комбинаций. Однако получить список самих вариантов шифра не так просто.

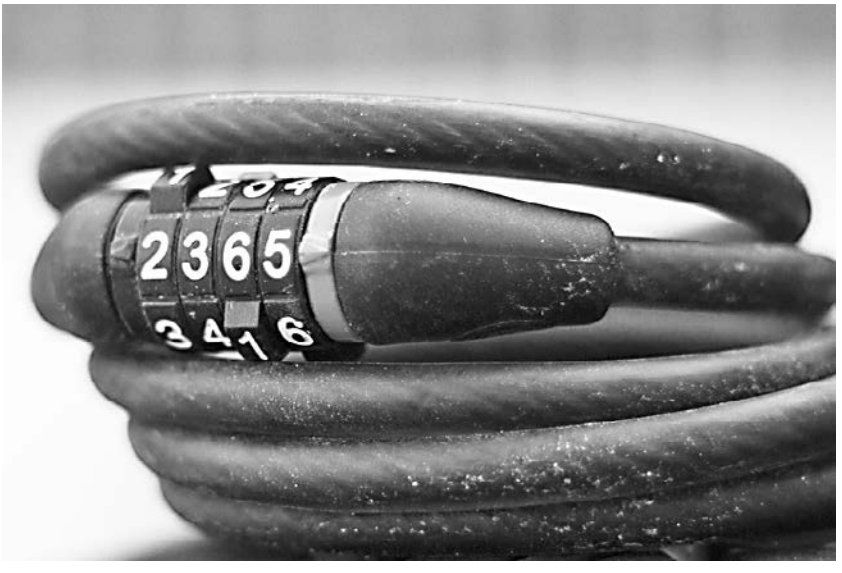


Рис. 6.3. Четырехзначный кодовый велосипедный замок поддерживает 10^4, или 10 000, возможных перестановок с повторениями (фото предоставлено Шоном Фишером (Shaun Fisher) по лицензии CC BY 2.0)

Один из способов получения списка перестановок с повторениями предполагает использование *вложенного цикла*, то есть цикла внутри другого цикла. Внутренний цикл обрабатывает каждый элемент множества, а внешний делает то же самое, повторяя операции внутреннего цикла. Для создания всех возможных k-элементных перестановок символов, выбранных из n-элементного множества, требуется k вложенных циклов.

Например, в файле `nestedLoopPermutations.py` содержится код, который генерирует все 4-элементные сочетания множества {*A*, *B*, *C*, *D*, *E*}:

```
for a in ['A', 'B', 'C', 'D', 'E']:
    for b in ['A', 'B', 'C', 'D', 'E']:
        for c in ['A', 'B', 'C', 'D', 'E']:
            for d in ['A', 'B', 'C', 'D', 'E']:
                print(a, b, c, d)
```

Эквивалентная программа на языке JavaScript содержится в файле `nestedLoopPermutations.html`:

```
<script>
for (a of ['A', 'B', 'C', 'D', 'E']) {
    for (b of ['A', 'B', 'C', 'D', 'E']) {
        for (c of ['A', 'B', 'C', 'D', 'E']) {
```

```
            for (d of ['A', 'B', 'C', 'D', 'E']) {
                document.write(a + b + c + d + "<br />")
            }
        }
    }
}
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
A A A A
A A A B
A A A C
A A A D
A A A E
A A B A
A A B B
-- пропущенный фрагмент --
E E E C
E E E D
E E E E
```

Генерация перестановок с помощью четырех вложенных циклов работает только для перестановок, содержащих ровно четыре символа. Вложенные циклы не могут генерировать перестановки произвольной длины. Вместо них допускается применить рекурсивную функцию, описанную в следующем разделе.

Вы можете легко разобраться в разнице между перестановками с повторениями и без них с помощью примеров, приведенных в текущей главе. Перестановки *без* повторений — это все возможные варианты упорядочения элементов множества, как в примере со схемой рассадки гостей на свадьбе. Перестановки *с* повторениями — это все возможные комбинации кодового замка; в данном случае порядок имеет значение и один и тот же элемент может встречаться более одного раза.

Перестановки с повторениями: взломщик паролей

Представьте, что вы получили секретный зашифрованный файл от недавно умершего журналиста. В своем последнем сообщении он сказал вам, что данный файл содержит доказательства уклонения от налогов одного нечистого на руку миллиардера. Чтобы открыть файл, необходимо ввести код, который состоит из четырех знаков, а самыми вероятными символами являются числа 2, 4, 8 и буквы J, P, B. Все они могут встречаться более одного раза, например JPB2, JJJJ или 2442.

Чтобы сгенерировать список потенциальных комбинаций паролей на основе имеющейся информации, вам нужно получить все возможные 4-элементные перестановки с повторениями, состоящие из элементов множества {J, P, B, 2, 4, 8}. Каждый

из четырех символов пароля может иметь одно из шести возможных значений, что дает $6 \times 6 \times 6 \times 6 = 6^4$, или 1296, перестановок. Нам нужно сгенерировать именно перестановки элементов множества {J, P, B, 2, 4, 8}, а не сочетания, потому что порядок имеет значение (пароль JPB2 отличается от B2JP).

Зададим наши три вопроса относительно рекурсивной функции нахождения перестановок. Вместо *k* мы будем использовать более описательное имя `permLength`.

Что представляет собой базовый случай? Аргумент `permLength`, равный `0`, означает, что длина перестановки равна нулю, а также сигнализирует, что аргумент `prefix` содержит всю перестановку, поэтому `prefix` должен быть возвращен в виде массива.

Какой аргумент передается рекурсивной функции при ее вызове? Строка `chars`, состоящая из символов, перестановки которых мы хотим получить, аргумент `permLength`, начальное значение которого равно длине строки `chars`, и аргумент `prefix`, представляющий собой пустую строку. В ходе каждого рекурсивного вызова значение аргумента `permLength` уменьшается на единицу и один символ строки `chars` добавляется к аргументу `prefix`.

Как этот аргумент приближается к базовому случаю? В конце концов значение аргумента `permLength` уменьшается до `0`.

Рекурсивный алгоритм получения перестановок с повторениями реализован в программе `permutationsWithRepetition.py`:

```
def getPermsWithRep(chars, permLength=None, prefix=''):
    indent = '.' * len(prefix)
    print(indent + 'Start, args=("' + chars + '", ' + str(permLength) + ', "' +
    prefix + '")')
    if permLength is None:
        permLength = len(chars)

    # БАЗОВЫЙ СЛУЧАЙ
    if (permLength == 0): ❶
        print(indent + 'Base case reached, returning', [prefix])
        return [prefix]

    # РЕКУРСИВНЫЙ СЛУЧАЙ
    # Создаем новый префикс путем добавления каждого символа к текущему префиксу
    results = []
    print(indent + 'Adding each char to prefix "' + prefix + '".')
    for char in chars:
        newPrefix = prefix + char ❷
        # Уменьшаем permLength на единицу после добавления одного символа к префиксу
        results.extend(getPermsWithRep (chars, permLength - 1, newPrefix)) ❸
    print(indent + 'Returning', results)
    return results

print('All permutations with repetition of JPB123:')
print(getPermsWithRep('JPB123', 4))
```

Эквивалентная программа на языке JavaScript находится в файле `permutations-WithRepetition.html`:

```
<script type="text/javascript">
function getPermsWithRep(chars, permLength, prefix) {
    if (permLength === undefined) {
        permLength = chars.length;
    }
    if (prefix === undefined) {
        prefix = "";
    }
    let indent = ".".repeat(prefix.length);
    document.write(indent + "Start, args=(\"" + chars + "\", " + permLength +
    ", \"" + prefix + "\")<br />");

    // БАЗОВЫЙ СЛУЧАЙ
    if (permLength === 0) { ❶
        document.write(indent + "Base case reached, returning " + [prefix] + "<br />");
        return [prefix];
    }
    // РЕКУРСИВНЫЙ СЛУЧАЙ
    // Создаем новый префикс путем добавления каждого символа к текущему префиксу
    let results = [];
    document.write(indent + "Adding each char to prefix \"" + prefix + "\".<br />");
    for (char of chars) {
        let newPrefix = prefix + char; ❷

        // Уменьшаем permLength на единицу после добавления одного символа к префиксу
        results = results.concat(getPermsWithRep(chars, permLength - 1,
          newPrefix)); ❸
    }
    document.write(indent + "Returning " + results + "<br />");
    return results;
}

document.write("<pre>All permutations with repetition of JPB123:<br />");
document.write(getPermsWithRep('JPB123', 4) + "</pre>");
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
All permutations with repetition of JPB123:
Start, args=("JPB123", 4, "")
Adding each char to prefix "".
.Start, args=("JPB123", 3, "J")
.Adding each char to prefix "J".
..Start, args=("JPB123", 2, "JJ")
..Adding each char to prefix "JJ".
...Start, args=("JPB123", 1, "JJJ")
```

```
...Adding each char to prefix "JJJ".
....Start, args=("JPB123", 0, "JJJJ")
....Base case reached, returning ['JJJJ']
....Start, args=("JPB123", 0, "JJJP")
....Base case reached, returning ['JJJP']
--пропущенный фрагмент--
Returning ['JJJJ', 'JJJP', 'JJJB', 'JJJ1', 'JJJ2', 'JJJ3',
'JJPJ', 'JJPP', 'JJPB', 'JJP1', 'JJP2', 'JJP3', 'JJBJ', 'JJBP',
'JJBB', 'JJB1', 'JJB2', 'JJB3', 'JJ1J', 'JJ1P', 'JJ1B', 'JJ11',
'JJ12', 'JJ13', 'JJ2J', 'JJ2P', 'JJ2B', 'JJ21', 'JJ22', 'JJ23',
'JJ3J', 'JJ3P', 'JJ3B', 'JJ31', 'JJ32', 'JJ33', 'JPJJ',
--пропущенный фрагмент--
```

У функции `getPermsWithRep()` есть строковый аргумент `prefix`, который по умолчанию является пустой строкой. После вызова функция выполняет проверку на соответствие условиям базового случая ❶. Если `permLength` (длина перестановок) равна `0`, возвращается массив с аргументом `prefix`.

Если это условие не выполняется, имеет место рекурсивный случай и для каждого символа в строке `chars` функция создает новый префикс ❷, который передается функции `getPermsWithRep()` в ходе ее рекурсивного вызова. В качестве аргумента `permLength` в ходе данного рекурсивного вызова передается `permLength - 1`.

Начальное значение аргумента `permLength` равно длине перестановок, и в результате каждого рекурсивного вызова оно уменьшается на единицу ❸. Изначально аргумент `prefix` — это пустая строка, в которую в результате каждого рекурсивного вызова добавляется один символ. Таким образом, к моменту достижения базового случая `k == 0` строка `prefix` представляет собой полную перестановку, длина которой равна `k`.

Для примера рассмотрим вызов функции `getPermsWithRep('ABC', 2)`. Аргумент `prefix` по умолчанию является пустой строкой. Функция выполняет рекурсивные вызовы, в ходе которых последовательно добавляет каждый из символов `ABC` к строке префикса. Вызов `getPermsWithRep('ABC', 2)` предусматривает следующие три рекурсивных вызова:

- `getPermsWithRep('ABC', 1, 'A')`;
- `getPermsWithRep('ABC', 1, 'B')`;
- `getPermsWithRep('ABC', 1, 'C')`.

Каждый из них выполнит свои три рекурсивных вызова, но в качестве значения `permLength` передаст `0` вместо `1`. Базовый случай возникает, когда `permLength == 0`, поэтому подобные вызовы возвратят собственные префиксы. Именно так генерируются все девять перестановок. Перестановки элементов более крупных множеств функция `getPermsWithRep()` будет осуществлять таким же способом.

Получение k-элементных сочетаний с помощью рекурсии

Как говорилось ранее, порядок имеет значение для перестановок, но не для сочетаний. Однако сгенерировать все *k*-элементные сочетания не так легко, поскольку нам нужно, чтобы алгоритм не формировал дубликаты: при создании 2-элементных сочетаний символов *AB* из набора {*A*, *B*, *C*} мы не хотим получить еще и *BA*, потому что оно эквивалентно *AB*.

Чтобы понять принцип написания рекурсивного кода для решения одноименной с заголовком раздела задачи, давайте визуально опишем процесс создания всех *k*-элементных сочетаний с помощью дерева. На рис. 6.4 показано дерево, содержащее все сочетания элементов множества {*A*, *B*, *C*, *D*}.

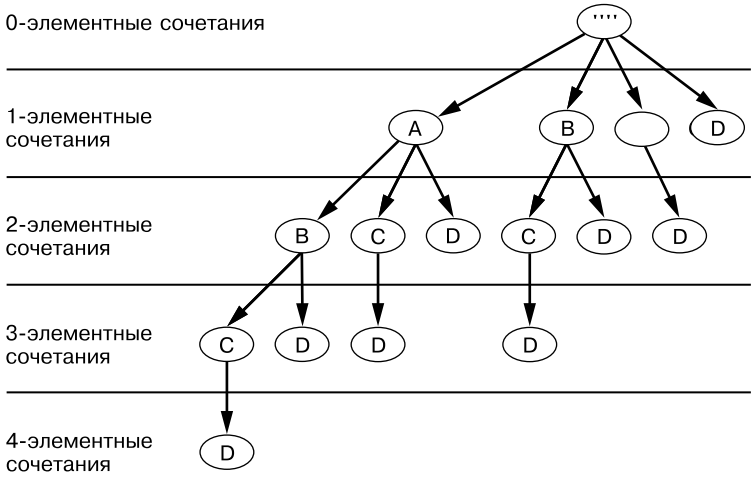


Рис. 6.4. Дерево, содержащее все возможные k-элементные сочетания (от 0 до 4) для множества {A, B, C, D}

Чтобы получить, например, 3-элементные сочетания на основе этого дерева, начните с верхнего корневого узла и выполните обход дерева в глубину до уровня 3-элементных сочетаний, по пути запоминая букву каждого узла (процесс поиска в глубину описан в главе 4). Для получения нашего первого 3-элементного сочетания мы переходим от корня к узлу *A* на первом уровне, затем к *B* на втором и, наконец, к *C* на третьем, где и останавливаемся, добившись нужного 3-элементного сочетания — *ABC*. Для генерации следующего сочетания *ABD* мы переходим от корня к *A*, потом к *B* и следом к *D*. После этого повторяем данный процесс для получения сочетаний *ACD* и *BCD*. Наше дерево предусматривает четыре узла на

третьем уровне, а значит, мы имеем четыре 3-элементных сочетания для множества {*A*, *B*, *C*, *D*}: *ABC*, *ABD*, *ACD* и *BCD*.

Обратите внимание, что корневой узел дерева (см. рис. 6.4) представляет собой пустую строку. Это уровень 0-элементных сочетаний, к которому принадлежат все комбинации, состоящие из нуля элементов множества, то есть это просто пустая строка. Дочерние узлы корня соответствуют элементам множества. В нашем случае это все элементы набора {*A*, *B*, *C*, *D*}. Хотя элементы множества не являются упорядоченными, при создании дерева нам следует проявлять последовательность и придерживаться порядка *ABCD*, поскольку дочерние элементы каждого узла соответствуют буквам, следующим за ним в строке *ABCD*: все узлы *A* имеют дочерние элементы *B*, *C* и *D*; все узлы *B* имеют дочерние элементы *C* и *D*; все узлы *C* имеют один дочерний элемент *D*; а узлы *D* не имеют дочерних элементов.

Хотя это не относится напрямую к рекурсивной функции нахождения сочетаний, здесь прослеживается закономерность в количестве *k*-элементных сочетаний на каждом из уровней:

- на уровнях 0-элементных и 4-элементных сочетаний находится только одно сочетание: пустая строка и *ABCD* соответственно;
- на уровнях 1-элементных и 3-элементных сочетаний — четыре: *A*, *B*, *C*, *D* и *ABC*, *ABD*, *ACD*, *BCD* соответственно;
- на уровне 2-элементных, в середине, — шесть: *AB*, *AC*, *AD*, *BC*, *BD* и *CD*.

Причина, по которой количество сочетаний увеличивается, достигает максимума в середине, а затем уменьшается, заключается в том, что *k*-элементные сочетания являются своеобразными противоположностями друг для друга. Например, 1-элементные сочетания состоят из элементов, не включенных в 3-элементные:

- 1-элементное сочетание *A* — противоположность для 3-элементного сочетания *BCD*;
- 1-элементное сочетание *B* — противоположность для 3-элементного сочетания *ACD*;
- 1-элементное сочетание *C* — противоположность для 3-элементного сочетания *ABD*;
- 1-элементное сочетание *D* — противоположность для 3-элементного сочетания *ABC*.

Создадим функцию `getCombos()`, которая принимает два аргумента: строку `chars`, состоящую из символов, сочетания которых мы хотим получить, и размер сочетаний `k`. Возвращаемое значение олицетворяет массив строк с сочетаниями символов из строки `chars`, каждая из которых имеет длину `k`.

Применим к аргументу `chars` технику «голова — хвост». Например, допустим, что мы вызвали `getCombos('ABC', 2)`, чтобы получить все 2-элементные сочетания символов из множества {*A*, *B*, *C*}. Функция будет использовать `A` в качестве головы, а `BC` — в качестве хвоста. На рис. 6.5 показано дерево для составления 2-элементных сочетаний символов из набора {*A*, *B*, *C*}.

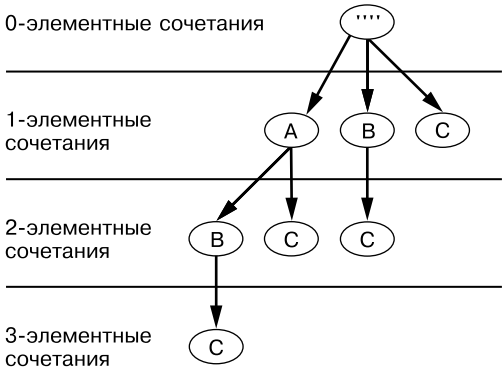


Рис. 6.5. Дерево, содержащее все возможные 2-элементные сочетания символов из множества {A, B, C}

Зададим наши три вопроса относительно этой рекурсивной функции.

Что представляет собой базовый случай? Первый базовый случай имеет место, когда аргумент `k` равен `0`. Это означает запрос 0-элементного сочетания, которое всегда представляет собой массив с пустой строкой вне зависимости от значения аргумента `chars`. Второй базовый случай возникает, когда аргумент `chars` является пустой строкой. При этом возвращается пустой массив, поскольку из пустой строки нельзя составить никаких сочетаний.

Какой аргумент передается рекурсивной функции при ее вызове? Для первого рекурсивного вызова передается хвост `chars` и значение `k - 1`, а в ходе второго — хвост аргумента `chars` и значение `k`.

Как этот аргумент приближается к базовому случаю? Поскольку в ходе каждого рекурсивного вызова значение `k` уменьшается на единицу, а из аргумента `chars` последовательно удаляются головные элементы, в конечном итоге либо `k` достигает `0`, либо аргумент `chars` превращается в пустую строку.

Программа Python для генерации сочетаний находится в файле `combinations.py`:

```
def getCombos(chars, k, indent=0):
    debugMsg = '.' * indent + "In getCombos('" + chars + "', " + str(k) + ")"
    print(debugMsg + ', start.')
```

```
    if k == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        print(debugMsg + " base case returns ['']")
        # Если запрашивается 0-элементное сочетание, возвращаем ' '
        # в качестве выборки, состоящей из нуля символов аргумента chars.
        return ['']
    elif chars == '':
        # БАЗОВЫЙ СЛУЧАЙ
        print(debugMsg + ' base case returns []')
        return [] # Если аргумент chars — пустая строка, то вне зависимости
                  # от значения k из его символов нельзя получить никаких сочетаний

    # РЕКУРСИВНЫЙ СЛУЧАЙ
    combinations = []
❶   # Первая часть: получение сочетаний, содержащих головной элемент:
    head = chars[:1]
    tail = chars[1:]
    print(debugMsg + " part 1, get combos with head '" + head + "'")
❷   tailCombos = getCombos(tail, k - 1, indent + 1)
    print('.' * indent + "Adding head '" + head + "' to tail combos:")
    for tailCombo in tailCombos:
        print('.' * indent + 'New combination', head + tailCombo)
        combinations.append(head + tailCombo)

❸   # Вторая часть: получение сочетаний, не содержащих головной элемент:
    print(debugMsg + " part 2, get combos without head '" + head + "')")
❹   combinations.extend(getCombos(tail, k, indent + 1))

    print(debugMsg + ' results are', combinations)
    return combinations

print('2-combinations of "ABC":')
print('Results:', getCombos('ABC', 2))
```

Эквивалентная программа на языке JavaScript находится в файле `combinations.html`:

```
<script type="text/javascript">
function getCombos(chars, k, indent) {
    if (indent === undefined) {
        indent = 0;
    }
    let debugMsg = ".".repeat(indent) + "In getCombos('" + chars + "', " + k + ")";
    document.write(debugMsg + ", start.<br />");
    if (k == 0) {
        // БАЗОВЫЙ СЛУЧАЙ
        document.write(debugMsg + " base case returns ['']<br />");
        // Если запрашивается 0-элементное сочетание, возвращаем ' '
        // в качестве выборки, состоящей из нуля символов аргумента chars.
        return [""];
    } else if (chars == "") {
```

```
        // БАЗОВЫЙ СЛУЧАЙ
        document.write(debugMsg + " base case returns []<br />");
        return []; // Если аргумент chars является пустой строкой,
                   // то вне зависимости от значения k из его символов
                   // нельзя получить никаких сочетаний
    }

    // РЕКУРСИВНЫЙ СЛУЧАЙ
    let combinations = [];
    // Первая часть: получение сочетаний, содержащих головной элемент: ❶
    let head = chars.slice(0, 1);
    let tail = chars.slice(1, chars.length);
    document.write(debugMsg + " part 1, get combos with head '" + head + "'<br />");
    let tailCombos = getCombos(tail, k - 1, indent + 1); ❷
    document.write(".".repeat(indent) + "Adding head '" + head + "' to tail
    combos:<br />");
    for (tailCombo of tailCombos) {
        document.write(".".repeat(indent) + "New combination " + head +
          tailCombo + "<br />");
        combinations.push(head + tailCombo);
    }
    // Вторая часть: получение сочетаний, не содержащих головной элемент: ❸
    document.write(debugMsg + " part 2, get combos without head '" + head + "')
    <br />");
    combinations = combinations.concat(getCombos(tail, k, indent + 1)); ❹

    document.write(debugMsg + " results are " + combinations + "<br />");
    return combinations;
}

document.write('<pre>2-combinations of "ABC":<br />');
document.write("Results: " + getCombos("ABC", 2) + "<br /></pre>");
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
2-combinations of "ABC":
In getCombos('ABC', 2), start.
In getCombos('ABC', 2) part 1, get combos with head 'A'
.In getCombos('BC', 1), start.
.In getCombos('BC', 1) part 1, get combos with head 'B'
..In getCombos('C', 0), start.
..In getCombos('C', 0) base case returns ['']
.Adding head 'B' to tail combos:
.New combination B
.In getCombos('BC', 1) part 2, get combos without head 'B')
..In getCombos('C', 1), start.
..In getCombos('C', 1) part 1, get combos with head 'C'
...In getCombos('', 0), start.
...In getCombos('', 0) base case returns ['']
..Adding head 'C' to tail combos:
```

```
..New combination C
..In getCombos('C', 1) part 2, get combos without head 'C')
...In getCombos('', 1), start.
...In getCombos('', 1) base case returns []
..In getCombos('C', 1) results are ['C']
.In getCombos('BC', 1) results are ['B', 'C']
 Adding head 'A' to tail combos:
New combination AB
New combination AC
In getCombos('ABC', 2) part 2, get combos without head 'A')
.In getCombos('BC', 2), start.
 .In getCombos('BC', 2) part 1, get combos with head 'B'
..In getCombos('C', 1), start.
..In getCombos('C', 1) part 1, get combos with head 'C'
...In getCombos('', 0), start.
...In getCombos('', 0) base case returns ['']
..Adding head 'C' to tail combos:
..New combination C
..In getCombos('C', 1) part 2, get combos without head 'C')
...In getCombos('', 1), start.
...In getCombos('', 1) base case returns []
..In getCombos('C', 1) results are ['C']
.Adding head 'B' to tail combos:
.New combination BC
.In getCombos('BC', 2) part 2, get combos without head 'B')
..In getCombos('C', 2), start.
..In getCombos('C', 2) part 1, get combos with head 'C'
...In getCombos('', 1), start.
...In getCombos('', 1) base case returns []
 ..Adding head 'C' to tail combos:
..In getCombos('C', 2) part 2, get combos without head 'C')
...In getCombos('', 2), start.
...In getCombos('', 2) base case returns []
..In getCombos('C', 2) results are []
.In getCombos('BC', 2) results are ['BC']
In getCombos('ABC', 2) results are ['AB', 'AC', 'BC']
Results: ['AB', 'AC', 'BC']
```

Каждый вызов функции `getCombos()` предусматривает два рекурсивных вызова для двух частей алгоритма. В нашем примере первая часть ❶ предполагает получение всех сочетаний, включающих головной элемент `A`. Данный шаг генерирует все комбинации *под* узлом A, находящимся в дереве на уровне 1-элементных сочетаний.

Для этого мы можем передать хвост (`tail`) и значение `k - 1` в рамках первого рекурсивного вызова `getCombos('BC', 1)` ❷. К каждому сочетанию, возвращаемому этим вызовом, мы добавляем `A`. Воспользуемся принципом «прыжка веры» и просто предположим, что функция `getCombos()` возвращает правильный список *k*-элементных сочетаний `['B', 'C']`, несмотря на то что мы ее еще не дописали. Теперь все *k*-комбинации, включающие головной элемент `A`, находятся в результирующем массиве `['AB', 'AC']`.

Вторая часть алгоритма ❸ генерирует все сочетания, которые не включают головной элемент `A`, то есть комбинации, создаваемые *справа* от узла дерева `A`. Для этого можно передать хвост и значение `k` в рамках второго рекурсивного вызова `getCombos('BC', 2)`. Данный вызов возвращает `['BC']`, так как `BC` является единственным 2-элементным сочетанием символов `B` и `C`.

Результаты двух рекурсивных вызовов `getCombos('ABC', 2)` — `['AB', 'AC']` и `['BC']` — конкатенируются и возвращаются в виде `['AB', 'AC', 'BC']` ❹. Сочетания большей длины функция `getCombos()` генерирует тем же способом.

Получение всех комбинаций сбалансированных скобок

Скобки называются *сбалансированными*, если в строке за каждой открывающей скобкой следует одна закрывающая. Например, скобки в строках `'()()'` и `'(())'` — сбалансированные, а в строках `')(()'` и `'(()'` — нет. Такие строки также называются *словами Дика* в честь математика Вальтера фон Дика.

На собеседованиях соискателя часто просят написать рекурсивную функцию, которая получает в качестве аргумента некоторое количество пар скобок и генерирует все их возможные сбалансированные комбинации. Например, вызов функции `getBalancedParens(3)` должен возвратить `['((()))', '(()())', '(())()', '()(())', '()()()']`. Обратите внимание, что вызов `getBalancedParens(`*`n`*`)` возвращает строки длиной $2n$ символов, поскольку каждая строка состоит из n пар скобок.

Мы могли бы попытаться решить эту задачу, найдя все перестановки пар скобок, но в результате получили бы не только сбалансированные, но и несбалансированные наборы. Даже если бы впоследствии эти недопустимые строки были отфильтрованы, для n пар скобок существует $2n!$ перестановок, так что данный алгоритм оказался бы слишком медленным и непрактичным.

Вместо этого допускается реализовать рекурсивную функцию для генерирования всех строк, состоящих из сбалансированных скобок. Наша функция `getBalancedParens()` принимает целое число, соответствующее количеству пар скобок, и возвращает список строк со сбалансированными скобками. Функция формирует эти строки, добавляя в них открывающую или закрывающую скобку: первая может быть добавлена только в том случае, если еще не все такие скобки были использованы, вторая же — если к настоящему моменту открывающих скобок было добавлено больше, чем закрывающих.

Мы будем отслеживать количество доступных для использования открывающих и закрывающих скобок с помощью параметров `openRem` и `closeRem`. Создаваемая в этот момент строка представляет собой еще один параметр функции с именем `current`,

который служит той же цели, что и `prefix` в программе `permutationsWithRepetition`. Первый базовый случай имеет место, когда значения `openRem` и `closeRem` равны `0`, то есть скобок для добавления в строку `current` больше не осталось. Второй базовый случай возникает после завершения рекурсивных вызовов, в ходе которых были получены списки строк со скобками, сбалансированными путем добавления открывающей и/или закрывающей скобки.

Ответим на три вопроса относительно рекурсивной функции `getBalancedParens()`.

Что представляет собой базовый случай? Первый базовый случай имеет место, когда количество доступных для добавления открывающих и закрывающих скобок достигает `0`, второй — после завершения рекурсивных вызовов.

Какой аргумент передается рекурсивной функции при ее вызове? Общее количество пар скобок (`pairs`), количество доступных для добавления открывающих и закрывающих скобок (`openRem` и `closeRem`) и формируемая в данный момент строка (`current`).

Как этот аргумент приближается к базовому случаю? По мере добавления открывающих и закрывающих скобок в строку `current` значения аргументов `openRem` и `closeRem` уменьшаются вплоть до нуля.

Код данной рекурсивной функции на языке Python содержится в файле `balanceParentheses.py`:

```
def getBalancedParens(pairs, openRem=None, closeRem=None, current='', indent=0):
    if openRem is None: ❶
        openRem = pairs
    if closeRem is None:
        closeRem = pairs

    print('.' * indent, end='')
    print('Start of pairs=' + str(pairs) + ', openRem=' +
    str(openRem) + ', closeRem=' + str(closeRem) + ', current="' + current + '"')
    if openRem == 0 and closeRem == 0: ❷
        # БАЗОВЫЙ СЛУЧАЙ
        print('.' * indent, end='')
        print('1st base case. Returning ' + str([current]))
        return [current] ❸
    # РЕКУРСИВНЫЙ СЛУЧАЙ
    results = []
    if openRem > 0: ❹
        print('.' * indent, end='')
        print('Adding open parenthesis.')
        results.extend(getBalancedParens(pairs, openRem - 1, closeRem,
        current + '(', indent + 1))
    if closeRem > openRem: ❺
        print('.' * indent, end='')
```

```
        print('Adding close parenthesis.')
        results.extend(getBalancedParens(pairs, openRem, closeRem - 1,
        current + ')', indent + 1))

    # БАЗОВЫЙ СЛУЧАЙ
    print('.' * indent, end='')
    print('2nd base case. Returning ' + str(results))
    return results ❻

print('All combinations of 2 balanced parentheses:')
print('Results:', getBalancedParens(2))
```

Эквивалентная программа на языке JavaScript находится в файле `balanceParentheses.html`:

```
<script type="text/javascript">
function getBalancedParens(pairs, openRem, closeRem, current, indent) {
    if (openRem === undefined) { ❶
        openRem = pairs;
    }
    if (closeRem === undefined) {
        closeRem = pairs;
    }
    if (current === undefined) {
        current = "";
    }    if (indent === undefined) {
        indent = 0;
    }

    document.write(".".repeat(indent) + "Start of pairs=" +
    pairs + ", openRem=" + openRem + ", closeRem=" +
    closeRem + ", current=\"" + current + "\"<br />");
    if (openRem === 0 && closeRem === 0) { ❷
        // БАЗОВЫЙ СЛУЧАЙ
        document.write(".".repeat(indent) +
        "1st base case. Returning " + [current] + "<br />");
        return [current]; ❸
    }
    // РЕКУРСИВНЫЙ СЛУЧАЙ
    let results = [];
    if (openRem > 0) { ❹
        document.write(".".repeat(indent) + "Adding open parenthesis.<br />");
        Array.prototype.push.apply(results, getBalancedParens(
        pairs, openRem - 1, closeRem, current + '(', indent + 1));
    }
    if (closeRem > openRem) { ❺
        document.write(".".repeat(indent) + "Adding close parenthesis.<br />");
        results = results.concat(getBalancedParens(
        pairs, openRem, closeRem - 1, current + ')', indent + 1));
    }
```

```
    // БАЗОВЫЙ СЛУЧАЙ
    document.write(".".repeat(indent) + "2nd base case. Returning " + results +
    "<br />");
    return results; ❻
}

document.write(<pre>"All combinations of 2 balanced parentheses:<br />");
document.write("Results: ", getBalancedParens(2), "</pre>");
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
All combinations of 2 balanced parentheses:
Start of pairs=2, openRem=2, closeRem=2, current=""
Adding open parenthesis.
.Start of pairs=2, openRem=1, closeRem=2, current="("
.Adding open parenthesis.
..Start of pairs=2, openRem=0, closeRem=2, current="(("
..Adding close parenthesis.
...Start of pairs=2, openRem=0, closeRem=1, current="(()"
...Adding close parenthesis.
....Start of pairs=2, openRem=0, closeRem=0, current="(())"
....1st base case. Returning ['(())']
...2nd base case. Returning ['(())']
..2nd base case. Returning ['(())']
.Adding close parenthesis.
..Start of pairs=2, openRem=1, closeRem=1, current="()"
..Adding open parenthesis.
...Start of pairs=2, openRem=0, closeRem=1, current="()("
...Adding close parenthesis.
....Start of pairs=2, openRem=0, closeRem=0, current="()()"
....1st base case. Returning ['()()']
...2nd base case. Returning ['()()']
..2nd base case. Returning ['()()']
.2nd base case. Returning ['(())', '()()']
2nd base case. Returning ['(())', '()()']
Results: ['(())', '()()']
```

При вызове функции `getBalancedParens()` ❶ пользователь должен передать ей один аргумент, равный количеству пар скобок. Однако в ходе рекурсивных вызовов ей должны быть направлены дополнительные аргументы, в том числе: количество оставшихся открывающих (`openRem`) и закрывающих скобок (`closeRem`) и формируемая в настоящий момент строка со сбалансированными скобками (`current`). Исходное значение `openRem` и `closeRem` совпадает со значением `pairs`, а строка `current` изначально пуста. Аргумент `indent` используется только для добавления отступа в выходных данных отладки. Отступ обозначает уровень рекурсии, соответствующий тому или иному вызову функции.

Сначала функция проверяет количество открывающих и закрывающих скобок, доступных для добавления ❷. Если значения обоих аргументов равны нулю, значит,

мы достигли первого базового случая и строка `current` полностью сформирована. Поскольку `getBalancedParens()` возвращает список строк, строку `current` помещаем в список и возвращаем его ❸.

Если же условие не соблюдается, имеет место рекурсивный случай. Когда количество доступных открывающих скобок превышает ноль ❹, функция вызывает `getBalancedParens()` для добавления открывающей скобки в строку `current`. Если доступных закрывающих скобок больше, чем открывающих ❺, функция вызывает `getBalancedParens()` для добавления закрывающей скобки в строку `current`. Такая проверка гарантирует, что в строке не окажется лишней закрывающей скобки, делающей строку несбалансированной, как, например, в случае со строкой `())`.

После завершения этих рекурсивных случаев возникает безусловный базовый случай, который возвращает все строки, возвращаемые из двух рекурсивных вызовов (и, разумеется, из рекурсивных вызовов, выполненных этими рекурсивными вызовами, и т. д.) ❻.

Булеан множества: поиск всех подмножеств множества

Булеан (также *степень множества* или *показательное множество*) — это совокупность всех подмножеств данного множества. Например, булеаном множества {*A*, *B*, *C*} является {{ }, {*A*}, {*B*}, {*C*}, {*A*, *B*}, {*A*, *C*}, {*B*, *C*}, { *A*, *B*, *C* }}. Булеан множества эквивалентен совокупности всех его возможных *k*-элементных сочетаний. То есть булеан множества {*A*, *B*, *C*} содержит все его 0-, 1-, 2- и 3-элементные сочетания.

Сгенерировать степень множества может потребоваться разве что на собеседовании. Скорее всего, больше вы никогда не столкнетесь с подобной задачей, даже в ходе работы, на которую устраиваетесь.

Чтобы найти все подмножества конкретного множества, мы могли бы многократно вызывать созданную ранее функцию `getCombos()`, передавая ей все вероятные аргументы *k*. Данный подход реализован в программах `powerSetCombinations.py` и `powerSetCombinations.html`, которые можно загрузить со страницы: https://nostarch.com/recursive-book-recursion.

Однако для решения этой задачи существует более эффективный способ. Допустим, у нас есть множество {*A*, *B*}, булеаном которого является {{*A*, *B*}, {*A*}, {*B*}, {}}. Теперь предположим, что мы добавили в это множество еще один элемент, *C*, и решили сгенерировать булеан набора {*A*, *B*, *C*}. Для этого добавляем к уже сгенерированным подмножествам те же самые подмножества, в которые был добавлен элемент *C*: {{*A*, *B*, *C*}, {*A*, *C*}, {*B*, *C*}, {*C*}}. В табл. 6.3 показано, как добавление дополнительных элементов в множество приводит к увеличению количества подмножеств, содержащихся в его булеане.

Таблица 6.3. Рост булеана по мере добавления в множество новых элементов

Множество с новым элементом	Новые подмножества для включения в булеан	Итоговый булеан
{ }	{ }	{{ }}
{A}	{A}	{{ }, **{A}**}
{A, B}	{B}, {A, B}	{{ }, {A}, **{B}**, **{A, B}**}
{A, B, C}	{C}, {A, C}, {B, C}, {A, B, C}	{{ }, {A}, {B}, **{C}**, {A, B}, **{A, C}**, **{B, C}**, **{A, B, C}**}
{A, B, C, D}	{D}, {A, D}, {B, D}, {C, D}, {A, B, D}, {A, C, D}, {B, C, D}, {A, B, C, D}	{{ }, {A}, {B}, {C}, **{D}**, {A, B}, {A, C}, **{A, D}**, {B, C}, **{B, D}**, **{C, D}**, {A, B, C}, {A, B, D}, {A, C, D}, **{B, C, D}**, **{A, B, C, D}**}

То, что булеаны больших множеств похожи на булеаны небольших, говорит о том, что для их генерации допускается использовать рекурсивную функцию. Базовым случаем является пустое множество, булеан которого состоит только из этого пустого множества. При реализации данной рекурсивной функции можно использовать технику «голова — хвост». После добавления каждого нового элемента мы будем генерировать булеан хвостового множества для включения в итоговый булеан. Затем к каждому из полученных на предыдущем этапе подмножеств добавим головной элемент. Вместе эти подмножества образуют полный булеан исходного множества, содержащегося в аргументе `chars`.

Ответим на наши три традиционных вопроса в контексте рекурсивного алгоритма для генерации булеана множества.

Что представляет собой базовый случай? Если аргумент `chars` выступает в качестве пустой строки (пустого множества), функция возвращает массив, содержащий только пустую строку, поскольку единственным подмножеством пустого множества является оно само.

Какой аргумент передается рекурсивной функции при ее вызове? Хвост аргумента `chars`.

Как этот аргумент приближается к базовому случаю? Поскольку в ходе рекурсивных вызовов из аргумента `chars` последовательно удаляются головные элементы, в конечном итоге он превращается в пустую строку.

Рекурсивная функция `getPowerSet()` реализована в программе `powerSet.py`:

```
def getPowerSet(chars, indent=0):
    debugMsg = '.' * indent + 'In getPowerSet("' + chars + '")'
    print(debugMsg + ', start.')
```

```
❶ if chars == '':
        # БАЗОВЫЙ СЛУЧАЙ
        print(debugMsg + " base case returns ['']")
        return ['']
    # РЕКУРСИВНЫЙ СЛУЧАЙ
    powerSet = []
    head = chars[0]
    tail = chars[1:]
    # Первая часть: получение подмножеств, не включающих головной элемент:
    print(debugMsg, "part 1, get sets without head '" + head + "'")
❷ tailPowerSet = getPowerSet(tail, indent + 1)

    # Вторая часть: получение подмножеств, включающих головной элемент:
    print(debugMsg, "part 2, get sets with head '" + head + "'")
    for tailSet in tailPowerSet:
        print(debugMsg, 'New set', head + tailSet)
      ❸ powerSet.append(head + tailSet)

    powerSet = powerSet + tailPowerSet
    print(debugMsg, 'returning', powerSet)
❹ return powerSet

print('The power set of ABC:')
print(getPowerSet('ABC'))
```

Эквивалентная программа на языке JavaScript содержится в файле `powerSet.html`:

```
<script type="text/javascript">
function getPowerSet(chars, indent) {
    if (indent === undefined) {
        indent = 0;
    }
    let debugMsg = ".".repeat(indent) + 'In getPowerSet("' + chars + '")';
    document.write(debugMsg + ", start.<br />");

    if (chars == "") { ❶
        // БАЗОВЫЙ СЛУЧАЙ
        document.write(debugMsg + " base case returns ['']<br />");
        return [''];
    }

    // РЕКУРСИВНЫЙ СЛУЧАЙ
    let powerSet = [];
    let head = chars[0];
    let tail = chars.slice(1, chars.length);

    // Первая часть: получение подмножеств, не включающих головной элемент:
    document.write(debugMsg +
    " part 1, get sets without head '" + head + "'<br />");
    let tailPowerSet = getPowerSet(tail, indent + 1); ❷
```

```
    // Вторая часть: получение подмножеств, включающих головной элемент:
    document.write(debugMsg +
    " part 2, get sets with head '" + head + "'<br />");
    for (tailSet of tailPowerSet) {
        document.write(debugMsg + " New set " + head + tailSet + "<br />");
        powerSet.push(head + tailSet); ❸
    }

    powerSet = powerSet.concat(tailPowerSet);
    document.write(debugMsg + " returning " + powerSet + "<br />");
    return powerSet; ❹
}

document.write("<pre>The power set of ABC:<br />")
document.write(getPowerSet("ABC") + "<br /></pre>");
</script>
```

Результат запуска этих программ выглядит следующим образом:

```
The power set of ABC:
In getPowerSet("ABC"), start.
In getPowerSet("ABC") part 1, get sets without head 'A'
.In getPowerSet("BC"), start.
.In getPowerSet("BC") part 1, get sets without head 'B'
..In getPowerSet("C"), start.
..In getPowerSet("C") part 1, get sets without head 'C'
...In getPowerSet(""), start.
...In getPowerSet("") base case returns ['']
..In getPowerSet("C") part 2, get sets with head 'C'
..In getPowerSet("C") New set C
..In getPowerSet("C") returning ['C', '']
.In getPowerSet("BC") part 2, get sets with head 'B'
.In getPowerSet("BC") New set BC
.In getPowerSet("BC") New set B
.In getPowerSet("BC") returning ['BC', 'B', 'C', '']
In getPowerSet("ABC") part 2, get sets with head 'A'
In getPowerSet("ABC") New set ABC
In getPowerSet("ABC") New set AB
In getPowerSet("ABC") New set AC
In getPowerSet("ABC") New set A
In getPowerSet("ABC") returning ['ABC', 'AB', 'AC', 'A', 'BC', 'B', 'C', '']
['ABC', 'AB', 'AC', 'A', 'BC', 'B', 'C', '']
```

Функция `getPowerSet()` принимает единственный аргумент — строку `chars`, содержащую символы, составляющие исходное множество. Базовый случай имеет место, когда аргумент `chars` представляет собой пустую строку ❶, то есть пустое множество. Напомню, что булеан — это совокупность всех подмножеств исходного множества. Единственным подмножеством пустого множества является оно само, поэтому базовый случай возвращает `['']`.

Рекурсивный случай разделен на две части. Первая часть предполагает получение булеана хвостового множества `chars`. Здесь мы воспользуемся принципом «прыжка веры» и просто предположим, что вызов `getPowerSet()` возвращает правильное значение ❷, несмотря на то что мы еще не дописали код этой функции.

Для формирования полного булеана множества `chars` во второй части рекурсивного случая мы создаем новые подмножества, добавляя головной элемент к каждому из подмножеств, составляющих булеан хвостового множества ❸. Вместе с подмножествами, полученными в первой части, эти новые подмножества образуют булеан множества `chars`, возвращаемый функцией в самом конце ❹.

Резюме

Перестановки и сочетания — это две области, к которым многим программистам сложно подступиться. Несмотря на то что применять рекурсию для решения типичных задач программирования нецелесообразно, она хорошо подходит для поиска ответов на более сложные вопросы, описанные в этой главе.

Глава началась с краткого введения в теорию множеств, лежащую в основе структур данных, с которыми работают наши рекурсивные алгоритмы. Множество — это совокупность отдельных элементов. Подмножество состоит из нуля, нескольких или всех элементов множества. Множество не предусматривает никакого порядка для своих элементов, тогда как перестановка представляет собой определенный способ их упорядочения. Сочетание, которое также не предусматривает никакого порядка, является набором, состоящим из нуля, нескольких или всех элементов множества, а k-элементное сочетание — выборкой из множества, состоящей из k элементов.

Перестановки и сочетания могут содержать элемент множества в одном или нескольких экземплярах. Они называются перестановками и сочетаниями без повторений и с повторениями соответственно и реализуются с помощью разных алгоритмов.

В этой главе также была рассмотрена задача генерирования сбалансированных скобок, которая часто затрагивается в ходе собеседования. Наш алгоритм формирует строки из сбалансированных пар скобок, начиная с пустой строки и последовательно добавляя в нее открывающие и закрывающие скобки. Такой подход предполагает возврат к созданным ранее строкам, что делает рекурсию идеальным решением.

Наконец, в этой главе была описана рекурсивная функция для получения булеана множества, то есть совокупности всех возможных k-элементных сочетаний. Рекурсивная функция, которую мы создали, является гораздо более эффективной по сравнению с многократным вызовом функции, генерирующей сочетания всех возможных размеров.

Дополнительные источники информации

Генерация перестановок и сочетаний — это лишь малая часть возможностей, предоставляемых областью математической логики, известной как *теория множеств*. Дополнительную информацию вы можете найти в следующих статьях «Википедии», а также в тех, на которые они ссылаются:

- https://ru.wikipedia.org/wiki/Теория_множеств;
- https://ru.wikipedia.org/wiki/Сочетание;
- https://ru.wikipedia.org/wiki/Перестановка.

Модуль `itertools` стандартной библиотеки Python содержит реализации различных алгоритмов, в том числе для генерации перестановок и сочетаний. Документацию можно найти по адресу https://docs.python.org/3/library/itertools.html.

