Рекурсия – одно из фундаментальных понятий в информатике и ключевой метод программирования, позволяющий многократно выполнять похожие вычисления. Несмотря на важность рекурсии для разработки алгоритмов и на то, что многие разработчики согласны с тем, что рекурсия трудна для начинающих, большинство книг по программированию не освещают эту тему подробно.

Данная книга призвана восполнить этот пробел и содержит подробное и всестороннее введение в рекурсию. Подробно анализируя широкий спектр вычислительных задач различной сложности, книга послужит полезным руководством для всех тех, кто хочет научиться думать и программировать рекурсивно.

Издание содержит специальные главы о наиболее распространенных типах рекурсии (линейной, хвостовой, множественной), а также о парадигмах разработки алгоритмов, где преобладает рекурсия («разделяй и властвуй» и перебор с возвратами). Поэтому его можно использовать как во вводных, так и в углублённых курсах по программированию и разработке алгоритмов. Книга освещает также низкоуровневые вопросы выполнения рекурсивных программ, отношение между рекурсией и итерацией, а также включает в себя большую главу по теоретической оценке стоимости вычисления рекурсивных программ, предлагая читателям возможность попутно вспомнить или изучить основы математики. И наконец, в книге рассматриваются комбинаторные задачи и взаимная рекурсия.

Примеры кода написаны на языке Python 3, но они достаточно просты для понимания также читателями, имеющими опыт работы с другими языками программирования. Преподавателям доступны решения более чем 120 упражнений в конце глав книги.

Помимо преподавателей и студентов книга будет полезна программистам-любителям, а также профессионалам. Первые почерпнут много познавательного и занимательного, например, из решений задач-головоломок, а вторые найдут в книге то, что ещё не встречалось в их практике.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

www.дмк.рф

CRC Press
Taylor & Francis Group

ISBN 978-5-97060-703-9

Введение в рекурсивное программирование

Мануэль Рубио-Санчес

Введение в рекурсивное программирование

Мануэль Рубио-Санчес

Введение в рекурсивное программирование

Introduction to Recursive Programming

Manuel Rubio-Sánchez

CRC Press
Taylor & Francis Group

Введение в рекурсивное программирование

Мануэль Рубио-Санчес

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2019

УДК 004.02
ББК 32.972
Р82

Р82 Мануэль Рубио-Санчес
Введение в рекурсивное программирование / пер. с англ. Е. А. Борисова. – М.: ДМК Пресс, 2019. – 436 с.: ил.

ISBN 978-5-97060-703-9

Книга охватывает почти весь круг теоретических и практических вопросов, относящихся к рекурсии и рекурсивному программированию, что делает её прекрасным дополнением к уже существующим немногочисленным книгам на эту тему. На множестве примеров и задач – от простых к сложным – читатель постепенно погружается в рекурсию, учится мыслить рекурсивно и, отталкиваясь от декларативной парадигмы программирования, создавать рекурсивные алгоритмы с использованием пошаговой методики и специальных схем декомпозиции задач. При этом автор беспристрастно сопоставляет рекурсивные алгоритмы с итерационными, отмечая достоинства и недостатки тех и других.

Все алгоритмы в книге реализованы на языке Python 3.

Издание предназначено студентам вузов, преподавателям, а также широкому кругу разработчиков, желающих эффективно применять рекурсивные алгоритмы в своей работе.

УДК 004.02
ББК 32.973

ISBN 978-1-4987-3528-5 (англ.)
ISBN 978-5-97060-703-9 (рус.)

Будущим поколениям

Оглавление

Предисловие

Рекурсия – одно из самых фундаментальных понятий в информатике и ключевая методика программирования, позволяющая, подобно итерации, многократно повторять вычисления. Это достигается за счёт использования методов, вызывающих самих себя, когда решение исходной задачи сводится к решению нескольких экземпляров той же самой задачи, но меньшего размера. Крайне важно то, что рекурсия – мощный подход к решению задач, позволяющий программистам разрабатывать лаконичные, интуитивно понятные и изящные алгоритмы.

Несмотря на значимость рекурсии для создания алгоритмов, большинство книг по программированию не уделяет внимания её деталям. Обычно они посвящают ей лишь одну-единственную главу или коротенький раздел, которых зачастую недостаточно для освоения понятий, необходимых для овладения предметом. Исключениями являются книги [11], [14], [15], посвящённые исключительно рекурсии. Данная книга также рассматривает рекурсию со всех сторон, но несколько отличается от предыдущих.

Многие преподаватели программирования и исследователи в области обучения информатике признают, что рекурсия трудна для студента-новичка. С учётом этого книга содержит несколько элементов, усиливающих ее педагогический эффект. Во-первых, перед погружением в более сложный материал она предлагает множество простых задач для закрепления основных понятий. Кроме того, одно из основных достоинств книги – использование пошаговой методики и специально созданных схем, сопровождающих и иллюстрирующих процесс разработки рекурсивных алгоритмов. Книга также содержит специальные главы по комбинаторным задачам и взаимной рекурсии. Эти темы позволяют расширить понимание рекурсии студентами, побуждая их применять освоенные понятия несколько иначе или более изощрённо. Наконец, вводные курсы программирования обычно сосредоточиваются на императивной парадигме программирования, когда студенты изучают прежде всего итерацию, усваивая то и овладевая тем, *как* работают программы. Рекурсия же подразумевает совсем иной способ мышления, когда упор делается на то, *что* вычисляют программы. В связи с этим некоторые исследователи призывают при объяснении рекурсии не раскрывать или откладывать на потом механизмы работы рекурсивных

программ (поток управления, деревья рекурсии, программный стек или связь между итерацией и хвостовой рекурсией), так как усвоенные идеи и приобретённые навыки итеративного программирования могут существенно помешать овладению рекурсией и декларативным программированием. По этой причине темы, связанные с итерацией и исполнением программы, раскрываются в конце книги, когда читатель уже овладел разработкой рекурсивных алгоритмов с чисто декларативных позиций.

В книге также есть большая глава по теоретическому анализу стоимости вычислений рекурсивных программ. С одной стороны, она содержит обширный материал о рекуррентных соотношениях – основном математическом инструменте анализа рекурсивных алгоритмов и времени их выполнения. С другой стороны, она содержит раздел с предварительными математическими сведениями, в которых даётся обзор понятий и свойств, необходимых не только для решения рекуррентных соотношений, но и для понимания условий и решений вычислительных задач этой книги. В связи с этим раздел предоставляет ещё и возможность попутно изучить или вспомнить немного элементарной математики. Желательно, чтобы читатель изучил этот материал, так как он важен во многих областях информатики.

Примеры кода написаны на языке Python 3, который сегодня является, видимо, самым популярным языком для вводных курсов программирования в ведущих университетах. Все они были проверены в основном в SPYDER (Scientific PYthon Development EnviRonment – научная среда разработки на языке Python). Читатель должен понимать, что цель книги не в изучении языка Python, а в овладении при решении задач навыками рекурсивного мышления. Поэтому такие аспекты, как простота и удобочитаемость кода, были важнее его эффективности. По этой причине код не содержит расширенных возможностей языка Python. Так что студенты, знакомые с такими языками программирования, как C++ или Java, должны без усилий понять этот код. Конечно, методы из данной книги могут быть реализованы различными способами, и читатели вольны писать более эффективные версии с включением более сложных конструкций языка Python или разрабатывать альтернативные алгоритмы. Наконец, в книге приводятся рекурсивные варианты итерационных алгоритмов, которые обычно сопутствуют другим хорошо известным рекурсивным задачам. Например, в книге приводятся рекурсивные версии метода разделения Хоара, используемого в алгоритме быстрой сортировки, или метода слияния в алгоритме сортировки слиянием.

В конце каждой главы предлагаются многочисленные упражнения, полные решения которых включены в руководство преподавателя, доступное на официальном веб-сайте книги (см. www.crcpress.com). Многие из них связаны с задачами из основного текста книги, что делает их подходящими кандидатами для экзаменов и заданий.

Код из данного текста также будет доступен для загрузки на веб-сайте книги. Кроме того, я буду поддерживать дополнительный веб-сайт книги *https://sites.google.com/view/recursiveprogrammingintro/*. Буду более чем признателен читателям за присланные комментарии, предложения по усовершенствованию, альтернативный (более ясный или эффективный) код, версии на других языках программирования или обнаруженные опечатки. Письма присылайте, пожалуйста, по адресу: *recursion.book@gmail.com*.

Целевая аудитория

Основная цель книги – на множестве примеров разнообразных вычислительных задач научить студентов думать и программировать рекурсивно. Она предназначена главным образом для студентов, обучающихся информатике или связанным с ней техническим дисциплинам, которые охватывают программирование и алгоритмы (например, биоинформатика, инжиниринг, математика, физика и т. д.). Книга может быть также полезна для программистов-любителей, студентов массовых открытых сетевых курсов или более опытных профессионалов, которые хотели бы освежить знакомый им материал или взглянуть на него иначе, проще.

Чтобы понять код в книге, студенты должны владеть некоторыми базовыми навыками программирования. Читатель должен быть знаком с такими понятиями базового курса программирования, как выражения, переменные, условные и циклические конструкции, методы, параметры и элементарные структуры данных (массивы или списки). Эти понятия не объясняются в книге. Кроме того, код в книге следует процедурной парадигме программирования и не использует объектно-ориентированные средства. Что касается языка Python, то знание его основ может быть полезно, но совсем не обязательно. Наконец, студент должен владеть математикой в объёме средней школы.

Книга также может оказаться полезной и для преподавателей информатики, причём не только как справочник с большой коллекцией разнообразных задач, но и, принимая во внимание описанные методику и схемы, как пособие по разработке рекурсивных решений. Более

того, преподаватели могут использовать её структуру для организации своих занятий. Книга могла бы использоваться как приемлемый учебник по вводному (CS1/2) курсу программирования, в углублённых курсах – по разработке и анализу алгоритмов (она, например, охватывает такие темы, как «разделяй и властвуй» или перебор с возвратами). Кроме того, поскольку книга закладывает прочный фундамент рекурсии, она может использоваться в качестве дополнительного материала в курсах, посвящённых структурам данных или функциональному программированию. Однако читатель должен иметь в виду, что книга не касается ни структур данных, ни понятий функционального программирования.

Содержание и структура книги

Глава 1 предполагает, что читатель не имеет никакого представления о рекурсии, и вводит основные её понятия, систему обозначений и даёт первые примеры рекурсивного кода.

Глава 2 описывает методику разработки рекурсивных алгоритмов, а также схем, способствующих рекурсивному мышлению, которые иллюстрируют исходную задачу и её декомпозицию (разложение) на меньшие экземпляры той же самой задачи. Это одна из самых важных глав, так как методика и рекурсивные схемы будут использоваться во всей остальной части книги. Желательно, чтобы читатели ознакомились с этой главой независимо от их предыдущих знаний о рекурсии.

Глава 3 даёт обзор основных математических принципов и обозначений. Кроме того, она описывает методы решения рекуррентных соотношений, которые являются основным математическим инструментом для теоретического анализа стоимости вычислений рекурсивных алгоритмов. Глава может быть опущена во вводных курсах по рекурсии. Однако она помещена в начало книги, чтобы заложить почву для оценки и сравнения эффективности различных алгоритмов, что важно в расширенных курсах по разработке и анализу алгоритмов.

Глава 4 посвящена «линейной рекурсии». Этот тип рекурсии приводит к простейшим рекурсивным алгоритмам, когда решение исходной вычислительной задачи вытекает из решения единственного её меньшего экземпляра (подзадачи). Хотя предлагаемые задачи можно легко решить посредством итерации, они идеально подходят для введения основных понятий рекурсии, а также в качестве примеров применения рекурсивной методики и рекурсивных схем.

Глава 5 посвящена особому типу линейной рекурсии, называемой «хвостовой рекурсией», когда рекурсивный вызов методом самого себя является последним действием этого метода. Особенность хвостовой рекурсии заключается в том, что она тесно связана с итерацией. Однако объяснение этой связи будет отложено до главы 11. А эта глава, напротив, посвящена решениям, основанным на чисто декларативном подходе с опорой исключительно на понятие рекурсии.

Преимущества рекурсии над итерацией проявляются главным образом в случае «множественной рекурсии», когда методы могут вызывать себя несколько раз, а алгоритмы основаны на объединении нескольких решений меньших экземпляров той же самой задачи. *Глава 6* вводит множественную рекурсию методами, основанными на известной парадигме разработки алгоритмов «разделяй и властвуй». Хотя часть примеров этой главы может использоваться во вводном курсе программирования, глава в целом уместна скорее для углублённых курсов и более сложных классов алгоритмов.

Глава 7 содержит дополнительные развивающие задачи, относящиеся к пазлам и фракталам. Они также решаются с помощью множественной рекурсии, но не предполагают применения подхода «разделяй и властвуй».

Рекурсия широко используется в комбинаторике – разделе математики, относящемся к подсчёту и применяемом в углублённом анализе алгоритмов. *Глава 8* предлагает использовать рекурсию для решения комбинаторных задач подсчёта, которые обычно не ограничиваются программированием текстов. Эта уникальная глава заставит читателя применять приобретённые навыки рекурсивного мышления к разнообразным семействам задач. Наконец, хотя некоторые примеры относятся к развивающим, многие из их решений появляются в ранних главах. Поэтому такие примеры могут использоваться и во вводном курсе программирования.

Глава 9 вводит понятие «взаимной рекурсии», когда несколько методов вызывает себя косвенно. Их решения гораздо сложнее, так как необходимо думать о нескольких задачах сразу. Тем не менее этот тип рекурсии опирается на те же основные понятия из предыдущих глав.

Глава 10 посвящена низкоуровневым аспектам рекурсивных программ. Она включает такие их аспекты, как трассировка и отладка, стек программы или деревья рекурсии. Кроме этого, она содержит краткое введение в мемоизацию (memorization) и динамическое программирование, которые являются ещё одной важной парадигмой разработки алгоритмов.

К итерации можно привести не только алгоритмы с хвостовой рекурсией; некоторые из них также разрабатываются итеративно. *Глава 11* подробно исследует связь между итерацией и хвостовой рекурсией. Кроме того, она содержит краткое введение во «вложенную рекурсию» и включает стратегию разработки простых функций с хвостовой рекурсией, которые обычно определяются итеративно, но с исключительно декларативным подходом. Эти две последние темы можно отнести к курьёзам рекурсии и опустить во вводных курсах.

Последняя *глава 12* посвящена перебору с возвратами (backtracking), который является ещё одним важным методом проектирования алгоритмов, применяющимся для поиска решения вычислительных задач в больших дискретных пространствах состояний. Такая стратегия обычно используется для решения задач соблюдения (удовлетворения) заданных ограничений (constraint satisfaction) и дискретной оптимизации. Например, в главе рассматриваются такие классические задачи, как расстановка на шахматной доске *N* ферзей, поиск пути в лабиринте, решение судоку или укладка рюкзака 0–1.

Примерные учебные курсы

Можно охватить лишь несколько глав. Для вводных курсов программирования достаточно глав 1, 2, 4, 5. Преподаватель должен решить, включать ли примеры глав 6–9 и освещать ли первый раздел главы 11.

Если студенты уже владеют навыками разработки линейно-рекурсивных методов, то расширенный курс по анализу и разработке алгоритмов мог бы включать главы 2, 3, 5, 6, 7, 9, 11 и 12. Тогда как главы 1 и 4 можно рекомендовать лишь для чтения, чтобы освежить знакомый материал.

В обеих из этих учебных программ глава 8 является необязательной. Наконец, вслед за пройденными главами важно ознакомиться и с главой 11.

Благодарности

Содержание этой книги использовалось в курсах обучения программированию в Университете Рея Хуана Карлоса в Мадриде (Испания). Я благодарен студентам за их отклики и предложения. Также я хотел бы поблагодарить Анхеля Веласкеса и членов исследовательской группы LITE (Лаборатория информационных технологий в образовании) за

полезные замечания по содержанию книги. Ещё хотел бы выразить благодарность Луису Фернандесу (Luís Fernández), профессору информатики Политехнического университета Мадрида, за его советы и опыт по обучению рекурсии. Особая благодарность – Джерту Лэнкриту и сотрудникам Лаборатории компьютерного слуха в Университете Калифорнии, Сан-Диего.

Мануэль Рубио-Санчес,
июль 2017

Глава 1

Основные понятия рекурсивного программирования

Итерация – от человека, рекурсия – от Бога.

– Лоуренс Питер Дейч

Рекурсия – широкое понятие, которое используется в таких разных дисциплинах, как математика, биоинформатика или лингвистика, и присутствует даже в искусстве и в природе. В программировании рекурсия понимается как мощная стратегия, позволяющая разрабатывать простые, компактные и изящные алгоритмы решения вычислительных задач. В данной главе вводятся терминология, обозначения и фундаментальные понятия рекурсии, которые в дальнейшем будут раскрыты в этой книге.

1.1. Распознание рекурсии

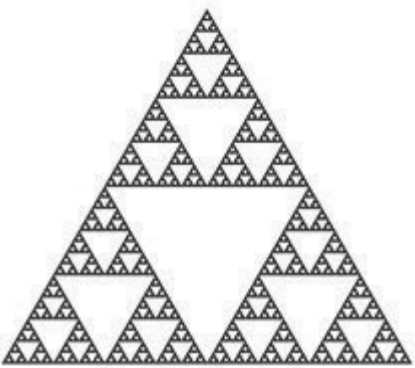
Говорят, что объект или понятие рекурсивны, когда в его состав входят более простые или меньшие подобные ему элементы. Природа даёт множество примеров этого свойства (см. рис. 1.1). Например, ветку дерева можно считать основой для меньших веток, которые отходят от неё и, в свою очередь, состоят из других, меньших, веток, и так далее до почки, листа или цветка. Строение кровеносных сосудов или рек тоже подобно структуре ветвления деревьев, когда бóльшая структура содержит подобные ей элементы, но в меньших масштабах. Другой родственный рекурсивный пример – капуста романеско (Romanesco broccoli), где отдельные маленькие цветки явно напоминают всё растение. Другие примеры включают горные цепи, облака или рисунок кожи животных.

Ветви дерева

Реки с притоками

Брокколи Romanesco

Спиральный эффект Дросте



Треугольник Серпиньского

Кукла-матрёшка

Рис. 1.1. Примеры рекурсивных объектов

Рекурсия также появляется в искусстве. Известный пример – эффект Дросте (Droste-effect), который представляет собой картинку, повторяю-

щуюся внутри себя. Теоретически такой процесс бесконечен, но на практике он, конечно же, заканчивается, когда наименьшая картинка становится настолько малой, что занимает всего один пиксель в цифровом изображении. Создаваемый компьютером фрактал – ещё один тип рекурсивного изображения. Например, треугольник Серпиньского состоит из трёх меньших идентичных треугольников, которые затем делятся ещё на три меньших. Бесконечно повторяя этот процесс, мы обнаружим, что каждый маленький треугольник имеет ту же структуру, что и оригинал. И последний, классический пример иллюстрации рекурсии – набор кукол-матрёшек. В этом ремесле каждая кукла имеет такой размер, чтобы уместиться внутри большей куклы. Отметим, что рекурсивный объект – это не одна полая кукла, а вся вложенная коллекция кукол. Таким образом, рекурсивное мышление предполагает, что вся коллекция кукол может быть описана как одна-единственная (наибольшая) кукла, которая содержит внутри себя меньшую коллекцию кукол.

Хотя в приведённых примерах рекурсивные объекты – явно материальные, рекурсия встречается и в огромном числе абстрактных понятий. В этом отношении рекурсию можно понимать как процесс определения понятий через их собственные определения. Таким способом можно выразить многие математические формулы и определения. Самые очевидные примеры – последовательности, n-й член которых задаётся некоторой формулой или процедурой, использующей предыдущие члены. Рассмотрим следующее рекурсивное определение:

$$s_n = s_{n-1} + s_{n-2}. \tag{1.1}$$

Формула говорит о том, что член последовательности s_n – это просто сумма двух предыдущих членов s_{n-1} и s_{n-2}. Сразу видно, что формула рекурсивна, так как определяемый ею объект s появляется в обеих частях уравнения. Таким образом, элементы последовательности явно определяются через самих себя. Кроме того, заметьте, что рекурсивная формула (1.1) описывает не конкретную последовательность, а целое семейство последовательностей, где каждый её член есть сумма двух предыдущих членов. Чтобы задать конкретную последовательность, мы должны предоставить дополнительную информацию. В данном случае достаточно задать любые два члена последовательности. Как правило, чтобы определить такую последовательность, достаточно задать два первых её члена. Например, если $s_1 = s_2 = 1$, то мы получим последовательность

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots,$$

которая является известной последовательностью чисел Фибоначчи.

Последовательности могут начинаться и с члена s_0.

Последовательность *s* можно считать функцией, которая в качестве аргумента получает положительное целое число *n*, а возвращает *n*-й член этой последовательности. В этом случае функция Фибоначчи, обозначенная просто как *F*, может быть определена как:

$$F(n) = \begin{cases} 1, & \text{если } n = 1, \\ 1, & \text{если } n = 2, \\ F(n-1) + F(n-2), & \text{если } n > 2. \end{cases} \quad (1.2)$$

Всюду в этой книге мы будем использовать для описания функции такие обозначения, когда определения включают два типа выражений или случаев. *Начальные условия* (base cases) соответствуют случаю, когда результат функции может быть получен тривиально без привлечения значений функции от дополнительных параметров. По определению, начальные условия для чисел Фибоначчи – это $F(1) = 1$ и $F(2) = 1$. *Рекурсивные условия* (recursive cases) представляют собой более сложные рекурсивные выражения, включающие обычно определённую функцию от предыдущих значений входных параметров. Функция Фибоначчи имеет одно рекурсивное условие: $F(n) = F(n - 1) + F(n - 2)$, если $n > 2$. Начальные условия необходимы для получения из рекурсивных условий конкретных значений членов последовательности. В заключение следует сказать, что рекурсивное определение может содержать несколько начальных и рекурсивных условий.

Ещё одна функция, которая может быть выражена рекурсивно, – это факториал некоторого неотрицательного целого числа *n*:

$$n! = 1 \times 2 \times \cdots \times (n - 1) \times n.$$

В этом случае рекурсивность функции не столь очевидна из-за явного отсутствия в правой части определения факториала. Но поскольку $(n - 1)! = 1 \times 2 \times \cdots \times (n - 1)$, можно переписать формулу рекурсивно: $n! = (n - 1)! \times n$. Наконец, по определению $0! = 1$, что следует из рекурсивной формулы для $n = 1$. Таким образом, функция факториала может быть определена рекурсивно как

$$n! = \begin{cases} 1, & \text{если } n = 0, \\ (n-1)! \times n, & \text{если } n > 0. \end{cases} \quad (1.3)$$

Рассмотрим также задачу вычисления суммы первых *n* положительных целых чисел. Соответствующая функция $S(n)$ может быть, очевидно, определена как:

$$S(n) = 1 + 2 + \cdots + (n-1) + n. \tag{1.4}$$

И снова мы пока не видим в правой части определения слагаемого, содержащего функцию S. Однако первые $n - 1$ членов можно сгруппировать так, чтобы получить $S(n - 1) = 1 + 2 + \ldots + (n - 1)$, что приводит к следующему рекурсивному определению:

$$S(n) = \begin{cases} 1, & \text{если } n = 1, \\ S(n-1) + n, & \text{если } n > 1. \end{cases} \tag{1.5}$$

Отметим, что $S(n - 1)$ – *подзадача*, подобная $S(n)$, но *более простая,* так как для получения её результата требуется меньшее количество операций. Таким образом, говорят, что подзадача имеет меньший *размер* (*размерность*). Кроме того, мы говорим, что выполнена *декомпозиция* (*разложение*) исходной задачи ($S(n)$) на *меньшие* для формирования рекурсивного определения. Таким образом, $S(n - 1)$ – меньший *экземпляр* исходной задачи.

Другой математический объект, рекурсивность которого не вполне очевидна, – неотрицательные целые числа. Эти числа можно разложить и определить рекурсивно через меньшие числа разными способами. Например, неотрицательное целое число n можно представить как предшествующее ему число плюс один:

$$n = \begin{cases} 0, & \text{если } n = 0, \\ predesessor(n) + 1, & \text{если } n > 0. \end{cases}$$

Заметим, что в рекурсивном условии n находится по обе стороны от знака равенства. Кроме того, если мы считаем, что функция *predecessor* обязательно возвращает неотрицательное целое число, то она неприменима к 0. Поэтому определение становится законченным только вместе с тривиальным начальным условием для $n = 0$.

Ещё один способ определения (неотрицательных) целых чисел – считать их упорядоченной последовательностью цифр. Например, число 5342 может быть конкатенацией следующих пар меньших чисел:

(5, 342), (53, 42), (534, 2).

На практике простейший способ декомпозиции целого числа – это представление его в виде соединения наименьшей его значащей цифры с остальной его частью. Поэтому целое число можно определить следующим образом:

$$n = \begin{cases} n, & \text{если } n < 10, \\ (n//10) \times 10 + (n\%10), & \text{если } n \geq 10, \end{cases}$$

где // и % – операции вычисления частного и остатка целочисленного деления соответственно, в обозначениях языка Python. Например, если n = 5342, то частное (n // 10) = 534, а остаток (n % 10) = 2 – наименьшая значащая цифра n. Понятно, что само число n можно восстановить, умножив частное на 10 и добавив к полученному результату остаток. Начальное условие имеет дело только с однозначными числами, которые, конечно же, не могут быть подвержены разложению.

Математика просто изобилует рекурсивными выражениями. Так, например, они часто используются для описания свойств функций. Следующее рекурсивное выражение говорит о том, что производная суммы функций есть сумма её производных:

$$[f(x) + g(x)]' = [f(x)]' + [g(x)]'.$$

В этом случае рекурсивный объект – это производная функции, обозначаемая как [-]', но не сами функции $f(x)$ и $g(x)$. Заметьте, что формула явно указывает на имеющую место декомпозицию, где некоторая первичная функция (являющаяся входным параметром производной функции) разложена на сумму функций $f(x)$ и $g(x)$.

Структуры данных тоже можно считать рекурсивными объектами. На рис. 1.2 показано, как могут быть подвергнуты рекурсивной декомпозиции структуры данных список и дерево.

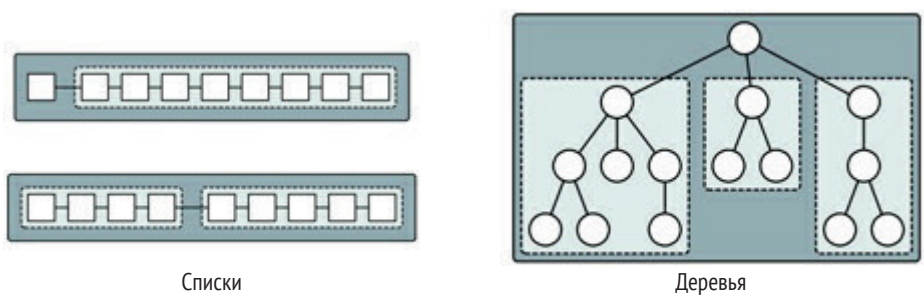


Рис. 1.2. Рекурсивная декомпозиция списков и деревьев

Список может состоять из одного элемента плюс ещё список (это обычное определение списка как абстрактного типа данных). Или же он может быть поделён на несколько списков (в этом, более широком контексте список – это любая коллекция элементов данных, которые линейно упорядочены в определённой последовательности, как в спис-

ках, массивах, кортежах и т. д.). Дерево же состоит из родительской вершины и множества (или списка) поддеревьев, корневой узел которых – прямой потомок исходной родительской вершины. Рекурсивные определения структур данных заканчиваются пустыми (начальными) значениями. Например, список из одного элемента мог бы состоять из этого элемента плюс пустой список. Наконец, обратите внимание, что в этих схемах более темные поля представляют рекурсивный объект в целом, тогда как светлые обозначают меньшие и подобные ему экземпляры.

Рекурсия может также использоваться для определения слов в словарях. Это может показаться невозможным, так как нас учат в школе, что толкование слова в словаре не может содержать внутри себя это же слово. Однако многие понятия могут быть определены правильно именно так. Рассмотрим термин «потомок» определенного предка. Обратите внимание, что он может быть определён так: некто, являющийся либо ребёнком своего предка, либо *потомком* любого из детей своего предка. В этом случае мы можем определить рекурсивную конструкцию, в которой множество потомков можно представить (генеалогическим) древом, как на рис. 1.3, где тёмные блоки содержат всех потомков общего предка (корня дерева), а светлые – потомков детей предка.

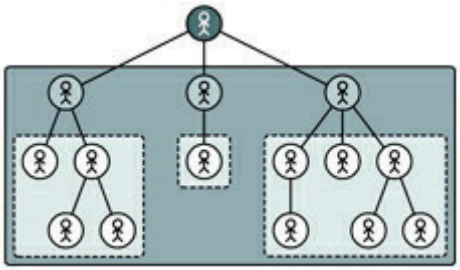


Рис. 1.3. Генеалогическое древо потомков человека – его детей и потомков их детей

1.2. Декомпозиция задачи

Вообще говоря, основная задача рекурсивного мышления и программирования – дать наши собственные рекурсивные определения объектам, понятиям, функциям, задачам и т. д. И если первый шаг обычно сводится к выявлению начальных условий, то главная задача состоит в определении рекурсивных условий. В связи с этим важно усвоить понятия (a) декомпозиции задачи и (b) индукции, которые мы вкратце рассмотрим в этой главе.

Цель книги будет заключаться в разработке рекурсивных алгоритмов решения *вычислительных задач*. Их следует понимать как вопросы, на которые могут ответить компьютеры и которые задаются в виде высказываний, описывающих отношения между заданным набором входных значений или параметров и множеством выходных значений, результатов или решений. Например: «Дано некое положительное целое число n. Найти сумму первых n положительных целых чисел» – это формулировка вычислительной задачи с одним входным параметром (n) и одним выходным значением, определяемым как $1 + 2 + \ldots + (n - 1) + n$. Экземпляр задачи – это определённый набор допустимых входных значений, которые позволяют нам получить решение задачи. Тогда как *алгоритм* – это логическая процедура, определяющая пошаговый процесс вычислений для получения по заданным входным значениям выходных значений. Таким образом, алгоритм определяет, как решить задачу. Заслуживает внимания то, что вычислительные задачи могут решаться различными алгоритмами. Цель этой книги состоит в том, чтобы объяснить, как разработать и реализовать рекурсивные алгоритмы и программы, где основным этапом является декомпозиция вычислительной задачи.

Декомпозиция – важное понятие в информатике, играющее главную роль не только в рекурсивном программировании, но и в решении задач вообще. Суть её состоит в разложении сложной задачи на меньшие и более простые подзадачи, которые проще выразить, вычислить, закодировать и решить. После чего решения подзадач используются для получения решения более сложной исходной задачи.

В контексте рекурсивного решения задач и программирования декомпозиция подразумевает разложение вычислительной задачи на несколько подзадач, некоторые из которых подобны исходной, как показано на рис. 1.4.

Отметим, что для решения задачи может потребоваться решение других, дополнительных задач, которые не являются подобными исходной. В книге будет несколько таких задач, но в этой вводной главе мы приведём только такие примеры, где исходные задачи разбиваются только на себе подобные.

В качестве первого примера снова рассмотрим задачу вычисления суммы первых n положительных целых чисел, обозначаемой как $S(n)$ и выраженной формулой (1.4). Есть несколько способов разбить задачу на меньшие подзадачи и сформировать рекурсивное определение $S(n)$. Во-первых, она зависит только от входного параметра n, который опре-

деляет также размер задачи. В этом примере начальное условие связано с наименьшим целым положительным числом $n = 1$, и очевидно, что $S(1) = 1$ – наименьший экземпляр задачи. В дальнейшем при рассмотрении подзадач нам, возможно, придётся уточнить начальное условие задачи. Поэтому мы должны подумать, каким образом мы будем уменьшать размер задачи – входной параметр n.

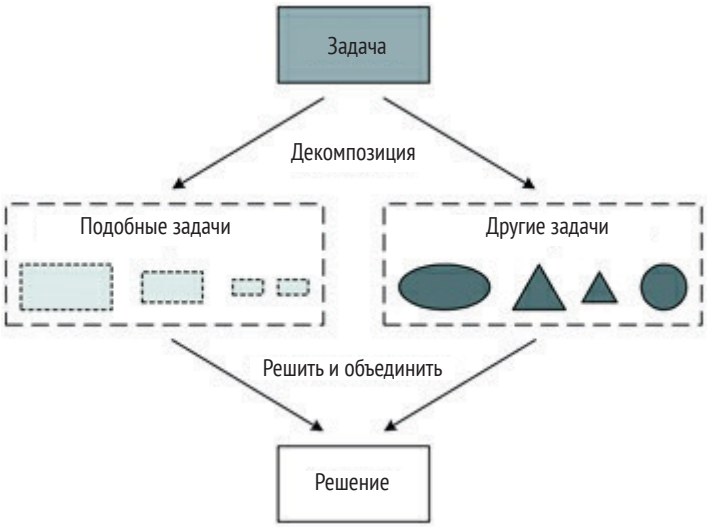


Рис. 1.4. Рекурсивное решение задачи

Первый вариант – уменьшать n на единицу. В этом случае наша цель – определить $S(n)$, используя подзадачу $S(n - 1)$. Соответствующее рекурсивное решение, полученное алгебраически в разделе 1.1, приведено в (1.5). Рекурсивное условие можно также вывести из графического представления задачи. Например, можно подсчитать общее число блоков в «треугольной» пирамиде, которая содержит n блоков в первом (нижнем) своём ряду, $n - 1$ – во втором и т. д. (верхний n-й ряд состоит только из одного блока), как показано на рис. 1.5(a) для $n = 8$.

Для рекурсивной декомпозиции задачи нам нужно найти подобные ей задачи. В этом случае нетрудно найти в исходном треугольнике меньшие подобные треугольники. Например, на рис. 1.5(b) показан треугольник высотой $n - 1$, включающий все блоки исходного треугольника, за исключением n блоков первого ряда. Поскольку этот меньший треугольник содержит ровно $S(n - 1)$ блоков, из этого следует, что $S(n) = S(n - 1) + n$.

Другой вариант декомпозиции с подзадачей суммирования $n - 1$ членов заключается в рассмотрении суммы $2 + \ldots + (n - 1) + n$. Однако тут важно отметить, что эта задача не подобна $S(n)$. Очевидно, что это не сумма первых положительных целых чисел, а частный случай более общей задачи суммирования всех целых чисел от некоторого начального значения m до некоторого большего значения n: $m + (m + 1) + \ldots + (n - 1) + n$, где $m \leqslant n$. Различие между обеими задачами можно также пояснить графически. На рис. 1.5 этой общей задаче соответствовал бы прямой трапециоид треугольника. В итоге можно вычислить сумму первых n положительных целых чисел, используя эту, более общую задачу, если задать $m = 1$. Однако её рекурсивное определение сложнее, так как требует вместо одного два входных параметра m и n.

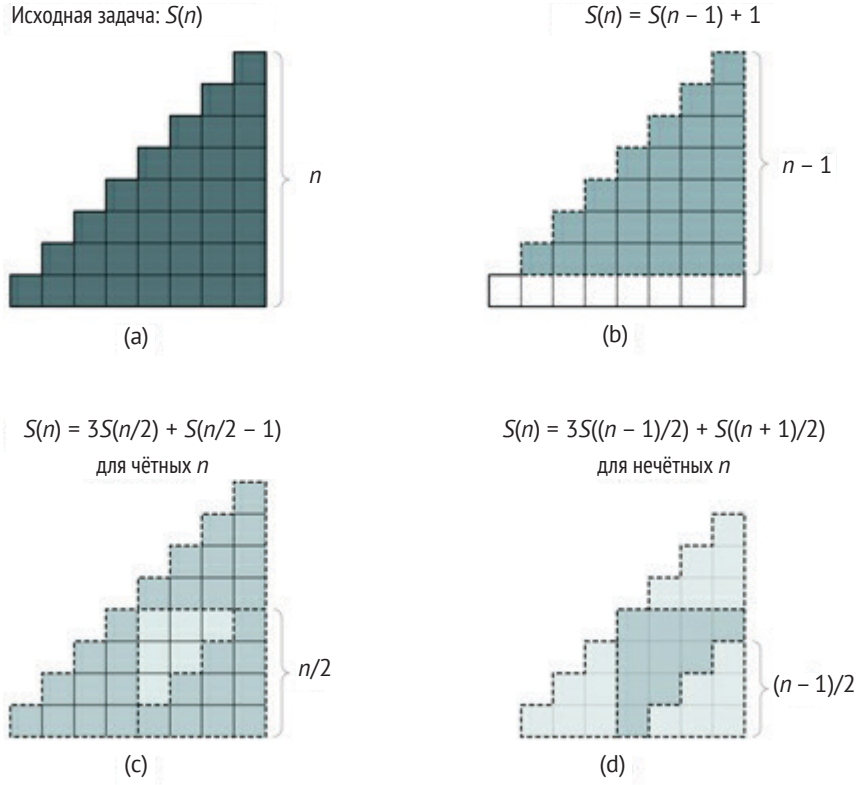


Рис. 1.5. Декомпозиции суммы первых положительных целых чисел

Другие варианты декомпозиции этой задачи состоят в том, чтобы уменьшать n с шагом, большим 1. Например, все входные значения n можно разбить на чётные и нечётные, чтобы получить декомпозицию,

приведённую на рис. 1.5(c) и 1.5(d). Если n – чётное, то внутри большого треугольника, соответствующего $S(n)$, можно разместить три треугольника высотой $n/2$. И поскольку оставшийся блок – тоже треугольник высотой $n/2 - 1$, рекурсивную формулу можно записать как $S(n) = 3S(n/2) + S(n/2 - 1)$. С другой стороны, если n – нечётное, то внутри треугольника можно разместить три треугольника высотой $(n - 1)/2$ и один высотой $(n + 1)/2$. Так что в этом случае рекурсивная формула примет вид $S(n) = 3S((n - 1)/2) + S((n + 1)/2)$. Окончательная рекурсивная функция:

$$S(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2, & \text{если } n = 2, \\ 3S(n/2) + S(n/2 - 1), & \text{если } n > 2 \text{ и } n \text{ – чётное}, \\ 3S((n-1)/2) + S((n+1)/2), & \text{если } n > 2 \text{ и } n \text{ – нечётное}. \end{cases} \quad (1.6)$$

Важно отметить, что определение нуждается в дополнительном начальном условии для $n = 2$. Иначе в рекурсивном условии для чётных n мы получили бы $S(2) = 3S(1) + S(0)$. Но по условию задачи $S(0)$ не определена, поскольку входными значениями для S должны быть только положительные целые числа. Таким образом, новое начальное условие исключает применение рекурсивной формулы для $n = 2$.

За счёт уменьшения размера задачи её подзадачи становятся значительно меньше исходной и потому могут решаться намного быстрее. Грубо говоря, если число подзадач, которые должны быть решены, мало и существует возможность разумного объединения их решений, то такая стратегия может привести к существенно более быстрым алгоритмам решения исходной задачи. Однако в этом частном примере код для (1.6) не обязательно эффективнее кода для (1.5). Интуитивно потому, что (1.6) требует решения двух подзадач (с различными параметрами), тогда как (1.5) содержит только одну подзадачу. Оценке стоимости выполнения рекурсивных алгоритмов посвящена глава 3.

В последнем способе суммирования первых n положительных целых чисел исходная задача делилась на две подзадачи меньшего размера, где новые входные параметры удовлетворяют заданным начальным условиям. В самом общем случае можно разбивать задачи на любое количество более простых подзадач до тех пор, пока новые параметры тесно связаны со значениями, заданными в начальных условиях. Например, рассмотрим следующее альтернативное рекурсивное определение функции Фибоначчи (эквивалент (1.2)):

$$F(n) = \begin{cases} 1, & \text{если } n = 1, \\ 1, & \text{если } n = 2, \\ 1 + \sum_{i=1}^{n-2} F(i), & \text{если } n > 2, \end{cases} \tag{1.7}$$

где $\sum_{i=1}^{n-2} F(i)$ есть сумма $F(1) + F(2) + \ldots + F(n-2)$ (см. раздел 3.1.4). В этом примере для некоторого значения n размера исходной задачи рекурсивное условие использует результаты всех меньших подзадач размером от 1 до $n - 2$.

В последнем примере декомпозиции задачи мы будем использовать списки, которые позволят нам обращаться к отдельным его элементам посредством числовых индексов (во многих языках программирования такую структуру данных называют «массивом», тогда как в Python это просто «список»). Задача заключается в суммировании значений элементов списка, обозначаемого как **a** и содержащего n чисел. Формально задача может быть записана как:

$$s(\mathbf{a}) = \sum_{i=0}^{n-1} \mathbf{a}[i] = \mathbf{a}[0] + \mathbf{a}[1] + \ldots + \mathbf{a}[n-1], \tag{1.8}$$

где $a[i]$ – $(i + 1)$-й элемент списка, поскольку 1-й элемент списка имеет индекс 0. На рис. 1.6(a) изображён частный случай списка из 9 элементов.

Что касается обозначений, то в этой книге мы будем считать, что *подсписок* некоторого списка **a** – это коллекция *смежных* элементов **a**, если явно не оговаривается иное. Напротив, в *подпоследовательности* некоторой последовательности **s** её элементы появляются в том же порядке, что и в **s**, но они не обязаны быть смежными. Другими словами, подпоследовательность может быть получена из исходной последовательности **s** удалением некоторых элементов **s** без изменения порядка следования оставшихся элементов.

Декомпозиция задачи заключается в уменьшении её размера на единицу. С одной стороны, список можно разбить на подсписок из первых $n - 1$ элементов (**a**[0: $n - 2$], где **a**[i : j] – подсписок от **a**[i] до **a**[$j - 1$] в обозначениях языка Python) и единственный элемент (**a**[$n - 1$]), соответствующий последнему числу в списке (см. рис.1.6(b)). В этом случае задачу можно определить рекурсивно таким образом:

$$s(\mathbf{a}) = \begin{cases} 0, & \text{если } n = 0, \\ s(\mathbf{a}[0{:}n-2]) + \mathbf{a}[n-1], & \text{если } n > 0. \end{cases} \tag{1.9}$$

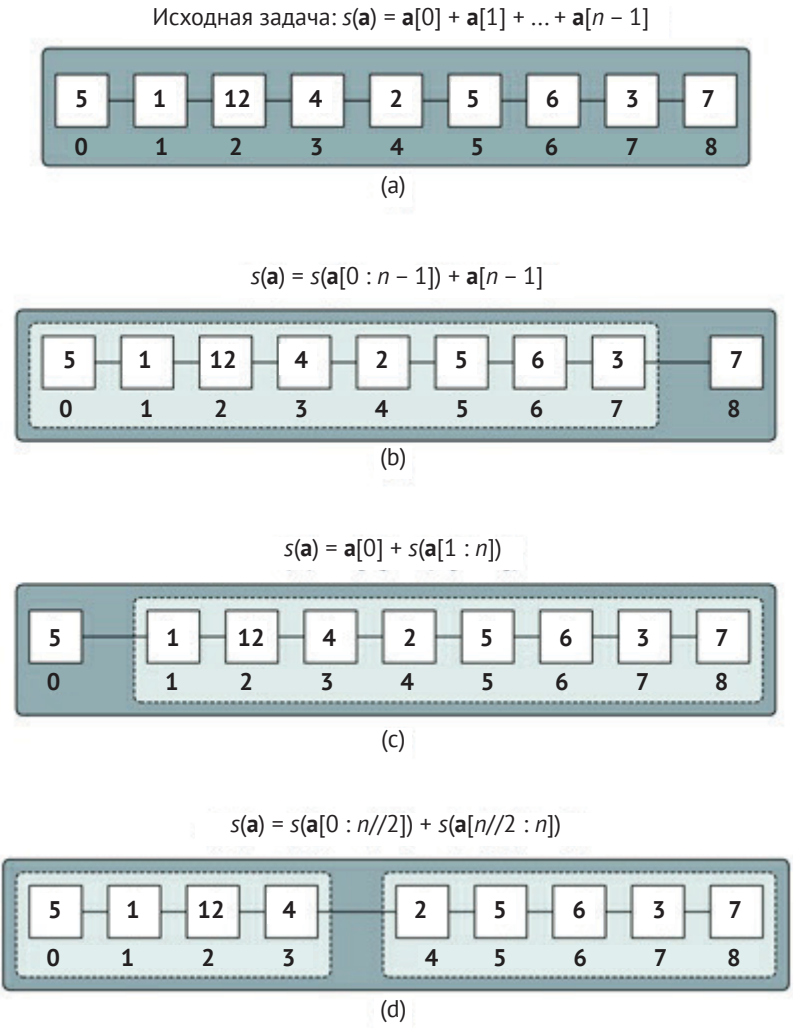


Рис. 1.6. Декомпозиции суммы элементов списка **a** из $n = 9$ чисел

В рекурсивном условии подзадача решается, естественно, для подсписка размером $n - 1$. Начальное условие рассматривает тривиальную ситуацию, когда список пуст и совсем не требует суммирования.

Вторым возможным начальным условием может быть $s(\mathbf{a}) = \mathbf{a}[0]$ для $n = 1$. Однако оно было бы избыточным при такой декомпозиции и поэтому не обязательно. Обратите внимание, что при $n = 1$ функция добавляет $\mathbf{a}[0]$ к нулевому результату применения функции к пустому списку. Таким образом, второе начальное условие можно опустить для краткости.

С другой стороны, исходный список можно также представить как его первый элемент $\mathbf{a}[0]$ и меньший список $\mathbf{a}[1:n]$ (см. рис. 1.6(c)). В этом случае задача может быть записана рекурсивно как:

$$s(\mathbf{a}) = \begin{cases} 0, & \text{если } n = 0, \\ \mathbf{a}[0] + s(\mathbf{a}[1:n]), & \text{если } n > 0. \end{cases} \tag{1.10}$$

Хотя обе декомпозиции очень похожи, код для каждой из них может весьма отличаться в разных языках программирования. В разделе 1.3 мы приведём несколько способов кодирования алгоритма решения задачи для каждой из этих декомпозиций.

Ещё один способ декомпозиции задачи – деление списка пополам, как на рис. 1.6(d). Это приводит к двум подзадачам размером примерно в половину исходной. Декомпозиция даёт следующее рекурсивное определение:

$$s(\mathbf{a}) = \begin{cases} 0, & \text{если } n = 0, \\ \mathbf{a}[0], & \text{если } n = 1, \\ s(\mathbf{a}[0:n//2]) + s(\mathbf{a}[n//2:n]), & \text{если } n > 1. \end{cases} \tag{1.11}$$

В отличие от предыдущих определений такая декомпозиция требует начального условия для $n = 1$. Без него функция никогда бы не вернула конкретного значения для непустого списка. Обратите внимание, что такое определение не складывало бы и не возвращало ни одного элемента списка. Для непустого списка рекурсивное условие применялось бы многократно, и этот процесс никогда бы не закончился. Такая ситуация называется *бесконечной рекурсией*. Например, если бы список содержал только один элемент, то рекурсивное условие добавило бы значение 0, соответствующее пустому списку, к результату такой же исходной задачи. Иными словами, мы пытались бы бесконечно вычислить $s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) + \ldots$. Очевидная проблема такого сценария – в том, что при $n = 1$ исходная задача $s(\mathbf{a})$ уже не может быть разбита на меньшие и более простые.

1.3. Рекурсивный код

Для использования рекурсии при разработке алгоритмов крайне важно знать, как разбить задачу на меньшие себе подобные и определить рекурсивные методы, опираясь на индукцию (см. раздел 1.4). Как только это сделано, дальше уже не составит труда преобразовать определения в код, особенно для таких базовых типов данных, как целые и вещественные числа, символы или логические (булевы) значения. Рассмотрим функцию (1.5), которая суммирует первые *n* целых положительных (натуральных) чисел. В языке Python такая функция может быть закодирована, как показано в листинге 1.1. Аналогия между (1.5) и функцией Python очевидна. Как и во многих примерах данной книги, обычный условный оператор – единственная управляющая конструкция, необходимая для реализации рекурсивной функции. Кроме того, внутри его тела обязательно должно быть имя функции, означающее *рекурсивный вызов*. Таким образом, мы говорим, что функция *вызывает* саму себя, или *обращается* к самой себе, и потому рекурсивна (существуют рекурсивные функции, которые не вызывают себя непосредственно в теле условного оператора, о чём говорится в главе 9).

Листинг 1.1. Суммирование первых *n* натуральных чисел

```
1  def sum_first_naturals(n):
2      if n == 1:
3          return 1    # Base case
4      else:
5           return sum_first_naturals(n - 1) + n   # Recursive case
```

Код функции в других языках программирования так же прост. На рис. 1.7 показаны эквивалентные коды на нескольких языках программирования. И здесь сходство кода и определения функции вполне очевидно. Хотя весь код книги написан на языке Python, перевести его на другие языки программирования будет довольно просто.

Важно, что в функции (1.5) и в соответствующем ей коде не проверяется условие $n > 0$. Для входного параметра такой тип условия, задаваемого в постановке задачи или в определении функции, известен как *предусловие*. Программисты могут считать, что предусловия соблюдаются всегда и потому не должны создавать код для их поиска или обработки.

Листинг 1.2 соответствует рекурсивному определению (1.6). Рекурсивная функция использует каскадный условный оператор для выбора одного из двух начальных (строки 2–5) и рекурсивных условий (стро-

ки 6–11), где два последних дважды вызывают саму рекурсивную функцию.

C, Java:

```
int sum_first_naturals(int n)
{
    if (n == 1)
        return 1;
    else:
         return sum_first_naturals(n-1) + n;
}
```

Pascal:

```
function sum_first_naturals(n: integer): integer;
begin
    if n=1 then
        sum_first_naturals := 1
    else
         sum_first_naturals := sum_first_naturals(n-1) + n;
end;
```

MATLAB®:

```
function result = sum_first_naturals(n)
    if n==1
        result = 1;
    else
        result = sum_first_naturals(n-1) + n;
end

```

Scala:

```
def sum_first_naturals(n: Int): Int = {
    if (n==1)
        return 1
    else
        return sum_first_naturals(n-1) + n
}
```

Haskell:

```
sum_first_naturals 1 = 1
sum_first_naturals n = sum_first_naturals (n - 1) + n
```

Рис. 1.7. Функции вычисления суммы первых *n* натуральных чисел в разных языках программирования

Листинг 1.2. Другой вариант суммирования первых *n* натуральных чисел

```
def sum_first_naturals(n):
    if n == 1:
        return 1
    elif n == 2:
        return 3
    elif n % 2:
        return (3 * sum_first_naturals_2((n - 1) / 2)
                + sum_first_naturals_2((n + 1) / 2))
    else:
        return (3 * sum_first_naturals_2((n / 2)
                + sum_first_naturals_2((n / 2 - 1))
```

Так же просто закодировать функцию, вычисляющую *n*-е число Фибоначчи согласно стандартному определению (1.2). В листинге 1.3 приводится соответствующий код, где оба начальных условия проверяются в логическом выражении условного оператора.