Кроме того, перестановки и сочетания рассматриваются в рамках курсов по статистике и теории вероятностей. Раздел сайта «Академия Хана», посвященный перестановкам и сочетаниям, можно найти по адресу https://www.khanacademy.org/math/statistics-probability/counting-permutations-and-combinations.

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Предусматривают ли множества определенный порядок элементов? А перестановки? Сочетания?
2. Сколько перестановок (без повторений) существует для множества, состоящего из n элементов?
3. Сколько сочетаний (без повторений) существует для множества, состоящего из n элементов?
4. Является ли $\{A, B, C\}$ подмножеством множества $\{A, B, C\}$?
5. По какой формуле вычисляется *число сочетаний из n по k*, то есть количество возможных комбинаций k элементов, выбранных из множества, состоящего из n элементов?
6. Определите, что из перечисленного ниже считается перестановками, а что — сочетаниями, с повторениями и без них:
   1) *AAA*, *AAB*, *AAC*, *ABA*, *ABB*, *ABC*, *ACA*, *ACB*, *ACC*, *BAA*, *BAB*, *BAC*, *BBA*, *BBB*, *BBC*, *BCA*, *BCB*, *BCC*, *CAA*, *CAB*, *CAC*, *CBA*, *CBB*, *CBC*, *CCA*, *CCB*, *CCC*;
   2) *ABC*, *ACB*, *BAC*, *BCA*, *CAB*;
   3) (нет), *A*, *B*, *C*, *AB*, *AC*, *BC*, *ABC*;
   4) (нет), *A*, *B*, *C*, *AA*, *AB*, *AC*, *BB*, *BC*, *CC*, *AAA*, *AAB*, *AAC*, *ABB*, *ABC*, *ACC*, *BBB*, *BBC*, *BCC*, *CCC*.

7. Нарисуйте древовидный граф, который можно использовать для генерации всех возможных сочетаний элементов множества {*A*, *B*, *C*, *D*}.
8. Ответьте на следующие три вопроса относительно каждого из рекурсивных алгоритмов, представленных в этой главе.

   А. Что представляет собой базовый случай?

   Б. Какой аргумент передается рекурсивной функции при ее вызове?

   В. Как этот аргумент приближается к базовому случаю?

   Затем воссоздайте рекурсивные алгоритмы, описанные в этой главе, не заглядывая в исходный код.

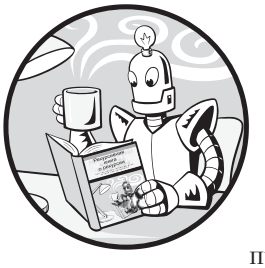
Практика

Решите каждую из следующих задач.

1. Функция генерации перестановок, описанная в этой главе, работает с символами, содержащимися в строке. Измените ее так, чтобы множества были представлены списками (в Python) или массивами (в JavaScript), а в качестве элементов могли использоваться значения любого типа. Например, ваша новая функция должна уметь генерировать перестановки целочисленных, а не строковых значений.
2. Функция генерации сочетаний, описанная в этой главе, работает с символами, содержащимися в строке. Измените ее так, чтобы множества были представлены списками (в Python) или массивами (в JavaScript), а в качестве элементов могли использоваться значения любого типа. Например, ваша новая функция должна уметь генерировать сочетания целочисленных, а не строковых значений.

7

Мемоизация и динамическое программирование



Здесь мы поговорим о мемоизации — технике ускорения работы рекурсивных алгоритмов. Нам предстоит обсудить ее суть, способы применения, а также ее полезность для сфер функционального и динамического программирования. Рассмотрим процесс написания кода для реализации мемоизации на примере алгоритма вычисления последовательности Фибоначчи из главы 2, воспользовавшись функциями стандартной библиотеки Python. Также выясним, почему мемоизация не может быть применена к каждой рекурсивной функции.

Мемоизация

Мемоизация — это метод запоминания значений, возвращаемых функцией при передаче ей конкретных аргументов. Например, если бы кто-нибудь попросил меня найти квадратный корень из 720, то есть число, которое при умножении само на себя дает 720, мне пришлось бы несколько минут просидеть с карандашом и листом бумаги (или вызвать функцию `Math.sqrt(720)` в JavaScript либо `math.sqrt(720)` в Python), чтобы получить значение 26,832815729997478. Если бы несколько секунд спустя меня снова попросили извлечь квадратный корень из 720, я бы не повторял расчеты, а воспользовался уже готовым ответом. Кэшируя ранее вычисленные результаты, мемоизация ускоряет работу алгоритма за счет увеличения объема используемой памяти.

Мемоизацию не стоит путать с *меморизацией*.

Нисходящее динамическое программирование

Мемоизация часто используется в *динамическом программировании*, которое предполагает разбиение большой задачи на перекрывающиеся подзадачи. Такой подход может напоминать уже знакомую нам рекурсию. Ключевое отличие состоит в том,

что в рамках динамического программирования применяются повторяющиеся рекурсивные случаи, соответствующие *перекрывающимся* подзадачам.

Для примера рассмотрим рекурсивный алгоритм вычисления последовательности Фибоначчи из главы 2. Рекурсивный вызов функции `fibonacci(6)` предполагает вызов `fibonacci(5)` и `fibonacci(4)`. Вызов `fibonacci(5)` приводит к вызову `fibonacci(4)` и `fibonacci(3)`. Подзадачи этого алгоритма перекрываются, так как вызов функции `fibonacci(4)` и многие другие вызовы повторяются, что делает генерирование чисел Фибоначчи задачей динамического программирования.

Неэффективность в данном случае заключается в многократном выполнении монотонных, излишних вычислений, потому что вызов `fibonacci(4)` всегда возвращает одно и то же целое число `3`. Вместо этого наша программа могла бы просто запомнить, что при передаче рекурсивной функции аргумента, равного `4`, функция должна немедленно возвратить значение `3`.

На рис. 7.1 показана древовидная диаграмма всех рекурсивных вызовов, включая избыточные, которые можно оптимизировать с помощью мемоизации. А вот быстрая сортировка и сортировка слиянием представляют собой рекурсивные алгоритмы типа «разделяй и властвуй», подзадачи которых не перекрываются и являются уникальными. Поэтому методы динамического программирования к этим алгоритмам сортировки не применяются.

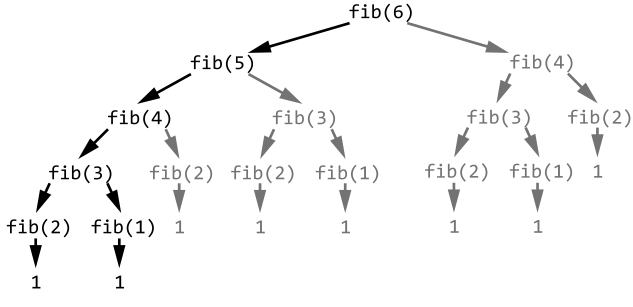


Рис. 7.1. Древовидная диаграмма рекурсивных вызовов функций, начиная с fibonacci(6). Избыточные вызовы выделены серым цветом

Одним из подходов, используемых в динамическом программировании, является мемоизация рекурсивной функции, позволяющая запомнить результаты предыдущих вычислений для будущих вызовов. Если можно повторно применить возвращенные ранее значения, решение перекрывающихся подзадач становится тривиальным.

Использование рекурсии совместно с мемоизацией называется *нисходящим динамическим программированием* или *динамическим программированием «сверху вниз»*. Этот процесс предполагает разбиение большой задачи на более мелкие

перекрывающиеся подзадачи. При противоположном методе, известном как *восходящее динамическое программирование* или *динамическое программирование «снизу вверх»*, построение решения исходной большой задачи начинается с решения меньших подзадач (часто связанных с базовым случаем). Итеративный алгоритм Фибоначчи, который начинается с базовых случаев нахождения первого и второго чисел последовательности, представляет собой пример восходящего динамического программирования. Подходы «снизу вверх» не предполагают использование рекурсивных функций.

Обратите внимание, что нисходящей или восходящей рекурсии не существует. Это часто используемые, но неправильные термины. Любая рекурсия изначально является нисходящей, поэтому термин *«нисходящая рекурсия»* избыточен. И ни один восходящий подход не использует рекурсию, поэтому такого понятия, как восходящая рекурсия, не существует.

Мемоизация в функциональном программировании

Мемоизация применима не ко всем функциям. Чтобы понять почему, нам следует обсудить *функциональное программирование* — парадигму программирования с упором на написание функций, не изменяющих глобальные переменные или какое-либо *внешнее состояние* (например, файлы на жестком диске, интернет-соединения или содержимое баз данных). Некоторые языки программирования, такие как Erlang, Lisp и Haskell, разработаны специально для функционального программирования, принципы и подходы которого могут быть реализованы практически на любом языке, включая Python и JavaScript.

В парадигме функционального программирования существуют понятия детерминированных и недетерминированных функций, побочных эффектов и чистых функций. Функция `sqrt()`, упомянутая в начале главы, — *детерминированная*, поскольку она всегда возвращает одинаковое значение при передаче ей одного и того же аргумента. Однако функция Python `random.randint()`, возвращающая случайное целое число, является *недетерминированной*, поскольку даже при передаче ей одних и тех же аргументов она может возвращать разные значения. Функция `time.time()`, возвращающая текущее время, также относится к числу недетерминированных, поскольку время постоянно движется вперед.

Побочными эффектами называются любые изменения, которые функция вносит во что-либо находящееся за пределами ее собственного кода и локальных переменных. Чтобы проиллюстрировать это, создадим функцию `subtract()`, которая реализует оператор вычитания Python (`-`):

```
>>> def subtract(number1, number2):
...     return number1 - number2
...
>>> subtract(123, 987)
-864
```

Функция `subtract()` не имеет побочных эффектов, так как ее вызов не влияет ни на какую часть программы, находящуюся за пределами ее кода. По состоянию программы или компьютера невозможно определить, сколько раз вызывалась функция `subtract()` — один, два или миллион. Функция может изменять значения локальных переменных внутри самой себя, но эти преобразования остаются изолированными от остальной части программы.

Теперь рассмотрим функцию `addToTotal()`, добавляющую числовой аргумент к значению глобальной переменной с именем `TOTAL`:

```
>>> TOTAL = 0
>>> def addToTotal(amount):
...     global TOTAL
...     TOTAL += amount
...     return TOTAL
...
>>> addToTotal(10)
10
>>> addToTotal(10)
20
>>> TOTAL
20
```

У функции `addToTotal()` есть побочный эффект, поскольку она изменяет элемент, существующий за пределами ее кода, — глобальную переменную `TOTAL`.

Побочной реакцией могут быть не только изменения глобальных переменных. К ним также относятся обновление или удаление файлов, вывод текста на экран, установка соединения с базой данных, аутентификация на сервере или любые другие манипуляции с данными, находящимися за пределами кода функции. Побочным является любой след, который оставляет вызов функции после возврата.

Если функция выступает в качестве детерминированной и не имеет подобных эффектов, она называется *чистой функцией*. Мемоизацию следует применять только к чистым функциям. Вы поймете почему, ознакомившись со следующими разделами, в которых мы применим мемоизацию к рекурсивной функции вычисления последовательности Фибоначчи и к нечистым функциям программы `doNotMemoize`.

Мемоизация рекурсивного алгоритма вычисления последовательности Фибоначчи

Давайте применим мемоизацию к нашей рекурсивной функции вычисления последовательности Фибоначчи из главы 2. Как вы помните, эта функция чрезвычайно неэффективна: на моем компьютере выполнение рекурсивного вызова `fibonacci(40)` занимает 57,8 секунды. Между тем его итеративная версия выполняется настолько быстро, что мой профилировщик кода даже не успевает засечь время и показывает 0,000 секунды.

Мемоизация может значительно ускорить работу рекурсивной версии данной функции. Например, на рис. 7.2 показано количество вызовов исходной и мемоизированной функции `fibonacci()` для нахождения первых 20 чисел Фибоначчи. Как видите, исходная функция выполняет огромное количество ненужных вычислений.

Количество вызовов исходной функции `fibonacci()` растет очень резко (верхний график), чего нельзя сказать о ее мемоизированной версии (нижний график).

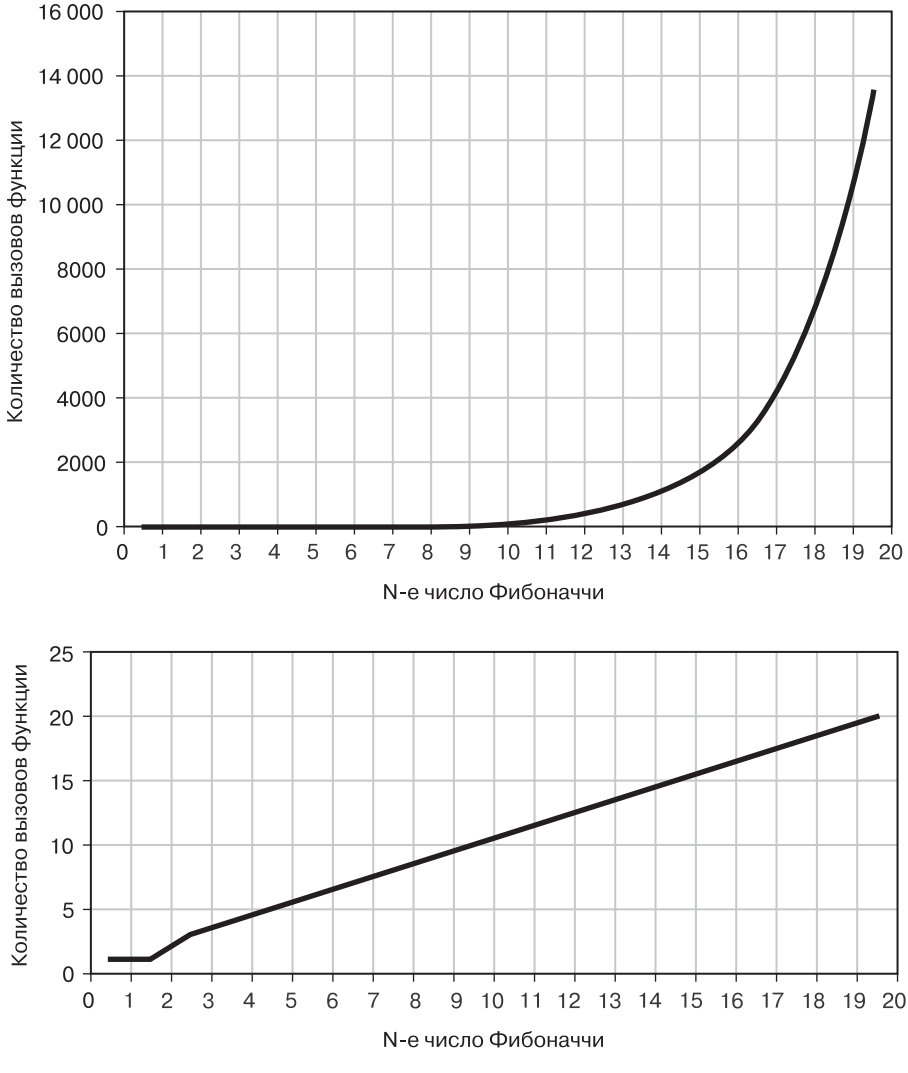


Рис. 7.2. Количество вызовов исходной функции fibonacci() растет очень резко *(верхний график)*, чего нельзя сказать о ее мемоизированной версии *(нижний график)*

Мемоизированная версия алгоритма Фибоначчи на языке Python находится в файле `fibonacciByRecursionMemoized.py`. В ней дополнения к исходной программе `fibonacciByRecursion.py` из главы 2 выделены жирным шрифтом:

```
fibonacciCache = {} ❶ # Создание глобального кэша

def fibonacci(nthNumber, indent=0):
    global fibonacciCache
    indentation = '.' * indent
    print(indentation + 'fibonacci(%s) called.' % (nthNumber))
    if nthNumber in fibonacciCache:
        # Если значение уже кэшировано, возвращаем его
        print(indentation + 'Returning memoized result: %s' %
        (fibonacciCache[nthNumber]))
        return fibonacciCache[nthNumber] ❷

    if nthNumber == 1 or nthNumber == 2:
        # БАЗОВЫЙ СЛУЧАЙ
        print(indentation + 'Base case fibonacci(%s) returning 1.' % (nthNumber))
        fibonacciCache[nthNumber] = 1 ❸ # Обновление кэша
        return 1
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        print(indentation + 'Calling fibonacci(%s) (nthNumber - 1).' % (nthNumber - 1))
        result = fibonacci(nthNumber - 1, indent + 1)

        print(indentation + 'Calling fibonacci(%s) (nthNumber - 2).' % (nthNumber - 2))
        result = result + fibonacci(nthNumber - 2, indent + 1)

        print('Call to fibonacci(%s) returning %s.' % (nthNumber, result))
        fibonacciCache[nthNumber] = result ❹ # Обновление кэша
        return result

print(fibonacci(10))
print(fibonacci(10)) ❺
```

Мемоизированная версия алгоритма Фибоначчи на языке JavaScript находится в файле `fibonacciByRecursionMemoized.html`. В ней дополнения к исходной программе `fibonacciByRecursion.html` из главы 2 выделены жирным шрифтом:

```
<script type="text/javascript">

❶ let fibonacciCache = {}; // Создание глобального кэша

function fibonacci(nthNumber, indent) {
    if (indent === undefined) {
        indent = 0;
    }
    let indentation = '.'.repeat(indent);
    document.write(indentation + "fibonacci(" + nthNumber + ") called.<br />");
```

```
    if (nthNumber in fibonacciCache) {
        // Если значение уже кэшировано, возвращаем его
        document.write(indentation +
        "Returning memoized result: " + fibonacciCache[nthNumber] + "<br />");
    ❷ return fibonacciCache[nthNumber];
    }

    if (nthNumber === 1 || nthNumber === 2) {
        // БАЗОВЫЙ СЛУЧАЙ
        document.write(indentation +
        "Base case fibonacci(" + nthNumber + ") returning 1.<br />");
    ❸ fibonacciCache[nthNumber] = 1; // Обновление кэша
        return 1;    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        document.write(indentation +
        "Calling fibonacci(" + (nthNumber - 1) + ") (nthNumber - 1).<br />");
        let result = fibonacci(nthNumber - 1, indent + 1);

        document.write(indentation +
        "Calling fibonacci(" + (nthNumber - 2) + ") (nthNumber - 2).<br />");
        result = result + fibonacci(nthNumber - 2, indent + 1);

        document.write(indentation + "Returning " + result + ".<br />");
    ❹ fibonacciCache[nthNumber] = result; // Обновление кэша
        return result;
    }
}

document.write("<pre>");
document.write(fibonacci(10) + "<br />");
❺ document.write(fibonacci(10) + "<br />");
document.write("</pre>");
</script>
```

Если вы сравните вывод текущей программы с выводом ее исходной рекурсивной версии из главы 2, то обнаружите, что он намного короче. Это говорит о значительном сокращении количества вычислений, необходимых для получения тех же результатов:

```
fibonacci(10) called.
Calling fibonacci(9) (nthNumber - 1).
.fibonacci(9) called.
.Calling fibonacci(8) (nthNumber - 1).
..fibonacci(8) called.
..Calling fibonacci(7) (nthNumber - 1).
--пропущенный фрагмент—
.......Calling fibonacci(2) (nthNumber - 1).
........fibonacci(2) called.
........Base case fibonacci(2) returning 1.
```

```
.......Calling fibonacci(1) (nthNumber - 2).
........fibonacci(1) called.
........Base case fibonacci(1) returning 1.
Call to fibonacci(3) returning 2.
......Calling fibonacci(2) (nthNumber - 2).
.......fibonacci(2) called.
.......Returning memoized result: 1
--пропущенный фрагмент–
Calling fibonacci(8) (nthNumber - 2).
.fibonacci(8) called.
.Returning memoized result: 21
Call to fibonacci(10) returning 55.
55
fibonacci(10) called.
Returning memoized result: 55
55
```

Чтобы применить мемоизацию к этой функции, создадим словарь (в Python) или объект (в JavaScript) в глобальной переменной с именем `fibonacciCache` ❶. Его ключами будут аргументы, передаваемые для параметра `nthNumber`, а значениями — целые числа, возвращаемые функцией `fibonacci()` при передаче ей данных аргументов. В ходе каждого вызова функция сначала проверяет наличие переданного ей аргумента `nthNumber` в кэше. Если он там присутствует, функция возвращает соответствующее кэшированное значение ❷. В противном случае функция работает как обычно (но непосредственно перед возвратом добавляет полученный результат в кэш ❸ ❹).

Мемоизированная функция, по сути, увеличивает количество базовых случаев в алгоритме Фибоначчи. Исходные базовые случаи относятся только к нахождению первого и второго чисел последовательности и немедленно возвращают значение `1`. Однако каждый раз, когда рекурсивный случай возвращает целое число, он становится базовым случаем для всех будущих вызовов `fibonacci()` с использованием этого аргумента. Его результат уже находится в переменной `fibonacciCache` и может быть незамедлительно возвращен. Если вы когда-либо вызывали функцию `fibonacci(99)`, данный вызов становится таким же базовым, как `fibonacci(1)` и `fibonacci(2)`. Другими словами, мемоизация повышает производительность рекурсивных функций с перекрывающимися подзадачами за счет увеличения числа базовых случаев. Обратите внимание, что, когда наша программа пытается найти десятое число Фибоначчи во второй раз ❺, она незамедлительно возвращает уже сохраненный результат `55`.

Имейте в виду: хоть мемоизация и может уменьшить количество избыточных вызовов в рекурсивном алгоритме, она не обязательно сдерживает рост числа кадров в стеке вызовов. Мемоизация не способна предотвратить переполнение стека, поэтому в некоторых случаях стоит отказаться от рекурсивного алгоритма в пользу более простой итеративной версии.

Модуль Python functools

Реализовать кэш при помощи добавления глобальной переменной и кода для управления ею в код каждой мемоизируемой функции — довольно рутинная задача. Стандартная библиотека Python предусматривает модуль `functools` с декоратором функций `@lru_cache()`, который автоматически мемоизирует декорируемую им функцию. В синтаксисе Python это означает добавление фрагмента `@lru_cache()` в строку, предшествующую инструкции `def`.

Кэш обычно имеет ограничения на размер используемой памяти. Фрагмент *lru* в имени декоратора расшифровывается как *least recently used* и означает, что при заполнении кэша до предела новые записи заменяют собой в первую очередь те значения, которые дольше всего не запрашивались. Алгоритм LRU является простым и быстрым, хотя для различных программных требований доступны и другие алгоритмы кэширования.

Программа `fibonacciFunctools.py` демонстрирует использование декоратора `@lru _cache()`. Дополнения к исходной программе `fibonacciByRecursion.py` из главы 2 выделены жирным шрифтом:

```
import functools

@functools.lru_cache()
def fibonacci(nthNumber):
    print('fibonacci(%s) called.' % (nthNumber))
    if nthNumber == 1 or nthNumber == 2:
        # БАЗОВЫЙ СЛУЧАЙ
        print('Call to fibonacci(%s) returning 1.' % (nthNumber))
        return 1
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        print('Calling fibonacci(%s) (nthNumber - 1).' % (nthNumber - 1))
        result = fibonacci(nthNumber - 1)

        print('Calling fibonacci(%s) (nthNumber - 2).' % (nthNumber - 2))
        result = result + fibonacci(nthNumber - 2)

        print('Call to fibonacci(%s) returning %s.' % (nthNumber, result))
        return result

print(fibonacci(99))
```

По сравнению с внесением дополнений, необходимых для реализации нашего собственного кэша в программе `fibonacciByRecursionMemoized.py`, наиболее простым подходом будет использовать декоратор Python `@lru_cache()`. Нахождение числа `Fibonacci(99)` с помощью обычного рекурсивного алгоритма заняло бы несколько столетий. Но благодаря мемоизации наша программа отображает результат `218922995834555169026` в мгновение ока.

Мемоизация — это полезная техника для рекурсивных функций с перекрывающимися подзадачами, однако ее можно применить к любой чистой функции для ускорения ее работы за счет использования дополнительного объема памяти компьютера.

Что происходит при мемоизации нечистых функций

Декоратор `@lru_cache` не следует добавлять к функциям, которые имеют побочные эффекты или не являются детерминированными, то есть к нечистым. Мемоизация экономит время, пропуская часть кода и возвращая кэшированное ранее значение. Это нормально в случаях с чистыми функциями, а с нечистыми может вызывать различные ошибки.

Если применять мемоизацию к недетерминированной функции, например возвращающей текущее время, это может привести к возврату неверных результатов. В случае с функциями, имеющими побочные эффекты, вроде вывода текста на экран, мемоизация заставляет функцию пропускать предусматриваемый ею побочный эффект. Программа `doNotMemoize.py` демонстрирует последствия применения декоратора функции `@lru_cache` (описанного в предыдущем разделе) для мемоизации этих нечистых функций:

```
import functools, time, datetime

@functools.lru_cache()
def getCurrentTime():
    # Эта недетерминированная функция возвращает разные значения
    # после каждого вызова
    return datetime.datetime.now()

@functools.lru_cache()
def printMessage():
    # Эта функция имеет побочный эффект в виде вывода текста на экран
    print('Hello, world!')

print('Getting the current time twice:')
print(getCurrentTime())
print('Waiting two seconds...')
time.sleep(2)
print(getCurrentTime())

print()

print('Displaying a message twice:')
printMessage()
printMessage()
```

Результат запуска этой программы выглядит следующим образом:

```
Getting the current time twice:
2022-07-30 16:25:52.136999
Waiting two seconds...
2022-07-30 16:25:52.136999

Displaying a message twice:
Hello, world!
```

Обратите внимание, что второй вызов функции `getCurrentTime()` возвращает тот же результат, что и первый, несмотря на то что он был выполнен две секунды спустя. А из двух вызовов `printMessage()` только первый приводит к отображению на экране сообщения `Hello, world!`.

Эти ошибки трудно заметить, потому что они не вызывают сбоя программы, а лишь искажают поведение функций. Вне зависимости от выбранного способа мемоизации функций не забывайте тщательно их тестировать.

Резюме

Мемоизация (не путать с меморизацией) — это метод оптимизации, позволяющий ускорить работу рекурсивных алгоритмов с перекрывающимися подзадачами за счет запоминания результатов предыдущих идентичных вычислений. Мемоизация часто используется в области динамического программирования. Повышая скорость работы за счет дополнительной компьютерной памяти, мемоизация делает возможным применение некоторых рекурсивных функций, которые сами по себе работают слишком долго.

Тем не менее мемоизация не предотвращает ошибок, связанных с переполнением стека. Имейте в виду, что мемоизация не является альтернативой для простого итеративного решения. Код, использующий рекурсию ради нее самой, автоматически не становится более элегантным по сравнению с нерекурсивным.

Мемоизацию следует применять только к чистым функциям, то есть к тем, которые являются детерминированными (возвращают одинаковые значения при получении одних и тех же аргументов) и не имеют побочных эффектов (то есть не влияют на то, что находится за пределами кода самой функции). Чистые функции часто используются в функциональном программировании, которое представляет собой парадигму программирования, предполагающую интенсивное применение рекурсии.

Мемоизация реализуется путем создания для каждой мемоизируемой функции специальной структуры данных под названием *«кэш»*. Вы можете написать соответствующий код самостоятельно или воспользоваться встроенным в Python декоратором `@functools.lru_cache()`: он автоматически мемоизирует функцию, к которой применяется.

Дополнительные источники информации

Алгоритмы динамического программирования не ограничиваются мемоизацией. К этим алгоритмам часто обращаются как на собеседованиях при принятии на работу, так и в ходе соревнований по программированию. На сайте Coursera есть бесплатный курс *Dynamic Programming, Greedy Algorithms* https://www.coursera.org/learn/dynamic-programming-greedy-algorithms. Организация freeCodeCamp опубликовала серию статей, посвященных динамическому программированию, с которыми вы можете ознакомиться по адресу https://www.freecodecamp.org/news/learn-dynamic-programing-to-solve-coding-challenges.

Если вы хотите больше узнать об алгоритме LRU и других функциях, связанных с кэшем, то можете обратиться к официальной документации по модулю Python `functools`, доступной по адресу https://docs.python.org/3/library/functools.html. Дополнительную информацию о других алгоритмах кэширования можно найти в статье «Википедии»: https://ru.wikipedia.org/wiki/Алгоритмы_кэширования.

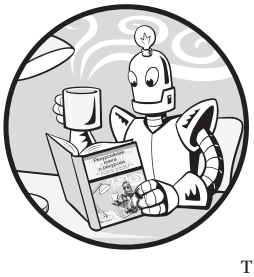
Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое мемоизация?
2. Чем задачи динамического программирования отличаются от обычных рекурсивных задач?
3. На что делается упор в функциональном программировании?
4. Какими двумя характеристиками должна обладать чистая функция?
5. Является ли функция, возвращающая текущие дату и время, детерминированной?
6. Каким образом мемоизация улучшает производительность рекурсивных функций с перекрывающимися подзадачами?
7. Может ли добавление декоратора `@lru_cache()` улучшить производительность функции сортировки слиянием? Если да, то как? Если нет, то почему?
8. Является ли изменение значения локальной переменной функции побочным эффектом?
9. Предотвращает ли мемоизация переполнение стека?

8

Оптимизация хвостовых вызовов



В предыдущей главе мы говорили об использовании мемоизации для оптимизации рекурсивных функций. А в этой мы обсудим метод, называемый *оптимизацией хвостовых вызовов*, который используется компилятором или интерпретатором для предотвращения переполнения стека. Оптимизацию хвостовых вызовов также называют *устранением хвостовых вызовов* или *устранением хвостовой рекурсии*.

Текущая глава объясняет оптимизацию хвостовых вызовов, но не пропагандирует ее. Я бы даже рекомендовал *никогда* не использовать данный метод. Как вы увидите далее, изменение кода функции для применения оптимизации хвостовых вызовов зачастую делает его гораздо менее понятным. Такой метод следует рассматривать скорее как обходной путь, заставляющий рекурсивный алгоритм работать, когда его вообще не следовало бы использовать. Помните, что сложное рекурсивное решение не обязательно самое элегантное: простые задачи программирования следует решать с помощью несложных нерекурсивных методов.

Многие реализации популярных языков программирования даже не предусматривают оптимизации хвостовых вызовов. К ним относятся интерпретаторы и компиляторы для Python, JavaScript и Java. Однако вам все равно следует ознакомиться с данным методом на случай, если вы столкнетесь с ним в проектах, над которыми работаете.

Принцип работы хвостовой рекурсии и оптимизации хвостовых вызовов

Чтобы воспользоваться оптимизацией хвостовых вызовов, функция должна предусматривать *хвостовую рекурсию*, при которой рекурсивный вызов является последней операцией перед возвратом из функции. В коде это имеет вид оператора `return`, возвращающего результаты рекурсивного вызова.

Вспомните программы `factorialByRecursion.py` и `factorialByRecursion.html` из главы 2. Они вычисляли факториал целого числа: например, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Приведенные расчеты могут выполняться рекурсивно, поскольку выражение `factorial(n)` эквивалентно `n * factorial(n — 1)`, а в базовом случае `n == 1` возвращается `1`.

Перепишем эти программы так, чтобы они использовали хвостовую рекурсию. В программе из файла `factorialTailCall.py` содержится функция `factorial()`, использующая хвостовую рекурсию:

```
def factorial(number, accum=1):
    if number == 1:
        # БАЗОВЫЙ СЛУЧАЙ
        return accum
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return factorial(number - 1, accum * number)

print(factorial(5))
```

Эквивалентная программа на языке JavaScript содержится в файле `factorialTailCall.html`:

```
<script type="text/javascript">
function factorial(number, accum=1) {
    if (number === 1) {
        // БАЗОВЫЙ СЛУЧАЙ
        return accum;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return factorial(number - 1, accum * number);
    }
}

document.write(factorial(5));
</script>
```

Обратите внимание, что рекурсивный случай функции `factorial()` завершается оператором `return`, возвращающим результаты рекурсивного вызова данной функции. Чтобы интерпретатор или компилятор смог оптимизировать хвостовой вызов, последней операцией рекурсивной функции должен быть возврат результатов рекурсивного вызова. Между рекурсивным вызовом и оператором `return` не должно находиться никаких инструкций. Базовый случай возвращает параметр `accum`, который представляет собой так называемый аккумулятор, описанный в следующем разделе.

Чтобы понять принцип оптимизации хвостовых вызовов, вспомните, что происходит при вызове функции (см. главу 1). Сначала создается кадр, который сохраняется

в стеке вызовов. Если внутри функции вызывается другая функция, создается еще один кадр, который помещается поверх первого в стеке вызовов. После возврата из функции программа автоматически удаляет кадры с вершины стека.

Стек переполняется, когда слишком много вызовов функций выполняется без возврата, в результате чего количество кадров превышает емкость стека. Для программ на языке Python она составляет 1000 вызовов функций, а для программ на языке JavaScript — около 10 000. Для обычных приложений такой емкости более чем достаточно, но рекурсивные алгоритмы могут с легкостью превысить этот предел и вызвать переполнение стека, приводящее к сбою программы.

Как говорилось в главе 2, в кадре хранятся локальные переменные и адрес возврата, обозначающий точку, с которой должно продолжиться выполнение программы после завершения работы функции. Однако, если последней операцией в рекурсивном случае является возврат результатов рекурсивного вызова, в сохранении локальных переменных нет необходимости. Функция ничего не делает с локальными переменными после рекурсивного вызова, поэтому текущий кадр можно сразу же удалить. Адрес возврата в следующем кадре может совпадать с адресом возврата, хранившимся в удаленном кадре.

Поскольку текущий кадр удаляется, а не сохраняется, стек вызовов не растет, что исключает вероятность его переполнения.

Как говорилось в главе 1, все рекурсивные алгоритмы допускается реализовывать с помощью стека и цикла. Поскольку оптимизация хвостовых вызовов устраняет потребность в стеке вызовов, мы, по сути, используем рекурсию для имитации итеративного кода цикла. Однако ранее в книге я говорил, что рекурсия больше всего подходит для решения задач, предусматривающих использование древовидных структур данных и поиск с возвратом. Без стека вызовов никакая хвостовая рекурсивная функция не могла бы осуществить поиск с возвратом. На мой взгляд, любой алгоритм оказался бы более простым и удобочитаемым, будучи реализованным с помощью цикла, а не хвостовой рекурсии. Использование рекурсии ради нее самой не делает код более элегантным.

Аккумуляторы в контексте хвостовой рекурсии

Недостаток хвостовой рекурсии состоит в том, что она требует реорганизации рекурсивной функции таким образом, чтобы последней операцией был возврат значения, возвращаемого рекурсивным вызовом. Это может еще сильнее затруднить восприятие кода. Действительно, функция `factorial()` в программах `factorialTailCall.py` и `factorialTailCall.html` менее понятна по сравнению с ее версией в `factorialByRecursion.py` и `factorialByRecursion.html` из главы 2.

В случае хвостового вызова `factorial()` новый параметр с именем `accum` отслеживает вычисляемое произведение по мере выполнения рекурсивных вызовов. Этот параметр, называемый *аккумулятором*, отслеживает промежуточный результат вычисления, который в противном случае сохранялся бы в локальной переменной. Не все хвостовые рекурсивные функции используют аккумуляторы, и тем не менее именно они позволяют обойти проблему хвостовой рекурсии, связанную с невозможностью применения локальных переменных после последнего рекурсивного вызова. Обратите внимание, что в функции `factorial()` в программе `factorialByRecursion.py` рекурсивный случай выглядел так: `return number * factorial(number - 1)`. То есть умножение выполнялось после завершения рекурсивного вызова `factorial(number - 1)`. Аккумулятор `accum` заменяет локальную переменную `number`.

Обратите также внимание, что базовый случай для `factorial()` теперь возвращает не `1`, а значение `accum`. К моменту вызова `factorial()` с `number == 1` и достижения базового случая параметр `accum` содержит окончательный результат, подлежащий возврату. Модификация кода для использования оптимизации хвостовых вызовов часто предполагает изменение базового случая для возврата значения аккумулятора.

Вы можете представить вызов функции `factorial(5)` как процесс преобразования в возвращаемое значение `return`, изображенный на рис. 8.1.

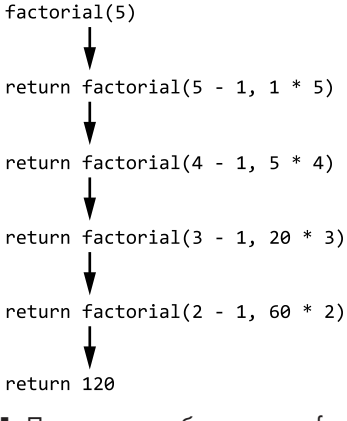


Рис. 8.1. Процесс преобразования factorial(5) в целое число 120

Использование рекурсивных вызовов в качестве последней операции функции и добавление аккумуляторов, скорее всего, сделает код еще более сложным для восприятия по сравнению с обычным рекурсивным. Но это не единственный недостаток хвостовой рекурсии, как мы увидим в следующем разделе.

Ограничения хвостовой рекурсии

Код хвостовых рекурсивных функций необходимо реорганизовать, чтобы сделать его пригодным для оптимизации хвостовых вызовов компилятора или интерпретатора. Однако не все компиляторы и интерпретаторы предусматривают такую функцию. Например, CPython (интерпретатор Python, загружаемый с сайта https://python.org) не реализует оптимизацию хвостовых вызовов. Даже если вы пишете свои рекурсивные функции так, что рекурсивный вызов становится их последней операцией, это не убережет вас от переполнения стека, которое неминуемо произойдет после выполнения достаточно большого количества вызовов.

Кроме того, оптимизация хвостовых вызовов, скорее всего, никогда не будет реализована в CPython. Гвидо ван Россум, создатель языка Python, объяснил, что оптимизация хвостовых вызовов может усложнить отладку программ, так как удаляет кадры из стека вызовов вместе с содержащейся в них отладочной информацией. Он также отмечает, что как только оптимизация хвостовых вызовов будет реализована, Python-программисты начнут писать код, полагаясь на эту функцию, и их программы не будут работать с интерпретаторами, не реализующими оптимизацию хвостовых вызовов.

Наконец, ван Россум справедливо заметил, что рекурсия не является неотъемлемой частью повседневного программирования. Информатики и математики склонны возводить рекурсию на пьедестал. Однако оптимизация хвостовых вызовов — это всего лишь способ обойти проблему переполнения стека и сделать работоспособными некоторые рекурсивные алгоритмы.

То, что CPython не поддерживает оптимизацию хвостовых вызовов, не означает, что этого не делают другие компиляторы или интерпретаторы языка Python. Если оптимизация хвостовых вызовов явно не описана в спецификации языка программирования, то она является не его внутренней функцией, а лишь некоторых из его компиляторов или интерпретаторов.

Отсутствие оптимизации хвостовых вызовов не уникальный случай для языка Python. Компилятор Java также не поддерживает ее, начиная с версии 8. Оптимизация хвостовых вызовов — часть версии JavaScript ECMAScript 6, однако по состоянию на 2022 год ее фактически поддерживает лишь реализация JavaScript в веб-браузере Safari. Чтобы проверить, предусматривает ли компилятор или интерпретатор вашего языка программирования данную оптимизацию, вам стоит написать хвостовую рекурсивную функцию для вычисления факториала и попытаться рассчитать с ее помощью факториал числа 100 000. Если программа даст сбой, значит, оптимизация хвостового вызова не реализована.

Лично я считаю, что хвостовую рекурсию лучше никогда не использовать. Как говорилось в главе 2, любой рекурсивный алгоритм можно реализовать с помощью цикла и стека. Оптимизация хвостовых вызовов предотвращает переполнение стека,

исключая использование стека вызовов. Следовательно, все алгоритмы хвостовой рекурсии могут быть реализованы с помощью одного лишь цикла. Поскольку циклы намного проще, чем рекурсивная функция, их следует использовать везде, где может возникнуть необходимость в оптимизации хвостовых вызовов.

Кроме того, проблемы возникают даже в том случае, если оптимизация хвостовых вызовов реализована. Поскольку хвостовая рекурсия возможна, лишь когда последним действием функции является возврат значения, возвращаемого рекурсивным вызовом, у нас не получится реализовать хвостовую рекурсию для алгоритмов, требующих выполнения двух и более рекурсивных вызовов. Например, несмотря на то, что оптимизация хвостовых вызовов в `fibonacci(n – 1)` и `fibonacci(n – 2)` алгоритма Фибоначчи может быть выполнена для последнего рекурсивного вызова, первый рискует привести к переполнению стека в случае передачи достаточно больших аргументов.

Примеры использования хвостовой рекурсии

Давайте рассмотрим некоторые из ранее показанных рекурсивных функций, чтобы выяснить, подходят ли они для использования хвостовой рекурсии. Имейте в виду, что, поскольку Python и JavaScript не осуществляют оптимизацию хвостовых вызовов, описанные далее хвостовые рекурсивные функции все равно будут вызывать ошибку переполнения стека. Эти примеры носят исключительно демонстрационный характер.

Обращение строки

Первым примером является программа для обращения строки, которую мы создали в главе 3. Код этой хвостовой рекурсивной функции на языке Python находится в файле `reverseStringTailCall.py`:

```
❶ def rev(theString, accum=''):
      if len(theString) == 0:
          # БАЗОВЫЙ СЛУЧАЙ
        ❷ return accum
      else:
          # РЕКУРСИВНЫЙ СЛУЧАЙ
          head = theString[0]
          tail = theString[1:]
        ❸ return rev(tail, head + accum)

text = 'abcdef'
print('The reverse of ' + text + ' is ' + rev(text))
```

Аналогичная программа на языке JavaScript — в файле `reverseStringTailCall.html`:

```
<script type="text/javascript">
❶ function rev(theString, accum='') {
```

```
    if (theString.length === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
     ❷ return accum;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        let head = theString[0];
        let tail = theString.substring(1, theString.length);
     ❸ return rev(tail, head + accum);
    }
}

let text = "abcdef";
document.write("The reverse of " + text + " is " + rev(text) + "<br />");
</script>
```

Преобразование исходных рекурсивных функций в программах `reverseString.py` и `reverseString.html` предполагает добавление аккумулятора. Если для этого параметра с именем `accum` не передается конкретное значение, то он по умолчанию представляет собой пустую строку ❶. Мы также изменяем базовый случай с `return ''` на `return accum` ❷, а рекурсивный случай — с `return rev(tail) + head` (который выполняет конкатенацию строк после возврата из рекурсивного вызова) на `return rev(tail, head + accum)` ❸. Вы можете представить вызов функции `rev('abcdef')` как процесс преобразования, изображенный на рис. 8.2.

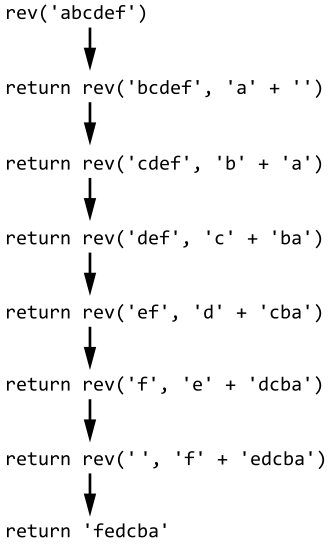


Рис. 8.2. Процесс преобразования rev('abcdef') в строку fedcba

Используя аккумулятор в качестве общей локальной переменной для всех вызовов функции, можно превратить `rev()` в хвостовую рекурсивную функцию.

Нахождение подстроки

Некоторые рекурсивные функции естественным образом заканчиваются использованием шаблона хвостовой рекурсии. Если вы посмотрите на функцию `findSubstringRecursive()` в программах `findSubstring.py` и `findSubstring.html` в главе 2, то заметите, что последним действием в рекурсивном случае является возврат значения, возвращаемого этим рекурсивным вызовом. Для того чтобы превратить ее в хвостовую рекурсивную функцию, никаких корректировок не требуется.

Вычисление экспоненты

Программы `exponentByRecursion.py` и `exponentByRecursion.html` из главы 2 не являются подходящими кандидатами для применения хвостовой рекурсии. Они предусматривают два рекурсивных случая для четного и нечетного значения параметра `n`. Это не проблема, если последним действием всех рекурсивных случаев будет возврат значения, возвращаемого рекурсивным вызовом.

Однако обратите внимание на код для рекурсивного случая `n is even` в программе на языке Python:

```
-- пропущенный фрагмент --
    elif n % 2 == 0:
        # РЕКУРСИВНЫЙ СЛУЧАЙ (когда n четное)
        result = exponentByRecursion(a, n / 2)
        return result * result
-- пропущенный фрагмент --
```

И на тот же код на языке JavaScript:

```
-- пропущенный фрагмент --
  } else if (n % 2 === 0) {
        // РЕКУРСИВНЫЙ СЛУЧАЙ (когда n четное)
        result = exponentByRecursion(a, n / 2);
        return result * result;
-- пропущенный фрагмент --
```

В рамках данного рекурсивного случая рекурсивный вызов не последнее действие. Можно было бы избавиться от локальной переменной `result` и дважды вызвать рекурсивную функцию, уменьшив рекурсивный случай до следующего фрагмента кода:

```
-- пропущенный фрагмент --
return exponentByRecursion(a, n / 2) * exponentByRecursion(a, n / 2)
-- пропущенный фрагмент --
```

Однако в итоге мы получили бы два рекурсивных вызова `exponentByRecursion()`, что не только без всякой нужды удвоило бы объем вычислений, выполняемых алгоритмом, но и сделало бы последней операцией функции умножение двух

возвращаемых значений. Та же проблема свойственна рекурсивному алгоритму Фибоначчи: если рекурсивная функция предусматривает несколько рекурсивных вызовов, то по крайней мере один из них не может быть последним действием, выполняемым функцией.

Определение четности числа

Чтобы определить, является ли целое число четным или нечетным, используйте оператор модуля `%`. Выражение `number % 2 == 0` возвращает `True`, если число `number` четное, и `False`, если нечетное. Однако если вы хотите создать более «элегантный» рекурсивный алгоритм, то лучше реализовать функцию `isOdd()`, содержащуюся в файле `isOdd.py` (остальной код данной программы представлен далее в текущем подразделе):

```
def isOdd(number):
    if number == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return False
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return not isOdd(number - 1)
print(isOdd(42))
print(isOdd(99))
-- пропущенный фрагмент --
```

Эквивалентная функция на языке JavaScript находится в файле `isOdd.html`:

```
<script type="text/javascript">

function isOdd(number) {
    if (number === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
        return false;
    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return !isOdd(number - 1);
    }
}
document.write(isOdd(42) + "<br />");
document.write(isOdd(99) + "<br />");
-- пропущенный фрагмент --
```

Функция `isOdd()` предусматривает два базовых случая. Когда значение аргумента `number` равно `0`, функция возвращает `False`, а это говорит о том, что число *четное*. Для простоты мы реализовали работу `isOdd()` только с положительными целыми числами. Рекурсивный случай возвращает значение, противоположное `isOdd(number - 1)`.

Чтобы понять, как это работает, рассмотрим пример: вызываемая функция `isOdd(42)` не может определить четность числа `42`, но знает, что ответ противоположен ответу

на вопрос, является ли четным или нечетным число `41`. Данная функция вернет `not isOdd(41)`. Такой вызов функции, в свою очередь, вернет логическое значение, противоположное `isOdd(40)`, и т. д., пока `isOdd(0)` не вернет `False`. Число рекурсивных вызовов определяет количество операторов `not`, которые будут применены к возвращаемым значениям до возврата итогового значения.

Однако передача данной рекурсивной функции большого аргумента приводит к переполнению стека. Вызов `isOdd(100000)` предполагает 100 001 вызов функции без возврата, что намного превышает емкость любого стека вызовов. Мы в состоянии изменить код функции так, чтобы последней операцией рекурсивного случая был возврат результата рекурсивного вызова, превратив ее в хвостовую рекурсивную функцию. Результатом является функция `isOddTailCall()` в файле `isOdd.py`, остальная часть кода которой выглядит так:

```
-- пропущенный фрагмент --
def isOddTailCall(number, inversionAccum=False):
    if number == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return inversionAccum
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        return isOddTailCall(number - 1, not inversionAccum)

print(isOddTailCall(42))
print(isOddTailCall(99))
```

Эквивалентный код на языке JavaScript находится в файле `isOdd.html`:

```
-- пропущенный фрагмент --
function isOddTailCall(number, inversionAccum) {
    if (inversionAccum === undefined) {
        inversionAccum = false;
    }

    if (number === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
        return inversionAccum;
    } else {        // РЕКУРСИВНЫЙ СЛУЧАЙ
        return isOddTailCall(number - 1, !inversionAccum);
    }
}

document.write(isOdd(42) + "<br />");
document.write(isOdd(99) + "<br />");
</script>
```

Если вышеупомянутый код на языках Python и JavaScript выполняется интерпретатором, поддерживающим оптимизацию хвостовых вызовов, то вызов

`isOddTailCall(100000)` не приведет к переполнению стека. Тем не менее оптимизация хвостовых вызовов гораздо медленнее для определения четности числа, чем оператор модуля `%`.

Если вы думаете, что обычная или хвостовая рекурсия представляет собой крайне неэффективный способ определения четности положительного целого числа, вы абсолютно правы. В отличие от итеративных решений рекурсивный подход может привести к аварийному завершению работы программы из-за переполнения стека. Применение оптимизации хвостовых вызовов не решает проблем эффективности, связанных с ненадлежащим использованием рекурсии. Сама по себе рекурсия не является более предпочтительной техникой по сравнению с итеративными подходами. И хвостовая рекурсия всегда проигрывает циклу или любому другому простому решению.

Резюме

Оптимизация хвостовых вызовов — это функция компилятора или интерпретатора языка программирования, которую можно применять к хвостовым рекурсивным функциям. В рекурсивном случае для таких функций последней операцией является возврат возвращаемого значения рекурсивного вызова. Это позволяет функции удалить текущий кадр и предотвратить переполнение стека вызовов при выполнении новых рекурсивных вызовов.

Хвостовая рекурсия — это обходной путь, позволяющий некоторым рекурсивным алгоритмам работать без сбоев с большими аргументами. Однако данный подход требует реорганизации кода и, возможно, добавления параметра-аккумулятора, что делает код менее удобочитаемым. Скорее всего, рано или поздно вы сочтете, что использовать рекурсивный алгоритм вместо итеративного не имеет смысла.

Дополнительные источники информации

Подробное обсуждение хвостовой рекурсии вы можете найти на сайте Stack Overflow по ссылке https://stackoverflow.com/questions/33923/what-is-tail-recursion.

Ван Россум (Van Rossum) о своем решении не использовать хвостовую рекурсию написал два поста: https://neopythonic.blogspot.com.au/2009/04/tail-recursion-elimination.html и https://neopythonic.blogspot.com.au/2009/04/final-words-on-tail-calls.html.

Стандартная библиотека Python содержит модуль под названием `inspect`, который позволяет просматривать кадры в стеке вызовов во время выполнения программы на языке Python. Официальную документацию можно найти по адресу https://docs.python.org/3/library/inspect.html, а руководство по его использованию — в блоге Дага Хеллмана (Doug Hellmann) *Python 3 Module of the Week*: https://pymotw.com/3/inspect.

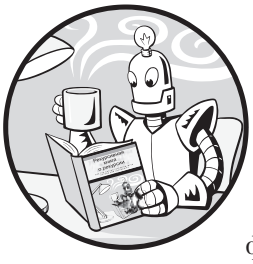
Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Чем полезна оптимизация хвостовых вызовов?
2. Каким должно быть последнее действие рекурсивной функции, чтобы она могла считаться хвостовой?
3. Все ли компиляторы и интерпретаторы осуществляют оптимизацию хвостовых вызовов?
4. Что такое аккумулятор?
5. В чем недостаток хвостовой рекурсии?
6. Можно ли переписать алгоритм быстрой сортировки (описанный в главе 5) для использования оптимизации хвостовых вызовов?

9

Рисование фракталов



Самое интересное применение рекурсии, безусловно, заключается в рисовании фракталов. *Фракталы* — это формы, которые повторяют сами себя, иногда хаотично, в разных масштабах. Сам термин был введен в 1975 году создателем фрактальной геометрии Бенуа Б. Мандельбротом и происходит от латинского слова *frāctus* — то есть «разбитый» или «треснувший» (как в случае разбитого стекла). К фракталам относится множество естественных и искусственных форм. В природе они встречаются в виде деревьев, листьев папоротника, горных хребтов, молний, береговых линий, речных сетей и снежинок. Математики, программисты и художники могут рисовать подобные сложные геометрические фигуры, опираясь на несколько рекурсивных правил.

Рекурсия позволяет создавать сложные фрактальные изображения, используя совсем небольшое количество строк кода. В текущей главе мы используем встроенный в Python модуль `turtle` для генерации нескольких распространенных фракталов с помощью кода. В случае реализации черепашьей графики с помощью JavaScript лучше воспользоваться библиотекой `jtg` Грега Реймера (Greg Reimer). Для простоты в этой главе представлены только программы для рисования фракталов на языке Python. Однако о `jtg` мы тоже поговорим.

Черепашья графика

Изначально *черепашья графика* была функцией языка Logo, предназначенного для помощи детям в освоении принципов программирования. С тех пор она была воспроизведена во многих языках и платформах. Ее центральной идеей является объект под названием *turtle* («черепаха»).

Эта черепашка действует как программируемое перо, рисующее линии в двумерном окне. Представьте себе настоящую черепаху с пером, которая ползает по земле,

рисуя при этом линию, описывающую ее путь. Она может регулировать размер и цвет пера, а также «не касаться им земли», чтобы не рисовать во время движения. Программы, реализующие подобную графику, способны создавать сложные геометрические рисунки (рис. 9.1).

Если поместить эти инструкции в циклы и функции, то даже небольшие программы смогут создавать впечатляющие фигуры. Рассмотрим следующую программу `Spiral.py`:

```
import turtle
turtle.tracer(1, 0) # Заставляет черепашку рисовать быстрее
for i in range(360):
    turtle.forward(i)
    turtle.left(59)
turtle.exitonclick() # Пауза, длящаяся до щелчка кнопкой мыши в окне программы
```

Когда вы ее запустите, откроется графическое окно и черепашка (обозначенная треугольником) начнет двигаться по спирали, как показано на рис. 9.1. В результате получится весьма красивый рисунок, хоть и не являющийся фракталом.

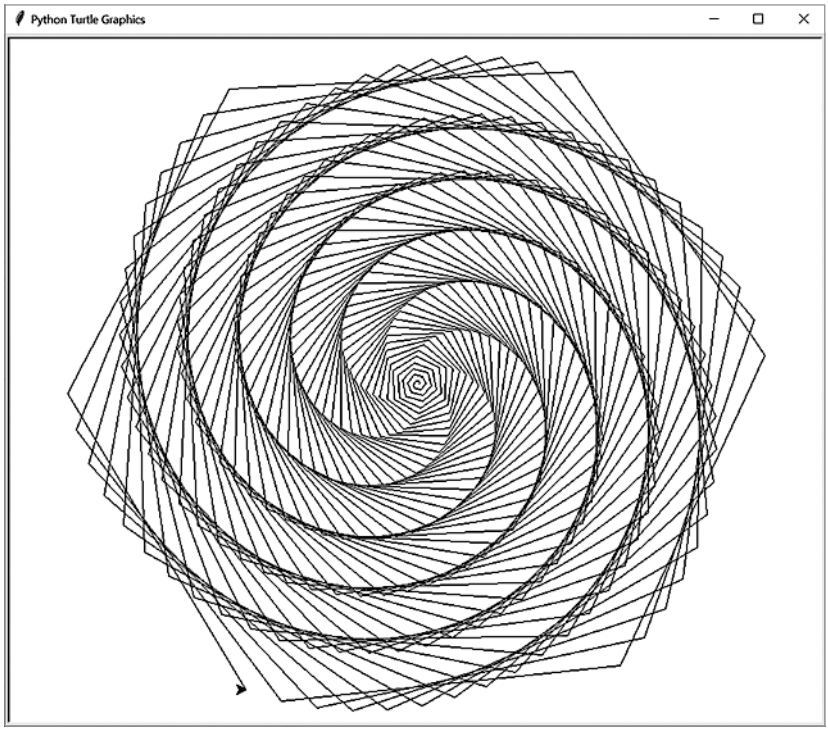


Рис. 9.1. Спираль, нарисованная программой с использованием модуля Python turtle

Окно графической системы использует декартовы координаты x и y. Значение координаты x увеличивается слева направо, а y — снизу вверх. Вместе они задают уникальный адрес любой точки в окне. По умолчанию *начало координат* (точка с координатами $x = 0$ и $y = 0$) находится в центре окна.

У черепашки также есть *курс*, то есть направление движения, обозначаемое числом от 0 до 359 (круг разбит на 360°). В модуле Python `turtle` значение 0 соответствует восточному направлению (к правому краю экрана) и увеличивается против часовой стрелки. Значение 90 соответствует северному направлению, 180 — западному, а 270 — южному. В библиотеке JavaScript `jtg` значение 0 выступает в качестве северного направления и увеличивается по часовой стрелке (рис. 9.2).

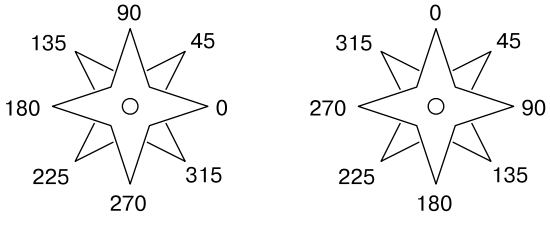


Рис. 9.2. Направления движения черепашки в модуле Python turtle *(слева)* и в библиотеке JavaScript jtg *(справа)*

Перейдите на страницу библиотеки JavaScript `jtg` по адресу https://inventwithpython.com/jtg и введите следующий код в текстовое поле внизу:

```
for (let i = 0; i < 360; i++) { t.fd(i); t.lt(59) }
```

В результате в основной области страницы будет нарисована спираль, как на рис. 9.1.

Основные функции модуля turtle

Наиболее часто используемыми функциями в черепашьей графике являются те, которые заставляют перо менять направление и двигаться вперед или назад. Функции `turtle.left()` и `turtle.right()` позволяют повернуть черепашку на определенное количество градусов, а `turtle.forward()` и `turtle.backward()` — переместить ее с текущей позиции.

В табл. 9.1 перечислены некоторые функции черепашьей графики. Первая (будет начинаться с `turtle.`) относится к Python, а вторая (начинается с `t.`) — к JavaScript. Полная документация по данному модулю Python доступна по адресу https://docs.python.org/3/library/turtle.html. Чтобы получить справку по JavaScript, нажмите клавишу F1 в программе `jtg`.

Таблица 9.1. Функции модуля Python turtle и библиотеки JavaScript jtg

Python	JavaScript	Описание
`goto(x, y)`	`xy(x, y)`	Перемещает черепашку в точку с координатами (x, y)
`setheading(deg)`	`heading(deg)`	Задает курс черепашки. В Python 0° это восточное направление (вправо), в JavaScript 0° — северное направление (вверх)
`forward(steps)`	`fd(steps)`	Перемещает черепашку на несколько шагов вперед в том направлении, в котором она смотрит
`backward(steps)`	`bk(steps)`	Перемещает черепашку на несколько шагов в направлении, противоположном тому, в котором она смотрит
`left(deg)`	`lt(deg)`	Поворачивает черепашку влево
`right(deg)`	`rt(deg)`	Поворачивает черепашку вправо
`penup()`	`pu()`	«Поднимает перо», чтобы черепашка не рисовала во время своего движения
`pendown()`	`pd()`	«Опускает перо», чтобы черепашка рисовала во время своего движения
`pensize(size)`	`thickness(size)`	Изменяет толщину рисуемых линий. По умолчанию это значение равно `1`
`pencolor(color)`	`color(color)`	Изменяет цвет рисуемых линий. Значением может быть строка, содержащая название распространенного цвета, например `red` (красный) или `white` (белый). По умолчанию используется черный цвет (`black`)
`xcor()`	`get.x()`	Возвращает координату x текущей позиции черепашки
`ycor()`	`get.y()`	Возвращает координату y текущей позиции черепашки
`heading()`	`get.heading()`	Возвращает текущий курс черепашки в виде числа с плавающей точкой от 0 до 359. В Python 0° это восточное направление (вправо), в JavaScript 0° — северное направление (вверх)
`reset()`	`reset()`	Удаляет все нарисованные линии и восстанавливает исходное положение и курс черепашки
`clear()`	`clean()`	Удаляет все нарисованные линии, но не перемещает черепашку

Функции, перечисленные в табл. 9.2, доступны только в модуле Python `turtle`.

Таблица 9.2. Функции, доступные только в модуле Python turtle

Python	Описание
`begin_fill()`	Начинает рисовать заполненную цветом фигуру. Линии, нарисованные после вызова этой функции, будут определять периметр данной фигуры
`end_fill()`	Продолжает процесс рисования заполненной цветом фигуры, начатый с помощью вызова `turtle.begin_fill()`
`fillcolor(`*`color`*`)`	Задает цвет для заполнения рисуемых фигур
`hideturtle()`	Скрывает треугольник, выступающий в роли черепашки
`showturtle()`	Отображает треугольник, выступающий в роли черепашки
`tracer(`*`drawingUpdates, delay`*`)`	Регулирует скорость рисования. Значение `0` параметра `delay` означает задержку 0 миллисекунд после рисования каждой линии. Чем выше значение параметра `drawingUpdates`, обозначающего количество линий, нарисованных перед обновлением экрана, тем быстрее происходит процесс рисования
`update()`	Выводит на экран все буферизованные линии (см. далее). Эту функцию следует вызывать после того, как черепашка закончит рисунок
`setworldcoordinates (`*`llx, lly, urx, ury`*`)`	Определяет показываемую в окне часть координатной плоскости. Первыми двумя аргументами являются координаты *x* и *y* левого нижнего угла окна. А последними двумя — координаты *x* и *y* правого верхнего угла окна
`exitonclick()`	Приостанавливает работу программы и закрывает окно, когда пользователь щелкает кнопкой мыши в любом месте. Без этой функции графическое окно закрывалось бы сразу по окончании работы программы

Модуль Python `turtle` отображает линии на экране без задержки. Однако при рисовании тысяч линий это может замедлить работу программы. Для ее ускорения можно использовать *буферизацию*, то есть приостановить вывод на экран нескольких линий, чтобы затем показать их все сразу.

Вызов функции `turtle.tracer(1000, 0)` позволяет задержать отображение рисунка до тех пор, пока программа не создаст 1000 линий. После того как ваша программа завершит рисование линий, вызовите `turtle.update()` еще раз, чтобы отобразить на экране все буферизованные линии. Если ваша программа по-прежнему слишком долго рисует изображение, задайте в качестве первого аргумента функции `turtle.tracer()` число побольше, например `2000` или `10000`.

Треугольник Серпинского

Самый простой фрактал, который можно нарисовать на бумаге, — это *треугольник Серпинского*, упомянутый в главе 1. Этот фрактал был описан польским математиком Вацлавом Серпинским в 1915 году (еще до появления термина «фрактал»). Однако сам узор известен уже сотни лет.

Чтобы создать треугольник Серпинского, нарисуйте равносторонний треугольник вроде того, который изображен слева на рис. 9.3. Затем нарисуйте перевернутый равносторонний треугольник внутри первого, как показано правее на рис. 9.3. В результате у вас получится фигура, напоминающая Трифорс из видеоигр *Legend of Zelda*.

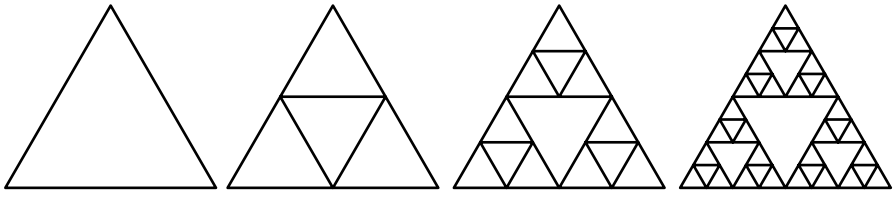


Рис. 9.3. Равносторонний треугольник *(слева)* и треугольники Серпинского, полученные путем рекурсивного добавления перевернутых треугольников

Когда вы начертите внутренний перевернутый треугольник, произойдет интересная вещь. Вы получите три новых равносторонних треугольника, обращенных вершиной вверх. Внутри каждого из них нарисуйте еще по одному перевернутому треугольнику, чтобы получить уже девять. Математически эту рекурсию можно продолжать бесконечно, чего нельзя сказать о рисовании таких треугольников на бумаге.

Объект, который подобен части самого себя, обладает свойством *самоподобия*. Рекурсивные функции позволяют создавать такие объекты, поскольку они многократно «вызывают» сами себя. На практике этот код должен в конце концов достичь базового случая, однако с точки зрения математики эти фигуры бесконечны.

Давайте напишем рекурсивную программу для создания треугольника Серпинского. Функция `drawTriangle()` нарисует равносторонний треугольник, а затем рекурсивно вызовет саму себя три раза, чтобы нарисовать внутренние равносторонние треугольники, как показано на рис. 9.4. Функция `midpoint()` делит каждую сторону треугольника пополам точкой. Эти точки равноудалены от двух других, переданных функции, и используются в качестве вершин внутренних треугольников.

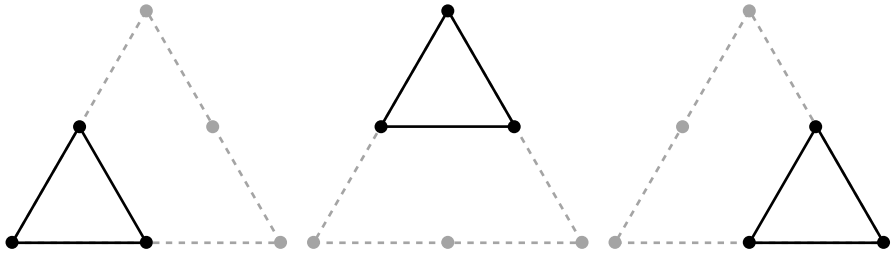


Рис. 9.4. Три внутренних треугольника с вершинами в равноудаленных точках

Обратите внимание, что данная программа вызывает функцию `turtle.setworldcoordinates(0, 0, 700, 700)`, помещая начало координат (0, 0) в нижний левый угол окна. Верхний правый угол имеет координаты $x = 700$ и $y = 700$. Код программы `sierpinskiTriangle.py` выглядит следующим образом:

```
import turtle
turtle.tracer(100, 0) # Увеличьте первый аргумент, чтобы ускорить процесс рисования
turtle.setworldcoordinates(0, 0, 700, 700)
turtle.hideturtle()
MIN_SIZE = 4 # Измените это значение, чтобы уменьшить/увеличить глубину рекурсии

def midpoint(startx, starty, endx, endy):
    # Возвратите координаты x и y точки, находящейся точно посередине между
    # двумя точками с заданными параметрами
    xDiff = abs(startx - endx)
    yDiff = abs(starty - endy)
    return (min(startx, endx) + (xDiff / 2.0), min(starty, endy) + (yDiff / 2.0))

def isTooSmall(ax, ay, bx, by, cx, cy):
    # Выясните, не является ли треугольник слишком маленьким, чтобы его
    # можно было нарисовать
    width = max(ax, bx, cx) - min(ax, bx, cx)
    height = max(ay, by, cy) - min(ay, by, cy)
    return width < MIN_SIZE or height < MIN_SIZE
def drawTriangle(ax, ay, bx, by, cx, cy):
    if isTooSmall(ax, ay, bx, by, cx, cy):
        # БАЗОВЫЙ СЛУЧАЙ
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        # Нарисуйте треугольник
        turtle.penup()
        turtle.goto(ax, ay)
        turtle.pendown()
        turtle.goto(bx, by)
        turtle.goto(cx, cy)
        turtle.goto(ax, ay)
        turtle.penup()
```

```
        # Найдите средние точки между точками A, B и C
        mid_ab = midpoint(ax, ay, bx, by)
        mid_bc = midpoint(bx, by, cx, cy)
        mid_ca = midpoint(cx, cy, ax, ay)
        # Нарисуйте три внутренних треугольника
        drawTriangle(ax, ay, mid_ab[0], mid_ab[1], mid_ca[0], mid_ca[1])
        drawTriangle(mid_ab[0], mid_ab[1], bx, by, mid_bc[0], mid_bc[1])
        drawTriangle(mid_ca[0], mid_ca[1], mid_bc[0], mid_bc[1], cx, cy)
        return

# Нарисуйте равносторонний треугольник Серпинского
drawTriangle(50, 50, 350, 650, 650, 50)

# Нарисуйте искаженный треугольник Серпинского
#drawTriangle(30, 250, 680, 600, 500, 80)

turtle.exitonclick()
```

На рис. 9.5 показан результат.

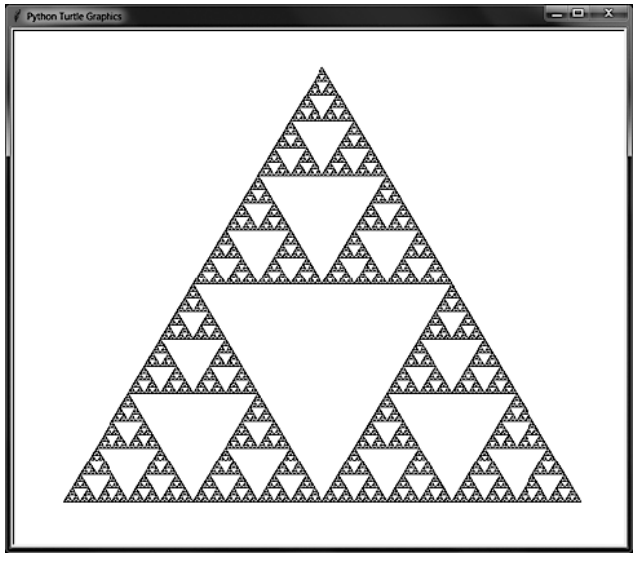


Рис. 9.5. Стандартный треугольник Серпинского

Треугольники Серпинского не обязательно формировать из равносторонних треугольников. Эти треугольники могут быть любого вида при условии, что вы используете середины сторон внешнего треугольника в качестве вершин для внутренних. Закомментируйте первый вызов функции `drawTriangle()` и раскомментируйте второй (под комментарием `# Нарисуйте искаженный треугольник Серпинского`)

и снова запустите программу. Результат ее выполнения будет выглядеть так, как показано на рис. 9.6.

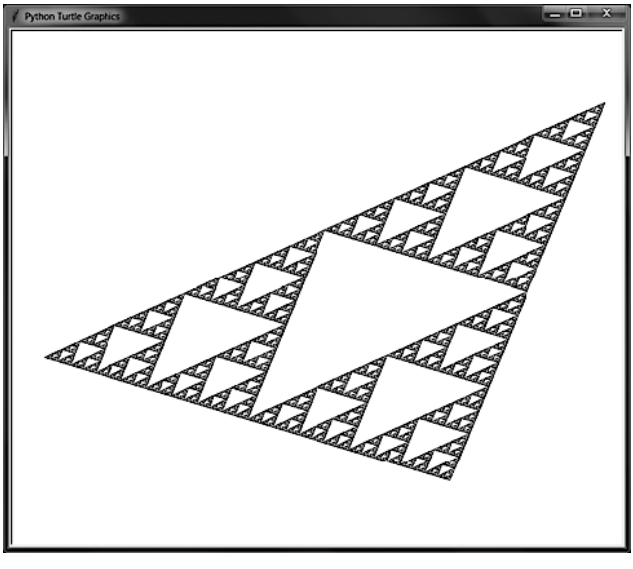


Рис. 9.6. Искаженный треугольник Серпинского

Функция `drawTriangle()` принимает шесть аргументов, соответствующих координатам *x* и *y* трех вершин треугольника. Поэкспериментируйте с разными значениями, чтобы настроить форму треугольника Серпинского. Вы также можете увеличить значение константы `MIN_SIZE`, чтобы ускорить достижение базового случая и сократить количество рисуемых фигур.

Ковер Серпинского

Используя вместо треугольников прямоугольники, можно получить фрактальный узор, известный как *ковер Серпинского*. Представьте черный прямоугольник, разбитый по линиям сетки 3 × 3, и мысленно вырежьте из него центральный прямоугольник. Повторите это же действие с каждым из окружающих восьми прямоугольников. Если делать это рекурсивно, можно получить узор, изображенный на рис. 9.7.

Программа Python, рисующая такой узор, использует функции `turtle.begin_fill()` и `turtle.end_fill()` для создания заполненных цветом фигур. Линии, которые рисует черепашка между этими вызовами, образуют форму, показанную на рис. 9.8.

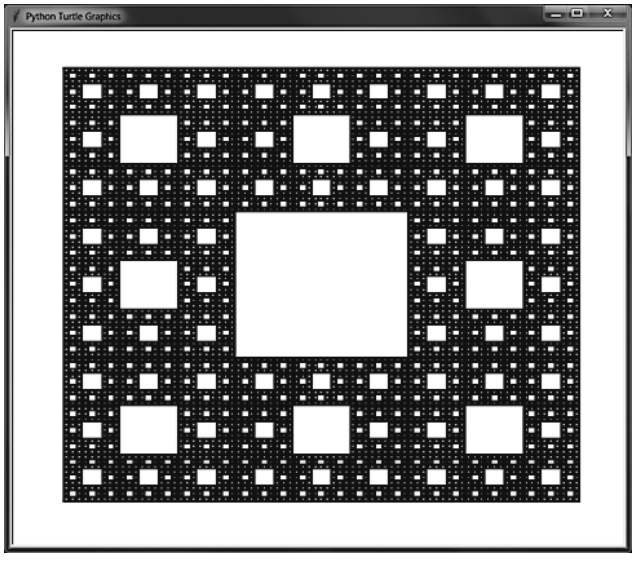


Рис. 9.7. Ковер Серпинского

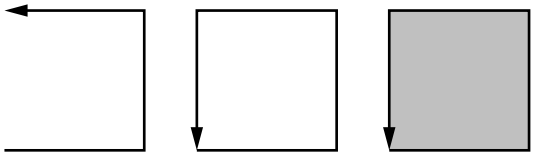


Рис. 9.8. Вызов turtle.begin_fill(), рисование пути и вызов turtle.end_fill() создают заполненную цветом фигуру

Базовый случай достигается, когда длина стороны прямоугольника становится меньше шести шагов. Допускается увеличить значение константы `MIN_SIZE`, чтобы ускорить достижение базового случая. Код программы `sierpinskiCarpet.py` выглядит следующим образом:

```
import turtle
turtle.tracer(10, 0) # Увеличьте первый аргумент, чтобы ускорить процесс рисования
turtle.setworldcoordinates(0, 0, 700, 700)
turtle.hideturtle()

MIN_SIZE = 6 # Измените это значение, чтобы уменьшить/увеличить глубину рекурсии
DRAW_SOLID = True

def isTooSmall(width, height):
    # Выясните, не является ли прямоугольник слишком маленьким, чтобы его
    # можно было нарисовать
    return width < MIN_SIZE or height < MIN_SIZE
```

```
def drawCarpet(x, y, width, height):
    # Координаты x и y соответствуют левому нижнему углу ковра

    # Переместите перо в нужное положение
    turtle.penup()
    turtle.goto(x, y)
    # Нарисуйте внешний прямоугольник
    turtle.pendown()
    if DRAW_SOLID:
        turtle.fillcolor('black')
        turtle.begin_fill()
    turtle.goto(x, y + height)
    turtle.goto(x + width, y + height)
    turtle.goto(x + width, y)
    turtle.goto(x, y)
    if DRAW_SOLID:
        turtle.end_fill()
    turtle.penup()

    # Нарисуйте внутренние прямоугольники
    drawInnerRectangle(x, y, width, height)

def drawInnerRectangle(x, y, width, height):
    if isTooSmall(width, height):
        # БАЗОВЫЙ СЛУЧАЙ
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ

        oneThirdWidth = width / 3
        oneThirdHeight = height / 3
        twoThirdsWidth = 2 * (width / 3)
        twoThirdsHeight = 2 * (height / 3)

        # Переместитесь в нужное положение
        turtle.penup()
        turtle.goto(x + oneThirdWidth, y + oneThirdHeight)

        # Нарисуйте внутренний прямоугольник
        if DRAW_SOLID:
            turtle.fillcolor('white')
            turtle.begin_fill()
        turtle.pendown()
        turtle.goto(x + oneThirdWidth, y + twoThirdsHeight)
        turtle.goto(x + twoThirdsWidth, y + twoThirdsHeight)
        turtle.goto(x + twoThirdsWidth, y + oneThirdHeight)
        turtle.goto(x + oneThirdWidth, y + oneThirdHeight)
        turtle.penup()
        if DRAW_SOLID:
            turtle.end_fill()
        # Нарисуйте внутренние прямоугольники в верхнем ряду
        drawInnerRectangle(x, y + twoThirdsHeight, oneThirdWidth, oneThirdHeight)
```

```
        drawInnerRectangle(x + oneThirdWidth, y + twoThirdsHeight,
          oneThirdWidth, oneThirdHeight)
        drawInnerRectangle(x + twoThirdsWidth, y + twoThirdsHeight,
          oneThirdWidth, oneThirdHeight)

        # Нарисуйте внутренние прямоугольники в среднем ряду
        drawInnerRectangle(x, y + oneThirdHeight, oneThirdWidth,
        oneThirdHeight)
        drawInnerRectangle(x + twoThirdsWidth, y + oneThirdHeight,
          oneThirdWidth, oneThirdHeight)

        # Нарисуйте внутренние прямоугольники в нижнем ряду
        drawInnerRectangle(x, y, oneThirdWidth, oneThirdHeight)
        drawInnerRectangle(x + oneThirdWidth, y, oneThirdWidth, oneThirdHeight)
        drawInnerRectangle(x + twoThirdsWidth, y, oneThirdWidth,
        oneThirdHeight)
drawCarpet(50, 50, 600, 600)
turtle.exitonclick()
```

Также разрешается задать для константы `DRAW_SOLID` значение `False`. Это позволит пропустить вызовы функций `turtle.begin_fill()` и `turtle.end_fill()`, чтобы нарисовать только контуры прямоугольников, как показано на рис. 9.9.

Попробуйте передать функции `drawCarpet()` другие аргументы. Первые два аргумента — это координаты *x* и *y* левого нижнего угла ковра, а последние два — его ширина и высота. Можете также увеличить значение константы `MIN_SIZE`, чтобы ускорить достижение базового случая и сократить количество рисуемых фигур.

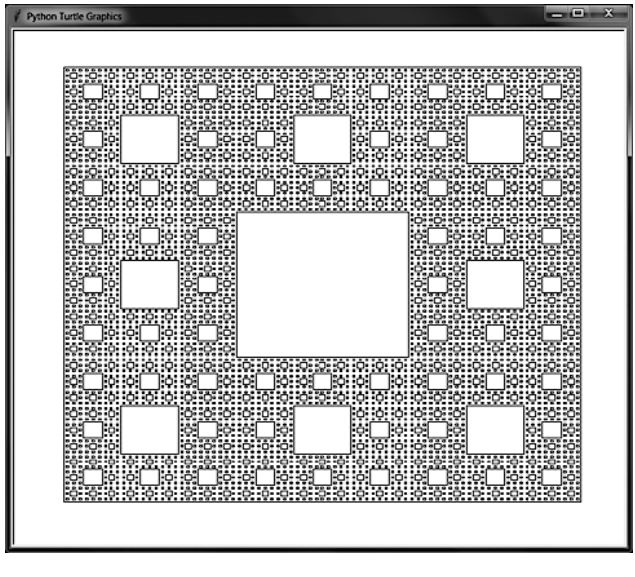


Рис. 9.9. Ковер Серпинского, образованный только контурами прямоугольников

Трехмерный ковер Серпинского, образованный кубами, называется *кубом Серпинского* или *губкой Менгера*. Впервые этот фрактал был описан математиком Карлом Менгером в 1926 году. На рис. 9.10 показана губка Менгера, созданная в видеоигре *Minecraft*.

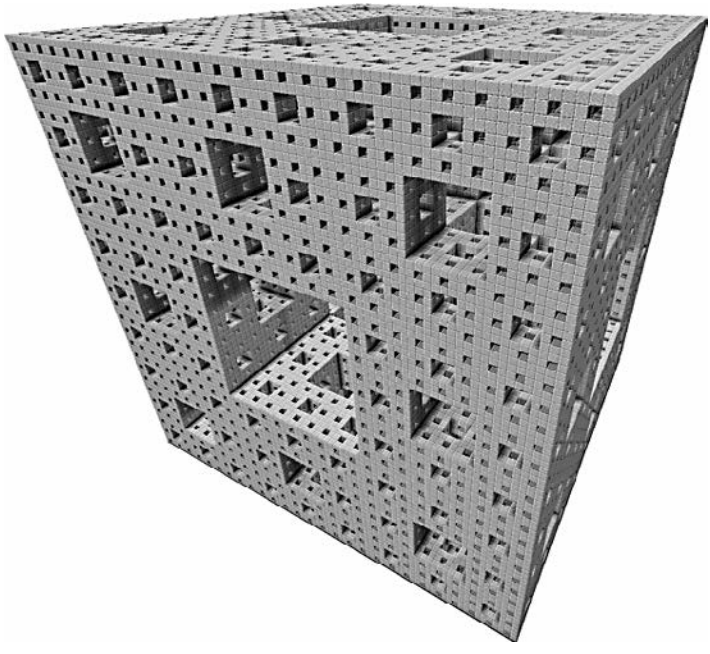


Рис. 9.10. Трехмерный фрактал под названием «губка Менгера»

Фрактальные деревья

Фракталами могут быть не только такие фигуры, как треугольник и ковер Серпинского, но и те, которые не обладают идеальным самоподобием. Фрактальная геометрия, созданная математиком Бенуа Б. Мандельбротом (его средний инициал рекурсивно ссылается на его полное имя Бенуа Б. Мандельброт), рассматривает такие естественные формы, как горы, береговые линии, растения, сети кровеносных сосудов и скопления галактик, в роли фракталов. При внимательном рассмотрении становится понятно, что эти объекты состоят из «грубых» форм, которые нелегко описать с помощью гладких кривых и прямых линий обычной геометрии.

В качестве примера, используя рекурсию, мы можем сгенерировать *фрактальные деревья*, как обладающие, так и не обладающие свойством идеального самоподобия. Формирование таких деревьев предполагает создание ветви с двумя дочерними ветвями, которые отходят от родительской под заданными углами

и укорачиваются на определенную величину. Получающаяся в итоге Y-образная форма рекурсивно повторяется, создавая убедительные изображения деревьев, приведенные на рис. 9.11 и 9.12.

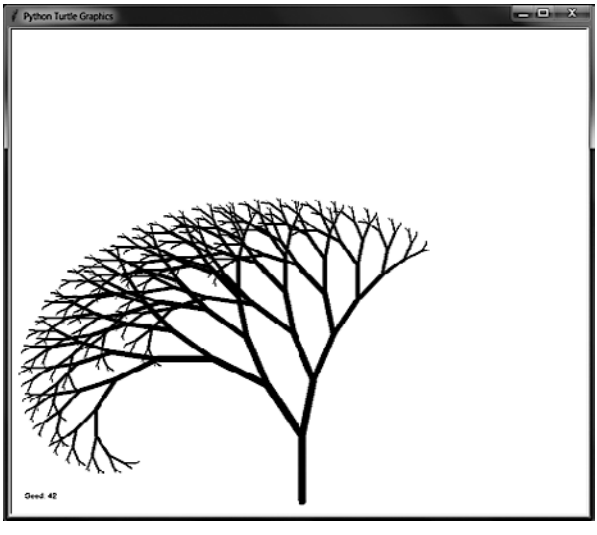


Рис. 9.11. Идеально самоподобное фрактальное дерево, полученное при использовании одинаковых углов и соотношений длин ветвей

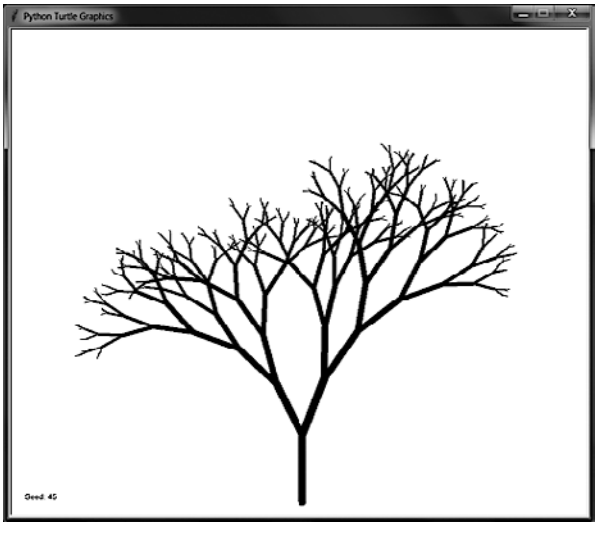


Рис. 9.12. Более реалистичное дерево, созданное с использованием случайных вариаций в значениях углов и соотношений длин ветвей

При создании фильмов и видеоигр эти рекурсивные алгоритмы могут использоваться в рамках так называемой *процедурной генерации*, позволяющей автоматически, а не вручную создавать 3D-модели различных объектов: деревьев, папоротников, цветов и других растений. С помощью алгоритмов компьютеры быстро генерируют целые леса, состоящие из миллионов уникальных деревьев, освобождая от этой работы 3D-художников.

Представленный далее код программы `fractalTree.py` каждые две секунды отображает на экране новое случайно сгенерированное дерево:

```
import random
import time
import turtle
turtle.tracer(1000, 0) # Увеличиваем первый аргумент, чтобы ускорить процесс рисования
turtle.setworldcoordinates(0, 0, 700, 700)
turtle.hideturtle()

def drawBranch(startPosition, direction, branchLength):
    if branchLength < 5:
        # БАЗОВЫЙ СЛУЧАЙ
        return

    # Переходим в начальную точку и задаем направление
    turtle.penup()
    turtle.goto(startPosition)
    turtle.setheading(direction)

    # Рисуем ветвь, толщина которой составляет 1/7 длины
    turtle.pendown()
    turtle.pensize(max(branchLength / 7.0, 1))
    turtle.forward(branchLength)
    # Зафиксируем положение конца ветви
    endPosition = turtle.position()
    leftDirection = direction + LEFT_ANGLE
    leftBranchLength = branchLength - LEFT_DECREASE
    rightDirection = direction - RIGHT_ANGLE
    rightBranchLength = branchLength - RIGHT_DECREASE
    # РЕКУРСИВНЫЙ СЛУЧАЙ
    drawBranch(endPosition, leftDirection, leftBranchLength)
    drawBranch(endPosition, rightDirection, rightBranchLength)

seed = 0
while True:
    # Получаем псевдослучайные числа для свойств ветви
    random.seed(seed)
    LEFT_ANGLE     = random.randint(10,  30)
    LEFT_DECREASE  = random.randint( 8,  15)
    RIGHT_ANGLE    = random.randint(10,  30)
    RIGHT_DECREASE = random.randint( 8,  15)
    START_LENGTH   = random.randint(80, 120)
```

```
    # Выводим на экран значение сида
    turtle.clear()
    turtle.penup()
    turtle.goto(10, 10)
    turtle.write('Seed: %s' % (seed))

    # Рисуем дерево
    drawBranch((350, 10), 90, START_LENGTH)
    turtle.update()
    time.sleep(2)

    seed = seed + 1
```

Данная программа создает идеально самоподобные деревья, поскольку переменные `LEFT_ANGLE`, `LEFT_DECREASE`, `RIGHT_ANGLE` и `RIGHT_DECREASE` изначально выбираются случайным образом, но остаются постоянными для всех рекурсивных вызовов. Функция `random.seed()` задает значение так называемого сида (стартовой позиции, с которой начинается последовательность) для функций Python, генерирующих случайные числа. Это значение заставляет программу генерировать числа, кажущиеся случайными, однако для создания каждой ветви дерева программа использует одну и ту же последовательность. Другими словами, один и то же *сид* производит одно и то же *дерево* при каждом запуске программы.

Чтобы увидеть, как это работает, введите следующий код в окно интерактивной оболочки Python:

```
>>> import random
>>> random.seed(42)
>>> [random.randint(0, 9) for i in range(20)]
 [1, 0, 4, 3, 3, 2, 1, 8, 1, 9, 6, 0, 0, 1, 3, 3, 8, 9, 0, 8]
>>> [random.randint(0, 9) for i in range(20)]
 [3, 8, 6, 3, 7, 9, 4, 0, 2, 6, 5, 4, 2, 3, 5, 1, 1, 6, 1, 5]
>>> random.seed(42)
>>> [random.randint(0, 9) for i in range(20)]
 [1, 0, 4, 3, 3, 2, 1, 8, 1, 9, 6, 0, 0, 1, 3, 3, 8, 9, 0, 8]
```

В текущем примере в качестве сида задано случайное число `42`. При генерировании 20 случайных целых чисел получаем `1`, `0`, `4`, `3` и т. д. Мы можем и дальше продолжать этот процесс, генерируя все новые случайные целые числа. Однако если сбросим значение сида на `42` и снова сгенерируем 20 случайных целых чисел, то получим те же «случайные» целые числа, как и раньше.

Если необходимо создать дерево, которое выглядит более естественно, замените строки после комментария `# Зафиксируем положение конца ветви` следующими строками (содержащийся в них код формирует новые случайные значения углов и длин ветвей для *каждого* рекурсивного вызова, что больше похоже на рост настоящих деревьев):

```
# Зафиксируем положение конца ветви
endPosition = turtle.position()
leftDirection = direction + random.randint(10, 30)
leftBranchLength = branchLength - random.randint(8, 15)
rightDirection = direction - random.randint(10, 30)
rightBranchLength = branchLength - random.randint(8, 15)
```

Поэкспериментируйте с разными диапазонами при вызове функции `random.randint()` или создайте более двух ветвей, увеличив количество рекурсивных вызовов.

Какова длина береговой линии Великобритании? Кривая и снежинка Коха

Прежде чем мы перейдем к обсуждению кривой и снежинки Коха, попробуйте ответить на вопрос: какова длина береговой линии Великобритании? Посмотрите на рис. 9.13. На карте слева указаны приблизительные измерения, согласно которым длина береговой линии составляет примерно 3220 км. На карте справа указаны более точные измерения, согласно которым ее длина составляет около 4500 км.

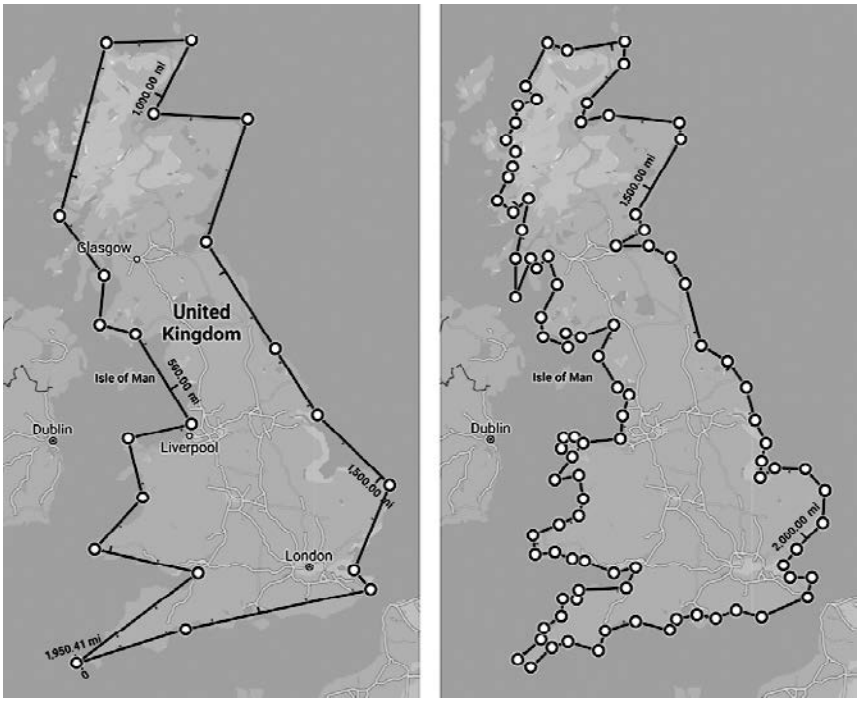


Рис. 9.13. Карта Великобритании с приблизительной длиной береговой линии *(слева)* и более точной *(справа)*. Во втором случае длина береговой линии больше на 1280 км

Ключевое открытие Мандельброта относительно таких фракталов, как береговая линия, заключается в том, что подобные формы имеют «неровности» при любом масштабе. Таким образом, по мере приближения и повышения точности ваших измерений береговая линия будет удлиняться, например следуя вверх по течению Темзы вглубь суши и снова возвращаясь к Ла-Маншу. Таким образом, на вопрос относительно длины береговой линии Великобритании нельзя ответить однозначно.