Листинг 1.3. Вычисление *n*-го числа Фибоначчи

```
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Реализация функции Фибоначчи (1.7) требует большего количества действий. Если начальные условия идентичны, суммирование в рекурсивном условии нуждается в операторе цикла или иной функции для вычисления суммы значений $F(1), F(2), \ldots, F(n-2)$. В листинге 1.4 приводится возможное решение с использованием цикла **for**.

Листинг 1.4. Другой вариант вычисления *n*-го числа Фибоначчи

```
def fibonacci_alt(n):
    if n == 1 or n == 2:
         return 1
    else:
         aux = 1
         for i in range(1, n - 1):
             aux += fibonacci_alt(i)
         return aux
```

Результат сложения можно сохранить во вспомогательной переменной-сумматоре `aux` с начальным значением 1 (строка 5). Цикл **`for`** просто добавляет к вспомогательной переменной члены $F(i)$ для $i = 1, \ldots, n - 2$ (строки 6–7). В итоге функция возвращает вычисленное и сохранённое в `aux` число Фибоначчи.

Для более сложных типов данных различий в кодах разных языков программирования может становиться ещё больше из-за низкоуровневых деталей. Например, при работе с такой структурой данных, как список, рекурсивным алгоритмам для вычленения подсписков нужно знать его длину. На рис. 1.8 приведены три варианта списков (или подобных им структур данных) с параметрами, необходимыми для вычленения подсписков в рекурсивных программах.

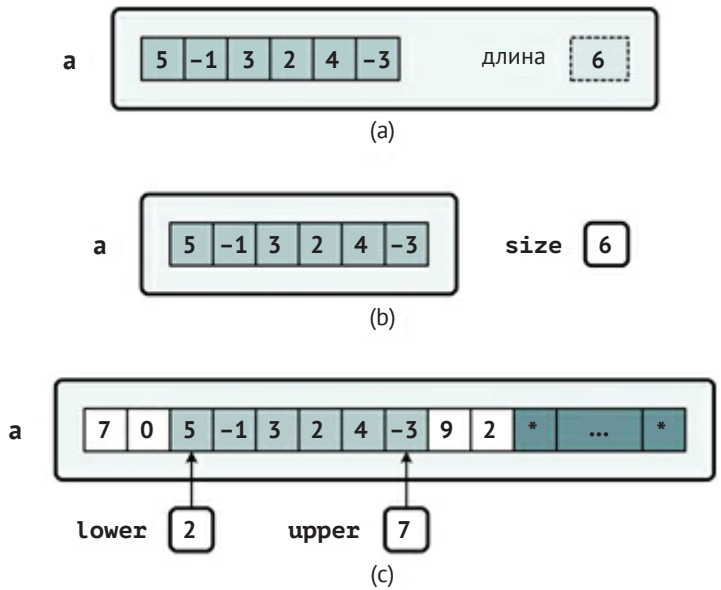


Рис. 1.8. Списочные структуры данных с параметрами, необходимыми для определения подсписков

В случае (а) длина списка **a** может быть получена без использования дополнительных переменных или параметров. Например, она может быть свойством или методом списка. В языке Python длину списка можно получить вызовом функции `len`. Однако в таких языках программирования, как C или Паскаль, получить длину стандартного массива невозможно. Если длину списка нельзя извлечь непосредственно из структуры данных, то для хранения и получения длины списка нужен

дополнительный параметр, например `size`, как в случае (b). Такой подход может применяться при работе с массивами – фиксированного размера или частично заполненными. Правда, в подобных случаях лучше использовать начальный и конечный индексы, задающие границы подсписка, как в случае (c). Отметим, что в этом случае фиксированный размер структуры данных может быть довольно большим (достаточным для нужд приложения), хотя истинная длина списков и подсписков может быть гораздо меньше. На рис. 1.8(c) изображён список, использующий только первые 10 элементов (остальные игнорируются). Внутри него определён подсписок из шести элементов с нижней (`lower`) и верхней (`upper`) индексными переменными, ограничивающими его пределы. Отметим, что элементы с этими индексами входят в подсписок.

Конструкции и синтаксис языка Python поддерживают высокий уровень алгоритмизации задач, избегая необходимости знать механизмы нижнего уровня, такие, например, как передача параметров. Однако его гибкость допускает большое разнообразие возможностей кодирования. В листинге 1.5 приводятся три решения задачи суммирования элементов списка, соответствующих трём способам декомпозиции на рис. 1.6, в которых единственным входным параметром является список, изображённый на рис. 1.8(a).

Функции `sum_list_length_1`, `sum_list_length_2` и `sum_list_length_3` реализуют рекурсивные определения (1.9), (1.10) и (1.11) соответственно. Заключительные строки листинга объявляют список **a** и печатают сумму его элементов, используя эти три функции. Отметим, что количество элементов n списка вычисляется методом `len`. В заключение напомним, что **a**[lower, upper] – это подсписок **a** от индекса `lower` до `upper` – 1, тогда как [lower:] равносильно [lower:len(**a**)]. Если размер списка нельзя получить непосредственно из списка, то его можно передать дополнительным параметром функции, как показано на рис. 1.8(b). Соответствующий код подобен листингу 1.5 и предлагается в качестве упражнения в конце главы.

В листинге 1.6 приводится другой вариант тех же функций с использованием больших списков и двух параметров (`lower` и `upper`), определяющих подсписки внутри этого списка, как показано на рис. 1.8(c).

Обратите внимание на аналогию этих функций с функциями из листинга 1.5. В данном случае список пуст, когда `lower` больше `upper`. Кроме того, подсписок содержит единственный элемент, если значения обоих индексов совпадают (напомним, что оба параметра указывают позиции элементов, входящих в подсписок).

Листинг 1.5. Рекурсивные функции суммирования элементов списка с единственным параметром – списком **a**

```
# Decomposition: s(a) => s(a[0:n-1]), a[n-1]
def sum_list_length_1(a):
    if len(a) == 0:
        return 0
    else:
        return (sum_list_length_1(a[0:len(a) - 1])
                + a[len(a) - 1])


# Decomposition: s(a) => a[0], s(a[1:n])
def sum_list_length_2(a):
    if len(a) == 0:
        return 0
    else:
        return a[0] + sum_list_length_2(a[1:len(a)])


# Decomposition: s(a) => s(a[0:n//2]), s(a[n//2:n])
def sum_list_length_3(a):
    if len(a) == 0:
        return 0
    elif len(a) == 1:
        return a[0]
    else:
        middle len(a) // 2
        return (sum_list_length_3(a[0:middle])
                + sum_list_length_3(a[middle:len(a)]))


# Some list:
a = [5, -1, 3, 2, 4, -3]

# Function calls:
print (sum_list_length_1(a))
print (sum_list_length_2(a))
print (sum_list_length_3(a))
```

Листинг 1.6. Другой вариант рекурсивных функций суммирования элементов подсписков списка **a**. Границы подсписков задаются двумя входными параметрами lower и upper – соответственно нижним и верхним индексами в списке **a**

```
# Decomposition: s(a) => s(a[0:n-1]), a[n-1]
def sum_list_limits_1(a, lower, upper):
    if lower > upper:
        return 0
    else:
        return a[upper] + sum_list_limits_1(a, lower, upper - 1)


# Decomposition: s(a) => a[0], s(a[1:n])
def sum_list_limits_2(a, lower, upper):
    if lower > upper:
        return 0
    else:
        return a[lower] + sum_list_limits_2(a, lower + 1, upper)


# Decomposition: s(a) => s(a[0:n//2]), s(a[n//2:n])
def sum_list_limits_3(a, lower, upper):
    if lower > upper:
        return 0
    elif lower == upper:
        return a[lower]    # or a[upper]
    else:
        middle len(upper + lower) // 2
        return (sum_list_limits_3(a, lower, upper)
                + sum_list_limits_3(a, middle + 1, upper))


# Some list:
a = [5, -1, 3, 2, 4, -3]

# Function calls:
print (sum_list_limits_1(a, 0, len(a) - 1))
print (sum_list_limits_2(a, 0, len(a) - 1))
print (sum_list_limits_3(a, 0, len(a) - 1))
```

1.4. Индукция

Индукция – ещё одно понятие, которое играет существенную роль при разработке рекурсивного кода. Термин имеет разные значения в зависимости от области и предмета его применения. В контексте рекурсивного программирования он связан с математическим доказательством методом индукции. Его ключевая идея состоит в том, что программисты должны предположить, что рекурсивный код, который они пытаются реализовать, правильно работает для простых и малых задач, даже если они ещё не написали ни одной строчки программы! Это понятие также называют иногда рекурсивной «убеждённостью» (leap of faith). В данном разделе рассматриваются эти чрезвычайно важные понятия.

1.4.1. Математические доказательства методом индукции

В математике доказательства методом индукции – это важный инструмент обоснования истинности некоторого утверждения. Простейшие доказательства касаются формул, зависящих от некоторого положительного (или неотрицательного) целого числа n. В таких случаях доказательства проверяют истинность формул для любого допустимого значения n. Метод индукции предполагает два шага:

a) начальное условие (основание). Проверяется, что формула верна для наименьшего значения n, скажем, $n = 0$;

b) шаг индукции. Сначала предполагается, что формула верна для некоторого произвольного значения n. Это предположение называется *гипотезой индукции*. Затем на основании этого предположения доказывается, что формула верна и для $n + 1$.

Если можно доказать истинность обоих шагов, то согласно индукции из этого следует, что утверждение справедливо для всех $n \geqslant n_0$. Утверждение должно быть истинным для n_0, а затем – для $n_0 + 1$, $n_0 + 2$ и так далее для каждого следующего шага индукции.

Рассмотрим ещё раз сумму первых n положительных чисел (1.4). Попробуем доказать справедливость следующего равенства (гипотеза индукции), содержащего квадратный полином:

$$S(n) = \sum_{i=1}^{n} i = n\,(n + 1)/2. \qquad (1.12)$$

Начальное условие, очевидно, истинно, так как $S(1) = 1(1 + 1)/2 = 1$. А на шаге индукции мы должны показать, имеет ли силу

$$S(n+1) = \sum_{i=1}^{n+1} i = (n+1)\,(n+2)/2 \tag{1.13}$$

при условии, что (1.12) верна. Для начала запишем $S(n + 1)$ как

$$S(n+1) = \sum_{i=1}^{n} i + (n+1).$$

Затем, считая, что имеет силу (1.12), заменим сумму полиномом:

$$S(n+1) = n(n+1)/2 + (n+1) = (n+1)(n/2+1) = (n+1)(n+2)/2.$$

Откуда видно, что (1.13) истинна, и на этом доказательство закончено.

1.4.2. Рекурсивная убеждённость

Для решения меньших подзадач рекурсивные функции обычно вызывают сами себя. И тут у начинающего программиста возникают вполне резонные сомнения и вопросы: действительно ли написанная таким образом рекурсивная функция будет работать и допустимо ли в теле функции вызывать её саму, если она ещё не завершилась? И даже несмотря на то, что в языках программирования с поддержкой рекурсии функции могут вызывать сами себя, крайне важно поверить, что они работают правильно для подзадач меньшего размера. Эту уверенность, играющую ту же роль, что и гипотеза в доказательствах методом индукции, называют рекурсивной «убеждённостью». Она – один из краеугольных камней рекурсивного мышления и в то же время «крепкий орешек», о который программисты-новички рискуют сломать себе зубы. И конечно же, это не догма, принимаемая безраздумно. Следующий мысленный эксперимент поясняет её роль в рекурсивном мышлении.

Представьте, что вы находитесь в огромной классной комнате, и учитель просит вас найти сумму целых чисел от 1 до 100. Если вы не знаете формулы (1.12), вам пришлось бы выполнить 99 сложений. Но предположим, что вам разрешено переговариваться с одноклассниками, и среди них есть один «умный», который убеждён, что может сложить первые n положительных целых чисел, когда $n \leqslant 99$. В этом случае ваша задача стала бы гораздо проще: вы могли бы просто попросить его вычислить сумму первых 99 положительных целых чисел (она равна 4950), а вам бы осталось только добавить к ней 100, чтобы получить 5050. Несмотря на то что вы якобы пытаетесь схитрить или воспользоваться некой

уловкой, такая стратегия, приведённая на рис. 1.9(a), – правильный способ (рекурсивного) мышления и решения задачи. При этом вы должны быть уверены лишь в том, что умный одноклассник действительно даст вам правильный ответ на ваш вопрос. Эта уверенность как раз и соответствует рекурсивной убеждённости.

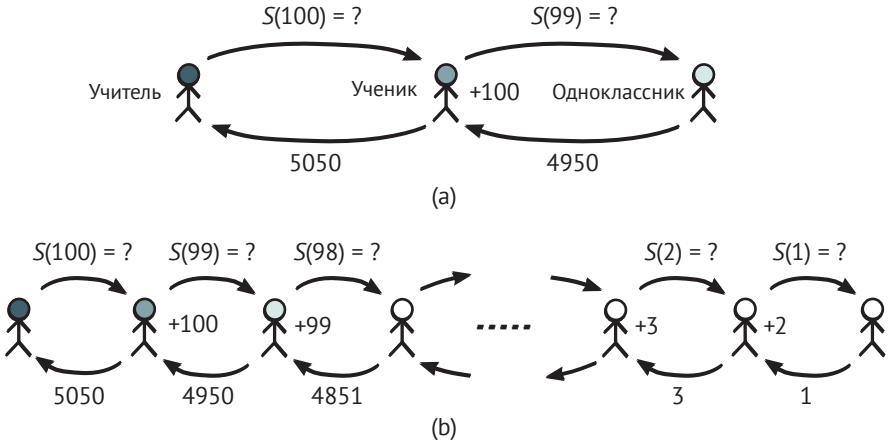


Рис. 1.9. Задача на устный счёт в классе. Учитель просит учеников сложить первые 100 положительных целых чисел.
$S(n)$ – сумма первых n положительных целых чисел

Для ответа на ваш вопрос у вашего одноклассника есть два пути. Если он действительно очень умён, то способен ответить на ваш вопрос без посторонней помощи, например применив формулу (1.12). Но это маловероятно ввиду сложности задачи. Поэтому, осознавая, что он не столь умён, как ему кажется, для получения ответа на заданный ему вопрос он, скорее всего, выберет другой путь и задаст аналогичный вопрос ещё одному «умному» однокласснику, то есть воспользуется вашей же стратегией. Иными словами, вы можете считать, что ваш друг – это просто ваш клон. Фактически при таком сценарии, представленном на рис. 1.9(b), все ученики в классе применили бы ваш подход. Отметим, что этот процесс явно рекурсивен: каждый ученик просит другого, чтобы тот решил более простую задачу (размер которой на каждом шаге уменьшается на 1), пока самый последний в этой цепочке ученик не даст ответ «1» для тривиального начального условия. Таким образом, каждый ученик может дать ответ всякому, кто попросит его вычислить полную сумму целых чисел, выполнив только одно-единственное сложение. Этот процесс продолжается до тех пор, пока вы не получите сумму первых 99 положительных целых чисел. После чего вы сможете

правильно ответить на вопрос учителя, добавив к этой сумме 100. Стоит отметить, что процесс не заканчивается по достижении начального условия (это одна из главных ошибок в понимании студентами начального условия). Кроме первого этапа, когда ученики последовательно задают вопросы другим одноклассникам, пока один из них не даст тривиальный ответ «1», есть и второй – когда они вычисляют сумму и передают результат однокласснику.

Этот подход можно формализовать, как в (1.5), и закодировать, как в листинге 1.1, где $S(n-1)$ или `sum_first_naturals(n - 1)` равносильны вашему обращению к однокласснику с просьбой решить подзадачу. Рекурсивная убеждённость заключается в вашей уверенности в том, что $S(n-1)$ в определении или `sum_first_naturals(n - 1)` в коде действительно дают правильный ответ.

Аналогично, при доказательстве методом индукции мы можем предположить, что процедура в целом должна быть правильной. Во-первых, ваш ответ будет верным, пока ваш одноклассник даёт вам правильный ответ. Но то же самое относится и к ученику, которого ваш одноклассник просит решить подзадачу. Это рассуждение может многократно применяться к каждому ученику до тех пор, пока, наконец, некий ученик не попросит другого сложить целые числа от 1 до 1. Так как последний ученик возвращает 1, что, конечно же, верно, все предыдущие ответы студентов должны быть по той же причине верными.

Всё это подразумевает, что программисты могут – и *должны* – выстраивать правильные рекурсивные условия, предполагая, что рекурсивные вызовы функции, включающие подобные им подзадачи меньшего размера, работают правильно. Неформально мы должны считать, что можем получить решения подзадач «бесплатно». Таким образом, если нам нужно решать различные задачи, то согласно декомпозиции на рис. 1.4 нам нет нужды решать подобные исходной подзадачи, так как можно считать, что эти решения уже найдены. Конечно, нам нужно будет так или иначе переработать их (изменить, расширить, объединить и т. д.), чтобы создать рекурсивные условия. Наконец, рекурсивные алгоритмы завершаются благодаря начальным условиям, которые не только истинны, но и позволяют алгоритмам завершаться.

1.4.3. Императивное и декларативное программирование

Парадигма программирования – это общая стратегия разработки программного обеспечения. Парадигма императивного программирования фокусируется на том, *как* работает программа, когда код явно

описывает поток управления и команды, которые изменяют значения переменных (то есть состояние программы). Из этой парадигмы следует итерационный код. Напротив, рекурсивный код опирается на декларативную парадигму программирования, которая фокусируется на том, *что* должны делать программы без явного описания потока управления, и может считаться противоположностью императивного программирования. Функциональное программирование следует декларативной парадигме, которая исключает побочные эффекты, а все действия сводятся к вычислению математических функций.

Таким образом, при разработке рекурсивного кода программисты нацелены думать о том, *что* должны делать функции, а не *как* они должны это делать. Отметим, что это имеет прямое отношение к функциональной абстракции и использованию программных библиотек, когда программисты, не беспокоясь о деталях реализации, могут сосредоточиться исключительно на функциональных возможностях, считая методы правильно работающими «черными ящиками». По этой причине с позиций высокоуровневого программирования программисты могут – и *должны* – рассматривать вызовы рекурсивных функций как обращения к «черным ящикам», дающим правильные результаты (как на рис. 1.9(a), а не думать обо всех рекурсивных шагах вплоть до начального условия (как на рис. 1.9(b)). Зачастую фиксация внимания на деталях низкого уровня, касающихся цепочки вызовов функций, только запутывает дело.

1.5. Рекурсия против итерации

Вычислительная мощь компьютеров заключается главным образом в их способности выполнять задачи многократно – итерационно или рекурсивно. Первые для многократного повторения действий используют такие конструкции, как циклы **while** или **for**, тогда как рекурсивные функции раз за разом вызывают самих себя, при каждом вызове вплоть до достижения начального условия решая одну и ту же задачу. Итерация и рекурсия эквивалентны в том смысле, что они решают одни и те же виды задач. Всякая итеративная программа может быть преобразована в рекурсивную, и наоборот. Выбор может зависеть от нескольких факторов, таких как вычислительная задача, эффективность, язык или парадигма программирования. Например, итерация предпочтительна в императивных языках, тогда как рекурсия широко используется в функциональном программировании, которое следует декларативной парадигме программирования.

Уже приведённые в книге примеры могут быть легко закодированы с помощью циклов. Поэтому выгода от применения рекурсии ещё не вполне ясна. На практике основное преимущество использования рекурсивных алгоритмов вместо итерационных заключается в том, что для многих вычислительных задач рекурсивный алгоритм гораздо проще проектировать и понимать. Рекурсивный алгоритм больше похож на логический подход к решению задач. Следовательно, он должен быть более понятным, изящным и коротким.

Кроме того, рекурсивные алгоритмы неявно используют для хранения информации программный стек, операции с которым (скажем, втолкнуть и вытолкнуть) прозрачны для программиста. Поэтому они предоставляют ясные альтернативы итерационным алгоритмам, где ответственность за явное управление стековой (или подобной ей) структурой данных ложится на программиста. Например, если условие задачи или структура данных предполагает дерево, рекурсивные алгоритмы будут гораздо проще и понятнее итерационных, поскольку для последних нужно реализовывать поиски в ширину или в глубину с использованием очередей и стеков соответственно.

С другой стороны, в самом общем случае рекурсивные алгоритмы не столь эффективны, как итерационные, и используют больше памяти. Эти недостатки связаны с использованием программного стека. Вообще, каждый вызов функции, рекурсивный он или нет, выделяет память в программном стеке и хранит в нём информацию, требующую дополнительных вычислительных затрат. Таким образом, рекурсивная программа может быть не только медленнее итерационной – большое количество вызовов при выполнении программы может привести к переполнению стека. Более того, некоторые рекурсивные определения сами по себе могут быть довольно медленными. Например, коды для чисел Фибоначчи в листингах 1.3 и 1.4 требуют экспоненциального времени выполнения, тогда как числа Фибоначчи могут быть вычислены (гораздо быстрее) за логарифмическое время. Помимо этого, рекурсивные алгоритмы сложнее для отладки (то есть для пошагового анализа программы с целью поиска и локализации ошибок), особенно если функции вызывают себя многократно, как в листингах 1.3 и 1.4.

Наконец, если в некоторых функциональных языках программирования циклы вообще не допускаются, то многие иные языки поддерживают как итерацию, так и рекурсию. Поэтому в них можно объединять оба этих стиля программирования так, чтобы создавать алгоритмы не только мощные, но и ясные для понимания (например, перебор с воз-

вратами). В листинге 1.4 приведён простой пример рекурсивной функции с применением цикла.

1.6. Типы рекурсии

Рекурсивные алгоритмы можно классифицировать по нескольким критериям. В этом последнем разделе кратко описываются типы рекурсивных функций и процедур, которые будут рассматриваться на протяжении всей книги. Каждый тип рекурсии мы проиллюстрируем на примере рекурсивных функций для вычисления чисел Фибоначчи ($F(n)$) с определёнными для каждого типа рекурсии параметрами. В заключение отметим, что рекурсивные алгоритмы могут относиться к нескольким типам рекурсии.

1.6.1. Линейная рекурсия

Линейный тип рекурсии появляется, когда методы вызывают себя только однажды. Есть два типа линейно-рекурсивных методов, однако мы будем применять термин «линейная рекурсия», когда имеем дело с методами, которые так или иначе обрабатывают результат рекурсивного вызова до выдачи или возврата своего собственного результата. Например, функция вычисления факториала в (1.3) относится к этой категории, так как выполняет только один рекурсивный вызов, а результат подзадачи умножается на n, чтобы получить результат функции. Функции в (1.5), (1.9) и (1.10) тоже являются очевидными примерами линейной рекурсии. Следующая функция представляет собой другой пример функции Фибоначчи:

$$f(n) = \begin{cases} 1, & \text{если } n = 1 \text{ или } n = 2, \\ \lfloor \Phi \cdot f(n-1) + 1/2 \rfloor, & \text{если } n > 2, \end{cases} \tag{1.14}$$

где $\Phi = (1 + \sqrt{5})/2 \approx 1.618$ – константа, известная под названием «золотое сечение», а $\lfloor \cdot \rfloor$ обозначает нижнюю границу. В этом случае $F(n) = f(n)$ – n-е число Фибоначчи. Такой тип линейно-рекурсивных алгоритмов подробно рассматривается в главе 4.

1.6.2. Хвостовая рекурсия

Второй тип линейной рекурсии называют «хвостовой рекурсией». Методы, относящиеся к этой категории, также вызывают себя лишь однажды, но рекурсивный вызов – это последнее действие в рекурсивном условии. Следовательно, оно никак не влияет на результат рекурсивно-

го вызова. Для примера рассмотрим следующую хвостовую рекурсивную функцию:

$$f(n, a, b) = \begin{cases} b, & \text{если } n = 1, \\ f(n-1, a+b, a), & \text{если } n > 1. \end{cases} \tag{1.15}$$

Обратите внимание, что в рекурсивном условии метод просто возвращает результат рекурсивного вызова, который не входит в математическое или логическое выражение. Поэтому рекурсивное условие определяет лишь отношение между наборами параметров, для которых функция возвращает одно и то же значение. Выполняя рекурсивные вызовы, алгоритм лишь изменяет параметры, пока нет возможности получить решение из начального условия. В этом случае числа Фибоначчи $F(n)$ могут быть получены из $f(n, 1, 1)$. Этому типу рекурсивных алгоритмов будет посвящена глава 5.

1.6.3. Множественная рекурсия

Рекурсия множественного типа возникает, когда в каком-то рекурсивном условии метод вызывает себя несколько раз (см. главу 7). Если метод вызывает себя дважды, то такой тип рекурсии называют «двойной рекурсией» («binary recursion»). Из приведённых ранее примеров к данному типу рекурсии относятся (1.2), (1.6), (1.7) и (1.11). Следующая функция использует множественную рекурсию для альтернативного определения чисел Фибоначчи, где $F(n) = f(n)$:

$$f(n) = \begin{cases} 1, & \text{если } n = 1 \text{ или } n = 2, \\ f(n/2+1)^2 - f(n/2-1)^2, & \text{если } n > 2 \text{ и } n - \text{чётное}, \\ f((n+1)/2)^2 + f((n-1)/2)^2, & \text{если } n > 2 \text{ и } n - \text{нечётное}. \end{cases} \tag{1.16}$$

Такой тип рекурсии обычно возникает в алгоритмах, разработанных согласно стратегии «разделяй и властвуй», которой посвящена глава 6.

1.6.4. Взаимная рекурсия

Говорят, что несколько методов относится к типу взаимно-рекурсивных, если они могут вызывать друг друга циклически. Например, функция *f* может вызывать другую функцию *g*, которая, в свою очередь, может вызвать третью функцию *h*, которая может завершаться вызовом первой функции *f*. Такую рекурсию называют ещё «косвенной», поскольку метод вызывает себя не непосредственно внутри своего тела, а через

циклическую цепочку вызовов. Например, рассмотрим две следующие функции Фибоначчи:

$$A(n) = \begin{cases} 0, & \text{если } n = 1, \\ A(n-1) + B(n-1), & \text{если } n > 1, \end{cases} \tag{1.17}$$

$$B(n) = \begin{cases} 1, & \text{если } n = 1, \\ A(n-1), & \text{если } n > 1. \end{cases} \tag{1.18}$$

Ясно, что A рекурсивна, так как вызывает саму себя, но рекурсия в B – косвенная, так как вызывает A, которая, в свою очередь, вызывает B. Таким образом, рекурсия возникает из-за того, что вызов B может породить другие вызовы B (в более простых экземплярах задачи). Эти функции также могут применяться для генерации чисел Фибоначчи. В частности, $F(n) = B(n) + A(n)$. Взаимной рекурсии будет посвящена глава 9.

1.6.5. Вложенная рекурсия

Вложенный тип рекурсии возникает, когда аргумент рекурсивной функции содержит другой рекурсивный вызов. Рассмотрим следующую функцию Фибоначчи:

$$f(n,s) = \begin{cases} 1+s, & \text{если } n = 1 \text{ или } n = 2, \\ f(n-1, s + f(n-2, 0)), & \text{если } n > 2. \end{cases} \tag{1.19}$$

Второй параметр рекурсивного вызова – выражение, включающее в себя еще один рекурсивный вызов функции. В этом случае числа Фибоначчи $F(n)$ могут быть получены из $f(n, 0)$. Такой тип рекурсии редок, но появляется в некоторых задачах и контекстах, связанных с хвостовой рекурсией. Краткий обзор вложенной рекурсии даётся в главе 11.

1.7. Упражнения

Упражнение 1.1. Что вычисляет следующая функция?

$$f(n) = \begin{cases} 1, & \text{если } n = 0, \\ f(n-1) \cdot n, & \text{если } n > 0. \end{cases}$$

Упражнение 1.2. Рассмотрим последовательность, заданную следующим рекурсивным определением: $s_n = s_{n-1} + 3$. Вычислите первые её четыре члена, полагая, что (a) $s_0 = 0$ и (b) $s_0 = 4$. Приведите нерекурсивное определение s_n для обоих случаев.

Упражнение 1.3. Рассмотрим последовательность, заданную следующим рекурсивным определением: $s_n = s_{n-1} + s_{n-2}$. Если заданы начальные значения s_1 и s_2, то этой информации достаточно, чтобы определить всю последовательность. Покажите, что всю последовательность можно определить также, если заданы два произвольных её члена s_i и s_j, где $i < j$. Наконец, найдите элементы последовательности между $s_1 = 1$ и $s_5 = 17$.

Упражнение 1.4. Множество «потомков человека» можно определить рекурсивно, как детей этого человека вместе с их потомками. Дайте математическое описание данного понятия в обозначениях теории множеств. В частности, определите функцию $D(p)$, где D обозначает потомков, а параметр p ссылается на отдельного человека. Кроме того, считайте, что можно пользоваться функцией $C(p)$, возвращающей множество детей человека p.

Упражнение 1.5. Пусть $F(n) = F(n - 1) + F(n - 2)$ представляет собой рекурсивную функцию, где n – положительное целое число, а $F(1)$ и $F(2)$ имеют произвольные начальные значения. Покажите, что для $n \geqslant 2$ её можно записать как $F(n) = F(2) + \sum_{i=1}^{n-2} F(i)$.

Упражнение 1.6. Реализуйте функцию вычисления факториала (1.3).

Упражнение 1.7. Как абстрактный тип данных список может быть либо пустым списком, либо списком из двух частей – элемента, называемого «головой», и списка (возможно, пустого), называемого «хвостом». Пусть **a** – список в языке Python. Существует несколько способов проверить, пуст ли он. Например, условие **a** == [] возвращает True, если список пуст, и False в противном случае. Кроме того, голова списка – это просто **a**[0], тогда как хвост определяется как **a**[1:]. Напишите альтернативную версию функции `sum_list_length_2` из листинга 1.5, которая, используя предыдущие элементы, избегает применения функции `len`.

Упражнение 1.8. Реализуйте три функции, вычисляющие сумму элементов списка чисел для каждой из трёх декомпозиций на рис. 1.6. Функции должны получать два входных параметра: список и его размер (длину) согласно сценарию на рис. 1.12(b). Кроме того, реализуйте вызов этих функций и вывод их результатов.

Упражнение 1.9. Докажите методом математической индукции следующее тождество для геометрических прогрессий, где n – положительное целое число, а $x \neq 1$ – некоторое вещественное число:

$$\sum_{i=0}^{n-1} x^i = (x^n - 1)/(x - 1).$$

Упражнение 1.10. Реализуйте пять рекурсивных функций из раздела 1.6 для вычисления чисел Фибоначчи $F(n)$. Поскольку некоторые из них, помимо n, требуют ещё нескольких параметров или не вычисляют само число Фибоначчи, реализуйте дополнительные функции-оболочки с параметром n, возвращающие сами числа Фибоначчи. Убедитесь, что они дают правильные результаты для $n = 1, \ldots, 10$.

Глава 2

Методика рекурсивного мышления

Наука гораздо больше совокупности знаний. Это – способ мышления.

– Карл Сэгэн

Декларативный подход – это возможность проектировать рекурсивные алгоритмы решения широкого спектра вычислительных задач, следуя систематическому подходу. Эта глава описывает обобщённый шаблон вывода рекурсивных решений из декларативной перспективы, которая раскладывает процесс рекурсивного мышления в последовательность шагов. Кроме того, глава предлагает несколько полезных схем, которые программисты могут использовать при проектировании рекурсивных условий. Они полезны, так как заставляют думать о декомпозиции задачи и об индукции – двух важнейших понятиях, лежащих в основе рекурсивного (декларативного) мышления.

Конечно, в некоторых случаях (например, при отладке) или для довольно сложных задач было бы полезно думать о деталях нижнего уровня, имея в виду последовательность действий, выполняемых рекурсивными процедурами. Но, несмотря на это, мы отложим пока рассмотрение этих низкоуровневых аспектов и сосредоточимся в этой главе на высокоуровневом декларативном мышлении.

2.1. Шаблон проектирования рекурсивных алгоритмов

Введённые в главе 1 рекурсивные функции, как и большинство алгоритмов из последующих глав книги, могут быть спроектированы по

шаблону методики, изображённой на рис. 2.1. В этом шаблоне упор на декларативное мышление делается на шаге 4, поскольку метод индукции заставляет сосредоточиться на том, *что* делает алгоритм, а не *как* он это делает. Последующие разделы объясняют каждый шаг шаблона и предостерегают от распространённых ловушек и недоразумений.

1. Определить **размер** задачи.
2. Определить **начальные условия**.
3. **Разложить** вычислительную задачу на подобные ей подзадачи меньшего размера и, возможно, на другие дополнительные задачи.
4. Определить **рекурсивные условия**, опираясь на **индукцию** и **схемы**.
5. **Протестировать** код.

Рис. 2.1. Общий шаблон проектирования рекурсивных алгоритмов

2.2. Размер задачи

Естественно, первый шаг решения любой вычислительной задачи – понять её. Для его продолжения должны быть чётко определены заданные условия задачи, её входы (исходные данные), выходы (результаты) и отношения между ними. Независимо от типа разрабатываемого алгоритма крайне важно определить размер задачи (или её входной размер). Его следует понимать как математическое выражение, включающее в себя величины, связанные с входными параметрами, которые определяют сложность задачи с точки зрения числа операций, необходимых алгоритму для её решения. При применении рекурсии это особенно важно, так как начальные и рекурсивные условия явно зависят от него.

Во многих задачах размер зависит лишь от одного фактора, коим может оказаться один из входных параметров. Например, размер задач, вычисляющих n-е число Фибоначчи, факториал n или сумму первых n положительных целых чисел (см. главу 1), – это и есть входной параметр n. Очевидно, что для таких задач решение меньших её экземпляров (например, для $n - 1$ или $n/2$) требует меньшего числа операций. В других случаях размер задачи может не задаваться явно входами, однако синтаксис и конструкции языка программирования позволяют рассчитать её размер. Например, при работе со списками размер задачи часто определяется их длиной, которая может быть получена вызовом функции len.

В других задачах размер может быть выражен как функция входных параметров. Рассмотрим, например, задачу вычисления суммы цифр некоторого положительного целого числа n. Хотя входным параметром является n, размер задачи вовсе не n, а количество цифр n, которое определяет количество выполняемых операций. Формально эта величина может быть выражена как $\lfloor \lg n \rfloor + 1$.

Размер задачи может также зависеть от нескольких входных параметров. Рассмотрим задачу добавления в список элементов подсписка (см. листинг 1.6), заданного индексами «lower» и «upper». Решение задачи требует добавления $n - m + 1$ элементов, где n и m – верхний и нижний индексы соответственно. Таким образом, необходимо произвести $n - m$ добавлений. Небольшая (на единицу) разница между обоими выражениями не имеет значения. Следует отметить, что для решения задачи не нужно знать, сколько именно операций будет выполнено. Напротив, достаточно знать, когда будет достигнуто начальное условие и каким образом сокращать размер задачи, чтобы разложить её на меньшие подзадачи. Для этой последней задачи размер сокращается за счёт уменьшения разности между n и m.

Кроме того, размер задачи не всегда определяет количество выполняемых алгоритмом операций. Рассмотрим задачу сложения элементов квадратной матрицы размером $n \times n$. Поскольку она содержит n^2 элементов, для суммирования потребуется $n^2 - 1$ сложений, что является функцией n. Однако в этом случае размер задачи – просто n, а не n^2, так как меньшие подзадачи возникают за счёт уменьшения n, а не n^2. Если бы матрица была размером $n \times m$, то размер задачи зависел бы от обоих параметров n и m. В частности, размер задачи мог быть nm, и тогда подзадачи меньшего размера можно было получать уменьшением n, m или обоих сразу.

Вообще говоря, размер – это свойство задачи, а не конкретного алгоритма, который её решает. Поэтому это не фактическое количество вычислений, выполненных определённым алгоритмом, так как задачи могут быть решены различными алгоритмами, время выполнения которых может варьироваться в широких пределах. Рассмотрим задачу поиска некоторого числа в сортированном списке из n чисел. В худшем случае (когда такого числа в списке нет и результатом будет False) она может быть решена бездумно за n шагов или более эффективно за $\lfloor \log_2(n) \rfloor + 1$ шагов с использованием «двоичного поиска». Но независимо от используемого алгоритма функция, описывающая время его выполнения, зависит только от n. Таким образом, размер задачи – n.

Однако для некоторых задач их размер может быть определён несколькими способами. Более того, выбор размера в процессе рекурсивной разработки влияет на остальную часть решения и в итоге приводит к различным алгоритмам решения задачи. Рассмотрим, например, задачу сложения двух неотрицательных целых чисел a и b с использованием только операций увеличения и уменьшения на единицу (см. раздел 4.1.2). Размер задачи может быть $a + b$, a, b или $\min(a, b)$, при этом конечные алгоритмы будут разными в зависимости от выбранного размера задачи.

2.3. Начальные условия

Начальные условия – это такие экземпляры задачи, для решения которых не нужна рекурсия, точнее не нужны рекурсивные вызовы. В самом общем случае начальное условие – это наименьший экземпляр задачи, результат которой определяется тривиально, а иногда даже не требует вычислений. Например, значение первого и второго чисел Фибоначчи – это просто 1, что прямо следует из их определения. Очевидно, что «сумма» первого положительного целого числа есть 1. Аналогично, «сумма» цифр неотрицательного целого числа n из одной цифры (когда $n < 10$), очевидно, равна самой этой цифре (n). Заметьте, что эти функции в начальных условиях просто возвращают значение, не выполняя ни сложения, ни других математических действий.

Некоторые методы могут требовать нескольких начальных условий. Например, в формуле (1.2) для чисел Фибоначчи задано два начальных условия – одно для $n = 1$ и другое для $n = 2$. Так как оба начальных условия имеют одно и то же значение (1), функция в листинге 1.3 может использовать одно логическое выражение ((n == 1) or (n == 2)), объединяющее оба условия. Другие методы должны использовать каскадный оператор **if**, чтобы проверить каждое начальное условие. Например, функция (1.6) определяет одно начальное условие для $n = 1$ и другое для $n = 2$, так как они возвращают разные значения для каждого из входов. По этой причине соответствующий код в листинге 1.2 тоже разделяет эти два условия.

Одно из достоинств рекурсивного программирования – лаконичность кода, что способствует его пониманию. В связи с этим рассмотрим функцию вычисления факториала (1.3). Если начальные условия представляют собой тривиальные (наименьшие) экземпляры задачи, можно было бы потребовать, чтобы определение функции включало

также случай 1! = 1. Но значение 1! можно получить и из рекурсивного условия, так как 1! = 0! × 1 = 1 × 1 = 1. Следовательно, хотя 1! = 1 – очевидно, верное начальное условие, его включение в формулу было бы избыточным и потому необязательно. Кроме того, такие необязательные условия могут вводить в заблуждение, поскольку программисты, пытающиеся понять рекурсивный код, могут посчитать их необходимыми для получения правильного результата функции. Таким образом, ради краткости и ясности обычно рекомендуется включать наименьшее количество начальных условий, приводящих к правильным рекурсивным определениям. Наконец, избыточные начальные условия обычно не оказывают существенного влияния на эффективность рекурсивной программы. В общем, если они и могут сократить количество вызовов рекурсивной функции для некоторых входных значений, то сэкономленное время вычисления зачастую незначительно. Например, если 1! включён в начальные условия в функцию вычисления факториала, то рекурсивный процесс выполнит лишь на один рекурсивный вызов меньше, что едва ли повлияет на время его выполнения. Кроме того, коду потребовались бы две проверки условия, и он мог бы стать даже менее эффективным, чем тот, где оба условия вычислялись бы до перехода к рекурсивному условию.

На примере функции вычисления факториала листинг 2.1 иллюстрирует эту и другие основные ловушки, касающиеся начальных условий. Во-первых, метод вычисления факториала обеспечивает безупречную реализацию математической функции. Метод `factorial_redundant` правилен, так как даёт правильный результат для любого неотрицательного целочисленного входного параметра. Однако он содержит дополнительное начальное условие – совершенно ненужное.

Кроме того, мы должны также добиваться общности – крайне желательного свойства программ. Общие функции – это такие функции, которые работают правильно в широком диапазоне входных параметров и потому более применимы и полезны. Функция вычисления факториала (1.3) имеет своим входным параметром неотрицательное целое число и определяется так, чтобы быть применимой не только к каждому положительному целому числу, но также и к нулю. Замена начального условия 0! = 1 на 1! = 1 привела бы к менее общей функции, так как она не была бы определена для $n = 0$. Метод `factorial_missing_base_case` реализует такую функцию. Если бы она была вызвана с входным параметром $n = 0$, она не дала бы правильного результата, погрузившись в бездну бесконечной рекурсии. А именно `factorial_missing_base_`

case(0) вызвала бы factorial_missing_base_case(-1), которая, в свою очередь, вызовет factorial_missing_base_case(-2) и т. д. На практике такой процесс (после выполнения большого количества вызовов функции) обычно останавливается с сообщением об ошибке времени выполнения программы (вроде «Переполнение стека» или «Превышена предельная глубина рекурсии»). Наконец, метод factorial_no_base_case всегда приводит к бесконечной рекурсии, так как не имеет начального условия и потому никогда не остановится.

Листинг 2.1. Неправильные представления начальных условий для функции вычисления факториала

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


def factorial_redundant(n):
    if n == 0 or n == 1:
        return 1
    else:
        return factorial_redundant(n - 1) * n


def factorial_missing_base_case(n):
    if n == 1:
        return 1
    else:
        return factorial_missing_base_case(n - 1) * n


def factorial_no_base_case(n):
    return factorial_no_base_case(n - 1) * n
```

Для многих задач существуют такие экземпляры, результат которых можно получить без использования рекурсии, даже если их размер достаточно велик. Рассмотрим, например, задачу поиска некоторого элемента в списке. Размер задачи определяется множеством элементов списка. Первое начальное условие отвечает пустому списку (наименьшему экземпляру задачи нулевого размера), когда результат – очевидно, False. Кроме того, алгоритм может определять положение (скажем,

в начале, в середине или в конце) найденного элемента списка и немедленно выдавать значение True, даже если список огромен (глава 5 посвящена этому виду поисковых задач). В обоих случаях нет никакой необходимости в дальнейшем выполнении рекурсивных вызовов.

2.4. Декомпозиция задачи

Следующий шаг рекурсивной методики состоит в выявлении подзадач, подобных, но меньших исходной (и, возможно, других разнообразных задач), как показано на рис. 1.4. Эти меньшие подзадачи будут использоваться на шаге 4 шаблона проектирования для определения рекурсивных условий. Процесс заключается в уменьшении размера задачи с целью определить её меньшие экземпляры и тем самым приблизить её к начальным условиям. Поэтому перед продолжением этого шага важно правильно установить размер задачи и ее начальные условия.

Многие рекурсивные решения уменьшают размер задачи на 1 или делят его надвое, как показано на рис. 1.5 и 1.6. Представление задач в виде схем полезно и очень рекомендуется, так как они помогают наглядно распознать подобные исходной подзадачи, то есть рекурсию, и могут также облегчить определение рекурсивных условий. На рис. 2.2 приведены дополнительные схемы, иллюстрирующие декомпозицию задачи суммирования первых положительных целых чисел, когда размер задачи уменьшается на 1.

Во-первых, поскольку рекурсия – это математическая функция, которая может быть выражена формулой, мы можем просто развернуть выражение так, чтобы идентифицировать исходную задачу и соответствующую подзадачу, как показано на рис. 2.2(a). На рис. 2.2(b) схема также содержит формулу, но использует вместо фигурных скобок поля, заключающие формулы. В этом случае более темный внешний блок представляет полную задачу, тогда как вложенный в него более светлый блок обозначает подзадачу. Схема на рис. 2.2(c) подобна схеме на рис. 2.2(b), но задача состоит в добавлении кружочков, выстроенных треугольником. Наконец, на рис. 2.2(d), чтобы показать задачу и подзадачу, суммируемые числа размещены в сопоставимых друг с другом прямоугольниках.

Эти довольно простые схемы целесообразны, поскольку позволяют обнаружить подзадачу внутри исходной задачи. При этом крайне важно подчеркнуть, что подзадача уже не делится на более мелкие. Например, сумма n – 2 положительных целых чисел не появляется явно как подзадача ни в одной из схем. Иными словами, схемы не должны показывать вложенность или древовидность подзадач. В этом – одна из основных

ловушек рекурсивного подхода, а также источник недоразумений. На рис. 2.3(a) и 2.3(c) показаны правильные декомпозиции задачи суммирования первых n положительных целых чисел ($S(n)$). Напротив, хотя вложенные подзадачи на рис. 2.3(b) или (унарное) дерево рекурсивных вызовов на рис. 2.3(d) правильно отражают задачу, они содержат дополнительные подзадачи ($S(1), S(2), \ldots, S(n-2)$), которых нет в рекурсивном определении (1.5). Поэтому такие подзадачи во избежание путаницы должны удаляться из схем и, следовательно, из самого рекурсивного процесса. Тем не менее вложенные и древовидные схемы могут быть полезными для других задач.

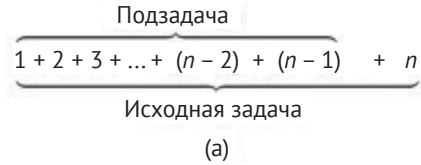
Стремление избежать вложенных схем не означает, что необходимо рассматривать только одну подзадачу. Многие рекурсивные определения требуют разложения исходной задачи на множество подзадач. Например, на рис. 2.4 приведены две схемы декомпозиции функции Фибоначчи. В частности, вариант (a) соответствует определению (1.2), а вариант (b) – определению (1.7). Несмотря на то что в схеме (b) показаны все подзадачи от $F(1)$ до $F(n-2)$, такое представление допустимо, так как эти подзадачи не являются вложенными.

В большинстве задач этой книги размер будет зависеть только от одного параметра или фактора, но в некоторых он будет зависеть от нескольких параметров. Например, матричные задачи обычно зависят от высоты (количества строк) и ширины (количества столбцов) матрицы. Поэтому задачи меньшего размера могут стать результатом уменьшения одного или обоих из этих параметров. В частности, многие рекурсивные алгоритмы делят матрицы на блоки – подматрицы. Такие «блочные матрицы» можно интерпретировать как исходную матрицу с вертикальными и горизонтальными разделителями, как показано в следующем примере с матрицей размером 4×7:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} = \left(\begin{array}{ccc|cccc} 3 & 2 & 1 & 0 & 3 & 4 & 9 \\ 7 & 2 & 3 & 3 & 1 & 7 & 5 \\ \hline 0 & 5 & 3 & 2 & 1 & 4 & 8 \\ 6 & 3 & 1 & 5 & 4 & 9 & 0 \end{array}\right),$$

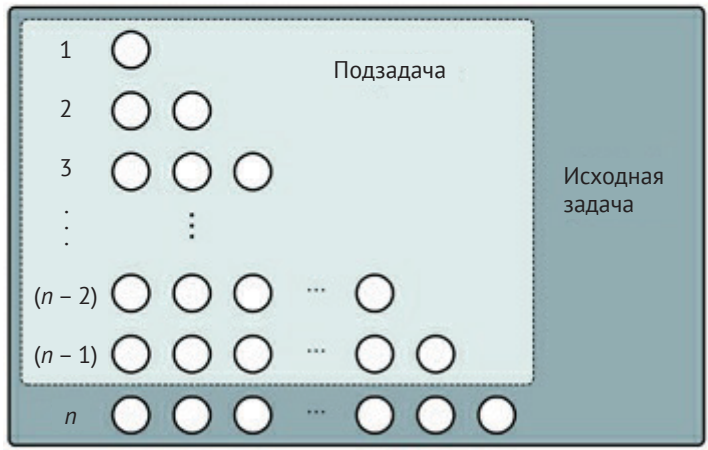
где $\mathbf{A}_{1,1} = \begin{pmatrix} 3 & 1 & 2 \\ 7 & 2 & 3 \end{pmatrix}$, $\mathbf{A}_{1,2} = \begin{pmatrix} 0 & 3 & 4 & 9 \\ 3 & 1 & 7 & 5 \end{pmatrix}$,

$\mathbf{A}_{2,1} = \begin{pmatrix} 0 & 5 & 3 \\ 6 & 3 & 1 \end{pmatrix}$, $\mathbf{A}_{2,2} = \begin{pmatrix} 2 & 1 & 4 & 8 \\ 5 & 4 & 9 & 0 \end{pmatrix}$.

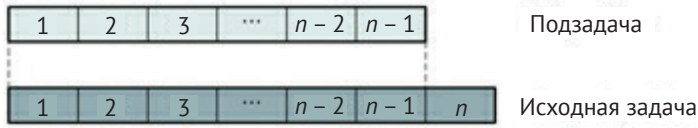


(a)

(b)



(c)



(d)

Рис. 2.2. Дополнительные схемы, иллюстрирующие декомпозицию суммы первых *n* положительных целых чисел с уменьшением размера задачи на единицу

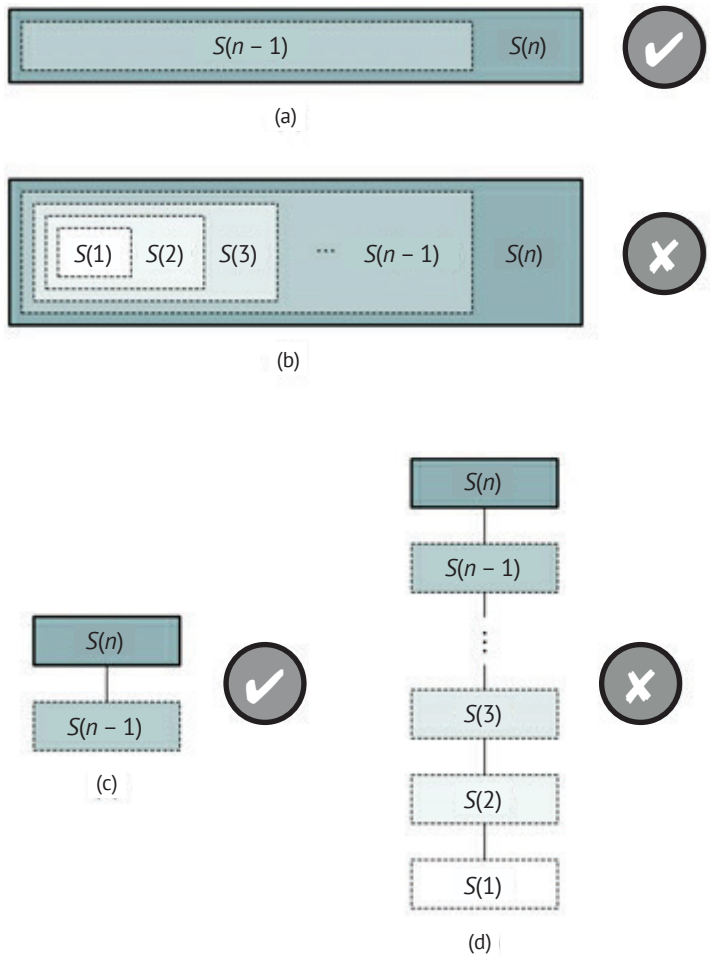
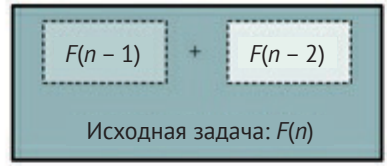


Рис. 2.3. При рекурсивном подходе обычно нет нужды представлять декомпозицию задачи всеми её экземплярами меньшего размера

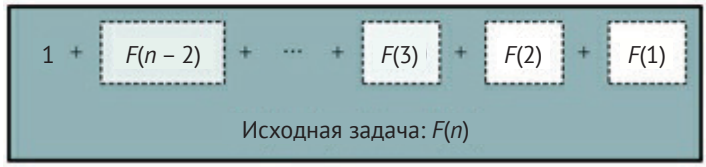
Здесь число строк и столбцов каждой подматрицы получено целочисленным делением на два исходных высоты и ширины матрицы.

Что касается эффективности метода деления размера задачи пополам, то он может приводить к более быстрым алгоритмам, чем постепенное уменьшение размера на 1. Поэтому такую стратегию нужно иметь в виду вообще и в особенности тогда, когда теория показывает возможное снижение стоимости вычислений алгоритма по сравнению с уменьшением размера задачи на единицу. Однако деление размера задачи пополам не всегда приводит к приемлемому рекурсивному ал-

горитму. Например, трудно получить простую рекурсивную формулу для функции $n!$ с использованием подзадачи $(n/2)!$.



(a)



(b)

Рис. 2.4. Декомпозиции функции Фибоначчи

2.5. Рекурсивные условия, индукция и схемы

Следующий шаг в шаблоне проектирования рекурсивных алгоритмов состоит из определения рекурсивных условий, одна из целей которого – понять, как построить полное решение исходной задачи, используя решения подобных ей подзадач, определённых на этапе декомпозиции. Как сказано в разделе 1.4, согласно методу индукции мы можем считать, что эти более простые решения готовы к использованию. Таким образом, главная проблема рекурсивного подхода – определить, как изменить, расширить или объединить решения подзадач, чтобы на последнем шаге схемы на рис. 1.4 получить полное решение исходной задачи.