Аналогичным свойством обладает периметр фрактала, образуемого *кривой Коха*. Кривая Коха, представленная в 1904 году шведским математиком Хельге фон Кохом, является одним из первых фракталов, описанных математически. Чтобы построить его, возьмите линию длиной b и разделите ее на три одинаковых отрезка, равных $b/3$. Замените средний отрезок равносторонним треугольником со сторонами $b/3$ и отсутствующим основанием. Добавление этого треугольника увеличивает длину исходной кривой, поскольку теперь она состоит не из трех, а из четырех отрезков $b/3$ (исходный средний отрезок линии был исключен). Затем добавьте такой же треугольник на полученных четырех отрезках. Процесс построения данного фрактала изображен на рис. 9.14.

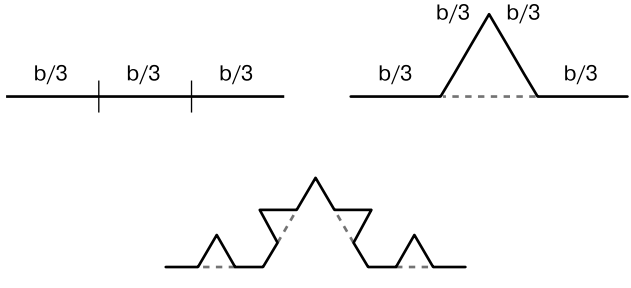


Рис. 9.14. Разделив линию на три равных отрезка *(слева)*, добавьте равносторонний треугольник без основания *(справа)*. Теперь мы имеем четыре отрезка длиной b/3, к каждому из которых тоже можно добавить треугольник *(внизу)*

Чтобы создать *снежинку Коха*, начнем с равностороннего треугольника и превратим его стороны в три кривые Коха, как показано на рис. 9.15.

Каждый раз, добавляя новый треугольник, вы увеличиваете длину кривой с $3b/3$ до $4b/3$. При последовательном повторении данной операции на трех сторонах равностороннего треугольника мы получим снежинку Коха, изображенную на рис. 9.16 (пунктирные линии — артефакты, обусловленные небольшими ошибками округления, из-за которых модуль `turtle` не может полностью удалить средний отрезок). Теоретически добавлять треугольники можно бесконечно, однако наша программа завершается, когда их размер достигает нескольких пикселов.

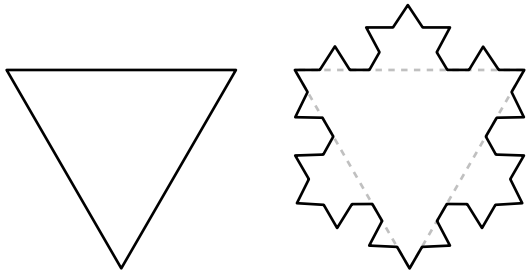


Рис. 9.15. Создание трех кривых Коха на трех сторонах равностороннего треугольника для формирования снежинки Коха

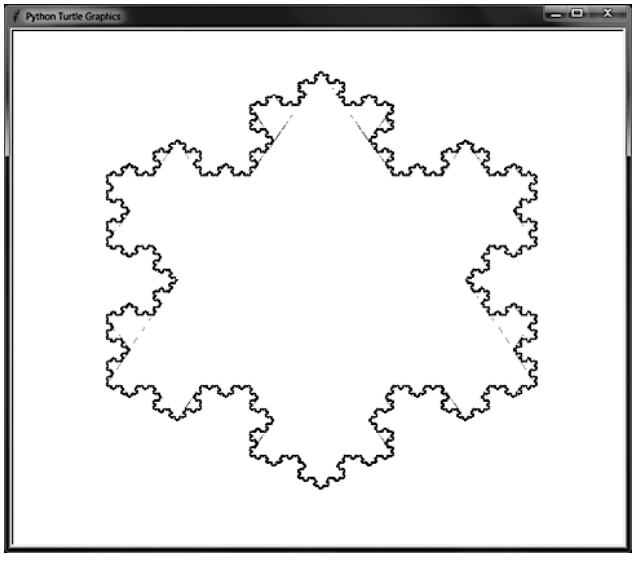


Рис. 9.16. Снежинка Коха. Часть внутренних линий остается из-за небольших ошибок округления

Код программы `kochSnowake.py` выглядит следующим образом:

```
import turtle
turtle.tracer(10, 0) # Увеличиваем первый аргумент, чтобы ускорить процесс рисования
turtle.setworldcoordinates(0, 0, 700, 700)
turtle.hideturtle()
turtle.pensize(2)

def drawKochCurve(startPosition, heading, length):
    if length < 1:
```

```
        # БАЗОВЫЙ СЛУЧАЙ
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        # Перемещаемся в исходное положение
        recursiveArgs = []
        turtle.penup()
        turtle.goto(startPosition)
        turtle.setheading(heading)
        recursiveArgs.append({'position':turtle.position(),
                              'heading':turtle.heading()})

        # Стираем средний отрезок
        turtle.forward(length / 3)
        turtle.pencolor('white')
        turtle.pendown()
        turtle.forward(length / 3)

        # Рисуем треугольник без нижней стороны
        turtle.backward(length / 3)
        turtle.left(60)
        recursiveArgs.append({'position':turtle.position(),
                              'heading':turtle.heading()})
        turtle.pencolor('black')
        turtle.forward(length / 3)
        turtle.right(120)
        recursiveArgs.append({'position':turtle.position(),
                              'heading':turtle.heading()})
        turtle.forward(length / 3)
        turtle.left(60)
        recursiveArgs.append({'position':turtle.position(),
                              'heading':turtle.heading()})
        for i in range(4):
            drawKochCurve(recursiveArgs[i]['position'],
                      recursiveArgs[i]['heading'],
                      length / 3)
        return

def drawKochSnowflake(startPosition, heading, length):
    # Снежинка Коха — это три кривые Коха на сторонах треугольника

    # Перемещаемся в исходное положение
    turtle.penup()
    turtle.goto(startPosition)
    turtle.setheading(heading)
    for i in range(3):
        # Фиксируем исходное положение и курс
        curveStartingPosition = turtle.position()
        curveStartingHeading = turtle.heading()
        drawKochCurve(curveStartingPosition,
                      curveStartingHeading, length)
```

```
        # Возвращаемся в исходное положение на данной стороне
        turtle.penup()
        turtle.goto(curveStartingPosition)
        turtle.setheading(curveStartingHeading)

        # Перемещаемся в исходное положение для создания следующей стороны
        turtle.forward(length)
        turtle.right(120)

drawKochSnowflake((100, 500), 0, 500)
turtle.exitonclick()
```

Снежинку Коха также иногда называют *островом Коха*, чья береговая линия обладает буквально бесконечной длиной. Несмотря на то что снежинка Коха умещается в конечной области страницы книги, длина ее периметра бесконечна. Это доказывает парадоксальную идею, что нечто конечное может заключать в себе бесконечность!

Кривая Гильберта

Заполняющая пространство кривая — это одномерная линия, которая изгибается, не пересекая саму себя, вплоть до заполнения двумерного пространства. Немецкий математик Давид Гильберт описал свою заполняющую пространство кривую в 1891 году. Если разделить двумерную область по линиям сетки, то одномерная кривая Гильберта сможет заполнить каждую ее ячейку.

На рис. 9.17 показаны первые три шага создания кривой Гильберта. Каждый последующий содержит четыре копии результата, полученного на предыдущем шаге, а пунктирная линия показывает, как эти четыре копии соединяются друг с другом.

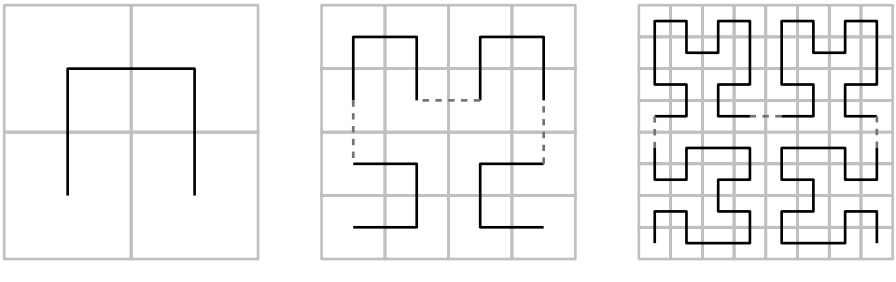


Рис. 9.17. Первые три шага создания кривой Гильберта

Когда ячейки превращаются в бесконечно малые точки, такая одномерная кривая оказывается способной заполнить все двумерное пространство так же, как это делает двумерный квадрат. Вопреки логике это создает двумерную форму из строго одномерной линии!

Код программы `hilbertCurve.py` выглядит следующим образом:

```
import turtle
turtle.tracer(10, 0) # Увеличим первый аргумент, чтобы ускорить процесс рисования
turtle.setworldcoordinates(0, 0, 700, 700)
turtle.hideturtle()

LINE_LENGTH  = 5 # Попробуем немного изменить длину линии
ANGLE = 90 # Попробуем изменить угол поворота на несколько градусов
LEVELS = 6 # Попробуем слегка изменить уровень рекурсии
DRAW_SOLID = False
#turtle.setheading(20) # Раскомментируйте эту строку,
                       # чтобы нарисовать кривую под углом

def hilbertCurveQuadrant(level, angle):
    if level == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
        turtle.right(angle)
        hilbertCurveQuadrant(level - 1, -angle)
        turtle.forward(LINE_LENGTH)
        turtle.left(angle)
        hilbertCurveQuadrant(level - 1, angle)
        turtle.forward(LINE_LENGTH)
        hilbertCurveQuadrant(level - 1, angle)
        turtle.left(angle)
        turtle.forward(LINE_LENGTH)
        hilbertCurveQuadrant(level - 1, -angle)
        turtle.right(angle)
        return

def hilbertCurve(startingPosition):
    # Переместимся в исходное положение
    turtle.penup()
    turtle.goto(startingPosition)
    turtle.pendown()
    if DRAW_SOLID:
        turtle.begin_fill()

    hilbertCurveQuadrant(LEVELS, ANGLE) # Рисуем левый нижний квадрант
    turtle.forward(LINE_LENGTH)

    hilbertCurveQuadrant(LEVELS, ANGLE) # Рисуем правый нижний квадрант
    turtle.left(ANGLE)
    turtle.forward(LINE_LENGTH)
    turtle.left(ANGLE)

    hilbertCurveQuadrant(LEVELS, ANGLE) # Рисуем правый верхний квадрант
    turtle.forward(LINE_LENGTH)
```

```
    hilbertCurveQuadrant(LEVELS, ANGLE) # Рисуем левый верхний квадрант

    turtle.left(ANGLE)
    turtle.forward(LINE_LENGTH)
    turtle.left(ANGLE)
    if DRAW_SOLID:
        turtle.end_fill()

hilbertCurve((30, 350))
turtle.exitonclick()
```

Поэкспериментируйте с этим кодом, уменьшив значение `LINE_LENGTH`, чтобы сократить длину отрезков, и увеличив значение `LEVELS`, чтобы добавить дополнительные уровни рекурсии. Поскольку в данной программе движение черепашки является относительным, вы можете удалить символ комментария на строке `turtle.setheading(20)`, чтобы нарисовать кривую Гильберта под углом 20°. На рис. 9.18 показана кривая, созданная при использовании параметра `LINE_LENGTH`, равного `10`, и параметра `LEVELS`, равного `5`.

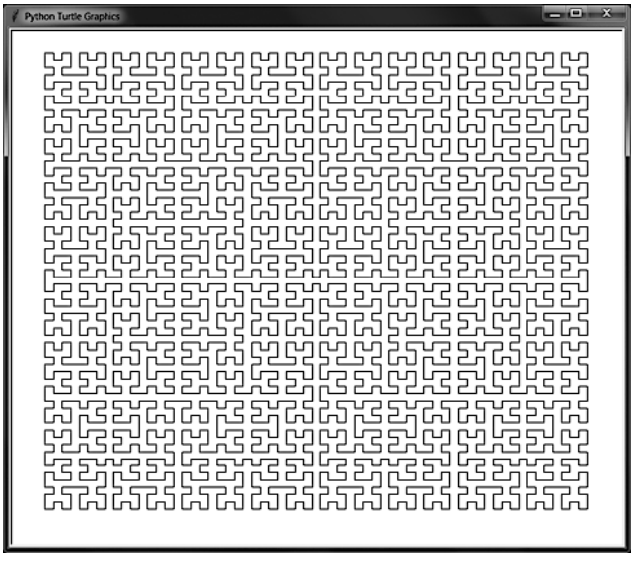


Рис. 9.18. Результат создания кривой Гильберта при использовании пяти уровней рекурсии и линии длиной 10

Кривая Гильберта делает повороты под углом 90° (то есть под прямым углом). Попробуйте изменить значение переменной `ANGLE` на несколько градусов, например, уменьшите его до `89` или `86`, и запустите программу, чтобы оценить результат. Вы также можете задать для переменной `DRAW_SOLID` значение `True`, чтобы получить заполненную цветом кривую Гильберта, показанную на рис. 9.19.

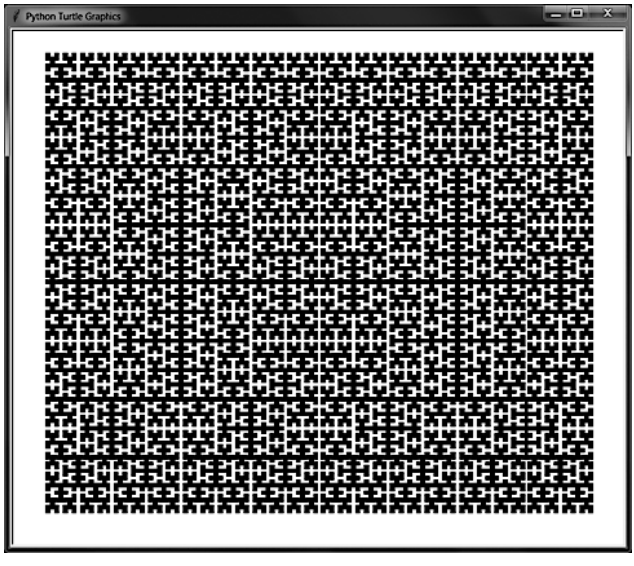


Рис. 9.19. Результат создания заполненной цветом кривой Гильберта при использовании шести уровней рекурсии и линии длиной 5

Резюме

Невероятно обширная область фракталов сочетает в себе самые интересные аспекты программирования и искусства, благодаря чему процесс написания данной главы оказался чрезвычайно увлекательным. Математики и информатики говорят о красоте и изяществе самых продвинутых концепций соответствующих областей знаний, однако рекурсивные фракталы способны превратить эту концептуальную красоту в визуальную, доступную для восприятия каждому человеку.

Нами было рассмотрено несколько фракталов, а также программ, позволяющих их нарисовать. В частности, мы обсудили треугольник и ковер Серпинского, процедурно генерируемые фрактальные деревья, кривую и снежинку Коха и кривую Гильберта. Все они были нарисованы с помощью модуля Python `turtle` и рекурсивных функций, вызывающих самих себя.

Дополнительные источники информации

Чтобы узнать больше о рисовании с помощью модуля Python `turtle`, рекомендую ознакомиться с простым руководством моего авторства, доступным по ссылке https://github.com/asweigart/simple-turtle-tutorial-for-python. Моя личная коллекция программ,

использующих данный модуль, представлена на странице https://github.com/asweigart/art-of-turtle-programming.

Вопрос о длине береговой линии Великобритании взят из названия статьи Мандельброта 1967 года. Данная идея подробно изложена в статье «Википедии» по адресу https://ru.wikipedia.org/wiki/Парадокс_береговой_линии. Дополнительную информацию о геометрии снежинки Коха можно найти по ссылке https://www.khanacademy.org/math/geometry-home/geometry-volume-surface-area/koch-snowflake/v/koch-snowflake-fractal.

На YouTube-канале 3Blue1Brown опубликованы отличные видео с анимацией фракталов, в частности *Fractals Are Typically Not Self-Similar* (https://youtu.be/gB9n2gHsHN4) и *Fractal Charm: Space-Filling Curves* (https://youtu.be/RU0wScIj36o).

Информацию о других заполняющих пространство кривых, для рисования которых используется рекурсия, в частности о кривой Пеано, кривой Госпера и кривой дракона, вы можете самостоятельно найти в Интернете.

Вопросы для закрепления

Проверьте, усвоили ли вы пройденный материал, ответив на следующие вопросы.

1. Что такое фракталы?
2. Что обозначают координаты *x* и *y* в декартовой системе координат?
3. Что такое начало координат в декартовой системе координат?
4. Что такое процедурная генерация?
5. Что такое сид?
6. Какова длина периметра снежинки Коха?
7. Что такое заполняющая пространство кривая?

Практика

Решите каждую из следующих задач.

1. С помощью модуля Python `turtle` создайте программу, которая рисует фрактал Вичека, изображенный на рис. 9.20. Эта программа аналогична той, к которой мы прибегали для рисования ковра Серпинского. Используйте функции `turtle.begin_fill()` и `turtle.end_fill()`, чтобы нарисовать первый большой черный квадрат. Затем разделите его на девять равных частей и нарисуйте белые квадраты посередине вверху, слева, справа и внизу. Повторите процесс для четырех угловых квадратов и центрального черного квадрата.

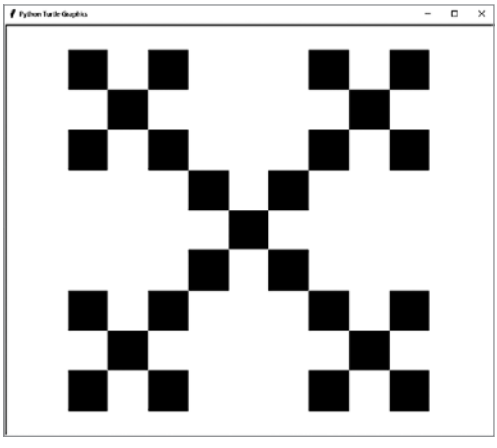


Рис. 9.20. Фрактал Вичека, нарисованный с использованием двух уровней рекурсии

2. С помощью модуля Python `turtle` создайте программу, которая рисует заполняющую пространство кривую Пеано. Эта программа похожа на программу для рисования кривой Гильберта. На рис. 9.21 показаны первые три итерации построения кривой Пеано. При построении кривой Гильберта область последовательно разбивается по линиям сетки 2×2, тогда как при построении кривой Пеано используется сетка 3×3.

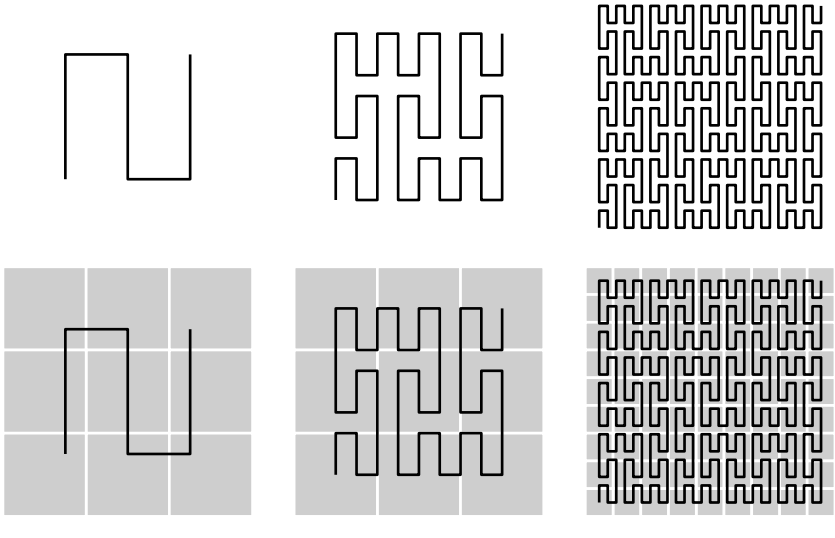


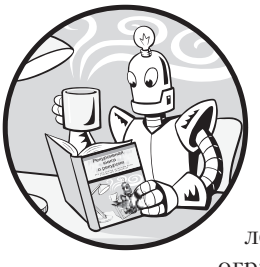
Рис. 9.21. Первые три итерации построения кривой Пеано *(слева направо)*. В нижнем ряду показано разбиение заполняемой области по линиям сетки 3×3, в каждой из ячеек которой находится та или иная часть кривой

ЧАСТЬ II

Проекты

10

Инструмент для поиска файлов



В этой главе вы напишете собственную рекурсивную программу для поиска файлов в соответствии со своими потребностями. Ваш компьютер уже предусматривает несколько команд и приложений для подобных задач, однако зачастую их возможности ограничены. Что, если вам необходимо использовать какой-то специфический критерий поиска, например найти все файлы с четным значением размера в байтах или с именами, содержащими все гласные буквы?

Скорее всего, вам никогда не придется выполнять именно такой поиск, однако рано или поздно может возникнуть потребность в использовании нестандартных критериев поиска. И будет лучше, если вы освоите принципы написания соответствующего кода заранее.

Как уже говорилось, рекурсия больше всего подходит для решения проблем, предусматривающих использование древовидной структуры данных. Файловая система на вашем компьютере подобна дереву (см. рис. 2.6). Каждая папка «разветвляется» на подпапки, которые, в свою очередь, могут разветвляться еще на одни. В данной главе мы напишем рекурсивную функцию для навигации по этому дереву.

ПРИМЕЧАНИЕ

Поскольку JavaScript не имеет доступа к папкам на вашем компьютере, программа для данного проекта будет написана только на языке Python.

Полный код программы для поиска файлов

Начнем с рассмотрения полного исходного кода рекурсивной программы для поиска файлов. В оставшейся части главы каждый ее раздел будет рассмотрен отдельно. Скопируйте исходный код программы в файл с именем `fileFinder.py`:

```
import os

def hasEvenByteSize(fullFilePath):
    """Возвращает True, если значение размера fullFilePath в байтах является четным.
    В противном случае возвращает False."""
    fileSize = os.path.getsize(fullFilePath)
    return fileSize % 2 == 0

def hasEveryVowel(fullFilePath):
    """Возвращает True, если имя fullFilePath содержит буквы a, e, i, o, u,
    иначе возвращает False."""
    name = os.path.basename(fullFilePath).lower()
    return ('a' in name) and ('e' in name) and ('i' in name) and ('o' in name)
    and ('u' in name)


def walk(folder, matchFunc):
    """Вызывает функцию сопоставления для каждого файла в папке
       и ее подпапках. Возвращает список файлов, для которых функция
       сопоставления вернула значение True."""
    matchedFiles = [] # Этот список содержит все совпадения.
    folder = os.path.abspath(folder) # Используйте абсолютный путь к папке.

    # Переберите в цикле все файлы и подпапки в папке:
    for name in os.listdir(folder):
        filepath = os.path.join(folder, name)
        if os.path.isfile(filepath):
            # Вызовите функцию сопоставления для каждого файла:
            if matchFunc(filepath):
                matchedFiles.append(filepath)
        elif os.path.isdir(filepath):
            # Рекурсивно вызовите функцию walk для каждой подпапки
            # и добавьте обнаруженные совпадения в matchedFiles:
            matchedFiles.extend(walk(filepath, matchFunc))
    return matchedFiles
print('All files with even byte sizes:')
print(walk('.', hasEvenByteSize))
print('All files with every vowel in their name:')
print(walk('.', hasEveryVowel))
```

Основой программы для поиска файлов является функция `walk()`, которая «обходит» все файлы в базовой папке и ее подпапках. Она вызывает одну из двух других функций, которые реализуют пользовательские критерии поиска. В контексте данной программы мы будем называть их *функциями сопоставления*. Вызов такой функции возвращает значение `True`, если файл соответствует критериям поиска, в противном случае он возвращает значение `False`.

Задача `walk()` состоит в том, чтобы вызвать функцию сопоставления один раз для каждого файла, содержащегося в каталогах, которые она обходит. Давайте рассмотрим код более подробно.

Функции сопоставления

В Python при вызове функции мы можем передавать в качестве аргументов сами функции. В следующем примере `callTwice()` дважды вызывает функцию, переданную ей в качестве аргумента, будь то `sayHello()` или `sayGoodbye()`:

```
>>> def callTwice(func):
...     func()
...     func()
...
>>> def sayHello():
...     print('Hello!')
...
>>> def sayGoodbye():
...     print('Goodbye!')
...
>>> callTwice(sayHello)
Hello!
Hello!
>>> callTwice(sayGoodbye)
Goodbye!
Goodbye!
```

Функция `callTwice()` вызывает любую функцию, переданную ей в качестве параметра `func`. Обратите внимание, что мы опускаем круглые скобки в аргументе функции и пишем `callTwice(sayHello)` вместо `callTwice(sayHello())`. Это связано с тем, что мы передаем саму функцию `sayHello()`, а не вызываем `sayHello()` и передаем возвращаемое ею значение.

Функция `walk()` принимает в качестве аргумента функцию соответствия, служащую критерием поиска. Это позволяет нам настраивать поведение программы для поиска файлов, не изменяя код `walk()`. Обсудим функцию `walk()` чуть позже. Сначала рассмотрим два примера функций сопоставления.

Поиск файлов с четным значением размера в байтах

Первая функция сопоставления находит файлы с четным значением размера в байтах:

```
import os

def hasEvenByteSize(fullFilePath):
    """Возвращает True, если размер fullFilePath в байтах — четное число.
    В противном случае возвращает False."""
    fileSize = os.path.getsize(fullFilePath)
    return fileSize % 2 == 0
```

Мы импортируем модуль os, который используется в этой программе для получения информации о файлах на компьютере, с помощью таких функций, как getsize(), basename() и т. д. Затем создаем функцию сопоставления с именем hasEvenByteSize(). Все функции сопоставления принимают один строковый аргумент fullFilePath и, если совпадение обнаружено, возвращают True, в противном случае — False.

Функция os.path.getsize() определяет размер файла fullFilePath в байтах. Затем мы используем оператор %, чтобы определить четность полученного числа. Если оно кратно двум, инструкция return возвращает значение True, а если нет, то False. Например, давайте определим размер приложения Блокнот, которое поставляется с ОС Windows (в macOS или Linux попробуйте применить эту функцию к программе /bin/ls):

```
>>> import os
>>> os.path.getsize('C:/Windows/system32/notepad.exe')
211968
>>> 211968 % 2 == 0
True
```

Функция сопоставления hasEvenByteSize() может использовать любую функцию Python для нахождения дополнительной информации о файле fullFilePath. Это дает возможность писать код с учетом любых критериев поиска. В процессе обхода папок и подпапок walk() вызывает функцию сопоставления для каждого файла, содержащегося в этих каталогах. Данная функция сопоставления возвращает True или False, сообщая walk() о том, соответствует ли тот или иной файл критериям поиска.

Поиск имен файлов, содержащих все гласные

Рассмотрим еще одну функцию сопоставления:

```
def hasEveryVowel(fullFilePath):
    """Возвращает True, если имя файла fullFilePath содержит
    гласные a, e, i, o, u, иначе возвращает False."""
    name = os.path.basename(fullFilePath).lower()
    return ('a' in name) and ('e' in name) and ('i' in name) and
      ('o' in name) and ('u' in name)
```

Мы вызываем функцию os.path.basename(), чтобы удалить имена папок из пути к файлу. Python сравнивает строки с учетом регистра, поэтому нам нужно гарантировать, что функция hasEveryVowel() не пропустит ни одной гласной в имени файла из-за того, что она написана в верхнем регистре. Например, вызов os.path.basename('C:/Windows/system32/notepad.exe') возвращает строку notepad.exe. Вызов метода lower() преобразует символы строки в нижний регистр, благодаря чему нам достаточно проверить наличие в ней лишь строчных гласных.

Далее, в разделе «Полезные функции стандартной библиотеки Python для работы с файлами» нам предстоит рассмотреть еще несколько функций для получения информации о файлах.

Мы используем оператор `return` с длинным выражением, которое оценивается как `True`, если `name` содержит `a`, `e`, `i`, `o` или `u`, то есть соответствует критериям поиска. В противном случае оператор `return` возвращает `False`.

Рекурсивная функция walk()

Функции сопоставления проверяют, соответствует ли файл критериям поиска, а функция `walk()` находит все файлы, подлежащие проверке. Рекурсивной функции `walk()` передается имя базовой папки для выполнения поиска и функция сопоставления, которая должна вызываться для каждого содержащегося в этом каталоге файла.

Функция `walk()` также рекурсивно вызывает сама себя для проверки каждой подпапки, находящейся в базовой папке. В ходе рекурсивного вызова эти вложенные папки играют роль базовой.

Ответим на три вопроса относительно текущей рекурсивной функции.

Что представляет собой базовый случай? Базовый случай имеет место, когда функция завершает обработку всех файлов и подпапок в базовом каталоге.

Какой аргумент передается рекурсивной функции при ее вызове? Базовая папка, в которой будет осуществляться поиск, и функция сопоставления для поиска файлов, соответствующих указанным критериям. Для всех папок внутри базовой выполняется рекурсивный вызов, в ходе которого соответствующая подпапка играет роль базового каталога.

Как этот аргумент приближается к базовому случаю? В конечном итоге функция либо завершает проверку всех вложенных папок, либо обнаруживает базовые папки, в которых отсутствуют подпапки.

На рис. 10.1 показан пример файловой системы вместе с рекурсивными вызовами функции `walk()`, которые она совершает при обходе базовой папки `C:\`.

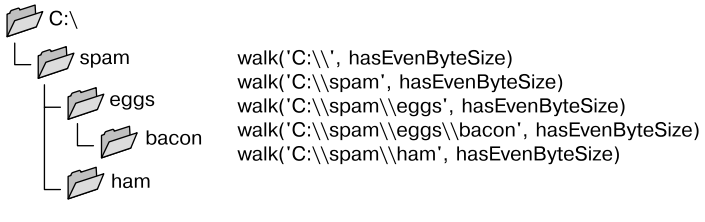


Рис. 10.1. Пример файловой системы и рекурсивных вызовов функции walk()

Давайте проанализируем код функции `walk()`:

```
def walk(folder, matchFunc):
    """Вызывает функцию сопоставления для каждого файла в папке
       и ее подпапках. Возвращает список файлов, для которых функция
       сопоставления вернула значение True."""
    matchedFiles = [] # Этот список содержит все совпадения.
    folder = os.path.abspath(folder) # Используйте абсолютный путь к папке.
```

Функция `walk()` имеет два параметра: `folder` — строка с именем базовой папки, в которой требуется осуществить поиск (мы можем передать значение `'.'`, чтобы указать на тот каталог, из которого запускается программа Python), и `matchFunc` — функция Python, которая принимает имя файла и возвращает `True`, если совпадение найдено. В противном случае функция возвращает `False`.

Следующая часть функции проверяет содержимое `folder`:

```
    # Переберите в цикле все файлы и подпапки в папке:
    for name in os.listdir(folder):
        filepath = os.path.join(folder, name)
        if os.path.isfile(filepath):
```

Цикл `for` вызывает `os.listdir()` для возвращения списка содержимого папки `folder`. Этот список включает все файлы и подкаталоги. Для каждого файла создаем полный абсолютный путь, присоединяя к имени каталога имя файла или папки. Если имя ссылается на файл, вызов функции `os.path.isfile()` возвращает `True`, и мы проверяем данный файл на соответствие критериям поиска:

```
            # Вызовите функцию сопоставления для каждого файла:
            if matchFunc(filepath):
                matchedFiles.append(filepath)
```

Иначе говоря, вызываем функцию сопоставления, передавая ей полный абсолютный путь к текущему файлу цикла `for`. Обратите внимание, что `matchFunc` — это имя одного из параметров функции `walk()`. Когда в качестве аргумента для параметра `matchFunc` передается функция наподобие `hasEvenByteSize()` и `hasEveryVowel()`, имеет место вызов функции `walk()`. Если `filepath` содержит файл, соответствующий критериям поиска, он добавляется в список совпадений:

```
        elif os.path.isdir(filepath):
            # Рекурсивно вызовите функцию walk для каждой подпапки и добавьте
            # обнаруженные совпадения в matchedFiles:
            matchedFiles.extend(walk(filepath, matchFunc))
```

В противном случае, если текущим файлом в цикле `for` является подпапка, вызов функции `os.path.isdir()` возвращает значение `True`. Затем мы передаем этот подкаталог в ходе рекурсивного вызова функции. Такой рекурсивный вызов возвращает

список всех отвечающих критериям файлов в подпапке (и вложенных в нее папках), которые затем добавляются в список совпадений:

```
    return matchedFiles
```

После завершения работы цикла `for` список совпадений должен содержать все отвечающие критериям файлы в этой директории (и во всех ее подпапках). Данный список становится возвращаемым значением для функции `walk()`.

Вызов функции walk()

Теперь, когда реализована функция `walk()` и несколько функций сопоставления, можно запустить процесс поиска файлов. В качестве первого аргумента мы передаем в `walk()` строку `'.'`, указывая, что базовой папкой для осуществления поиска является *текущий каталог*, из которого была запущена программа:

```
print('All files with even byte sizes:')
print(walk('.', hasEvenByteSize))
print('All files with every vowel in their name:')
print(walk('.', hasEveryVowel))
```

Результат выполнения программы зависит от того, какие именно файлы находятся на вашем компьютере. В данном случае он просто демонстрирует способ написания кода для использования нужных вам критериев поиска. Например, вывод программы может выглядеть следующим образом:

```
All files with even byte sizes:
['C:\\Path\\accesschk.exe', 'C:\\Path\\accesschk64.exe',
'C:\\Path\\AccessEnum.exe', 'C:\\Path\\ADExplorer.exe',
'C:\\Path\\Bginfo.exe', 'C:\\Path\\Bginfo64.exe',
'C:\\Path\\diskext.exe', 'C:\\Path\\diskext64.exe',
'C:\\Path\\Diskmon.exe', 'C:\\Path\\DiskView.exe',
'C:\\Path\\hex2dec64.exe', 'C:\\Path\\jpegtran.exe',
'C:\\Path\\Tcpview.exe', 'C:\\Path\\Testlimit.exe',
'C:\\Path\\wget.exe', 'C:\\Path\\whois.exe']
All files with every vowel in their name:
['C:\\Path\\recursionbook.bat']
```

Полезные функции стандартной библиотеки Python для работы с файлами

Рассмотрим операции, которые могут вам пригодиться при написании собственных функций сопоставления. Стандартная библиотека модулей, поставляемая вместе с Python, содержит несколько полезных функций для получения информации о файлах. Многие из них являются частью модулей `os` и `shutil`, поэтому ваша

программа должна будет выполнить команду `import os` или `import shutil`, прежде чем вызвать эти функции.

Поиск информации об имени файла

Полный путь к файлу, передаваемый функциям сопоставления, может быть разбит на базовое имя и имя каталога с помощью функций `os.path.basename()` и `os.path.dirname()`. Вы также можете вызвать `os.path.split()`, чтобы получить эти имена в виде кортежа. Введите следующий код в окне интерактивной оболочки Python (в ОС macOS или Linux попробуйте использовать в качестве имени файла `/bin/ls`):

```
>>> import os
>>> filename = 'C:/Windows/system32/notepad.exe'
>>> os.path.basename(filename)
'notepad.exe'
>>> os.path.dirname(filename)
'C:/Windows/system32'
>>> os.path.split(filename)
('C:/Windows/system32', 'notepad.exe')
>>> folder, file = os.path.split(filename)
>>> folder
'C:/Windows/system32'
>>> file
'notepad.exe'
```

При проверке файла на соответствие критериям поиска к этим строковым значениям можно применить любой строковый метод Python, например, как это было сделано в случае с методом `lower()` в функции сопоставления `hasEveryVowel()`.

Поиск информации о временных метках файла

Файлы предусматривают временные метки, указывающие момент их создания, последнего изменения и последнего обращения к ним. Функции Python `os.path.getctime()`, `os.path.getmtime()` и `os.path.getatime()` возвращают эти временные метки в виде значений с плавающей точкой, указывающих количество секунд, прошедших с начала *эпохи Unix*, которая наступила в полночь 1 января 1970 года по всемирному координированному времени (Coordinated Universal Time, UTC). Введите следующий код в окне интерактивной оболочки:

```
> import os
> filename = 'C:/Windows/system32/notepad.exe'
> os.path.getctime(filename)
1625705942.1165037
> os.path.getmtime(filename)
1625705942.1205275
> os.path.getatime(filename)
1631217101.8869188
```

Программы с легкостью используют значения с плавающей точкой, поскольку они представляют собой обычные числа. Однако для того, чтобы сделать их более простыми для восприятия, потребуются функции из модуля Python `time`. Функция `time.localtime()` преобразует временную метку Unix в объект `struct_time`, ссылающийся на часовой пояс компьютера. Объект `struct_time` имеет несколько атрибутов, имена которых начинаются с `tm_`, для получения информации о дате и времени. Введите следующий код в окне интерактивной оболочки:

```
>>> import os
>>> filename = 'C:/Windows/system32/notepad.exe'
>>> ctimestamp = os.path.getctime(filename)
>>> import time
>>> time.localtime(ctimestamp)
time.struct_time(tm_year=2021, tm_mon=7, tm_mday=7, tm_hour=19,
tm_min=59, tm_sec=2, tm_wday=2, tm_yday=188, tm_isdst=1)
>>> st = time.localtime(ctimestamp)
>>> st.tm_year
2021
>>> st.tm_mon
7
>>> st.tm_mday
7
>>> st.tm_wday
2
>>> st.tm_hour
19
>>> st.tm_min
59
>>> st.tm_sec
2
```

Имейте в виду, что атрибут `tm_mday` обозначает день месяца в диапазоне от `1` до `31`, а `tm_wday` — день недели, причем понедельник соответствует значению `0`, вторник — значению `1` и т. д., вплоть до воскресенья, соответствующего `6`.

Если вы хотите превратить объект `time_struct` в короткую, удобочитаемую строку, передайте его функции `time.asctime()`:

```
>>> import os
>>> filename = 'C:/Windows/system32/notepad.exe'
>>> ctimestamp = os.path.getctime(filename)
>>> import time
>>> st = time.localtime(ctimestamp)
>>> time.asctime(st)
'Wed Jul  7 19:59:02 2021'
```

Функция `time.localtime()` возвращает объект `struct_time`, использующий местное время, а `time.gmtime()` возвращает объект, использующий часовой пояс UTC или среднее время по Гринвичу (GMT). Введите в окне интерактивной оболочки следующий код:

```
>>> import os
>>> filename = 'C:/Windows/system32/notepad.exe'
>>> ctimestamp = os.path.getctime(filename)
>>> import time
>>> ctimestamp = os.path.getctime(filename)
>>> time.localtime(ctimestamp)
time.struct_time(tm_year=2021, tm_mon=7, tm_mday=7, tm_hour=19,
tm_min=59, tm_sec=2, tm_wday=2, tm_yday=188, tm_isdst=1)
>>> time.gmtime(ctimestamp)
time.struct_time(tm_year=2021, tm_mon=7, tm_mday=8, tm_hour=0,
tm_min=59, tm_sec=2, tm_wday=3, tm_yday=189, tm_isdst=0)
```

Взаимодействие между функциями `os.path` (которые возвращают временные метки Unix) и функциями `time` (возвращающими объекты `struct_time`) может сбить с толку. На рис. 10.2 показана цепочка команд, начинающаяся со строки с именем файла и заканчивающаяся получением отдельных фрагментов временной метки.

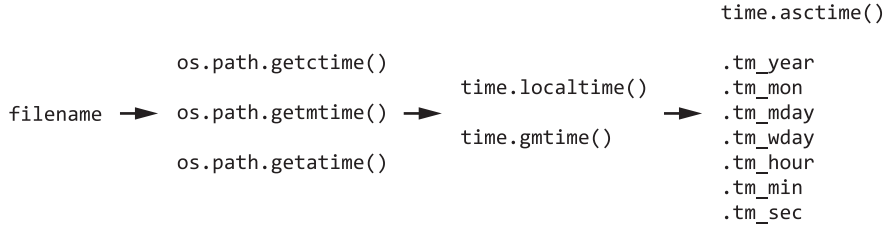


Рис. 10.2. Переход от имени файла к отдельным атрибутам временной метки

Наконец, функция `time.time()` возвращает количество секунд, прошедших с начала эпохи Unix до текущего момента.

Изменение файлов

После того как функция `walk()` возвратит список файлов, соответствующих критериям поиска, можно переименовать, удалить или выполнить над ними другую операцию. Модули `shutil` и `os` стандартной библиотеки Python предусматривают соответствующие функции. Кроме того, модуль `send2trash`, созданный сторонними разработчиками, позволяет отправлять файлы в корзину вместо того, чтобы безвозвратно их удалять.

Чтобы переместить файл, вызовите функцию `shutil.move()`, передав ей в виде аргументов имена перемещаемого файла и папки, в которую его нужно переместить. Например, можете выполнить следующие команды:

```
>>> import shutil
>>> shutil.move('spam.txt', 'someFolder')
'someFolder\\spam.txt'
```

Функция `shutil.move()` возвращает строку, содержащую новый путь к файлу. Имеется возможность переименовать файл в момент его перемещения:

```
>>> import shutil
>>> shutil.move('spam.txt', 'someFolder\\newName.txt')
'someFolder\\newName.txt'
```

Если второй аргумент не содержит имени папки, допускается просто указать новое имя файла, чтобы переименовать его в той директории, в которой он находится:

```
>>> import shutil
>>> shutil.move('spam.txt', 'newName.txt')
'newName.txt'
```

Имейте в виду, что функция `shutil.move()` отвечает как за перемещение, так и за переименование файлов, подобно команде `mv` в Unix и macOS. Отдельной функции `shutil.rename()` не существует.

Чтобы скопировать файл, используйте функцию `shutil.copy()`, передав ей два аргумента — имя копируемого файла и новое имя для его копии. Например, вы можете выполнить следующие команды:

```
>>> import shutil
>>> shutil.copy('spam.txt', 'spam-copy.txt')
'spam-copy.txt'
```

Функция `shutil.copy()` возвращает имя копии. Чтобы удалить файл, вызовите функцию `os.unlink()` и передайте ей имя удаляемого файла:

```
>>> import os
>>> os.unlink('spam.txt')
>>>
```

В данном случае используется слово *unlink* («отвязать»), а не *delete* («удалить»), поскольку с технической точки зрения происходит исключение имени, связанного с файлом. Но так как большинство файлов имеют лишь одно связанное с ними имя, удаление этой связи приводит к удалению самого файла. Не переживайте, если эти концепции кажутся вам непонятными, просто имейте в виду, что функция `os.unlink()` отвечает за удаление файла.

Вызов функции `os.unlink()` навсегда стирает файл, что может представлять опасность в случае наличия ошибки в программе, приводящей к удалению не того файла. Вместо нее вы можете использовать функцию `send2trash()` одноименного модуля, чтобы поместить файл в корзину. Для установки этого модуля введите код `run python -m pip install --user send2trash` в командной строке Windows или `run python3 -m pip install` в окне терминала macOS или Linux. После установки модуля вы сможете импортировать его с помощью команды `import send2trash`.

Введите в окне интерактивной оболочки следующий код:

```
>>> open('deleteme.txt', 'w').close() # Создайте пустой файл
>>> import send2trash
>>> send2trash.send2trash('deleteme.txt')
```

В данном примере создается пустой файл с именем `deleteme.txt`. После вызова `send2trash.send2trash()` (модуль и функция имеют одно и то же имя) этот файл перемещается в корзину.

Резюме

Описанная в главе программа для поиска файлов использует рекурсию для «обхода» содержимого базовой папки и всех ее каталогов. Функция `walk()` рекурсивно обходит эти папки, проверяя содержащиеся в них файлы на предмет соответствия пользовательским критериям поиска. Данные критерии реализованы в виде функций сопоставления, которые передаются функции `walk()`. Благодаря этому для изменения критериев поиска нам достаточно написать новые функции вместо того, чтобы модифицировать код `walk()`.

В нашем проекте были использованы две функции сопоставления для поиска файлов с четным размером в байтах или файлов, содержащих в своем имени гласные буквы. Однако вы можете написать собственные функции, чтобы передать их для `walk()`.

В этом и заключается мощь программирования: вы способны создавать функции, недоступные в коммерческих приложениях, исходя из собственных потребностей.

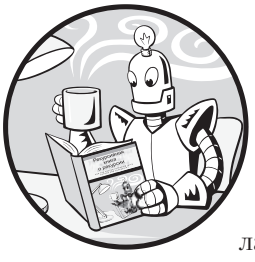
Дополнительные источники информации

Документацию по встроенной в Python функции `os.walk()` (напоминающей функцию `walk()` из описанной в данной главе программы) можно найти по адресу https://docs.python.org/3/library/os.html#os.walk. Чтобы узнать больше о файловой системе компьютера и функциях Python для работы с файлами, прочитайте главу 9 моей книги «Автоматизация рутинных задач с помощью Python», доступную по адресу https://automatetheboringstuff.com/2e/chapter9.

Модуль `datetime` стандартной библиотеки Python предусматривает множество способов взаимодействия с данными временных меток. Вы можете узнать о них подробнее из главы 17 моей книги «Автоматизация рутинных задач с помощью Python» (https://automatetheboringstuff.com/2e/chapter17).

11

Генератор лабиринтов



В главе 4 был описан рекурсивный алгоритм для прохождения лабиринтов. Теперь пришло время рассмотреть рекурсивный алгоритм для их генерации. В этой главе мы будем создавать лабиринты в таком же формате, в каком это делала программа для прохождения лабиринтов из главы 4. Итак, если вы любите проходить или создавать лабиринты, то у вас есть возможность применить свои навыки программирования к решению соответствующих задач.

Наш алгоритм начинает работу с начальной точки лабиринта, а затем рекурсивно посещает соседние, «вырезая» коридоры. Если алгоритм достигает тупика, по соседству с которым нет доступных для посещения точек, он возвращается к предыдущим, ранее не посещавшимся, и продолжает путь оттуда. К тому моменту, когда алгоритм возвращается в начальную точку, весь лабиринт оказывается сгенерированным.

Рекурсивный алгоритм поиска с возвратом, который мы будем использовать в данном проекте, как правило, создает лабиринты с длинными коридорами (пространствами, соединяющими развилки), и пройти их не составляет труда. Реализовать его гораздо легче, чем алгоритмы Краскала или Уилсона, поэтому именно предлагаемый способ хорошо подходит для знакомства с данной темой.

Полный код программы для создания лабиринта

Начнем с рассмотрения полного кода программ на языках Python и JavaScript, они реализуют рекурсивный алгоритм поиска с возвратом для создания лабиринта. В оставшейся части главы каждый раздел кода будет рассмотрен отдельно.

Скопируйте следующий код на языке Python в файл с именем `mazeGenerator.py`:

```
import random

WIDTH = 39 # Ширина лабиринта (значение должно быть нечетным)
HEIGHT = 19 # Высота лабиринта (значение должно быть нечетным)
assert WIDTH % 2 == 1 and WIDTH >= 3
assert HEIGHT % 2 == 1 and HEIGHT >= 3
SEED = 1
random.seed(SEED)

# Используйте эти символы для отображения лабиринта на экране:
EMPTY = ' '
MARK = '@'
WALL = chr(9608) # Символ 9608 представляет собой '█'
NORTH, SOUTH, EAST, WEST = 'n', 's', 'e', 'w'

# Для начала создайте структуру данных заполненного лабиринта:
maze = {}
for x in range(WIDTH):
    for y in range(HEIGHT):
        maze[(x, y)] = WALL # Каждая точка изначально занята стеной

 def printMaze(maze, markX=None, markY=None):
    """ Отображает структуру данных лабиринта, хранящуюся в переменной maze.
    Аргументы markX и markY — это координаты текущего местоположения
    алгоритма '@' в процессе создания лабиринта."""

    for y in range(HEIGHT):
        for x in range(WIDTH):
            if markX == x and markY == y:
                # Отобразите метку '@' здесь:
                print(MARK, end='')
            else:
                # Отобразите часть стены или пустое пространство:
                print(maze[(x, y)], end='')
        print() # Добавьте символ новой строки после отображения ряда символов

def visit(x, y):
    """Вырезайте пустые пространства в точках с координатами x и y,
    а затем рекурсивно перемещайтесь в соседние, ранее не посещавшиеся точки.
    Эта функция возвращается к предыдущим точкам, когда метка достигает тупика."""
    maze[(x, y)] = EMPTY # Вырезает пустое пространство
                         # в точке с координатами x и y.
    printMaze(maze, x, y) # Отображает лабиринт по мере его создания
    print('\n\n')

    while True:
        # Проверяет, какие из точек по соседству с меткой еще не посещались:
        unvisitedNeighbors = []
        if y > 1 and (x, y - 2) not in hasVisited:
            unvisitedNeighbors.append(NORTH)
```

```
        if y < HEIGHT - 2 and (x, y + 2) not in hasVisited:
            unvisitedNeighbors.append(SOUTH)

        if x > 1 and (x - 2, y) not in hasVisited:
            unvisitedNeighbors.append(WEST)

        if x < WIDTH - 2 and (x + 2, y) not in hasVisited:
            unvisitedNeighbors.append(EAST)
        if len(unvisitedNeighbors) == 0:
            # БАЗОВЫЙ СЛУЧАЙ
            # Все соседние точки были посещены, достигнут тупик.
            # Вернитесь в предыдущую точку:
            return
        else:
            # РЕКУРСИВНЫЙ СЛУЧАЙ
            # Случайным образом выберите непосещенную соседнюю точку для посещения:
            nextIntersection = random.choice(unvisitedNeighbors)

            # Переместите метку в непосещенную соседнюю точку:
            if nextIntersection == NORTH:
                nextX = x
                nextY = y - 2
                maze[(x, y - 1)] = EMPTY # Развилка
            elif nextIntersection == SOUTH:
                nextX = x
                nextY = y + 2
                maze[(x, y + 1)] = EMPTY # Развилка
            elif nextIntersection == WEST:
                nextX = x - 2
                nextY = y
                maze[(x - 1, y)] = EMPTY # Развилка
            elif nextIntersection == EAST:
                nextX = x + 2
                nextY = y
                maze[(x + 1, y)] = EMPTY # Развилка

            hasVisited.append((nextX, nextY)) # Отметьте точку как посещенную
            visit(nextX, nextY) # Рекурсивно посещайте эту точку

# Вырезание путей в структуре данных лабиринта:
hasVisited = [(1, 1)] # Начните с посещения левого верхнего угла
visit(1, 1)

# Отобразите итоговую структуру данных лабиринта:
printMaze(maze)
```

Скопируйте следующий код на языке JavaScript в файл с именем `mazeGenerator.html`:

```
<script type="text/javascript">

const WIDTH = 39; // Ширина лабиринта (значение должно быть нечетным)
const HEIGHT = 19; // Высота лабиринта (значение должно быть нечетным)
```

```
console.assert(WIDTH % 2 == 1 && WIDTH >= 3);
console.assert(HEIGHT % 2 == 1 && HEIGHT >= 3);

// Используйте эти символы для отображения лабиринта на экране:
const EMPTY = " ";
const MARK = "@";
const WALL = "&#9608;"; // Символ 9608 представляет собой '█'
const [NORTH, SOUTH, EAST, WEST] = ["n", "s", "e", "w"];

// Для начала создайте структуру данных заполненного лабиринта:
let maze = {};
for (let x = 0; x < WIDTH; x++) {
    for (let y = 0; y < HEIGHT; y++) {
        maze[[x, y]] = WALL; // Каждая точка изначально занята стеной
    }
}

function printMaze(maze, markX, markY) {
    // Отображает структуру данных лабиринта, хранящуюся в переменной maze.
    // Аргументы markX и markY — это координаты текущего местоположения
    // алгоритма '@' в процессе создания лабиринта
    document.write('<code>');
    for (let y = 0; y < HEIGHT; y++) {
        for (let x = 0; x < WIDTH; x++) {
            if (markX === x && markY === y) {
                // Отобразите метку '@' здесь:
                document.write(MARK);
            } else {
                // Отобразите часть стены или пустое пространство:
                document.write(maze[[x, y]]);
            }
        }
        document.write('<br />'); // Добавьте символ новой строки
                                  // после отображения ряда символов
    }
    document.write('</code>');
}

function visit(x, y) {
    // Вырезайте пустые пространства в точках с координатами x и y,
    // а затем рекурсивно перемещайтесь в соседние, ранее не посещавшиеся
    // точки. Эта функция возвращается
    // к предыдущим точкам, когда метка достигает тупика

    maze[[x, y]] = EMPTY; // Вырезает пустое пространство в точке
                          // с координатами x и y.
    printMaze(maze, x, y); // Отображает лабиринт по мере его создания
    document.write('<br /><br /><br />');

    while (true) {
        // Проверяет, какие из точек по соседству с меткой еще не посещались
        let unvisitedNeighbors = [];
```

```
        if (y > 1 && !JSON.stringify(hasVisited).includes(JSON.
        stringify([x, y - 2]))) {
            unvisitedNeighbors.push(NORTH);
        }
        if (y < HEIGHT - 2 &&
        !JSON.stringify(hasVisited).includes(JSON.stringify([x, y + 2]))) {
            unvisitedNeighbors.push(SOUTH);
        }
        if (x > 1 &&
        !JSON.stringify(hasVisited).includes(JSON.stringify([x - 2, y]))) {
            unvisitedNeighbors.push(WEST);
        }
        if (x < WIDTH - 2 &&
        !JSON.stringify(hasVisited).includes(JSON.stringify([x + 2, y]))) {
            unvisitedNeighbors.push(EAST);
        }
        if (unvisitedNeighbors.length === 0) {
            // БАЗОВЫЙ СЛУЧАЙ
            // Все соседние точки были посещены, достигнут тупик.
            // Вернитесь в предыдущую точку:
            return;
        } else {
            // РЕКУРСИВНЫЙ СЛУЧАЙ
            // Случайным образом выберите непосещенную соседнюю точку для посещения:
            let nextIntersection = unvisitedNeighbors[
            Math.floor(Math.random() * unvisitedNeighbors.length)];

            // Переместите метку в непосещенную соседнюю точку:
            let nextX, nextY;
            if (nextIntersection === NORTH) {
                nextX = x;
                nextY = y - 2;
                maze[[x, y - 1]] = EMPTY; // Развилка
            } else if (nextIntersection === SOUTH) {
                nextX = x;
                nextY = y + 2;
                maze[[x, y + 1]] = EMPTY; // Развилка
            } else if (nextIntersection === WEST) {
                nextX = x - 2;
                nextY = y;
                maze[[x - 1, y]] = EMPTY; // Развилка
            } else if (nextIntersection === EAST) {
                nextX = x + 2;
                nextY = y;
                maze[[x + 1, y]] = EMPTY; // Развилка
            }
            hasVisited.push([nextX, nextY]); // Отметьте точку как посещенную
            visit(nextX, nextY); // Рекурсивно посещайте эту точку
        }
    }
}
```

```
// Вырезание путей в структуре данных лабиринта:
let hasVisited = [[1, 1]]; // Начните с посещения левого верхнего угла
visit(1, 1);

// Отобразите итоговую структуру данных лабиринта:
printMaze(maze);
</script>
```

После запуска программы окно терминала или браузера начнет заполняться текстом по мере выполнения каждого шага построения лабиринта. Чтобы просмотреть весь вывод, воспользуйтесь прокруткой.

Изначально структура данных лабиринта представляет собой заполненное двумерное пространство. Рекурсивный алгоритм поиска с возвратом получает начальную точку, после чего обходит ранее не посещавшиеся соседние точки, вырезая коридоры в ходе этого процесса. Затем он рекурсивно вызывает сам себя для посещения соседней точки, в которой он ранее не бывал. Если все смежные точки уже были осмотрены, значит, алгоритм достиг тупика, поэтому он возвращается к предыдущим точкам в поисках их неисследованных «соседей». Программа завершается, когда алгоритм возвращается в начальное положение.

Вы можете увидеть данный алгоритм в действии, запустив генератор лабиринтов. В процессе вырезания коридоров (рис. 11.1) лабиринта координаты *x* и *y* текущего положения алгоритма обозначаются символом `@`. Обратите внимание на пятое изображение в верхнем правом углу рис. 11.1. На нем видно, как после достижения тупика алгоритм вернулся в посещенную ранее точку в поисках новых направлений для исследования.

Проанализируем код более подробно.

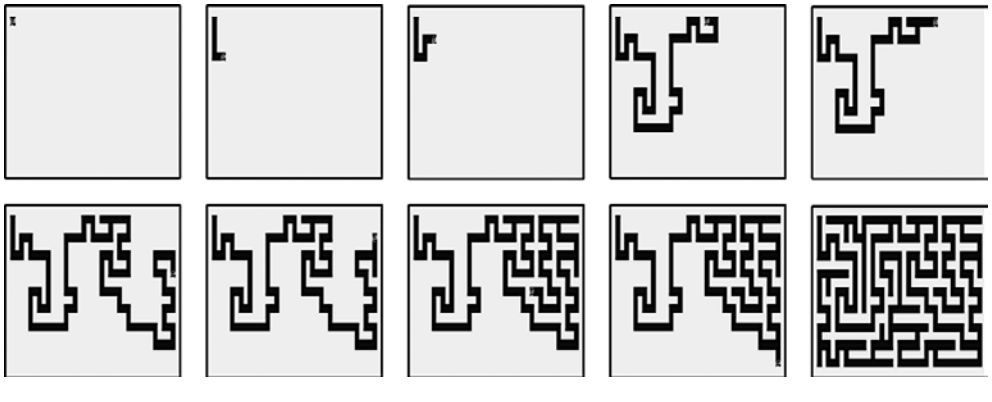


Рис. 11.1. Процесс создания лабиринта с помощью рекурсивного алгоритма поиска с возвратом

Задание констант генератора лабиринта

Генератор лабиринта использует несколько констант, значения которых мы можем скорректировать перед запуском программы, чтобы изменить размер и структуру лабиринта. Код Python для задания этих констант выглядит следующим образом:

```
import random

WIDTH = 39 # Ширина лабиринта (значение должно быть нечетным)
HEIGHT = 19 # Высота лабиринта (значение должно быть нечетным)
assert WIDTH % 2 == 1 and WIDTH >= 3
assert HEIGHT % 2 == 1 and HEIGHT >= 3
SEED = 1
random.seed(SEED)
```

Код JavaScript выглядит так:

```
<script type="text/javascript">

const WIDTH = 39; // Ширина лабиринта (значение должно быть нечетным)
const HEIGHT = 19; // Высота лабиринта (значение должно быть нечетным)
console.assert(WIDTH % 2 == 1 && WIDTH >= 3);
console.assert(HEIGHT % 2 == 1 && HEIGHT >= 3);
```

Константы `WIDTH` и `HEIGHT` определяют размер лабиринта. Они должны быть нечетными, потому что наша структура данных требует наличия стен между посещаемыми точками. Чтобы убедиться, что `WIDTH` и `HEIGHT` заданы правильно, используем специальные утверждения, позволяющие остановить выполнение программы, если значения констант четные или слишком маленькие.

Программа использует случайное начальное значение (сид) для воспроизведения одного и того же лабиринта. Версия текущей программы на языке Python позволяет задать это значение, вызвав функцию `random.seed()`. К сожалению, в JavaScript нет возможности явно задать сид, поэтому при каждом запуске программы будут генерироваться разные лабиринты.

ПРИМЕЧАНИЕ

«Случайные» числа, генерируемые программой Python, на самом деле предсказуемы, поскольку основаны на конкретном начальном значении; программа генерирует один и тот же «случайный» лабиринт при использовании одного и того же начального значения. Это может быть полезно при отладке программы, поскольку позволяет воспроизвести тот же самый лабиринт, если в нем была обнаружена ошибка.

Далее в коде Python задаются значения еще нескольких констант:

```
# Используйте эти символы для отображения лабиринта на экране:
EMPTY = ' '
MARK = '@'
WALL = chr(9608) # Символ 9608 представляет собой '█'
NORTH, SOUTH, EAST, WEST = 'n', 's', 'e', 'w'
```

Эквивалентный код JavaScript выглядит следующим образом:

```
// Используйте эти символы для отображения лабиринта на экране:
const EMPTY = " ";
const MARK = "@";
const WALL = "&#9608;"; // Символ 9608 представляет собой '█'
const [NORTH, SOUTH, EAST, WEST] = ["n", "s", "e", "w"];
```

Константы `EMPTY` и `WALL` влияют на то, как лабиринт отображается на экране. Константа `MARK` используется для указания положения алгоритма в лабиринте в процессе его работы. Константы `NORTH`, `SOUTH`, `EAST` и `WEST` соответствуют направлениям, в которых эта метка может перемещаться по лабиринту, и применяются для повышения удобочитаемости кода.

Создание структуры данных лабиринта

Структура данных лабиринта представляет собой словарь Python или объект JavaScript с ключами в виде кортежей Python или массивов JavaScript, содержащих координаты *x* и *y* каждой точки лабиринта. Значение этих ключей представляет собой строку, содержащуюся в константе `WALL` или `EMPTY`, которая определяет, является ли текущая точка частью стены лабиринта или пустым пространством.

Например, лабиринт, показанный на рис. 11.2, представлен следующей структурой данных:

```
{(0, 0): '█', (0, 1): '█', (0, 2): '█', (0, 3): '█', (0, 4): '█',
 (0, 5): '█', (0, 6): '█', (1, 0): '█', (1, 1): ' ', (1, 2): ' ',
 (1, 3): ' ', (1, 4): ' ', (1, 5): ' ', (1, 6): '█', (2, 0): '█',
 (2, 1): '█', (2, 2): '█', (2, 3): '█', (2, 4): '█', (2, 5): ' ',
 (2, 6): '█', (3, 0): '█', (3, 1): ' ', (3, 2): '█', (3, 3): ' ',
 (3, 4): ' ', (3, 5): ' ', (3, 6): '█', (4, 0): '█', (4, 1): ' ',
 (4, 2): '█', (4, 3): ' ', (4, 4): '█', (4, 5): '█', (4, 6): '█',
 (5, 0): '█', (5, 1): ' ', (5, 2): ' ', (5, 3): ' ', (5, 4): ' ',
 (5, 5): ' ', (5, 6): '█', (6, 0): '█', (6, 1): '█', (6, 2): '█',
 (6, 3): '█', (6, 4): '█', (6, 5): '█', (6, 6): '█'}
```

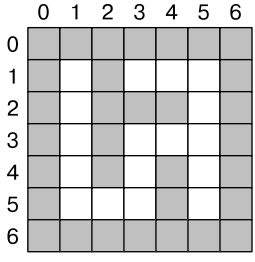


Рис. 11.2. Пример лабиринта, который можно представить с помощью структуры данных

В начале выполнения программы для каждой точки задано значение `WALL`. Затем рекурсивная функция `visit()` вырезает коридоры и развилки лабиринта, заменяя это значение на `EMPTY`:

```
# Для начала создайте структуру данных заполненного лабиринта:
maze = {}
for x in range(WIDTH):
    for y in range(HEIGHT):
        maze[(x, y)] = WALL # Каждая точка изначально занята стеной
```

Соответствующий код на языке JavaScript выглядит следующим образом:

```
// Для начала создайте структуру данных заполненного лабиринта:
let maze = {};
for (let x = 0; x < WIDTH; x++) {
    for (let y = 0; y < HEIGHT; y++) {
        maze[[x, y]] = WALL; // Каждая точка изначально занята стеной
    }
}
```

Мы создаем пустой словарь (в Python) или объект (в JavaScript) и сохраняем его в глобальной переменной `maze`. Циклы `for` перебирают все возможные координаты x и y, задавая для каждой из них значение `WALL`, чтобы создать заполненный лабиринт. После вызова функция `visit()` начнет вырезать коридоры лабиринта из этой структуры данных, задавая для некоторых ее точек значение `EMPTY`.

Вывод структуры данных лабиринта на экран

Для представления лабиринта в виде структуры данных программа Python использует словарь, а программа JavaScript — объект. В этой структуре ключи олицетворяют списки или массивы, содержащие два целых числа — координаты x и y, а значениями являются односимвольные строки `WALL` или `EMPTY`. Таким образом,

можно получить доступ к заполненной или пустой точке лабиринта с координатами x и y с помощью `maze[(x, y)]` в Python и `maze[[x, y]]` в JavaScript.

Код функции `printMaze()` на языке Python начинается следующим образом:

```
def printMaze(maze, markX=None, markY=None):
    """ Отображает структуру данных лабиринта, хранящуюся в переменной maze.
    Аргументы markX и markY — это координаты текущего местоположения
    алгоритма '@' в процессе создания лабиринта."""

    for y in range(HEIGHT):
        for x in range(WIDTH):
```

Эквивалент функции `printMaze()`, но на языке JavaScript:

```
function printMaze(maze, markX, markY) {
    // Отображает структуру данных лабиринта, хранящуюся в переменной maze.
    // Аргументы markX и markY — это координаты текущего местоположения
    // алгоритма '@' в процессе создания лабиринта.
    document.write('<code>');
    for (let y = 0; y < HEIGHT; y++) {
        for (let x = 0; x < WIDTH; x++) {
```

Функция `printMaze()` отображает на экране структуру данных лабиринта, присвоенную ей в качестве параметра `maze`. При передаче целочисленных аргументов `markX` и `markY` константа `MARK` (для которой задано значение `@`) отображается в точке с соответствующими координатами x и y при выводе лабиринта на экран. Чтобы гарантировать отображение символов моноширинным шрифтом, версия программы на языке JavaScript использует HTML-тег `<code>` перед выводом лабиринта на экран. Отсутствие данного HTML-тега привело бы к искажению структуры лабиринта, показываемой в браузере.

Внутри функции вложенные циклы `for` перебирают каждую точку в структуре данных лабиринта с координатами y в диапазоне от `0` до (но не включая) `HEIGHT` и x от `0` до (но не включая) `WIDTH`.

Внутри подцикла `for`, если текущие координаты x и y совпадают с координатами метки (точки, обрабатываемой алгоритмом в данный момент), программа отображает значение `@`, содержащееся в константе `MARK`. Код Python делает это следующим образом:

```
            if markX == x and markY == y:
                # Отобразите метку '@' здесь:
                print(MARK, end='')
            else:
                # Отобразите часть стены или пустое пространство:
                print(maze[(x, y)], end='')

        print() # Добавьте символ новой строки после отображения ряда символов
```

Эквивалентный код JavaScript выглядит так:

```
            if (markX === x && markY === y) {
                // Отобразите метку '@' здесь:
                document.write(MARK);
            } else {
                // Отобразите часть стены или пустое пространство:
                document.write(maze[[x, y]]);
            }
        }
        document.write('<br />'); // Добавьте символ новой строки
                                  // после отображения ряда символов
    }
    document.write('</code>');
}
```

В ином случае программа отображает символ константы `WALL` или `EMPTY` в точке с координатами *x* и *y* в структуре данных `maze`, используя код `maze[(x, y)]` в Python и `maze[[x, y]]` в JavaScript. После того как внутренний цикл `for` завершает перебор координат *x*, мы добавляем символ новой строки для создания дополнительного ряда символов.

Использование рекурсивного алгоритма поиска с возвратом

Функция `visit()` реализует рекурсивный алгоритм поиска с возвратом. Эта функция предусматривает список (в Python) или массив (в JavaScript) для отслеживания координат *x* и *y* тех точек, которые уже посещались в ходе предыдущих вызовов `visit()`. Она также на месте изменяет глобальную переменную `maze`, в которой хранится структура данных лабиринта. Код функции `visit()` на языке Python начинается следующим образом:

```
def visit(x, y):
    """Вырезайте пустые пространства в точках с координатами x и y,
    а затем рекурсивно перемещайтесь в соседние, ранее не посещавшиеся точки.
    Эта функция возвращается к предыдущим точкам, когда метка достигает тупика."""
    maze[(x, y)] = EMPTY  # Вырезает пустое пространство в точке
                          # с координатами x и y.
    printMaze(maze, x, y) # Отображает лабиринт по мере его создания
    print('\n\n')
```

Эквивалентный код на языке JavaScript выглядит так:

```
function visit(x, y) {
    // Вырезайте пустые пространства в точках с координатами x и y,
    // а затем рекурсивно перемещайтесь в соседние, ранее не посещавшиеся
    // точки. Эта функция возвращается к предыдущим точкам,
    // когда метка достигает тупика
```

```
maze[[x, y]] = EMPTY; // Вырезает пустое пространство
                      // в точке с координатами x и y
printMaze(maze, x, y); // Отображает лабиринт по мере его создания
document.write('<br /><br /><br />');
```

Функция `visit()` принимает в качестве аргументов координаты x и y той точки лабиринта, которую алгоритм обрабатывает в данный момент. Затем функция модифицирует структуру данных в переменной `maze`, превращая соответствующую точку в пустое пространство. Чтобы позволить пользователю наблюдать за процессом создания лабиринта, она вызывает функцию `printMaze()`, передавая в качестве аргументов x и y координаты текущего положения метки.

Затем рекурсивный алгоритм поиска с возвратом вызывает `visit()` с координатами ранее не исследованной соседней точки. Код данного фрагмента на языке Python выглядит следующим образом:

```
while True:
    # Проверяет, какие из точек, находящихся по соседству с меткой,
    # еще не посещались:
    unvisitedNeighbors = []
    if y > 1 and (x, y - 2) not in hasVisited:
        unvisitedNeighbors.append(NORTH)

    if y < HEIGHT - 2 and (x, y + 2) not in hasVisited:
        unvisitedNeighbors.append(SOUTH)

    if x > 1 and (x - 2, y) not in hasVisited:
        unvisitedNeighbors.append(WEST)

    if x < WIDTH - 2 and (x + 2, y) not in hasVisited:
        unvisitedNeighbors.append(EAST)
```

Эквивалентный код на языке JavaScript выглядит так:

```
while (true) {
    // Проверяет, какие из точек, находящихся по соседству с меткой,
    // еще не посещались
    let unvisitedNeighbors = [];
    if (y > 1 && !JSON.stringify(hasVisited).includes(JSON.
    stringify([x, y - 2]))) {
        unvisitedNeighbors.push(NORTH);
    }
    if (y < HEIGHT - 2 &&
    !JSON.stringify(hasVisited).includes(JSON.stringify([x, y + 2]))) {
        unvisitedNeighbors.push(SOUTH);
    }
    if (x > 1 &&
    !JSON.stringify(hasVisited).includes(JSON.stringify([x - 2, y]))) {
        unvisitedNeighbors.push(WEST);
    }
```

```
if (x < WIDTH - 2 &&
!JSON.stringify(hasVisited).includes(JSON.stringify([x + 2, y]))) {
    unvisitedNeighbors.push(EAST);
}
```

Цикл `while` выполняется до тех пор, пока у текущей точки в лабиринте остаются непосещенные соседи. Мы создаем список или массив этих неисследованных позиций в переменных `unvisitedNeighbors`. Четыре оператора `if` проверяют, не находится ли текущая точка с координатами x и y на границе лабиринта (то есть имеются ли по соседству с ней доступные для проверки объекты) и не присутствуют ли координаты x и y смежных точек в списке или в массиве `hasVisited`.

Если все прилегающие позиции уже посещались, функция завершает свою работу, и алгоритм возвращается в предыдущее положение. Код Python проверяет условия базового случая так:

```
if len(unvisitedNeighbors) == 0:
    # БАЗОВЫЙ СЛУЧАЙ
    # Все соседние точки были посещены, достигнут тупик.
    # Вернитесь в предыдущую точку:
    return
```

Код JavaScript делает то же самое следующим образом:

```
if (unvisitedNeighbors.length === 0) {
    // БАЗОВЫЙ СЛУЧАЙ
    // Все соседние точки были посещены, достигнут тупик.
    // Вернитесь в предыдущую точку:
    return;
```

Рекурсивный алгоритм поиска с возвратом достигает базового случая, когда у текущей точки не остается непосещенных соседей. В таком случае функция просто прекращает свою работу. Сама функция `visit()` не возвращает никакого значения. Вместо этого рекурсивная функция вызывает `visit()` для изменения структуры данных лабиринта в глобальной переменной `maze` в качестве побочного эффекта. После возврата из первоначального вызова функции `maze()` глобальная переменная `maze` содержит полностью сгенерированный лабиринт.

Код Python для рекурсивного случая выглядит следующим образом:

```
else:
    # РЕКУРСИВНЫЙ СЛУЧАЙ
    # Случайным образом выберите непосещенную соседнюю точку для посещения:
    nextIntersection = random.choice(unvisitedNeighbors)

    # Переместите метку в непосещенную соседнюю точку:

    if nextIntersection == NORTH:
        nextX = x
        nextY = y - 2
```

```
            maze[(x, y - 1)] = EMPTY # Развилка
        elif nextIntersection == SOUTH:
            nextX = x
            nextY = y + 2
            maze[(x, y + 1)] = EMPTY # Развилка
        elif nextIntersection == WEST:
            nextX = x - 2
            nextY = y
            maze[(x - 1, y)] = EMPTY # Развилка
        elif nextIntersection == EAST:
            nextX = x + 2
            nextY = y
            maze[(x + 1, y)] = EMPTY # Развилка

        hasVisited.append((nextX, nextY)) # Отметьте точку как посещенную
        visit(nextX, nextY) # Рекурсивно посещайте эту точку
```

Эквивалентный код на языке JavaScript выглядит так:

```
        } else {
            // РЕКУРСИВНЫЙ СЛУЧАЙ
            // Случайным образом выберите непосещенную соседнюю точку для посещения:
            let nextIntersection = unvisitedNeighbors[
            Math.floor(Math.random() * unvisitedNeighbors.length)];

            // Переместите метку в непосещенную соседнюю точку:
            let nextX, nextY;
            if (nextIntersection === NORTH) {
                nextX = x;
                nextY = y - 2;
                maze[[x, y - 1]] = EMPTY; // Развилка
            } else if (nextIntersection === SOUTH) {
                nextX = x;
                nextY = y + 2;
                maze[[x, y + 1]] = EMPTY; // Развилка
            } else if (nextIntersection === WEST) {
                nextX = x - 2;
                nextY = y;
                maze[[x - 1, y]] = EMPTY; // Развилка
            } else if (nextIntersection === EAST) {
                nextX = x + 2;
                nextY = y;
                maze[[x + 1, y]] = EMPTY; // Развилка
            }
            hasVisited.push([nextX, nextY]); // Отметьте точку как посещенную
            visit(nextX, nextY); // Рекурсивно посещайте эту точку
        }
    }
}
```

Список или массив `unvisitedNeighbors` содержит одну или несколько констант, определяющих направление, — `NORTH`, `SOUTH`, `WEST` и `EAST`. Мы выбираем какую-либо

из них для следующего рекурсивного вызова `visit()`, а затем задаем для переменных `nextX` и `nextY` значения координат соседней точки с учетом выбранного маршрута.

Следом добавляем координаты *x* и *y*, содержащиеся в `nextX` и `nextY`, в список или массив `hasVisited` перед выполнением рекурсивного вызова для соответствующей смежной точки. Таким образом, функция `visit()` продолжает обходить соседние точки, вырезая коридоры лабиринта путем изменения значения посещаемого объекта в `maze` на `EMPTY`. Для коридора, соединяющего текущую позицию с соседней, также задается значение `EMPTY`.

При отсутствии соседних точек, то есть при достижении базового случая, алгоритм просто возвращается в предыдущее положение. В функции `visit()` выполнение программы возвращается к началу цикла `while`. Код в цикле `while` снова проверяет, какие из близлежащих точек еще не просматривались, и выполняет рекурсивный вызов `visit()` для одного из них или завершает свою работу при отсутствии непосещенных точек.

После вырезания коридоров лабиринта и исследования всех точек выполняются возвраты из рекурсивных вызовов. В конечном итоге происходит возврат из первоначального вызова функции `visit()`, к моменту которого переменная `maze` содержит полностью сгенерированный лабиринт.

Запуск цепочки рекурсивных вызовов

Рекурсивная функция `visit()` использует две глобальные переменные — `maze` и `hasVisited`. Переменная `hasVisited` представляет собой список или массив, содержащий координаты *x* и *y* всех посещенных алгоритмом точек, начиная с начальной `(1, 1)`. Код данного фрагмента программы на языке Python выглядит следующим образом:

```
# Вырезание путей в структуре данных лабиринта:
hasVisited = [(1, 1)] # Начните с посещения левого верхнего угла
visit(1, 1)

# Отобразите итоговую структуру данных лабиринта:
printMaze(maze)
```

Эквивалентный код на языке JavaScript выглядит так:

```
// Вырезание путей в структуре данных лабиринта:
let hasVisited = [[1, 1]]; // Начните с посещения левого верхнего угла
visit(1, 1);

// Отобразите итоговую структуру данных лабиринта:
printMaze(maze);
</script>
```

После добавления в переменную `hasVisited` координат `(1, 1)` (верхний левый угол лабиринта) мы вызываем `visit()`, передавая ей текущие координаты. Вызов данной функции запустит цепочку рекурсивных вызовов, которые сгенерируют коридоры лабиринта. К моменту возврата из этого вызова переменная `hasVisited` будет содержать координаты всех точек лабиринта, а переменная `maze` — полностью сформированный лабиринт.

Резюме

Теперь вы знаете, что рекурсию можно использовать не только для прохождения лабиринтов (путем обхода их древовидных структур данных), но и для их формирования с помощью рекурсивного алгоритма поиска с возвратом. Этот алгоритм вырезает коридоры в лабиринте, возвращаясь к предыдущим точкам при достижении тупика. К моменту возвращения алгоритма в исходное положение лабиринт оказывается полностью сгенерированным.

Можно представить односвязный лабиринт без замкнутых маршрутов в виде такой древовидной структуры данных, как ориентированный ациклический граф. А поскольку рекурсия хорошо подходит для решения задач, предполагающих использование древовидных структур данных и поиск с возвратом, мы можем применить для его создания соответствующий рекурсивный алгоритм.

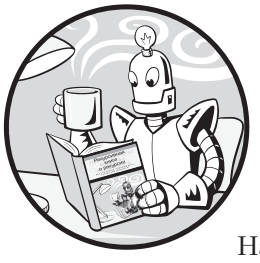
Дополнительные источники информации

В «Википедии» есть статья о создании лабиринтов, в которой также описывается рекурсивный алгоритм поиска с возвратом: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_backtracker. Чтобы продемонстрировать процесс вырезания коридоров лабиринта с помощью рекурсивного алгоритма, я создал браузерную анимацию, которая доступна по ссылке https://scratch.mit.edu/projects/17358777.

Если вас интересует тема создания лабиринтов, прочитайте книгу Джемиса Бака (Jamis Buck) *Mazes for Programmers: Code Your Own Twisty Little Passages* (Pragmatic Bookshelf, 2015).

12

Решатель «пятнашек»



Головоломка под названием «пятнашки» представляет собой квадрат 4 × 4 и 15 пронумерованных прямоугольных костяшек, помещенных в него.

На месте отсутствующей 16-й костяшки — пустая область, в которую необходимо сдвигать соседние фигуры. Цель игрока — упорядочить костяшки по возрастанию их номеров, как показано на рис. 12.1. В некоторых версиях такой игры на костяшках нанесены фрагменты изображения, которые образуют цельную картину после решения (составления) головоломки.

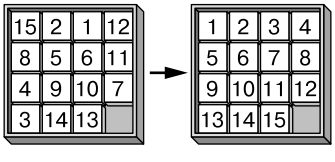


Рис. 12.1. Головоломка «пятнашки» в перемешанном *(слева)* и упорядоченном состоянии *(справа)*

Математики доказали, что даже самую сложную комбинацию «пятнашек» можно упорядочить за 80 ходов.

Рекурсивный алгоритм решения «пятнашек»

Алгоритм решения «пятнашек» напоминает алгоритм прохождения лабиринта. Каждую конфигурацию (то есть расположение) костяшек можно рассматривать в качестве развилки в лабиринте. Перемещение костяшки в одном из четырех направлений похоже на выбор коридора, ведущего к следующей развилке.

Как и в случае с лабиринтом, который олицетворяет ориентированный ациклический граф (DAG), головоломка «пятнашки» представляет собой древовидный граф, изображенный на рис. 12.2. Состояния головоломки — это вершины, которые предусматривают до четырех ребер (соответствующих направлению перемещения костяшки), связывающих их с другими узлами (то есть результирующими состояниями). Корневой узел соответствует начальному состоянию головоломки, а конечный узел — состоянию решенной головоломки. Путь от корневого узла к конечному — это последовательность перемещений костяшек, позволяющая решить задачу.

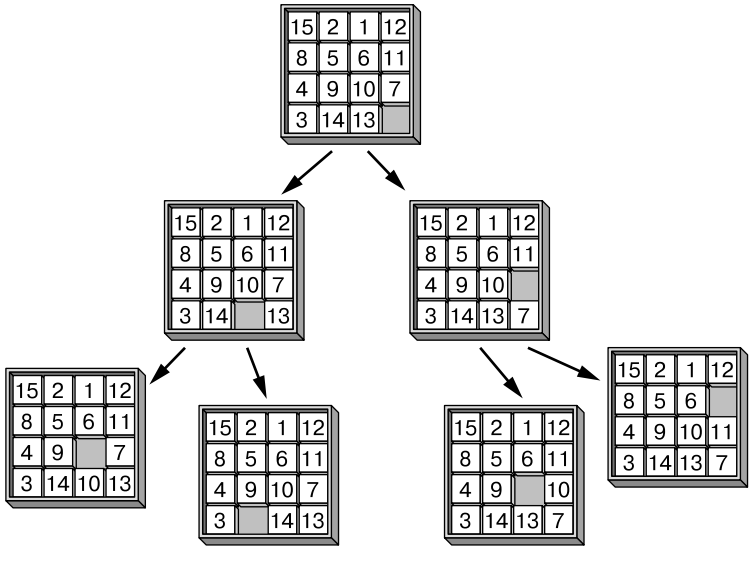


Рис. 12.2. Процесс решения «пятнашек» может быть представлен в виде графа, узлы которого соответствуют состояниям головоломки, а ребра — вариантам перемещения костяшек

Для решения «пятнашек» существуют изощренные алгоритмы, однако вместо их применения мы можем просто рекурсивно исследовать весь древовидный граф для нахождения пути, ведущего от корневого узла к вершине, соответствующей состоянию решенной головоломки. Это можно сделать с помощью алгоритма поиска в глубину. Однако в отличие от лабиринта древовидный граф «пятнашек» не ациклический, а скорее *ненаправленный*, так как мы можем двигаться вдоль ребер в обоих направлениях, отменяя то или иное перемещение костяшки.

На рис. 12.3 показан пример ненаправленного ребра, связывающего два узла. Поскольку между ними можно перемещаться бесконечно, наш алгоритм решения «пятнашек» подвержен риску переполнения стека.

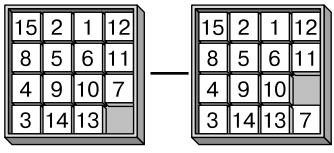


Рис. 12.3. Ребра древовидного графа «пятнашек» не являются направленными (рисуются без стрелки), потому что перемещение костяшки можно отменить, вернув ее в исходное положение

Чтобы оптимизировать алгоритм, стоит избегать перемещений костяшек, отменяющих предыдущий ход. Однако подобная оптимизация сама по себе не предотвратит переполнение стека. Несмотря на то что она позволяет сделать *ребра* в древовидном графе направленными, она не превращает граф алгоритма для решения «пятнашек» в DAG, потому что он имеет петли, связывающие нижние узлы с верхними. Такие петли возникают при перемещении костяшки по кругу, как показано на рис. 12.4.

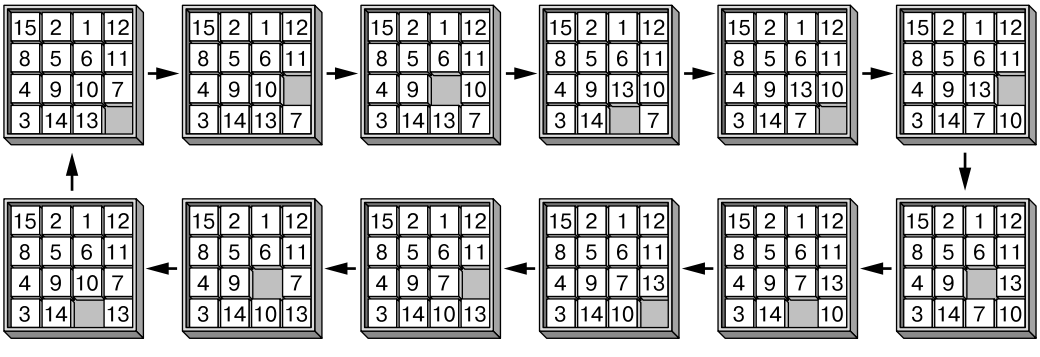


Рис. 12.4. Пример петли в графе алгоритма для решения «пятнашек»

Наличие цикличности в графе означает, что алгоритм в состоянии возвращаться от нижних узлов к верхним. При этом он может застрять в таком замкнутом цикле и никогда не исследовать ветвь, соответствующую решению головоломки. На практике это привело бы к переполнению стека.

Тем не менее использовать рекурсию для решения «пятнашек» вполне реально. Просто нужно предусмотреть базовый случай с указанием максимального количества ходов, чтобы избежать переполнения стека. После достижения установленного предельного значения алгоритм возвратится к более ранним узлам. Если задача не решится за десять попыток, программа предпримет еще одну — одиннадцатую. Если и этого не хватит, она попробует 12 ходов и т. д. Такой маневр предотвратит попадание алгоритма в бесконечный цикл и гарантирует продолжение поиска возможных решений, предусматривающих другое предельное количество ходов.

Полный код программы для решения «пятнашек»

Начнем с рассмотрения полного кода программы для решения «пятнашек». В оставшейся части главы каждый раздел кода будет рассмотрен отдельно.

Скопируйте код на языке Python в файл с именем `slidingTileSolver.py`:

```
import random, time

DIFFICULTY = 40 # Число случайных перемещений костяшек для получения
                # начального состояния головоломки.
SIZE = 4 # Игровое поле размером SIZE x SIZE
random.seed(1) # Выбор головоломки для решения

BLANK = 0
UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'

def displayBoard(board):
    """Отображение на экране костяшек, хранящихся в переменной 'board'."""
    for y in range(SIZE): # Итерация всех строк
        for x in range(SIZE): # Итерация всех столбцов
            if board[y * SIZE + x] == BLANK:
                print('__ ', end='') # Отображение пустой области
            else:
                print(str(board[y * SIZE + x]).rjust(2) + ' ', end='')
        print() # Вывод новой строки в конце строки

def getNewBoard():
    """ Возврат списка, представляющего новую головоломку."""
    board = []
    for i in range(1, SIZE * SIZE):
        board.append(i)
    board.append(BLANK)
    return board

def findBlankSpace(board):
    """Возврат списка координат [x, y] пустой области."""
    for x in range(SIZE):
        for y in range(SIZE):
            if board[y * SIZE + x] == BLANK:
                return [x, y]
def makeMove(board, move):
    """Изменение переменной 'board' на месте для выполнения перемещения 'move'."""
    bx, by = findBlankSpace(board)
    blankIndex = by * SIZE + bx

    if move == UP:
        tileIndex = (by + 1) * SIZE + bx
```

```
    elif move == LEFT:
        tileIndex = by * SIZE + (bx + 1)
    elif move == DOWN:
        tileIndex = (by - 1) * SIZE + bx
    elif move == RIGHT:
        tileIndex = by * SIZE + (bx - 1)

    # Меняем местами костяшки  в положениях с индексами blankIndex и tileIndex:
    board[blankIndex], board[tileIndex] = board[tileIndex], board[blankIndex]
def undoMove(board, move):
    # Осуществляем перемещение, противоположное 'move', чтобы отменить его
    # на поле 'board'.
    if move == UP:
        makeMove(board, DOWN)
    elif move == DOWN:
        makeMove(board, UP)
    elif move == LEFT:
        makeMove(board, RIGHT)
    elif move == RIGHT:
        makeMove(board, LEFT)
def getValidMoves(board, prevMove=None):
    """Возврат списка допустимых ходов для этого поля. Если указано
    значение prevMove, ход, который может его отменить, не применяется."""

    blankx, blanky = findBlankSpace(board)

    validMoves = []
    if blanky != SIZE - 1 and prevMove != DOWN:
        # В нижней строке нет пустой области
        validMoves.append(UP)

    if blankx != SIZE - 1 and prevMove != RIGHT:
        # В правом столбце нет пустой области
        validMoves.append(LEFT)

    if blanky != 0 and prevMove != UP:
        # В верхней строке нет пустой области
        validMoves.append(DOWN)

    if blankx != 0 and prevMove != LEFT:
        # В левом столбце нет пустой области
        validMoves.append(RIGHT)

    return validMoves
def getNewPuzzle():
    # Получите новую головоломку, выполнив случайные перемещения
    # из решенного состояния
    board = getNewBoard()
    for i in range(DIFFICULTY):
        validMoves = getValidMoves(board)
```

```
        makeMove(board, random.choice(validMoves))
    return board

def solve(board, maxMoves):
    """Пробуем решить головоломку, содержащуюся в переменной 'board',
    не более чем за 'maxMoves' ходов. Возвращается True,
    если головоломка решена, в противном случае — False."""
    print('Attempting to solve in at most', maxMoves, 'moves...')
    solutionMoves = [] # Список значений UP, DOWN, LEFT, RIGHT.
    solved = attemptMove(board, solutionMoves, maxMoves, None)

    if solved:
        displayBoard(board)
        for move in solutionMoves:
            print('Move', move)
            makeMove(board, move)
            print() # Переход на новую строку
            displayBoard(board)
            print() # Переход на новую строку

        print('Solved in', len(solutionMoves), 'moves:')
        print(', '.join(solutionMoves))
        return True # Головоломка была решена
    else:
        return False # Головоломку невозможно решить за maxMoves ходов

def attemptMove(board, movesMade, movesRemaining, prevMove):
    """Рекурсивная функция, которая пробует все возможные ходы
    на поле 'board', пока не найдет решение или не достигнет предельного
    числа ходов 'maxMoves'. Возвращает True, если решение было найдено.
    В этом случае 'maxMoves' содержит последовательность ходов для решения
    головоломки. Возвращает False, если значение 'movesRemaining' меньше 0."""

    if movesRemaining < 0:
        # БАЗОВЫЙ СЛУЧАЙ — закончились допустимые ходы
        return False

    if board == SOLVED_BOARD:
        # БАЗОВЫЙ СЛУЧАЙ — головоломка решена
        return True

    # РЕКУРСИВНЫЙ СЛУЧАЙ — проверка каждого из допустимых ходов:
    for move in getValidMoves(board, prevMove):
        # Делаем ход:
        makeMove(board, move)
        movesMade.append(move)

        if attemptMove(board, movesMade, movesRemaining - 1, move):
            # Если головоломка решена, возвращается True:
            undoMove(board, move) # Восстановление исходного состояния головоломки
            return True
```

```
        # Отмена хода, чтобы подготовиться к совершению следующего хода:
        undoMove(board, move)
        movesMade.pop() # Удаление последнего хода, который был отменен
    return False # БАЗОВЫЙ СЛУЧАЙ — найти решение невозможно

# Запуск программы:
SOLVED_BOARD = getNewBoard()
puzzleBoard = getNewPuzzle()
displayBoard(puzzleBoard)
startTime = time.time()

maxMoves = 10
while True:
    if solve(puzzleBoard, maxMoves):
        break # Выход из цикла при нахождении решения
    maxMoves += 1
print('Run in', round(time.time() - startTime, 3), 'seconds.')
```

Скопируйте код на языке JavaScript в файл с именем `slidingTileSolver.html`:

```
<script type="text/javascript">
const DIFFICULTY = 40; // Число случайных перемещений костяшек для получения
                       // начального состояния головоломки
const SIZE = 4; // Игровое поле размером SIZE x SIZE

const BLANK = 0;
const UP = "up";
const DOWN = "down";
const LEFT = "left";
const RIGHT = "right";

function displayBoard(board) {
    // Отображение на экране костяшек, хранящихся в переменной 'board'.
    document.write("<pre>");
    for (let y = 0; y < SIZE; y++) { // Итерация всех строк
        for (let x = 0; x < SIZE; x++) { // Итерация всех столбцов
            if (board[y * SIZE + x] == BLANK) {
                document.write('__ '); // Отображение пустой области
            } else {
                document.write(board[y * SIZE + x].toString().padStart(2) + " ");
            }
        }
        document.write("<br />"); // Вывод новой строки в конце ряда
    }
    document.write("</pre>");
}

function getNewBoard() {
    // Возврат списка, представляющего новую головоломку
    let board = [];
    for (let i = 1; i < SIZE * SIZE; i++) {
```

```
        board.push(i);
    }
    board.push(BLANK);
    return board;
}
function findBlankSpace(board) {
    // Возврат массива координат [x, y] пустой области
    for (let x = 0; x < SIZE; x++) {
        for (let y = 0; y < SIZE; y++) {
            if (board[y * SIZE + x] === BLANK) {
                return [x, y];
            }
        }
    }
}

function makeMove(board, move) {
    // Изменение переменной 'board' на месте для выполнения перемещения 'move'
    let bx, by;
    [bx, by] = findBlankSpace(board);
    let blankIndex = by * SIZE + bx;

    let tileIndex;
    if (move === UP) {
        tileIndex = (by + 1) * SIZE + bx;
    } else if (move === LEFT) {
        tileIndex = by * SIZE + (bx + 1);
    } else if (move === DOWN) {
        tileIndex = (by - 1) * SIZE + bx;
    } else if (move === RIGHT) {
        tileIndex = by * SIZE + (bx - 1);
    }

    // Меняем местами костяшки  в положениях с индексами blankIndex и tileIndex:
    [board[blankIndex], board[tileIndex]] = [board[tileIndex],
      board[blankIndex]];
}
function undoMove(board, move) {
    // Выполняем перемещение, противоположное 'move', чтобы отменить его
    // на поле 'board'
    if (move === UP) {
        makeMove(board, DOWN);
    } else if (move === DOWN) {
        makeMove(board, UP);
    } else if (move === LEFT) {
        makeMove(board, RIGHT);
    } else if (move === RIGHT) {
        makeMove(board, LEFT);
    }
}
```

```
function getValidMoves(board, prevMove) {
    // Возврат списка допустимых ходов для этого поля. Если указано
    // значение prevMove, ход, который может его отменить, не применяется.

    let blankx, blanky;
    [blankx, blanky] = findBlankSpace(board);

    let validMoves = [];
    if (blanky != SIZE - 1 && prevMove != DOWN) {
        // В нижней строке нет пустой области
        validMoves.push(UP);
    }
    if (blankx != SIZE - 1 && prevMove != RIGHT) {
        // В правом столбце нет пустой области
        validMoves.push(LEFT);
    }
    if (blanky != 0 && prevMove != UP) {
        // В верхней строке нет пустой области
        validMoves.push(DOWN);
    }
    if (blankx != 0 && prevMove != LEFT) {
        // В левом столбце нет пустой области
        validMoves.push(RIGHT);
    }
    return validMoves;
}
function getNewPuzzle() {
    // Получаем новую головоломку, выполнив случайные перемещения
    // из решенного состояния
    let board = getNewBoard();
    for (let i = 0; i < DIFFICULTY; i++) {
        let validMoves = getValidMoves(board);
        makeMove(board, validMoves[Math.floor(Math.random() *
          validMoves.length)]);
    }
    return board;
}

function solve(board, maxMoves) {
    // Пробуем решить головоломку, содержащуюся в переменной 'board',
    // не более чем за 'maxMoves' ходов. Возвращается True, если головоломка
    // решена, в противном случае — False
    document.write("Attempting to solve in at most " + maxMoves + " moves...<br />");
    let solutionMoves = []; // Список значений UP, DOWN, LEFT, RIGHT.
    let solved = attemptMove(board, solutionMoves, maxMoves, null);

    if (solved) {
        displayBoard(board);
        for (let move of solutionMoves) {
            document.write("Move " + move + "<br />");
            makeMove(board, move);
```

```
            document.write("<br />"); // Переход на новую строку
            displayBoard(board);
            document.write("<br />"); // Переход на новую строку
        }
        document.write("Solved in " + solutionMoves.length + " moves:<br />");
        document.write(solutionMoves.join(", ") + "<br />");
        return true; // Головоломка была решена
    } else {
        return false; // Головоломку невозможно решить за maxMoves ходов
    }
}
function attemptMove(board, movesMade, movesRemaining, prevMove) {
    // Рекурсивная функция, которая пробует все возможные ходы на поле
    // 'board', пока не найдет решение или не достигнет предельного числа
    // ходов 'maxMoves'. Возвращает True, если решение было найдено. В этом
    // случае 'maxMoves' содержит последовательность ходов для решения
    // головоломки. Возвращает False, если значение 'movesRemaining' меньше 0

    if (movesRemaining < 0) {
        // БАЗОВЫЙ СЛУЧАЙ – закончились допустимые ходы
        return false;
    }
    if (JSON.stringify(board) == SOLVED_BOARD) {
        // БАЗОВЫЙ СЛУЧАЙ – головоломка решена
        return true;
    }
    // РЕКУРСИВНЫЙ СЛУЧАЙ – проверка каждого из допустимых ходов:
    for (let move of getValidMoves(board, prevMove)) {
        // Сделайте ход:
        makeMove(board, move);
        movesMade.push(move);

        if (attemptMove(board, movesMade, movesRemaining - 1, move)) {
            // Если головоломка решена, возвращается True:
            undoMove(board, move); // Восстановление исходного состояния головоломки
            return true;
        }

        // Отмена хода, чтобы подготовиться к совершению следующего хода:
        undoMove(board, move);
        movesMade.pop(); // Удаление последнего хода, который был отменен
    }
    return false; // БАЗОВЫЙ СЛУЧАЙ – найти решение невозможно
}

// Запустите программу:
const SOLVED_BOARD = JSON.stringify(getNewBoard());
let puzzleBoard = getNewPuzzle();
displayBoard(puzzleBoard);
let startTime = Date.now();
```

```
let maxMoves = 10;
while (true) {
    if (solve(puzzleBoard, maxMoves)) {
        break; // Выход из цикла при нахождении решения
    }
    maxMoves += 1;
}
document.write("Run in " + Math.round((Date.now() - startTime) / 100) /
  10 + " seconds.<br />");
</script>
```

Результат запуска программы выглядит следующим образом:

```
 7  1  3  4
 2  5 10  8
__  6  9 11
13 14 15 12
Attempting to solve in at most 10 moves...
Attempting to solve in at most 11 moves...
Attempting to solve in at most 12 moves...
-- пропущенный фрагмент --
 1  2  3  4
 5  6  7  8
 9 10 11 __
13 14 15 12

Move up

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 __

Solved in 18 moves:
left, down, right, down, left, up, right, up, left, left, down,
right, right, up, left, left, left, up
Run in 39.519 seconds.
```

Обратите внимание, что при запуске кода JavaScript в браузере он должен будет завершить работу, прежде чем на экране отобразятся какие-либо выходные данные. Вам может показаться, что программа зависла, а ваш браузер предложит остановить ее выполнение. Проигнорируйте соответствующее предупреждение и позвольте приложению работать до тех пор, пока оно не разгадает головоломку.