2.5.1. Рекурсивное мышление посредством схем

Часто полезно представить мыслительный процесс или «мысленную модель» вывода рекурсивных условий в виде схемы. На рис. 2.5 приведена общая процедура рекурсивного мышления, когда декомпозиция задачи приводит к единственной, подобной ей подзадаче.

Процесс начинается с рассмотрения основного экземпляра задачи некоторого размера, определяемого входными параметрами, изображенными в верхнем левом блоке. Применение рекурсивного метода к этим параметрам должно привести к результатам, представленным в

верхнем правом блоке. Отметим, что этот верхний ряд блоков – просто ещё один способ определения условий задачи. Для определения рекурсивных условий мы сначала выбираем некоторую декомпозицию, приводящую к меньшему экземпляру задачи. Новые, более простые параметры для подзадачи, подобной исходной, показаны в левом нижнем блоке. Рекурсивный вызов метода с этими параметрами должен, таким образом, выдать результаты (правый нижний блок), которые получаются применением условий задачи к более простым входным параметрам. И наконец, поскольку согласно методу индукции эти результаты следует считать правильными, мы выводим рекурсивные условия с учётом изменений или расширений, позволяющих добиться общего решения исходной задачи (верхний правый блок).

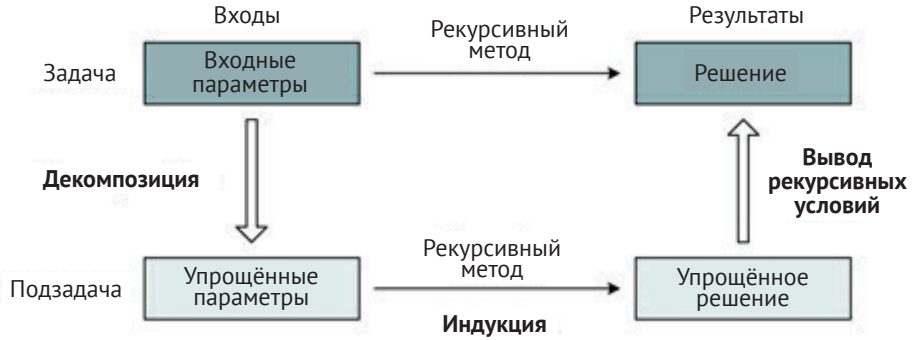
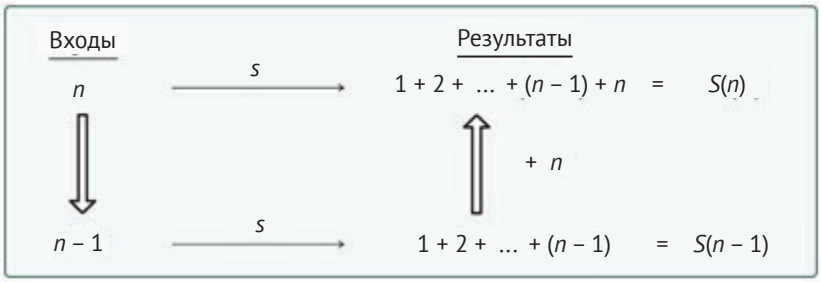


Рис. 2.5. Общая схема обдумывания рекурсивных условий (когда декомпозиция задачи приводит к единственной, подобной ей подзадаче)

Для суммы первых n положительных целых чисел ($S(n)$) схема может быть такой:



Для начала можно выбрать декомпозицию уменьшением размера задачи на единицу. Цель такой частной декомпозиции – выяснить, как можно получить $S(n)$, изменив или расширив решение подзада-

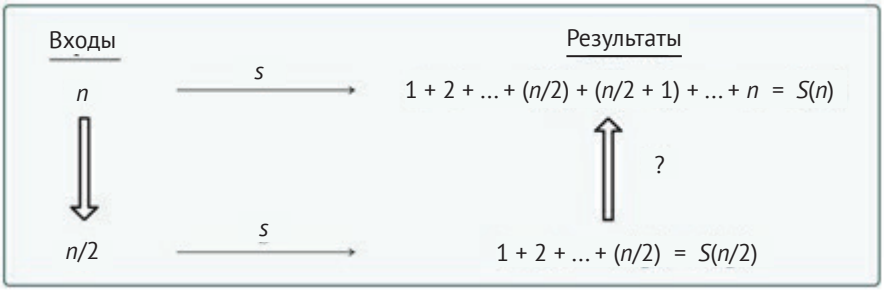
чи $S(n-1)$. В этом случае легко видеть, что $S(n)$ получается добавлением n к $S(n-1)$. Поэтому рекурсивное условие определяется как $S(n) = S(n-1) + n$.

Важно понимать, что нам нужно всего лишь определить декомпозицию и установить, как получить результат исходной задачи, используя результат подзадачи. Следовательно, нам нужно думать только о процессах, обозначенных на схеме толстыми стрелками. С другой стороны, тонкие стрелки обозначают лишь итог решения отдельных экземпляров задачи (для различных входных параметров), который полностью определен условиями задачи. Отметим, что здесь не принимается никаких алгоритмических проектных решений об отношении между входом и выходом, поскольку они задаются условиями задачи.

Достоинство схемы на рис. 2.5 – в её общности: её можно применять к многочисленным задачам для получения рекурсивных решений. Поэтому объяснения большинства примеров из этой книги опираются именно на неё. Конечно, не исключено применение и других графических представлений, таких как на рис. 1.5, 1.6 и 2.2. Такая альтернативная визуализация особенно полезна, поскольку она показывает не только подзадачи, но и их связь с исходной задачей, что помогает объединять, расширять или изменять решения подзадач ради решения исходной задачи. Другими словами, они дают наглядное представление о том, как определить рекурсивные условия. Например, из схемы на рис. 2.2 очевидно, что решение задачи заключается в добавлении n к результату подзадачи размера $n-1$.

Конечно, более сложные схемы, специально приспособленные для отдельных задач, могут помочь найти различные декомпозиции, которые могли бы в конечном счете привести к более эффективным алгоритмам. Например, представление суммы первых n положительных целых чисел в виде блоков, выложенных треугольными пирамидами, как на рис. 1.5, позволяет легко вывести рекурсивные условия, когда размер подзадач равен примерно половине исходного. Во-первых, напомним, что задачу можно свести к подсчёту числа блоков в треугольной структуре. Кроме того, поскольку представление позволяет наложить поверх исходной задачи несколько подзадач (то есть меньшие треугольные структуры) одновременно, визуализация показывает, как «заполнить» бóльшую пирамиду меньшими. Поэтому схемы иллюстрируют, как получить рекурсивные условия путём сложения результатов (в данном случае – количество блоков) меньших подзадач. Например, на рис. 1.5(c) для получения результата исходной задачи нужно сложить результаты трёх подзадач размера $n/2$ и результат подзадачи размера $n/2 - 1$.

Тем не менее рекурсивное решение методом деления размера задачи $S(n)$ пополам можно вывести и из общей схемы на рис. 2.5. В этом случае мы имеем:



Однако здесь возникает вопрос: как (если это возможно) получить $S(n)$ путём изменения или расширения $S(n/2)$. Первая очевидная идея приводит к $S(n) = S(n/2) + (n/2 + 1) + \ldots + n$. И хотя она правильная, её реализация потребует использования либо цикла, либо другой рекурсивной функции для вычисления суммы последних $n/2$ членов. Отметим, что на этапе построения функции (S) получить сумму $(n/2 + 1) + \ldots + n$ рекурсивным обращением к ней же невозможно, так как эта сумма не является экземпляром исходной задачи. Однако её можно преобразовать, например таким образом:

$$(n/2 + 1) + (n/2 + 2) + \ldots + (n/2 + n/2) = (n/2)^2 + (1 + 2 + \ldots + n/2) =$$

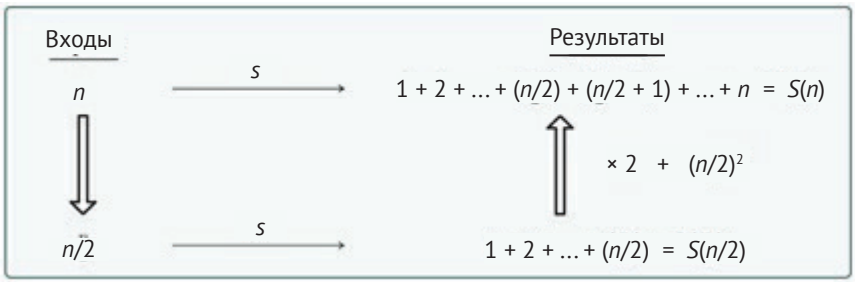
$$= (n/2)^2 + S(n/2).$$

Она не только упрощает выражение, но и содержит $S(n/2)$, чем можно воспользоваться для получения гораздо более простого рекурсивного условия:

$$S(n) = S(n/2) + (n/2 + 1) + \ldots + n = S(n/2) + S(n/2) + (n/2)^2 =$$

$$= 2S(n/2) + (n/2)^2. \qquad (2.1)$$

Теперь общую схему можно изобразить следующим образом:



Из неё следует, что $S(n)$ может быть получена умножением результата $S(n/2)$ на 2 и добавлением к нему $(n/2)^2$. Хотя декомпозиция задачи приводит к единственной подзадаче ($S(n/2)$), этот пример показывает, что для решения исходной задачи результат её подзадачи может использоваться несколько раз. В данном случае можно считать, что рекурсивное условие использует $S(n/2)$ *дважды*.

Это рекурсивное условие можно также получить из схемы на рис. 2.6, где $S(n)$ изображается в виде треугольной пирамиды и тем самым доказывается, что $S(n) = 2S(n/2) + (n/2)^2$. Естественность такой схемы, где решение задачи сводится к сложению блоков, облегчает получение рекурсивных условий.

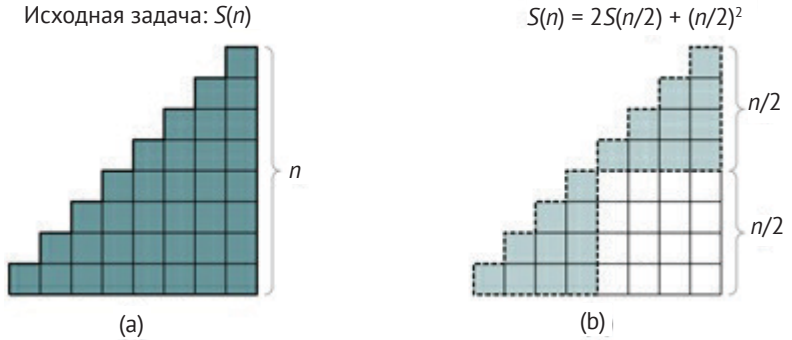


Рис. 2.6. Схема декомпозиции задачи о сумме первых n положительных целых чисел $S(n)$, использующая две подзадачи в половину размера исходной

2.5.2. Конкретные экземпляры задачи

В предыдущем примере сложения первых n положительных целых чисел результат рекурсивного метода легко выразить в виде общей функции от входного параметра n. Иными словами, выражения из правой части общей схемы здесь получаются легко. Однако в других задачах описать результаты рекурсивных методов (скажем, некоторой формулой) гораздо сложнее, и потому труднее понять, как вывести решение задачи из результатов её подзадач. В таких случаях на первых порах полезен подход, заключающийся в анализе конкретных экземпляров задачи.

Рассмотрим задачу сложения цифр некоторого неотрицательного целого числа n и допустим, что выбрана декомпозиция, уменьшающая количество цифр на 1 за счёт отбрасывания наименьшей значащей цифры исходного числа. Эта информация представлена в левых блоках общей схемы. Однако определение выхода, как и функции от n, в

правых блоках схемы может оказаться сложным. Например, его можно описать следующей суммой:

$$\sum_{i=0}^{\lfloor \log_{10} n \rfloor} (n//10^i)\%10. \qquad (2.2)$$

Несмотря на то что мы могли бы использовать эту формулу для вывода рекурсивных условий, мы продолжим анализировать конкретные экземпляры задачи. Например (ради простоты можно отказаться от использования имени метода в колонке результатов и от маркировки стрелок):

Из этих схем нетрудно видеть, что результат метода – это сумма последней (подчеркнутой) цифры исходного числа и результата подзадачи.

2.5.3. Альтернативные обозначения

Другой подход заключается в использовании некоторых альтернативных общих обозначений, которые упрощают создание схем. Пусть $d_{m-1}...d_1d_0$ – последовательность цифр некоторого неотрицательного целого числа n по основанию 10. Иными словами, $n = d_{m-1} \cdot 10^{m-1} + ... + d_1 \cdot 10 + d_0$, где $0 \leqslant d_i \leqslant 9$ для всех i и $d_{m-1} \neq 0$, если $m > 1$. В этом случае общая схема была бы такой:

а код функции был бы таким, как показано в листинге 2.2, где (n%10) представляет d_0. Отметим, что начальное условие этой задачи выполняется, когда n состоит из одной цифры (то есть $n < 10$), а его результат, очевидно, равен n.

Листинг 2.2. Вычисление суммы цифр неотрицательного целого числа

```
1  def add_digits(n):
2      if n < 10:
3          return n
4      else:
5           return add_digits(n // 10) + (n % 10)
```

И наконец, на практике для более сложных задач может потребоваться анализ многих примеров и сценариев (например, разделять чётные или нечётные входные параметры), приводящих к различным рекурсивным условиям.

2.5.4. Процедуры

До сих пор нам могло показаться, что если результат определяется формулой, то методу всегда соответствует функция, возвращающая значение. И потому он может использоваться в выражениях других методов, куда возвращает определённое значение от заданных входных параметров. Однако в некоторых языках программирования (например, в Паскале) существуют методы, называемые «процедурами», которые не возвращают значений. Вместо этого они могут изменять структуры данных, передаваемые методу в качестве аргументов, или просто отображать некоторую информацию на экране. Оказывается, что и в таких случаях также можно пользоваться общей схемой декомпозиции задачи.

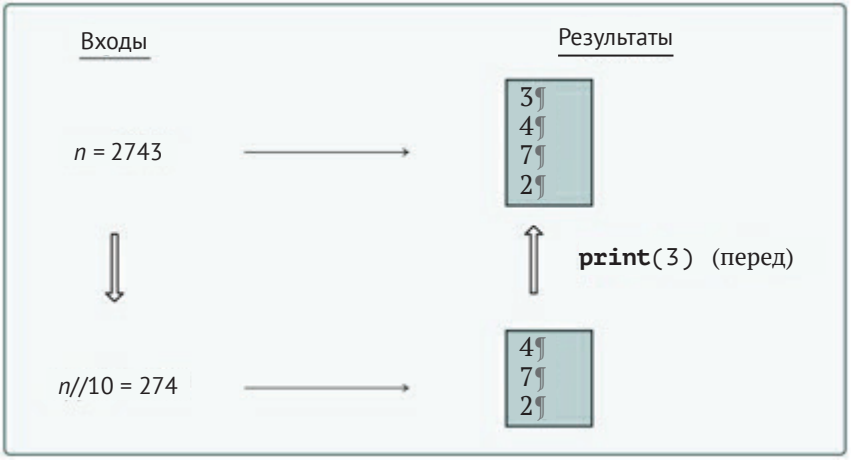
Рассмотрим задачу вывода на экран цифр некоторого неотрицательного целого числа n – по вертикали и в обратном порядке. То есть наи-

меньшая значащая цифра должна появиться в 1-й строке, следующая – во 2-й и т. д. Например, если n = 2743, программа должна вывести на экран следующие строки:

Во-первых, размер этой задачи – количество цифр в числе n. Начальное условие выполняется, когда n состоит из одной цифры (n < 10), и алгоритм просто должен вывести n. Как в предыдущем примере, самая простая декомпозиция рассматривает n//10 с отбрасыванием из исходного числа наименьшей значащей цифры. На рис. 2.7 показана возможная схема декомпозиции задачи.

Рис. 2.7. Схема декомпозиции задачи вывода на экран цифр неотрицательного целого числа – вертикально и в обратном порядке: (a) – частный случай (n = 2743), (b) – общий случай числа из m цифр ($n = d_{m-1} \ldots d_1 d_0$)

Кроме того, для такого типа процедур также применима общая схема:



В этом примере результатом являются не столько сами числовые значения, сколько последовательность команд вывода строк на экран. Для такой задачи решением будет вывод на экран наименьшей значащей цифры исходного числа с последующим рекурсивным вызовом метода для оставшихся цифр. При этом крайне важна последовательность выполнения этих операций. В частности, наименьшая значащая цифра должна быть выведена *до* вывода остальных. Соответствующий код приведён в листинге 2.3.

Листинг 2.3. Вывод на экран цифр неотрицательного целого числа – вертикально и в обратном порядке

```
def print_digits_reversed_vertically(n):
    if n < 10:
        print n
    else:
         print (n % 10)
         print_digits_reversed_vertically(n // 10)
```

2.5.5. Несколько подзадач

Некоторые алгоритмы требуют разбиения исходной задачи на несколько подобных ей подзадач, а процесс осмысления рекурсивных условий подобен изображённому на рис. 2.5. Как показано на рис. 2.8, схемы просто должны включать различные подзадачи и соответствующие им решения согласно выбранной декомпозиции. В таких случаях, помимо расширения и изменения решения каждой из подзадач, рекурсивные условия обычно должны ещё и объединять их.

Для заданного непустого списка из n целых чисел ($n \geqslant 1$) рассмотрим задачу поиска в нём наибольшего значения. В этом примере мы разделим размер задачи пополам так, чтобы работать с одной и с другой половинами списка, как показано на рис. 1.6(d). Следующая схема иллюстрирует процесс осмысления задачи на конкретном примере:

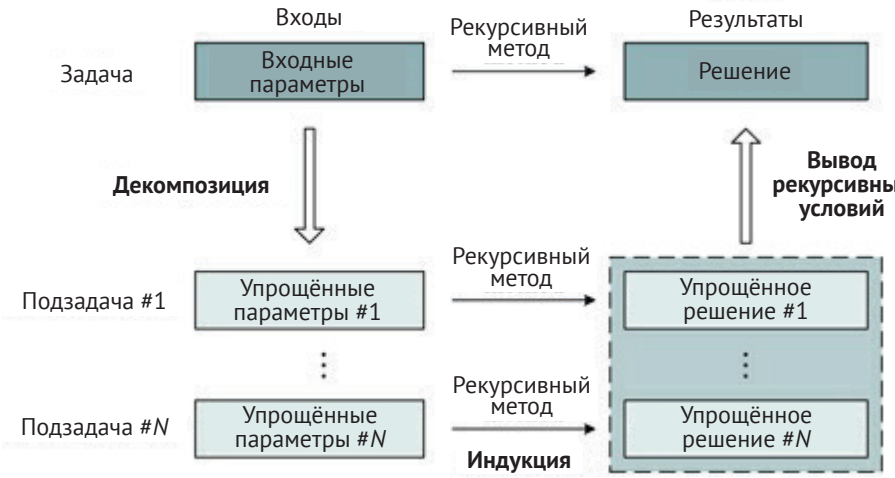


Рис. 2.8. Общая схема осмысления рекурсивных условий, когда задача разбивается на несколько (*N*) себе подобных подзадач

Толстые и тонкие стрелки обозначают решения задач и подзадач, соответственно.

На рис. 2.9 представлена альтернативная схема декомпозиции и процесс рекурсивного осмысления задачи.

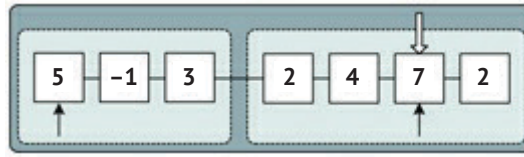


Рис. 2.9. Альтернативная схема декомпозиции методом «разделяй и властвуй» и процесс осмысления задачи поиска наибольшего значения в списке

В каждом из вариантов рекурсивный вызов возвращает наибольшее значение в каждой половине. Поэтому рекурсивное условие может возвращать просто максимальное значение из двух половин. Рекурсивная функция (*f*) определена следующим образом:

$$f(\mathbf{a}) = \begin{cases} \mathbf{a}[0], & \text{если } n = 1, \\ \max\left(f(\mathbf{a}[0 : n//2]), f(\mathbf{a}[n//2 : n])\right), & \text{если } n > 1. \end{cases} \quad (2.3)$$

В листинге 2.4 приводятся два способа кодирования функции. Версия, использующая границы `lower` и `upper`, обычно быстрее. Конечно, эта задача допускает также рекурсивные решения на основе единствен-

ной подзадачи, размер которой уменьшается на 1. Этот подход столь же эффективен, как метод «разделяй и властвуй», но на практике может привести к ошибкам времени выполнения программы для больших списков.

Листинг 2.4. Вычисление максимального значения в списке методом «разделяй и властвуй»

```
def max_list_length_DaC(a):
    if len(a) == 1:
        return a[0]
    else:
        middle len(upper + lower) // 2
        m1 = max_list_length_DaC(a[0:middle)
        m2 = max_list_length_DaC(a[middle:len(a))
        return max(m1, m2)


def max_list_limits_DaC(a, lower, upper):
    if lower == upper:
       return a[lower]    # or a[upper]
    else:
        middle = (upper + lower) // 2
        m1 = max_list_limits_DaC(a, lower, upper)
        m2 = max_list_limits_DaC(a, middle + 1, upper))
        return max(m1, m2)


# Some list:
v = [5, -1, 3, 2, 4, 7, 2]

# Function calls:
print (max_list_length_DaC(v))
print (max_list_limits_DaC(v, 0, len(v) - 1)
```

2.6. Тестирование

Тестирование – основной этап в любом процессе разработки программного обеспечения. В контексте данной книги его основная цель – обнаружение ошибок в коде. Таким образом, тестирование – это отладка разработанного программного обеспечения на различных экземплярах (различных входных параметрах) задачи с целью обнаружения непо-

ладок. Начинающим программистам рекомендуется тестировать свой код, так как способность находить и исправлять ошибки (например, с помощью отладчика) является основным навыком программиста. Кроме того, тестирование – это ценнейший опыт по устранению ловушек и созданию более эффективного и надёжного кода.

Помимо проверки правильности начальных и рекурсивных условий, при тестировании рекурсивного кода программисты должны обращать особое внимание на возможные сценарии, приводящие к бесконечным рекурсиям. Обычно они возникают из-за недостаточных начальных условий или из-за ошибочных рекурсивных условий. Рассмотрим, например, функцию из листинга 2.5.

Листинг 2.5. Неправильный код для определения чётности неотрицательного целого числа *n*

```
def is_even_incorrect(n):
    if n == 0:
        return True
    else:
         return is_even_incorrect(n - 2)
```

Её цель – определить, является ли некоторое неотрицательное целое число *n* чётным. И начальные, и рекурсивные условия верны. Конечно, если число *n* – чётное, значит, (*n* – 2) – тоже чётное, и функция должна вернуть для обоих целых чисел одно и то же логическое значение. Однако `is_even_incorrect` работает только для чётных чисел. Пусть $f(n)$ обозначает `is_even_incorrect(n)`, тогда вызов $f(7)$ приведёт к следующей цепочке рекурсивных вызовов:

$$f(7) \to f(5) \to f(3) \to f(1) \to f(-1) \to f(-3) \to \ldots,$$

что является бесконечной рекурсией, так как процесс не останавливается на начальном условии. То, что функция не имеет начального условия, возвращающего False, – это сигнал о том, что её нужно исправить (однако не все логические функции требуют двух начальных условий для возвращения True или False соответственно). В данном случае метод можно исправить, добавив в него ещё одно начальное условие. В листинге 2.6 приведена функция, работающая для любого допустимого параметра ($n \geqslant 0$).

Листинг 2.6. Правильный код для определения чётности неотрицательного целого числа *n*

```
def is_even_correct(n):
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
         return is_even_correct(n - 2)
```

Другой пример – функция из листинга 2.7, которая для вычисления суммы первых *n* положительных целых чисел ($S(n)$) использует рекурсивное условие (2.1).

Листинг 2.7. Ошибочное суммирование первых *n* положительных целых чисел, порождающее бесконечную рекурсию для большинства значений *n*

```
def sum_first_naturals_3(n):
    if n == 1:
        return 1
    else:
         return 2 * sum_first_naturals_3(n / 2) + (n / 2)**2
```

Оно – неполное и генерирует бесконечную рекурсию для значений *n*, не являющихся степенью двух. Во-первых, поскольку язык Python считает *n* вещественным числом, $n/2$ – тоже вещественное число. Поэтому если параметр *n* окажется нечётным в каком-нибудь рекурсивном вызове функции, то $n/2$ будет иметь дробную часть, и такими же будут аргументы последующих рекурсивных вызовов. Таким образом, алгоритм не остановится в начальном условии при $n = 1$ (которое является целым числом без дробной части) и продолжит вызывать функции с меньшими и меньшими аргументами. Например, пусть $f(n)$ – это `sum_first_naturals_3(n)`. Вызов $f(6)$ порождает следующую цепочку рекурсивных вызовов:

$$f(6) \to f(3) \to f(1.5) \to f(0.75) \to f(0.375) \to f(0.1875) \to \ldots,$$

которая никогда не прервётся на начальном условии. Алгоритм работает лишь в одном случае – когда первый параметр *n* является степенью двух, поскольку каждое деление на два даёт чётное число, и по достижении $n = 2$ мы получим $n = 1$, для которого функция может, наконец, вернуть вместо очередного вызова самой себя конкретное значение.

Метод `sum_first_naturals_3` работает неправильно, потому что его аргументы – вещественные числа. Таким образом, можно заменить вещественное деление целочисленным, как в листинге 2.8, что сделает аргументы целочисленными и предотвратит бесконечную рекурсию.

Листинг 2.8. Неполный код при сложении первых *n* положительных чисел

```
def sum_first_naturals_4(n):
    if n == 1:
        return 1
    else:
         return 2 * sum_first_naturals_4(n // 2) + (n // 2)**2
```

Однако и после этого функция все ещё будет работать неправильно для тех аргументов, которые не являются степенью двух. Проблема в том, что `sum_first_naturals_4` – тоже неполная. В частности, несмотря на правильность рекурсивного условия, оно применимо только к чётным значениям *n*. На рис. 2.10 показано, как в декомпозиции задачи получить рекурсивное условие $S(n) = 2S((n-1)/2) + ((n+1)/2)^2$ для нечётных *n*.

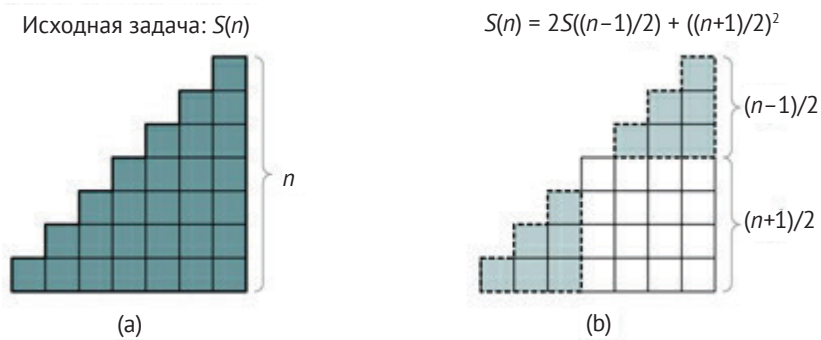


Рис. 2.10. Схема декомпозиции суммы первых *n* положительных целых чисел *S*(*n*) на две подзадачи (примерно) в половину размера исходной для нечётных значений *n*

Следовательно, окончательная функция:

$$S(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2 \times S(n/2) + (n/2)^2, & \text{если } n > 1 \text{ и } n \text{ – чётное}, \\ 2 \times S((n-1)/2 + ((n+1)/2)^2, & \text{если } n > 1 \text{ и } n \text{ – нечётное}, \end{cases}$$

а соответствующий код – в листинге 2.9.

Листинг 2.9. Сумма первых *n* положительных чисел с двумя подзадачами, размером (примерно) в половину исходной

```
def sum_first_naturals_5(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return 2 * sum_first_naturals_5(n / 2) + (n / 2)**2
    else:
         return (2 * sum_first_naturals_5(n - 1) / 2)
                 + (n + 1) / 2)**2)
```

При новом рекурсивном условии каждый, начиная с исходного, аргумент функции `sum_first_naturals_5` также будет целым числом. В итоге замена вещественного деления (/) целочисленным (//) также приводит к правильному алгоритму.

2.7. Упражнения

Упражнение 2.1. Пусть n – некоторое положительное целое число. Рассмотрим задачу подсчёта количества единичных разрядов в двоичном представлении n (то есть по основанию 2). Например, для $n = 25_{10} = 11001_2$ (нижний индекс указывает основание системы счисления) результат содержит три единичных разряда. Определите размер задачи и его математическое выражение.

Упражнение 2.2. Рассмотрим функцию суммирования первых n положительных целых чисел (1.5). Определите обобщённую функцию, которая применима также ко всем неотрицательным целым числам. Другими словами, измените функцию с учётом того, что она может получать в качестве аргумента $n = 0$. После этого закодируйте функцию.

Упражнение 2.3. Примените схемы, подобные схемам на рис. 2.6 и 2.10, для вывода рекурсивных определений суммы первых n положительных целых чисел ($S(n)$), когда рекурсивное условие суммирует результаты четырёх подзадач (примерно) в половину размера исходной. После этого определите и закодируйте полную рекурсивную функцию.

Упражнение 2.4. Рассмотрим задачу вывода на экран цифр некоторого неотрицательного целого числа n вертикально в прямом порядке, где старший разряд должен появиться в 1-й строке, следующий за

ним – во 2-й и т. д. Например, если n = 2743, программа должна выдать на экран следующие строки:

Определите размер задачи и её начальное условие. Для иллюстрации декомпозиции задачи приведите её схему для любого неотрицательного входа – целого числа $n = d_{m-1} \ldots d_1 d_0$, где m – количество цифр n, и покажите на схеме, как получить решение задачи по результату её подзадачи. В заключение выведите рекурсивное условие и закодируйте метод.

Упражнение 2.5. Используя принцип «разделяй и властвуй», определите общую схему задачи поиска наибольшего значения в списке из n элементов. Используйте соответствующие общие обозначения вместо конкретных примеров.

Упражнение 2.6. Определите рекурсивную функцию, вычисляющую наибольшее значение в списке из n элементов, когда декомпозиция просто уменьшает размер задачи на 1.

Глава 3

Анализ времени выполнения рекурсивных алгоритмов

Чем быстрее идёшь, тем меньше становишься.

– Альберт Эйнштейн

Анализ алгоритмов – область, изучающая теоретическую оценку ресурсов, необходимых алгоритму для решения вычислительной задачи. Эта глава посвящена анализу времени выполнения или «временно́й вычислительной сложности» рекурсивных алгоритмов решения задач, размер которых зависит от единственного фактора (который присутствует в большинстве задач этой книги). Она охватывает практически весь спектр математических средств, необходимых для оценки и сравнения эффективности различных алгоритмов. В частности, в главе приводятся два метода решения рекуррентных соотношений – рекурсивных математических функций, описывающих стоимость вычисления рекурсивных алгоритмов. Эти методы используются для преобразования рекуррентных соотношений в эквивалентные нерекурсивные формулы, которые проще понять и сравнить. Кроме того, в главе приводится краткий обзор основных математических понятий и обозначений, используемых при анализе алгоритмов. Анализу затрат памяти или пространственной сложности рекурсивных алгоритмов будет посвящена глава 6.

3.1. Предварительные математические соглашения

Этот раздел представляет собой краткое введение в основные математические определения и свойства, используемые при анализе вычисли-

тельной сложности алгоритмов вообще, а также ряда конкретных задач из этой книги.

3.1.1. Степени и логарифмы

В следующем списке приводится обзор основных свойств степеней и логарифмов:

- $b^1 = b$
- $b^x b^y = b^{x+y}$
- $b^{-x} = 1/b^x$
- $\log_b b = 1$
- $\log_b(xy) = \log_b(x) + \log_b(y)$
- $\log_b(x^y) = y \log_b x$
- $\log_b a = 1/\log_a b$
- $\log_b(b^x) = x$
- $b^0 = 1$
- $(b^x)^y = b^{xy} = (b^y)^x$
- $(ab)^x = a^x b^x$
- $\log_b 1 = 0$
- $\log_b(x/y) = \log_b(x) - \log_b(y)$
- $\log_b x = \log_a x/\log_a b$
- $x^{\log_b y} = y^{\log_b x}$
- $b^{\log_b a} = a$

где a, b, x и y – произвольные вещественные числа, за исключением того, что: 1) основание логарифма должно быть положительным и отличным от 1, 2) логарифм определён только для положительных чисел и 3) знаменатель при делении не может быть равным нулю. Например, $\log_b x = \log_a x/\log_a b$ имеет место только при $a > 0$, $a \neq 1$, $b > 0$, $b \neq 1$ и $x > 0$.

Логарифмы и степени положительных чисел – монотонно возрастающие функции. Поэтому если $x \leqslant y$, то $\log_b x \leqslant \log_b x$ и $b^x \leqslant b^y$ (для допустимых значений x, y и b).

3.1.2. Биномиальные коэффициенты

Биномиальный коэффициент C_n^m – это целое число, которое появляется в полиномиальном представлении биномиальной степени $(1 + x)^n$. Его можно определить как:

$$C_n^m = \begin{cases} 1, & \text{если } m = 0 \text{ или } n = m, \\ n!/(m!\,(n-m)!) & \text{иначе,} \end{cases} \tag{3.1}$$

где n и m – целые числа, удовлетворяющие условию $n \geqslant m \geqslant 0$. Кроме того, биномиальный коэффициент может быть определён рекурсивно:

$$C_n^m = \begin{cases} 1, & \text{если } m = 0 \text{ или } n = m, \\ C_{n-1}^{m-1} + C_{n-1}^m & \text{иначе.} \end{cases} \tag{3.2}$$

Биномиальные коэффициенты играют важную роль в комбинаторике. В частности, C_n^m определяет количество способов выбора m различных элементов из множества, состоящего из n элементов, когда порядок выбора не имеет значения.

3.1.3. Пределы и правило Лопиталя

Стоимость вычисления различных алгоритмов можно сравнить с помощью предела отношения функций. Прежде всего

$$k/\infty = 0, \quad \infty/k = \infty,$$

где k – константа, а символ ∞ следует понимать как значение предела. Кроме того, так как функции оценки стоимости вычислений, вообще говоря, возрастают (за исключением констант), их предел стремится к бесконечности при стремлении к бесконечности их аргументов. Например:

$$\lim_{n\to\infty} \log_b n = \infty \text{ или } \lim_{n\to\infty} n^a = \infty$$

при допустимом основании логарифма b и $a > 0$. Поэтому зачастую при сравнении стоимости вычислений алгоритмов может возникнуть неопределённость вида

$$\lim_{n\to\infty} f(n)/g(n) = \infty/\infty.$$

Обычно она разрешается упрощением дроби $f(n)/g(n)$, пока не появится возможность получить отличный от неопределённости результат. Известный способ упрощения дроби – применение правила Лопиталя:

$$\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n), \tag{3.3}$$

где $f'(n)$ и $g'(n)$ – производные $f(n)$ и $g(n)$ соответственно. Формально правило Лопиталя верно только тогда, когда существует предел в правой части (3.3), что обычно имеет место.

3.1.4. Суммы и произведения

Поскольку суммы появляются не только в определениях функций и в формулах, которые приходится кодировать, но и в оценках эффективности итерационных и рекурсивных алгоритмов, важно владеть математическим аппаратом суммирования. Сумма, или суммирование, – это просто сложение множества математических объектов (целых или вещественных чисел, векторов, матриц, функций и т. д.). Сумму значений некоторой функции $f(i)$ для последовательности целочисленных

значений i от начального значения m до конечного значения n можно записать коротко с использованием символа суммирования:

$$\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \ldots + f(n). \tag{3.4}$$

Таким образом, результат сложения – это просто сумма членов, которые получаются при подстановке в функцию $f(i)$ всех значений i от от m до n. Например:

$$\sum_{i=0}^{4} k \cdot i^2 = k \cdot 0^2 + k \cdot 1^2 + k \cdot 2^2 + k \cdot 3^2 + k \cdot 4^2,$$

где $f(i) = k \cdot i^2$.

Важно понимать, что результат суммирования не зависит от индексной переменной i. Например, сумма в (3.4) зависит от функции f и заданных граничных значений m и n, но не от конкретных значений i. Кстати, заметьте, что i нет в правой части (3.4). Поэтому целочисленный индекс i – это просто вспомогательная переменная, позволяющая одним выражением задать все слагаемые суммы. Таким образом, индексная переменная может иметь любое имя – скажем, j или k:

$$\sum_{i=m}^{n} f(i) = \sum_{j=m}^{n} f(j) = \sum_{k=m}^{n} f(k).$$

Более того, с помощью замены индексной переменной одну и ту же сумму можно выразить различными способами:

$$\sum_{i=m}^{n} f(i) = \sum_{i=m-1}^{n-1} f(i+1) = \sum_{i=m}^{n} f(n-i+m).$$

Во второй сумме смещены пределы (и аргумент f) суммы, а в третьей элементы суммы складываются в обратном порядке (при $i = m$ к сумме добавляется $f(n)$, а при $i = n$ к сумме добавляется $f(m)$). Наконец, если нижний предел суммы m больше верхнего n, сумма по умолчанию считается равной 0.

3.1.4.1. *Основные свойства сумм*

Следующие основные свойства полезны для упрощения и преобразования сумм и могут быть легко получены из свойств сложения и умножения суммируемых элементов. Во-первых:

$$\sum_{i=1}^{n} 1 = \overbrace{1 + 1 + \ldots + 1 + 1}^{n \text{ раз}} = n.$$

Обратите внимание, что $f(i)$ – постоянная величина (1), не зависящая от индекса i. Точно так же

$$\sum_{i=1}^{n} k = \overbrace{k + k + \ldots + k + k}^{n \text{ раз}} = k \cdot n.$$

Предыдущий пример показывает также, что если один и тот же множитель в f не зависит от индекса, его можно вынести за знак суммы:

$$\sum_{i=1}^{n} k = \sum_{i=1}^{n} (k \cdot 1) = \sum_{i=1}^{n} k \cdot 1 = k \sum_{i=1}^{n} 1 = k \cdot n. \tag{3.5}$$

В этом случае постоянный множитель k (считаем, умноженный на 1) не зависит от i и может быть вынесен за знак суммы, чтобы стать множителем всей суммы. Из (3.5) видно, что скобки внутри суммы не нужны в тех случаях, когда f не содержит внутри себя операций сложения. В общем виде это свойство выглядит так:

$$\sum_{i=m}^{n} k \cdot f(i) = k \sum_{i=m}^{n} f(i),$$

что следует из дистрибутивного закона умножения суммы, где выражение можно упростить, вынося общий множитель k из всех слагаемых. Множитель k может быть произведением нескольких множителей, которые не зависят от индекса и могут содержать верхние и нижние пределы, как показано в следующем примере:

$$\sum_{i=m}^{n} a^m n^2 i^3 = a^m n^2 \sum_{i=m}^{n} i^3,$$

где a – некоторая константа.

Наконец, суммы, в которых функция f состоит из нескольких слагаемых, можно разложить на несколько сумм, а именно:

$$\sum_{i=m}^{n} (f_1(i) + f_2(i)) = \sum_{i=m}^{n} f_1(i) + \sum_{i=m}^{n} f_2(i).$$

3.1.4.2. *Арифметическая прогрессия*

Арифметическая прогрессия – это числовая последовательность s_i (i = 0, 1, 2, ...), каждый член которой больше предыдущего на заданную константу d (которая может быть отрицательной). Иными словами,

$s_i = s_{i-1} + d$ для $i > 0$, что является рекурсивным определением. Элементы арифметической прогрессии можно также определить нерекурсивно следующим образом:

$$s_i = s_{i-1} + d = s_{i-2} + 2 \cdot d = \ldots = s_0 + i \cdot d. \tag{3.6}$$

Хотя теоретически арифметическая прогрессия бесконечна, а её сумма равна

$$\sum_{i=0}^{\infty} s_i,$$

для оценки времени выполнения алгоритмов интерес представляют только конечные последовательности

$$\sum_{i=m}^{n} s_i,$$

называемые также частичными суммами.

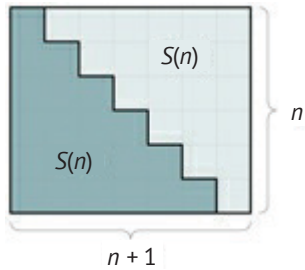
При анализе эффективности рекурсивных и итерационных алгоритмов часто используется сумма первых n положительных целых чисел $S(n)$, под которой подразумевается частичная сумма (n элементов) арифметической прогрессии ($s_0 = 1$ и $d = 1$):

$$S(n) = \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} (s_0 + i \cdot d) = \sum_{i=0}^{n-1} (1 + i) = \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} i = n + \sum_{i=0}^{n-1} i = \sum_{i=1}^{n} i.$$

Как показано в разделе 1.4.1, результат этой суммы можно представить в виде квадратного полинома от n (1.12). Эту формулу просто вывести, сложив две суммы $S(n)$, одна из которых суммируется в возрастающем порядке, а другая – в убывающем. Тогда из полученного результата следует, что $2S(n) = n(n + 1)$, поскольку в правой части равенства мы имеем n слагаемых, каждое из которых равно $n + 1$. После деления правой части на 2 получаем

$$S(n) = \sum_{i=1}^{n} i = n(n+1)/2.$$

Графическая иллюстрация этой интересной идеи приведена на рис. 3.1, где $S(n)$ – площадь треугольной «лесенки» из квадратиков размера 1×1. Две такие «лесенки» можно соединить так, чтобы получился прямоугольник размером $n \times (n + 1)$. Из чего следует, что площадь $S(n)$ каждой «лесенки» равна $n(n + 1)/2$.



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

$$\Rightarrow \quad S(n) = (n(n+1))/2$$

Рис. 3.1. Графический способ вывода формулы для суммы $S(n)$ первых n положительных целых чисел

Подобную формулу можно получить и для частичной суммы n членов арифметической прогрессии ($s_i = s_{i-1} + d$ для некоторого начального значения s_0) $\sum_{i=0}^{n-1} s_i = n(s_0 + s_{n-1})/2$. Иными словами, эта сумма есть среднее арифметическое первого и последнего членов последовательности, умноженное на количество её членов.

3.1.4.3. *Геометрическая прогрессия*

Геометрическая прогрессия – это числовая последовательность s_i ($i = 0, 1, 2, \ldots$), каждый член которой равен предыдущему, умноженному на заданную константу r. Иными словами, $s_i = r \cdot s_{i-1}$ для $i > 0$. Нерекурсивный вариант этой же формулы:

$$s_i = r \cdot s_{i-1} = r^2 \cdot s_{i-2} = \ldots = r^i \cdot s_0. \tag{3.7}$$

Частичная сумма геометрической прогрессии выражается следующей формулой:

$$\sum_{i=m}^{n} s_i = \sum_{i=m}^{n} r^i s_0 = s_0 \sum_{i=m}^{n} r^i = s_0 (r^m - r^{n+1})/(1-r), \tag{3.8}$$

где $r \neq 1$. Последнее равенство можно легко вывести следующим образом. Пусть S представляет собой значение суммы. Создадим другую сумму, умножив S на r. А теперь вычтем из одной другую (где большинство членов сокращается) и решим полученное уравнение относительно S:

$$S = \sum_{i=m}^{n} r^i = (r^m - r^{n+1})/(1-r) = (r^{n+1} - r^m)/(r-1). \tag{3.9}$$

В заключение отметим, что при $|r| < 1$ сумма бесконечной геометрической прогрессии сходится к постоянному значению:

$$\sum_{i=0}^{\infty} r^i = 1/(1-r).$$

3.1.4.4. *Дифференцирование*

Ещё одна полезная сумма:

$$S = \sum_{i=1}^{n} i \cdot r^i = 1 \cdot r^1 + 2 \cdot r^2 + 3 \cdot r^3 + \ldots + n \cdot r^n, \qquad (3.10)$$

которую можно считать гибридом сумм арифметической и геометрической прогрессий. Формулу для этой суммы можно вывести следующим образом. Сначала рассмотрим частичную сумму геометрической прогрессии и соответствующую ей сокращённую формулу:

$$T = 1 + r + r^2 + \ldots + r^n = (r^{n+1} - 1)/(r - 1).$$

Затем продифференцируем её по r:

$$\frac{dT}{dr} = 1 + 2r + 3r^2 + \ldots + nr^{n-1} = (nr^{n+1} - (n+1)r^n + 1)/(r-1)^2 \qquad (3.11)$$

и умножим на r, чтобы получить формулу для (3.10):

$$S = \sum_{i=1}^{n} i \cdot r^i = r + 2r^2 + 3r^3 + \ldots + nr^n = r \cdot (nr^{n+1} - (n+1)r^n + 1)/(r-1)^2. \qquad (3.12)$$

Эти формулы можно использовать для упрощения похожих сумм. Например, пусть

$$S = \sum_{i=1}^{N+1} i \cdot 2^{i-1} = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2^2 + 4 \cdot 2^3 + \ldots + (N+1) \cdot 2^N.$$

Эта сумма – частный случай (3.11), когда $r = 2$ и $n = N + 1$. После подстановки этих значений в общую формулу получаем:

$$S = ((N+1)2^{N+2} - (N+2)2^{N+1} + 1)/1 = 2^{N+1}(2N + 2 - N - 2) + 1 = N\,2^{N+1} + 1.$$

3.1.4.5. *Произведения*

Подобно сумме, произведение нескольких последовательных значений некоторой функции $f(i)$ для целочисленных значений i – от начального m до конечного n – можно записать следующим образом:

$$\prod_{i=m}^{n} f(i) = f(m) \cdot f(m+1) \cdot \ldots \cdot f(n-1) \cdot f(n), \qquad (3.13)$$

что по нашему соглашению при $m > n$ равно 1. Например, функцию вычисления факториала можно записать как

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n,$$

где ради математического удобства считается, что 0! = 1.

В произведениях, как и в суммах, не зависящие от индексной переменной множители можно выносить за знак произведения. При этом если произведение состоит из n членов, то этот множитель должен быть возведён в степень n:

$$\prod_{i=1}^{n} k \cdot f(i) = k^n \prod_{i=1}^{n} f(i).$$

Кроме того, в самом общем случае произведение сумм не равно сумме произведений:

$$\prod_{i=m}^{n} (f_1(i) + f_2(i)) \neq \prod_{i=m}^{n} f_1(i) + \prod_{i=m}^{n} f_2(i).$$

Наконец, логарифм произведения есть сумма логарифмов его множителей:

$$\log \prod_{i=m}^{n} f(i) = \sum_{i=m}^{n} \log f(i).$$

3.1.5. Верхняя и нижняя границы

Нижняя граница $\lfloor x \rfloor$ вещественного числа x – это наибольшее целое число, не превосходящее x. Аналогично, верхняя граница $\lceil x \rceil$ вещественного числа x – это наименьшее целое число, не меньшее x. Формально они могут быть определены как

$$\lfloor x \rfloor = \max \{m \in \boldsymbol{Z} \mid m \leqslant x\}, \quad \lceil x \rceil = \min \{m \in \boldsymbol{Z} \mid m \geqslant x\},$$

где $\boldsymbol{Z}$ – множество целых чисел, а символ «|» читается как «такие, что». Следующий список включает несколько основных свойств нижних и верхних границ:

- $\lfloor x \rfloor \leqslant x$
- $\lfloor x + n \rfloor = \lfloor x \rfloor + n$
- $\lfloor x \rfloor + \lfloor y \rfloor \leqslant \lfloor x + y \rfloor$
- $\lceil x \rceil \geqslant x$
- $\lceil x + n \rceil = \lceil x \rceil + n$
- $\lceil x \rceil + \lceil y \rceil \geqslant \lceil x + y \rceil$

- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$
- $n - 2\lfloor n/2 \rfloor = 0 \Leftrightarrow n$ – чётное
- $n//2 = \lfloor n/2 \rfloor$
- $n \gg m = \lfloor n/2^m \rfloor$
- $\lfloor \log_{10} p \rfloor + 1$ = количество десятичных цифр
- $\lfloor \log_2 p \rfloor + 1$ = количество двоичных цифр (битов)

где x – вещественное число, n – целое число, m – неотрицательное целое число и p – положительное целое число. Операция // вычисляет частное от целочисленного деления, а операция $\gg$ сдвигает целое число вправо на один двоичный разряд. Такие операции есть в языке Python и других языках программирования. Наконец, символ $\Leftrightarrow$ означает «тогда и только тогда».

3.1.6. Тригонометрия

На рис. 3.2 изображён прямоугольный треугольник, а ниже – связанные с ним основные тригонометрические определения и свойства:

- $\sin(\alpha) = a/c$
- $\cos(\alpha) = b/c$
- $\tan(\alpha) = \sin(\alpha)/\cos(\alpha) = a/b$
- $\cot(\alpha) = \cos(\alpha)/\sin(\alpha) = b/a$
- $\sin(0) = 0$
- $\cos(0) = 1$
- $\sin(30^\circ) = \sin(\pi/6) = 1/2$
- $\cos(30^\circ) = \cos(\pi/6) = \sqrt{3}/2$
- $\sin(45^\circ) = \sin(\pi/4) = \sqrt{2}/2$
- $\cos(45^\circ) = \cos(\pi/4) = \sqrt{2}/2$
- $\sin(60^\circ) = \sin(\pi/3) = \sqrt{3}/2$
- $\cos(60^\circ) = \cos(\pi/3) = 1/2$
- $\sin(90^\circ) = \sin(\pi/2) = 1$
- $\cos(90^\circ) = \cos(\pi/2) = 0$
- $\sin(\alpha) = -\sin(-\alpha)$
- $\cos(\alpha) = \cos(-\alpha)$

где sin, cos, tan и cot обозначают тригонометрические функции синус, косинус, тангенс и котангенс соответственно. В большинстве языков программирования параметр тригонометрических функций измеряется в радианах (1 радиан = $180^\circ/\pi$).

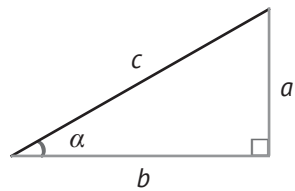


Рис. 3.2. Прямоугольный треугольник как иллюстрация тригонометрических определений

3.1.7. Векторы и матрицы

Матрица – прямоугольная таблица из чисел (или элементов других типов данных – символьных, логических и т. д.), состоящая из строк и столбцов. Формально матрица **A** размерности $n \times m$ содержит n строк и m столбцов чисел, где $n \geqslant 1$ и $m \geqslant 1$. Обычно матрицы заключаются в квадратные или круглые скобки:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,m} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,m} \end{pmatrix},$$

где $a_{i,j}$ обозначает отдельный элемент матрицы **A** или вход в неё по строке i и столбцу j.

Если одна из размерностей равна 1, то такой математический объект называют *вектором*. Если же обе размерности равны 1, то объект становится *скаляром*, то есть числом. В этой книге мы будем придерживаться стандартов, используя для обозначения матриц прописные буквы и жирный шрифт (**A**), для векторов – строчные буквы и жирный шрифт (**a**), для скаляров – строчные буквы или символы и курсив (a).

Результатом транспонирования матрицы **A** размерности $n \times m$ является матрица $\mathbf{A}^{\mathsf{T}}$ размерности $m \times n$, строками которой выступают столбцы матрицы **A** (а столбцами – строки **A**). Например, если

$$\mathbf{A} = \begin{pmatrix} 3 & 4 & 2 \\ 1 & 8 & 5 \end{pmatrix}, \text{ то } \mathbf{A}^{\mathsf{T}} = \begin{pmatrix} 3 & 1 \\ 4 & 8 \\ 2 & 5 \end{pmatrix}.$$

Матрицы (и векторы) можно складывать и умножать. Сумма **A** + **B** – это матрица, элементы которой равны $a_{i,j} + b_{i,j}$. Иными словами, матрицы (и векторы) складываются поэлементно. Поэтому **A** и **B** должны иметь одинаковую размерность. Типичный пример суммы векторов:

$$(4, -1, 2) + (2, 3, -7) = (6, 2, -5).$$

Умножение матриц гораздо сложнее сложения. Пусть **a** и **b** – два n-мерных вектора (столбца). Их *скалярное произведение* $\mathbf{a}^{\mathsf{T}}\mathbf{b}$ (в других обозначениях – $\mathbf{a} \cdot \mathbf{b}$ или $(\mathbf{a},\mathbf{b})$) – определяется как сумма их поэлементных произведений:

$$\mathbf{a}^{\mathsf{T}}\mathbf{b} = \sum_{i=1}^{n} a_i b_i. \tag{3.14}$$

n-мерные векторы (то есть определённые в пространстве R_n) имеют также геометрическую интерпретацию. Можно показать, что

$$\mathbf{a}^{\mathrm{T}}\mathbf{b} = |\,\mathbf{a}\,| \cdot |\,\mathbf{b}\,| \cdot \cos(\alpha), \tag{3.15}$$

где α – угол между векторами **a** и **b**, а $|\cdot|$ обозначает модуль (евклидову норму, абсолютную величину, длину) вектора:

$$|\,\mathbf{a}\,| = \sqrt{a_1^2 + \ldots + a_n^2}. \tag{3.16}$$

Если **A** и **B** – матрицы размерностей $n \times m$ и $m \times p$ соответственно, то их произведение $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ – матрица размерности $n \times p$, элементы которой определяются как

$$c_{i,j} = a_{i,1} \cdot b_{1,j} + a_{i,2} \cdot b_{2,j} + \ldots + a_{i,m} \cdot b_{m,i} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}.$$

При этом число столбцов $\mathbf{A}\cdot(m)$ должно совпадать с числом строк **B**. Элемент $c_{i,j}$ равен скалярному произведению i-й строки **A** и j-го столбца **B** (которые можно считать векторами). Например, результатом произведения

$$\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A} = \begin{pmatrix} 3 & 1 \\ 4 & 8 \\ 2 & 5 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 & 2 \\ 1 & 8 & 5 \end{pmatrix} = \begin{pmatrix} 10 & 20 & 11 \\ 20 & 80 & 48 \\ 11 & 48 & 29 \end{pmatrix}$$

является *симметричная* матрица, так как она не меняется при транспонировании.

Векторы можно также считать точками. Геометрически сложение двух векторов **u** и **v** можно понимать как новый вектор, конечная точка которого – результат «соединения» **u** и **v**, как показано на рис. 3.3(a). Отсюда следует, что вектор, который начинается в конечной точке **u** и заканчивается в конечной точке **v**, есть вектор **v** – **u**, как показано на рис. 3.3(b).

Рис. 3.3. Геометрическая интерпретация сложения и вычитания векторов

Наконец, двумерный вектор можно повернуть против часовой стрелки на α градусов (или радиан) путём умножения его на матрицу «поворота»:

$$\mathbf{R} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}, \tag{3.17}$$

как показано на рис. 3.4, где **u** – вектор-столбец (как обычно принимается в математической литературе).

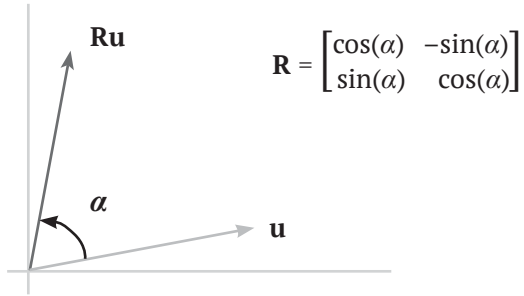


Рис. 3.4. Матрица поворота (против часовой стрелки)

3.2. Временна́я сложность вычислений

Временна́я сложность алгоритма – это теоретическая оценка времени или количества операций, необходимых для решения задачи. (В листинге 3.1 приведён простой способ оценки времени выполнения кода с помощью модуля `time` языка Python.)

Листинг 3.1. Вычисление времени выполнения кода с помощью модуля time

```
import time

t = time.time()
# execute some code here
elapsed_time = time.time() - t
print (elapsed_time)
```

Временна́я сложность определяется некоторой функцией T, зависящей от исходного размера задачи и подсчитывающей количество операций для выполнения конкретной задачи. В информатике эффективность алгоритмов обычно определяется поведением функции T при достаточно больших размерах задачи. Более того, ключевым фактором становится скорость роста функции T при стремлении размера задачи

к бесконечности. В последующих подразделах рассматриваются именно эти темы, а также математический аппарат, используемый обычно для оценки временно́й сложности вычислений. Несмотря на то что временна́я сложность алгоритмов решения задач может зависеть от многих факторов, в этой книге мы ограничимся её зависимостью лишь от одного фактора – размера задачи.

3.2.1. Порядок роста функций

Функция $T(n)$ оценки времени выполнения алгоритма может содержать несколько слагаемых, которые по-разному влияют на её значение для заданного размера задачи n. Пусть, например, $T(n) = 0{,}5n^2 + 2000n + 50\,000$. При небольших значениях n все слагаемые почти в равной степени влияют на рост $T(n)$. Однако с увеличением n первое слагаемое $(0{,}5n^2)$ начинает влиять на рост функции значительно больше, чем два других (даже вместе взятых). Следовательно, *порядок роста* функции характеризуется её старшим членом. На рис. 3.5 приведены отдельные графики для $0{,}5n^2$ и $2000n + 50\,000$. Для больших значений n очевидно, что первое слагаемое доминирует над двумя другими и, следовательно, характеризует скорость роста $T(n)$. Коэффициенты полинома были выбраны так, чтобы показать, что сами они не оказывают существенного влияния на скорость роста функции.

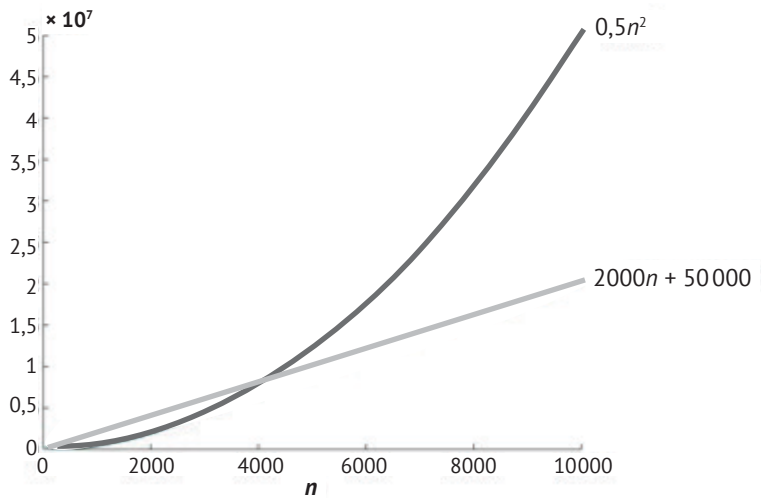


Рис. 3.5. Порядок роста функции определяется её старшим членом. Для $T(n) = 0{,}5n^2 + 2000n + 50\,000$ порядок роста – квадратичный, поскольку для больших значений n её старший член $0{,}5n^2$, очевидно, доминирует над младшими (даже взятыми вместе)

На рис. 3.6 приведены графики наиболее распространённых функций, характеризующих порядок роста вычислительной сложности.

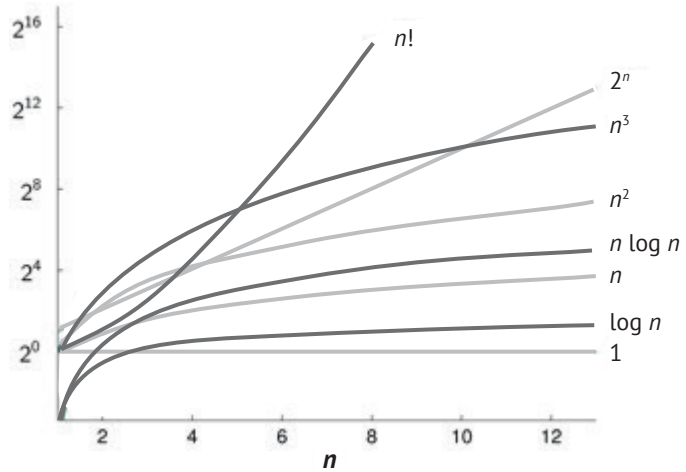


Рис. 3.6. Типичные порядки роста вычислительной сложности

Для достаточно больших значений n их можно упорядочить следующим образом:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!,$$

где условно считается, что $f(n) < g(n)$, если $\lim_{n\to\infty} f(n)/g(n) = 0$.

Приведённые выше типичные порядки роста называются (слева направо) константными, логарифмическими, линейными, линейно-логарифмическими, квадратичными, кубическими, показательными и факториальными.

Поскольку масштаб оси Y на рис. 3.6 – логарифмический, разница между порядками роста может показаться несколько меньше, чем есть на самом деле (каждое следующее деление означает 16-кратный рост времени выполнения алгоритма). В табл. 3.1 приведены фактические значения этих функций, где явно выделяется быстрый рост показательной и факториальной функций.

Задачи, которые не могут быть решены за полиномиальное время, обычно считаются трудноразрешимыми (трудными), так как время их выполнения довольно велико даже при умеренных размерах. Напротив, задачи, которые могут быть решены за полиномиальное время, считаются легкоразрешимыми (лёгкими). Однако граница между лёгкими и трудными задачами довольно условна. Если время выполнения алгоритма оценивается полиномиальной функцией высокой степени, то на

практике может потребоваться довольно много времени для достижения её полного решения или получения промежуточных результатов.

Таблица 3.1. Фактические значения типичных функций оценки вычислительной сложности

1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	0	1	0	1	1	2	1
1	1	2	2	4	8	4	2
1	2	4	8	16	64	16	24
1	3	8	24	64	512	256	40 320
1	4	16	64	256	4096	65 536	$2.09 \cdot 10^{13}$
1	5	32	160	1024	32 768	4 295 967 296	$2.63 \cdot 10^{35}$

3.2.2. Асимптотические обозначения

Важной особенностью порядка роста является то, что при его определении обычно пренебрегают постоянными множителями. Подобно слагаемым низших порядков, они опускаются, поскольку с ростом размера задачи их влияние на вычислительную эффективность становится гораздо меньше фактического порядка роста. Более того, нет смысла и в точном определении эффективности алгоритма, поскольку оно зависит от множества факторов, включая аппаратные возможности компьютера, язык программирования, компилятор, интерпретатор и т. д.

Таким образом, на практике достаточно предположить, что для выполнения элементарных машинных команд требуется постоянное время, не имеющее существенного значения.

Теоретический анализ сложности алгоритмов и задач основан на так называемых «асимптотических обозначениях», которые при больших размерах задач позволяют отбрасывать слагаемые низших порядков и постоянные множители. В частности, асимптотические обозначения позволяют определить множества, которые можно использовать для определения «асимптотических границ».

Обозначение «O» определяется следующим множеством:

$$O(g(n)) = \left\{ f(n) : \exists c > 0 \text{ и } n_0 > 0 \mid 0 \leqslant f(n) \leqslant c \cdot g(n), \forall n \geqslant n_0 \right\}.$$

Если функция $f(n)$ принадлежит этому множеству, то $g(n)$ – *асимптотическая верхняя граница* $f(n)$. Это означает, что $g(n)$ будет больше $f(n)$, но согласно определению это справедливо только на интервале от некоторого $n_0 > 0$ до бесконечности, причём $g(n)$ может быть умножена на достаточно большую положительную константу c. На рис. 3.7(а) приведена

графическая иллюстрация этого определения. Если $g(n)$ – асимптотическая верхняя граница $f(n)$, то должна существовать возможность найти такую положительную константу c, что $c \cdot g(n) \geqslant f(n)$, но только для n, не меньшего некоторого положительного значения n_0 (то, что происходит при $n < n_0$, не имеет значения). Чтобы доказать, что функция принадлежит $O(g(n))$, достаточно найти хотя бы одну пару (поскольку она не одна) констант c и n_0, отвечающих данному определению. Например, если определение верно для некоторой пары c и n_0, то оно также верно для бóльших значений c и n_0. В связи с этим нет необходимости искать самые маленькие значения c и n_0, отвечающие определению O (то же справедливо для всех обозначений, приводимых ниже).

Эффективность алгоритмов чаще всего определяется так называемым *худшим случаем*, когда среди задач одинакового размера выбирается та, что может потребовать наибольшего количества ресурсов (времени, памяти и т. д.). Поскольку обозначение «O» определяет асимптотическую верхнюю границу, оно гарантирует, что конкретному алгоритму даже при больших размерах задачи потребуется в самом худшем случае не более определенного количества ресурсов. Например, время выполнения алгоритма quicksort, сортирующего список или массив из n элементов (см. раздел 6.2.2), вообще говоря, оценивается как $O(n^2)$, поскольку в худшем случае он требует порядка n^2 сравнений. Однако в лучших и умеренных случаях quicksort может работать быстрее (за время порядка $n \log n$).

Напротив, обозначение «Ω» определяет *асимптотическую нижнюю границу*:

$$\Omega(g(n)) = \left\{f(n): \exists c > 0 \text{ и } n_0 > 0 \mid 0 \leqslant c \cdot g(n) \leqslant f(n), \forall n \geqslant n_0\right\}.$$

На рис. 3.7(a) приведена графическая иллюстрация этого определения. Обозначение «Ω» полезно для определения нижней границы ресурсов, необходимых для решения задачи, независимо от применяемого алгоритма. Например, можно теоретически доказать, что любой алгоритм сортировки списка из n вещественных чисел требует в худшем случае $\Omega(n \log n)$ сравнений.

Наконец, обозначение «Θ» определяет *асимптотические жёсткие границы*:

$$\Theta(g(n)) = \left\{f(n): \exists c_1 > 0, c_2 > 0 \text{ и } n_0 > 0 \mid 0 \leqslant c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n), \forall n \geqslant n_0\right\}.$$

Если $f(n) \in \Theta(g(n))$, то $f(n)$ и $g(n)$ имеют примерно одинаковый порядок роста. Таким образом, $g(n)$ при определённых значениях констант c_1 и c_2 будет для $f(n)$ и верхней, и нижней асимптотической границей, как показано на рис. 3.7(c). Иными словами, $f(n) \in O(g(n))$ и $f(n) \in \Omega(g(n))$.

Например, алгоритм сортировки слиянием для списка или массива из n элементов всегда требует (в лучшем и худшем случаях) порядка $n \log n$ сравнений. Поэтому мы говорим, что его время выполнения – $\Theta(n \log n)$.

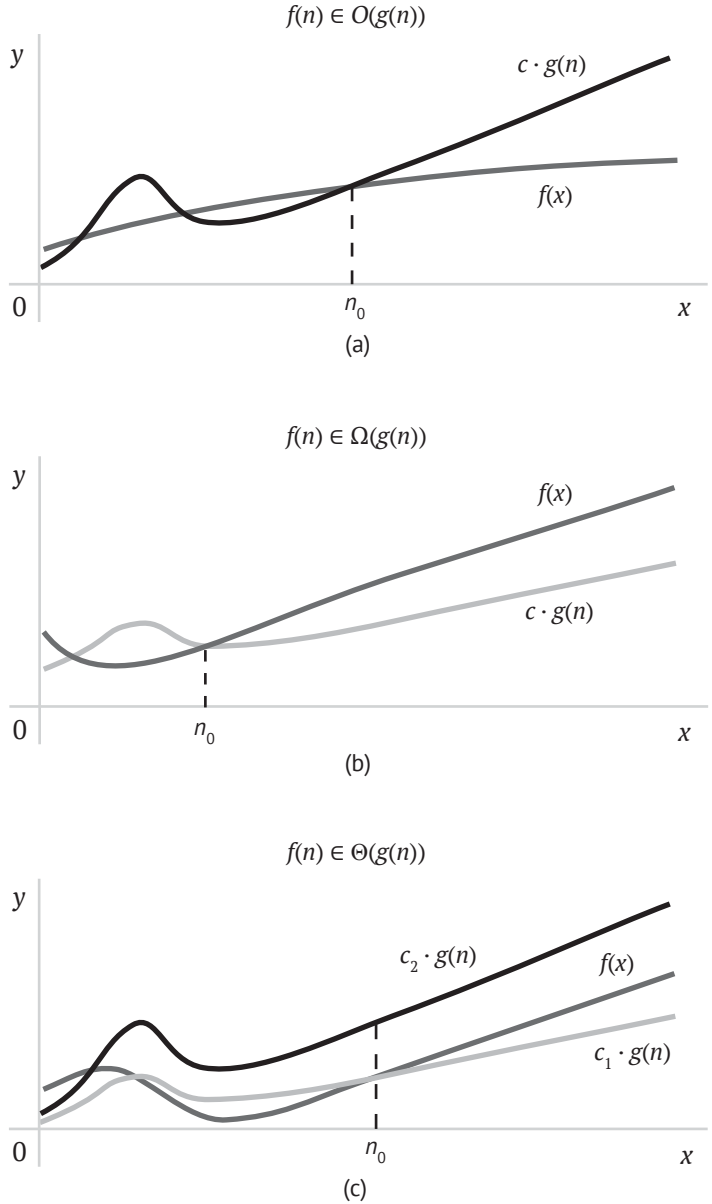


Рис. 3.7. Графическая иллюстрация асимптотических обозначений вычислительной сложности

При определении асимптотических границ константами и слагаемыми низших порядков в функции $g(n)$ пренебрегают. Например, несмотря на очевидность того, что $3n^2 + 10n \in O(5n^2 + 20n)$, достаточно сказать, что $3n^2 + 10n \in O(n^2)$, поскольку для асимптотических обозначений важен лишь порядок роста. В связи с этим читатель, видимо, обратил внимание на отсутствие основания логарифма в обозначении порядка роста функции. Причина тому – то, что логарифмы с разными основаниями отличаются только постоянным множителем. Пусть ρ представляет собой некоторую асимптотическую границу, а a и b – два разных основания логарифмов. Следующие равенства показывают, что все логарифмы независимо от их основания имеют один и тот же порядок:

$$\rho \log_a g(n) = \rho \log_b g(n)/\log_b a = \rho \cdot \overbrace{1/\log_b a}^{\text{константа}} \log_b g(n) = \rho \log_b g(n).$$

Таким образом, в обозначении порядка роста основание логарифма не нужно.

Наконец, при определении порядка роста функции можно использовать пределы благодаря следующим равносильным утверждениям, связывающим их с определениями асимптотических обозначений границ:

$$f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} f(n)/g(n) < \infty \text{ (константа или ноль),}$$

$$f(n) \in \Omega(g(n)) \Leftrightarrow \lim_{n\to\infty} f(n)/g(n) > 0 \text{ (константа > 0 или } \infty),$$

$$f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} f(n)/g(n) = \text{константа} > 0.$$

3.3. Рекуррентные соотношения

Время выполнения или количество операций, выполняемых рекурсивным алгоритмом, определяется рекуррентными соотношениями (или просто рекуррентностью), то есть некоторой математической рекурсивной функцией T, определяющей стоимость вычисления этого алгоритма. Рассмотрим код из листинга 1.1, который суммирует первые n положительных целых чисел. Во-первых, число операций, которые он должен выполнить, очевидно, зависит от входного параметра n. Следовательно, T будет функцией n.

В начальном условии (при $n = 1$) метод выполняет основные действия, приведённые на рис. 3.8.

Перед их выполнением программа должна сохранить низкоуровневую информацию (например, параметры или адрес возврата). Считаем, что это требует a_0 единиц времени, где a_0 – просто константа. Следующее действие – проверка условия со временем выполнения a_1. Поскольку результат – True, следующее действие – это переход к третьей строке метода, который требует a_2 единиц времени. Наконец, на последнем шаге метод возвращает значение 1, требуя a_3 единиц времени. Итого при $n = 1$ методу нужно $a = a_0 + a_1 + a_2 + a_3$ единиц времени. Таким образом, можно считать, что $T(1) = a$. Точное значение a для асимптотической сложности вычисления метода безразлично. Важно только то, что a – постоянная величина, не зависящая от n.

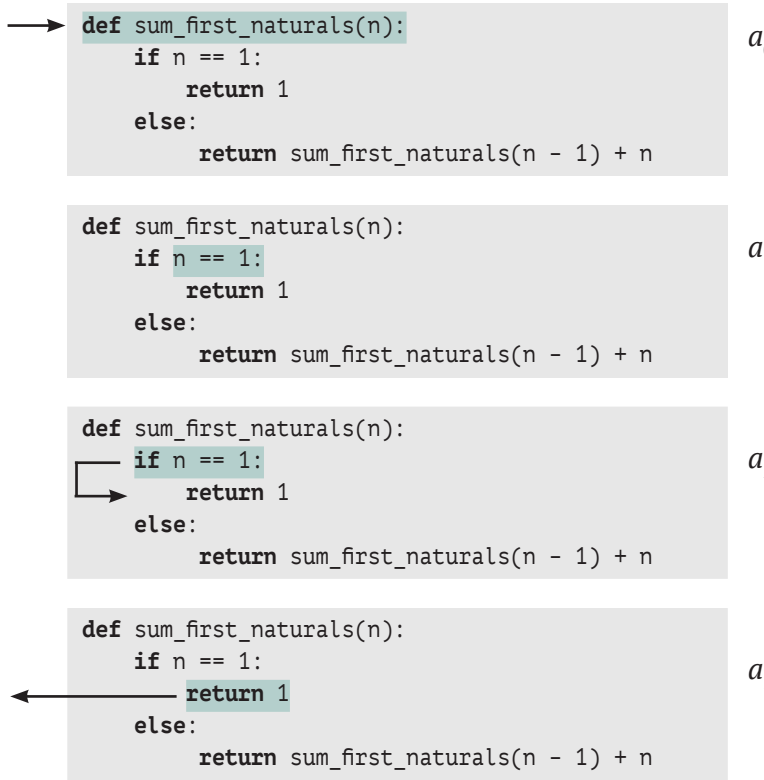


Рис. 3.8. Последовательность действий, выполняемых функцией из листинга 1.1 при начальном условии

Действия при выполнении рекурсивного условия (когда $n > 1$) приведены на рис. 3.9.

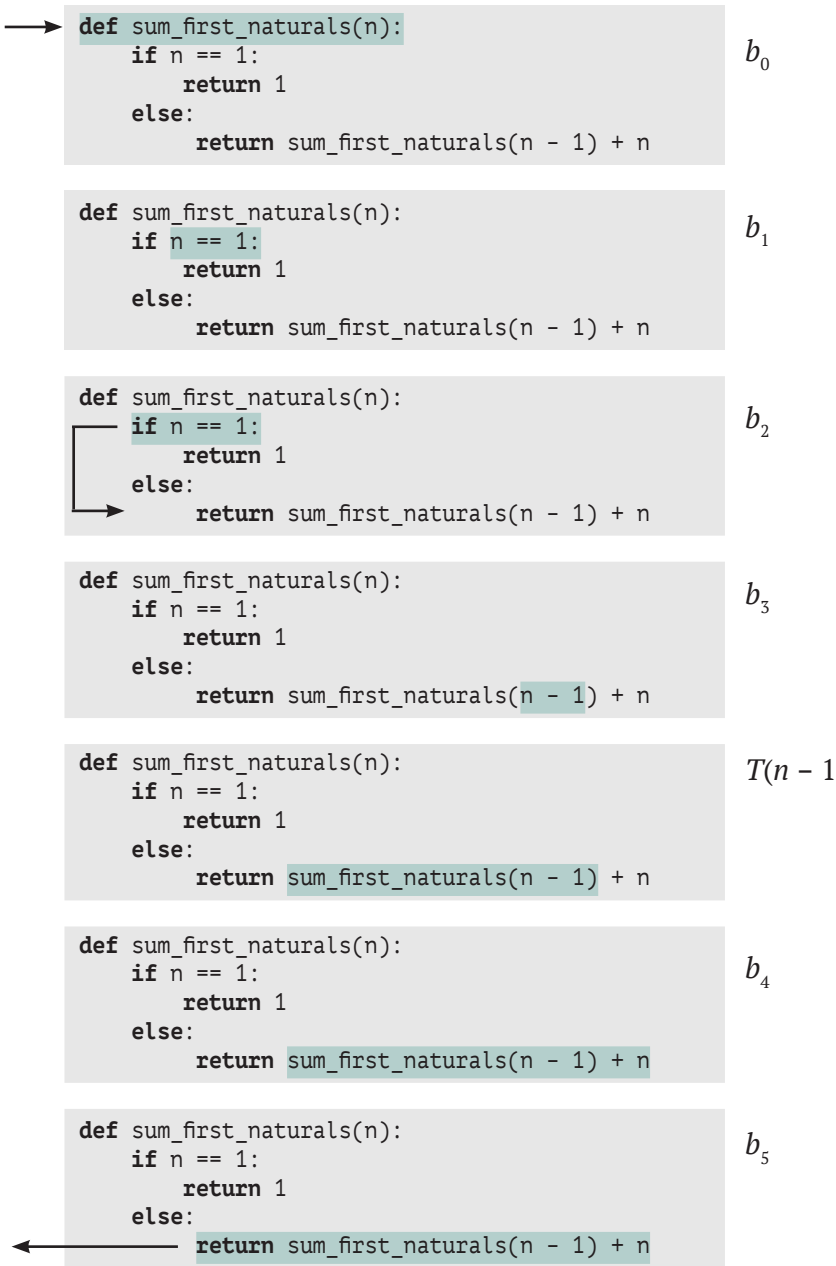


Рис. 3.9. Последовательность действий, выполняемых функцией из листинга 1.1 при рекурсивном условии

Пусть $b = \sum_{i=0}^{5} b_i$ – общее время выполнения основных действий (сохранение низкоуровневой информации, проверка условия, переход к рекурсивному условию, вычитание единицы из n, добавление n к результату рекурсивного вызова и возвращение результата), также являющееся константой, точное значение которой не важно для асимптотической сложности вычисления. Но, кроме b, необходимо оценить время на рекурсивный вызов. Поскольку он решает полную задачу размера $n - 1$, можно определить его как $T(n - 1)$, а всё рекуррентное соотношение $T(n)$ можно определить следующим образом:

$$T(n) = \begin{cases} a, & \text{если } n = 1, \\ T(n-1) + b, & \text{если } n > 1, \end{cases} \tag{3.18}$$

где, например, $T(3) = T(2) + b = T(1) + b + b = a + 2b$.

Хотя T правильно описывает стоимость вычисления алгоритма, выяснить порядок её роста непросто из-за её рекурсивности. Поэтому следующим шагом анализа будет преобразование рекурсивной функции в эквивалентную нерекурсивную формулу. Этот процесс называют «решением» рекуррентного соотношения. В данном случае нетрудно видеть, что $T(n)$ – линейная функция n:

$$T(n) = b(n - 1) + a = b \cdot n - b + a \in \Theta(n).$$

Следующие разделы посвящены общим методам решения рекуррентных соотношений.

Кроме того, в этом вводном курсе мы будем упрощать рекуррентные соотношения ради облегчения их анализа. Рассмотрим, например, код из листинга 2.9. Связанная с ним функция стоимости выполнения может быть определена как:

$$T(n) = \begin{cases} a, & \text{если } n = 1, \\ T(n/2) + b, & \text{если } n > 1 \text{ и } n \text{ – чётное}, \\ T(n/2) + c, & \text{если } n > 1 \text{ и } n \text{ – нечётное}. \end{cases} \tag{3.19}$$

Это рекуррентное соотношение является сложным для анализа по двум причинам. С одной стороны, оно содержит более одного рекурсивного условия. С другой – здесь не исключена работа с функцией *floor*, связанной с техническими особенностями её реализации и требующей более сложной математики (например, неравенств). Более того, для определения порядка роста этой функции нет нужды усложнять её про-

веркой чётности n и прочими лишними деталями. Поскольку алгоритм опирается на решения подзадач (примерно – в случае нечётного n) половинного размера, вместо (3.19) можно работать со следующим рекуррентным соотношением:

$$T(n) = \begin{cases} a, & \text{если } n = 1, \\ T(n/2) + b, & \text{если } n > 1. \end{cases} \tag{3.20}$$

Кроме того, в (3.20), используемой взамен (3.19), мы также предположим, что размер задачи (n) будет достаточно высокой степенью двух.

Таким образом, в этой книге мы будем рассматривать рекуррентные соотношения только с одним рекурсивным условием, которое не содержит функций *floor* или *ceiling*. Это позволит нам находить точные нерекурсивные формулы рекуррентных соотношений, чей порядок роста можно оценивать жёсткими асимптотическими границами Θ.

3.3.1. Метод расширения

Метод расширения (итерации или обратной подстановки) применяется прежде всего для решения рекуррентных соотношений, рекурсивное условие которых содержит одну ссылку на рекурсивную функцию (в отдельных случаях он может быть применён к рекуррентным соотношениям с несколькими ссылками на рекурсивную функцию). Его идея заключается в последовательном упрощении рекуррентного соотношения вплоть до выявления на некотором шаге i общей закономерности. После чего в функцию подставляются конкретные значения в предположении, что начальное условие достигнуто на этом шаге i. Приводимые ниже примеры иллюстрируют эту процедуру, а в разделе 10.2.1 рассматривается соответствующий визуальный подход, называемый «методом дерева».

3.3.1.1. Общие рекуррентные соотношения

Рассмотрим функцию (3.18). Её рекурсивное условие

$$T(n) = T(n - 1) + b \tag{3.21}$$

можно неоднократно применять (к аргументам с меньшими значениями) для «расширения» слагаемого T в правой части. Например, $T(n - 1) = T(n - 2) + b$ – это результат подстановки $(n - 1)$ вместо n в (3.21). Таким образом, мы приходим к соотношению

$$T(n) = [T(n - 2) + b] + b = T(n - 2) + 2b,$$

где выражение в квадратных скобках – это расширение $T(n-1)$. Этот приём можно применить снова, расширив $T(n-2)$ до $T(n-3)+b$. Таким образом, на третьем шаге мы получим:

$$T(n) = [T(n-3)+b] + 2b = T(n-3) + 3b.$$

После нескольких таких расширений у нас должна появиться возможность выявить закономерность и вывести общую формулу, соответствующую некоторому шагу i. Общая формула для этой функции:

$$T(n) = T(n-i) + i \cdot b. \tag{3.22}$$

В конце концов, при некотором значении i процесс достигнет начального условия. Для функции (3.18) оно определяется как $T(1)$. Следовательно, слагаемое $T(n-i)$ достигнет начального условия при $n-i=1$ или при $i=n-1$. Подстановка этого значения в (3.22) позволяет нам избавиться от переменной i в этой формуле и получить полностью нерекурсивное определение $T(n)$:

$$T(n) = T(1) + (n-1)\,b = a + (n-1)\,b = b \cdot n - b + a \in \Theta(n).$$

Таким образом, код из листинга 1.1 выполняется за линейное от n время.

На рис. 3.10 приведена справка о методе расширения, который мы теперь будем применять к другим рекуррентным соотношениям.

1. Выписать рекурсивное условие из рекуррентного соотношения.
2. Несколько раз расширить рекурсивный член в правой части, пока не удастся выявить общую формулу для произвольного шага i.
3. Определить значение i, при котором выполняется начальное условие.
4. Подставить полученное значение i в общую формулу.

Рис. 3.10. Метод расширения (краткая справка)

Рассмотрим процесс расширения функции (3.20):

$$\begin{aligned} T(n) &= T(n/2) + b && \{\text{шаг 1}\} \\ &= T(n/4) + b + b = T(n/4) + 2b && \{\text{шаг 2}\} \\ &= T(n/8) + b + 2b = T(n/8) + 3b && \{\text{шаг 3}\} \\ &= T(n/16) + b + 3b = T(n/16) + 4b && \{\text{шаг 4}\} \ldots \end{aligned}$$

Её общая формула для рекуррентного соотношения на шаге i:

$$T(n) = \mathrm{T}(n/2^i) + i \cdot b. \tag{3.23}$$

Начальное условие $T(1)$ выполняется при $n/2^i = 1$, то есть при $n = 2^i$ или при $i = \log_2 n$. Подставив полученное значение i в (3.23), получаем:

$$T(n) = T(1) + b \log_2 n = a + b \log_2 n \in \Theta(\log n).$$

Поскольку её порядок роста – логарифмический, код листинга 1.1 с линейным порядком роста асимптотически медленнее кода листинга 2.9. Интуитивно это понятно: первый уменьшает размер задачи на 1, тогда как второй делит размер исходной задачи пополам и за счёт этого приходит к начальному условию за меньшее количество рекурсивных вызовов функции.

Тонкость рекуррентных соотношений при делении размера задачи на целочисленную константу $k \geqslant 2$ заключается в том, что в них не должно быть начального условия для $n = 0$. Математически оно никогда не достижимо, и мы не сможем получить значение i на шаге 3 метода расширения. Ловушка кроется в том, что аргументом функции $T(n)$ должно быть только целое число, поэтому дробь n/k на самом деле предполагает целочисленное деление. А это значит, что следующим за $T(1)$ расширением будет не $T(1/k)$, а $T(0)$. Поэтому для правильного применения метода расширения в рекуррентные соотношения нужно включать дополнительное начальное условие, чаще всего для $n = 1$.

В предыдущих примерах было довольно просто в процессе расширения обнаружить общую формулу на произвольном шаге i. Для следующих рекуррентных соотношений это сделать несколько сложнее, так как они содержат вычисление сумм, о которых шла речь в разделе 3.1.4.

Рассмотрим следующее рекуррентное соотношение:

$$T(n) = \begin{cases} a, & \text{если } n = 0, \\ T(n-1) + b \cdot n + c, & \text{если } n > 0. \end{cases} \tag{3.24}$$

Процесс его расширения:

$$\begin{aligned} T(n) &= T(n-1) + b \cdot n + c = [T(n-2) + b(n-1) + c] + b \cdot n + c \\ &= T(n-2) + 2b \cdot n - n + 2c = [T(n-3) + b(n-2) + c] + 2b \cdot n - b + 2c \\ &= T(n-3) + 3b \cdot n - b(1+2) + 3c = [T(n-4) + b(n-3) + c] + 3b \cdot n - b(1+2) + 3c \\ &= T(n-4) + 4b \cdot n - b(1+2+3) + 4c \\ &\quad \ldots \\ &= T(n-i) + i \cdot b \cdot n - b(1+2+3+\ldots+(i-1)) + i \cdot c, \end{aligned}$$

где квадратными скобками выделены расширения тех членов, которые включают функцию T. Последний шаг содержит общую формулу, которую можно записать как

$$T(n) = T(n-i) + i \cdot b \cdot n - b\sum_{j=1}^{i-1} j + i \cdot c = T(n-i) + i \cdot b \cdot n - b(i-1)\, i/2 + i \cdot c. \quad (3.25)$$

Два распространённых заблуждения при использовании сумм в формулах на шаге i: (1) использование i в качестве индексной переменной суммы и (2) выбор n в качестве её верхнего предела. Важно отметить, что верхним пределом суммы является $i - 1$, откуда следует, что i не может быть индексной переменной суммы.

Наконец, начальное условие $T(0)$ достигается при $i = n$, поэтому подстановка его значения в (3.25) даёт формулу

$$T(n) = b \cdot n^2 - b \cdot n(n-1)/2 + c \cdot n + a = b \cdot n^2/2 + (c + b/2)/n + a \in \Theta(n^2),$$

которая является полиномом 2-й степени.

Следующее рекуррентное соотношение появляется в алгоритмах типа «разделяй и властвуй», таких, например, как сортировка слиянием (см. главу 6):

$$T(n) = \begin{cases} a, & \text{если } n = 1, \\ 2T(n/2) + b \cdot n + c, & \text{если } n > 1. \end{cases} \quad (3.26)$$

Процесс его расширения:

$$\begin{aligned} T(n) &= 2T(n/2) + b \cdot n + c = 2[2T(n/4) + b \cdot n/2 + c] + b \cdot n + c \\ &= 4T(n/4) + 2b \cdot n + 2c + c = 4[2T(n/8) + b \cdot n/4 + c] + 2b \cdot n + 2c + c \\ &= 8T(n/8) + 3b \cdot n + 4c + 2c + c = 8[2T(n/16) + b \cdot n/8 + c] + 3b \cdot n + 4c + +2c + c \\ &= 16T(n/16) + 4b \cdot n + 8c + 4c + 2c + c \\ &\quad \ldots \\ &= 2^i T(n/2^i) + i \cdot b \cdot n + c(1 + 2 + 4 + \ldots + 2^{i-1}). \end{aligned}$$

И здесь в квадратных скобках – расширения членов, содержащих T. В этом случае скобки особенно полезны, поскольку каждый из расширяемых членов должен умножаться на 2. Студентам всегда стоит использовать их, так как пренебрежение ими может приводить к многочисленным ошибкам.

В данном случае общая формула содержит частичную сумму геометрической прогрессии

$$T(n) = 2^i T(n/2^i) + i \cdot b \cdot n + c\sum_{j=0}^{i-1} 2^j = 2^i T(n/2^i) + i \cdot b \cdot n + c(2^i - 1). \qquad (3.27)$$

Начальное условие $T(1)$ достигается при $n/2^i = 1$, откуда следует, что $i = \log_2 n$. Поэтому подстановка i в (3.27) даёт:

$$T(n) = nT(1) + b \cdot n \log_2 n + c(n-1) = b \cdot n \log_2 n + n(a + c) - c \in \Theta(n \log n),$$

где старший член – $b \cdot n \log_2 n$.

3.3.1.2. *Основная теорема*

Основная теорема (master theorem) может использоваться в качестве быстрого способа определения временно́й сложности вычисления алгоритмов, основанных на стратегии «разделяй и властвуй». В частности, её можно применить к рекуррентным соотношениям следующего вида:

$$T(n) = \begin{cases} c, & \text{если } n = 1, \\ aT(n/b) + f(n), & \text{если } n > 1, \end{cases}$$

где $a \geqslant 1$, $b > 1$, $c \geqslant 0$ и f – асимптотически положительная функция. Подобные алгоритмы обычно относятся к тем задачам, исходный размер которых делится на некоторое положительное число b, не меньшее двух. Последующая обработка или объединение решений подзадач требуют $f(n)$ операций. В зависимости от вклада слагаемых $aT(n/b)$ и $f(n)$ в стоимость выполнения алгоритма основная теорема в её самой общей форме утверждает, что можно определить асимптотическую жёсткую границу для T в следующих трех случаях:

1) если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторой константы $\varepsilon > 0$,
   то $T(n) \in \Theta(n^{\log_b a})$;
2) если $f(n) = \Theta(n^{\log_b a} (\log n)^k)$, где $k \geqslant 0$,
   то $T(n) \in \Theta(n^{\log_b a} (\log n)^{k+1})$;
3) если $f(n) = \Omega(n^{\log_b a + \varepsilon})$, где $\varepsilon > 0$ и $f(n)$ удовлетворяет так называемому условию регулярности $a \cdot f(n/b) \leqslant d \cdot f(n)$ для некоторой константы $d < 1$ и для всех достаточно больших n,
   то $T(n) \in \Theta(f(n))$.

Например, для

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ T(n/2) + 2^n, & \text{если } n > 1, \end{cases}$$

можно найти такое $\varepsilon > 0$, что порядок $f(n) = 2^n$ будет больше $n^{(\log_2 1)+\varepsilon} = n^{\varepsilon}$. Действительно, в этом примере допустимо любое значение $\varepsilon > 0$, поскольку порядок показательной функции 2^n всегда выше порядка полиномиальной. Таким образом, это рекуррентное соотношение относится к третьему случаю основной теоремы, который подразумевает, что $T(n) \in \Theta(2^n)$.

Если $f(n)$ – полином степени k, то можно воспользоваться следующей более простой версией основной теоремы:

$$T(n) \in \begin{cases} \Theta(n^k), & \text{если } a \cdot b^k < 1, \\ \Theta(n^k \log n), & \text{если } a \cdot b^k = 1, \\ \Theta(n \log_b a), & \text{если } a \cdot b^k > 1, \end{cases} \tag{3.28}$$

Этот результат может быть получен методом расширения. Рассмотрим следующее рекуррентное соотношение:

$$T(n) = \begin{cases} c, & \text{если } n = 1, \\ aT(n/b) + d \cdot n^k, & \text{если } n > 1, \end{cases} \tag{3.29}$$

где $a \geqslant 1$, $b > 1$, $c \geqslant 0$ и $d \geqslant 0$.

Процесс его расширения:

$$\begin{aligned} T(n) &= aT(n/b) + dn^k = a[aT(n/b^2) + d(n/b)^k] + dn^k \\ &= a^2T(n/b^2) + dn^k(1 + a/b^k) = a^2[aT(n/b^3) + d(n/b^2)k] + dn^k(1 + a/b^k) \\ &= a^3T(n/b^3) + dn^k(1 + a/b^k + a^2/b^{2k}) \\ &\quad \cdots \\ &= a^iT(n/b^i) + dn^k \sum_{j=0}^{i-1} (a/b^k)^j. \end{aligned}$$

Начальное условие $T(1) = c$ достигается при $i = \log_b n$, а подстановка этого значения в формулу даёт:

$$T(n) = c \cdot a^{\log_b n} + d \cdot n^k \sum_{j=0}^{\log_b n - 1} (a / b^k)^j = c \cdot n^{\log_b a} + d \cdot n^k \sum_{j=0}^{\log_b n - 1} (a/b^k)^j,$$

где в зависимости от значений a, b и k возможны три случая.

1. Если $a < b^k$, то $\sum_{j=0}^{\log_b n - 1} (a/b^k)^j$ будет постоянной величиной (которая не может быть бесконечностью). Отметим, что бесконечная сум-

ма $\sum_{i=0}^{\infty} r^i = 1/(r-1)$ – константа при $r < 1$ (то есть не расходится). Таким образом, в этом случае

$$T(n) = c \cdot n^{\log_b a} + dKn^k$$

для некоторой постоянной K. Кроме того, поскольку $a < b^k$, то есть $\log_b a < k$, то n^k – член высшего порядка. Следовательно:

$$T(n) \in \Theta(n^k).$$

2. Если $a = b^k$, то

$$T(n) = cn^k + dn^k \sum_{j=0}^{\log_b n - 1} 1 = cn^k + dn^k \log_b n,$$

из чего следует, что

$$T(n) \in \Theta(n^k \log n).$$

3. Если $a > b^k$, то, вычислив сумму геометрической прогрессии, получим:

$$\begin{aligned} T(n) &= c \cdot n^{\log_b a} + dn^k((a/b^k)^{\log_b n} - 1))/(a/b^k - 1) \\ &= c \cdot n^{\log_b a} + dn^k(n^{\log_b a}/n^k - 1)\, K \\ &= (c + dK)\, n^{\log_b a} - dKn^k, \end{aligned}$$

где $K = 1/(a/b^k - 1)$ – константа. Наконец, поскольку $a > b^k$, то есть $\log_b a > k$, мы имеем:

$$T(n) \in \Theta(n^{\log_b a}).$$

3.3.1.3. *Дополнительные примеры*

Рекуррентное соотношение для времени выполнения кодов из листингов 1.2 и 2.4 могло бы быть таким:

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2T(n/2) + 1, & \text{если } n > 1, \end{cases} \tag{3.30}$$

в предположении, что постоянные множители равны 1 и размер подзадачи равен половине размера исходной задачи. $T(n/2)$ умножается на 2, потому что в рекурсивных условиях алгоритм вызывает себя дважды с разными аргументами. Кроме того, к этому произведению добавляется константа 1, так как результаты подзадач должны быть сложены перед завершением метода. Наконец, ради простоты мы пренебрежём

начальным условием для $n = 2$. Это допущение не повлияет на порядок роста функции времени выполнения.

Рекуррентное соотношение можно решить методом расширения следующим образом:

$$\begin{aligned} T(n) &= 2\,T(n/2) + 1 = 2\,[2T(n/4) + 1] + 1 \\ &= 4\,T(n/4) + 2 + 1 = 4\,[2T(n/8) + 1] + 2 + 1 \\ &= 8\,T(n/8) + 4 + 2 + 1 = 8[2T(n/16) + 1] + 4 + 2 + 1 \\ &= 16\,T(n/16) + 8 + 4 + 2 + 1 \\ &\quad \ldots \\ &= 2^i\,T(n/2^i) + \sum_{j=0}^{i-1} 2^j = 2^iT(n/2^i) + 2^i - 1. \end{aligned}$$

Начальное условие $T(1) = 1$ выполняется при $n = 2^i$. Подстановка даёт:

$$T(n) = n + n - 1 = 2n - 1 \in \Theta(n),$$

где $T(n)$ – линейная функция n. Естественно, этот результат соответствует основной теореме (см. (3.28)), так как рекуррентное соотношение – особый случай (3.29), где $a = 2$, $b = 2$ и $k = 0$. Поэтому $T(n) \in \Theta(2^{\log_2 n}) = \Theta(n)$.

Рассмотрим функцию из листинга 1.5. Первые два метода уменьшают размер задачи на 1. Таким образом, соответствующее им рекуррентное соотношение могло бы быть таким:

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ T(n-1) + 1, & \text{если } n > 1, \end{cases}$$

что является особым случаем (3.21), где $T(n) \in \Theta(n)$. Третий метод делит размер исходной задачи пополам, и решения подзадач не нуждаются в последующей обработке. Поэтому его рекуррентное соотношение совпадает с (3.30).

В заключение мы рассмотрим временную сложность вычислений кодов из листингов 2.2 и 2.3. Рекуррентное соотношение для них (пренебрегая постоянными множителями):

$$T(n) = \begin{cases} 1, & \text{если } n < 10, \\ T(n/10) + 1, & \text{если } n \geq 10. \end{cases} \tag{3.31}$$

Оно отличается от предыдущих рекуррентных соотношений тем, что начальное условие определено на интервале. Тем не менее, предположив, что входным параметром метода будет степень 10, можно использовать альтернативное определение рекуррентного соотношения:

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ T(n/10) + 1, & \text{если } n > 1. \end{cases} \tag{3.32}$$

Обе функции равнозначны при $n = 10^p$ и $p \geqslant 0$.

Второе рекуррентное соотношение – особый случай (3.29), когда $a = 1$, $b = 10$ и $k = 0$. Таким образом, согласно основной теореме, его сложность – логарифмическая, поскольку $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$.

Тот же результат можно получить методом расширения:

$$\begin{aligned} T(n) &= T(n/10) + 1 = T(n/100) + 1 + 1 \\ &= T(n/10^2) + 2 = T(n/1000) + 1 + 2 \\ &= T(n/10^3) + 3 = T(n/10000) + 1 + 3 \\ &= T(n/10^4) + 4 \\ &\quad \ldots \\ &= T(n/10^i) + i. \end{aligned}$$

Начальное условие выполняется при $n/10^i = 1$, то есть при $i = \log_{10} n$.

Подстановка даёт:

$$T(n) = T(1) + \log_{10} n = 1 + \log_{10} n \in \Theta(\log n).$$

3.3.2. Общий метод решения разностных уравнений

Метод расширения эффективен при решении рекуррентных соотношений, в которых рекурсивная функция появляется только однажды. В этом разделе описывается мощный подход, называемый здесь «общим методом решения разностных уравнений», позволяющий решать рекуррентные соотношения, в которых рекурсивная функция встречается более одного раза. В частности, метод может использоваться для решения рекуррентных соотношений следующего вида:

$$\begin{aligned} T(n) = &- a_1 T(n-1) - \ldots - a_k T(n-k) && \{\text{«}T\text{-разности»}\} \\ &+ P_1^{d_1}(n) b_1^n + \ldots + P_s^{d_s}(n) b_s^n && \{\text{полином} \times \text{экспонента}\}, \end{aligned} \tag{3.33}$$

где a_i и b_i – константы, а $P_i^{d_i}(n)$ – полиномы от n степени d_i. Слагаемые правой части определения (уравнения), содержащие функцию T, могут встречаться несколько раз. Кроме того, их параметром обязательно должна быть разность n и некоторой целочисленной константы. Поэтому такие рекуррентные соотношения обычно называют «разностными уравнениями», а их T-члены – «T-разностями», чтобы подчеркнуть, что аргументы функции T не могут быть дробями вида n/b, где b – константа

(в противном случае их следует привести к виду (3.33)). Кроме того, в правой части определения может быть несколько слагаемых, представляющих собой произведения полинома на экспоненту степени n.

В следующих подразделах приводится подробное описание общего метода решения разностных уравнений, начиная с простых рекуррентных соотношений с постепенным их усложнением за счёт добавления к ним новых элементов.

3.3.2.1. *Однородные рекуррентные соотношения: характеристический полином с различными корнями*

Однородное рекуррентное соотношение содержит только T-разности:

$$T(n) = -a_1 T(n-1) - \ldots - a_k T(n-k).$$

Первый шаг его решения состоит в переносе всех членов правой части уравнения в левую:

$$T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = 0.$$

После этого подстановкой $T(n-z) = x^{k-z}$, где $z = 0, \ldots, k$, определяется соответствующий ему *характеристический полином*:

$$x^k + a_1 x^{k-1} + \ldots + a_{k-1} x + a_k.$$

Всё, что мы сделали, – заменили $T(n)$ на x^k, $T(n-1)$ на x^{k-1} и т. д. Коэффициент a_k при T-разности с наименьшим аргументом становится константой полинома. Следующий шаг заключается в нахождении k корней характеристического полинома. Если r_i ($i = 1, 2, \ldots, k$) – его корни, то полином можно разложить на множители:

$$(x - r_1)(x - r_2) \ldots (x - r_k).$$

Если все его корни различны, то формула для T будет такой:

$$T(n) = C_1 r_1^n + \ldots + C_k r_k^n. \qquad (3.34)$$

В итоге значения констант C_i зависят от начальных условий T и могут быть найдены из решения системы k линейных уравнений с k неизвестными переменными. Для составления такой системы уравнений нам требуется k начальных значений T (начальных условий T для малых значений n).

В следующем примере этот подход применяется к функции Фибоначчи (1.2), приведённой в листинге 1.3. Её можно записать следующим образом:

$$T(n) = \begin{cases} 0, & \text{если } n = 0, \\ 1, & \text{если } n = 1, \\ T(n-1) + T(n-2), & \text{если } n > 1, \end{cases} \tag{3.35}$$

где к начальному условию для $n = 1$ добавлено ещё одно – для $n = 0$, которое, как мы вскоре увидим, окажется очень полезным и даже необходимым. Первый шаг заключается в переносе всех членов T-разностей в левую часть рекурсивного тождества:

$$T(n) - T(n-1) - T(n-2) = 0.$$

Это однородное рекуррентное соотношение. Его характеристический полином:

$$x^2 - x - 1,$$

корни которого

$$r_1 = (1 + \sqrt{5})/2 \quad \text{и} \quad r_2 = (1 - \sqrt{5})/2$$

различны. Следовательно, формулой этого рекуррентного соотношения будет:

$$T(n) = C_1 r_1^{\,n} + C_2 r_2^{\,n} = C_1((1 + \sqrt{5})/2)^n + C_2((1 - \sqrt{5})/2)^n. \tag{3.36}$$

Последний шаг – это нахождение значений констант C_i решением системы линейных уравнений, которые получаются подстановкой в общую формулу (3.36) малых значений n из начальных условий рекурсивной функции (3.35). Первое, самое простое уравнение получается из якобы избыточного начального условия для $n = 0$, которое было добавлено в рекурсивное определение (3.35). Для второго уравнения используем начальное условие для $n = 1$. В результате получаем следующую систему линейных уравнений:

$$\begin{cases} C_1 + C_2 = 0 = T(0) \\ ((1+\sqrt{5})/2)C_1 + ((1-\sqrt{5})/2)C_2 = 1 = T(1) \end{cases}.$$

Её решение:

$$C_1 = 1/\sqrt{5} \quad \text{и} \quad C_2 = -1/\sqrt{5}.$$

Поэтому функцию Фибоначчи можно записать так:

$$T(n) = F(n) = 1/\sqrt{5}(((1 + \sqrt{5})/2)^n - ((1 - \sqrt{5})/2)^n) \in \Theta(((1 + \sqrt{5})/2)^n), \tag{3.37}$$

что соответствует показательной функции. Отметим, что в формуле (3.36) r_1^n явно доминирует над r_2^n, поскольку, с одной стороны, $r_1 > r_2$, а с

другой – $|r_2| < 1$, из чего следует, что r_2^n стремится к 0 с ростом n. Наконец, несмотря на сложность и иррациональность формулы, её результатом будет, очевидно, целое число.