Рекурсивная функция `attemptMove()` нашей программы решает головоломку, перебирая все возможные конфигурации костяшек. Функции передается ход, и, если он позволяет решить загадку, возвращается логическое значение `True`. В противном случае вызывается `attemptMove()` для проверки остальных возможных ходов и возвращается логическое значение `False`, если ни один из них не позволяет найти решение за допустимое количество ходов. Рассмотрим эту функцию более подробно чуть позже.

Структура данных, которую мы используем для создания игрового поля, представляет собой список (в Python) или массив целых чисел (в JavaScript), где `0` соответствует пустой области. В текущей программе эта структура данных хранится в переменной `board`. Индекс `board[y * SIZE + x]` соответствует точке игрового поля с координатами x и y, как показано на рис. 12.5. Например, если значение константы `SIZE` равно `4`, то точка с координатами $x = 3$ и $y = 1$ соответствует положению с индексом `board[1 * 4 + 3]`.

Это небольшое вычисление позволяет использовать одномерный массив или список для хранения местоположений на двумерном игровом поле. Данный метод программирования подходит не только для нашего проекта, его также можно применить к любой двумерной структуре данных, которую необходимо сохранить в массиве или списке, например к двумерному изображению, хранимому в виде потока байтов.

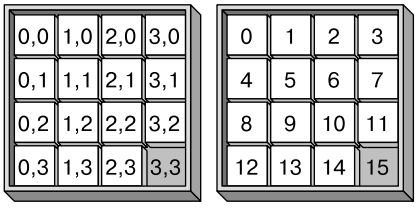


Рис. 12.5. Координаты x и y каждого местоположения на игровом поле *(слева)* и соответствующий индекс структуры данных *(справа)*

Рассмотрим несколько примеров структур данных. Игровое поле с перемешанными костяшками, показанное в левой части рис. 12.1, будет представлено следующим образом:

```
[15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, 0]
```

Решенная головоломка из правой части рис. 12.1 будет представлена в виде:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
```

Все функции в нашей программе будут ожидать получения структур данных именно в таком формате.

К сожалению, версия «пятнашек» с игровым полем 4×4 предусматривает так много возможных ходов, что на ее решение обычному ноутбуку могут потребоваться недели. Допускается изменить значение константы `SIZE` с `4` на `3`, чтобы решить более простую версию головоломки с игровым полем 3×3. Структура данных такой решенной головоломки будет выглядеть следующим образом:

```
[1, 2, 3, 4, 5, 6, 7, 8, 0].
```

Задание констант программы

В самом начале программы задаются значения нескольких констант, чтобы сделать код более удобочитаемым. На языке Python это выглядит так:

```
import random, time

DIFFICULTY = 40 # Число случайных перемещений костяшек для получения
                # начального состояния головоломки
SIZE = 4 # Игровое поле размером SIZE x SIZE
random.seed(1) # Выбор головоломки для решения

BLANK = 0
UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'
```

Эквивалентный код на языке JavaScript выглядит следующим образом:

```
<script type="text/javascript">
const DIFFICULTY = 40; // Число случайных перемещений костяшек для получения
                       // начального состояния головоломки
const SIZE = 4; // Игровое поле размером SIZE x SIZE

const BLANK = 0;
const UP = "up";
const DOWN = "down";
const LEFT = "left";
const RIGHT = "right";
```

Для воспроизведения одной и той же последовательности случайных чисел программа на языке Python использует начальное значение (сид), равное `1`. Один и тот же сид всегда воспроизводит одинаковую «случайную» головоломку, что бывает полезно при отладке программы. Вы можете изменить начальное значение на любое целое число, чтобы сгенерировать другие головоломки. JavaScript не предусматривает возможности задания сида, поэтому в файле `slidingTileSolver.html` нет соответствующей функции.

Константа `SIZE` задает размер игрового поля. Вы можете изменить ее значение, однако стандартным является зона 4×4, тогда как 3×3 удобнее при тестировании, поскольку программа решает такие головоломки довольно быстро. Константа `BLANK` используется в структуре данных головоломки для представления пустой области, и ее значение всегда должно быть равно `0`. Константы `UP`, `DOWN`, `LEFT` и `RIGHT` применяются для того, чтобы сделать код более понятным, по аналогии с константами `NORTH`, `SOUTH`, `WEST` и `EAST` в программе для создания лабиринтов в главе 11.

Представление головоломки «пятнашки» в виде данных

Структура данных игрового поля «пятнашек» — это простой список или массив целых чисел. Ее способность отображать игровое поле обусловлена тем, как она используется различными функциями программы, в частности `displayBoard()`, `getNewBoard()` и `findBlankSpace()`.

Отображение игрового поля

Первая функция, `displayBoard()`, выводит на экран структуру данных игровой площадки. Код функции `displayBoard()` на языке Python выглядит следующим образом:

```
def displayBoard(board):
    """Отображение на экране костяшек, хранящихся в переменной 'board'."""
    for y in range(SIZE): # Итерация всех строк
        for x in range(SIZE): # Итерация всех столбцов
            if board[y * SIZE + x] == BLANK:
                print('__ ', end='') # Отображение пустой области
            else:
                print(str(board[y * SIZE + x]).rjust(2) + ' ', end='')
        print() # Вывод новой строки в конце строки.
```

Код JavaScript для функции `displayBoard()` выглядит так:

```
function displayBoard(board) {
    // Отображение на экране костяшек, хранящихся в переменной 'board'
    document.write("<pre>");
    for (let y = 0; y < SIZE; y++) { // Итерация всех строк
        for (let x = 0; x < SIZE; x++) { // Итерация всех столбцов
            if (board[y * SIZE + x] == BLANK) {
                document.write('__ '); // Отображение пустой области
            } else {
                document.write(board[y * SIZE + x].toString().padStart(2) + " ");
            }
        }
        document.write("<br />"); // Вывод новой строки в конце ряда
    }
    document.write("</pre>");
}
```

Пара вложенных циклов `for` просматривает все строки и столбцы игрового поля. Первый цикл `for` перебирает координаты *y*, а второй цикл `for` — координаты *x*. Это связано с тем, что программе необходимо отобразить все столбцы на одной строке, прежде чем добавить символ новой строки и перейти на следующую.

Оператор `if` проверяет, соответствуют ли текущие координаты *x* и *y* положению пустой области. Если да, то программа выводит на экран два символа нижнего подчеркивания с пробелом в конце. В противном случае код в блоке `else` выводит на экран номер костяшки с пробелом в конце. Этот пробел отделяет номера костяшек друг от друга на экране. Если номер костяшки состоит из одной цифры, метод `rjust()` или `padStart()` вставляет дополнительный пробел, чтобы выровнять на экране однозначные номера относительно двузначных.

Например, состояние головоломки, изображенной слева на рис. 12.1, представлено следующей структурой данных:

```
[15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, 0]
```

Когда эта структура передается функции `displayBoard()`, она отображает на экране следующий текст:

```
15  2  1 12
 8  5  6 11
 4  9 10  7
 3 14 13 __
```

Создание новой структуры данных игрового поля

Затем функция `getNewBoard()` возвращает новую структуру данных игрового поля, соответствующую состоянию решенной головоломки. Код Python для функции `getNewBoard()` выглядит следующим образом:

```
def getNewBoard():
    """ Возврат списка, представляющего новую головоломку."""
    board = []
    for i in range(1, SIZE * SIZE):
        board.append(i)
    board.append(BLANK)
    return board
```

Код JavaScript для функции `getNewBoard()` выглядит так:

```
function getNewBoard() {
    // Возврат списка, представляющего новую головоломку
    let board = [];
    for (let i = 1; i < SIZE * SIZE; i++) {
        board.push(i);
    }
    board.push(BLANK);
    return board;
}
```

Функция `getNewBoard()` возвращает структуру данных игрового поля, соответствующую целочисленному значению константы `SIZE` (то есть 3×3 или 4×4). Цикл `for` генерирует список или массив целых чисел от `1` до (но не включая) возведенного в квадрат значения `SIZE` и помещает в его конец значение `0` (хранящееся в константе `BLANK`), которое представляет пустую область в правом нижнем углу игрового поля.

Нахождение координат пустого квадрата

Для нахождения координат x и y пустой области игрового поля наша программа использует функцию `findBlankSpace()`, код которой на языке Python выглядит так:

```
def findBlankSpace(board):
    """Возврат списка координат [x, y] пустой области."""
    for x in range(SIZE):
        for y in range(SIZE):
            if board[y * SIZE + x] == BLANK:
                return [x, y]
```

Аналогичный код JavaScript выглядит следующим образом:

```
function findBlankSpace(board) {
    // Возврат массива координат [x, y] пустой области
    for (let x = 0; x < SIZE; x++) {
        for (let y = 0; y < SIZE; y++) {
            if (board[y * SIZE + x] === BLANK) {
                return [x, y];
            }
        }
    }
}
```

Как и `displayBoard()`, функция `findBlankSpace()` предусматривает два вложенных цикла `for`, которые перебирают все позиции в структуре данных игрового поля. Когда код `board[y * SIZE + x]` обнаруживает пустую область, он возвращает соответствующие координаты x и y в виде двух целых чисел в списке Python или массиве JavaScript.

Совершение хода

Затем функция `makeMove()` принимает два аргумента: структуру данных игрового поля и направление перемещения костяшки `UP` (вверх), `DOWN` (вниз), `LEFT` (влево) или `RIGHT` (вправо). Этот код содержит много повторений, поэтому в нем используются короткие имена переменных `bx` и `by`, содержащих координаты x и y пустой области.

При совершении хода значение перемещаемой костяшки в структуре данных игровой площадки меняется на `0`, что соответствует пустой области. Код Python для функции `makeMove()` выглядит следующим образом:

```
def makeMove(board, move):
    """Изменение переменной 'board' на месте для выполнения перемещения 'move'."""
    bx, by = findBlankSpace(board)
    blankIndex = by * SIZE + bx

    if move == UP:
        tileIndex = (by + 1) * SIZE + bx
    elif move == LEFT:
        tileIndex = by * SIZE + (bx + 1)
    elif move == DOWN:
        tileIndex = (by - 1) * SIZE + bx
    elif move == RIGHT:
        tileIndex = by * SIZE + (bx - 1)

    # Меняем местами костяшки в положениях с индексами blankIndex и tileIndex:
    board[blankIndex], board[tileIndex] = board[tileIndex],
      board[blankIndex]
```

Код JavaScript для функции `makeMove()` выглядит так:

```
function makeMove(board, move) {
    // Изменение переменной 'board' на месте для выполнения перемещения 'move'
    let bx, by;
    [bx, by] = findBlankSpace(board);
    let blankIndex = by * SIZE + bx;
    let tileIndex;
    if (move === UP) {
        tileIndex = (by + 1) * SIZE + bx;
    } else if (move === LEFT) {
        tileIndex = by * SIZE + (bx + 1);
    } else if (move === DOWN) {
        tileIndex = (by - 1) * SIZE + bx;
    } else if (move === RIGHT) {
        tileIndex = by * SIZE + (bx - 1);
    }

    // Меняем местами костяшки в положениях с индексами blankIndex и tileIndex:
    [board[blankIndex], board[tileIndex]] = [board[tileIndex],
      board[blankIndex]];
}
```

Операторы `if` определяют индекс перемещаемой костяшки на основе значения параметра `move`. Затем функция «сдвигает» костяшку, заменяя значение `BLANK` в позиции `board[blankIndex]` номером костяшки в позиции `board[tileIndex]`. Функция `makeMove()` не возвращает никакого значения, а сразу изменяет структуру данных `board`.

Чтобы поменять местами значения двух переменных, язык Python использует синтаксис `a, b = b, a`. В JavaScript для этого значения нужно поместить в массив, например `[a, b] = [b, a]`. Мы используем данный синтаксис в конце кода функции, чтобы поменять местами значения в `board[blankIndex]` и `board[tileIndex]`.

Отмена хода

Теперь реализуем поиск с возвратом для рассматриваемого рекурсивного алгоритма, то есть предусмотрим для нашей программы возможность отмены хода. Для этого будет достаточно выполнить перемещение в направлении, противоположном первоначальному. Код Python для функции `undoMove()` выглядит следующим образом:

```
def undoMove(board, move):
    # Осуществляем перемещение, противоположное 'move', чтобы отменить
    # его на поле 'board'
    if move == UP:
        makeMove(board, DOWN)
    elif move == DOWN:
        makeMove(board, UP)
    elif move == LEFT:
        makeMove(board, RIGHT)
    elif move == RIGHT:
        makeMove(board, LEFT)
```

Код JavaScript для функции `undoMove()` выглядит так:

```
function undoMove(board, move) {
    // Осуществляем перемещение, противоположное 'move', чтобы отменить
    // его на поле 'board'
    if (move === UP) {
        makeMove(board, DOWN);
    } else if (move === DOWN) {
        makeMove(board, UP);
    } else if (move === LEFT) {
        makeMove(board, RIGHT);
    } else if (move === RIGHT) {
        makeMove(board, LEFT);
    }
}
```

Мы уже запрограммировали логику, позволяющую менять значения местами, в функцию `makeMove()`, поэтому `undoMove()` вызывает эту функцию с указанием направления, противоположного направлению аргумента `move`. Таким образом, гипотетический ход `someMove`, совершенный на гипотетическом игровом поле `someBoard` в результате вызова функции `makeMove(someBoard, someMove)`, может быть отменен вызовом функции `undoMove(someBoard, someMove)`.

Настройка новой головоломки

Чтобы получить новую головоломку с перемешанными костяшками, недостаточно просто расположить их случайным образом, так как при некоторых их конфигурациях загадка оказывается нерешаемой. Необходимо сделать множество случайных ходов, начиная из состояния решенной головоломки. Тогда задача будет сводиться к определению перемещений, позволяющих отменить эти случайные ходы и вернуться к исходной упорядоченной конфигурации.

Однако не всегда получается делать ходы в каждом из четырех направлений. Например, если пустая область находится в правом нижнем углу, как на рис. 12.6, то костяшки могут быть перемещены либо вниз, либо вправо. Кроме того, если предыдущим действием было перемещение костяшки с номером 7 вверх, то ее сдвиг вниз не считается допустимым шагом, так как он отменяет предшествующий ход.

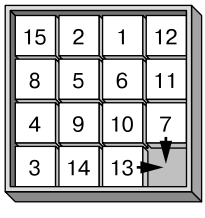


Рис. 12.6. Если пустой квадрат находится в правом нижнем углу, то костяшки можно переместить только вниз и вправо

В подобной ситуации может пригодиться функция `getValidMoves()`, возвращающая допустимые направления перемещения для текущей структуры данных игрового поля:

```
def getValidMoves(board, prevMove=None):
    """Возврат списка допустимых ходов для этого поля. Если указано
    значение prevMove, ход, который может его отменить, не применяется."""

    blankx, blanky = findBlankSpace(board)

    validMoves = []
    if blanky != SIZE - 1 and prevMove != DOWN:
        # В нижней строке нет пустой области
        validMoves.append(UP)

    if blankx != SIZE - 1 and prevMove != RIGHT:
        # В правом столбце нет пустой области
        validMoves.append(LEFT)
```

```
    if blanky != 0 and prevMove != UP:
        # В верхней строке нет пустой области
        validMoves.append(DOWN)

    if blankx != 0 and prevMove != LEFT:
        # В левом столбце нет пустой области
        validMoves.append(RIGHT)

    return validMoves
```

Код JavaScript для данной функции выглядит следующим образом:

```
function getValidMoves(board, prevMove) {
    // Возврат списка допустимых ходов для этого поля. Если указано
    // значение prevMove, ход, который может его отменить, не применяется

    let blankx, blanky;
    [blankx, blanky] = findBlankSpace(board);

    let validMoves = [];
    if (blanky != SIZE - 1 && prevMove != DOWN) {
        // В нижней строке нет пустой области
        validMoves.push(UP);
    }
    if (blankx != SIZE - 1 && prevMove != RIGHT) {
        // В правом столбце нет пустой области
        validMoves.push(LEFT);
    }
    if (blanky != 0 && prevMove != UP) {
        // В верхней строке нет пустой области
        validMoves.push(DOWN);
    }
    if (blankx != 0 && prevMove != LEFT) {
        // В левом столбце нет пустой области
        validMoves.push(RIGHT);
    }
    return validMoves;
}
```

Первым делом функция `getValidMoves()` вызывает `findBlankSpace()` и сохраняет координаты x и y пустого квадрата в переменных `blankx` и `blanky`. Затем функция создает переменную `validMoves` с пустым списком (Python) или массивом (JavaScript) для хранения всех допустимых направлений перемещения.

Если мы посмотрим на рис. 12.5, то обнаружим, что координата y, равная `0`, соответствует верхнему ряду игрового поля. Если значение `blanky` (координата y пустой области) не равно `0`, значит, пустая область находится не в верхнем ряду. Если предыдущий ход не предполагал перемещения костяшки вниз (`DOWN`),

то движение *вверх* является допустимым, поэтому код добавляет значение `UP` в список `validMoves`.

Аналогичным образом положения в крайнем левом ряду игровой доски имеют координату x, равную `0`, положения в нижнем ряду — координату y, равную `SIZE - 1`, а положения в крайнем правом ряду — координату x, равную `SIZE - 1`. Использование выражения `SIZE - 1` гарантирует работоспособность кода вне зависимости от того, какой размер поля выбран: 3×3, 4×4 или любой другой. Функция `getValidMoves()` выполняет соответствующую проверку для всех четырех направлений, а затем возвращает список допустимых ходов (`validMoves`).

Затем функция `getNewPuzzle()` возвращает структуру данных с перемешанными костяшками. Их расположение не может быть абсолютно случайным, потому что некоторые конфигурации создают головоломки, которые невозможно решить. Чтобы избежать этого, функция `getNewPuzzle()` упорядочивает игровое поле, начиная с состояния решенной головоломки, а затем выполняет большое количество случайных ходов. Решение такой головоломки, по сути, сводится к нахождению ходов, отменяющих эти перемещения. Код Python для функции `getNewPuzzle()` выглядит следующим образом:

```
def getNewPuzzle():
    # Получите новую головоломку, выполнив случайные перемещения
    # из решенного состояния
    board = getNewBoard()
    for i in range(DIFFICULTY):
        validMoves = getValidMoves(board)
        makeMove(board, random.choice(validMoves))
    return board
```

Эквивалентный код на языке JavaScript выглядит так:

```
function getNewPuzzle() {
    // Получите новую головоломку, выполнив случайные перемещения
    // из решенного состояния
    let board = getNewBoard();
    for (let i = 0; i < DIFFICULTY; i++) {
        let validMoves = getValidMoves(board);
        makeMove(board, validMoves[Math.floor(Math.random() * validMoves.length)]);
    }
    return board;
}
```

При вызове функция `getNewBoard()` получает структуру данных решенной головоломки. Цикл `for` сначала вызывает `getValidMoves()` для получения списка допустимых ходов с учетом текущего состояния поля, а затем функцию `makeMove()` для выполнения случайно выбранного хода из списка. Функция `random.choice()` в Python и функции `Math.floor()` и `Math.random()` в JavaScript выбирают значение

из списка или массива `validMoves` случайным образом, вне зависимости от содержащейся в нем комбинации значений `UP`, `DOWN`, `LEFT` и `RIGHT`.

Константа `DIFFICULTY` определяет количество случайных перемещений, выполняемых функцией `makeMove()` в цикле `for`. Чем выше целочисленное значение этой константы, тем сложнее будет решить головоломку. Несмотря на то что некоторые ходы могут по чистой случайности отменять предыдущие, например, после перемещения влево может быть выполнено перемещение вправо, достаточное количество ходов позволит функции сформировать перемешанную конфигурацию костяшек. В целях тестирования для константы `DIFFICULTY` выбрано значение `40`, что позволяет программе найти решение примерно за минуту. Для получения более реалистичной головоломки значение `DIFFICULTY` следует изменить на `200`.

После создания «перемешанной» структуры данных в переменной `board` функция `getNewPuzzle()` возвращает ее.

Рекурсивное решение «пятнашек»

Теперь, когда написана функция для создания и манипулирования структурой данных головоломки, разработаем функции, которые решают ее: рекурсивно перемещают костяшки в каждом возможном направлении и проверяют, приводит ли это загадку в законченное упорядоченное состояние.

Функция `attemptMove()` выполняет однократное перемещение костяшки по игровому полю, а затем рекурсивно вызывает сама себя один раз для каждого допустимого хода. Эта функция предусматривает несколько базовых случаев. Если структура данных игровой доски находится в состоянии решенной головоломки, функция возвращает логическое значение `True`; если достигнуто максимальное количество шагов, она возвращает логическое значение `False`. Кроме того, если рекурсивный вызов возвратил `True`, то и функция `attemptMove()` должна возвратить `True`, а если рекурсивные вызовы для всех допустимых ходов возвратили `False`, то и `attemptMove()` должна возвратить `False`.

Функция solve()

Функция `solve()` принимает в качестве аргументов структуру данных игрового поля и максимальное количество ходов, которые должен выполнить алгоритм перед тем, как возвращаться назад. Затем она выполняет первый вызов функции `attemptMove()`. Если возвращается `True`, код в функции `solve()` отображает последовательность ходов, которая позволяет решить головоломку. Если возвращается `False`, код в функции `solve()` сообщает пользователю, что решение не удалось найти за указанное максимальное количество ходов.

Начальная часть кода Python для функции `solve()` выглядит следующим образом:

```
def solve(board, maxMoves):
    """Пробуем решить головоломку, содержащуюся в переменной 'board',
    не более чем за 'maxMoves' ходов. Возвращается True, если головоломка
    решена, в противном случае — False."""
    print('Attempting to solve in at most', maxMoves, 'moves...')
    solutionMoves = [] # Список значений UP, DOWN, LEFT, RIGHT.
    solved = attemptMove(board, solutionMoves, maxMoves, None)
```

Эквивалентный код на языке JavaScript выглядит так:

```
function solve(board, maxMoves) {
    // Пробуем решить головоломку, содержащуюся в переменной 'board',
    // не более чем за 'maxMoves' ходов. Возвращается True, если головоломка
    // решена, в противном случае — False.
    document.write("Attempting to solve in at most " + maxMoves + "
      moves...<br />");
    let solutionMoves = []; // Список значений UP, DOWN, LEFT, RIGHT.
    let solved = attemptMove(board, solutionMoves, maxMoves, null);
```

Функция `solve()` предусматривает два параметра: `board` и `maxMoves`. Первый содержит структуру данных головоломки, которую нужно решить, а второй задает максимальное количество ходов, которое функция должна сделать при попытке решения задачи. Список (массив) `solutionMoves` содержит последовательность значений `UP`, `DOWN`, `LEFT` и `RIGHT`, позволяющих правильно упорядочить костяшки. Функция `attemptMove()` изменяет этот список (массив) на месте при выполнении рекурсивных вызовов.

Если исходная функция `attemptMove()` находит решение и возвращает `True`, значит, список `solutionMoves` содержит последовательность ходов для решения головоломки.

Затем функция `solve()` совершает первоначальный вызов `attemptMove()` и сохраняет возвращаемые значения `True` или `False` в переменной `solved`. Остальная часть функции `solve()` обрабатывает эти два случая:

```
    if solved:
        displayBoard(board)
        for move in solutionMoves:
            print('Move', move)
            makeMove(board, move)
            print() # Переход на новую строку
            displayBoard(board)
            print() # Переход на новую строку

        print('Solved in', len(solutionMoves), 'moves:')
        print(', '.join(solutionMoves))
        return True # Головоломка была решена
    else:
        return False # Головоломку невозможно решить за maxMoves ходов
```

Эквивалентный код на языке JavaScript выглядит так:

```
    if (solved) {
        displayBoard(board);
        for (let move of solutionMoves) {
            document.write("Move " + move + "<br />");
            makeMove(board, move);
            document.write("<br />"); // Переход на новую строку
            displayBoard(board);
            document.write("<br />"); // Переход на новую строку
        }
        document.write("Solved in " + solutionMoves.length + " moves:<br />");
        document.write(solutionMoves.join(", ") + "<br />");
        return true; // Головоломка была решена
    } else {
        return false; // Головоломку невозможно решить за maxMoves ходов
    }
}
```

Если функция `attemptMove()` находит решение, программа выполняет все ходы, содержащиеся в списке или массиве `solutionMoves`, и отображает игровое поле после перемещения каждой костяшки. Это говорит о том, что ходы, отобранные с помощью `attemptMove()`, действительно позволяют решить головоломку. Наконец, сама функция `solve()` возвращает `True`. Если функции `attemptMove()` не удается найти решение, то `solve()` просто возвращает `False`.

Функция attemptMove()

Рассмотрим рекурсивную функцию `attemptMove()`, лежащую в основе нашего алгоритма. Вспомните наш древовидный граф: вызов `attemptMove()` для определенного направления подобен перемещению вдоль соответствующего ребра графа от одного узла к другому. Рекурсивный вызов `attemptMove()` позволяет переместиться вниз по дереву. После возврата из этого рекурсивного вызова `attemptMove()` возвращается к предыдущему узлу. После возврата `attemptMove()` к корневому узлу выполнение программы снова начинается с функции `solve()`.

Начальная часть кода Python для функции `attemptMove()` выглядит следующим образом:

```
def attemptMove(board, movesMade, movesRemaining, prevMove):
    """Рекурсивная функция, которая пробует все возможные ходы
    на поле 'board', пока не найдет решение или не достигнет предельного
    числа ходов 'maxMoves'. Возвращает True, если решение было найдено.
    В этом случае 'maxMoves' содержит последовательность ходов для решения
    головоломки. Возвращает False, если значение
    'movesRemaining' меньше 0."""

    if movesRemaining < 0:
        # БАЗОВЫЙ СЛУЧАЙ — закончились допустимые ходы
        return False
```

```
    if board == SOLVED_BOARD:
        # БАЗОВЫЙ СЛУЧАЙ – головоломка решена
        return True
```

Эквивалентный код на языке JavaScript выглядит так:

```
function attemptMove(board, movesMade, movesRemaining, prevMove) {
    // Рекурсивная функция, которая пробует все возможные ходы
    // на поле 'board', пока не найдет решение или не достигнет предельного
    // числа ходов 'maxMoves'. Возвращает True, если решение было найдено.
    // В этом случае 'maxMoves' содержит последовательность ходов
    // для решения головоломки. Возвращает False, если
    // значение 'movesRemaining' меньше 0.

    if (movesRemaining < 0) {
        // БАЗОВЫЙ СЛУЧАЙ – закончились допустимые ходы
        return false;
    }
    if (JSON.stringify(board) == SOLVED_BOARD) {
        // БАЗОВЫЙ СЛУЧАЙ – головоломка решена
        return true;
    }
```

Функция `attemptMove()` предусматривает четыре параметра. Параметр `board` содержит структуру данных головоломки, которую требуется решить. Параметр `movesMade` включает в себя список или массив, который функция `attemptMove()` изменяет на месте, добавляя значения `UP`, `DOWN`, `LEFT` и `RIGHT`, соответствующие направлениям шагов, сделанных рекурсивным алгоритмом. В случае успешного решения задачи параметр `movesMade` будет содержать ходы, которые позволили привести к данному результату. Кроме того, на этот список или массив ссылается переменная `solutionMoves` в функции `solve()`.

Функция `solve()` использует свою переменную `maxMoves` в качестве параметра `movesRemaining` при первоначальном вызове `attemptMove()`. Каждый рекурсивный вызов передает `maxMoves - 1` в качестве следующего значения `maxMoves`, что приводит к постепенному уменьшению данного параметра. Когда он оказывается меньше `0`, функция `attemptMove()` прекращает выполнять рекурсивные вызовы и возвращает `False`.

Наконец, параметр `prevMove` содержит значение `UP`, `DOWN`, `LEFT` или `RIGHT`. Оно соответствует направлению перемещения, выполненного в ходе предыдущего вызова функции `attemptMove()`, что предотвращает его отмену. При исходном вызове `attemptMove()` функция `solve()` передает в качестве этого параметра значение `None` (Python) или `null` (JavaScript), поскольку в настоящий момент предыдущих перемещений еще не существует.

Начальная часть кода функции `attemptMove()` проверяет условия двух базовых случаев и возвращает `False`, если `movesRemaining` меньше `0`, или `True`, если структура

данных в переменной board соответствует состоянию решенной головоломки в константе SOLVED_BOARD.

Следующая часть кода функции attemptMove() совершает все ходы, допустимые для конкретного игрового поля. Код Python выглядит следующим образом:

```
# РЕКУРСИВНЫЙ СЛУЧАЙ – проверка каждого из допустимых ходов:
for move in getValidMoves(board, prevMove):
    # Делаем ход:
    makeMove(board, move)
    movesMade.append(move)

    if attemptMove(board, movesMade, movesRemaining - 1, move):
        # Если головоломка решена, возвращается True:
        undoMove(board, move) # Восстановление исходного состояния головоломки
        return True
```

Эквивалентный код на языке JavaScript выглядит так:

```
// РЕКУРСИВНЫЙ СЛУЧАЙ – проверка каждого из допустимых ходов:
for (let move of getValidMoves(board, prevMove)) {
    // Сделайте ход:
    makeMove(board, move);
    movesMade.push(move);

    if (attemptMove(board, movesMade, movesRemaining - 1, move)) {
        // Если головоломка решена, возвращается True:
        undoMove(board, move); // Восстановление исходного состояния головоломки
        return true;
    }
```

Цикл for задает для переменной move каждое из направлений, возвращаемых функцией getValidMoves(). При совершении хода вызывается makeMove(), чтобы изменить соответствующим образом структуру данных игрового поля и добавить этот шаг в список или массив movesMade.

Затем код рекурсивно вызывает функцию attemptMove() для изучения диапазона всех вероятных перемещений с учетом количества оставшихся ходов (movesRemaining). Этому рекурсивному вызову передаются переменные board и movesMade. Для параметра рекурсивного вызова movesRemaining код задает значение movesRemaining - 1, чтобы оно уменьшалось на единицу. Кроме того, он задает move в качестве значения prevMove, чтобы предотвратить отмену предыдущего хода.

Если рекурсивный вызов возвращает True, значит, решение найдено и сохранено в списке или массиве movesMade. Функция undoMove() вызывается, чтобы переменная board содержала исходную головоломку после возврата выполнения

программы к `solve()`, а затем возвращается `True`, чтобы подтвердить, что решение было найдено.

Следующая часть кода Python для функции `attemptMove()` выглядит так:

```
        # Отмена хода, чтобы подготовиться к совершению следующего хода:
        undoMove(board, move)
        movesMade.pop() # Удаление последнего хода, который был отменен
    return False # БАЗОВЫЙ СЛУЧАЙ – найти решение невозможно.
```

Эквивалентный код на языке JavaScript выглядит следующим образом:

```
        // Отмена хода, чтобы подготовиться к совершению следующего хода:
        undoMove(board, move);
        movesMade.pop(); // Удаление последнего хода, который был отменен
    }
    return false; // БАЗОВЫЙ СЛУЧАЙ – найти решение невозможно
}
```

Если функция `attemptMove()` возвращает `False`, значит, решение не найдено. В таком случае мы вызываем `undoMove()` и удаляем последний ход из списка (массива) `movesMade`.

Все это делается для каждого из допустимых направлений. Если ни один из вызовов `attemptMove()` для данных направлений не находит решения за максимальное количество шагов, функция `attemptMove()` возвращает значение `False`.

Запуск программы

Функция `solve()` полезна для совершения начального вызова `attemptMove()`, однако программа все еще нуждается в некоторой настройке. Соответствующий код Python выглядит так:

```
# Запуск программы:
SOLVED_BOARD = getNewBoard()
puzzleBoard = getNewPuzzle()
displayBoard(puzzleBoard)
startTime = time.time()
```

Эквивалентный код на языке JavaScript выглядит следующим образом:

```
// Запуск программы:
const SOLVED_BOARD = JSON.stringify(getNewBoard());
let puzzleBoard = getNewPuzzle();
displayBoard(puzzleBoard);
let startTime = Date.now();
```

Сначала в константе SOLVED_BOARD сохраняется упорядоченная структура данных. Она соответствует решенной головоломке, а возвращается функцией getNewBoard(). Эта константа не задается в начале исходного кода, поскольку функция getNewBoard() должна быть определена до ее вызова.

Далее getNewPuzzle() возвращает случайную конфигурацию загадки, структура данных которой сохраняется в переменной puzzleBoard. Если вы хотите получить для решения не случайную, а определенную конфигурацию ребуса, то можете заменить вызов getNewPuzzle() необходимым вам списком или массивом.

Структура данных игрового поля, содержащаяся в puzzleBoard, отображается на экране, а текущее время сохраняется в startTime для того, чтобы программа могла рассчитать время выполнения алгоритма. Соответствующая часть кода Python выглядит следующим образом:

```
maxMoves = 10
while True:
    if solve(puzzleBoard, maxMoves):
        break # Выход из цикла при нахождении решения
    maxMoves += 1
print('Run in', round(time.time() - startTime, 3), 'seconds.')
```

Эквивалентный код на языке JavaScript выглядит так:

```
let maxMoves = 10;
while (true) {
    if (solve(puzzleBoard, maxMoves)) {
        break; // Выход из цикла при нахождении решения
    }
    maxMoves += 1;
}
document.write("Run in " + Math.round((Date.now() - startTime) / 100) /
  10 + " seconds.<br />");
</script>
```

Сначала программа пытается разгадать загадку, содержащуюся в puzzleBoard, максимум за десять ходов. Бесконечный цикл while вызывает функцию solve(). Если решение найдено, solve() выводит его на экран и возвращает True. В этом случае код может выйти из бесконечного цикла while и отобразить общее время выполнения алгоритма.

В противном случае, если функция solve() возвращает False, значение maxMoves увеличивается на 1 и цикл снова вызывает solve(). Это позволяет программе постепенно увеличивать максимально допустимое количество ходов для решения головоломки. Так продолжается до тех пор, пока функция solve() не возвратит True.

Резюме

Головоломка «пятнашки» — это хороший пример применения принципов рекурсии к решению реальной задачи. Рекурсия позволяет выполнять поиск в глубину в древовидном графе состояний нашей головоломки, чтобы ее разгадать. Однако чисто рекурсивный алгоритм в данном случае не работает, поэтому нам пришлось внести некоторые коррективы.

Проблема заключается в том, что «пятнашки» предусматривают огромное количество возможных конфигураций костяшек, а граф не является ориентированным и ациклическим, так как он содержит петли, а его ребра не являются направленными. Алгоритм решения этой головоломки должен исключить вероятность отмены предыдущих ходов, чтобы обойти граф в одном направлении. Он также должен предусматривать максимальное количество ходов, после достижения которого он начнет двигаться в обратном направлении. В противном случае зацикливание вызовет переполнение стека.

Я не утверждаю, что рекурсия — это лучший способ решения «пятнашек». Просто все головоломки, кроме самых примитивных, предусматривают слишком много комбинаций, чтобы обычный ноутбук смог решить их за разумное время. Головоломка «пятнашки» нравится мне тем, что она позволяет применить такие теоретические идеи, как ориентированный ациклический граф и поиск в глубину. Данная головоломка была изобретена более века назад, а появление компьютеров открыло перед нами богатые возможности для изучения методов ее решения.

Дополнительные источники информации

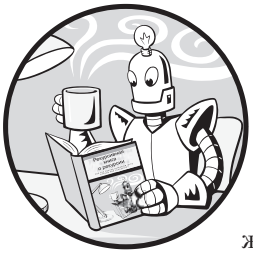
В статье «Википедии» по адресу https://ru.wikipedia.org/wiki/Игра_в_15 вы можете ознакомиться с историей изобретения головоломки «пятнашки», а также с ее математическим описанием.

Исходный код Python для игровой версии «пятнашек» вы можете найти в моей «Большой книге проектов Python»[1] и в Интернете: https://inventwithpython.com/bigbookpython/project68.html.

[1] *Свейгарт Э.* Большая книга проектов Python. — Питер, 2022.

13

Генератор фракталов



В главе 9 мы рассмотрели программы, рисующие множество известных фракталов с помощью модуля Python `turtle`, а в этой главе реализуем проект, с помощью которого вы сможете создавать собственные фрактальные рисунки. Такой генератор фракталов будет использовать модуль Python `turtle` для превращения простых фигур в сложные с помощью минимального объема кода.

Проект, описанный здесь, предусматривает девять готовых фракталов. Однако, чтобы реализовать свои творческие задумки, измените код уже готовых фракталов по своему усмотрению и получите совершенно другие изображения или же напишите нужный код с нуля.

ПРИМЕЧАНИЕ

Подробное описание функций модуля turtle дается в главе 9.

Встроенные фракталы

С помощью компьютера можно создавать неограниченное количество фракталов. На рис. 13.1 показаны девять фракталов, предусмотренных нашим генератором. Их мы и будем использовать в текущей главе. Они реализуются посредством функций, которые рисуют базовую форму, например квадрат или равносторонний треугольник, а затем рекурсивно вносят небольшие изменения для создания разнообразных изображений.

Вы можете создать любой из этих фракталов, задав для константы `DRAW_FRACTAL` в верхней части программы значение от `1` до `9`, а затем запустив программу. Если для `DRAW_FRACTAL` установить значение `10` или `11`, то на экран выведется соответственно квадрат или треугольник, из которых состоят эти фигуры, как показано на рис. 13.2.

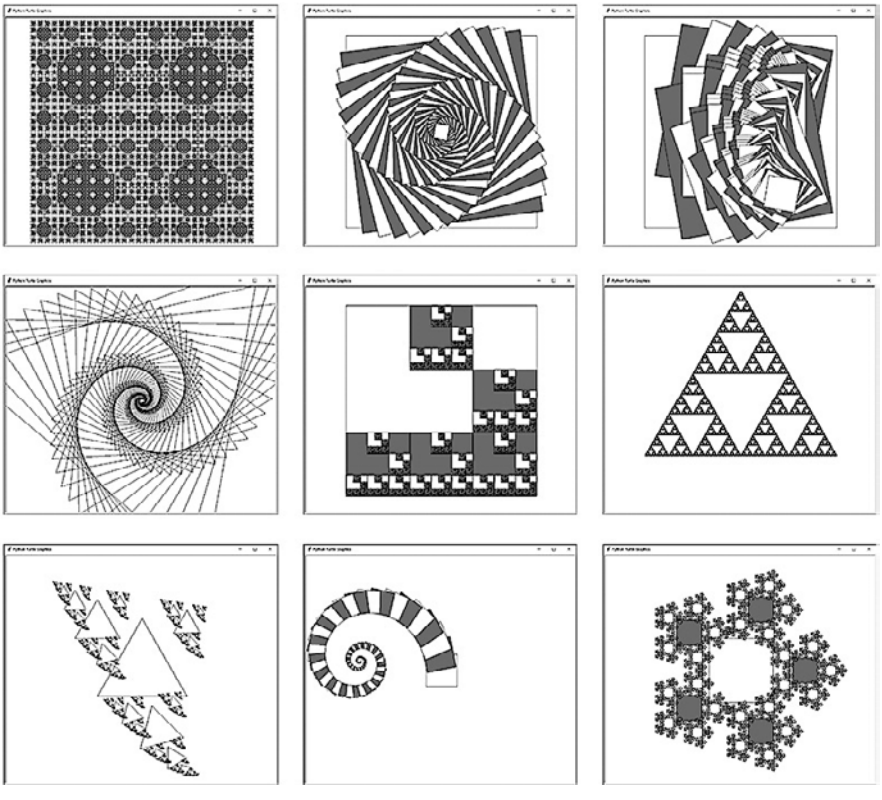


Рис. 13.1. Девять примеров фракталов, предусмотренных нашим генератором

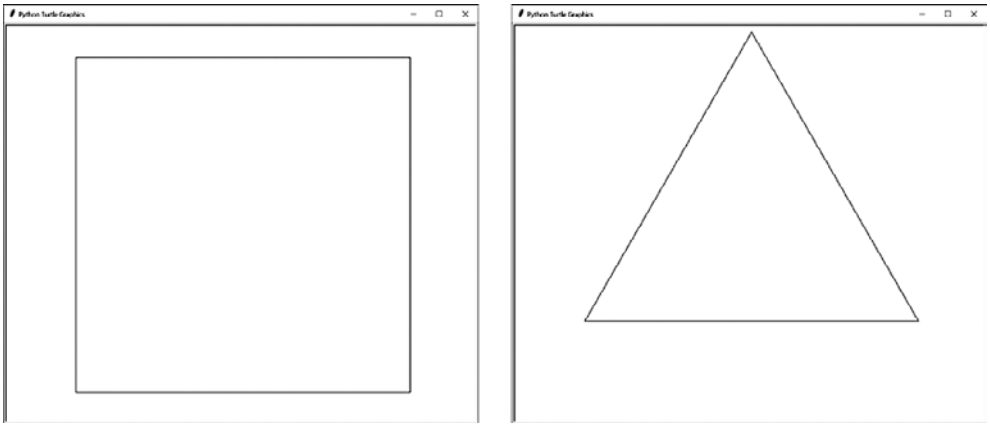


Рис. 13.2. Результат вызова функции drawFilledSquare() *(слева)* и drawTriangleOutline() *(справа)*

Данные базовые формы — заполненный белым или серым цветом квадрат и контур треугольника — используются функцией `drawFractal()` для формирования удивительных изображений.

Алгоритм генератора фракталов

Алгоритм генератора фракталов состоит из двух основных компонентов: функции для рисования фигуры и рекурсивной функции `drawFractal()`.

Первый позволяет создать базовую форму. Генератор фракталов предусматривает две такие функции — `drawFilledSquare()` и `drawTriangleOutline()`. Результат их работы показан на рис. 13.2, но вы можете создать и собственные. Мы передаем функцию для рисования фигуры в функцию `drawFractal()` в качестве аргумента точно так же, как мы передавали функции сопоставления в функцию `walk()` в главе 10.

Функция `drawFractal()` также предусматривает параметр, определяющий изменения размера, положения и угла поворота фигур между рекурсивными вызовами `drawFractal()`. Подробнее об этих деталях поговорим чуть позже, а сейчас давайте рассмотрим один пример: фрактал 7, напоминающий волну.

Программа создает фрактал «Волна», вызывая функцию `drawTriangleOutline()`, результат работы которой — один треугольник. Дополнительные аргументы функции `drawFractal()` заставляют ее трижды рекурсивно вызвать саму себя. На рис. 13.3 показан треугольник, созданный в результате первого вызова `drawFractal()`, и треугольники, полученные в результате трех последующих рекурсивных вызовов.

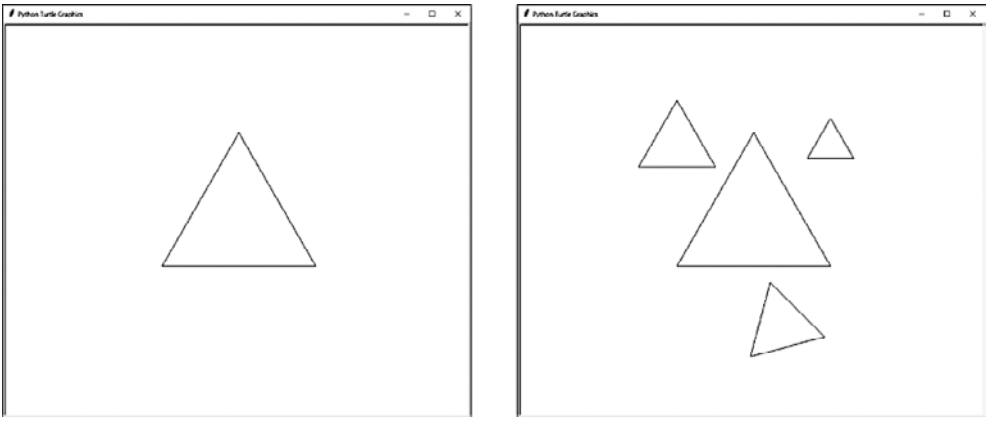


Рис. 13.3. Треугольник, созданный в результате первого вызова drawFractal() *(слева)*, и треугольники, полученные в результате трех рекурсивных вызовов *(справа)*

Первый рекурсивный вызов приказывает функции `drawFractal()` вызвать `drawTriangleOutline()` для создания треугольника вдвое меньшего размера, который должен быть расположен сверху и левее от исходного. Второй рекурсивный вызов приводит к формированию треугольника сверху и правее от исходного, размер его составляет 30 % от размера первоначального. Третий рекурсивный вызов генерирует треугольник вдвое меньшего размера под основным и повернутый относительно него на 15°.

Каждый из описанных трех рекурсивных вызовов функции `drawFractal()` выполняет еще три подобных вызова, создавая девять новых треугольников, которые претерпевают такие же изменения размера, положения и угла поворота относительно своего базового треугольника. Верхний левый треугольник всегда вдвое меньше исходного, а нижний всегда повернут на 15°. На рис. 13.4 показаны треугольники, полученные на первом и втором уровнях рекурсии.

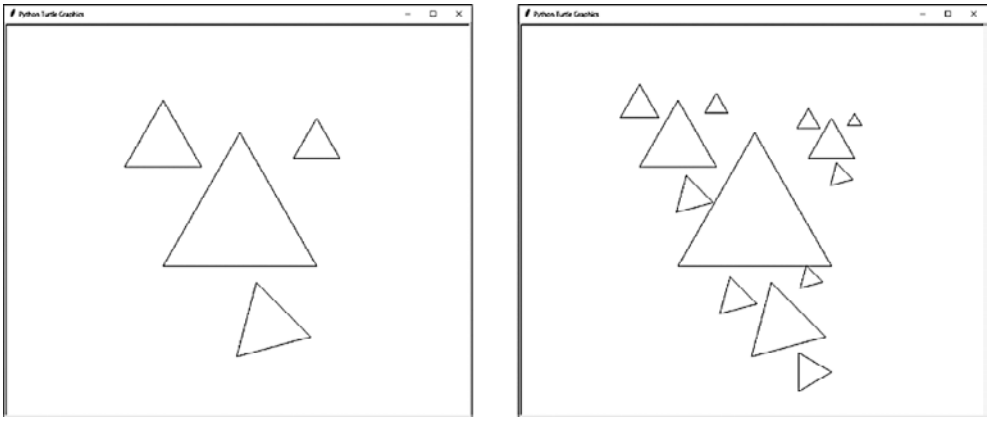


Рис. 13.4. Первый уровень рекурсивных вызовов функции drawFractal() *(слева)* и девять новых треугольников, полученных на втором уровне рекурсии *(справа)*

Каждый из девяти вызовов функции `drawFractal()`, создающих девять новых треугольников, совершает по три рекурсивных вызова этой же функции, получая 27 дополнительных фигур на следующем уровне рекурсии. В конце концов треугольники становятся слишком маленькими и функция `drawFractal()`, достигая одного из базовых случаев, прекращает выполнять рекурсивные вызовы. Другой базовый случай имеет место, когда глубина рекурсии доходит до предельного уровня. Так или иначе, эти рекурсивные вызовы порождают фрактал «Волна», показанный на рис. 13.5.

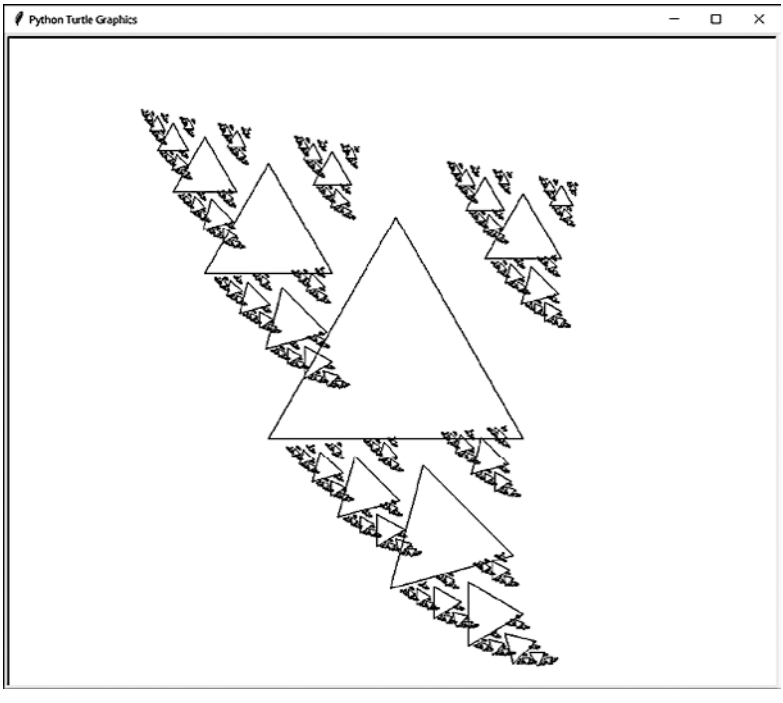


Рис. 13.5. Итоговый фрактал «Волна», созданный в результате последовательности рекурсивных вызовов функции для рисования треугольника

Девять фракталов, изображенных на рис. 13.1, были созданы с помощью всего лишь двух функций для рисования фигур и внесения нескольких изменений в аргументы `drawFractal()`. Давайте ознакомимся с кодом генератора фракталов, чтобы понять, как ему это удалось.

Полный код программы для рисования фракталов

Введите следующий код в новый файл и сохраните его под именем `fractalArtMaker.py`. Эта программа использует встроенный в Python модуль `turtle`, поэтому для реализации проекта мы не будем использовать код JavaScript:

```
import turtle, math

DRAW_FRACTAL = 1 # Задайте значение от 1 до 11 и запустите программу
```

```
turtle.tracer(5000, 0) # Увеличьте первый аргумент, чтобы ускорить процесс рисования
turtle.hideturtle()

def drawFilledSquare(size, depth):
    size = int(size)

    # Перемещение пера в правый верхний угол перед рисованием:
    turtle.penup()
    turtle.forward(size // 2)
    turtle.left(90)
    turtle.forward(size // 2)
    turtle.left(180)
    turtle.pendown()

    # Чередование белого и серого цвета заливки
    # (контур в обоих случаях будет черным):
    if depth % 2 == 0:
        turtle.pencolor('black')
        turtle.fillcolor('white')
    else:
        turtle.pencolor('black')
        turtle.fillcolor('gray')

    # Рисование квадрата:
    turtle.begin_fill()
    for i in range(4): # Рисование четырех линий
        turtle.forward(size)
        turtle.right(90)
    turtle.end_fill()

def drawTriangleOutline(size, depth):
    size = int(size)
    # Перемещение черепашки на вершину равностороннего треугольника:
    height = size * math.sqrt(3) / 2
    turtle.penup()
    turtle.left(90) # Направляем ее вверх
    turtle.forward(height * (2/3)) # Перемещение черепашки в верхний угол
    turtle.right(150) # Направляем ее на правый нижний угол
    turtle.pendown()

    # Рисование трех сторон треугольника:
    for i in range(3):
        turtle.forward(size)
        turtle.right(120)

def drawFractal(shapeDrawFunction, size, specs, maxDepth=8, depth=0):
    if depth > maxDepth or size < 1:
        return # БАЗОВЫЙ СЛУЧАЙ

    # Сохранение позиции и курса в начале этого вызова функции:
    initialX = turtle.xcor()
    initialY = turtle.ycor()
```

```
    initialHeading = turtle.heading()
    # Вызов функции для рисования фигуры:
    turtle.pendown()
    shapeDrawFunction(size, depth)
    turtle.penup()

    # РЕКУРСИВНЫЙ СЛУЧАЙ
    for spec in specs:
        # Каждый словарь в specs имеет ключи 'sizeChange', 'xChange',
        # 'yChange' и 'angleChange'. Величина изменения размера (size)
        # и координат x и y умножается на значение параметра size. Величина
        # изменения координат x и y прибавляется к координатам текущего
        # положения черепашки. Величина изменения угла (angle)
        # прибавляется к значению ее текущего курса.
        sizeCh = spec.get('sizeChange', 1.0)
        xCh = spec.get('xChange', 0.0)
        yCh = spec.get('yChange', 0.0)
        angleCh = spec.get('angleChange', 0.0)

        # Возвращаем черепашку в начальную точку фигуры:
        turtle.goto(initialX, initialY)
        turtle.setheading(initialHeading + angleCh)
        turtle.forward(size * xCh)
        turtle.left(90)
        turtle.forward(size * yCh)
        turtle.right(90)

        # Выполняем рекурсивный вызов:
        drawFractal(shapeDrawFunction, size * sizeCh, specs, maxDepth,
        depth + 1)
if DRAW_FRACTAL == 1:
    # Четыре угла:
    drawFractal(drawFilledSquare, 350,
        [{'sizeChange': 0.5, 'xChange': -0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'xChange': -0.5, 'yChange': -0.5},
         {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': -0.5}], 5)
elif DRAW_FRACTAL == 2:
    # Спираль из квадратов:
    drawFractal(drawFilledSquare, 600, [{'sizeChange': 0.95,
        'angleChange': 7}], 50)
elif DRAW_FRACTAL == 3:
    # Двойная спираль из квадратов:
    drawFractal(drawFilledSquare, 600,
        [{'sizeChange': 0.8, 'yChange': 0.1, 'angleChange': -10},
         {'sizeChange': 0.8, 'yChange': -0.1, 'angleChange': 10}])
elif DRAW_FRACTAL == 4:
    # Спираль из треугольников:
    drawFractal(drawTriangleOutline, 20,
        [{'sizeChange': 1.05, 'angleChange': 7}], 80)
elif DRAW_FRACTAL == 5:
```

```
    # Планер из игры Конвея "Жизнь":
    third = 1 / 3
    drawFractal(drawFilledSquare, 600,
        [{'sizeChange': third, 'yChange': third},
         {'sizeChange': third, 'xChange': third},
         {'sizeChange': third, 'xChange': third, 'yChange': -third},
         {'sizeChange': third, 'yChange': -third},
         {'sizeChange': third, 'xChange': -third, 'yChange': -third}])
elif DRAW_FRACTAL == 6:
    # Треугольник Серпинского:
    toMid = math.sqrt(3) / 6
    drawFractal(drawTriangleOutline, 600,
        [{'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 0},
         {'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 120},
         {'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 240}])
elif DRAW_FRACTAL == 7:
    # Волна:
    drawFractal(drawTriangleOutline, 280,
        [{'sizeChange': 0.5, 'xChange': -0.5, 'yChange': 0.5},
         {'sizeChange': 0.3, 'xChange': 0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'yChange': -0.7, 'angleChange': 15}])
elif DRAW_FRACTAL == 8:
    # Рог:
    drawFractal(drawFilledSquare, 100,
        [{'sizeChange': 0.96, 'yChange': 0.5, 'angleChange': 11}], 100)
elif DRAW_FRACTAL == 9:
    # Снежинка:
    drawFractal(drawFilledSquare, 200,
        [{'xChange': math.cos(0 * math.pi / 180),
          'yChange': math.sin(0 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(72 * math.pi / 180),
          'yChange': math.sin(72 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(144 * math.pi / 180),
          'yChange': math.sin(144 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(216 * math.pi / 180),
          'yChange': math.sin(216 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(288 * math.pi / 180),
          'yChange': math.sin(288 * math.pi / 180), 'sizeChange': 0.4}])
elif DRAW_FRACTAL == 10:
    # Заполненный цветом квадрат:
    turtle.tracer(1, 0)
    drawFilledSquare(400, 0)
elif DRAW_FRACTAL == 11:
    # Контур треугольника:
    turtle.tracer(1, 0)
    drawTriangleOutline(400, 0)
else:
    assert False, 'Set DRAW_FRACTAL to a number from 1 to 11.'

turtle.exitonclick() # Щелкните в окне, чтобы выйти
```

После запуска эта программа отобразит первое из девяти фрактальных изображений, показанных на рис. 13.1. Как я уже говорил, вы можете изменить значение константы `DRAW_FRACTAL` в начале программы на любое целое число от `1` до `9` и снова запустить ее, чтобы увидеть новый фрактал. Разобравшись в принципе работы программы, попробуйте создать собственные функции для рисования фигур, а затем вызывать функцию `drawFractal()` для генерации фракталов в соответствии с придуманным дизайном.

Задание констант и настройка конфигурации модуля turtle

Первые строки кода отвечают за базовую настройку программы, использующей модуль `turtle`:

```
import turtle, math

DRAW_FRACTAL = 1 # Задайте значение от 1 до 11 и запустите программу

turtle.tracer(5000, 0) # Увеличьте первый аргумент, чтобы ускорить процесс рисования
turtle.hideturtle()
```

Программа импортирует модуль для рисования `turtle`, а также модуль `math`, содержащий функцию `math.sqrt()`, с помощью которой будет создаваться треугольник Серпинского, и функции `math.cos()` и `math.sin()` для фрактала «Снежинка».

Для константы `DRAW_FRACTAL` можно задать любое целочисленное значение от `1` до `9`, чтобы создать один из фракталов, предусмотренных программой. Вы также можете установить для нее значение `10` или `11`, чтобы вывести на экран квадрат или треугольник соответственно.

Перед началом процесса рисования также необходимо вызывать некоторые функции модуля `turtle`. Вызов `turtle.tracer(5000, 0)` позволяет ускорить отрисовку фрактала. Аргумент `5000` приказывает модулю обработать 5000 инструкций, прежде чем отображать их результат на мониторе, а аргумент `0` задает паузу 0 миллисекунд после выполнения каждой такой инструкции. В противном случае модуль `turtle` будет визуализировать изображение после рисования каждого отдельного элемента, что сильно замедлит работу программы.

Если же вы хотите наблюдать за процессом генерации изображения, то измените данный вызов на `turtle.tracer(1, 10)`. Это поможет найти и устранить проблемы, связанные с рисованием собственных фракталов.

Вызов `turtle.hideturtle()` скрывает на экране треугольник, обозначающий текущее положение и курс черепашки (*курс* — это то же самое, что и *направление*). Мы вызываем `turtle.hideturtle()`, чтобы символ пера не присутствовал в итоговом изображении.

Работа с функциями для рисования фигур

Функция `drawFractal()` использует переданную ей функцию для рисования фигуры, чтобы создать отдельные фрагменты фрактала. Обычно они представляют собой такие простые формы, как квадрат или треугольник. Красота и сложность фракталов являются следствием рекурсивных вызовов этой функции, осуществляемых функцией `drawFractal()` для каждого компонента фрактала.

Рассматриваемая функция для рисования фракталов предусматривает два параметра: `size` и `depth`. Первый отвечает за длину сторон создаваемого квадрата или треугольника. Функции для рисования фигур всегда должны передавать в `turtle.forward()` аргументы, основанные на значении параметра `size`, чтобы длины получаемых объектов были пропорциональны значению `size` на каждом уровне рекурсии. Вместо таких вызовов, как `turtle.forward(100)` или `turtle.forward(200)`, используйте аргументы, основанные на значении `size`, например `turtle.forward(size)` или `turtle.forward(size * 2)`. В Python-модуле `turtle` функция `turtle.forward(1)` перемещает черепашку на один *юнит* (здесь — абстрактная единица измерения), который вовсе не обязательно соответствует одному пикселу.

Второй параметр определяет глубину рекурсии `drawFractal()`, у которой значение `depth` при исходном вызове равно `0`. Рекурсивные вызовы этой функции используют в качестве значения данного параметра `depth + 1`. В примере с фракталом «Волна» первый треугольник в центре окна имеет аргумент `depth`, равный `0`. Три последующих, созданных на следующем этапе, имеют глубину `1`. Девять треугольников, окружающих эти три, — глубину `2` и т. д.

Ваша функция для рисования фигуры может игнорировать этот аргумент, однако с его помощью можно создать интересные вариации базовой формы. Например, `drawFilledSquare()` использует параметр `depth` для поочередного рисования белых и серых квадратов. Имейте это в виду при создании собственных функций для генератора фракталов, так как они должны принимать аргументы `size` и `depth`.

Функция drawFilledSquare()

Функция `drawFilledSquare()` создает закрашенный квадрат со стороной, равной `size`. Мы используем функции `turtle.begin_fill()` и `turtle.end_fill()`, чтобы заполнить черный контур квадрата белым или серым цветом в зависимости от четности значения аргумента `depth`. Поскольку к этим квадратам применена заливка, любые фигуры, нарисованные в дальнейшем поверх них, частично их перекроют.

Как и остальные функции для рисования в генераторе фракталов, `drawFilledSquare()` принимает параметры `size` и `depth`:

```
def drawFilledSquare(size, depth):
    size = int(size)
```

Если значение аргумента `size` — дробное число, изображения, создаваемые модулем `turtle`, получатся слегка асимметричными. Чтобы этого избежать, первая строка кода функции округляет `size` в меньшую сторону до ближайшего целого числа.

При формировании квадрата функция предполагает, что черепашка находится в его центре. Поэтому сначала она должна переместиться в правый верхний угол фигуры относительно своего исходного положения:

```
# Перемещение пера в правый верхний угол перед рисованием:
turtle.penup()
turtle.forward(size // 2)
turtle.left(90)
turtle.forward(size // 2)
turtle.left(180)
turtle.pendown()
```

Функция `drawFractal()` всегда держит перо «на земле», поэтому, чтобы предотвратить рисование линии при перемещении кисти в исходное положение, `drawFilledSquare()` необходимо вызвать `turtle.penup()`. Чтобы найти исходное положение относительно середины квадрата, черепашка должна сначала переместиться вперед на половину длины квадрата (то есть `size // 2`), затем повернуться на 90° и переместиться вперед еще на `size // 2` единиц (юнитов) в правый верхний угол. Поскольку к моменту достижения исходного положения черепашка оказывается направленной не в ту сторону, она поворачивается на 180° и опускает перо, чтобы начать рисовать.

Обратите внимание, что направления *вверх* и *вправо* определяются относительно исходного курса черепашки. Код будет работать одинаковым образом, даже если изначально она направлена под углом 0, 90 или 42°. При создании собственных функций для рисования фигур старайтесь использовать такие относительные движения кисти, как `turtle.forward()`, `turtle.left()` и `turtle.right()`, вместо абсолютных вроде `turtle.goto()`. Затем аргумент `depth` сообщает функции, каким цветом должен быть раскрашен квадрат — белым или серым:

```
# Чередование белого и серого цвета заливки
# (контур в обоих случаях будет черным):
if depth % 2 == 0:
    turtle.pencolor('black')
    turtle.fillcolor('white')
else:
    turtle.pencolor('black')
    turtle.fillcolor('gray')
```

Если `depth` — четное число, то условие `depth % 2 == 0` истинно (`True`) и в качестве *цвета заливки* используется белый. В противном случае — серый. Но граница

квадрата, определяемая *цветом пера*, остается черной. Чтобы изменить любой из этих оттенков, используйте в качестве их значения названия обыкновенных цветов, например `red` (красный) или `yellow` (желтый), либо цветовые коды HTML, состоящие из символа решетки и шести шестнадцатеричных цифр, например `#24FF24` (лаймово-зеленый) или `#AD7100` (коричневый).

На сайте https://html-color.codes представлены таблицы, содержащие множество цветовых кодов HTML. В книге же показаны лишь черно-белые фракталы, однако с помощью своего компьютера вы можете создавать яркие фрактальные изображения!

После настройки тонов можно нарисовать четыре линии квадрата:

```
# Рисование квадрата:
turtle.begin_fill()
for i in range(4): # Рисование четырех линий
    turtle.forward(size)
    turtle.right(90)
turtle.end_fill()
```

Для того чтобы сообщить модулю `turtle`, что мы собираемся нарисовать закрашенную фигуру, а не просто ее контур, вызываем функцию `turtle.begin_fill()`. Далее следует цикл `for`, который рисует линию длиной `size` и поворачивает черепашку на 90° вправо. Цикл `for` повторяет эти действия четыре раза, чтобы получился квадрат. Когда функция вызывает `turtle.end_fill()`, на экране отображается заполненный цветом квадрат.

Функция drawTriangleOutline()

Вторая функция для рисования фигуры определяет периметр равностороннего треугольника, длина сторон которого равна значению `size`. Эта функция рисует треугольник с одним углом вверху и двумя углами внизу. Параметры данного равностороннего треугольника показаны на рис. 13.6.

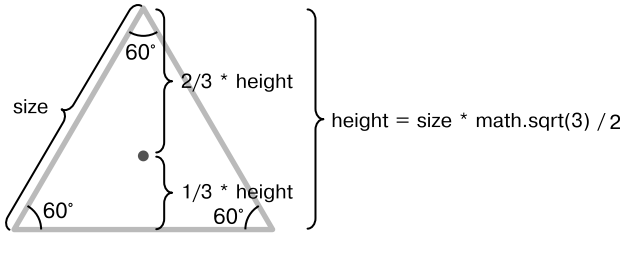


Рис. 13.6. Параметры равностороннего треугольника, длина стороны которого равна значению параметра size

Прежде чем приступать к рисованию, необходимо определить высоту треугольника на основе длины его сторон. Из школьного курса геометрии известно, что высота h равностороннего треугольника со сторонами длиной L равна $(\sqrt{3} \times L) / 2$. В нашей функции L соответствует параметру `size`, поэтому код для задания переменной `height` (высота) выглядит следующим образом:

```
height = size * math.sqrt(3) / 2
```

Из курса геометрии также известно, что центр треугольника находится на расстоянии одной третьей высоты от нижней стороны и двух третьих высоты от его вершины. Благодаря этой информации можно переместить черепашку в исходное положение:

```
def drawTriangleOutline(size, depth):
    size = int(size)

    # Перемещение черепашки на вершину равностороннего треугольника:
    height = size * math.sqrt(3) / 2
    turtle.penup()
    turtle.left(90) # Направляем ее вверх
    turtle.forward(height * (2/3)) # Перемещение черепашки в верхний угол
    turtle.right(150) # Направляем ее в правый нижний угол
    turtle.pendown()
```

Чтобы достичь верхнего угла, направляем кисть вверх, поворачивая ее на 90° влево относительно исходного курса, который составлял 0°, а затем перемещаем ее вперед на `height * (2/3)` единиц. Черепашка по-прежнему смотрит вверх, поэтому для рисования правой стороны треугольника ее нужно повернуть вправо на 90°, а затем еще на 60°, чтобы в итоге она смотрела в направлении правого нижнего угла треугольника. Для этого мы вызываем функцию `turtle.right(150)`.

Теперь черепашка готова начать рисовать треугольник, поэтому мы опускаем перо при помощи функции `turtle.pendown()` и поручаем циклу `for` создать три стороны треугольника:

```
    # Рисование трех сторон треугольника:
    for i in range(3):
        turtle.forward(size)
        turtle.right(120)
```

Процесс рисования треугольника предполагает трехкратное повторение перемещения вперед на число шагов, равное значению `size`, с поворотом вправо на 120°. В результате третьего и последнего поворота черепашка оказывается ориентированной в исходном направлении. Последовательность этих перемещений представлена на рис. 13.7.

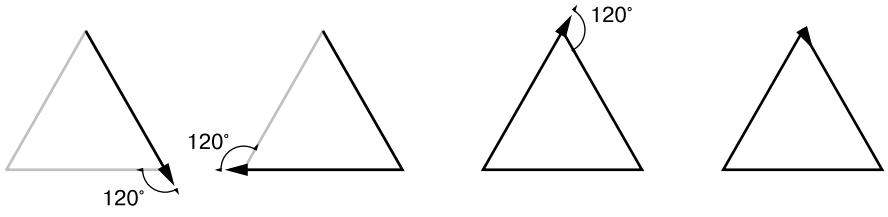


Рис. 13.7. Процесс рисования равностороннего треугольника предполагает трехкратное повторение перемещения с поворотом вправо на 120°

Функция `drawTriangleOutline()` рисует только контур, а не заполненную цветом фигуру, поэтому она не вызывает `turtle.begin_fill()` и `turtle.end_fill()`, как `drawFilledSquare()`.

Использование функции для рисования фракталов

Теперь, когда у нас есть две функции для рисования базовых фигур, давайте рассмотрим основную — `drawFractal()`. Она предусматривает три обязательных и один необязательный параметр: `shapeDrawFunction`, `size`, `specs` и `maxDepth`.

В качестве параметра `shapeDrawFunction` можно передать такую функцию, как `drawFilledSquare()` или `drawTriangleOutline()`. Параметром `size` является исходное значение `size`, переданное функции для рисования фигур. Как правило, подходящее исходное значение этого параметра — число от `100` до `500`, хотя это зависит от кода вашей функции, так что поиск правильного значения может потребовать некоторых экспериментов.

В качестве параметра `specs` передается список словарей, определяющих изменение размера, положения и угла поворота фигур в ходе рекурсивных вызовов функции `drawFractal()`. Мы поговорим об этих спецификациях более подробно чуть позже.

Параметр `maxDepth`, указывающий максимальное количество рекурсивных вызовов функции `drawFractal()`, применяется с целью предотвращения переполнения стека. По умолчанию `maxDepth` имеет значение `8`, но вы можете изменить его, если хотите, чтобы ваш фрактал содержал больше или меньше компонентов.

Пятый параметр — `depth` — используется в ходе рекурсивных вызовов функции `drawFractal()` и по умолчанию равен `0`. Указывать его при вызове `drawFractal()` нет необходимости.

Настройка функции

Первым делом `drawFractal()` проверяет условия двух базовых случаев:

```
def drawFractal(shapeDrawFunction, size, specs, maxDepth=8, depth=0):
    if depth > maxDepth or size < 1:
        return # БАЗОВЫЙ СЛУЧАЙ
```

Если значение `depth` превышает `maxDepth`, функция прекращает выполнение рекурсивных вызовов и завершает свою работу. Другой базовый случай имеет место, когда `size` оказывается меньше `1`. В этой ситуации рисуемые фигуры становятся слишком маленькими, чтобы их можно было разглядеть на экране, и поэтому функция просто заканчивает свое выполнение.

Мы отслеживаем исходное положение и курс черепашки с помощью трех переменных: `initialX`, `initialY` и `initialHeading`. Таким образом, вне зависимости от изменения ее позиции и курса в результате применения функции для рисования фигуры функция `drawFractal()` может восстановить исходное положение черепашки и ее заданное направление перед выполнением очередного рекурсивного вызова:

```
    # Сохранение позиции и курса в начале этого вызова функции:
    initialX = turtle.xcor()
    initialY = turtle.ycor()
    initialHeading = turtle.heading()
```

Функции `turtle.xcor()` и `turtle.ycor()` возвращают абсолютные координаты черепашки на экране, а `turtle.heading()` — ее текущий курс в градусах.

Следующие несколько строк вызывают функцию для рисования фигуры, переданную в качестве параметра `shapeDrawFunction`:

```
    # Вызов функции для рисования фигуры:
    turtle.pendown()
    shapeDrawFunction(size, depth)
    turtle.penup()
```

Поскольку значение, переданное в качестве параметра `shapeDrawFunction`, представляет собой функцию, код `shapeDrawFunction(size, depth)` вызывает ее, используя значения `size` и `depth`. Программа опускает перо перед вызовом `shapeDrawFunction()` и поднимает после него, гарантируя, что перо в момент рисования находится в нужном положении.

Использование словаря спецификаций

Остальная часть кода функции `drawFractal()` после вызова `shapeDrawFunction()` отвечает за выполнение рекурсивных вызовов `drawFractal()` с учетом спецификаций, содержащихся в словарях списка `specs`. Для каждого словаря

функция `drawFractal()` выполняет один рекурсивный вызов самой себя. Если список `specs` содержит только один словарь, то каждый вызов `drawFractal()` приводит к выполнению только одного рекурсивного вызова этой функции. Если список `specs` содержит три словаря, то каждый вызов `drawFractal()` приводит к выполнению трех рекурсивных вызовов.

Словари, содержащиеся в `specs`, предоставляют спецификации для каждого рекурсивного вызова. Все эти словари предусматривают ключи `sizeChange`, `xChange`, `yChange` и `angleChange`. Они определяют, как должны изменяться размер фрактала, положение и курс черепашки при рекурсивном вызове `drawFractal()`. В табл. 13.1 описаны четыре ключа словарей со спецификациями.

Таблица 13.1. Ключи словарей со спецификациями

Ключ	Значение по умолчанию	Описание
`sizeChange`	`1.0`	Значение размера следующей рекурсивной формы равно текущему размеру, умноженному на данное значение
`xChange`	`0.0`	Координата *x* следующей рекурсивной формы представляет собой текущую координату *x*, увеличенную на произведение нынешнего размера и данного значения
`yChange`	`0.0`	Координата *y* следующей рекурсивной формы представляет собой текущую координату *y*, увеличенную на произведение нынешнего размера и данного значения
`angleChange`	`0.0`	Начальный угол следующей рекурсивной формы равен текущему значению начального угла, увеличенному на данное значение

Рассмотрим словарь спецификаций для фрактала «Четыре угла», который изображен вверху слева на рис. 13.1. При вызове функции `drawFractal()` для данной фигуры в качестве параметра `specs` передается следующий список словарей:

```
[{'sizeChange': 0.5, 'xChange': -0.5, 'yChange': 0.5},
 {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': 0.5},
 {'sizeChange': 0.5, 'xChange': -0.5, 'yChange': -0.5},
 {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': -0.5}]
```

Этот список `specs` содержит четыре словаря, поэтому каждый вызов функции `drawFractal()`, рисующий квадрат, рекурсивно вызывает `drawFractal()` четыре раза, чтобы нарисовать четыре дополнительных квадрата. Последовательность создания чередующихся белых и серых квадратов показана на рис. 13.8.

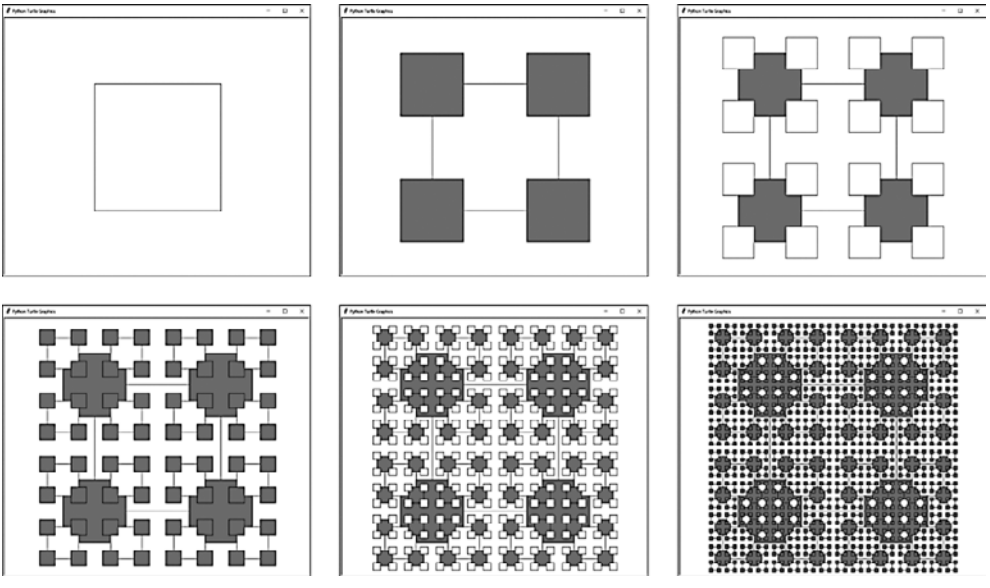


Рис. 13.8. Этапы создания фрактала «Четыре угла» слева направо и сверху вниз. Каждый рекурсивный вызов создает еще четыре квадрата по углам исходного, чередуя их цвета между белым и серым

Чтобы определить размер очередного квадрата, значение ключа `sizeChange` умножается на текущее значение параметра `size`. Первый словарь в списке `specs` содержит значение `sizeChange`, равное `0.5`, поэтому в ходе следующего рекурсивного вызова значением аргумента `size` будет `350 * 0.5`, или `175` единиц. В результате очередной квадрат оказывается вдвое меньше предыдущего. Значение `sizeChange`, равное `2.0`, удваивало бы размер каждого следующего квадрата. Если в словаре нет ключа `sizeChange`, используется значение по умолчанию `1.0`, не приводящее к изменению размера очередного квадрата.

Чтобы определить координату *x* дочернего квадрата, значение `xChange` из первого словаря, в данном случае `-0.5`, умножается на `size`. Если `size` равно `350`, это означает, что координата *x* такого квадрата смещена на `-175` единиц относительно текущего положения черепашки. При таком значении `xChange` и ключа `yChange`, равном `0.5`, новый квадрат оказывается смещен влево и вверх относительно родительского на расстояние, составляющее 50 % от его размера. При этом центр следующего прямоугольника совпадает с верхним левым углом предыдущего.

Если вы посмотрите на три других словаря в списке `specs`, то заметите, что все их параметры `sizeChange` равны `0.5`. Однако они отличаются значениями `xChange` и `yChange`, которые центрируют следующие квадраты в остальных трех углах предшествующего.

Словари в списке `specs` не содержат значения `angleChange`, поэтому в данном примере используется значение по умолчанию, равное 0°. Положительное значение `angleChange` соответствует повороту против часовой стрелки, а отрицательное — по часовой стрелке.

Каждый словарь представляет собой отдельный квадрат, который должен быть нарисован в ходе каждого рекурсивного вызова функции. Если бы мы удалили первый словарь из списка `specs`, то всякий вызов `drawFractal()` создавал бы только три квадрата, как показано на рис. 13.9.

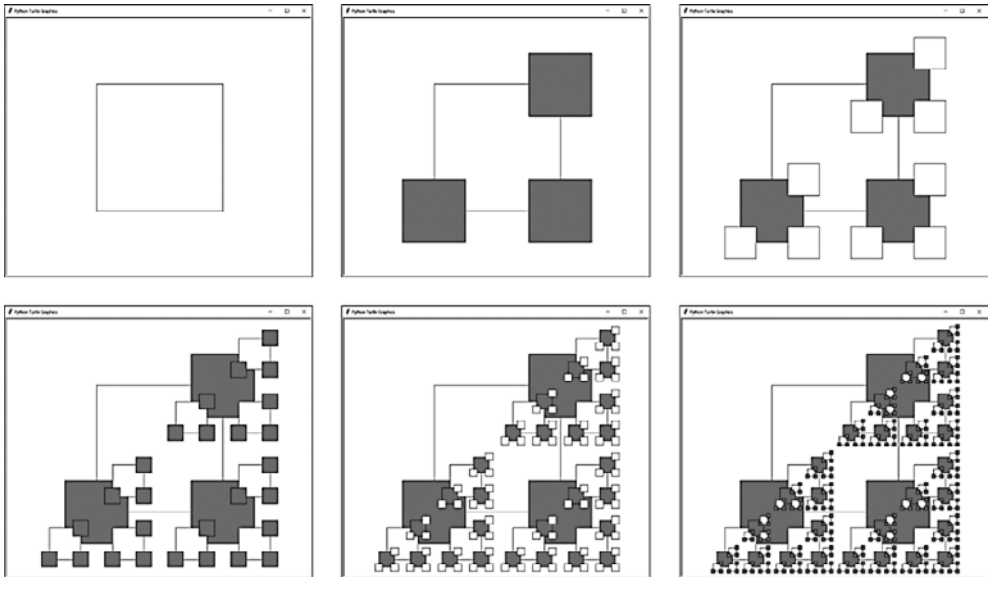


Рис. 13.9. Фрактал «Четыре угла», полученный в результате удаления первого словаря из списка specs

Применение спецификаций

А теперь посмотрим, как код функции `drawFractal()` выполняет все вышеописанные действия:

```
    # РЕКУРСИВНЫЙ СЛУЧАЙ
    for spec in specs:
        # Каждый словарь в specs имеет ключи 'sizeChange', 'xChange',
        # 'yChange' и 'angleChange'. Величина изменения размера (size)
        # и координат x и y умножается на значение параметра size. Величина
        # изменения координат x и y прибавляется к координатам текущего
        # положения черепашки. Величина изменения угла (angle)
        # прибавляется к значению ее текущего курса.
```

```
sizeCh = spec.get('sizeChange', 1.0)
xCh = spec.get('xChange', 0.0)
yCh = spec.get('yChange', 0.0)
angleCh = spec.get('angleChange', 0.0)
```

Цикл `for` присваивает отдельный словарь спецификаций в списке `specs` переменной цикла `spec` при каждой итерации. Вызовы метода словаря `get()` извлекают значения ключей `sizeChange`, `xChange`, `yChange` и `angleChange` из конкретного словаря и присваивают их переменным с более короткими именами `sizeCh`, `xCh`, `yCh` и `angleCh`. Если ключ в словаре отсутствует, метод `get()` заменяет значение по умолчанию.

Затем положение и курс черепашки сбрасываются до величин, указанных при первом вызове функции `drawFractal()`. Это значит, что в результате рекурсивных вызовов предыдущих итераций цикла черепашка не окажется не в том месте. Затем ее курс и положение изменяются в соответствии со значениями переменных `angleCh`, `xCh` и `yCh`:

```
# Возвращаем черепашку в начальную точку фигуры:
turtle.goto(initialX, initialY)
turtle.setheading(initialHeading + angleCh)

turtle.forward(size * xCh)
turtle.left(90)
turtle.forward(size * yCh)
turtle.right(90)
```

Изменение координат x и y происходит с учетом текущего курса черепашки. Если он составляет 0°, то относительная ось X черепашки совпадает с фактической осью X на экране. Однако если траектория черепашки составляет, скажем, `45`, то ее относительная ось X оказывается наклонена под углом 45°. При перемещении «вправо» вдоль этой относительной оси X перо будет двигаться в направлении правого верхнего угла.

Вот почему при перемещении вперед на `size * xCh` шагов черепашка движется вдоль своей относительной оси X. При отрицательном значении `xCh` вызов функции `turtle.forward()` заставляет ее перемещаться влево вдоль относительной оси X. Вызов `turtle.left(90)` направляет ее вдоль относительной оси Y, а `turtle.forward(size * yCh)` перемещает кисть в начальную точку следующей фигуры. Однако вызов `turtle.left(90)` изменяет курс черепашки, поэтому для его восстановления вызывается функция `turtle.right(90)`.

На рис. 13.10 показано, как эти четыре строки кода перемещают черепашку вправо вдоль ее относительной оси X и вверх вдоль относительной оси Y. При этом она оказывается ориентирована в правильном направлении вне зависимости от того, какой была ее изначальная траектория.

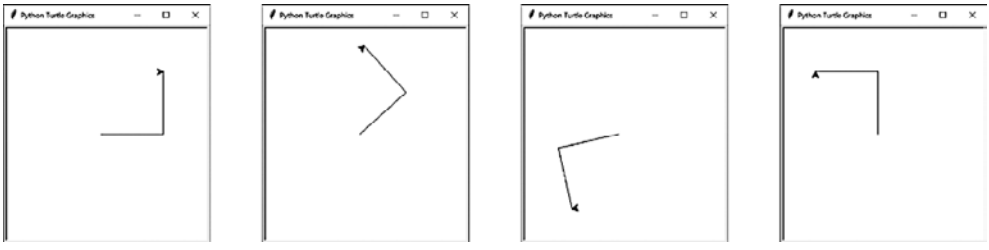


Рис. 13.10. На каждом из этих четырех изображений черепашка перемещается на 100 единичных шагов вправо и вверх вдоль относительных осей X и Y, заданных ее начальным курсом

Восстановив правильное положение и направление черепашки для рисования следующей фигуры, мы совершаем рекурсивный вызов `drawFractal()`:

```
# Выполняем рекурсивный вызов:
drawFractal(shapeDrawFunction, size * sizeCh, specs, maxDepth,
depth + 1)
```

Аргументы `shapeDrawFunction`, `specs` и `maxDepth` передаются рекурсивному вызову `drawFractal()` без изменений. В качестве параметра `size` используется значение `size * sizeCh`, определяющее размер следующей фигуры, а в качестве параметра `depth` передается `depth + 1`, инкрементирующее значение глубины рекурсии в ходе очередного вызова функции для рисования фигуры.

Создание фракталов

Теперь, когда мы разобрались с принципом работы функций для рисования фигур и рекурсивной функции `drawFractal()`, рассмотрим девять фракталов, встроенных в нашу программу. Их изображения представлены на рис. 13.1.

Четыре угла

Рисование «Четырех углов» начинается с создания большого квадрата. В ходе рекурсивного вызова функции в четырех углах исходного квадрата формируются четыре квадрата поменьше в соответствии со спецификациями:

```
if DRAW_FRACTAL == 1:
    # Четыре угла:
    drawFractal(drawFilledSquare, 350,
        [{'sizeChange': 0.5, 'xChange': -0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'xChange': -0.5, 'yChange': -0.5},
         {'sizeChange': 0.5, 'xChange': 0.5, 'yChange': -0.5}], 5)
```

В данном случае вызов `drawFractal()` ограничивает максимальную глубину рекурсии пятью уровнями, так как при использовании большей величины детали фрактала становятся трудноразличимыми. Этот фрактал показан на рис. 13.8.

Спираль из квадратов

«Спираль из квадратов» также начинается с рисования большого квадрата, однако в ходе каждого последующего рекурсивного вызова создается только одна дополнительная фигура:

```
elif DRAW_FRACTAL == 2:
    # Спираль из квадратов:
    drawFractal(drawFilledSquare, 600, [{'sizeChange': 0.95,
        'angleChange': 7}], 50)
```

Новый квадрат немного меньше предыдущего и повернут относительно него на 7°. Центры всех прямоугольников остаются неизменными, поэтому в спецификацию не требуется добавлять ключи `xChange` и `yChange`. Заданная по умолчанию глубина рекурсии `8` слишком мала для изображения интересного фрактала, поэтому мы увеличиваем ее до `50`, чтобы сгенерировать гипнотический спиральный узор.

Двойная спираль из квадратов

«Двойная спираль из квадратов» похожа на предыдущий фрактал, за исключением того, что при каждом рекурсивном вызове создается не один, а два меньших квадрата. В результате получается интересный веерообразный узор, так как фигуры, нарисованные позднее, частично скрывают ранее нарисованные прямоугольники:

```
elif DRAW_FRACTAL == 3:
    # Двойная спираль из квадратов:
    drawFractal(drawFilledSquare, 600,
        [{'sizeChange': 0.8, 'yChange': 0.1, 'angleChange': -10},
         {'sizeChange': 0.8, 'yChange': -0.1, 'angleChange': 10}])
```

Новые квадраты создаются чуть выше или ниже предыдущих и поворачиваются относительно них на 10 или –10°.

Спираль из треугольников

Фрактал «Спираль из треугольников» представляет собой вариацию «Спирали из квадратов» и использует функцию `drawTriangleOutline()` вместо `drawFilledSquare()`:

```
elif DRAW_FRACTAL == 4:
    # Спираль из треугольников:
    drawFractal(drawTriangleOutline, 20,
        [{'sizeChange': 1.05, 'angleChange': 7}], 80)
```

В отличие от «Спирали из квадратов» при генерации данного фрактала начальное значение параметра `size` составляет 20 единиц и немного увеличивается на каждом уровне рекурсии. Благодаря тому что значение ключа `sizeChange` превышает `1.0`, размеры фигур постоянно увеличиваются. Это означает, что базовый случай имеет место на 80-м уровне рекурсии, потому что базовый случай, при котором значение параметра `size` становится меньше `1`, никогда не достигается.

Планер из игры Конвея «Жизнь»

Игра Конвея «Жизнь» является известным примером клеточного автомата. Простые правила этой игры создают интересные и хаотичные узоры на двумерной сетке. Один из таких узоров — «Планер», состоящий из пяти клеток поля 3×3:

```
elif DRAW_FRACTAL == 5:
    # Планер из игры Конвея "Жизнь":
    third = 1 / 3
    drawFractal(drawFilledSquare, 600,
        [{'sizeChange': third, 'yChange': third},
         {'sizeChange': third, 'xChange': third},
         {'sizeChange': third, 'xChange': third, 'yChange': -third},
         {'sizeChange': third, 'yChange': -third},
         {'sizeChange': third, 'xChange': -third, 'yChange': -third}])
```

Наш фрактал «Планер» содержит дополнительные планеры внутри каждой из пяти клеток. Переменная `third` помогает точно задать положение рекурсивных фигур в сетке 3×3.

Вы можете найти реализацию игры Конвея «Жизнь» на языке Python в моей «Большой книге проектов Python» и в Интернете по адресу https://inventwithpython.com/bigbookpython/project13.html. К сожалению, Джон Конвей, математик и профессор, придумавший игру «Жизнь», скончался от осложнений, вызванных COVID-19, в апреле 2020 года.

Треугольник Серпинского

Мы уже создавали треугольник Серпинского в главе 9, однако наш генератор фракталов тоже способен это сделать, используя функцию `drawTriangleOutline()`. В конце концов, треугольник Серпинского — это всего лишь равносторонний треугольник, внутри которого нарисованы еще три равносторонних треугольника поменьше:

```
elif DRAW_FRACTAL == 6:
    # Треугольник Серпинского:
    toMid = math.sqrt(3) / 6
    drawFractal(drawTriangleOutline, 600,
        [{'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 0},
         {'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 120},
         {'sizeChange': 0.5, 'yChange': toMid, 'angleChange': 240}])
```

Центры этих меньших треугольников смещены на `size * math.sqrt(3) / 6` единиц относительно центра базового треугольника. В ходе трех рекурсивных вызовов курс черепашки меняется на 0, 120 и 240°, прежде чем она начинает двигаться вверх вдоль своей относительной оси *Y*.

Волна

Мы обсуждали фрактал «Волна» (см. рис. 13.5) в начале текущей главы. Этот относительно простой фрактал образуется путем рекурсивного создания трех небольших треугольников, слегка отличающихся от исходного:

```
elif DRAW_FRACTAL == 7:
    # Волна:
    drawFractal(drawTriangleOutline, 280,
        [{'sizeChange': 0.5, 'xChange': -0.5, 'yChange': 0.5},
         {'sizeChange': 0.3, 'xChange': 0.5, 'yChange': 0.5},
         {'sizeChange': 0.5, 'yChange': -0.7, 'angleChange': 15}])
```

Рог

Фрактал «Рог» напоминает бараний рог:

```
elif DRAW_FRACTAL == 8:
    # Рог:
    drawFractal(drawFilledSquare, 100,
        [{'sizeChange': 0.96, 'yChange': 0.5, 'angleChange': 11}], 100)
```

Данный простой фрактал состоит из квадратов, каждый из которых немного меньше, смещен вверх и повернут на 11° относительно предыдущего. Чтобы скрутить рог в тугую спираль, мы увеличили максимальную глубину рекурсии до 100.

Снежинка

Последний фрактал под названием «Снежинка» состоит из квадратов, выложенных в виде пятиугольника. Он похож на фрактал «Четыре угла», но вместо четырех в нем используется пять равноудаленных друг от друга рекурсивных квадратов:

```
elif DRAW_FRACTAL == 9:
    # Снежинка:
    drawFractal(drawFilledSquare, 200,
        [{'xChange': math.cos(0 * math.pi / 180),
          'yChange': math.sin(0 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(72 * math.pi / 180),
          'yChange': math.sin(72 * math.pi / 180), 'sizeChange': 0.4},
         {'xChange': math.cos(144 * math.pi / 180),
          'yChange': math.sin(144 * math.pi / 180), 'sizeChange': 0.4},
```

```
        {'xChange': math.cos(216 * math.pi / 180),
         'yChange': math.sin(216 * math.pi / 180), 'sizeChange': 0.4},
        {'xChange': math.cos(288 * math.pi / 180),
         'yChange': math.sin(288 * math.pi / 180), 'sizeChange': 0.4}])
```

Такой фрактал использует для определения величины смещения квадратов вдоль осей *X* и *Y* тригонометрические функции косинуса и синуса, реализованные в функциях Python `math.cos()` и `math.sin()`. Поскольку в круге 360°, для равномерного распределения в нем пяти рекурсивных квадратов мы размещаем их в точках, соответствующих 0, 72, 144, 216 и 288°. Функции `math.cos()` и `math.sin()` работают со значениями углов в радианах, а не в градусах, поэтому надо умножить эти числа на `math.pi / 180`.

Так каждый квадрат оказывается окруженным пятью другими квадратами, каждый из которых тоже окружен пятью квадратами и т. д. В результате формируется изображение, напоминающее снежинку.

Создание отдельного квадрата или треугольника

Вы также можете задать для константы `DRAW_FRACTAL` значение `10` или `11`, чтобы узнать, к какому результату приводит однократный вызов функций `drawFilledSquare()` и `drawTriangleOutline()` в окне модуля `turtle`. Эти фигуры будут нарисованы с использованием значения `size`, равного `400`:

```
elif DRAW_FRACTAL == 10:
    # Заполненный цветом квадрат:
    turtle.tracer(1, 0)
    drawFilledSquare(400, 0)
elif DRAW_FRACTAL == 11:
    # Контур треугольника:
    turtle.tracer(1, 0)
    drawTriangleOutline(400, 0)
turtle.exitonclick() # Щелкните в окне, чтобы выйти
```

После создания объекта на основе значения константы `DRAW_FRACTAL` программа вызывает функцию `turtle.exitonclick()`, чтобы окно модуля оставалось открытым до тех пор, пока пользователь не щелкнет в нем кнопкой мыши. После щелчка программа завершит свою работу.

Создание собственных фракталов

Вы можете рисовать собственные фракталы, изменяя спецификацию, передаваемую функции `drawFractal()`. Для начала подумайте, сколько рекурсивных вызовов должен генерировать каждый вызов `drawFractal()` и как должны меняться размер, положение и угол поворота соответствующих фигур. Можно использовать существующие функции или создать собственные.

Например, на рис. 13.11 показаны девять встроенных в наш генератор фракталов, при создании которых поменялись местами функции рисования квадрата и треугольника. Некоторые из них не представляют ничего выдающегося, а другие порождают неожиданную красоту.

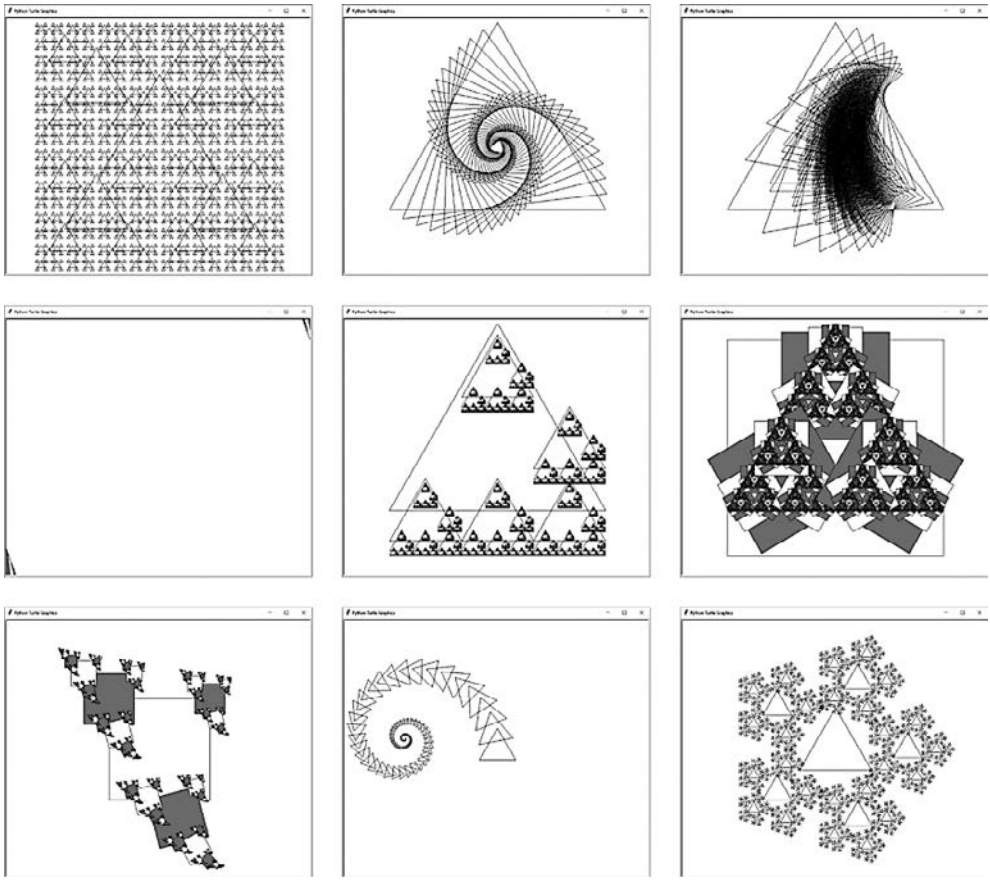


Рис. 13.11. Девять фракталов, предусмотренных в нашем генераторе, при создании которых поменялись местами функции рисования квадрата и треугольника

Резюме

Изображения, созданные с помощью генератора фракталов, демонстрируют бесконечные возможности рекурсии. Простая рекурсивная функция `drawFractal()` в сочетании с функцией для рисования фигур способна создать огромное разнообразие детализированных геометрических рисунков.

В основе генератора фракталов лежит рекурсивная функция `drawFractal()`, которая принимает другую функцию в качестве аргумента. Эта вторая функция многократно воссоздает базовую форму, используя значения размера, положения и угла поворота, заданные в списке со словарями спецификаций.

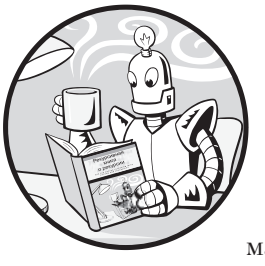
Вы можете протестировать неограниченное количество функций для рисования фигур и спецификаций. Раскройте свой творческий потенциал и создайте собственные фрактальные изображения, поэкспериментировав с кодом этой программы.

Дополнительные источники информации

В Интернете существуют веб-сайты, позволяющие создавать фракталы. Например, сайт Interactive Fractal Tree (https://www.visnos.com/demos/fractal) предусматривает ползунковые регуляторы для изменения параметров угла наклона и размера ветвей фрактального двоичного дерева. Сайт Procedural Snowflake (https://procedural-snowflake.glitch.me) генерирует новые снежинки в окне вашего браузера. Фрактальная машина Нико (https://sciencevsmagic.net/fractal) создает анимированные фрактальные изображения. Вы можете найти и другие подобные сайты в Сети по запросу «создание фракталов» или «генератор фракталов онлайн».

14

Создание эффекта Дросте



Эффект Дросте — это техника размещения рекурсивного изображения, получившая свое название в честь иллюстрации 1904 года, размещенной на банке какао голландской марки Droste. На этой иллюстрации была изображена медсестра, держащая поднос с банкой какао Droste, на которой была нарисована медсестра, держащая... Думаю, вы уже догадались (рис. 14.1).

В этой главе мы напишем программу для создания эффекта Дросте. Она будет генерировать похожие рекурсивные изображения из любой имеющейся у вас фотографии или рисунка, будь на нем посетитель музея, рассматривающий собственное фото, кошка перед монитором компьютера с изображением кошки перед монитором компьютера или что-то совершенно другое.

Используя графическую программу, вроде Microsoft Paint или Adobe Photoshop, подготовьте изображение, залив пурпурным цветом ту его область, в которой хотите разместить рекурсивное изображение. Наша программа будет использовать библиотеку изображений Pillow для чтения этих данных и создания рекурсивного изображения.

Сначала мы рассмотрим процесс установки библиотеки Pillow и принцип работы алгоритма программы для создания эффекта Дросте. Затем подробно проанализируем ее исходный код на языке Python.

Рис. 14.1. Рекурсивная иллюстрация на банке какао Droste

Установка библиотеки Python Pillow

Для реализации проекта вам потребуется библиотека Pillow, позволяющая программам на языке Python создавать и изменять файлы изображений в таких форматах, как PNG, JPEG и GIF. Она предусматривает ряд распространенных функций для работы с изображениями, включая функции масштабирования, копирования и кадрирования.

Чтобы установить эту библиотеку в ОС Windows, откройте окно командной строки и выполните команду `py -m pip install --user pillow`. Для установки данной библиотеки в ОС macOS или Linux откройте окно терминала и выполните команду `python3 -m pip install --user pillow`. Она заставляет Python использовать программу установки PIP для загрузки модуля из официального репозитория пакетов Python, доступного по адресу https://pypi.org.

Чтобы убедиться, что установка прошла успешно, откройте терминал Python и выполните команду `from PIL import Image` (хотя библиотека называется Pillow, установленный модуль Python имеет название `PIL`). Если никакие сообщения об ошибках не появились, значит, установка прошла корректно.

Официальную документацию по модулю Pillow можно найти по адресу https://pillow.readthedocs.io.

Подготовка изображения

Следующий шаг — подготовка изображения путем закрашивания его части пурпурным цветом, который в модели RGB (красный, зеленый, синий) имеет значение (255, 0, 255). Пурпурный цвет часто используется в компьютерной графике для обозначения пикселов картинки, которые должны быть прозрачными. Наша программа будет обрабатывать эти пурпурные пикселы как зеленый экран при производстве видео, заменяя их уменьшенной версией исходного изображения, которое, в свою очередь, тоже будет содержать небольшую пурпурную область для замены. Базовый случай будет иметь место, когда в уменьшенной иллюстрации не останется пурпурных пикселов. В этот момент алгоритм завершит свою работу.

На рис. 14.2 показана последовательность рисунков, сгенерированных в результате рекурсивной замены пурпурной области исходным изображением. В нашем примере посетитель музея стоит перед картиной, закрашенной пурпурными пикселами, что позволяет превратить ее саму в экспонат. Данное исходное изображение доступно по адресу https://inventwithpython.com/museum.png.

Убедитесь, что для закрашивания области изображения вы используете чистый пурпурный цвет (255, 0, 255). Некоторые инструменты графических программ создают эффект выцветания, позволяющий придавать изображениям более естественный вид. Например, инструмент Кисть в программе Photoshop создает полупрозрачные пурпурные пикселы у контура закрашенной области, поэтому вместо него следует использовать инструмент Перо, который рисует необходимым цветом. Если ваша графическая программа не позволяет указать точное значение цвета в RGB, можете использовать в качестве образца цвета PNG-изображение, доступное по адресу https://inventwithpython.com/magenta.png.

Пурпурная область в изображении может иметь любой размер и форму, это не обязательно должен быть прямоугольник. На рис. 14.2 заметно, что посетитель музея перекрывает собой закрашенную зону, то есть находится перед рекурсивным изображением.

При создании собственных картинок с помощью программы для создания эффекта Дросте вам следует использовать файлы в формате PNG, а не JPEG. Для минимизации размера JPEG-файла используются алгоритмы сжатия *с потерями*, что вносит в изображение небольшие артефакты. Обычно они незаметны для человеческого глаза и не влияют на общее качество изображения. Однако при использовании сжатия с потерями пикселы чистого пурпурного цвета (255, 0, 255) могут быть заменены другими оттенками пурпурного. Чтобы этого избежать, лучше использовать PNG-изображения, ведь они применяют сжатие *без потерь*.

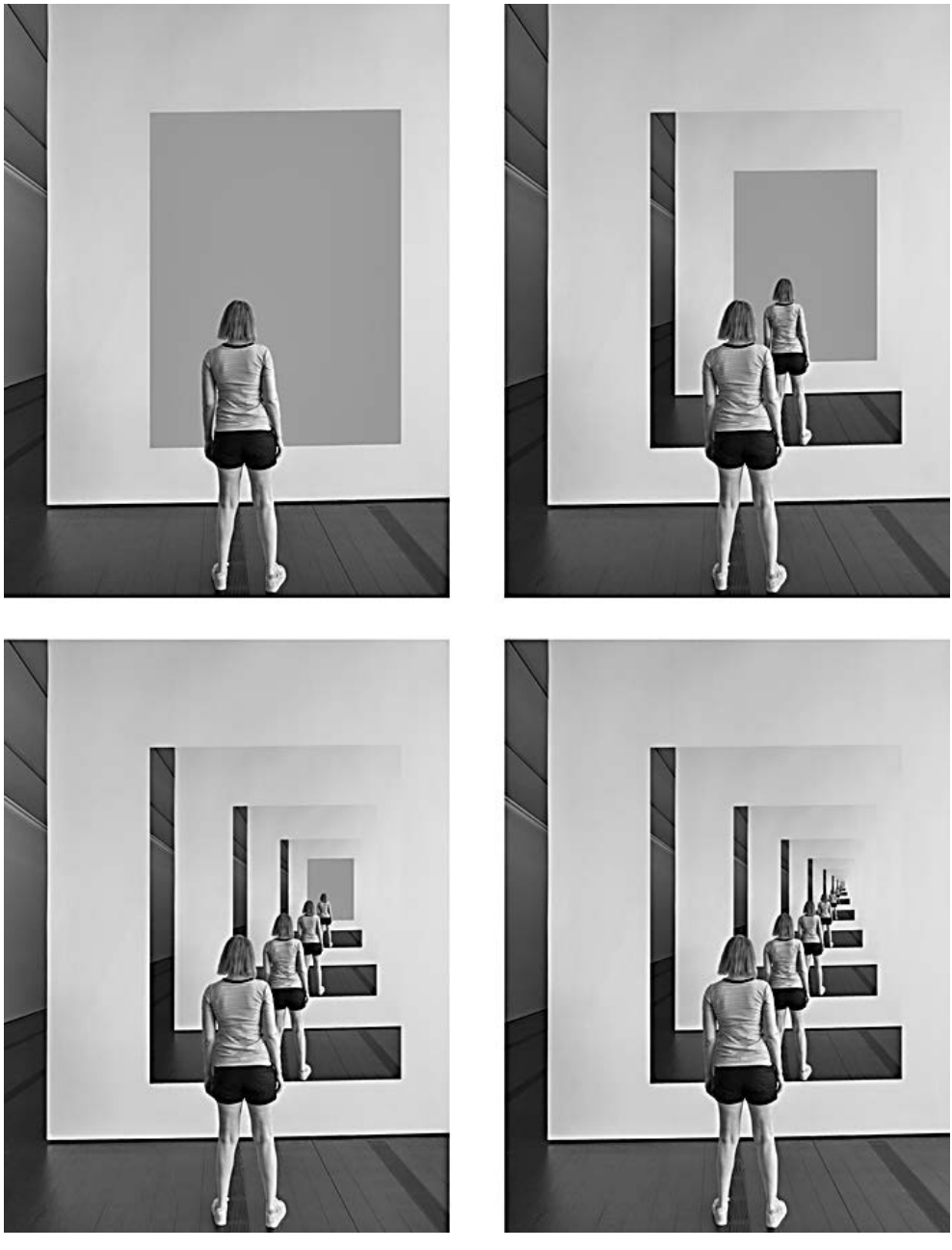


Рис. 14.2. Рекурсивная вставка изображения в область, заполненную пурпурными пикселами. Для тех, кто рассматривает черно-белое изображение в книге, пурпурная область — это прямоугольник, на который смотрит посетитель музея

Полный код программы для создания эффекта Дросте

Далее приведен исходный код программы `drostemaker.py`. Поскольку она использует библиотеку Pillow, доступную только для Python, в книге отсутствует эквивалентный код на языке JavaScript:

```
from PIL import Image

def makeDroste(baseImage, stopAfter=10):
    # Если baseImage представляет собой строку с именем файла изображения,
    # загрузите это изображение:
    if isinstance(baseImage, str):
        baseImage = Image.open(baseImage)

    if stopAfter == 0:
        # БАЗОВЫЙ СЛУЧАЙ
        return baseImage
    # Пурпурный цвет имеет максимальное значение красного/синего/альфа-каналов
    # и нулевое значение зеленого канала:
    if baseImage.mode == 'RGBA':
        magentaColor = (255, 0, 255, 255)
    elif baseImage.mode == 'RGB':
        magentaColor = (255, 0, 255)

    # Определение размера базового изображения и его пурпурной области:
    baseImageWidth, baseImageHeight = baseImage.size
    magentaLeft = None
    magentaRight = None
    magentaTop = None
    magentaBottom = None
    for x in range(baseImageWidth):
        for y in range(baseImageHeight):
            if baseImage.getpixel((x, y)) == magentaColor:
                if magentaLeft is None or x < magentaLeft:
                    magentaLeft = x
                if magentaRight is None or x > magentaRight:
                    magentaRight = x
                if magentaTop is None or y < magentaTop:
                    magentaTop = y
                if magentaBottom is None or y > magentaBottom:
                    magentaBottom = y

    if magentaLeft is None:
        # БАЗОВЫЙ СЛУЧАЙ — изображение не содержит пурпурных пикселов
        return baseImage

    # Получение уменьшенной версии базового изображения:
    magentaWidth = magentaRight - magentaLeft + 1
    magentaHeight = magentaBottom - magentaTop + 1
    baseImageAspectRatio = baseImageWidth / baseImageHeight
    magentaAspectRatio = magentaWidth / magentaHeight
```

```
    if baseImageAspectRatio < magentaAspectRatio:
        # Приведение ширины уменьшенной версии изображения
        # к ширине пурпурной области:
        widthRatio = magentaWidth / baseImageWidth
        resizedImage = baseImage.resize((magentaWidth,
        int(baseImageHeight * widthRatio) + 1), Image.NEAREST)
    else:
        # Приведение высоты уменьшенной версии изображения
        # к высоте пурпурной области:
        heightRatio = magentaHeight / baseImageHeight
        resizedImage = baseImage.resize((int(baseImageWidth *
        heightRatio) + 1, magentaHeight), Image.NEAREST)

    # Замена пурпурной области уменьшенной версией изображения:
    for x in range(magentaLeft, magentaRight + 1):
        for y in range(magentaTop, magentaBottom + 1):
            if baseImage.getpixel((x, y)) == magentaColor:
                pix = resizedImage.getpixel((x - magentaLeft, y - magentaTop))
                baseImage.putpixel((x, y), pix)

    # РЕКУРСИВНЫЙ СЛУЧАЙ:
    return makeDroste(baseImage, stopAfter=stopAfter - 1)

recursiveImage = makeDroste('museum.png')
recursiveImage.save('museum-recursive.png')
recursiveImage.show()
```

Перед запуском проекта поместите графический файл в папку с файлом `drostemaker.py`. Приложение сохранит рекурсивную картинку под именем `museum-recursive.png`, а затем откроет ее в программе для просмотра изображений. Если вы хотите применить эту программу к собственной иллюстрации, часть которой вы закрасили пурпурным цветом, замените `museum.png` в `makeDroste('museum.png')` в конце исходного кода именем вашего файла, а `museum-recursive.png` в `save('museum-recursive.png')` — именем, под которым вы хотите сохранить получившееся в результате работы интерпретатора изображение.

Настройка

Программа для создания эффекта Дросте предусматривает только одну функцию, `makeDroste()`, которая принимает объект Pillow `Image` или строку с именем графического файла и возвращает объект Pillow `Image`, в котором пурпурные пикселы рекурсивно заменяются уменьшенной версией исходной картинки:

```
from PIL import Image

def makeDroste(baseImage, stopAfter=10):
    # Если baseImage представляет собой строку с именем файла изображения,
    # загрузите это изображение:
    if isinstance(baseImage, str):
        baseImage = Image.open(baseImage)
```

Программа стартует с импорта класса `Image` из библиотеки Pillow (соответствующий модуль Python называется `PIL`). В функции `makeDroste()` мы проверяем, является ли параметр `baseImage` строкой, и если да, то заменяем его объектом Pillow `Image`, загруженным из соответствующего графического файла.

Затем проверяем, равно ли значение параметра `stopAfter` нулю (`0`). Если да, значит, мы достигли одного из базовых случаев алгоритма и функция возвращает объект Pillow `Image` базового изображения:

```
if stopAfter == 0:
    # БАЗОВЫЙ СЛУЧАЙ
    return baseImage
```

Если в момент вызова функции не передается конкретное значение параметра `stopAfter`, то она использует значение по умолчанию, равное `10`. В ходе выполняемого чуть позже рекурсивного вызова `makeDroste()` в качестве аргумента передается значение `stopAfter - 1`, благодаря чему оно постепенно уменьшается, приближаясь к базовому случаю, где оно равно `0`.

Например, при передаче значения `0` для `stopAfter` функция сразу же возвращает изображение, идентичное исходному. При значении `1` функция заменяет пурпурную область рекурсивным изображением один раз, выполняет один рекурсивный вызов, достигает базового случая и завершает свою работу. При значении `2` функция выполняет два рекурсивных вызова и т. д.

Задание этого параметра предотвращает переполнение стека в тех случаях, когда пурпурная область очень большая. Кроме того, лучше задать значение меньше `10`, чтобы ограничить количество рекурсивных изображений, помещенных в базовое. Например, четыре изображения, представленные на рис. 14.2, были созданы путем передачи значений `0`, `1`, `3` и `7` для параметра `stopAfter`.

Затем проверяется цветовая модель базового изображения. Это может быть `RGB` в случае с картинкой, состоящей из красных, зеленых и синих пикселов, или `RGBA`, если она предусматривает альфа-канал. *Альфа-значение* определяет степень прозрачности пиксела. Код данного фрагмента программы выглядит следующим образом:

```
# Пурпурный цвет имеет максимальное значение красного/синего/альфа-каналов
# и нулевое значение зеленого канала:
if baseImage.mode == 'RGBA':
    magentaColor = (255, 0, 255, 255)
elif baseImage.mode == 'RGB':
    magentaColor = (255, 0, 255)
```

Программа для создания эффекта Дросте должна знать цветовую модель изображения, чтобы найти пурпурные пикселы. Значения каждого канала находятся в диапазоне от `0` до `255`, а пурпурные пикселы имеют максимальное значение красного и синего каналов и нулевое значение зеленого. Кроме того, если цветовая модель

предусматривает альфа-канал, то его значение может составлять от `255` (непрозрачный) до `0` (полностью прозрачный). Переменной `magentaColor` присваивается значение чистого пурпурного цвета в виде кортежа в зависимости от цветовой модели изображения, заданной в `baseImage.mode`.

Поиск пурпурной области

Прежде чем программа сможет рекурсивно вставить изображение в пурпурную область, она должна найти ее границы в исходном изображении. Этот процесс предполагает нахождение крайних левого, правого, верхнего и нижнего пурпурных пикселов.

Хотя сама закрашенная область не обязательно должна представлять собой идеальный прямоугольник, приложению необходимо определить прямоугольную зону, чтобы правильно масштабировать изображение для вставки. Например, на рис. 14.3 показана картина «Мона Лиза», на которой прямоугольник описан вокруг пурпурной области, определяющей границы рекурсии.

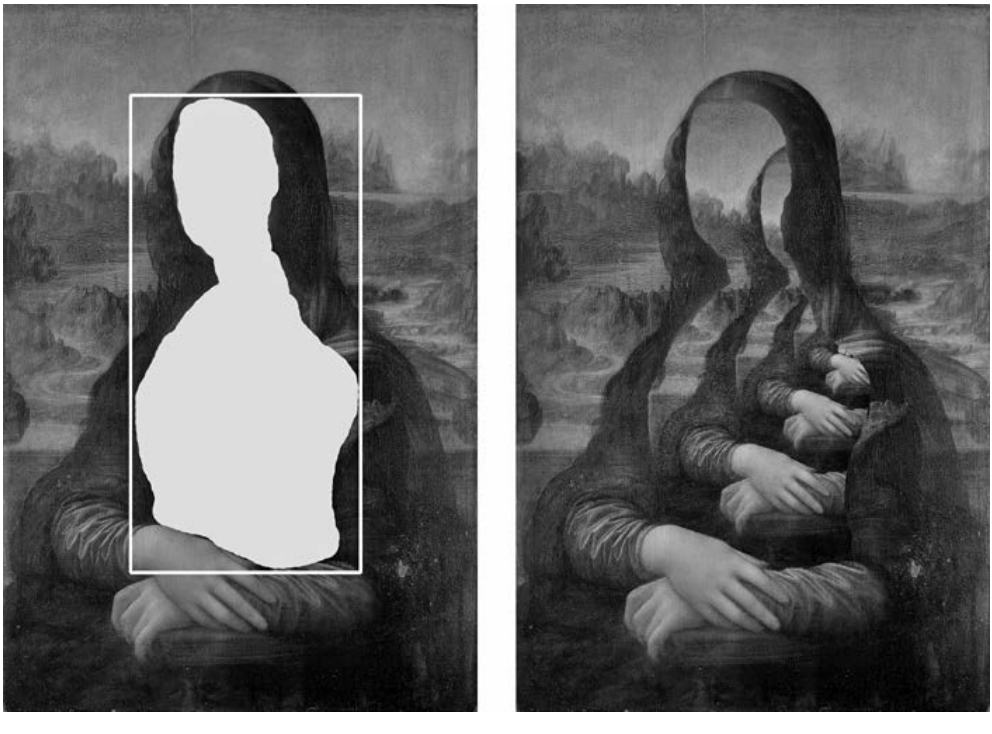


Рис. 14.3. Базовое изображение с пурпурной областью, обведенной белым прямоугольником *(слева)*, и полученное в итоге рекурсивное изображение *(справа)*

Чтобы правильно масштабировать и расположить уменьшенное изображение, программа извлекает значение ширины и высоты базового изображения из атрибута `size` объекта Pillow `Image` в `baseImage`. Следующие строки кода инициализируют четыре переменные, соответствующие четырем краям пурпурной области, — `magentaLeft`, `magentaRight`, `magentaTop` и `magentaBottom` — значением `None`:

```
# Определение размера базового изображения и его пурпурной области:
baseImageWidth, baseImageHeight = baseImage.size
magentaLeft = None
magentaRight = None
magentaTop = None
magentaBottom = None
```

В следующей части кода значения этих четырех переменных заменяются целочисленными значениями координат `x` и `y`:

```
for x in range(baseImageWidth):
    for y in range(baseImageHeight):
        if baseImage.getpixel((x, y)) == magentaColor:
            if magentaLeft is None or x < magentaLeft:
                magentaLeft = x
            if magentaRight is None or x > magentaRight:
                magentaRight = x
            if magentaTop is None or y < magentaTop:
                magentaTop = y
            if magentaBottom is None or y > magentaBottom:
                magentaBottom = y
```

Вложенные циклы `for` перебирают все возможные значения координат `x` и `y` базового изображения, проверяя соответствие цвета пиксела в каждой его точке чистому пурпурному цвету, хранящемуся в `magentaColor`. Затем программа обновляет значение переменной `magentaLeft` при нахождении пурпурного пиксела слева от текущей координаты, хранящейся в данной переменной, и повторяет этот процесс для трех других переменных.

К моменту завершения работы вложенных циклов `for` переменные `magentaLeft`, `magentaRight`, `magentaTop` и `magentaBottom` будут содержать значения, определяющие границы пурпурной области в базовом изображении. Если изображение не содержит пурпурных пикселов, в этих переменных останется исходное значение `None`:

```
if magentaLeft is None:
    # БАЗОВЫЙ СЛУЧАЙ — изображение не содержит пурпурных пикселов
    return baseImage
```

Если после завершения работы вложенных циклов `for` переменная `magentaLeft` (или любая другая из четырех) по-прежнему имеет значение `None`, значит, изображение не содержит пурпурных пикселов. Это базовый случай нашего рекурсивного алгоритма, поскольку с каждым рекурсивным вызовом функции `makeDroste()` пурпурная область становится все меньше и меньше. По достижении базового случая функция возвращает объект Pillow `Image`, содержащийся в `baseImage`.

Изменение размера базового изображения

Нам нужно изменить размер базового изображения ровно настолько, чтобы оно перекрывало пурпурную область. На рис. 14.4 показано уменьшенное полупрозрачное изображение, наложенное на исходное. Оно обрезается таким образом, чтобы перекрыть только те части первоначального, которые заполнены пурпурными пикселами.

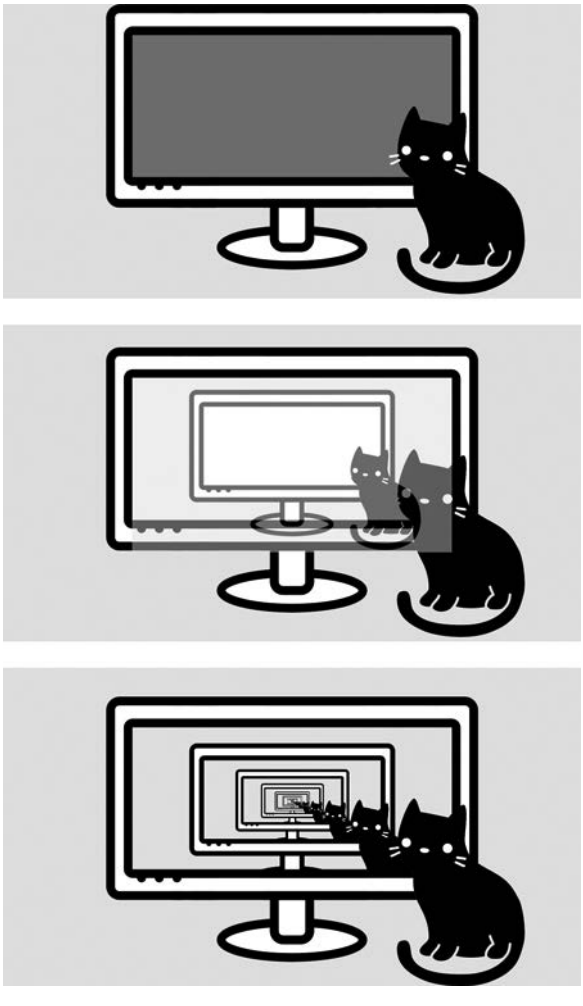


Рис. 14.4. Базовое изображение с закрашенным пурпурным цветом монитором *(вверху)*, полупрозрачное уменьшенное изображение, наложенное поверх базового *(посередине)*, и окончательное рекурсивное изображение, в котором были заменены только пурпурные пикселы *(внизу)*

Мы не можем просто подогнать базовое изображение к размерам пурпурной области, потому что они могут иметь разные соотношения ширины и высоты, из-за чего вставленное рекурсивное изображение рискует получиться растянутым или сжатым, как на рис. 14.5.

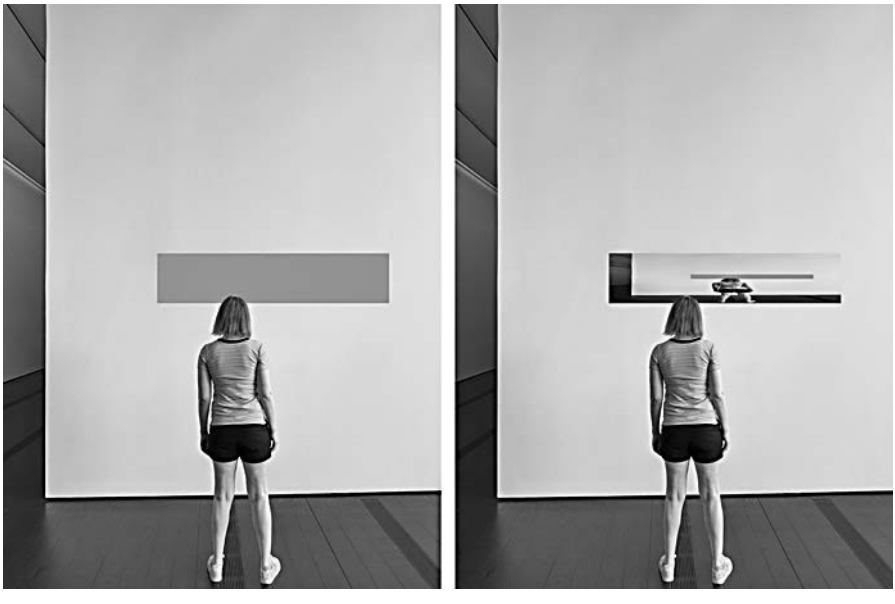


Рис. 14.5. При подгонке базового изображения к размерам пурпурной области рекурсивное изображение может получиться растянутым или сжатым вследствие разницы в соотношении их сторон

Вместо этого необходимо сделать уменьшенное изображение достаточно большим, чтобы оно полностью перекрыло пурпурную область, но при этом сохранило исходное соотношение сторон. Мы можем либо приравнять ширину уменьшенного изображения к ширине пурпурной области, чтобы высота первого была равна или превышала высоту закрашенной зоны, либо приравнять высоту уменьшенного изображения к высоте пурпурной области так, чтобы его ширина была равна ширине этой области или превышала ее.

Чтобы рассчитать правильные размеры уменьшенного изображения, программа должна определить соотношение сторон как базового изображения, так и пурпурной области:

```
# Получение уменьшенной версии базового изображения:
magentaWidth = magentaRight - magentaLeft + 1
magentaHeight = magentaBottom - magentaTop + 1
baseImageAspectRatio = baseImageWidth / baseImageHeight
magentaAspectRatio = magentaWidth / magentaHeight
```

Используя значения переменных `magentaRight` и `magentaLeft`, можно рассчитать ширину пурпурной области. Фрагмент кода `+ 1` представляет собой необходимую, но небольшую корректировку: если правый край закрашенной площади имеет координату *x*, равную 11, а ее левый край имеет координату *x*, равную 10, то ее ширина составляет два пиксела. Чтобы получить это правильное значение, необходимо использовать выражение `magentaRight - magentaLeft + 1`, а не `magentaRight - magentaLeft`.

Поскольку соотношение сторон представляет собой результат деления ширины на высоту, у изображений с малым соотношением сторон ширина превышает высоту, и наоборот. Соотношение сторон 1,0 описывает идеальный квадрат. Следующие строки кода задают размеры уменьшенного изображения после сравнения соотношений сторон базового изображения и пурпурной области:

```
if baseImageAspectRatio < magentaAspectRatio:
    # Приведение ширины уменьшенной версии изображения
    # к ширине пурпурной области:
    widthRatio = magentaWidth / baseImageWidth
    resizedImage = baseImage.resize((magentaWidth,
    int(baseImageHeight * widthRatio) + 1), Image.NEAREST)
else:
    # Приведение высоты уменьшенной версии изображения
    # к высоте пурпурной области:
    heightRatio = magentaHeight / baseImageHeight
    resizedImage = baseImage.resize((int(baseImageWidth *
    heightRatio) + 1, magentaHeight), Image.NEAREST)
```

Если соотношение сторон базового изображения меньше, чем пурпурной области, то уменьшенное изображение подгоняется к ней по ширине. Если же соотношение сторон базового изображения больше, подгонка выполняется по высоте. Затем рассчитывается недостающее измерение: высота базового изображения умножается на отношение ширины закрашенной зоны к ширине базового изображения или ширина базового изображения — на отношение высоты пурпурной области к высоте базового изображения. Таким образом гарантируется, что уменьшенное изображение полностью перекроет область заливки, сохранив исходное соотношение сторон.

Функция `resize()` вызывается один раз для создания нового объекта Pillow `Image`, подогнанного под ширину или высоту пурпурной области базового изображения. Первый аргумент представляет собой кортеж со значениями ширины и высоты нового изображения. Второй — константа `Image.NEAREST` из библиотеки Pillow, которая приказывает методу `resize()` использовать алгоритм ближайшего соседа при изменении размера изображения. Это не позволяет методу `resize()` смешивать цвета пикселов для создания сглаженного изображения.

Подобное недопустимо, потому что так можно смешать пурпурные пикселы с соседними пикселами другого цвета в уменьшенном изображении. Наша функция `makeDroste()` ищет пикселы, имеющие чистый пурпурный цвет — (255, 0, 255) в модели RGB, и игнорирует пикселы, цвет которых хоть сколько-нибудь отличается от этого значения. В результате вокруг пурпурных областей может появиться розоватый контур, который испортит итоговое изображение. Алгоритм ближайшего соседа не допускает такого размытия, а значит, не изменяет цвет необходимых пикселов.

Рекурсивное размещение изображения внутри изображения

После изменения размера базовой картинки можно поместить ее уменьшенную копию поверх исходной. Однако пикселы сжатого изображения должны располагаться только поверх пурпурных пикселов первоначального. Уменьшенное изображение будет расположено таким образом, чтобы его верхний левый угол находился в верхнем левом углу пурпурной области:

```
# Замена пурпурной области уменьшенной версией изображения:
for x in range(magentaLeft, magentaRight + 1):
    for y in range(magentaTop, magentaBottom + 1):
        if baseImage.getpixel((x, y)) == magentaColor:
            pix = resizedImage.getpixel((x - magentaLeft, y - magentaTop))
            baseImage.putpixel((x, y), pix)
```

Два вложенных цикла `for` перебирают все пикселы пурпурной области. Помните, что эта область не обязательно должна представлять собой идеальный прямоугольник, поэтому стоит проверить, является ли пиксел с текущими координатами пурпурным. Если да, то получаем цвет пиксела с соответствующими координатами в уменьшенном изображении и помещаем его поверх базового. К моменту завершения работы двух вложенных циклов `for` пурпурные пикселы на первоначальной картинке будут заменены пикселами ее уменьшенной версии.

Однако ужатое изображение само по себе может содержать пурпурные пикселы, которые станут частью исходного, как показано в верхнем правом углу на рис. 14.2. В данном случае необходимо передать измененный базовый рисунок функции `makeDroste()` в ходе рекурсивного вызова:

```
# РЕКУРСИВНЫЙ СЛУЧАЙ:
return makeDroste(baseImage, stopAfter=stopAfter - 1)
```

Такая строка кода отвечает за рекурсивный вызов функции в нашем алгоритме и является последней в коде функции `makeDroste()`. Этот рекурсивный вызов

обрабатывает новую пурпурную область уменьшенного изображения. Обратите внимание, что в качестве параметра `stopAfter` передается значение `stopAfter - 1`, что гарантирует приближение к базовому случаю, в котором его значение равно `0`.

Наконец, программа для создания эффекта Дросте передает изображение `'museum.png'` функции `makeDroste()` для получения объекта Pillow `Image` с рекурсивным изображением. Сохраним его в виде нового графического файла с именем `museum-recursive.png` и отобразим итог в новом окне:

```
recursiveImage = makeDroste('museum.png')
recursiveImage.save('museum-recursive.png')
recursiveImage.show()
```

Вы можете использовать любые другие имена и изображения на вашем компьютере, которые хотите изменить с помощью данной программы.

Обязательно ли реализовать функцию `makeDroste()` с помощью рекурсии? Вовсе нет. Обратите внимание, что в этой задаче не задействована древовидная структура данных, а алгоритм не выполняет поиск с возвратом. То есть рекурсия представляет собой излишне сложный способ решения поставленной задачи.

Резюме

В этой главе была разработана программа, создающая рекурсивные изображения с эффектом Дросте. Разработанная программа заменяет область базового изображения, закрашенную чистым пурпурным цветом, его уменьшенной версией. Поскольку ужатая копия также содержит пурпурную область, подобная замена будет рекурсивно повторяться до тех пор, пока залитая пурпурным цветом зона не исчезнет.

Базовый случай нашего рекурсивного алгоритма имеет место, когда в изображении не остается пурпурных пикселов, которые можно было бы заменить, или когда значение счетчика `stopAfter` достигает `0`. До тех пор рекурсивные вызовы функции `makeDroste()` будут выполняться.

Вы можете создать эффект Дросте на собственной фотографии, закрасив ее часть пурпурным цветом и обработав с помощью созданной в этой главе программы. Посетитель музея, смотрящий на собственное изображение, кошка перед монитором компьютера, на котором запечатлена кошка перед монитором компьютера, и портрет безликой Моны Лизы — это лишь несколько примеров сюрреалистических иллюстраций, которые можно сгенерировать, используя данную рекурсивную программу.

Дополнительные источники информации

Статья «Википедии» об эффекте Дросте (https://ru.wikipedia.org/wiki/Эффект_Дросте) содержит примеры картинок, в которых он используется. Литография голландского художника Маурица Эшера «Картинная галерея» (Print Gallery) является известным примером изображения, содержащего само себя. Подробнее о нем вы можете узнать в статье «Википедии» по ссылке https://en.wikipedia.org/wiki/Print_Gallery_(M._C._Escher).

В видео под названием *The Neverending Story (and Droste Effect)*, опубликованном на YouTube-канале Numberphile, доктор Клиффорд Столл (Clifford Stoll) обсуждает рекурсию и иллюстрацию с банки какао Droste (https://youtu.be/EeuLDnOupCI).

Руководство по работе с библиотекой Pillow можно изучить в главе 19 моей книги «Автоматизация рутинных задач с помощью Python», доступной по адресу https://automatetheboringstuff.com/2e/chapter19.

Эл Свейгарт

Рекурсивная книга о рекурсии

Перевел с английского С. Черников

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*А. Аверьянов, Е. Рафалюк-Бузовская*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

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

Дата изготовления: 04.2023. Наименование: книжная продукция. Срок годности: не ограничен.

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

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

Подписано в печать 27.02.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 800. Заказ 0000.