В качестве следующего примера рассмотрим взаимно-рекурсивные функции (1.17) и (1.18). Для применения к ним нашего метода их нужно переопределить исключительно через самих себя. С одной стороны, $B(n) = A(n-1)$ подразумевает, что $B(n-1) = A(n-2)$. Кроме того, $A(2) = 1$. Таким образом, $A(n)$ можно переопределить как:

$$A(n) = \begin{cases} 0, & \text{если } n = 1, \\ 1, & \text{если } n = 2, \\ A(n-1) + A(n-2), & \text{если } n \geq 3. \end{cases} \quad (3.38)$$

С другой стороны, поскольку $A(n-1) = B(n)$ и $A(n) = B(n+1)$, их можно подставить в (1.17), чтобы получить $B(n+1) = B(n) + B(n-1)$. Кроме того, поскольку $B(2) = 0$, то $B(n)$ можно определить как:

$$B(n) = \begin{cases} 1, & \text{если } n = 1, \\ 0, & \text{если } n = 2, \\ B(n-1) + B(n-2), & \text{если } n \geq 3. \end{cases} \quad (3.39)$$

Обе эти функции имеют вид (3.36), и единственное различие между ними – в значениях констант. А именно:

$$A(n) = (1/2 - 1/2\sqrt{5})\,((1+\sqrt{5})/2)^n + (1/2 + 1/2\sqrt{5})\,((1-\sqrt{5})/2)^n, \quad (3.40)$$

$$B(n) = (-1/2 + 3/2\sqrt{5})\,((1+\sqrt{5})/2)^n + (-1/2 - 3/2\sqrt{5})\,((1-\sqrt{5})/2)^n, \quad (3.41)$$

где можно положить, что $A(0) = 1$ и $B(0) = -1$. Наконец, нетрудно видеть, что $A(n) + B(n)$ – числа Фибоначчи $F(n)$, так как сложение (3.40) и (3.41) даёт (3.37).

3.3.2.2. *Однородные рекуррентные соотношения: характеристический полином с кратными корнями*

В предыдущем разделе показано, что если кратность корней r_i, соответствующих члену $(x - r_i)$ в разложении на множители характеристического полинома, равна 1, то формула для $T(n)$ будет содержать член $C_i r_i^n$. Если же кратность m корня r больше 1, то, исходя из $(x - r)^m$, $T(n)$ будет содержать m членов вида «*константа* × *полином* × r^n», при этом полиномы должны иметь различные степени n в диапазоне от 0 до $m - 1$. Например, член $(x-2)^4$ в разложении на множители характеристического

полинома привёл бы к четырём следующим слагаемым в формуле для $T(n)$:

$$C_1 2^n + C_2 n 2^n + C_3 n^2 2^n + C_4 n^3 2^n$$

для некоторых констант C_i. Поэтому $T(n)$ можно записать в общем виде как

$$T(n) = C_1 P_1(n) r_1^n + \ldots + C_k P_k(n) r_k^n, \tag{3.42}$$

где $P_i(n)$ – полиномы вида n^c для некоторого c.

В качестве примера рассмотрим следующее рекуррентное соотношение:

$$T(n) = 5T(n-1) - 9T(n-2) + 7T(n-3) - 2T(n-4),$$

для которого $T(0) = 0$, $T(1) = 2$, $T(2) = 11$ и $T(3) = 28$. Его можно записать как

$$T(n) - 5T(n-1) + 9T(n-2) - 7T(n-3) + 2T(n-4) = 0,$$

что приводит к характеристическому полиному:

$$x^4 - 5x^3 + 9x^2 - 7x + 2.$$

Его можно разложить на множители (например, с помощью правила Руффини) следующим образом:

$$(x-1)^3(x-2),$$

из чего следует, что формула рекуррентного соотношения будет иметь вид:

$$\begin{aligned} T(n) &= C_1 \cdot 1 \cdot 1^n + C_2 \cdot n \cdot 1^n + C_3 \cdot n^2 \cdot 1^n + C_4 \cdot 1 \cdot 2^n \\ &= C_1 + C_2 n + C_3 n^2 + C_4 2^n, \end{aligned}$$

в которой есть слагаемые, связанные с корнем $r = 1$. Константы определяются из решения следующей системы линейных уравнений:

$$\begin{cases} C_1 + C_4 = 0 = T(0) \\ C_1 + C_2 + C_3 + 2C_4 = 2 = T(1) \\ C_1 + 2C_2 + 4C_3 + 4C_4 = 11 = T(2) \\ C_1 + 3C_2 + 9C_3 + 8C_4 = 28 = T(3) \end{cases}.$$

Её решение: $C_1 = -1$, $C_2 = -2$, $C_3 = 3$ и $C_4 = 1$. (В листинге 3.2 приводится код для решения системы линейных уравнений вида $\mathbf{A}x = \mathbf{b}$ с помощью пакета NumPy.) В итоге получаем:

$$T(n) = -1 - 2n + 3n^2 + 2^n \in \Theta(2^n)$$

с экспоненциальным порядком роста.

Листинг 3.2. Решение системы линейных уравнений $\mathbf{A}x = \mathbf{b}$

```
import numpy as np

A = np.array([[1, 0, 0, 1], [1, 1, 1, 2],
              [1, 2, 4, 4], [1, 3, 9, 8]])
b = np.array([0, 2, 11, 28])
x = np.linalg.solve(A, b)
```

3.3.2.3. *Неоднородные рекуррентные соотношения*

Неоднородные рекуррентные соотношения содержат в своей правой части нерекурсивные члены. Для таких рекуррентных соотношений тоже можно вывести общую формулу, если нерекурсивные члены представляют собой произведение полинома на экспоненту (см. (3.33)). Процедура решения – та же, но каждому члену $P_i^{d_i}(n)b_i^n$ в характеристическом полиноме соответствует слагаемое $(x - b_i)^{d_i+1}$, где d_i – степень полинома $P_i(n)$.

Рассмотрим следующее рекуррентное соотношение:

$$T(n) = 2T(n-1) - T(n-2) + 3^n + n3^n + 3 + n + n^2.$$

Как и в предыдущих примерах, первый шаг – это перенос T-разностей в левую часть рекуррентного соотношения:

$$T(n) - 2T(n-1) + T(n-2) = 3^n + n3^n + 3 + n + n^2.$$

На следующем шаге нужно представить каждый член правой части в виде произведения полинома на экспоненту. Если этот член состоит только из экспоненты, то естественно умножить его на полином $n^0 = 1$. Если же он представляет собой только полином, то он умножается на экспоненту 1^n. Таким образом, рекуррентное соотношение можно записать в виде:

$$T(n) - 2T(n-1) + T(n-2) = 1 \cdot 3^n + n \cdot 3^n + 3 \cdot 1^n + n \cdot 1^n + n^2 \cdot 1^n.$$

Если какая-то экспонента встречается в правой части несколько раз, она должна умножаться только на один полином. Следовательно, экспоненты правой части нужно сгруппировать:

$$T(n) - 2T(n-1) + T(n-2) = (1 + n) \cdot 3^n + (3 + n + n^2) \cdot 1^n.$$

На следующем шаге определяется характеристический полином. В левой части мы имеем $(x^2 - 2x + 1) = (x - 1)^2$. В правой части члену $(1 + n) \cdot 3^n$ соответствует $(x - 3)^2$, где 3 – основание экспоненты, а 2 – степень полинома (1) плюс 1. Аналогично члену $(3 + n + n^2) \cdot 1^n$ соответствует $(x - 1)^3$. Таким образом, характеристический полином имеет вид:

$$(x - 1)^2 (x - 3)^2 (x - 1)^3 = (x - 1)^5 (x - 3)^2,$$

а формула для $T(n)$ имеет следующий вид:

$$T(n) = C_1 + C_2 n + C_3 n^2 + C_4 n^3 + C_5 n^4 + C_6 3^n + C_7 n 3^n.$$

Проиллюстрируем данный подход следующим примером неоднородного рекуррентного соотношения:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ 2T(n-1) + n + 2^n, & \text{если } n > 0. \end{cases}$$

Сначала приводим рекурсивное условие к каноническому виду:

$$T(n) - 2T(n - 1) = n \cdot 1^n + 1 \cdot 2^n.$$

Соответствующий ему разложенный на множители характеристический полином:

$(x - 2)$ {из $T(n) - 2\,T(n - 1)$} $\cdot\, (x - 1)^2$ {из $n \cdot 1^n$} $\cdot\, (x - 2)$ {из $1 \cdot 2^n$} $= (x - 1)^2 (x - 2)^2$,

из чего следует, что формулой рекуррентного соотношения будет:

$$T(n) = C_1 + C_2 n + C_3 2^n + C_4 n 2^n.$$

Поскольку у нас есть четыре неизвестные константы, нам нужны значения T для четырёх различных n. Это значит, что к уже имеющемуся значению из начального условия $T(0) = 1$ нужно добавить из рекурсивного условия $T(n) = 2\,T(n - 1) + n + 2^n$ ещё три – $T(1)$, $T(2)$ и $T(3)$, а именно: $T(1) = 5$, $T(2) = 16$ и $T(3) = 43$. Теперь на основании этих данных можно записать следующую систему линейных уравнений:

$$\begin{cases} C_1 + C_3 = 1 = T(0) \\ C_1 + C_2 + 2C_3 + 2C_4 = 5 = T(1) \\ C_1 + 2C_2 + 4C_3 + 8C_4 = 16 = T(2) \\ C_1 + 3C_2 + 8C_3 + 24C_4 = 43 = T(3). \end{cases}$$

Её решение: $C_1 = -2$, $C_2 = -1$, $C_3 = 3$, $C_4 = 1$. Поэтому формула для $T(n)$:

$$T(n) = -2 - n + 3 \cdot 2^n + n 2^n \in \Theta(n 2^n).$$

3.3.2.4. *Дробные аргументы рекурсивной функции*

Общий метод вывода формулы для рекуррентных соотношений применим лишь тогда, когда аргументы их T-членов представляют собой разности вида $n - b$, где b – некоторая целочисленная константа. Однако в отдельных случаях эту задачу можно решить, когда аргументы T-членов представляют собой дроби вида n/b, где b – некоторая целочисленная константа. Ключевая идея здесь заключается в предположении, что n – это степень b, и последующем преобразовании дроби в разность путём замены $n = b^k$.

Рассмотрим рекуррентное соотношение (3.30) с рекурсивным условием

$$T(n) = 2T(n/2) + 1,$$

предположив, что n – степень двух. Поскольку аргумент T-члена в правой части – $n/2$, необходимо произвести замену $n = 2^k$, чтобы получить:

$$T(2^k) = 2T(2^k/2) + 1 = 2T(2^{k-1}) + 1.$$

В этом новом определении $T(2^k)$ есть функция k. Поэтому её можно заменить на $t(k)$ следующим образом:

$$t(k) = 2t(k - 1) + 1,$$

а такое рекуррентное соотношение можно решить общим методом для разностных уравнений. То есть сначала привести его к виду

$$t(k) - 2t(k - 1) = 1 \cdot 1^n$$

и записать его характеристический полином

$$(x - 2)(x - 1).$$

Тогда сама функция должна иметь следующий вид:

$$t(k) = C_1 + C_2 2^k. \tag{3.43}$$

Выполнив обратную замену, получаем общее решение рекуррентного соотношения от исходной переменной n:

$$T(n) = C_1 + C_2 n. \tag{3.44}$$

Последний шаг – определение констант C_1 и C_2. Их можно найти из (3.43) или из (3.44). Для T мы можем использовать начальные условия $T(1) = 1$ и $T(2) = 3$. Аналогичные начальные условия для t: $t(0) = T(2^0) = T(1) = 1$ и $t(1) = T(2^1) = T(2) = 3$. В любом случае, система линейных уравнений будет иметь вид:

$$\begin{cases} C_1 + C_2 = 1 = T(1) = t(0) \\ C_1 + 2C_2 = 3 = T(2) = t(1) \end{cases}.$$

Её решение: $C_1 = -1$ и $C_2 = 2$, а формулой для $T(n)$ будет:

$$T(n) = 2n - 1 \in \Theta(n).$$

В следующем рекурсивном определении

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2T(n/4) + \log_2 n, & \text{если } n > 1, \end{cases}$$

можно предположить, что аргументом рекурсивной функции T будет степень четырёх. В этом случае замена переменной $n = 4^k$ приводит к

$$T(4^k) = 2T(4^k/4) + \log_2 4^k = 2T(4^{k-1}) + k \log_2 4 = 2T(4^{k-1}) + 2k.$$

Следующий шаг – замена $t(k) = T(4^k)$:

$$t(k) = 2t(k - 1) + 2k.$$

Мы назовём эту операцию «заменой функции», чтобы подчеркнуть, что замена касается только T-членов (t-членами) и не влияет на все остальные члены (в данном случае – $2k$). Приводим рекуррентное соотношение к каноническому виду

$$t(k) - 2t(k - 1) = 2k \cdot 1^k,$$

характеристический полином которого –

$$(x - 2)(x - 1)^2.$$

Таким образом, формула для этого рекуррентного соотношения:

$$t(k) = C_1 2^k + C_2 + C_3 k.$$

Чтобы вернуться к функции T, нужно произвести обратную замену $k = \log_4 n$. Однако нам необходимо выражение для 2^k. Заметим, что $n = 4^k = (2 \cdot 2)^k = (2^2)^k = (2^k)^2$, поэтому $2^k = \sqrt{n}$, что даёт:

$$T(n) = C_1 \sqrt{n} + C_2 + C_3 \log_4 n.$$

Наконец, исходя из начального условия $T(1) = 1$, можно вычислить $T(4) = 2T(1) + \log_2 4 = 4$ и $T(16) = 2T(4) + \log_2 16 = 12$, что позволяет нам найти константы C_i из решения следующей системы линейных уравнений:

$$\begin{cases} C_1 + C_2 = 1 = T(1) \\ 2C_1 + C_2 + C_3 = 4 = T(4) \\ 4C_1 + C_2 + 2C_3 = 12 = T(16). \end{cases}.$$

Её решение: $C_1 = 5$, $C_2 = -4$ и $C_3 = -2$, а формула для $T(n)$:

$$T(n) = 5\sqrt{n} - 4 - 2\log_4 n \in \Theta(n^{1/2}).$$

Если аргумент T в рекурсивном выражении – квадратный корень, можно воспользоваться заменой $n = 2^{(2k)}$. Рассмотрим следующее рекуррентное соотношение:

$$T(n) = 2T(\sqrt{n}) + \log_2 n, \tag{3.45}$$

где $T(2) = 1$ и $n = 2^{(2k)}$ (заметим, что $2^{(2k)} \neq 4^k$). Это последнее ограничение для n гарантирует, что рекурсивная процедура завершится при начальном условии для $n = 2$. После замены переменной получим:

$$T(2^{(2k)}) = 2T(2^{(2k/2)}) + \log_2 2^{(2k)} = 2T(2^{(2k-1)}) + 2^k,$$

а после замены функции $t(k) = T(2^{(2k)})$ рекуррентное соотношение принимает вид

$$t(k) = t(k-1) + 2^k,$$

характеристический полином которого

$$(x - 2)(x - 2).$$

Поэтому новое рекуррентное соотношение будет иметь вид:

$$t(k) = C_1 2^k + C_2 k 2^k.$$

Для отмены замены переменной нужно применить подстановку $k = \log_2(\log_2 n)$ или $2^k = \log_2 n$. Таким образом, рекуррентное соотношение как функция n будет иметь вид:

$$T(n) = C_1 \log_2 n + C_2(\log_2(\log_2 n)) \log_2 n. \tag{3.46}$$

Наконец, чтобы найти константы, используем начальные условия $T(2) = 1$ и $T(4) = 4$. Для этого нужно решить следующую систему линейных уравнений:

$$\begin{cases} C_1 = 1 = T(2) \\ 2C_1 + 2C_2 = 4 = T(4) \end{cases}.$$

Её решение: $C_1 = C_2 = 1$. Поэтому конечная формула для $T(n)$:

$$T(n) = \log_2 n + (\log_2(\log_2 n)) \log_2 n \in \Theta((\log(\log n)) \log n). \tag{3.47}$$

3.3.2.5. *Многократные замены переменной или функции*

Предыдущее рекуррентное соотношение (3.45) можно решить, применив две последовательные замены переменной (и функции). Во-первых, замена $n = 2^k$ приводит к

$$T(2^k) = 2T(\sqrt{n^k}) + \log_2 2^k = T(2^{k/2}) + k.$$

Заменив $T(2^k)$ на $t(k)$, получаем следующее рекуррентное соотношение:

$$t(k) = 2t(k/2) + k,$$

которое всё ещё нельзя решить общим методом, так как оно не является разностным уравнением. Но зато можно произвести ещё одну замену переменной $k = 2^m$, чтобы привести его к нужному виду:

$$t(2^m) = 2t(2^{m-1}) + 2^m,$$

что после замены функции $t(2^m) = u(m)$ сводится к разностному уравнению

$$u(m) = 2u(m - 1) + 2^m.$$

Его характеристический полином $(x - 2)^2$, а это значит, что рекуррентное соотношение будет иметь вид:

$$u(m) = C_1 2^m + C_2 m 2^m.$$

Отмена последней замены переменной приводит к

$$t(k) = C_1 k + C_2(\log_2(k))k$$

и в итоге к

$$T(n) = C_1 \log_2 n + C_2(\log_2(\log_2 n))\log_2 n,$$

что равносильно (3.46). Поэтому решением является (3.47).

Стратегия применения нескольких замен переменных или функций может использоваться для решения более сложных рекуррентных соотношений, таких, например, как следующее нелинейное соотношение:

$$T(n) = \begin{cases} 1/3, & \text{если } n = 1, \\ nT(n/2)^2, & \text{если } n > 1, \end{cases}$$

где n – степень двух. Сначала можно применить замену переменной $n = 2^k$, что приводит к

$$T(2^k) = 2^k T(2^k/2)^2 = 2^k T(2^{k-1})^2.$$

После замены функции $T(2^k) = t(k)$ получаем

$$t(k) = 2^k t(k-1)^2,$$

что всё ещё невозможно решить общим методом. Но после логарифмирования обеих частей равенства

$$\log_2 t(k) = \log_2 2^k t(k-1)^2 = k + 2\log_2 t(k-1)$$

можно произвести более сложную замену функции $\log_2 t(k) = u(k)$, которая даст рекуррентное соотношение

$$u(k) = k + 2u(k-1),$$

которое уже можно решить общим методом. Его характеристический полином

$$(x-2)(x-1)^2,$$

который подразумевает, что $u(k)$ имеет следующий вид:

$$u(k) = C_1 2^k + C_2 + C_3 k.$$

Отменив замены, получим сначала

$$t(k) = 2^{C_1 2^k + C_2 + C_3 k},$$

а затем

$$T(n) = 2^{C_1 n + C_2 + C_3 \log_2 n}.$$

Последний шаг, как обычно, состоит из определения констант. Используя начальное условие $T(1) = 1/3$, получаем $T(2) = 2[T(1)]^2 = 2/9$ и $T(4) = 4[T(2)]^2 = 4(2/9)^2 = 16/81$. Воспользуемся этими значениями для составления следующей системы (нелинейных) уравнений:

$$\begin{cases} 2^{C_1+C_2} = 1/3 = T(1) \\ 2^{2C_1+C_2+C_3} = 2/9 = T(2) \\ 2^{4C_1+C_2+2C_3} = 16/81 = T(4) \end{cases},$$

которую (после логарифмирования по основанию 2 обеих частей уравнений) можно привести к системе линейных уравнений:

$$\begin{cases} C_1 + C_2 = -\log_2 3 \\ 2C_1 + C_2 + C_3 = 1 - \log_2 9 = 1 - 2\log_2 3 \\ 4C_1 + C_2 + 2C_3 = 4 - \log_2 81 = 4 - 4\log_2 3. \end{cases}.$$

Её решение: $C_1 = 2 - \log_2 3 = \log_2(4/3)$, $C_2 = -2$, $C_3 = -1$.

Таким образом, $T(n)$ можно выразить следующей формулой:

$$T(n) = 2^{(\log_2(4/3))n - 2 - \log_2 n} = 2^{\log_2(4/3)n} \cdot 2^{-2} \cdot 2^{\log_2(1/n)} = (4/3)^n(1/4)n \in \Theta((4/3)^n n).$$

3.4. Упражнения

Упражнение 3.1. Докажите следующее тождество:

$$(a/b^k)^{\log_b n} = n^{\log_b a}/n^k.$$

Упражнение 3.2. Используя пределы, покажите, что $\log n! \in \Theta(n \log n)$. Подсказка: при стремлении n к бесконечности $n!$ можно заменить приближением Стирлинга $n! \approx \sqrt{2\pi n}(n/e)^n$.

Упражнение 3.3. Покажите, что $n \log n \in O(n^{1+a})$, где $a > 0$. Используйте пределы и правило Лопиталя.

Упражнение 3.4. Пусть m и n – произвольные целые числа. Определите $\sum_{i=m}^{n} 1$.

Упражнение 3.5. Запишите сумму первых n нечётных целых чисел с использованием символа суммирования и упростите её (результат должен быть известным полиномом).

Упражнение 3.6. Воспользуйтесь следующим тождеством:

$$\sum_{i=1}^{n} i(i+1)/2 = \sum_{i=1}^{n} i(n-i+1),$$

чтобы получить формулу суммы квадратов первых n целых чисел $(1^2 + 2^2 + \ldots + n^2)$.

Упражнение 3.7. Покажите, что частичная сумма n членов арифметической прогрессии ($s_i = s_{i-1} + d$ для некоторого начального s_0) равна

$$\sum_{i=1}^{n} s_i = n(s_0 + s_{n-1})/2.$$

Упражнение 3.8. Алгоритм проверяет справа налево биты двоичного представления чисел от 1 до $2^n - 1$ (где n – количество битов каждого числа), пока не находит единичный бит. Для заданного n определите, используя символы суммирования, общее количество проверенных алгоритмом битов. Например, для $n = 4$ алгоритм проверит 26 затенённых битов на рис. 3.11.

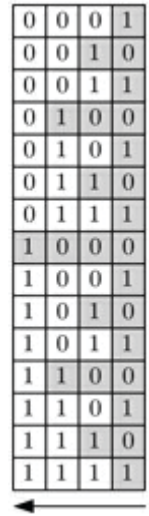


Рис. 3.11. Количество проверенных (затенённых) битов для чисел от 1 до $2^n - 1$, где $n = 4$

Упражнение 3.9. Определите рекуррентные соотношения для времени выполнения алгоритмов, реализующих функции (1.14), (1.15), (1.16) и (1.19).

Упражнение 3.10. Выведите формулы для следующих рекуррентных соотношений, не вычисляя коэффициентов C_i (начальные условия здесь не требуются).

$$T(n) = 4T(n-1) - 5T(n-2) + 2T(n-3) + n - 3 + 5n^2 \cdot 2^n$$
$$T(n) = T(n-1) + 3n - 3 + n^3 \cdot 3^n$$
$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3) + 3 + n^2 + n \cdot 2^n$$

Упражнение 3.11. Определите рекуррентное соотношение для времени выполнения кода из листинга 2.6, решите его методом расширения и общим методом для разностных уравнений и, наконец, определите его порядок роста.

Упражнение 3.12. Решите методом расширения следующее рекуррентное соотношение:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j), & \text{если } n > 0. \end{cases}$$

Упражнение 3.13. Решите методом расширения следующее рекуррентное соотношение:

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2T(n/4) + \log_2 n, & \text{если } n > 1. \end{cases}$$

Упражнение 3.14. Решите рекуррентное соотношение (3.18) общим методом для разностных уравнений.

Упражнение 3.15. Определите рекуррентное соотношение для времени выполнения кода из листинга 2.9, решите его с помощью основной теоремы, метода расширения и общего метода для разностных уравнений, а затем определите его порядок роста.

Упражнение 3.16. Решите следующее рекуррентное соотношение с помощью основной теоремы, метода расширения, общего метода для разностных уравнений и определите его порядок роста:

$$T(n) = \begin{cases} 1, & \text{если } n = 1, \\ 3T(n/2) + n, & \text{если } n > 1, \end{cases}$$

где n – степень 2.

Упражнение 3.17. Решите рекуррентное соотношение (3.32) общим методом для разностных уравнений.

Упражнение 3.18. Решите следующие рекуррентные соотношения:

1) $T(n) = 2T(n-1) + 3n - 2$, где $T(0) = 0$.
2) $T(n) = T(n/2) + n$, где $T(1) = 1$ и n – степень 2.
3) $T(n) = T(n/\alpha) + n$, где $T(1) = 0$ и $\alpha \geqslant 2$ – целое.
4) $T(n) = T(n/3) + n^2$, где $T(1) = 1$ и n – степень 3.
5) $T(n) = 3T(n/3) + n^2$, где $T(1) = 0$ и n – степень 3.
6) $T(n) = 2T(n/4) + n$, где $T(1) = 1$ и n – степень 4.
7) $T(n) = T(n/2) + \log_2 n$, где $T(1) = 1$ и n – степень 2.
8) $T(n) = 4\,T(n/2) + n$, где $T(1) = 1$ и n – степень 2.
9) $T(n) = 2\,T(n/2) + n \log_2 n$, где $T(1) = 1$ и n – степень 2.
10) $T(n) = 3/2T(n/2) - 1/2T(n/4) - 1/n$, где $T(1) = 1$, $T(2) = 3/2$ и n – степень 2.

Глава 4

Линейная рекурсия I: основные алгоритмы

Трудное начинается с лёгкого, великое – с малого.
Долгий путь начинается с первого шага.

– Лао Цзы

Рекурсивные методы можно классифицировать по количеству обращений к самому себе в рекурсивных условиях. Эта глава исследует методы, которые не только вызывают себя лишь раз, но и обрабатывают результат рекурсивного вызова до того, как создать или вернуть свой собственный результат. Этот общий тип рекурсии известен как *линейная рекурсия*, когда единственный рекурсивный вызов не является последним действием метода. В данной главе будет предложено множество задач и соответствующих им линейно-рекурсивных решений, которые будут разрабатываться, опираясь на понятия и методику, введенные в главах 1 и 2. Если же единственный рекурсивный вызов является последним действием метода, то говорят, что имеет место *хвостовая рекурсия*. Этому особому типу рекурсии будет посвящена глава 5.

Линейная рекурсия – самый простой тип рекурсии, который может быть альтернативой итерации, многократно выполняющей определённую последовательность операций. Хотя обсуждаемые в этой главе задачи легко решить с помощью итерации, линейная рекурсия предлагает очень прозрачные примеры рекурсивного мышления и программирования. Поэтому в главе приводятся примеры применения рекурсивных понятий (декомпозиции и индукции), которые читатель обязан постичь, прежде чем заняться более сложными задачами и типами рекурсии.

4.1. Арифметические операции

Этот раздел представляет рекурсивные решения для нескольких простейших арифметических вычислений. Мы изучим их в целях иллюстрации, поскольку большинство из них включает в себя простые операции, которые реализуются посредством элементарных команд или выражений.

4.1.1. Степенная функция

Классическая задача для иллюстрации рекурсии – степенная функция. Цель состоит в том, чтобы вычислить b в степени n:

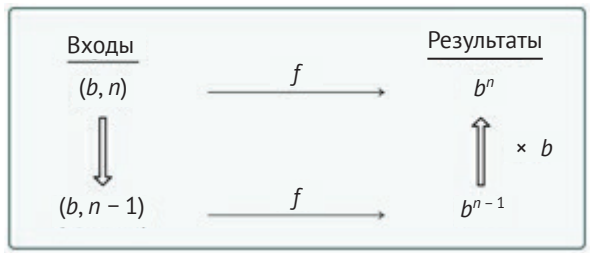
$$b^n = \prod_{i=1}^{n} b = \underbrace{b \times b \times \cdots b \times b}_{n \text{ раз}},$$

где основание степени b – вещественное число, а её показатель (экспонента) n – неотрицательное целое число (в Python степень вычисляется операцией **). Последующие подразделы исследуют алгоритмы вычисления степени за линейное и логарифмическое время.

4.1.1.1. *Вычисление степени за линейное время*

Согласно представленной на рис. 2.1 методике первый шаг состоит из определения размера задачи, который зависит от входных параметров. В этой задаче для определения числа операций в алгоритме вычисления степени важен лишь показатель степени. Ясно, что время выполнения растёт с ростом n, поскольку результат требует большего количества умножений. Таким образом, размер задачи – это показатель степени n. Начальное условие соответствует тривиальным экземплярам задачи минимального размера. Так как n – неотрицательное целое число, наименьший экземпляр класса – это b^0, равный 1.

На этапе декомпозиции нужно рассмотреть наименьшую и простейшую из подзадач, подобных исходной, которая получается путём уменьшения размера исходной задачи. В этом разделе мы будем уменьшать её размер на 1, что приведёт к следующей схеме:



где $f(b, n) = b^n$ – функция от двух параметров b и n. Очевидно, если нам известно решение $f(b, n - 1) = b^{n-1}$, то в рекурсивном условии остаётся сделать лишь одно – умножить её на b, чтобы получить b^n. Функцию вместе с её начальным условием можно реализовать, как показано в листинге 4.1.

Листинг 4.1. Степенная функция для неотрицательных степеней с линейным временем выполнения

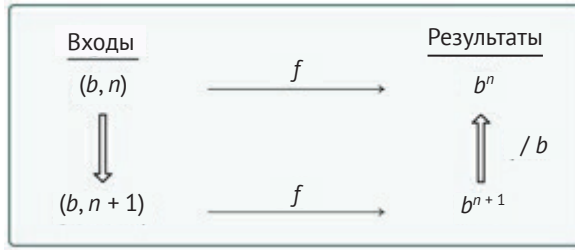
```
1  def power_linear(b, n):
2      if n == 0:
3          return 1
4      else:
5          return b * power_linear(b, n - 1)
```

Это пример линейной рекурсии, так как в рекурсивном условии есть только один вызов функции, не являющийся последней выполняемой алгоритмом операцией. Несмотря на то что вызов функции является последним элементом выражения (в строке 5), метод сначала вычисляет её и только потом умножает результат на b. Таким образом, функция не относится к хвостовой рекурсии. Также важно отметить, что функция не нуждается в начальном условии для $n = 1$, так как оно было бы избыточным. Наконец, вычислительная сложность этого алгоритма линейно зависит от n. В частности, время выполнения может быть определено так:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ T(n-1), & \text{если } n > 0, \end{cases} \tag{4.1}$$

где $T(n) = n + 1 \in \Theta(n)$.

Если условием задачи допускались бы ещё и отрицательные целочисленные показатели степени, то размер задачи был бы абсолютной величиной n. В этом случае начальное условие также задаётся для $n = 0$, а рекурсивное условие идентично случаю $n > 0$. Когда n – отрицательное, декомпозиция должна уменьшать размер задачи таким образом, чтобы соответствующая подзадача стремилась к начальному условию. Таким образом, для этого рекурсивного условия необходимо увеличивать n. Схема процесса рассуждений была бы такой:



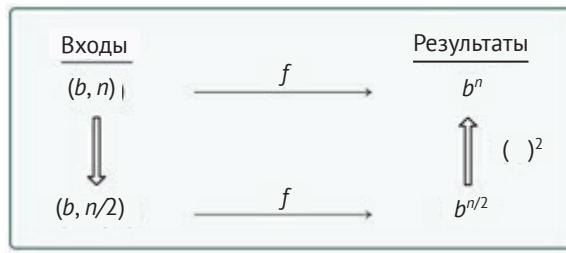
Таким образом, для получения отрицательной степени b^n результат подзадачи должен делиться на b. Поэтому рекурсивное условие должно быть $f(b, n) = f(b, n + 1)/b$, как показано в листинге 4.2.

Листинг 4.2. Степенная функция с линейным временем выполнения

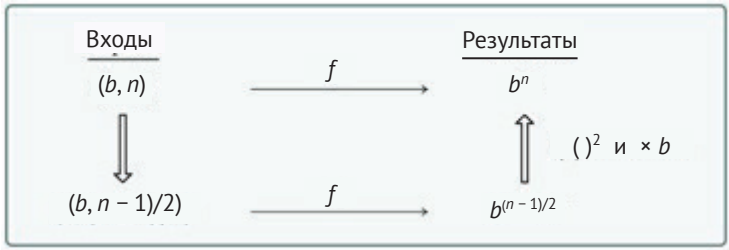
```
1  def power_gtntral_linear(b, n):
2      if n == 0:
3          return 1
4      elif n > 0:
5          return b * power_gtntral_linear(b, n - 1)
6      else:
7          return power_gtntral_linear(b, n + 1) / b
```

4.1.1.2. *Вычисление степени за логарифмическое время*

Существует возможность реализовать более эффективные алгоритмы вычисления степени за логарифмическое время, если на этапе декомпозиции разбить задачу пополам – на две подзадачи в половину размера исходной. Поскольку n – неотрицательное целое число, нам понадобятся две схемы в зависимости от четности n. Если n – чётное, то согласно рекурсивному подходу процесс можно представить следующим образом:



Таким образом, чтобы получить b^n, результат подзадачи нужно возвести в квадрат. И наоборот, если n – нечётное, следует рассмотреть подзадачу с целочисленным размером $(n - 1)/2$. Её схема:



В этом случае результат подзадачи тоже возводится в квадрат, после чего нужно умножить его на основание b. С этими рекурсивными условиями и тривиальным начальным условием рекурсивная функция выглядит следующим образом:

$$f(b,n)=\begin{cases}1, & \text{если } n=0,\\ f(b,n/2)^2, & \text{если } n>0 \text{ и } n \text{ – чётное},\\ f(b,(n-1)/2)^2\cdot b, & \text{если } n>0 \text{ и } n \text{ – нечётное}.\end{cases}$$

Правильная реализация линейно-рекурсивной функции показана в листинге 4.3. Целочисленное деление (//) вместо вещественного деления (/) не обязательно и применено, чтобы подчеркнуть, что второй входной параметр – целое число. Кроме того, если n – нечётное, то $(n - 1)//2 = n//2$. Однако в коде используется первое выражение, поскольку оно математически точнее.

Листинг 4.3. Степенная функция для неотрицательных степеней с логарифмическим временем выполнения

```
def power_logarithmic(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return power_logarithmic(b, n // 2)**2
    else:
        return b * (power_logarithmic(b, (n - 1) // 2)**2)
```

Время выполнения можно определить как:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ T(n/2) + 1, & \text{если } n > 0, \end{cases}$$

где $T(n) = 2 + \log_2 n$ для $n > 0$. Таким образом, $T(n) \in \Theta(\log n)$. Высокая производительность достигается за счёт деления размера задачи пополам на этапе декомпозиции. При этом функция в каждом рекурсивном условии должна выполнить только один рекурсивный вызов. Например, код в листинге 4.4 не выполняется за логарифмическое время, несмотря на то что декомпозиция делит задачу пополам. Проблема здесь в том, что он вычисляет результат одной и той же подзадачи дважды, используя два идентичных, но совершенно ненужных рекурсивных вызова.

Листинг 4.4. Неэффективная реализация степенной функции с линейным временем выполнения

```
def power_alt(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return power_alt(b, n // 2) * power_alt(b, n // 2)
    else:
        return (power_alt(b, (n - 1) // 2)
                * power_alt(b, (n - 1) // 2) * b)
```

Временна́я стоимость выполнения этой функции:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ 2T(n/2) + 1, & \text{если } n > 0, \end{cases}$$

где $T(n) = n + 1 \in \Theta(n)$. Отметим, что она аналогична времени выполнения (4.1) функции в листинге 4.1. Таким образом, сверхпроизводительность, которую можно было бы получить за счёт деления размера задачи пополам, здесь утрачивается из-за двух идентичных рекурсивных вызовов в рекурсивном условии.

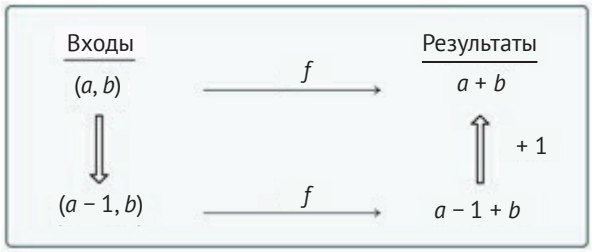
4.1.2. Медленное сложение

Эта задача заключается в сложении двух неотрицательных целых чисел a и b без использования обычных арифметических операций, таких

как сложение, вычитание, умножение или деление. Вместо этого разрешается лишь увеличивать или уменьшать число на единицу (конечное число раз). Эта простая задача поможет показать, как выбор размера влияет на способ декомпозиции задачи, приводя к разным рекурсивным алгоритмам.

Согласно шаблону на рис. 2.1 первый шаг – это определение размера задачи. Поскольку нам разрешено лишь добавлять или вычитать 1 из a и b, разумно считать, что эту простую операцию следует выполнять $a + b$ раз до тех пор, пока не выполнится начальное условие (вскоре мы увидим, что это можно сделать всего лишь за a или b операций). Если мы выбираем в качестве размера задачи $a + b$, то начальное условие выполняется при $a = b = 0$, что соответствует экземпляру наименьшего размера. Кроме того, мы должны рассмотреть другие возможные начальные условия, когда результат может быть легко получен без выполнения рекурсивного вызова. Очевидно, что для этой задачи результат равен b при $a = 0$ и равен a при $b = 0$. Более того, с такими начальными условиями становится излишним начальное условие для $a = 0$ *и* $b = 0$. Наконец, эти начальные условия гарантируют, что в рекурсивном условии a и b будут положительными.

На следующем шаге декомпозиции мы можем уменьшить размер задачи на 1. Поскольку в качестве размера задачи выбрано $a + b$, можно вычитать единицу как из a, так и из b. Рекурсивная схема при уменьшении a была бы такой:



Если реализуемая функция – f, то очевидно, что $f(a, b) = f(a - 1, b) + 1$ определяет рекурсивное условие, и его можно выразить математически как:

$$f(a,b) = \begin{cases} b, & \text{если } a = 0, \\ a, & \text{если } b = 0, \\ f(a-1,b)+1, & \text{если } a > 0,\ b > 0. \end{cases} \tag{4.2}$$

В листинге 4.5 приведена реализация функции в Python. Если бы мы выбрали в качестве размера задачи a, в определении функции появились бы те же два начальных условия, а при декомпозиции задачи пришлось бы уменьшать на единицу значение a. Поэтому получающаяся функция была бы точно такой же. С другой стороны, выбрав в качестве размера задачи b, пришлось бы изменить рекурсивное условие на $f(a, b) = f(a, b - 1) + 1$, поскольку уменьшать придётся b, а не a.

Листинг 4.5. Медленное сложение двух неотрицательных целых чисел

```
def slow_addition(a, b):
    if a == 0:
        return b
    elif b == 0:
        return a
    else:
        return slow_addition(a - 1, b) + 1
```

Функция в листинге 4.5 может быть медленной для больших значений a. С другой стороны, если размером задачи считать наименьший входной параметр, то есть $\min(a, b)$, то можно создать более эффективный алгоритм. В этом случае два начальных условия $f(0, b) = b$ и $f(a, 0) = a$ соответствуют наименьшим экземплярам задачи (начальное условие $f(0, 0)$ становится лишним). Для рекурсивных условий мы должны найти декомпозицию, гарантирующую уменьшение размера задачи. Первый подход состоит в уменьшении наименьшего входного параметра, чтобы размер подзадачи был $\min(a, b) - 1$. Например, при $a < b$ рекурсивное правило было бы $f(a, b) = f(a - 1, b) + 1$, а при $a \geqslant b$ мы применили бы правило $f(a, b) = f(a, b - 1) + 1$. Таким образом, математическая функция была бы следующей:

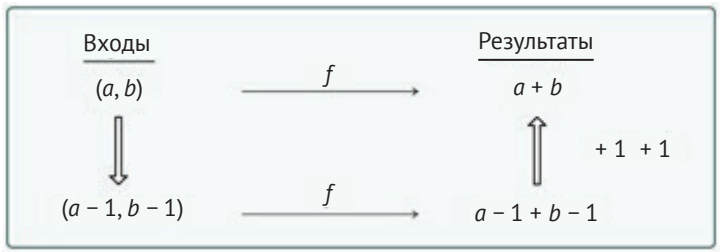
$$f(a,b) = \begin{cases} b, & \text{если } a = 0, \\ a, & \text{если } b = 0 \text{ (и } a \neq 0), \\ f(a-1,b)+1, & \text{если } a < b \text{ (и } a \neq 0,\ b \neq 0), \\ f(a,b-1)+1, & \text{если } b \leq a \text{ (и } a \neq 0, b \neq 0), \end{cases}$$

и её можно закодировать, как в листинге 4.6.

Листинг 4.6. Ускоренное медленное сложение двух неотрицательных целых чисел

```
def quicker_slow_addition(a, b):
    if a == 0:
        return b
    elif b == 0:
        return a
    elif a < b:
        return quicker_slow_addition(a - 1, b) + 1
    else:
        return quicker_slow_addition(a, b - 1) + 1
```

Вторая идея заключается в уменьшении обоих параметров, и она тоже гарантирует уменьшение размера задачи. В частности, отметим, что $\min(a - 1, b - 1) = \min(a, b) - 1$. Для этой декомпозиции рекурсивная схема была бы такой:



Следовательно, рекурсивным условием для $a > 0$ и $b > 0$ будет $f(a, b) = f(a - 1, b - 1) + 1 + 1$. По условию задачи оно справедливо, если допустить, что увеличение на 1 выполняется *постоянное* количество раз (в случае $f(a - 1, b - 1)$ – дважды), что не зависит от входных параметров. Соответствующий этому случаю код приведён в листинге 4.7.

Листинг 4.7. Ещё одно ускоренное медленное сложение двух неотрицательных целых чисел

```
def quicker_slow_addition_alt(a, b):
    if a == 0:
        return b
    elif b == 0:
        return a
    else:
        return quicker_slow_addition_alt(a - 1, b - 1) + 1 + 1
```

4.1.3. Двойная сумма

Следующий пример относится к вычислению двойной суммы вида:

$$f(m,n)=\sum_{i=1}^{m}\sum_{j=1}^{n}(i+j). \tag{4.3}$$

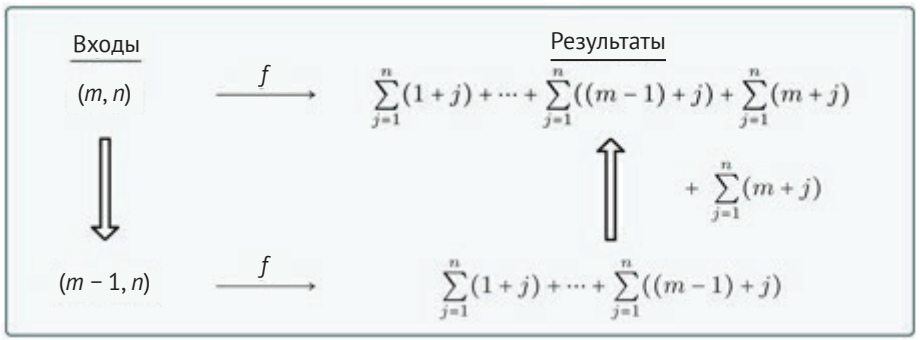
Пользуясь свойствами сумм, она может быть сведена к простому выражению:

$$f(m,n)=\sum_{i=1}^{m}\sum_{j=1}^{n}(i+j)=\sum_{i=1}^{m}(i\cdot n+n(n+1)/2)=$$

$$=nm(m+1)/2+mn(n+1)/2=mn(m+n+2)/2.$$

Однако мы решим эту задачу рекурсивно с использованием двух рекурсивных функций (по одной на каждую сумму).

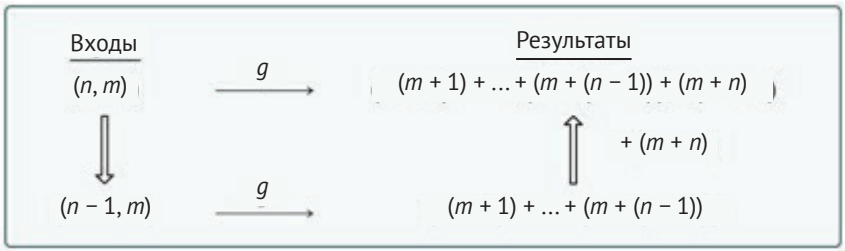
Во-первых, внешняя сумма – функция параметров m и n, размер которой – m. Функция возвращает 0 для её начального условия, когда m меньше наименьшего индекса (то есть когда $m \leqslant 0$). Общая схема для рекурсивного условия в предположении, что размер подзадачи $m-1$:



Очевидно, всё, что нужно сделать с результатом $f(m-1, n)$ для получения $f(m, n)$, – это добавить к нему внутреннюю сумму $\sum_{j=1}^{n}(m+j)$, которую можно считать функцией g, зависящей от n и m. Поэтому f можно определить следующим образом:

$$f(m,n)=\begin{cases}0, & \text{если } m\le 0,\\ f(m-1,n)+g(n,m), & \text{если } m>0.\end{cases}$$

$g(n, m)$ тоже можно определить рекурсивно. Во-первых, поскольку она складывает n слагаемых, её размер равен n. Как и все суммы, она возвращает 0 в начальном условии, которое имеет место при $n \leqslant 0$. Общая схема ее рекурсивного условия, также уменьшающего размер задачи на единицу:



Из схемы видно, что для получения $g(n, m)$ нужно добавить $(m + n)$ к результату подзадачи. Таким образом, функция g имеет вид:

$$g(n, m) = \begin{cases} 0, & \text{если } n \leq 0, \\ g(n-1, m) + (m+n), & \text{если } n > 0. \end{cases}$$

В листинге 4.8 приведена реализация обеих сумм.

Листинг 4.8. Рекурсивные функции вычисления двойной суммы (4.3)

```
def inner_sum(n, m):
    if n <= 0:
        return 0
    else:
        return inner_sum(n - 1, m) + (m + n)


def outer_sum(m, n):
    if m <= 0:
        return 0
    else:
        return outer_sum(m - 1, n) + inner_sum(n, m)
```

4.2. Системы счисления

Числа можно представлять по-разному в зависимости от конкретной *базы* или основания системы счисления. Как правило, мы используем систему счисления по основанию 10, когда последовательность цифр,

скажем, 142 представляет собой число $1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$. В общем случае для некоторого основания b значение числа x можно представить уникальной последовательностью из m цифр $d_{m-1} \ldots d_0$:

$$x = \sum_{i=0}^{m-1} d_i b^i, \tag{4.4}$$

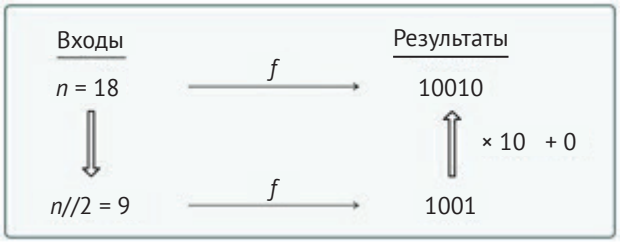
где $0 \leqslant d_i < b$ и $d_{m-1} \neq 0$ (то есть начальные нули опускаются). Таким образом, различные основания приводят к различным последовательностям цифр, представляющим одно и то же число. Что касается обозначений, то основание можно указывать в нижнем индексе, который обычно опускается для $b = 10$. Например, $142_{10} = 142$, но $142_5 = 1 \cdot 5^2 + 4 \cdot 5^1 + 2 \cdot 5^0 = 25 + 20 + 2 = 47$. В этом разделе мы рассмотрим алгоритмы преобразования чисел из одной системы счисления в другую.

4.2.1. Двоичное представление неотрицательного целого числа

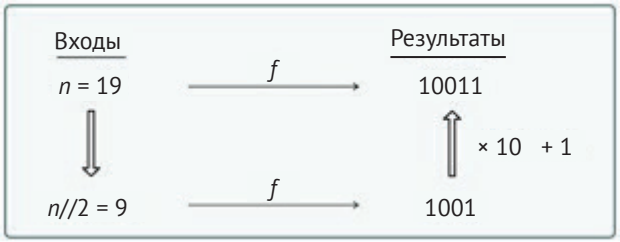
Цель этого примера – разработка рекурсивной функции, которая для заданного неотрицательного целого числа n по основанию 10 возвращает новое целое число – его двоичное представление, состоящее из последовательности двоичных цифр (или битов). Результатом будет также десятичное число, но его цифры будут либо нулями, либо единицами. Таким образом, можно считать, что его цифры фактически соответствуют битам.

Поскольку мы должны создать последовательность «битов», размер задачи – это число битов в двоичном представлении n, или математически $\lfloor \log_2 n \rfloor + 1$ (для $n > 0$). При этом формула для разработки рекурсивного алгоритма нам не нужна. Всё, что нам нужно, – чётко определить размер задачи, что позволит нам определить начальные условия и разбить задачу на подзадачи. В частности, наименьшие экземпляры задачи соответствуют числам, которые содержат один-единственный бит, которым могут быть 0 или 1. Таким образом, начальное условие с результатом n возникает при $n < 2$.

Для вывода рекурсивного условия нужно решить, как уменьшать размер задачи. Самый простой способ заключается в уменьшении количества битов на 1, что достигается целочисленным делением n на 2 (оно сдвигает биты на одну позицию вправо, причём наименьший значащий бит отбрасывается). Начнём разбор с конкретных экземпляров задачи. Например:

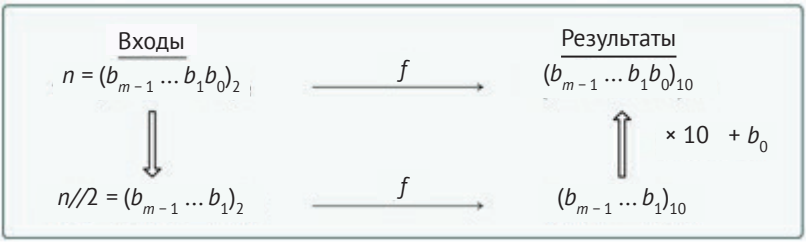


и



Отметим, что результат $f(18) = 10010$ выражен по основанию 10. На схемах показано, что желаемый результат можно получить, умножая результат подзадачи на 10 и затем добавляя к нему 1, если n – нечётное. Поэтому рекурсивное условие видится таким: $f(n) = 10f(n \,//\, 2) + n \,\%\, 2$.

Более строго схему рекурсивного процесса осмысления задачи можно обобщить следующим образом:



где b_i представляет бит. В частности, $f(n//2)$ – десятичное число, которое представляет последовательность «битов» n, за исключением самого младшего (b_0), чьё значение – $n\%2$ (или $n\&1$, где & обозначает поразрядную операцию AND). Чтобы добавить отброшенный бит, нужно умножить $f(n//2)$ на 10 (то есть сдвинуть цифры на одну позицию влево, добавив 0) и добавить его к результату. Таким образом, на самом деле рекурсивным условием действительно будет $f(n) = 10f(n \,//\, 2) + n \,\%\, 2$, полученное путём на основании анализа конкретных экземпляров задачи. Окончательная функция:

$$f(n) = \begin{cases} n, & \text{если } n < 2, \\ 10\,f(n//2) + n\%2, & \text{если } n \geq 2. \end{cases}$$

В листинге 4.9 приведён код для решения задачи. В итоге двоичное представление $142 = 142_{10}$ – это $10001110_2 = 128 + 8 + 4 + 2$.

Листинг 4.9. Двоичное представление неотрицательного целого числа

```
1  def decimal_binary(n):
2      if n < 0:
3          return n
4      else:
5          return 10 * decimal_binary(n // 2) + (n % 2)
```

4.2.2. Приведение десятичного числа к другому основанию

Задача из раздела 4.2.1 – частный случай более общей задачи, которая заключается в преобразовании десятичного числа к другому основанию b, где $b \geqslant 2$. Её можно решить аналогично с применением схем. На рис. 4.1 показаны шаги общего алгоритма решения задачи. В частности, он показывает, как привести десятичное число 142 к основанию 5, то есть $142_{10} = 1032_5$. Полное решение задачи показано на рис. 4.1(a), идея которого состоит в последовательном выполнении целочисленного деления n на b до достижения им значения 0. На каждом шаге вычисляются также остатки от деления b, которые образуют цифры в новом представлении числа по основанию b, чьи позиции определяются степенью 10. В схеме на рис. 4.1(b) подзадача ($28_{10} = 103_5$) исходной задачи выделена более светлым фоном.

Для разработки рекурсивного решения нужно определить размер задачи. Для этой задачи он равен количеству делений n на b до достижения им значения 0. Количество делений – это, в сущности, количество цифр по основанию b. Начальное условие выполняется, когда n представляет собой единственную цифру по основанию b. Другими словами, если $n < b$, то результат равен n.

Рекурсивное условие требует идентификации подзадачи в пределах исходной, как показано на рис. 4.1(b). Отметим, что подзадача подобна исходной задаче, начиная с $28 = 142//5$. Таким образом, декомпозиция состоит из выполнения целочисленного деления $n//b$, которое умень-

шает размер задачи на 1. Впоследствии нужно будет определить, как следует изменить решение подзадачи (103), чтобы получить решение исходной задачи (1032). Решение заключается в умножении результата подзадачи на 10 и добавлении к нему $n\%b$ (равного 2 в этом примере). Таким образом, функцию можно закодировать, как показано в листинге 4.10.

Листинг 4.10. Приведение неотрицательного целого числа *n* к основанию *b*

```
def decimal_to_base(n, b):
    if n < b:
        return n
    else:
        return 10 * decimal_to_base(n // b, b) + (n % b)
```

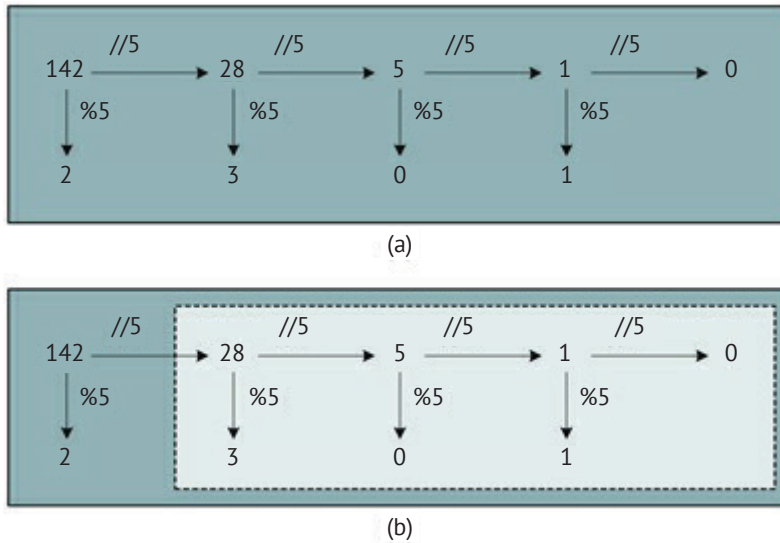


Рис. 4.1. Приведение десятичного числа 142 к основанию 5 (1032)

4.3. Строки

В этом разделе разбираются две задачи, касающиеся строк, которые представляют собой последовательности (цепочки) символов и являются фундаментальным типом данных во многих языках программирования.

4.3.1. Обращение строки

Рассмотрим задачу обращения строки. В частности, создадим функцию *f*, которая получает на входе строку и возвращает её обращение – цепочку из тех же символов, следующих в обратном порядке. Например, *f*('*abcd*') = '*dcba*'.

Размер задачи – это длина входной строки. Начальное условие выполняется при пустой входной строке, когда результатом, очевидно, является тоже пустая строка. Для уменьшения размера задачи в рекурсивном условии следует исключать по одному символу входной строки. Явные кандидаты для этого – первый или последний символ строки. Исключение первого символа приводит к следующей схеме, где входная строка ***s*** записывается в виде последовательности символов $s_0\, s_1 \ldots s_{n-2}\, s_{n-1}$:

Таким образом, функция должна просто добавить первый символ в конец результата подзадачи, связанной с $s_1 \ldots s_{n-2} s_{n-1}$ (символ «+» обозначает конкатенацию строк). Эта рекурсивная функция вместе с начальным условием реализована в листинге 4.11.

Листинг 4.11. Обращение строки *s*

```
def reverse_string(s):
    if s == '':
        return ''
    else:
        return reverse_string(s[1:]) + s[0]
```

4.3.2. Является ли строка палиндромом?

Цель следующей задачи – выяснить, является ли строка палиндромом, то есть цепочкой символов, которая одинаково читается в прямом и обратном направлениях (например, 'радар'). Её размер – длина строки, так как ею определяется количество операций, необходимых для

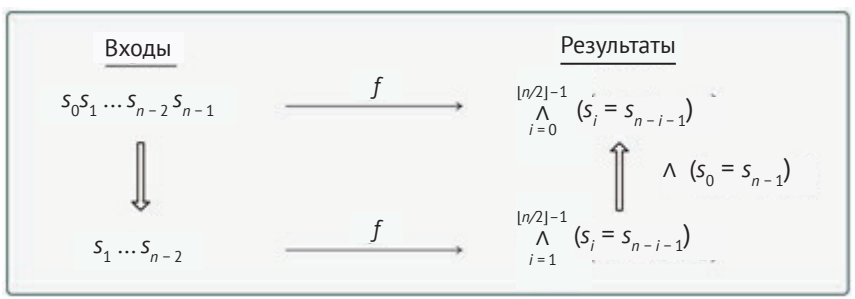
получения результата. У этой задачи два начальных условия: (a) когда строка пуста и (b) когда она состоит из одного символа. В обоих случаях результатом будет True (Истина). Как и для функции из листинга 2.6, здесь необходимо второе начальное условие, поскольку при декомпозиции задачи мы будем уменьшать её размер на две единицы. В данном случае подзадача исключает первый и последний символы исходной строки длиной $n \geqslant 2$. Результатом функции для некоторой строки $s = s_0 s_1 \ldots s_{n-2} s_{n-1}$ длиной n будет выражение:

$$(s_0 = s_{n-1}) \wedge (s_1 = s_{n-2}) \wedge \ldots \wedge (s_{\lfloor n/2 \rfloor - 1} = s_{n-\lfloor n/2 \rfloor}),$$

где $\wedge$ – логическая операция И (конъюнкция), а s_i – символ строки $\boldsymbol{s}$ в позиции i (первый символ – в позиции 0). Заметим, что сумма индексов сопоставляемых символов равна $n - 1$. Кроме того, необходимо сравнить только $\lfloor n/2 \rfloor$ пар символов. Приведённое выражение можно записать короче, подобно сумме:

$$\bigwedge_{i=0}^{\lfloor n/2 \rfloor - 1} (s_i = s_{n-i-1}).$$

Соответствующая рекурсивная схема:



Таким образом, логическая функция может быть определена как:

$$f(s) = \begin{cases} \text{True}, & \text{если } n < 2, \\ (s_0 = s_{n-1}) \wedge f(s_{1\ldots n-2}), & \text{если } n \geq 2, \end{cases}$$

где $s_{1\ldots n-2}$ обозначает подстроку $s_1 \ldots s_{n-2}$.

Наконец, листинг 4.12 представляет код с временной сложностью $\Theta(n)$.

Листинг 4.12. Функция проверки строки на палиндром

```
def is_palindrome(s):
    n = len(s)
    if n <= 1:
        return True
    else:
        return (s[0] == s[n - 1]) and is_palindrome(s[1:n - 1])
```

4.4. Дополнительные задачи

Этот раздел содержит несколько классических задач, которые имеют изящные рекурсивные решения.

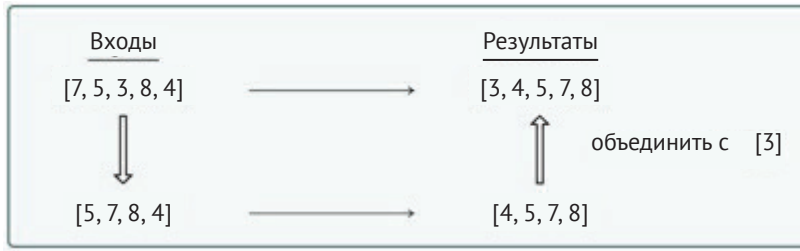
4.4.1. Сортировка выбором

Сортировки выбором – один из самых простых способов сортировки элементов списка. Для заданного списка из n чисел предложить метод его сортировки в порядке возрастания. Алгоритм начинается с поиска и размещения наименьшего элемента списка в первой позиции выходного списка. На втором шаге метод ищет минимальный элемент среди оставшихся 1...(n – 1) элементов и помещает его во вторую позицию списка вывода. Эта процедура выполняется (n – 1) раз, пока в итоге массив не отсортируется. Ясно, что в конце k-го шага наименьшие k чисел уже отсортированы, а после (n – 1) шага будет полностью отсортирован весь список.

Теперь рассмотрим линейно-рекурсивную версию алгоритма. Во-первых, размер задачи – это длина списка (n). Начальное условие выполняется, когда список состоит из одного элемента и не нуждается в сортировке, являясь, очевидно, её результатом. Кроме того, если вход пуст, алгоритм может вернуть ещё и пустой список. Таким образом, для начального условия метод возвращает результат при $n \leqslant 1$.

Для вывода рекурсивного условия декомпозиция должна уменьшать размер задачи на 1, когда наименьший элемент списка исключается из него. Это можно сделать двумя способами. Во-первых, элемент можно менять местами с первым элементом списка. Это, естественно, не меняет результата (сортированного списка), но позволяет после обмена местами исключить первый элемент списка. Поэтому если входной список [**7**, 5, **3**, 8, 4], он превращается в [**3**, 5, 7, 8, 4] после обмена местами 3 и 7. В этом случае входом подзадачи размера n – 1 был бы

список [5, 7, 8, 4], а рекурсивная схема для этого частного примера могла бы быть такой:

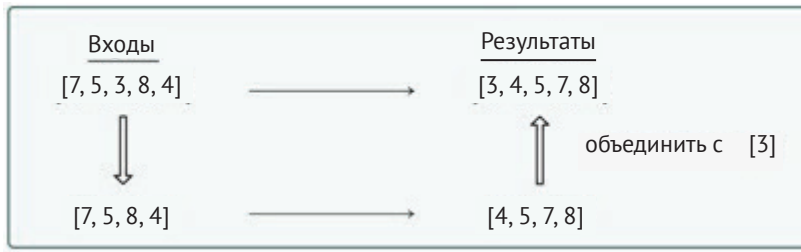


Чтобы решить исходную задачу, рекурсивное правило должно добавить исключённый наименьший элемент ([3]) в начало выходного списка подзадачи (заметьте, что результаты – это сортированные списки). В листинге 4.13 приведена реализация метода с использованием функции min для определения наименьшего значения в списке, метода index, возвращающего позицию элемента в списке, и операции + (конкатенация) для объединения списков. Важной деталью этого алгоритма является то, что он должен создать новую копию входного списка (в строке 5), чтобы не изменить его при вызове метода. В частности, без этой копии метод поменял бы местами первый и наименьший элементы списка.

Листинг 4.13. Рекурсивный алгоритм сортировки выбором

```
1  def select_sort_rec(a):
2      if len(a) <= 1:
3          return a
4      else:
5          b = list(a)
6          min_index = b.index(min(b))
7          aux = b[min_index]
8          b[min_index] = b[0]
9          b[0] = aux
10
11         return [aux] + select_sort_rec(b[1:])
```

Другой способ уменьшения размера задачи – исключение наименьшего элемента прямым удалением его из списка. В этом случае рекурсивная схема была бы такой:



Её единственное отличие от предыдущей схемы – порядок следования элементов во входном параметре подзадачи. Но он совершенно не влияет на рекурсивное правило, где также нужно добавлять удалённый наименьший элемент ([3]) к списку вывода подзадачи. В листинге 4.14 приведена возможная реализация функции, использующая для удаления наименьшего элемента метод remove. В строке 5 также создаётся копия входного списка, чтобы не изменить его очередным вызовом метода.

Листинг 4.14. Другой рекурсивный алгоритм сортировки выбором

```
1  def select_sort_rec_alt(a):
2      if len(a) <= 1:
3          return a
4      else:
5          b = list(a)
6          m = min(b)
7          b.remove(m)
8
9          return [n] + select_sort_rec_alt(b)
```

Наконец, оценка времени выполнения этих алгоритмов задаётся формулой (3.24), так как поиск (и удаление) наименьшего элемента списка требует порядка n операций. Таким образом, алгоритмы работают за время $\Theta(n^2)$.

4.4.2. Схема Горнера

Цель этой задачи – вычислить полином степени n:

$$P(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 \qquad (4.5)$$

для некоторого вещественного значения x. Сумма состоит из степеней x, умноженных на коэффициенты c_i. Простейший алгоритм, вычисляю-

щий каждую степень отдельно, потребовал бы порядка n^2 умножений. Тогда как схема Горнера позволяет сократить число умножений до n. Её идея – представить полином в виде:

$$P(x) = c_0 + x(c_1 + x(c_2 + \ldots + x(c_{n-1} + c_n x))).$$

Ясно, что размер задачи – это степень n полинома. Таким образом, начальное условие выполняется при $n = 0$ с очевидным результатом 0. На практике коэффициенты c должны быть определяющим полином списком (или похожей структурой данных вроде массива) из $n + 1$ элементов. Поэтому начальное условие достигается, когда длина этого списка равна 1.

Чтобы воспользоваться рекурсией, нужно найти подзадачу меньшего размера, подобную исходной. Декомпозиция задачи с уменьшением ее размера на 1:

$$P(x) = \overbrace{c_0 + x\underbrace{(c_1 + x(c_2 + \ldots + x(c_{n-1} + x \cdot c_n))\ldots)}_{\text{подзадача}}}^{\text{полная задача}}.$$

Если пренебречь коэффициентом c_0 и первым умножением на x, то получающаяся подзадача имеет точно такую же структуру, как у исходной задачи. Если $p(c, x)$ – функция, вычисляющая заданный своими коэффициентами c полином от x, то рекурсивная формула будет такой:

$$p(c, x) = c_0 + x \cdot p(c_{1\ldots n}, x),$$

где список $c_{1\ldots n}$ – это тот же список c без первого элемента. Полная рекурсивная функция будет такой:

$$p(c, x) = \begin{cases} c_0, & \text{если } n = 1, \\ c_0 + x \cdot p(c_{1\ldots n}, x), & \text{если } n > 1, \end{cases}$$

а соответствующий ей код приведён в листинге 4.15.

Листинги 4.15. Вычисление полинома по схеме Горнера

```
1  def horner(c, x):
2      if len(c) == 1:
3          return c[0]
4      else:
5          return c[0] + x * horner(c[1:], x)
```

Важно отметить, что уменьшение размера задачи, начиная с c_n вместо c_0, не приводит к схеме Горнера. В этом случае рекурсивная формула была бы такой:

$$p(c, x) = p(c_{0\ldots n-1}, x) + c_n x^n,$$

– и вычисляла бы полином, как в (4.5). Если бы степень x^n вычислялась за линейное или логарифмическое время, то метод потребовал бы $\Theta(n^2)$ или $\Theta(n \log n)$ операций соответственно. Однако схема Горнера более эффективна, так как полином вычисляется за линейное время ($\Theta(n)$).

4.4.3. Треугольник Паскаля

Треугольник Паскаля – треугольное представление биномиальных коэффициентов $\binom{n}{m}$[1], приведённое на рис. 4.2, где треугольник (a) содержит сами биномиальные коэффициенты, а треугольник (b) – их фактические целочисленные значения.

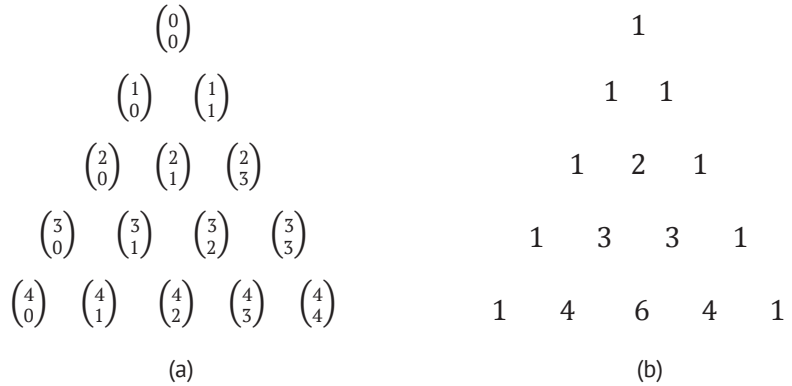


Рис. 4.2. Треугольник Паскаля

Очередная наша задача – определить список биномиальных коэффициентов n-й строки:

$$\binom{n}{0}\binom{n}{1}\ldots\binom{n}{n-1}\binom{n}{n},$$

где первые и последние элементы всегда равны 1. Задачу можно рассмотреть с рекурсивной точки зрения, принимая во внимание определение (3.2). Графически это означает, что биномиальный коэффициент – это сумма пары коэффициентов, расположенных непосредственно над ним в предыдущей строке треугольника Паскаля. Например, $\binom{4}{3} = \binom{3}{2} + \binom{3}{3}$. Поэтому строку треугольника Паскаля можно вычислить по значени-

[1] $\binom{n}{m}$ – иное обозначение C_n^m. – *Прим. перев.*

ям предыдущей строки. На рис. 4.3 показано отношение между задачей (n-й строкой) и подзадачей ((n – 1)-й строкой).

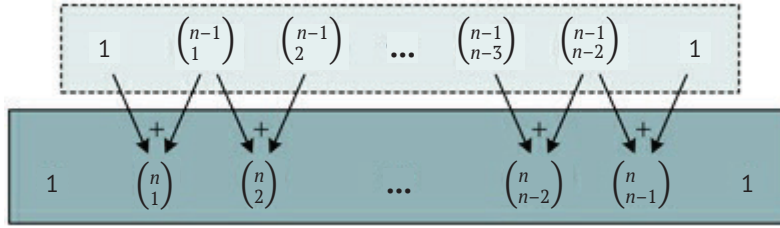


Рис. 4.3. Декомпозиция и восстановление строки треугольника Паскаля

Размер задачи – n. Таким образом, начальное условие соответствует $n = 0$, когда результат – это просто список, содержащий 1 ([1]). Для $n = 1$ нельзя применить рекурсивное правило (3.2). Поэтому поначалу кажется, что необходимо дополнительное начальное условие для $n = 1$ с результатом ([1,1]). Но поскольку все строки для $n > 0$ начинаются и заканчиваются с 1, эти элементы можно включить в рекурсивное условие по умолчанию. Таким образом, особое начальное условие для $n = 1$ становится излишним.

Листинг 4.16. Функция, генерирующая n-ю строку треугольника Паскаля

```
1  def pascal(n):
2      if n == 0:
3          return [1]
4      else:
5          row = [1]
6          previous_row = pascal(n - 1)
7          for i in range(len(previous_row) - 1):
8              row.append(previous_row[i] + previous_row[i + 1]
9          row.append(1)
10         return row
```

В частности, предполагается, что в рекурсивном условии известно решение подзадачи, вычисляющей все элементы строки, кроме крайних (их можно просто добавлять, не вычисляя). В листинге 4.16 приведено возможное решение задачи. Рекурсивное условие начинается с добавления 1 в итоговый список (строку). Строка 6 вычисляет результат подзадачи размера n – 1, а в строках 7 и 8 в цикле **for** пары целых чисел подзадачи складываются (как показано на рис. 4.3) и их сумма добавляется к результату. В заключение в конец строки добавляется 1, на чём

и заканчивается решение. В упражнении 4.14 предлагается заменить цикл рекурсивной функцией.

4.4.4. Резистивная цепь

Цель следующей задачи – упростить электрическую цепь на рис. 4.4, напоминающую многозвенную схему, содержащую несколько уровней резисторов, сопротивление которых равно r.[2] В частности, можно заменить всю цепь эквивалентной, состоящей из одного резистора.

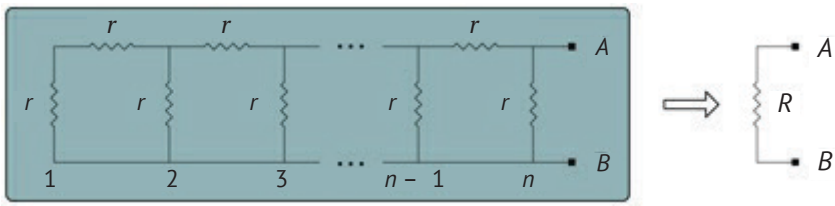


Рис. 4.4. Задача о резистивной цепи

Таким образом, задача состоит в определении значения сопротивления R (как функции сопротивления r), приводящего к эквивалентной цепи. На рис. 4.5 показано, как преобразовать цепь из двух резисторов с сопротивлениями r_1 и r_2, если они соединены последовательно (a) или параллельно (b).

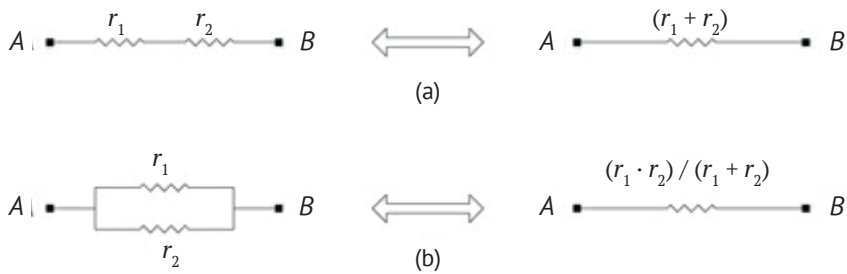


Рис. 4.5. Эквивалентность резисторных соединений

При последовательном соединении получающееся сопротивление равно $r_1 + r_2$, а при параллельном новое сопротивление $r = (r_1 \cdot r_2)/(r_1 + r_2)$, или иначе:

$$1/r = 1/r_1 + 1/r_2. \qquad (4.6)$$

[2] Такое соединение называется ещё резисторной схемой лестничного типа или резисторной «лестницей». – *Прим. перев.*

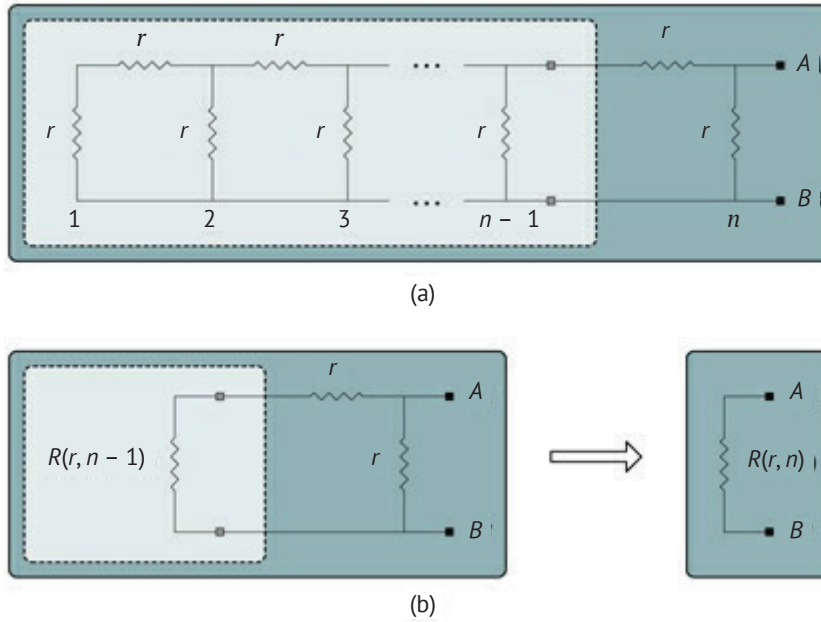


Рис. 4.6. Декомпозиция задачи о резистивной цепи и вывод рекурсивного условия методом индукции

Эти правила можно применять последовательно к парам резисторов, пока не получится цепь из одного резистора. Однако данный процесс утомителен, и вместо него предлагается короткое и изящное рекурсивное решение задачи.

Задача имеет два входных параметра: сопротивление r и число звеньев n в цепи. Очевидно, размер задачи – n (r влияет на выходное значение задачи (результат), но не влияет на время выполнения алгоритма). Пусть $R(r, n)$ обозначает рекурсивную функцию. Начальное условие выполняется при $n = 1$, когда начальная цепь состоит только из одного резистора. Значит, в этом случае $R(r, 1) = r$. Для вывода рекурсивного условия нужно отыскать в исходной задаче подзадачу с точно такой же структурой. Рисунок 4.6(a) показывает декомпозицию задачи с уменьшением её размера на 1. Цепь, соответствующая подзадаче, может быть заменена единственным резистором с сопротивлением $R(r, n - 1)$, как показано на рис. 4.6(b), когда на основании метода индукции можно считать её значение известным. Наконец, легко упростить получающуюся цепь, состоящую всего из трёх резисторов. Во-первых, левый и верхний резисторы соединены последовательно. Значит, их можно объединить в один резистор с сопротивлением $R(r, n - 1) + r$. Наконец, этот

новый резистор соединён параллельно с правым резистором. Применяя (4.6), $R(r, n)$ можно определить как:

$$1/R(r, n) = 1/r + 1/(R(r, n - 1) + r).$$

Таким образом, рекурсивная функция:

$$R(r, n) = \begin{cases} r, & \text{если } n = 1, \\ 1/(1/r + 1/(R(r, n-1) + r)), & \text{если } n > 1. \end{cases}$$

В листинге 4.17 приведён соответствующий код. И наконец, любопытное замечание: можно показать, что $R(r, n) = r\,F(2n - 1) / F(2n)$, где F – функция Фибоначчи.

Листинг 4.17. Функция, решающая задачу о резистивной цепи

```
def circuit(n, r):
    if n == 1:
        return r
    else:
        return 1 / (1 / r + 1 / (circuit(n - 1, r) + r))
```

4.5. Упражнения

Упражнение 4.1. Листинг 2.6 содержит линейно-рекурсивную логическую функцию, определяющую, является ли неотрицательное целое число n чётным, когда декомпозиция уменьшает размер задачи (n) на два. Определите и закодируйте альтернативный метод с уменьшением размера задачи на 1.

Упражнение 4.2. Реализуйте рекурсивную функцию вычисления b^n за логарифмическое время для вещественного основания b и целой степени n, которая может быть отрицательной.

Упражнение 4.3. Реализуйте рекурсивную функцию вычисления n-й степени квадратной матрицы. Результатом вычисления будет другая квадратная матрица той же размерности. Используйте пакет NumPy и его методы:

- `identity`: возвращает единичную матрицу;
- `shape`: определяет размерность матрицы;
- `dot`: умножает матрицы.

Наконец, вычислите матрицы:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

для $n = 1, \ldots, 10$ и напечатайте элементы их первой строки и второго столбца. Каковы эти числа?

Упражнение 4.4. Реализуйте рекурсивные функции вычисления «медленного» произведения двух неотрицательных целых чисел m и n. Алгоритму разрешено добавлять и вычитать числа, но запрещено использовать операцию умножения (*). Примените декомпозиции с уменьшением одного или обоих входных параметров на 1. Кроме того, изобразите схему осмысления рекурсивного процесса разработки задачи. В качестве примера воспользуйтесь общей схемой на рис. 2.5 или схемой вычисления произведения n и m как площади прямоугольника высотой n и шириной m, где результат – это количество единичных клеток внутри прямоугольника $n \times m$, как на рис. 4.7.

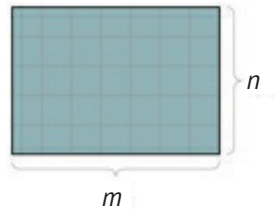


Рис. 4.7. Произведение двух неотрицательных целых чисел n и m как площадь прямоугольника размером $n \times m$

Упражнение 4.5. Реализуйте рекурсивную функцию более эффективного «медленного» произведения, применив декомпозицию делением обоих входных параметров пополам. Используйте прямоугольные схемы на рис. 4.7.

Упражнение 4.6. В языке Python, как и в других языках программирования, метод может быть параметром другого метода. Определите и закодируйте общую рекурсивную функцию вычисления суммы:

$$g(m, n, f) = \sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \ldots + f(n-1) + f(n),$$

где m и n – целые числа, а f – функция. Используйте её для вычисления и печати результата функции:

$$\sum_{i=1}^{n} i^3,$$

для $n = 0, \ldots, 4$.

Упражнение 4.7. Определите и закодируйте функцию вычисления количества цифр в неотрицательном целом числе n.

Упражнение 4.8. Определите и закодируйте функцию, которая для заданного десятичного числа n, состоящего только из нулей и единиц, возвращает число, двоичное представление которого совпадает с десятичным представлением исходного числа n. Например, если $n = 10110_{10}$, то функция должна вернуть 22, поскольку $10110_2 = 22$.

Упражнение 4.9. Выведите рекурсивное условие для задачи «преобразование десятичного числа к основанию b» из раздела 4.2.2 с применением общей схемы осмысления рекурсивных условий на рис. 2.5.

Упражнение 4.10. Вспомните задачу, которая обрабатывает (двоичные) числа n (справа налево). Выведите и закодируйте рекурсивный алгоритм, который для конкретного числа x из n битов определяет позицию его младшего значащего единичного бита. В частности, учтите, что младший бит может быть в первой позиции. Например, для $x = 01110\underline{1}00$ младшая (самая правая) единица находится в позиции 3.

Упражнение 4.11. Напишите рекурсивный метод решения упражнения 3.8, использующий в своих вычислениях функцию из упражнения 4.10. Напечатайте результаты для чисел из $n = 1, \ldots, 5$ битов.

Упражнение 4.12. Закодируйте рекурсивную функцию, возвращающую количество гласных в заданной строке.

Упражнение 4.13. Биномиальный коэффициент $\binom{n}{m}$ – функция двух целочисленных параметров n и m. Определите для вычисления биномиального коэффициента три линейно-рекурсивные функции, используя три декомпозиции: уменьшающую n, уменьшающую m и уменьшающую оба параметра. Используйте определение (3.1) для вывода рекурсивных условий.

Упражнение 4.14. Замените цикл в листинге 4.16 рекурсивной функцией. Она должна получать строку треугольника Паскаля и возвращать строку с суммами соответствующих членов исходной строки. Например, если вход – [1, 3, 3, 1], то результат должен быть [4, 6, 4].

Упражнение 4.15. Рассмотрим неотрицательное целое число n, чьи цифры всегда расположены в неубывающем порядке слева направо, на-

пример 24667. Иными словами, если $d_{m-1} \ldots d_1\ d_0$ – последовательность из (m) цифр числа n, то $d_i \leqslant d_j$, если $i < j$. Пусть задана дополнительная цифра $0 \leqslant x \leqslant 9$. Напишите функцию, которая возвращает такое целое число со вставленной в n цифрой x, что его цифры тоже расположены в неубывающем порядке слева направо. Например, если $n = 24667$ и $x = 5$, функция должна вернуть 24**5**667. Постарайтесь избежать избыточных начальных условий.

Упражнение 4.16. Это упражнение подобно упражнению 4.15. Заданы сортированный в порядке возрастания список чисел (**a**) и число x. Реализуйте функцию вставки x в список без нарушения его сортировки.

Рис. 4.8. Шаг алгоритма сортировки вставкой

Упражнение 4.17. Метод алгоритма «сортировки вставкой» сортирует список многократным применением процедуры на рис. 4.8. На i-м шаге элементы с индексами от 0 до $i - 1$ уже отсортированы, тогда как остальная часть списка не (обязательно) отсортирована. Для продолжения процесса метод вставляет элемент с индексом i в сортированный подсписок так, чтобы получающийся список (с индексами от 0 до i) оставался отсортированным. Если эта операция выполняется от индекса (шага) 1 (или 0) до $n - 1$, то список останется отсортированным после последнего шага. Реализуйте рекурсивный вариант метода, который для сортировки списка использует решение упражнения 4.16.

Глава 5

Линейная рекурсия II: хвостовая рекурсия

Вы хороши лишь настолько, насколько хорош ваш последний поступок.

– Стивен Ричардс

Хвостовая линейная рекурсия, или «концевая» рекурсия, – это разновидность линейной рекурсии, когда рекурсивные условия вызывают метод только один раз. Но в отличие от обычной линейной рекурсии хвостовой рекурсивный вызов – это последнее выполняемое методом действие. Например, функция (1.15) относится к хвостовой рекурсии, так как в рекурсивном условии она ничего не делает с результатом вызова функции. Это значит, что она вернёт значение, полученное непосредственно в начальном условии. Кроме того, такие функции могут требовать большего количества параметров для сохранения промежуточных результатов, которые будут использоваться в начальных условиях для получения конечного результата. Хвостовые рекурсивные методы встречались ранее в листингах 2.3, 2.5 и 2.6.

В предыдущей главе все методы, за исключением методов из раздела 4.1.1.2, уменьшали размер задачи на 1 или 2. Однако в этой главе представлены в основном алгоритмы, которые делят размер задачи пополам. В этом случае рекурсивные вызовы решают одну задачу в половину размера исходной, так как методы вызывают себя лишь однажды. Некоторые авторы относят эту стратегию к методу «разделяй и властвуй». Однако мы будем использовать этот термин только для тех алгоритмов, которые должны будут решить *несколько* независимых подзадач, размеры которых – лишь часть исходной задачи. Такие

алгоритмы представлены в главе 11. Наконец, особенность хвостовой рекурсии в том, что она тесно связана с итерацией, и в этой главе мы рассмотрим эту связь.

5.1. Логические функции

Этот вводный раздел исследует, как выбор начальных условий может привести к линейным или хвостовым рекурсивным алгоритмам для некоторых логических (булевых) функций.

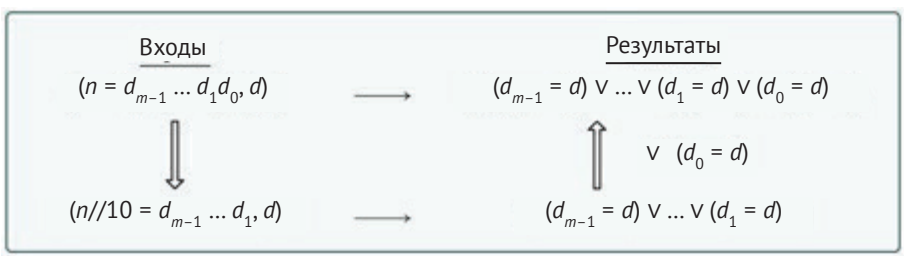
5.1.1. Есть ли в неотрицательном целом числе заданная цифра?

Дано неотрицательное целое число n и некоторая (тоже целая) цифра d от 0 до 9. Цель задачи – определить, содержит ли n цифру d. Таким образом, рекурсивным методом будет логическая функция с параметрами n и d. Кроме того, будем считать, что n не может начинаться с цифры 0 (например, 358 не может быть записано как 0358), за исключением случая $n = 0$.

5.1.1.1. *Линейно-рекурсивный алгоритм*

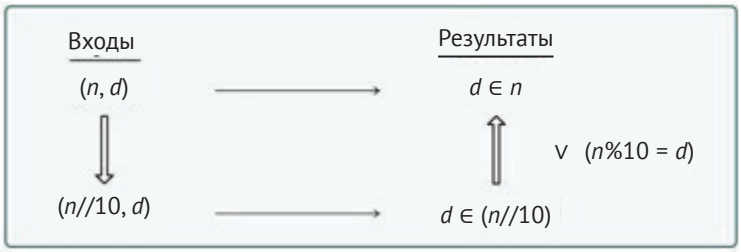
Первый шаг заключается в определении размера задачи, который, очевидно, равен количеству m цифр исходного числа n. Начальное условие выполняется, когда n состоит из одной цифры (то есть $n < 10$), а его результат – это результат сравнения ($n = d$). Допустим пока, что это – единственное начальное условие.

Рекурсивное условие будет довольно громоздким, если выражать значение каждой цифры через значение n. Поэтому можно применить иное обозначение, обсуждавшееся в разделе 2.5.3. В частности, пусть $d_{m-1} \ldots d_1 d_0$ – последовательность из m цифр числа n по основанию 10. Тогда можно создать следующую схему с уменьшением размера задачи на 1 путём деления исходного числа n на 10 с отбрасыванием его наименьшей значащей цифры:



где ∨ обозначает логическую операцию ИЛИ (дизъюнкцию). Из схемы ясно видно, что если d – цифра n, то она либо входит в $n//10$ (то есть в результат рекурсивного вызова), либо равна d_0.

В схеме допускаются и другие обозначения с использованием слов или выражений, не вполне корректных с логической или математической точки зрения. Например, можно воспользоваться явно неподходящим здесь символом принадлежности ∈, который применяется только к множествам. Так, $d \in n$ на самом деле будет означать принадлежность цифры d множеству цифр числа n. Несмотря на это, схема выполняет свое назначение – точно определить рекурсивное условие функции.



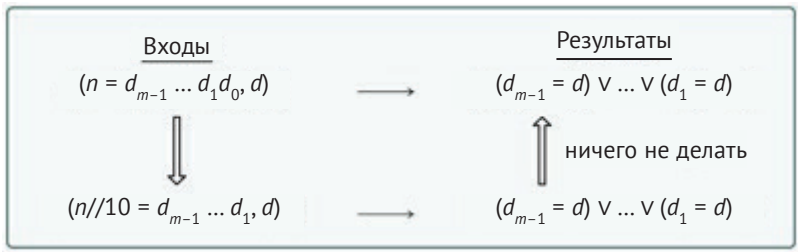
В листинге 5.1 приведён код, реализующий искомую функцию, где n%10 – наименьшая значащая цифра n. Заметьте, что метод – линейно-рекурсивный, так как функция вызывает себя лишь раз и должна обработать результат подзадачи (то есть вычислить результат логического выражения с операцией `or`). Кроме того, если условие (n%10 == d) истинно, то компилятор или интерпретатор с «закорачиванием» (short-circuit) автоматически возвращал бы значение True, не вызывая `contains_digit(n//10, d)`. В частности, использование «закорачивания» позволяет избегать ненужных вычислений согласно свойствам логических операций: True ∨ b = True и False ∧ b = False, где b – некоторое логическое выражение. Поскольку в этих случаях результат операции не зависит от значения b, то b можно не вычислять вообще.

Листинг 5.1. Линейно-рекурсивная логическая функция, определяющая наличие в неотрицательном целом числе n заданной цифры d

```
1  def contains_digit(n, d):
2      if n < 10:
3          return n == d
4      else:
5          return (n % 10 == d) or contains_digit(n // 10, d)
```

5.1.1.2. *Алгоритм хвостовой рекурсии*

Большинство языков программирования поддерживает так называемое «закорачивание». Если его реализовать в дополнительном начальном условии, то это приведёт к хвостовому рекурсивному алгоритму. В частности, для этой задачи алгоритм может вернуть True, как только обнаружит цифру d в числе n. Поэтому можно вернуть True, если последняя цифра n равна d (то есть если $n\%10 = d$). Взглянув на это начальное условие, можно убедиться, что в рекурсивном условии $n\%10 = d_0 \neq d$. Поэтому рекурсивная схема теперь будет такой:



Она подразумевает, что результат подзадачи – это в точности результат исходной задачи и функция может вернуть результат рекурсивного вызова, не обрабатывая его. Вместе с начальными условиями она приводит к хвостовому рекурсивному алгоритму, который можно закодировать, как показано в листинге 5.2.

Листинг 5.2. Хвостовая рекурсивная логическая функция, определяющая, содержит ли неотрицательное целое число n цифру d

```
1  def contains_digit_tail(n, d):
2      if n < 10:
3          return n == d
4      elif n % 10 == d:
5          return True
6      else:
7          return contains_digit_tail(n // 10, d)
```

Важно понимать, что хотя выбранная декомпозиция делит входной параметр n на константу 10, размер задачи уменьшается всего на 1. Временна́я сложность алгоритма является логарифмом от числа n, но линейно зависит от количества его цифр m.

Наконец, эта задача имеет аналогии, опирающиеся на такие структуры данных, как списки, массивы и т. д., поскольку они тоже представ-

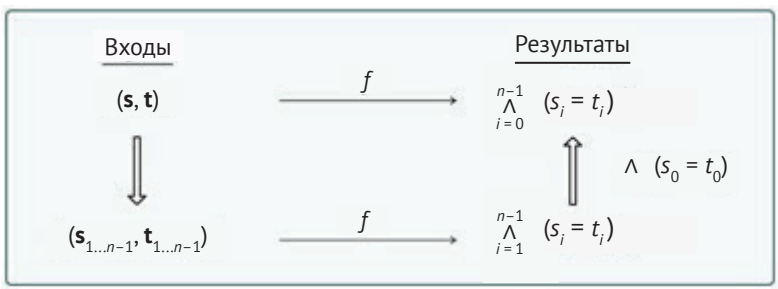
ляют собой последовательности элементов. Например, эта задача очень похожа на задачу о принадлежности некоторого символа строке или некоторого элемента списку. Хотя коды могут быть разными, но основное рассуждение, по сути, то же.

5.1.2. Равны ли строки?

Следующая задача – сравнение двух строк (конечно, в языке Python мы можем просто воспользоваться операцией ==). Логическая функция, решающая эту задачу, будет иметь два строковых входных параметра. Если их длины различны, алгоритм немедленно вернёт False уже в начальном условии. Таким образом, сравнивать строки имеет смысл при одинаковой их длине, которая и является размером задачи.

5.1.2.1. *Линейно-рекурсивный алгоритм*

Наименьший экземпляр задачи – когда обе строки пусты и результат, очевидно, равен True. В этом разделе мы предположим, что никаких дополнительных начальных условий, приводящих к линейно-рекурсивной функции, нет. Применим декомпозицию с уменьшением размера задачи на 1. Она сводится к поштучному отбрасыванию совпавших символов (в одной и той же позиции) в обеих входных строках. Отбрасывание первых символов приводит к следующей рекурсивной схеме, где **s** и **t** – две входные строки длиной n:



Ясно, что метод должен проверить совпадение первых символов u, оставшихся подстрок посредством рекурсивного вызова. Следовательно, логическая функция будет такой:

$$f(s,t) = \begin{cases} \text{False}, & \text{если } length(s) \neq length(t), \\ \text{True}, & \text{если } n = 0, \\ (s_0 = t_0) \wedge f(s_{1\ldots n-1}, t_{1\ldots n-1}), & \text{если } n > 0. \end{cases}$$

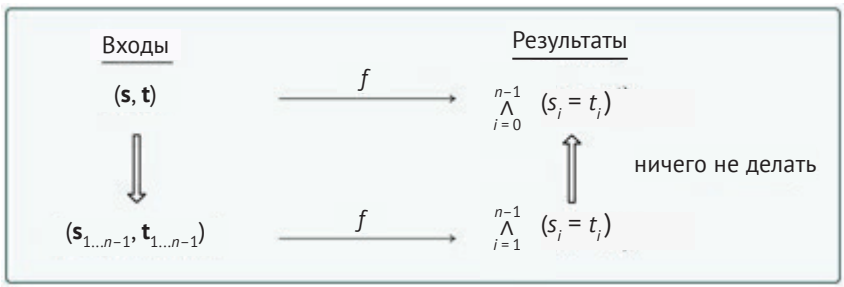
В листинге 5.3 приводится соответствующий код линейно-рекурсивной функции.

Листинг 5.3. Линейно-рекурсивная функция, сопоставляющая две строки

```
def equal_strings(s, t):
    if len(s) != len(t):
        return False
    elif s == '':
        return True
    else:
        return s[0] == t[0] and equal_strings(s[1:], t[1:])
```

5.1.2.2. *Алгоритм хвостовой рекурсии*

На примере раздела 5.1.1 можно учесть «короткое замыкание», добавив дополнительное начальное условие. В частности, алгоритм может сразу вернуть False при обнаружении первого несовпадения символов в одной и той же позиции строк. Таким образом, можно добавить начальное условие с проверкой $s_0 \neq t_0$. В этом случае алгоритм мог бы сразу вернуть False. Такое начальное условие гарантирует, что в рекурсивном условии $s_0 = t_0$. Таким образом, новая рекурсивная схема будет такой:



И опять результат подзадачи является в точности результатом исходной задачи, которая приводит к хвостовому рекурсивному алгоритму. В итоге вместе с начальными условиями функция может быть закодирована, как в листинге 5.4.

5.2. Алгоритмы поиска в списке

Этот раздел посвящён алгоритмам, решающим классическую задачу поиска позиции i элемента x в списке $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$. Другими слова-

ми, для заданных **a** и x найти индекс i, для которого $a_i = x$. Если элемент x не единственный в списке, то метод решения задачи может вернуть любой индекс, указывающий на позицию x в списке.

Для списка длины n метод решения задачи должен вернуть целочисленный индекс в пределах от 0 до $n - 1$, если элемент x найден в списке. Напротив, если список не содержит x, алгоритм должен вернуть некоторое значение за пределами этого интервала. Некоторые реализации Python просто возвращают логическое значение False. Однако такой возможности нет во многих других языках (C, Java, Паскаль и т. д.), где методы могут возвращать только значения одного типа данных. Поскольку индексы – это, безусловно, целые числа, в этих языках результат должен быть также целым числом. Таким образом, ради совместимости с другими языками программирования мы будем использовать целочисленное значение –1 в случае, когда элемент не найден в списке.

Листинг 5.4. Функция с хвостовой рекурсией, сопоставляющая две строки

```
1  def equal_strings_tail(s, t):
2      if len(s) != len(t):
3          return False
4      elif s == '':
5          return True
6      elif s[0] != t[0]:
7          return False
8      else:
9          return equal_strings_tail(s[1:], t[1:])
```

5.2.1. Линейный поиск

Сначала допустим, что список не (обязательно) сортирован. Любой алгоритм потребовал бы в худшем случае, когда элемент вообще не принадлежит списку, n сравнений. Отметим, что любой метод должен проверить каждый элемент списка, чтобы удостовериться, что его нет в списке. В этом случае алгоритм осуществляет «линейный поиск» элемента (последовательный перебор) в списке, время выполнения которого $O(n)$.

Ясно, что размер задачи – n. Если список пуст, метод может просто вернуть –1. Другое начальное условие может вернуть позицию элемента, если он найден. Это начальное условие будет зависеть от способа де-

композиции. Если уменьшать размер задачи на 1, отбрасывая последний элемент списка, то начальное условие должно проверить, находится ли x в последней позиции. Если действительно $a_{n-1} = x$, то метод может просто вернуть $n - 1$. Иначе будет выполняться рекурсивный вызов для решения подзадачи размера $n - 1$, а её результатом будет именно окончательный результат подзадачи, которая приводит к хвостовому рекурсивному решению. В листинге 5.5 приводится возможная реализация описанной функции линейного поиска.

Листинг 5.5. Хвостовая рекурсия при линейном поиске элемента в списке

```
def linear_search_tail(a, x):
    n = len(a)
    if a == []:
        return -1
    elif a[n - 1] == x:
        return n - 1
    else:
        return linear_search_tail(a[:n - 1], x)
```

В заключение стоит отметить, что в предыдущей декомпозиции позиции элементов подсписка совпадают с позициями исходного списка. Но если бы декомпозиция отбрасывала первый элемент списка, то все индексы в подсписке были бы на 1 меньше, чем в исходном. Например, если список – [2, 8, 5], то в первом случае элемент 8 в подсписке [2, 8] тоже был бы в позиции 1, а во втором он находился бы в позиции 0 в подсписке [8, 5]. Это приводит к более сложным методам, которые требуют дополнительного параметра. Например, в листинге 5.6 приведено решение, добавляющее 1 к результату каждого рекурсивного вызова. В тривиальном случае, когда $x = a_0$, метод возвращает 0, но для пустого списка начальное условие сложнее. Отметим, что по достижении этого начального условия алгоритм уже добавил n единиц в предыдущих n рекурсивных вызовах. Таким образом, он должен вернуть $n - 1$, где n – длина исходного списка (но не длину входного параметра, так как в этот момент он равен 0), чтобы в итоге вернуть –1, поскольку x не найден. Так как n не может быть получен из пустого списка, его нужно передавать дополнительным параметром каждому вызову функции, чтобы восстановить в начальном условии. По этой причине код требует охватывающего метода, когда третьим параметром рекурсивной функции является n.

Листинг 5.6. Линейно-рекурсивный линейный поиск элемента в списке

```
def linear_search_linear(a, x, n):
    if a == []:
        return -n - 1
    elif a[0] == x:
        return 0
    else:
        return 1 + linear_search_linear(a[1:], x, n)


def linear_search_linear_wrapper(a, x):
        return linear_search_linear(a, x, len(a))
```

В последней декомпозиции алгоритм начинает поиск элемента с индекса 0, постепенно достигая индекса $n - 1$. Другое решение заключается в явном указании индекса элемента, который будет каждый раз сравниваться с *x*, в рекурсивном вызове. Для этого вводится новый параметр-сумматор, который увеличивается при каждом вызове функции и приводит к хвостовой рекурсии в листинге 5.7, возвращающей правильное значение –1 для пустого списка. Функция должна вызываться с параметром-сумматором индекса, равным 0, из охватывающего метода-оболочки. Наконец, обратите внимание, что результат хранится именно в этом параметре.

Листинг 5.7. Альтернативный хвостовой рекурсивный линейный поиск элемента в списке

```
def linear_search_tail_alt(a, x, index):
    if a == []:
        return -1
    elif a[0] == x:
        return index
    else:
        return linear_search_tail_alt(a[1:], x, index + 1)


def linear_search_alt_tail_wrapper(a, x):
        return linear_search_tail_alt(a, x, 0)
```

5.2.2. Двоичный поиск в сортированном списке

Если исходный список **a** уже отсортирован, то найти в нём нужный элемент можно гораздо быстрее. Пусть список отсортирован в порядке

возрастания, то есть $a_i \leqslant a_{i+1}$ для $i = 0, \ldots, n-2$. Задача может быть решена за время $O(\log n)$ алгоритмами «двоичного поиска». Основная идея таких методов (а они могут быть разными) заключается в последовательном делении размера задачи пополам. Рисунок 5.1 иллюстрирует декомпозицию задачи.

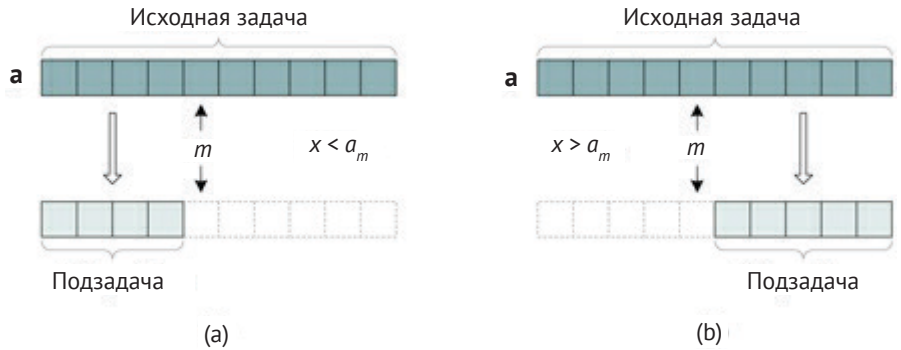


Рис. 5.1. Декомпозиция для алгоритмов двоичного поиска

Сначала алгоритмы вычисляют средний индекс m. Если $x = a_m$, они заканчивают работу в начальном условии, возвращая m. Иначе, поскольку список отсортирован, можно продолжить поиск x в одном из двух подсписков – справа или слева от m. В частности, если $x < a_m$, то x не может оказаться в правом подсписке $[a_m, \ldots, a_{n-1}]$, и методы могут продолжить поиск x в левом подсписке $[a_0, \ldots, a_{m-1}]$, как показано на рис. 5.1(a). Аналогично, если $x > a_m$, то x может появиться только в $[a_{m+1}, \ldots, a_{n-1}]$, как показано на рис. 5.1(b). В сущности, этот подход делит размер задачи (n) пополам. Поэтому время выполнения будет определяться формулой (3.20), то есть в самом худшем случае методы будут выполняться за логарифмическое время от n. Наконец, функции возвращают –1 (в начальном условии), если список пуст.

Подобно функциям линейного поиска в листингах 5.6 и 5.7, методы двоичного поиска должны включать дополнительные входные параметры для указания индексов исходного списка. Например, рассмотрим поиск 8 в списке [1, 3, 3, 5, 6, 8, 9]. Если в рекурсивном вызове функции мы передаём ей только подсписок [6, 8, 9], то в нём 8 теперь имеет индекс 1, тогда как в исходном списке значение 8 находится в позиции 5. Таким образом, нужно указать, что первый индекс подсписка должен быть именно 5. Другими словами, нужно определить дополнительный параметр – начальную позицию подсписка в исходном списке. В листинге 5.8 приведена реализация метода, который включает не только

этот параметр (`lower`), но и ещё один параметр (`upper`) – конечный индекс подсписка в исходном списке. Таким образом, `lower` и `upper` просто задают границы подсписка внутри **a**. Параметр `upper` в Python не обязателен, поскольку для определения верхнего индекса подсписка надо использовать длину списка (упражнение 5.4 предлагает реализовать метод, использующий только параметр `lower`). Однако код в листинге 5.8 проще в том смысле, что требует меньше арифметических операций, не передаёт сам себе подсписок (рекурсивные вызовы используют весь список **a**) и показывает, как можно реализовать метод в других языках программирования, где длина списков не доступна напрямую. В частности, если `lower` больше upper, то список пуст, а метод возвращает –1. В противном случае функция проверяет начальное условие $x = a_m$. Если x оно неверно, метод выполняет один из двух рекурсивных вызовов (с соответствующим подсписком) в зависимости от условия $x < a_m$. Наконец, метод-оболочка используется для инициализации индексов `lower` и `upper` значениями 0 и n – 1 соответственно.

Листинг 5.8. Двоичный поиск элемента в списке

```
def binary_search(a, x, lower, upper):
    if lower > upper:     # empty list
        return -1
    else:
        middle = (lower + upper) // 2

        if x == a[middle]:
            return middle
        elif x < a[middle]:
            return binary_search(a, x, lower, middle - 1)
        else:
            return binary_search(a, x, middle + 1, upper)


def binary_search(a, x):
        return binary_search(a, x, 0, len(a) - 1)
```

5.3. Двоичные деревья поиска

Двоичные (бинарные) деревья поиска – это структура данных, используемая для хранения элементов данных, каждому из которых назначен определенный ключ (будем считать ключи уникальными), подобно вхо-

дам в словаре языка Python. Это – важная структура данных, позволяющая эффективно выполнять такие операции, как поиск, вставка или удаление элементов. Данные в каждом узле двоичного дерева – это пара (ключ, элемент), а его узлы упорядочены по значению ключей, которые можно сравнивать операцией < или некоторой равнозначной ей функцией (то есть полное бинарное отношение предшествования). В частности, для любого узла дерева каждый ключ левого поддерева меньше ключа этого узла, тогда как все ключи правого поддерева больше ключа этого узла. Это основное «свойство двоичного дерева поиска», подразумевающее, что каждое поддерево двоичного дерева поиска – тоже двоичное дерево поиска.

Например, эта структура данных может использоваться для хранения информации о днях рождения. На рис. 5.2 приведено двоичное дерево, позволяющее находить дни рождения семи человек по их именам-строкам, которые мы будем считать уникальными (для уникальной идентификации каждого человека можно, конечно, добавить ещё фамилии или прозвища). Ключи могут быть строками, поскольку их тоже можно сравнивать, например, в лексикографическом порядке (в Python можно просто воспользоваться операцией <). Поэтому имена в двоичном дереве поиска отсортированы так же, как в обычном словаре. Например, если взять корневой узел «Emma», то можно заметить, что все имена в его левом поддереве располагаются в словаре до него, а все имена в правом поддереве – после него. Более того, этим свойством обладают все узлы двоичного дерева поиска.

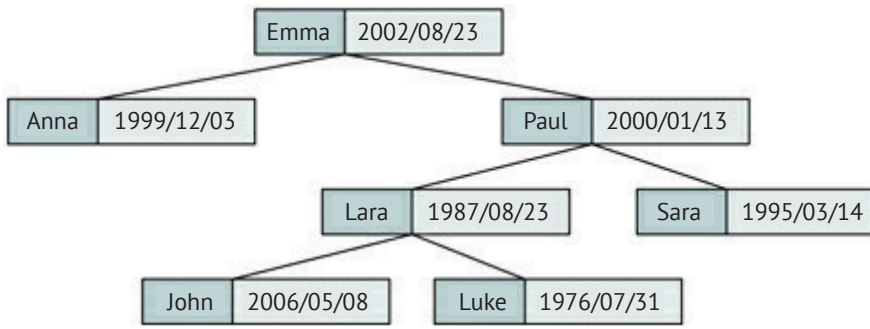


Рис. 5.2. Двоичное дерево поиска с информацией о днях рождения

Прежде чем реализовать двоичное дерево поиска, нужно решить, как его представить. Существует несколько способов (один из наиболее общих подходов – воспользоваться объектно-ориентированными возможностями, определив, например, класс «узел дерева»), но в этой кни-

ге мы будем использовать обычные списки. В частности, каждый узел дерева будет списком из четырех элементов: ключа, элемента, левого и правого поддеревьев, где поддеревья – это тоже двоичные деревья. Таким образом, двоичное дерево на рис. 5.2 представлялось бы следующим списком:

```
[Emma, 2002/08/23, [Anna, 1999/12/03, [], []], [Paul, 2000/01/13,
 [Lara, 1987/08/23, [John, 2006/05/08, [], []],
[Luke, 1976/07/31, [], []]], [Sara, 1995/03/14, [], []]],                (5.1)
```

изображённым на рис. 5.3. Обратите внимание, что для всех листовых узлов их левые и правые поддеревья – пустые списки.

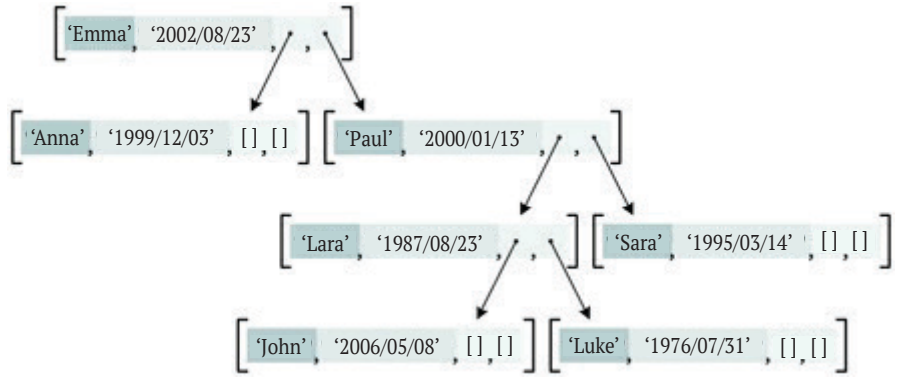


Рис. 5.3. Двоичное дерево поиска (см. рис. 5.2 и 5.1), где каждый узел – список из четырёх элементов: имени (строка), дня рождения (строка), левого поддерева (список) и правого поддерева (список)

5.3.1. Поиск элемента

Цель следующей задачи – найти в двоичном дереве поиска элемент с заданным ключом *k* и получить его значение. Можно предположить, что размер задачи – высота дерева. Тривиальное начальное условие выполняется, когда список, представляющий двоичное дерево, пуст. В этом случае алгоритм может просто вернуть None. Есть другая ситуация, когда алгоритм может выдать результат, не выполняя рекурсивного вызова. Если ключ корневого узла равен *k*, то элемент сразу же найден и является результатом решения задачи. Таким образом, в рекурсивных условиях нужно убедиться, что корневой узел не содержит искомого элемента.

Цель следующего нашего шага – найти соответствующую декомпозицию задачи, уменьшающую ее размер. Нам уже известно, что деревья составляются рекурсивно из поддеревьев. Таким образом, нужно про-

извести поиск элемента в двух поддеревьях двоичного дерева. Это гарантирует уменьшение размера задачи на 1, так как корневой узел уже проверен. Однако легко видеть, что в этой задаче можно также избежать поиска во всем поддереве. Если ключ k меньше ключа корневого узла (k_{root}), то понятно, что согласно свойству двоичного дерева поиска искомого элемента не может быть в правом поддереве. Аналогично, если $k > k_{root}$, элемента не может быть в левом поддереве. Рисунок 5.4 иллюстрирует эту идею, где x_{root} – элемент, сохранённый в корневом узле.

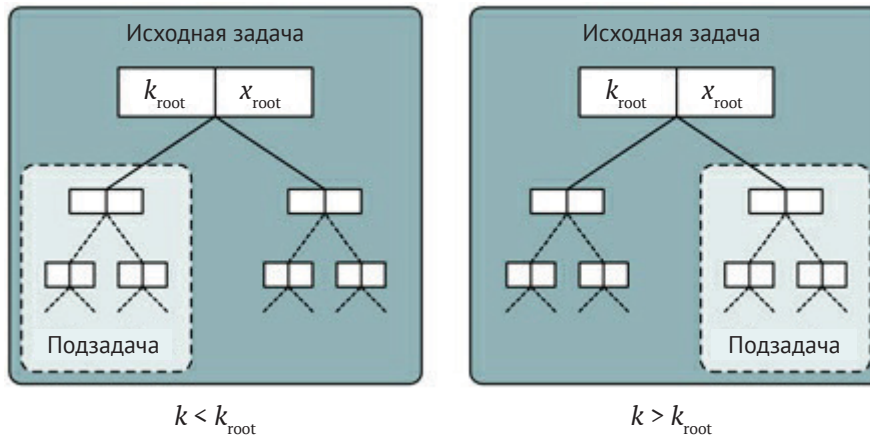


Рис. 5.4. Декомпозиция для некоторых алгоритмов поиска в двоичном дереве

Понятно, что для этой частной задачи существует два рекурсивных условия. Если $k < k_{root}$, метод должен продолжить поиск в левом поддереве посредством рекурсивного вызова, тогда как если $k > k_{root}$, он должен искать элемент в правом поддереве. В листинге 5.9 приведена реализация алгоритма поиска, где каждый узел двоичного дерева – это список из четырёх компонентов, как описано в разделе 5.3.

Листинг 5.9. Алгоритм поиска элемента с заданным ключом в двоичном дереве

```
def bst_search(T, key):
    if T == []:
        return None
    elif T[0] == key:
        return T[1]     # return the root item
    elif key < T[0]:
        return bst_search(T[2], key)     # search in left subtree
    else:
        return bst_search(T[3], key)     # search in right subtree
```

В итоге временна́я стоимость алгоритма определяется в самом худшем случае высотой двоичного дерева. Если дерево сбалансировано (то есть имеет примерно равное количество узлов на одном и том же уровне в левых и правых поддеревьях), то время его работы – ($\log n$), где n – число узлов в дереве, поскольку при каждом рекурсивном вызове он пропускает примерно половину узлов. Но если дерево имеет линейную структуру, то оценка времени работы алгоритма – $O(n)$.

5.3.2. Вставка элемента

Цель этой задачи – вставить элемент с заданным ключом k в двоичное дерево так, чтобы получающееся дерево тоже удовлетворяло основному свойству двоичного дерева. Метод представляет собой процедуру, которая получает два параметра: (1) кортеж x из ключа и значения элемента и (2) двоичное дерево $\boldsymbol{T}$. Он должен, в частности, расширить список $\boldsymbol{T}$, добавив в него новый лист.

И здесь можно считать, что размер задачи – это высота дерева. Самое простое начальное условие выполняется для пустого дерева. В этом случае алгоритм должен расширить дерево, включив в него один узел, у которого, очевидно, нет дочерних записей. При декомпозиции задачи мы будем придерживаться схемы на рис. 5.4. Очевидно, если дерево не пусто и k меньше ключа корневого узла, то процедура вставит узел в левое поддерево. Иначе она вставит его в правое поддерево. Сразу предположим, что $k < k_{\text{root}}$.

Есть два возможных сценария, приводящих к начальному и рекурсивному условиям. Если у корневого узла нет левого поддерева, то процедура может просто заменить соответствующий пустой список узлом, содержащим ключ и элемент (и два пустых поддерева). Но если у корневого узла есть непустое левое поддерево, то метод должен вставить в него новый узел, что, естественно, приводит к рекурсивному вызову. То же справедливо и для правого поддерева. В листинге 5.10 приведена возможная реализация процедуры.

5.4. Схемы разбиения

Известная задача в информатике – разбиение списка следующим способом.

1. В списке выбирается «опорный» элемент – обычно первый, средний или выбранный наугад.
2. Элементы входного списка меняются местами внутри него таким образом, чтобы начальные элементы были не больше опорного, а все остальные больше него.

Листинг 5.10. Процедура вставки элемента с заданным ключом в двоичное дерево

```
def insert_binary_tree(x, T):
    if T == []:
        T.extend([x[0], x[1], [], []])
    else:
       if x[0] < T[0]:
          if T[2] == []:
              T[2] = [x[0], x[1], [], []]
          else:
             insert_binary_tree(x, T[2])
       else:
          if T[3] == []:
              T[3] = [x[0], x[1], [], []]
          else:
             insert_binary_tree(x, T[3])
```

Рисунок 5.5 иллюстрирует идею на конкретном примере. Задача относится к числу важнейших, поскольку метод разбиения лежит в основе таких выдающихся алгоритмов, как quickselect (см. раздел 5.5) или quicksort (см. раздел 6.2.2). Есть несколько известных и эффективных алгоритмов, также использующих «схему разбиения» и решающих ту же задачу. Самый популярный из них – метод разбиения, разработанный Чарльзом Хоаром, который мы вкратце рассмотрим в дальнейшем. Но сначала мы разберём более простые и интуитивно понятные рекурсивные алгоритмы.

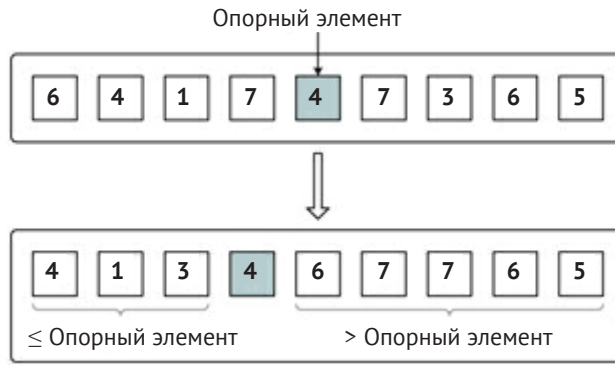
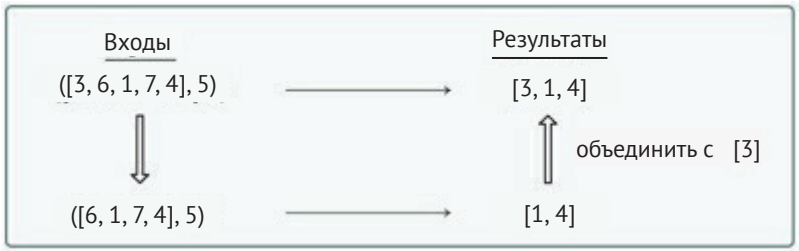


Рис. 5.5. Разделение списка, используемого в алгоритмах quicksort и quickselect

5.4.1. Основная схема разбиения

Простая идея заключается в просмотре всего входного списка с целью построить: (1) новый список из элементов, не бóльших опорного, и (2) другой список из элементов, бóльших опорного. После чего их можно объединить, вставив между ними опорный элемент.

Так как обе задачи очень похожи, мы разберём только первую. В частности, вход задачи – список длины *n* и некоторое значение *x*, исполняющее роль опорного элемента. Её размер – очевидно, длина списка. Начальное условие возникает при пустом входном списке, когда алгоритм просто возвращает пустой список. Для вывода рекурсивного условия применим декомпозицию с уменьшением размера задачи на 1 путём отбрасывания первого элемента списка. В этом случае можно сформировать следующие рекурсивные схемы в зависимости от соотношения между 1-м и опорным элементами списка. Если 1-й элемент не больше опорного, то схема для конкретного списка может быть такой:



Ясно, что алгоритм должен объединить 1-й элемент списка с результатом подзадачи. Напротив, если 1-й элемент больше опорного, то схема будет такой:

В этом случае алгоритм просто должен вернуть решение (список) подзадачи. В листинге 5.11 приведён линейно-рекурсивный код, решающий обе задачи (обратите внимание, что в рекурсивных условиях, выполняющих объединение списков, вызов функции не является последним действием метода).

Листинг 5.11. Вспомогательные методы для разбиения списка

```
1  def get_smaller_than_or_equal_to(a, x):
2      if a == []:
3          return []
4      elif a[0] <= x:
5          return [a[0]] + get_smaller_than_or_equal_to(a[1:], x)
6      else:
7          return get_smaller_than_or_equal_to(a[1:], x)
8
9
10 def get_greater_than(a, x):
11     if a == []:
12         return []
13     elif a[0] > x:
14         return [a[0]] + get_greater_than(a[1:], x)
15     else:
16         return get_greater_than(a[1:], x)
```

Время выполнения методов можно оценить как:

$$T(n) = \begin{cases} 1, & \text{если } n = 0, \\ T(n-1)+1, & \text{если } n > 0. \end{cases}$$

Это значит, что время выполнения методов линейно относительно n. Наконец, функции можно заменить выражениями языка Python, реализующими «фильтры». В частности, функции `get_smaller_than_or_equal_to(a, x)` и `get_greater_than(a, x)` возвращают те же списки, что и выражения [y f o r y i n i f y <= x] и [y f o r y i n i f y > x] соответственно.

5.4.2. Метод разбиения Хоара

В листинге 5.12 приведена итерационная версия алгоритма разбиения Хоара. В частности, он выполняет внутреннее разбиение списка **a** на подсписки, которые определяются своими нижним и верхним индексами.

Сначала метод выбирает опорный элемент (в данном случае – в середине подсписка) и меняет его с первым элементом подсписка (строки 3–6). Затем он устанавливает левый индекс на второй позиции подсписка, а правый – на последней (строки 8 и 9). После чего метод запускает основной цикл алгоритма. В нём он последовательно увеличивает левый индекс, пока тот не укажет на элемент, больший, чем опорный (строки 13 и 14). Таким же образом он уменьшает правый индекс,

пока тот не укажет на элемент, не больший опорного (строки 16 и 17). После этого (строки 19–22) алгоритм меняет элементы, на которые ссылаются левый и правый индексы (если левый индекс меньше правого). Этот процесс повторяется до тех пор, пока индексы не пересекутся (то есть пока левый индекс не станет больше правого, что проверяется в строке 24). В конце цикла элементы до правого индекса будут не больше опорного, а элементы после левого индекса (и до конца подсписка) будут больше опорного. В заключение процедура разбиения меняет местами элемент с правым индексом и опорный (строки 26 и 27) и возвращает его окончательную позицию. На рис. 5.6 приведён конкретный пример метода разбиения.

Листинг 5.12. Итерационный алгоритм разбиения Хоара

```
1  def partition_Hoare(a, lower, upper):
2      if upper >= 0:
3          middle = (lower + upper) // 2
4          pivot = a[middle]
5          a[middle] = a[lower]
6          a[lower] = pivot
7
8          left = lower + 1
9          right = upper
10
11         finished = False
12         while not finished:
13             while left <= right and a[left] <= pivot:
14                 left = left + 1
15
16             while a[right] > pivot:
17                 right = right - 1
18
19             if left < right:
20                  aux = a[left]
21                  a[left] = a[right]
22                  a[right] = aux
23
24             finished = left > right
25
26         a[lower] = a[right]
27         a[right] = pivot
28
29         return right
```

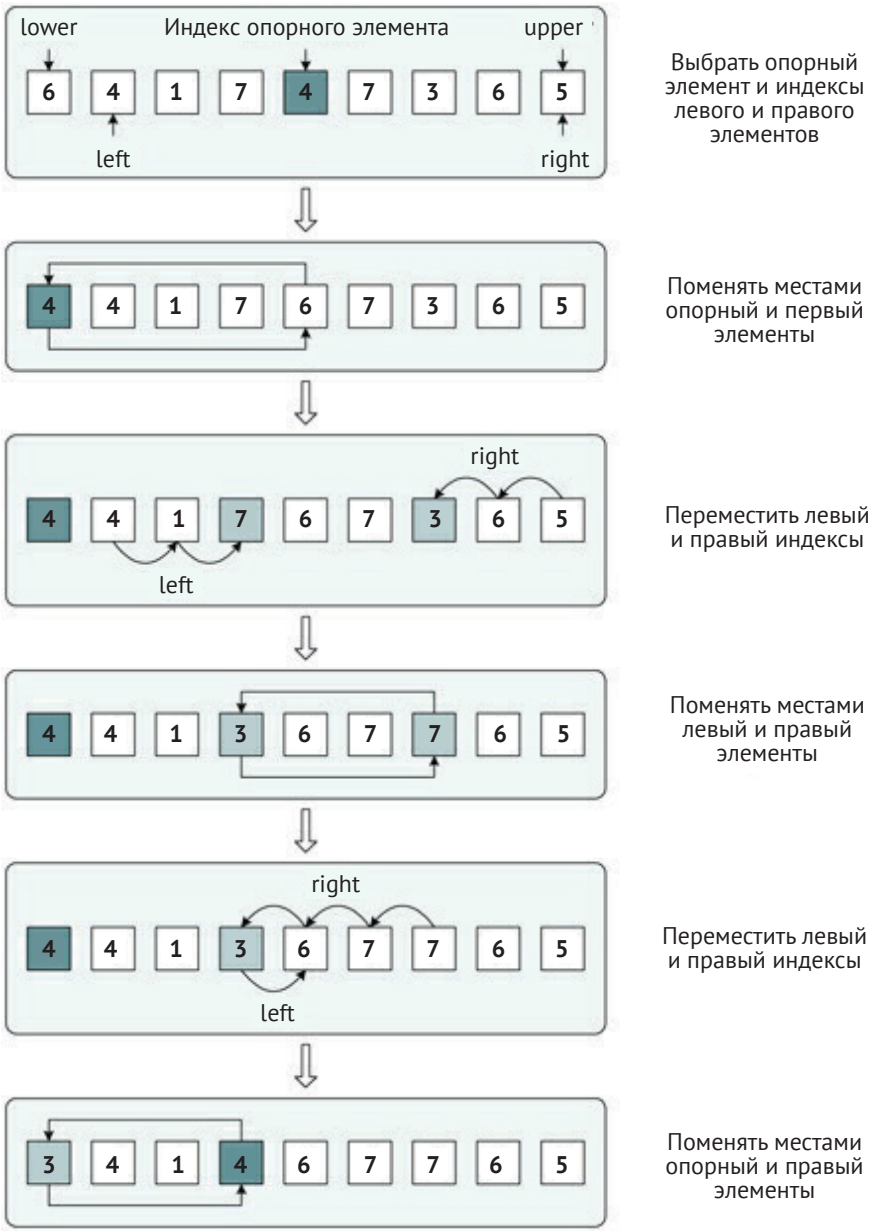


Рис. 5.6. Пример метода разбиения Хоара

Рассмотрим теперь алгоритм хвостовой рекурсии, который должен заменить основной цикл в схеме разбиения Хоара. Для заданных входного списка **a**, начальных значений левого и правого индексов, а так-

же опорного элемента метод должен, подобно циклу в методе Хоара, разбить список и вернуть заключительную позицию правого индекса. Размер задачи – это разность между левым и правым индексами, поскольку именно она определяет количество операций увеличения и уменьшения индексов до их пересечения. Именно в тот момент, когда левый индекс становится больше правого, выполняется начальное условие, а метод просто возвращает правый индекс.

В рекурсивном условии мы будем сокращать размер задачи путём увеличения левого индекса и/или уменьшения правого. Возможны два разных сценария, приведённых на рис. 5.7, где p – опорный элемент, left и right – соответственно левый и правый индексы.

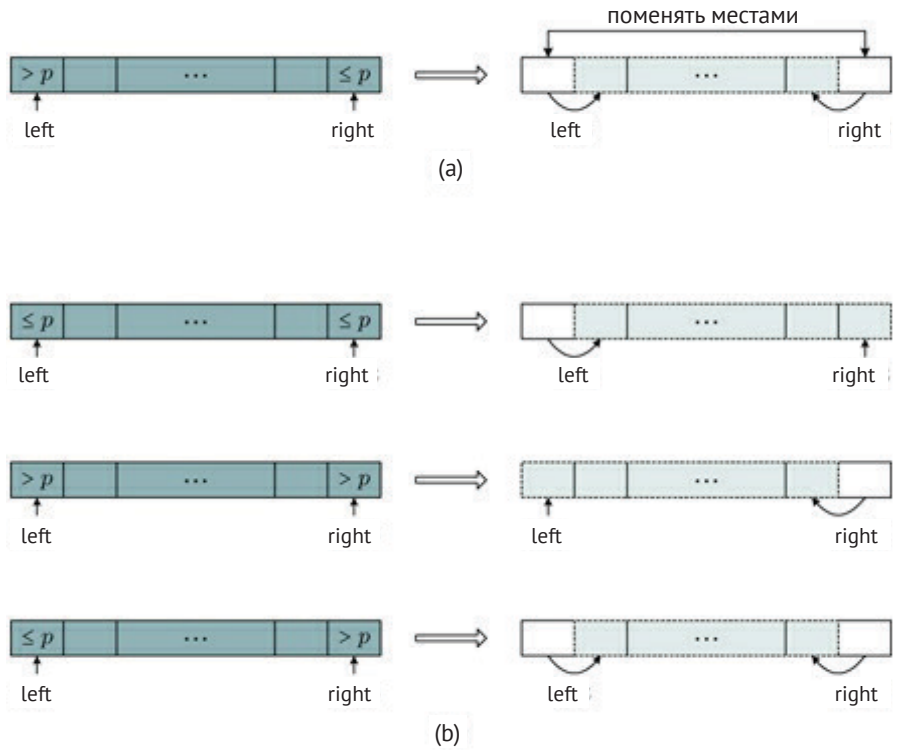


Рис. 5.7. Декомпозиция задачи о разбиении Хоара

Если a[left] > p и a[right] ⩽ p, то сначала метод должен поменять местами элементы с этими индексами и только после этого выполнить вызов функции с продвижением обоих индексов, как показано на рис. 5.7(a). Это приводит к первому рекурсивному условию, уменьшающему размер задачи на две единицы. Если предыдущее условие не выполняет-

ся, метод не поменяет местами элементы, а изменит, по крайней мере, один из индексов (уменьшив тем самым размер задачи), как показано на рис. 5.7(b). Если a[left] ⩽ p, он увеличит левый индекс, а если a[right] > p – уменьшит правый. После этого метод вызывает сам себя с новыми значениями индексов. В листинге 5.13 приводится соответствующий код вместе с методом-оболочкой, завершающим алгоритм разбиения. Отметим, что он идентичен листингу 5.12, цикл которого заменён вызовом рекурсивной функции. Наконец, метод работает за время $O(n)$, так как время выполнения рекурсивного метода может быть в самом худшем случае оценено рекуррентным соотношением $T(n) = T(n - 1) + 1$.

Листинг 5.13. Альтернативная рекурсивная версия схемы разбиения Хоара

```
def partition_Hoare_rec(a, left, right, pivot):
    if left > right:
        return right
    else:
        if a[left] > pivot and a[right] <= pivot:
            aux = a[left]
            a[left] = a[right]
            a[right] = aux
            return partition_Hoare_rec(a, left + 1,
                                       right - 1, pivot)
        else:
            if a[left] <= pivot:
                left = left + 1
            if a[right] > pivot:
                right = right - 1
            return partition_Hoare_rec(a, left, right, pivot)


def partition_Hoare_rec(a, lower, upper):
    if upper >= 0:
       middle = (lower + upper) // 2
       pivot = a[middle]
       a[middle] = a[lower]
       a[lower] = pivot

       right = partition_Hoare_rec(a, lower + 1, upper, pivot)

        a[lower] = a[right]
        a[right] = pivot

        return right
```

5.5. Алгоритм quickselect

Алгоритм quickselect, также разработанный Чарльзом Хоаром, – это алгоритм поиска выбором, основанный на схеме разбиения Хоара (см. раздел 5.4.2). В частности, он находит в несортированном списке k-е наименьшее число (называемое также статистикой k-го порядка). Пусть экземпляр задачи определяется входными параметрами – числовым списком **a**, нижним и верхним индексами внутри списка и положительным целочисленным значением k. Тогда алгоритм должен найти k-е наименьшее число в заданном подсписке. Размер задачи – длина подсписка. Наименьшие экземпляры задачи соответствуют спискам из одного элемента. Поэтому начальное условие выполняется, когда нижний и верхний индексы совпадают, а метод просто возвращает этот единственный элемент списка.

Для больших списков метод сначала применяет схему разбиения Хоара, который делит список на три подсписка и упорядочивает каждый из них. Первый подсписок содержит элементы, не большие выбранного опорного элемента списка. За ним следует подсписок, состоящий только из одного опорного элемента. И наконец, третий подсписок содержит элементы, бóльшие опорного.

Пусть i_p обозначает индекс опорного элемента. Поскольку функция, реализующая схему разбиения Хоара, возвращает i_p, можно проверить, равен ли он $k - 1$ (так как индексы начинаются с 0, а j-й элемент находится в позиции $j - 1$). Если это так, то опорный элемент – k-й наименьший элемент в **a**, и метод заканчивается на этом начальном условии. Этот сценарий приведён на рис. 5.8(a).

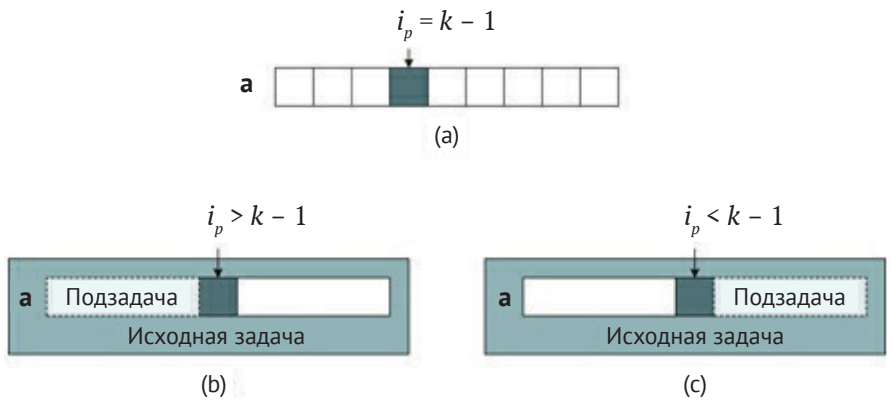


Рис. 5.8. Начальное условие и декомпозиция задачи, используемые алгоритмом quickselect

В рекурсивном условии алгоритм уменьшает размер задачи за счёт перехода либо к левому от опорного элемента подсписку, либо к правому. Если $i_p > k - 1$, то позиция искомого элемента будет меньше i_p, алгоритм может сосредоточиться на подсписке слева от опорного, как показано на рис. 5.8(b). Иначе, если $i_p < k - 1$, алгоритм продолжает решать подзадачу в правом от опорного подсписке, как показано на рис. 5.8(c). В заключение приведём листинг 5.14 с реализацией хвостовой рекурсивной функции.

Листинг 5.14. Хвостовой рекурсивный алгоритм quickselect

```
def quickselect(a, lower, upper, k):
    if lower == upper:
       return a[lower]
    else:
        pivot_index = partition_Hoare_wrapper(a, lower, upper)

        if pivot_index == k - 1:
           return a[pivot_index]
        elif pivot_index < k - 1:
           return quickselect(a, pivot_index + 1, upper, k)
        else:
           return quickselect(a, lower, pivot_index - 1, k)
```

Оценка времени выполнения алгоритма зависит от положения опорного элемента после выполнения декомпозиции. Если он всегда находится в середине списка, время выполнения характеризуется функцией

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ T(n/2) + c_n, & \text{если } n > 1, \end{cases}$$

поскольку алгоритму требуется решить подзадачу примерно в половину размера исходной задачи, так как основанные на разбиении методы работают за линейное время. Это лучший вариант развития событий, когда $T(n) \in \Theta(n)$. Однако если опорный элемент всегда оказывается в крайних позициях списка, время выполнения может быть оценено функцией

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ T(n-1) + c_n, & \text{если } n > 1, \end{cases}$$

которая является квадратичной, то есть $T(n) \in \Theta(n^2)$. Эта ситуация соответствует худшему варианту развития событий в алгоритме.

5.6. Двоичный поиск корня функции

Рассмотрим непрерывную действительную функцию $f(x)$, определенную на интервале $[a, b]$. Алгоритм двоичного поиска (также известный как метод деления интервала пополам, половинного деления, бисекции или дихотомии) – это способ поиска приближения z корня $f(x)$ на интервале (a, b). Напомним, что корень, или ноль $f(x)$, – это значение r, для которого $f(r) = 0$.

Входные параметры задачи – границы интервала a и b, где $a \leqslant b$, функция f и малое значение $\epsilon > 0$, задающее точность приближения, то есть выходной параметр функции z должен удовлетворять условию $|z - r| \leqslant \epsilon$. Таким образом, чем меньше ϵ, тем точнее приближение. Ещё одно обязательное условие – то, что $f(a) \cdot f(b) < 0$. Это гарантия того, что корень находится внутри интервала (a, b). Наконец, приближение z определяется серединой интервала (a, b), то есть $z = (a + b)/2$.

Метод последовательно делит пополам тот интервал, где должен быть корень. На рис. 5.9 приведено несколько шагов этого процесса.

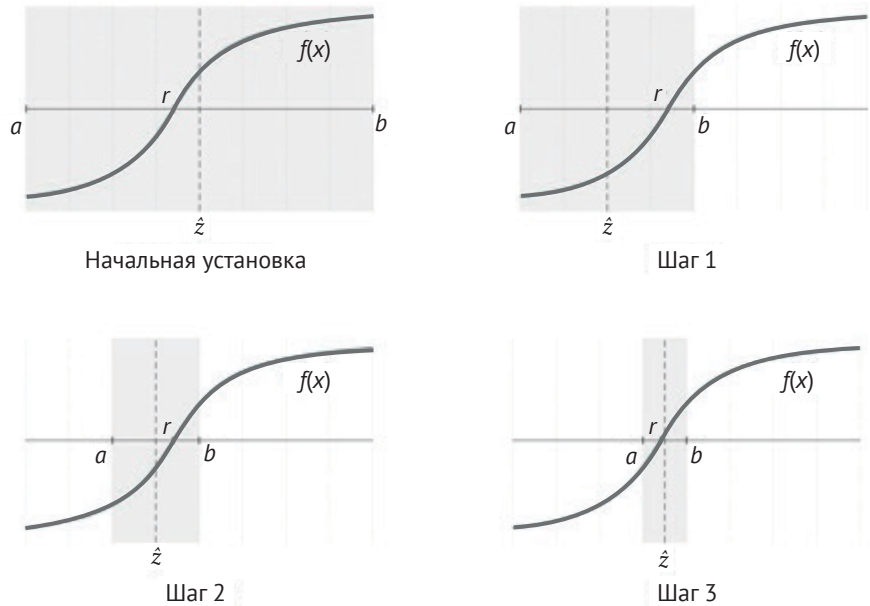


Рис. 5.9. Несколько шагов алгоритма двоичного поиска корня

В исходном положении корень расположен слева от первого приближения z или середины интервала ($r < z$), поэтому на следующем шаге

граница b смещается ближе к этой середине. Точно так же на шаге 1 $z < r$, поэтому на следующем шаге a получит значение z. Эта процедура применяется многократно до тех пор, пока интервал не станет достаточно маленьким, чтобы обеспечить для z необходимую точность.

Размер задачи зависит от длины отрезка $[a, b]$, а начальное условие выполняется при $b - a \leqslant 2\epsilon$, когда метод возвращает $z = (a + b)/2$. Заметим, что это условие обеспечивает выполнение условия $| z - r | \leqslant \epsilon$, как видно по рис. 5.10. В частности, если z – середина отрезка $[a, b]$, то расстояние между z и r не может быть больше ϵ. Кроме того, метод может просто вернуть z, если $f(z) = 0$ (или $f(z)$ достаточно мала).

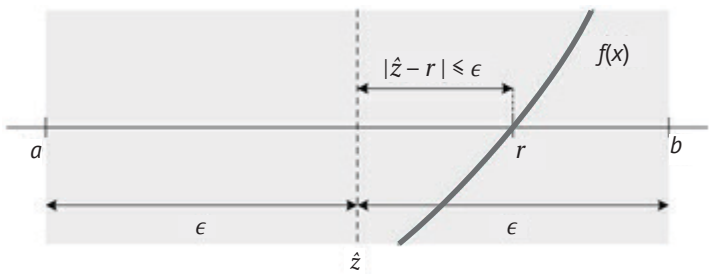


Рис. 5.10. Начальное условие алгоритма двоичного поиска ($b - a \leqslant 2\epsilon$)

Декомпозиция задачи заключается в делении её размера пополам. Это достигается заменой одного из граничных значений отрезка $[a, b]$ его серединой z. Поскольку знаки значений функции на концах отрезка должны быть противоположными, z должен заменить b, когда $f(a)$ и $f(z)$ имеют противоположные знаки. Иначе z заменяет a. Это приводит к хвостовому рекурсивному методу в листинге 5.15, содержащему два рекурсивных условия. В коде используется функция $f(x) = x^2 - 2$, которая позволяет найти приближение $\sqrt{2}$, поскольку это корень $f(x)$. Отметим, что начальный интервал $[0, 4]$ содержит $\sqrt{2}$. Наконец, погрешность приближения будет не более 10^{-10} (результат получится с точностью до девяти десятичных цифр после десятичной запятой).

Листинг 5.15. Алгоритм двоичного поиска корня функции

```
def f(x):
    return x * x - 2


def bisection(a, b, f, epsilon):
    z = (a + b) / 2

```

```
    if f(z) == 0 or b - a <= 2 * epsilon:
        return z
    elif (f(a) > 0 and f(z) < 0) or (f(a) < 0 and f(z) > 0):
        return bisection(a, z, f, epsilon)
    else:
        return bisection(z, b, f, epsilon)


# Print an approximation of the square root of 2
print (bisection(0, 4, f, 10**(-10)))
```

5.7. Задача лесоруба

В этой задаче лесорубу нужно заготовить *w* погонных метров древесины на лесном участке из *n* деревьев. У него есть специальная машина, пилу которой можно установить на любую высоту *h* так, чтобы срезать деревья выше этой высоты. Пусть *w* – целое число (не превосходящее суммарной высоты всех деревьев на участке), а высоты *n* деревьев (тоже целочисленные) заданы в списке ***t***. Цель задачи – найти наибольшее значение *h*, которое позволит лесорубу заготовить по крайней мере[1] *w* погонных метров древесины. На рис. 5.11 приведён пример задачи для $n = 20$ деревьев, где высота самого высокого дерева $H = 12$. Если нужно заготовить 10 погонных метров леса, то лесоруб должен установить высоту пилы $h = 8$, чтобы спилить в точности 10 погонных метров. И даже если нужно заготовить 7, 8 или 9 погонных метров леса, то оптимальная высота *h* всё равно была бы равна 8. Несмотря на то что лесоруб заготовил бы несколько больше погонных метров древесины, *h* не должна быть больше, поскольку срез на высоте 9 дал бы только 6 погонных метров древесины.

Задача представляет интерес с алгоритмической точки зрения, поскольку может быть решена несколькими способами. Например, деревья можно сначала отсортировать в порядке убывания их высот, а затем спиливать от самого высокого до самого низкого до получения оптимального результата. Этот подход работал бы за время $O(n \log n)$, если бы мы применяли такие общие алгоритмы сортировки, как сортировка слиянием или quicksort (см. главу 6). Так как высоты – это целые числа, можно применить линейные алгоритмы сортировки за линейное время, как в методе сортировки подсчётом (см. упражнение 5.3), который привел бы к решению, работающему за время $O(n + H)$. Таким образом, при малых значениях *H* этот подход мог бы быть наиболее эффективным.

[1] То есть с минимальным излишком. – *Прим. перев.*

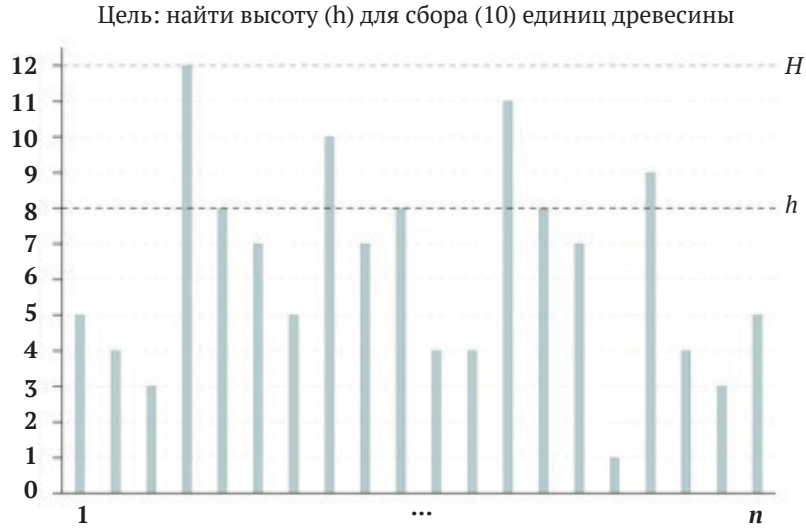


Рис. 5.11. Пример задачи лесоруба

Теперь рассмотрим алгоритм решения задачи методом «двоичного поиска», который работает за время $O(n \log H)$ и не требует сортировки деревьев по высоте. Сразу отметим, что алгоритм начинает работу с $h = H - 1$, постепенно уменьшая её на 1 до получения нужного количества леса.

Для простоты предположим, что метод вычисляет количество леса, полученное для каждой новой высоты h независимо от количества, полученного для других высот. Это значит, что вычисление количества леса, собранного с n деревьев, требует порядка n операций. Поэтому для решения всей задачи алгоритм потребовал бы $O(nH)$ операций в самом худшем случае (для каждой из возможных высот H алгоритм должен будет выполнить n вычислений). В частности, количество леса, собранного для некоторой высоты h, может быть получено линейно-рекурсивной функцией из листинга 5.16 (в упражнении 11.2 предлагается реализовать хвостовую рекурсивную версию этой функции). В начальном условии, когда список высот деревьев пуст, метод возвращает ноль. Рекурсивное условие обрабатывает высоту первого дерева из списка. Если она больше h, то дерево будет спилено, а функция должна вернуть (за счёт рекурсивного вызова) разницу между высотой дерева t_0 и h плюс длина уже заготовленной с остальной части деревьев древесины. Если высота дерева не больше h, то дерево не спиливается, и метод может просто вернуть величину уже заготовленного леса.

Листинг 5.16. Функция, вычисляющая количество древесины, если деревья срезаются на высоте *h*

```
def compute_wood(t, h):
    if t == []:
       return 0
    else:
        if t[0] > h:
           return t[0] - h + compute_wood(t[1:], h)
        else:
           return compute_wood(t[1:], h)
```

Вместо постепенного уменьшения *h* на 1 следующий алгоритм использует стратегию, подобную двоичному поиску элемента в списке или методу деления пополам. Идея состоит в том, чтобы начать поиск высоты *h* с середины интервала [0, *H*] и постепенно делить его пополам, пока не будет получено решение. Таким образом, метод получает в качестве параметров высóты деревьев, погонный метраж заготовляемой древесины, а также нижний и верхний пределы интервала поиска *h*.

Размер задачи – это разность между верхним и нижним пределами (сначала равна *H*). Первое начальное условие выполняется, если их значения равны. В этом случае высота *h* будет именно этим значением. В противном случае алгоритм вычислит «среднюю» высоту как (целочисленное) среднее от нижнего и верхнего пределов. Если суммарная длина спиленной на этой средней высоте древесины равна *w*, то метод возвращает эту среднюю высоту во втором начальном условии.

На рис. 5.12 показана используемая в алгоритме декомпозиция задачи. Исходная задача показана вверху с учётом нижнего и верхнего пределов. После вычисления общей длины древесины, которая могла быть собрана при средней высоте (h_m), возможны два сценария. Первый сценарий имеет место, когда h_m превышает *w*. В этом случае можно, очевидно, исключить все высоты ниже h_m и продолжить поиск оптимальной высоты (рекурсивным вызовом), изменив нижний предел на h_m. Важно отметить, что h_m все еще может быть решением, так как собранный для h_{m+1} лес может быть меньше необходимого количества *w*. Второй сценарий имеет место, когда собранный при h_m лес меньше *w*. В этом случае можно отказаться от всех высот выше средней, так как они не позволят лесорубу получить достаточного количества древесины. Поэтому соответствующее рекурсивное условие должно вызвать метод, заменив верхний предел h_{m-1}.

Рис. 5.12. Декомпозиция задачи лесоруба

Наконец, в предыдущей декомпозиции, когда верхний предел всего на единицу больше нижнего, средняя высота равна нижнему пределу. В этом случае первое рекурсивное условие не должно уменьшать размер задачи (пределы не должны меняться). Таким образом, чтобы работать должным образом, алгоритм нуждается в дополнительном начальном условии. Когда возникает такая ситуация, метод должен вернуть либо нижний, либо верхний предел. В частности, если количество леса, соответствующего верхнему пределу, не меньше w, то верхний предел – правильный результат. Иначе им будет нижний предел.

В листинге 5.17 приведён код, соответствующий упомянутым начальным и рекурсивным условиям.

Листинг 5.17. Алгоритм двоичного поиска для задачи лесоруба

```
def collect_wood(t, wood, lower, upper):
        middle_h = (lower + upper) // 2
        wood_at_middle = compute_wood(t, middle_h)

    if wood_at_middle == wood or lower == upper:
        return middle_h
    elif lower == upper - 1:
        if compute_wood(t, upper) >= wood:
            return upper
        else:
            return lower
    elif wood_at_middle > wood:
        return collect_wood(t, wood, middle_h, upper)
    else:
        return collect_wood(t, wood, lower, middle_h - 1)
```

Наконец, на рис. 5.13 приведена последовательность шагов алгоритма по решению задачи для конкретного примера на рис. 5.11.

Шаг 1 соответствует начальному условию, когда нижний предел равен нулю, а верхний – $H = \max\{t_i\}$ для $i = 1, \ldots, n$. Шаги 2 и 3 соответствуют применению первого и второго рекурсивных условий соответственно. Наконец, шаг 4 соответствует дополнительному начальному условию, когда верхний предел (8) на 1 больше нижнего (7). Решение – $h = 8$, поскольку при такой высоте лесоруб получает требуемое количество леса (10 единиц).

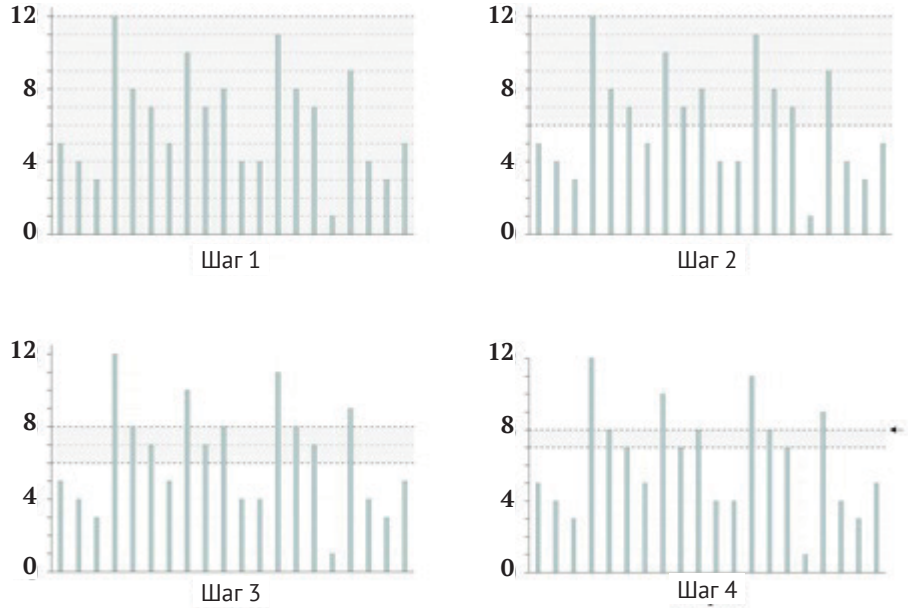


Рис. 5.13. Шаги алгоритма двоичного поиска на примере задачи лесоруба для $w = 10$

5.8. Алгоритм Евклида

Один из первых алгоритмов в истории известен как алгоритм Евклида, названный в честь древнегреческого математика Евклида, который описал его в своих «Началах» (прибл. 300 год до н. э.). Его цель – найти наибольший общий делитель (**g**reatest **c**ommon **d**ivisor, **gcd**), или сокращённо НОД, двух неотрицательных ненулевых целых чисел m и n, то есть наибольшее положительное целое число k, на которое делятся без остатка и m, и n (очевидно, m/k и n/k – целые числа). Например, НОД(20, 24) = 4. Его можно также считать произведением общих простых множителей m и n. Для $m = 20 = 2 \cdot 2 \cdot 5$ и $n = 24 = 2 \cdot 2 \cdot 2 \cdot 3$ произведение общих множителей $2 \cdot 2 = 4$.

Существует несколько вариантов метода. Первому варианту соответствует следующая функция:

$$\mathrm{gcd1}(m,n) = \begin{cases} n, & \text{если } m = 0, \\ \mathrm{gcd1}(n,m), & \text{если } m > n, \\ \mathrm{gcd1}(m,n-m) & \text{в противном случае,} \end{cases} \tag{5.2}$$

которую можно закодировать, как показано в листинге 5.18.

Листинг 5.18. Алгоритм Евклида для вычисления НОД двух неотрицательных целых чисел

```
1  def gcd1(m, n):
2      if n == 0:
3          return n
4      elif m > n:
5          return gcd1(n, m)
6      else:
7          return gcd1(m, n - m)
```

Во-первых, это – функция с хвостовой рекурсией, поскольку её значение определяется вызовом самой себя в рекурсивных условиях. Кроме того, заметьте, что в начальном условии метод возвращает значение аргумента функции, что является обычным для многих рекурсивных хвостовых вызовов. В частности, ясно, что НОД для $n > 0$ и для 0 – это n, так как и n, и 0 делятся на n без остатка. Первое рекурсивное условие просто меняет местами аргументы. Это гарантирует, что во втором рекурсивном условии первый из них будет не меньше второго. В этом последнем рекурсивном условии уменьшение размера задачи (который зависит от m и n) на 1 или деление его пополам не работает. Вместо этого рекурсивное условие уменьшает второй параметр путём вычитания из него m (отметим, что $n - m \geqslant 0$, так как $n \geqslant m$). Оно реализует математическое свойство

$$\text{НОД}(m, n) = \text{НОД}(m, n - m), \tag{5.3}$$

которое не вполне очевидно. Показать его истинность можно следующим образом. Пусть $n \geqslant m$, $m = a \cdot z$ и $n = b \cdot z$, где $a \leqslant b$ и $z = \text{НОД}(m, n)$ – произведение общих простых множителей n и m. Это означает, что a и b не содержат общих простых множителей. Кроме того, пусть $b = a + c$, тогда n и m можно выразить как:

$$n = b \cdot z = (a + c)\, z = (a_1 \cdot \ldots \cdot a_k + c_1 \cdot \ldots \cdot c_l)\, z, \tag{5.4}$$

$$m = (a_1 \cdot \ldots \cdot a_k)\, z, \tag{5.5}$$

где $a_1, \ldots, a_k$ и $c_1, \ldots, c_l$ – простые делители a и c соответственно. Ключ к доказательству в том, что a и c не имеют общих простых делителей (то есть $a_i \neq c_j$), потому что иначе в (5.4) их можно было бы вынести за скобки, предположив, что a и b имеют общие простые делители, а это неверно. Если же a и c не имеют общих простых делителей, то $z = \text{НОД}(a \cdot z, c \cdot z)$, и мы можем заключить, что:

$$\text{НОД}(a \cdot z, b \cdot z) = z = \text{НОД}(a \cdot z, c \cdot z) \cdot \text{НОД}(m, n) = \text{НОД}(m, n - m).$$

Кроме того, можно показать, что алгоритм гарантирует достижение начального условия, поскольку значения аргументов уменьшаются, пока один из них не станет нулевым. Для $m = 20$ и $n = 24$ метод выполняет следующие рекурсивные вызовы:

$$\begin{aligned}\text{gcd1}(20, 24) &= \text{gcd1}(20, 4) = \text{gcd1}(4, 20) = \\ &= \text{gcd1}(4, 16) = \text{gcd1}(4, 12) = \text{gcd1}(4, 8) = \\ &= \text{gcd1}(4, 4) = \text{gcd1}(4, 0) = \text{gcd1}(0, 4) = 4.\end{aligned} \tag{5.6}$$

Так как хвостовые рекурсивные функции не изменяют результаты рекурсивных вызовов, все они возвращают одно и то же значение, которое получается в начальном условии (gcd1(0, 4) = 4). Поэтому хвостовые рекурсивные функции, по сути, определяют отношения между наборами аргументов. Например, в (5.6) все пары (20, 24), (20, 4) и т. д. связаны со значением 4. В заключение приведём более эффективный вариант алгоритма, который используется в настоящее время:

$$\text{gcd2}(m, n) = \begin{cases} n, & \text{если } m = 0, \\ \text{gcd}\,2(n\%\, m, m), & \text{если } m \neq 0. \end{cases} \tag{5.7}$$

Доказательство свойства рекурсивного условия здесь подобно (5.3). Пусть $n \geqslant m$, $m = a \cdot z$ и $n = b \cdot z$, где z – произведение общих простых делителей n и m и $a \leqslant b$. Это значит, что a и b не имеют общих простых делителей. Кроме того, пусть $b = q + r$, где q и r – частное и остаток от деления b/a. Ключ этого доказательства в том, что a и r не имеют общих простых делителей, поскольку иначе они были бы простыми делителями b, что невозможно, так как a и b не имеют общих простых делителей. Отсюда $z = \text{НОД}(r \cdot z, a \cdot z)$. Таким образом, мы заключаем, что

$$\text{НОД}(a \cdot z, b \cdot z) = z = \text{НОД}(r \cdot z, a \cdot z) \cdot \text{НОД}(m, n) = \text{НОД}(n \,\%\, m, m).$$

5.9. Упражнения

Упражнение 5.1. Определите и реализуйте линейную и хвостовую рекурсивные логические функции, определяющие, содержит ли неотрицательное целое число n нечётную цифру.

Упражнение 5.2. Гораздо удобнее отображать полиномы в виде математической формулы, а не в виде списка его коэффициентов. Напишите метод, который получает список длины n с коэффициен-

тами полинома степени $n - 1$ и печатает его математическую запись. Например, для входного списка $\boldsymbol{p} = [3, -5, 0, 1]$, соответствующего полиному $x^3 - 5x^1 + 3$ (коэффициент p_i соответствует степени x^i), метод должен напечатать строку, подобную: $+1x^3 - 5x^1 + 3$. Будем считать, что $p_{n-1} \neq 0$, за исключением случая, когда полином представляет собой константу 0. В заключение определите его асимптотическую стоимость вычисления.

Упражнение 5.3. Алгоритм «сортировки подсчётом» – это метод сортировки списка из n целых чисел в диапазоне $[0, k]$, где k достаточно малое. Метод работает за время $O(n + k)$, откуда следует, что он работает за время $O(n)$ при $k \in O(n)$.

Для заданного списка **a** метод создаёт новый список **b**, который содержит количество вхождений каждого целого числа в **a**, как показано на рис. 5.14 (шаг 1).

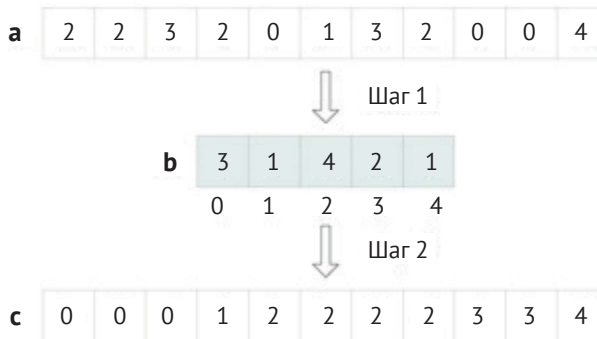


Рис. 5.14. Шаги алгоритма сортировки подсчётом

Например, $b_2 = 4$, так как целое число 2 появляется в **a** четыре раза. Реализуйте хвостовую рекурсивную процедуру, которая получает списки **a** и **b** (изначально нулевой) и заполняет список **b** количествами вхождений целых чисел в **a**.

Кроме того, реализуйте линейно-рекурсивную функцию, которая получает список **b** с количествами вхождений и возвращает новый список **c** – отсортированную версию **a**, как показано на рис. 5.14 (шаг 2).

Наконец, реализуйте функцию алгоритма сортировки подсчётом, которая вызывает предыдущие два метода, и определите её асимптотическую стоимость вычисления.

Упражнение 5.4. Реализуйте альтернативную листингу 5.8 версию алгоритма двоичного поиска, не использующего параметр `upper`. Определите затем его асимптотическую стоимость вычисления.

Упражнение 5.5. Реализуйте логическую функцию алгоритма «двоичного поиска», которая просто определяет, есть ли элемент x в списке **a**. Функции должны иметь всего два входных параметра x и a и опираться на декомпозицию задачи делением её размера пополам. В заключение определите её асимптотическую стоимость вычисления.

Упражнение 5.6. Напишите функцию, которая ищет элемент с наименьшим ключом в двоичном дереве поиска T, представленного списком, элементы которого состоят из четырех компонент, как в разделе 5.3.

Упражнение 5.7. Пусть **a** – сортированный (в порядке возрастания) список *различных* целых чисел. Цель этой задачи – *эффективный* поиск элемента, значение которого совпадает с его позицией (индексом) в списке. Иными словами, нужно найти элемент i, для которого $a_i = i$. Например, если **a** = [–3, –1, 2, 5, 6, 7, 9], то результатом будет 2, так как $a_2 = 2$. Отметим, что первый элемент списка расположен в позиции 0. Для простоты считайте, что в **a** не более одного элемента, удовлетворяющего условию $a_i = i$. Если в списке нет такого элемента, функция должна вернуть значение –1. Наконец, определите её асимптотическую стоимость вычисления.

Упражнение 5.8. Пусть **a** – список из n целых чисел, упорядоченных так, что чётные числа расположены перед нечётными (индекс любого чётного числа меньше индекса любого нечётного числа). Иными словами, для любого чётного a_i и любого нечётного a_j их индексы находятся в отношении $j > i$. Цель задачи – разработать эффективный рекурсивный алгоритм определения наибольшего индекса чётного числа. Например, если список **a** = [2, –4, 10, 8, 0, 12, 9, 3, –15, 3, 1], то метод возвращает $i = 5$, так как $a_5 = 12$ – чётное, а $a_6 = 9$ – нечётное (как a_7, a_8 и т. д.). Если в **a** нет чётных чисел, то результат должен быть –1. Если же **a** состоит только из чётных чисел, то результат должен быть n – 1. После этого определите асимптотическую стоимость вычисления функции.

Упражнение 5.9. Метод Ньютона (метод касательных) служит для поиска корня вещественной дифференцируемой функции $f(x)$ последовательным приближением к нему или уточнением его значения. Он опирается на следующее рекурсивное правило $x_n = x_{n-1} - f(x_{n-1}) / f'(x_{n-1})$, геометрическая интерпретация которого приведена на рис. 5.15.

Пусть у нас есть начальное приближение x_{n-1} корня r функции $f(x)$. Процедура проводит касательную (тангенсоиду) к $f(x)$ в точке x_{n-1} и вычисляет новое приближение корня (x_n) – точку пересечения этой касательной с осью X (заметьте, что x_n становится ближе к r). Таким образом,

начав с некоторого исходного значения x_0, $z = x_n$ будет приближением корня $f(x)$ после n-го применения рекурсивного правила.

Например, с помощью этой процедуры всего за несколько шагов можно получить достаточно точное приближение квадратного корня числа. Скажем, нам нужно вычислить $\sqrt{a}$, которое является некоторым неизвестным значением x. В этом случае у нас есть следующие тождества: $x = \sqrt{a}$, то есть $x^2 = a$ или $x^2 - a = 0$. Поэтому квадратным корнем из a будет корень функции $x^2 - a$. В этом случае согласно методу Ньютона рекурсивная формула будет такой:

$$x_n = x_{n-1} - x_{n-1}/2 + a/2x_{n-1}. \tag{5.8}$$

Реализуйте линейную и хвостовую рекурсивные функции, получающие в качестве аргументов значение a, его положительное начальное приближение x_0 и количество шагов n и возвращающие последнее приближение $z = x_n$ после n-кратного применения (5.8). После этого определите их асимптотическую стоимость вычисления.

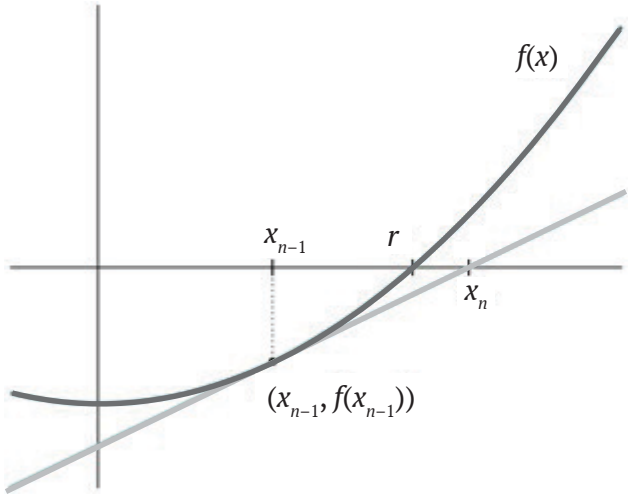


Рис. 5.15. Основная идея метода Ньютона

Глава 6

Множественная рекурсия I: «разделяй и властвуй»

ι α ι ́ ρ ε ι κ α ι β α σ ι ́ λ ε υ ε (Разделяй и властвуй).

– Филип II Македонский

Такие преимущества рекурсии над итерацией, как ясность кода или неявное управление стеком (см. раздел 10.3.5), проявляются главным образом при использовании множественной рекурсии. Основанные на этом типе рекурсии методы вызывают себя несколько раз, по крайней мере, в одном рекурсивном условии. Поэтому такие алгоритмы решают несколько простых подзадач, а затем должны объединить, расширить и/или изменить их результаты для получения решения исходной задачи.

Множественной рекурсии посвящены три главы книги, которые, как и вся книга, содержат дополнительные примеры. Эта глава охватывает важный класс алгоритмов с множественной рекурсией для задач, декомпозиция которых основана на делении их размера на некоторую константу. Говорят, что такие алгоритмы следуют принципу «разделяй и властвуй» – одной из самых важных парадигм разработки алгоритмов. Его можно рассматривать как общую стратегию решения задач, состоящую в разбиении задачи на несколько подобных себе подзадач, размер которых составляет некоторую часть размера исходной задачи. Поэтому такой подход имеет прямое отношение к рекурсии (итерационные алгоритмы тоже могут следовать этой стратегии), поскольку он опирается на рекурсивную декомпозицию задачи (см. рис. 1.4). Функция в листинге 1.2 и третий метод в листинге 1.5 – примеры такого подхода к разработке алгоритма.

В некоторых источниках термин «разделяй и властвуй» применяется и к тем алгоритмам, декомпозиция которых сводится к единственной подзадаче в половину размера исходной (с единственным вызовом метода). Однако многие авторы полагают, что этот термин должен применяться только тогда, когда решение предполагает разбиение задачи на две и более подзадач. В этой книге принимается именно это последнее соглашение. Таким образом, методы, делящие размер задачи пополам, но вызывающие себя лишь раз, уже включены в более ранние главы (в основном в главу 5). В следующих разделах описаны классические рекурсивные алгоритмы, следующие принципу «разделяй и властвуй».

6.1. Отсортирован ли список?

В этой задаче сортировки (скажем, по возрастанию) входом является список **a** из n элементов, которые можно отсортировать операцией $\leqslant$ (или некоторой функцией, позволяющей сравнивать элементы, реализуя полное бинарное отношение следования), а выходом – логическое значение, которое можно выразить как

$$f(a) = \bigwedge_{i=0}^{n-2} (a_i \leq a_{i+1}). \qquad (6.1)$$

Таким образом, результат равен True, если элементы списка располагаются (не строго) в возрастающем или неубывающем порядке.

Размер этой задачи – количество элементов списка. Если список содержит один элемент, то результат – очевидно, True. Кроме того, можно считать, что пустой список тоже упорядочен.

Задачу можно решить линейно-рекурсивным методом, который на этапе декомпозиции уменьшает размер задачи на 1. Однако мы представим решение, которое делит входной список задачи пополам и приводит её к следующей декомпозиции:

$$f(a) = (a_0 \leqslant a_1) \wedge \ldots \wedge (a_{\lfloor n/2 \rfloor - 2} \leqslant a_{\lfloor n/2 \rfloor - 1}) \qquad \{\text{подзадача 1}\}$$
$$\wedge$$
$$(a_{\lfloor n/2 \rfloor - 1} \leqslant a_{\lfloor n/2 \rfloor}) \wedge (a_{\lfloor n/2 \rfloor} \leqslant a_{\lfloor n/2 \rfloor + 1}) \wedge \ldots \wedge (a_{n-2} \leqslant a_{n-1}). \qquad \{\text{подзадача 2}\}$$

Очевидно, если весь список отсортирован по возрастанию, то обе его половины тоже должны быть отсортированы по возрастанию. Таким образом, метод может вызвать себя дважды с двумя соответствующими подсписками и выполнить логическую операцию **И** с их результатами. Наконец, объединение результатов подзадач требует дополнительного

шага. В частности, последний элемент первого подсписка $a_{\lfloor n/2 \rfloor - 1}$ должен быть не больше первого элемента второго подсписка $a_{\lfloor n/2 \rfloor}$. Это условие также требует ещё одной операции **И** в рекурсивном условии. В листинге 6.1 приведена возможная реализация метода, который работает за время $O(n)$, поскольку его стоимость вычисления:

$$T(n) = \begin{cases} 1, & \text{если } n \leq 1, \\ 2\,T(n/2) + 1, & \text{если } n > 1. \end{cases}$$

Листинг 6.1. Функция, определяющая, отсортирован ли список по возрастанию

```
def is_list_sorted(a):
    n = len(a)
    if n <= 1:
        return True
    else:
        return (is_list_sorted(a[0:n // 2])
                and a[n // 2 - 1] <= a[n // 2]
                and is_list_sorted(a[n // 2:n]))
```

6.2. Сортировка

Алгоритм сортировки произвольного списка (массива, последовательности и т. д.) – одна из наиболее изученных задач в информатике. Её можно решить многими способами, которые могут использоваться, чтобы ввести основные понятия, связанные с вычислительной сложностью и анализом времени выполнения, парадигмами разработки алгоритмов или структурами данных. Алгоритмы сортировки из этой главы предполагают, что вход задачи – это список **a** из n вещественных чисел. Выход – перегруппировка (или перестановка) элементов, которая даёт другой список **a**′, где $a'_i \leqslant a'_{i+1}$ для $i = 0, \ldots, n - 2$ ($\leqslant$ можно считать логической функцией, позволяющей определить, предшествует ли один элемент другому, и реализующей полное бинарное отношение следования). Выбор вещественного типа для элементов списка подразумевает, что алгоритмы должны выполнять сравнения операцией $\leqslant$. Можно показать, что любой алгоритм решения этой задачи требует $\Omega(n \log n)$ сравнений (то есть, по крайней мере, порядка $n \log n$ проверок $\leqslant$). Наконец, есть алгоритмы сортировки, не использующие сравнение элементов и требующие $\Omega(n)$ операций. Однако для их применения элементы списка должны удовлетворять определенным условиям. Например,

алгоритм «сортировки подсчётом» сортирует только целые числа, принадлежащие малому интервалу [0, k] (см. упражнение 9.2).

6.2.1. Алгоритм сортировки слиянием

Алгоритм сортировки слиянием – один из широко используемых примеров для демонстрации возможностей подхода «разделяй и властвуй». В то время как многие алгоритмы сортировки работают в худшем случае за время $O(n^2)$ (см. раздел 4.4.1 и упражнение 4.17), алгоритм сортировки слиянием работает за время $\Theta(n \log n)$. Иными словами, его вычислительная эффективность сулит лучшие перспективы.

Размер задачи – длина списка (n). Наименьшие экземпляры задачи возникают при $n \leqslant 1$, когда алгоритм просто должен вернуть входной список. В рекурсивном условии алгоритм разбивает задачу путём деления входного списка на две половины, порождая тем самым две разные подзадачи (примерно) в половину размера исходной. В следующей схеме приводится конкретный пример рекурсивного процесса осмысления будущего алгоритма:

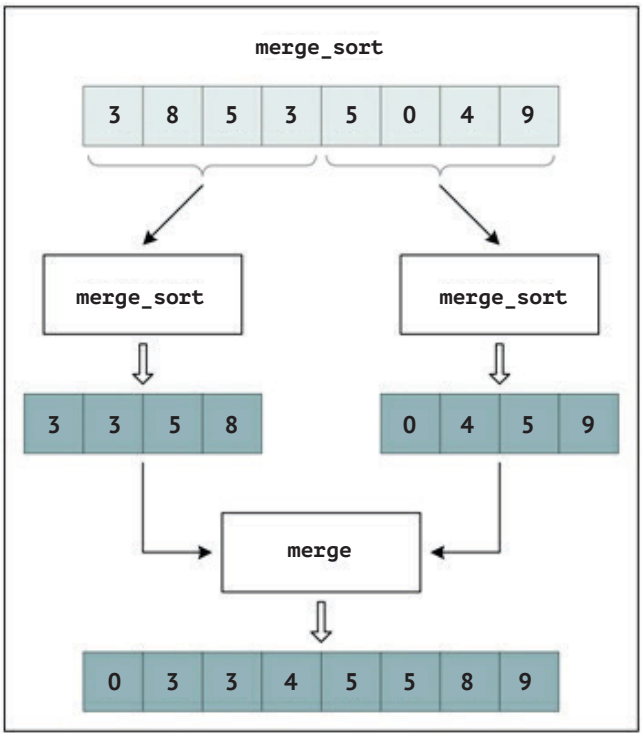
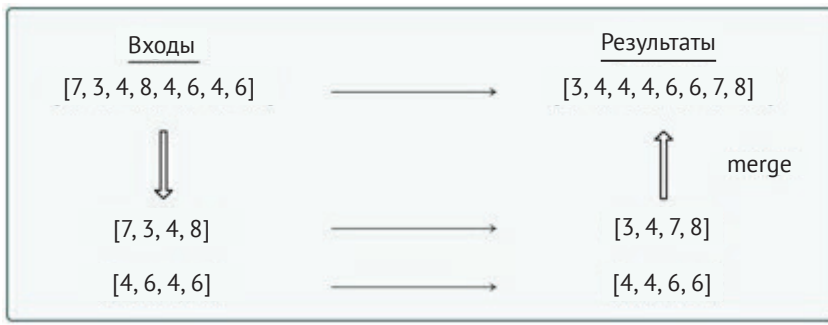


Рис. 6.1. Алгоритм сортировки слиянием

Рекурсивное условие здесь сложнее, чем в предыдущих примерах, так как окончательный сортированный список не может быть создан простой операцией (например, суммированием, вычислением максимума двух чисел, соединением двух списков и т. д.). В этом случае необходимо решить новую задачу слияния (объединения) решений подзадач. В частности, для получения итогового сортированного списка рекурсивное условие должно «слить» два сортированных списка (выходы двух подзадач). В листинге 6.2 приводится компактная реализация метода сортировки слиянием в предположении, что решающая эту задачу функция merge (см. ниже) уже реализована. Отметим лёгкость восприятия кода и простоту алгоритма, который в точности отвечает стратегии «разделяй и властвуй».

Листинг 6.2. Метод сортировки слиянием

```
def merge_sort(a):
    n = len(a)
    if n <= 1:
        return a
    else:
        a1 = merge_sort(a[0:n // 2])
        a2 = merge_sort(a[n // 2:n])
        return merge(a1, a2)
```



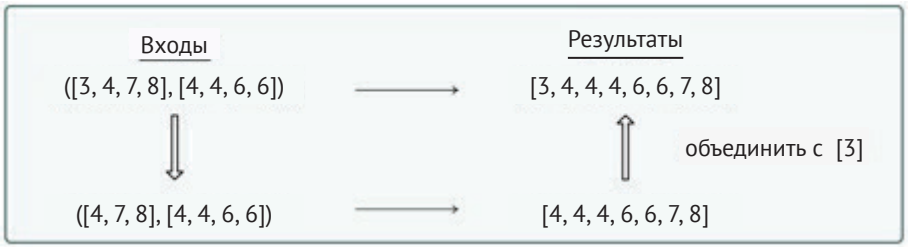
Оценка времени выполнения этого алгоритма:

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ 2\,T(n/2) + f(n), & \text{если } n > 1, \end{cases}$$

где мы также предположили, что разделение списка требует постоянного числа операций. Иными словами, мы решили, что $a[0{:}n//2]$ и $a[n//2{:}n]$

можно получить за время $\Theta(1)$. Кроме того, $f(n)$ оценивает количество операций, необходимых методу слияния для объединения двух сортированных списков (приблизительно) из $n/2$ элементов. В этом случае, согласно основной теореме (3.28), если бы $f(n)$ была линейной функцией, то алгоритм сортировки слиянием работал бы за (оптимальное) время порядка $\Theta(n \log n)$. Скоро мы увидим, что на самом деле задачу слияния можно решить за линейное время. Вообще, сталкиваясь с подобной задачей, разработчики алгоритма всегда стремятся создать наиболее эффективную комбинацию методов, снижающую издержки алгоритма «разделяй и властвуй».

Входы задачи слияния – два сортированных списка **a** и **b** с длинами n_a и n_b соответственно. Выход – другой сортированный список длины $n = n_a + n_b$. Можно считать, что размер задачи – это количество операций, необходимых рекурсивному алгоритму, чтобы «спуститься» до тривиального решения. При таком сценарии размер задачи будет $m = \min(n_a, n_b)$, поскольку если один из списков пуст, то решением задачи будет, очевидно, второй список. Это – начальные условия. Для рекурсивного условия можно использовать следующую схему с подсписками из предыдущего примера:



Декомпозиция уменьшает размер задачи на 1 за счёт отбрасывания наименьшего элемента в обоих списках. Естественно, он может оказаться в начальной позиции списков, поскольку они отсортированы по возрастанию. В примере наименьший элемент (3) – первый в первом входном списке. Очевидно, что рекурсивное условие должно просто присоединить этот элемент к результату подзадачи. В листинге 6.3 приведён соответствующий код. Оценка времени его выполнения характеризуется функцией

$$T(m) = \begin{cases} 1, & \text{если } m = 0, \\ T(m-1)+1, & \text{если } m > 0, \end{cases}$$

в предположении, что хвост списка (то есть $x[1:]$) можно получить за конечное время. Так как его нерекурсивное выражение – $T(m) = m + 1$, функция слияния выполняется за линейное время от m и от n тоже. А это значит, что алгоритм сортировки слиянием работает за время $\Theta(n \log n)$.

Листинг 6.3. Метод для слияния двух сортированных списков

```
# Lists a and b are sorted in ascending order
def merge(a, b):
    if a == []:
        return b
    elif b == []:
        return a
    else:
        if a[0] < b[0]:
           return [a[0]] + merge(a[1:], b)
        else:
           return [b[0]] + merge(a, b[1:])
```

6.2.2. Алгоритм быстрой сортировки

Алгоритм быстрой сортировки quicksort – ещё один метод, разработанный Чарльзом Хоаром. В нём применён подход «разделяй и властвуй», и он получил свое название из-за своей поразительной эффективности. В отличие от алгоритма сортировки слиянием время его выполнения в худшем случае – $O(n^2)$. Однако в лучших и средних случаях он выполняется за время $\Theta(n \log n)$ и может быть практически в несколько раз быстрее алгоритма сортировки слиянием.

Чтобы понять разницу между двумя методами, отметим, что декомпозиция в сортировке слиянием проста. В частности, входной список можно разделить пополам, используя просто соответствующие индексные диапазоны (0:n//2 и n//2:n). Однако объединение результатов подзадач требует решения еще одной задачи – слияния, которая не очень проста. Напротив, в алгоритме quicksort декомпозиция сложна, однако этап объединения не только тривиален, но в некоторых реализациях даже не всегда нужен.

А именно вместо простого деления списка пополам декомпозиция quicksort использует схему разделения, подобную схеме из раздела 5.4. Рисунок 6.2 иллюстрирует такой тип декомпозиции, когда подсписки внутри исходного списка разделены опорным элементом: элементы правого списка не больше опорного, а левого – больше опорного.

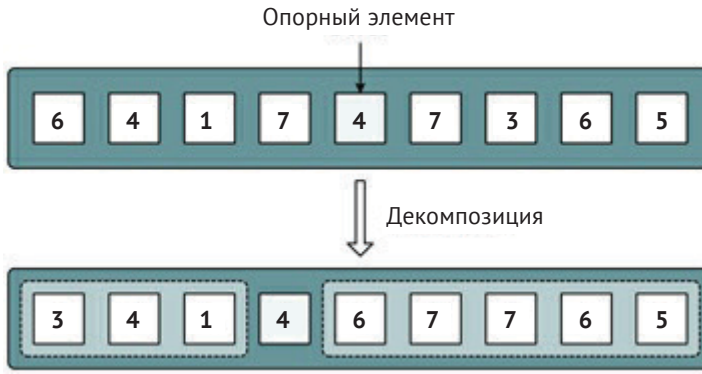
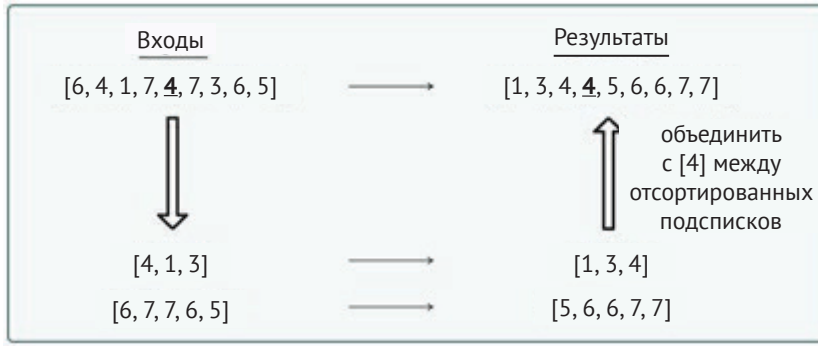


Рис. 6.2. Декомпозиция алгоритма quicksort

Примерная рекурсивная схема при такой декомпозиции могла бы быть такой:



Из схемы видно, что сортировка исходного списка требует сначала решения двух подзадач посредством рекурсивных вызовов и затем объединения отсортированных подсписков при обработке опорного элемента между ними. Таким образом, после рекурсивного решения подзадач объединение их результатов становится простым. Наконец, важная деталь этой декомпозиции заключается в том, что опорный элемент удаляется из списка не превосходящих его элементов. Это – гарантия того, что размер подзадачи действительно меньше размера исходной задачи.

В листинге 6.4 приводится более медленный вариант метода, основанный на простых схемах разделения из раздела 5.4.1.

Сначала он проверяет, соответствует ли вход начальному условию (которое является тем же, что в методе сортировки слиянием). В рекурсивном условии общая стратегия состоит в том, что в качестве опорно-

го элемента берётся первый элемент списка. При таком выборе возможен наихудший случай, когда входной список уже отсортирован. Кроме того, этот алгоритм может оказаться плохим ещё и тогда, когда вход почти отсортирован. Поскольку такие ситуации возникают на практике довольно часто, функция использует в качестве опорного средний элемент входного списка. После этого метод, согласно декомпозиции, получает два подсписка (удаляя опорный элемент из содержащего его подсписка) и наконец объединяет результаты подзадач, вставляя опорный элемент между ними.

Листинг 6.4. Вариант алгоритма quicksort

```
def quicksort_variant(a):
    n = len(a)
    if n <= 1:
        return a
    else:
        pivot = a[n // 2]
        v1 = get_smaller_than_or_equal_to(a, pivot)
        v1.remove(pivot)
        v2 = get_greater_than(a, pivot)
        return (quicksort_variant(v1) + [pivot]
                + quicksort_variant(v2))
```

Метод сортировки слиянием в листинге 6.2 и функция quicksort в листинге 6.4 не меняют входной список и возвращают свои результаты в новом списке, что требует вдвое больше памяти. Такие типы алгоритмов называются алгоритмами «внешней» сортировки. Вместо этого можно реализовать варианты, которые изменяют входной список и не требуют дополнительной памяти для всего списка длины *n*. Такие алгоритмы называются алгоритмами «внутренней» сортировки. В листинге 6.5 приводится алгоритм quicksort внутренней сортировки, использующий нижний и верхний пределы для указания границ подсписка внутри исходного списка. Код включает рекурсивную версию метода разбиения Хоара из листинга 5.13 (конечно, более эффективная итерационная версия тоже допустима), выполняющего декомпозицию задачи.

Оценка времени выполнения алгоритма подобна таковой для алгоритма quickselect. Основное различие – в том, что вместо решения одной подзадачи quicksort решает две, вызывая себя дважды. Наилучший случай – когда опорный элемент всегда находится в середине списка. В этом случае время выполнения характеризуется функцией

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ T(n/2) + c^n, & \text{если } n > 1, \end{cases}$$

где $T(n) \in \Theta(n \log n)$. Наоборот, худший случай – когда опорный элемент всегда располагается в крайнем положении списка. В этом случае время выполнения определяется функцией

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ T(n-1) + c^n, & \text{если } n > 1, \end{cases}$$

которая имеет квадратичный порядок роста ($T(n) \in \Theta(n^2)$).

Листинг 6.5. Алгоритм внутренней сортировки quicksort

```
def quick_sort_inplace(a, lower, upper):
    if lower < upper:
        pivot_index = partition_Hoare_wrapper (a, lower, upper)

        quick_sort_inplace(a, lower, pivot_index - 1)
        quick_sort_inplace(a, pivot_index + 1, upper)
```

6.3. Мажоритарный элемент списка

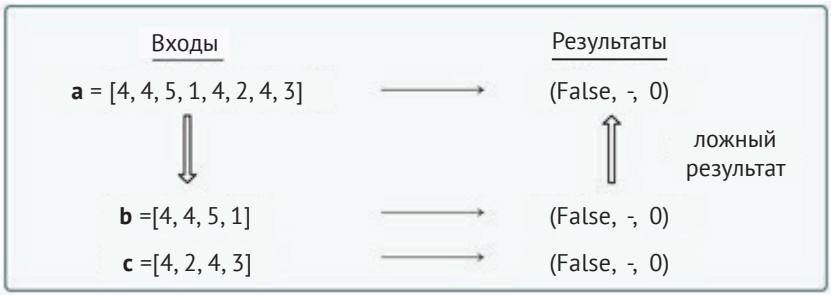
Говорят, что в списке есть «мажоритарный элемент» (или доминирующий, преобладающий), если *более* половины его элементов одинаковы. В этой классической задаче цель состоит в том, чтобы выяснить, есть ли в списке длины n мажоритарный элемент, и, если есть, найти его. Кроме того, задача предполагает, что элементы сравниваются за (постоянное) время $\Theta(1)$ при использовании операции сравнения (==). Иными словами, запрещаются сравнения <, >, ⩽ или ⩾ (в примерах будут использоваться только целочисленные операции).

Для решения задачи существует несколько вариантов применения принципа «разделяй и властвуй». Мы рассмотрим метод, возвращающий кортеж (или список) из трёх значений:

1) признак «содержит ли список мажоритарный элемент» (логическая переменная);
2) мажоритарный элемент, если он есть. Иначе это значение не определено (функция может просто вернуть None);
3) количество вхождений мажоритарного элемента (0, если его нет).

Размер задачи – n. Первое начальное условие может соответствовать пустому списку, результатом которого был бы кортеж (False, x, 0), где значение x не определено. Кроме того, если список содержит один элемент, функция должна вернуть (True, a_0, 1).

Рекурсивное условие выводится путём деления списка пополам и решения соответствующих подзадач. Чтобы понять, как объединить решения подзадач, построим на конкретном примере начальные рекурсивные схемы. Например:

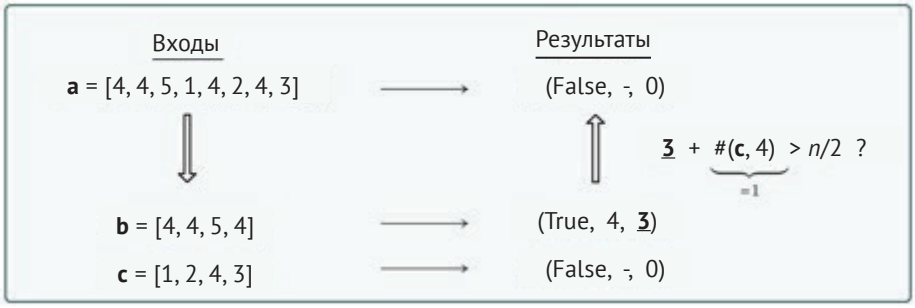


В этом конкретном случае результат обеих подзадач – False. Это значит, что входной список не содержит мажоритарного элемента (хотя список содержит ровно $n/2$ вхождений элемента 4, этого недостаточно для истинного результата). Вообще, это можно доказать следующим образом. Сначала исходный список делится пополам (независимо от чётности n, так как $n = \lfloor n/2 \rfloor + \lfloor n/2 \rfloor$) на два подсписка – **b** длины $\lfloor n/2 \rfloor$ и **c** длины $\lfloor n/2 \rfloor$. Если результат обеих подзадач – False, то элемент может появиться не более $\lfloor \lfloor n/2 \rfloor /2 \rfloor$ раз в **b** и $\lfloor \lfloor n/2 \rfloor /2 \rfloor$ раз в **c**. Сложение этих величин даёт:

$$\lfloor \lfloor n/2 \rfloor /2 \rfloor + \lfloor \lfloor n/2 \rfloor /2 \rfloor \leqslant (\lfloor n/2 \rfloor + \lfloor n/2 \rfloor)/2 = n/2 \leqslant n/2.$$

Отсюда можно заключить, что элемент не может появиться в исходном списке более $n/2$ раз.

Схема для другого конкретного примера могла бы быть такой:



В этом случае элемент 4 трижды появляется в первом подсписке и потому является мажоритарным, так как $3 > n/2 = 2$. Поэтому алгоритм должен посчитать число вхождений 4 во второй подсписок (обозначенный $\#(c, 4)$), чтобы определить, является ли он мажоритарным для исходного списка **a**. Это можно сделать с помощью простой линейно-рекурсивной функции (см. листинг 6.6), получающей на входе список **a** и элемент x. Если **a** пуст, результат, очевидно, 0. В противном случае результатом будет вызов метода, который применяется к хвосту списка **a** ($a_{1\ldots n-1}$) и x (плюс единица, если $a_0 = x$).

Листинг 6.6. Подсчёт количества элементов в списке

```
1  def occurrences_in_list(a, x):
2      if a == []:
3          return 0
4      else:
5          return int (a[0] ==x) + occurrences_in_list(a[1:], x)
```

В листинге 6.7 приводится возможная реализация функции, решающей задачу поиска мажоритарного элемента. Начальные условия – в строках 3–6. Строки 8 и 9 делят входной список пополам. Строка 11 вызывает метод для первого подсписка, и если в нём есть мажоритарный элемент (строка 12), то вычисляется количество вхождений этого элемента во второй подсписок (строка 13). Если общее количество вхождений элемента (в двух подсписках) больше $n/2$ (строка 14), метод возвращает кортеж (строка 15) со значениями (True, мажоритарный элемент, количество появлений во входном списке). Строки 17–21 аналогичны, но меняют списки ролями. Наконец, если функция не вернула результат, то список вообще не содержит мажоритарного элемента (строка 23).

В рекурсивных условиях метод должен вызвать себя дважды для каждой половины входного списка, а также вычислить количество вхождений элемента в оба подсписка длиной (примерно) $n/2$. Поскольку эта последняя вспомогательная функция выполняется за линейное время, временна́я сложность метода может быть оценена функцией

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ T(n/2) + c^n, & \text{если } n > 1. \end{cases}$$

Таким образом, временна*я* сложность алгоритма – порядка $\Theta(n \log n)$ (см. (3.28)). В заключение добавим, что эта задача может быть решена

за линейное время с помощью алгоритма «большинства голосов» Бойера–Мура (Boyer–Moore majority vote algorithm).

Листинг 6.7. Решение задачи о мажоритарном элементе

```
def majority_element_in_list(a):
    n = len(a)
    if n == 0:
        return (False, None, 0)
    elif n == 0:
        return (True, a[0], 1)
    else:
        b = a[0:n // 2]
        c = a[n // 2:n]

        t = majority_element_in_list(b):
        if t[0]:
            occurrences = occurrences_in_list(c, t[1])
            if t[2] + occurrences > n / 2
                return (True, t[1], t[2] + occurrences)

        t = majority_element_in_list(c):
        if t[0]:
            occurrences = occurrences_in_list(b, t[1])
            if t[2] + occurrences > n / 2
                return (True, t[1], t[2] + occurrences)

        return (False, None, 0)
```

6.4. Быстрое целочисленное умножение

Известный ещё со школы классический алгоритм поразрядного умножения двух неотрицательных N-разрядных целых чисел требует времени порядка n^2. В этом разделе мы разберём более быстрый алгоритм Карацубы. Метод можно применять к числам в любой системе счисления, но мы сосредоточимся на умножении двоичных чисел. В частности, пусть x и y – два неотрицательных целых числа, состоящих из b_x и b_y двоичных разрядов соответственно. Применив подход «разделяй и властвуй», каждое двоичное число можно разделить на два следующим образом:

$$x = a \cdot 2^m + b, y = c \cdot 2^m + d, \tag{6.2}$$

где $m = \min(\lfloor b_x/2 \rfloor, \lfloor b_y/2 \rfloor)$.

Например, для $x = 594$ и $y = 69$ декомпозиция будет такой:

$$x = 1001010010_2 = 1001010\{a = 74\}010\{b = 2\} = 74 \cdot 2^3 + 2,$$
$$y = 1000101_2 = 1000\{c = 8\}101\{d = 5\} = 8 \cdot 2^3 + 5,$$

где $b_x = 10$, $b_y = 7$, $m = 3$, $a = 74$, $b = 2$, $c = 8$ и $d = 5$. Таким образом, меньшее число (в данном случае – y) разделено на две части (примерно) с одинаковым количеством двоичных разрядов, а младшие части разбиения (b и d) имеют одинаковое количество двоичных разрядов.

Во многих языках программирования вычислить значения a, b, c и d можно с помощью операций поразрядного сдвига (в языке Python такими операциями являются << и >>). Сдвиг x влево на m разрядов ($x << m$) с добавлением m младших нулей равнозначен $\lfloor x \cdot 2^m \rfloor$. Сдвиг x вправо на m разрядов ($x >> m$) с отбрасыванием m младших разрядов равнозначен $\lfloor x/2^m \rfloor$. Такие поразрядные операции полезны при целочисленном умножении (или делении) на степень двух и могут использоваться для разбиения x и y согласно (6.2) следующим образом:

$$a = x >> m,\ b = x - (a << m),\ c = y >> m,\ d = y - (c << m),$$

где круглые скобки необходимы в силу правил старшинства операций.

Согласно декомпозиции, произведение x и y можно записать как

$$x \cdot y = (a \cdot 2^m + b)(c \cdot 2^m + d) = a \cdot c2^{2m} + (a \cdot d + b \cdot c)2^m + b \cdot d. \qquad (6.3)$$

Этот первоначальный (примитивный) подход разбивает исходную задачу (умножения) на четыре меньшие подзадачи умножения ac, ad, bc и bd, решение которых можно получить четырьмя рекурсивными вызовами (затратами времени на умножение на степень двух можно пренебречь, поскольку его очень эффективно заменяют операции поразрядного сдвига). Однако этот подход не намного эффективнее «школьного», так как он тоже требует n^2 поразрядных умножений для двух n-разрядных чисел. Предложенный Карацубой алгоритм может снизить затраты времени примерно до $1{,}585 \cdot n$ за счёт перегруппировки слагаемых и увеличения числа операций сложения и вычитания, которые оказывают незначительное влияние на асимптотическую сложность вычислений. В частности, произведение xy можно заново переписать как

$$x \cdot y = a \cdot c2^{2m} + ((a + b)(c + d) - a \cdot c - b \cdot d)2^m + b \cdot d, \qquad (6.4)$$

где требуется всего три более простых произведения $a \cdot c$, $b \cdot d$ и $(a + b)(c + d)$, что приводит к более быстрому алгоритму, выполняющему только три рекурсивных вызова.

В листинге 6.8 приведена реализация метода Карацубы вместе со вспомогательной функцией, вычисляющей количество двоичных разрядов неотрицательного целого числа n, равное $\lfloor \log_2 n \rfloor + 1$ для $n \geqslant 1$.

Листинг 6.8. Быстрый алгоритм Карацубы умножения двух неотрицательных целых чисел

```
def number_of_bits(n):
    if n < 2:
        return 1
    else:
        return 1 + number_of_bits(n >> 1)


def multiply_karatsuba(x, y):
    if x == 0 or y == 0:
        return 0
    elif x == 1:
        return y
    elif y == 1:
        return x
    else:
        n_bits_x = number_of_bits(x)
        n_bits_y = number_of_bits(y)

        m = min(n_bits_x // 2, n_bits_y // 2)

        a = x >> m
        b = x - (a << m)
        c = y >> m
        d = y - (c << m)

        ac = multiply_karatsuba(a, c)
        bd = multiply_karatsuba(b, d)
        t = multiply_karatsuba(a + b, c + d) - ac - bd

        return (ac << (2 * m)) + (t << m) + bd
```

Размер задачи – $\min(b_x, b_y)$. Таким образом, начальные условия должны проверить входы на равенство 0 или 1, приводящие к тривиальным результатам. Рекурсивные условия реализуют (6.4) тремя рекурсивными вызовами.

Что касается эффективности этого алгоритма, то предположим, что оба входа имеют одинаковое количество двоичных разрядов n, являю-

щееся степенью двух (то есть $n = 2^k$). Такое допущение возможно, поскольку к входным целочисленным значениям всегда можно добавить ведущие нули так, чтобы предположение выполнилось. В этом случае время выполнения алгоритма описывается функцией:

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ 3T(n/2) + c^n + d, & \text{если } n > 1. \end{cases}$$

Слагаемое $c^n + d$ возникает из-за сложений, вычитаний, операций сдвига и вызовов `number_of_bits`, которые выполняются за линейное время относительно n. Применяя основную теорему, $T(n) \in \Theta(n \log_2 3) = \Theta(n \cdot 1{,}585\ldots)$. Таким образом, этот метод эффективнее подхода, связанного с (6.3), чья стоимость вычислений описывается функцией:

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ 4T(n/2) + e^n + f, & \text{если } n > 1, \end{cases}$$

где $T(n) \in \Theta(n^2)$.

Алгоритм Карацубы использует меньше умножений (соответствующих рекурсивным вызовам), чем школьный метод, но выполняет больше сложений и вычитаний. На практике алгоритм будет быстрее для больших значений n. Однако если n мало, дополнительные операции могут сделать его медленнее традиционного метода. В любом случае, метод примечателен тем, что он может сократить количество операций умножения, которые значительно дороже сложения или вычитания.

6.5. Умножение матриц

Две матрицы можно перемножить, предварительно разбив их на блоки (подматрицы), как описано в разделе 2.4. Применив принцип «разделяй и властвуй», рассмотрим теперь простой рекурсивный метод, требующий n^3 элементарных умножений для вычисления произведения двух матриц размерности $n \times n$ (используя спецпакет NumPy языка Python). Кроме того, рассмотрим алгоритм Штрассена, требующий для получения результата примерно $n^{2{,}8}$ элементарных умножений.

6.5.1. Умножение матриц методом «разделяй и властвуй»

Пусть **A** и **B** – матрицы размерностей $p \times q$ и $q \times r$ соответственно. Размер задачи зависит от этих трех размерностей p, q и r. Тривиальное начальное условие имеет место при $p = q = r = 1$, когда результат – просто

число. Кроме того, некоторые реализации могут рассматривать размерности 0. В таких случаях результатом должна быть пустая матрица, о чём мы также коротко скажем.

Довольно простой способ декомпозиции задачи – разделить каждую матрицу на четыре блока (образующих блочную матрицу размерности 2×2). В этом случае их произведение определяется следующим образом:

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{2,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{pmatrix}. \quad (6.5)$$

Обратите внимание, что формула похожа на умножение двух матриц размерности 2×2. Например, верхний левый блок результата ($\mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$) можно считать произведением первой строки (блоков) **A** и первого столбца (блоков) **B**.

Декомпозиция приводит к вычислению восьми более простых произведений матриц. Таким образом, в рекурсивном условии метод вызовет себя восемь раз. Результат каждого произведения должен добавляться и помещаться в стек так, чтобы получить матрицу-результат. В листинге 6.9 приводится возможная реализация.

Рекурсивное условие сначала определяет каждую из меньших блочных матриц, добавляет их произведения и выстраивает матрицу-результат методами *vstack* и *hstack*. Одно из начальных условий вычисляет простое произведение при $p = q = r = 1$. Кроме того, в коде учитывается также случай пустых входных матриц, поскольку они могут появиться в рекурсивном условии при разбиении матрицы, одна из размерностей которой равна 1 (очевидно, вектор не может быть разбит на четыре вектора согласно (6.5)). Таким образом, если какая-то из размерностей равна 0, в соответствующем начальном условии создаётся пустая матрица размерности $p \times r$, которая должным образом обрабатывается средствами языка Python.

Предыдущий метод создаёт матрицы размерности 1×1 (в начальном условии) и постепенно накапливает их в стеке для формирования конечной матрицы размерности $p \times r$. Кроме того, отметим, что в методы не передаются размерности входных матриц.

Другой более эффективный способ состоит в передаче матриц **A** и **B** целиком в каждом вызове метода, тогда как сами перемножаемые блоки задаются как дополнительные параметры своими границами внутри

этих матриц (см. листинг 1.6). Кроме того, результат можно сохранить в матрице-параметре **C** размерности $p \times r$ (передаваемой по ссылке). В листинге 6.10 приведена возможная реализация этого альтернативного решения.

Листинг 6.9. Умножение матриц по принципу «разделяй и властвуй»

```
import numpy as np


def matrix_mult(A, B):
    p = A.shape[0]
    q = A.shape[1]
    r = B.shape[1]

    if p == 0 or q == 0 or r == 0:
        return np.zeros((p, r))
    elif p == 1 and q == 1 and r == 1:
        return np.matrix([[A[0, 0] * B[0, 0]]])
    else:
        A11 = A[0:p // 2, 0:q // 2]
        A21 = A[p // 2:p, 0:q // 2]
        A12 = A[0:p // 2, q // 2:q]
        A22 = A[p // 2:p, q // 2:q]

        B11 = B[0:q // 2, 0:r // 2]
        B21 = B[q // 2:q, 0:r // 2]
        B12 = B[0:q // 2, r // 2:r]
        B22 = B[q // 2:q, r // 2:r]

        C11 = matrix_mult(A11, B11) + matrix_mult(A12, B21)
        C12 = matrix_mult(A11, B12) + matrix_mult(A12, B22)
        C21 = matrix_mult(A21, B11) + matrix_mult(A22, B21)
        C22 = matrix_mult(A21, B12) + matrix_mult(A22, B22)

        return np.vstack([np.hstack([C11, C12]),
                          np.hstack([C21, C22])])


A == np.matrix([[2, 3, 1, -3], [4, -2, 1, 2]])
B == np.matrix([[2, 3, 1], [4, -1, -5], [0, -6, 3], [1, -1, 1]])
print(matrix_mult(A, B))
```

Листинг 6.10. Альтернативное умножение матриц по принципу «разделяй и властвуй»

```
import numpy as np


def add_matrices_limits(A, B, C, lp, up, lr, ur):
    for i in range(lp, up + 1)
        for k in range(lr, ur + 1)
            C[i, k] = A[i, k] + B[i, k]

def matrix_mult_limits(A, B, C, lp, up, lq, uq, lr, ur):
    mp = (lp + up) // 2
    mq = (lq + uq) // 2
    mr = (lr + ur) // 2

    if lp == up and lq == uq and lr == ur:
       C[mp, mr] = A[mp, mq] * B[mq, mr]
    elif lp <= up and lq <= uq and lr <= ur:

        M1 = np.zeros((A.shape[0], B.shape[1]))
        M2 = np.zeros((A.shape[0], B.shape[1]))

        matrix_mult_limits(A, B, M1, lp, mp, lq, mq, lr, mr)
        matrix_mult_limits(A, B, M2, lp, mp, mq + 1, uq, lr, mr)
        add_matrices_limits(M1, M2, C, lp, mp, lr, mr)

        matrix_mult_limits(A, B, M1, lp, mp, lq, mq, mr + 1, ur)
        matrix_mult_limits(A, B, M2, lp, mp, mq + 1, uq, mr + 1, ur)
        add_matrices_limits(M1, M2, C, lp, mp, mr + 1, ur)

        matrix_mult_limits(A, B, M1, mp + 1, up, lq, mq, lr, mr)
        matrix_mult_limits(A, B, M2, mp + 1, up, mq + 1, uq, lr, mr)
        add_matrices_limits(M1, M2, C, mp + 1, up, lr, mr)

        matrix_mult_limits(A, B, M1, mp + 1, up, lq, mq, mr + 1, ur)
        matrix_mult_limits(
               A, B, M2, mp + 1, up, mq + 1, uq, mr + 1, ur)
        add_matrices_limits(M1, M2, C, mp + 1, up, mr + 1, ur)

def matrix_mult_limits_vrapper(A, B):
    C = np.zeros((A.shape[0], B.shape[1]))
    matrix_mult_limits(A, B, C, 0, A.shape[0] - 1,
                       0, A.shape[1] - 1, 0, B.shape[1] - 1)
    return C
```

Метод `matrix_mult_limits` в каждом вызове передает матрицы A и B целиком, сохраняя результат в третьем параметре – матрице C размерности $p \times r$. Остальные параметры – это нижние и верхние границы перемножаемых подматриц, связанные с размерностями p, q и r. Начальное условие в `matrix_mult_limits` выполняется, когда обе подматрицы, скажем `A[i,j]` и `B[j,k]`, – скаляры. В этом случае метод просто сохраняет их произведение в строке i и столбце k матрицы C. Метод не является функцией, так как он не *возвращает* результат (матрицу). Вместо этого он изменяет параметр C, где хранится результат. Наконец, итерационный метод `add_matrices_limits` складывает элементы подматриц, переданных в двух первых входных параметрах, и сохраняет результат в своём третьем параметре (подматрицы задаются параметрами-границами).

6.5.2. Алгоритм Штрассена умножения матриц

Самая затратная по времени арифметическая операция в предыдущих алгоритмах – это скалярное умножение в двух начальных условиях. Оба они, подобно простой итерационной версии с тройным циклом, требуют $p \cdot q \cdot r$ таких умножений. Однако небезынтересно оценить временную сложность для исходных матриц размерности $n \times n$. В этом случае время выполнения можно определить следующей функцией:

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ 8\,T(n/2) + 4\Theta(n^2), & \text{если } n > 1, \end{cases} \tag{6.6}$$

поскольку методы вызывают сами себя восемь раз и должны выполнить четыре сложения матриц стоимостью порядка n^2. Поэтому согласно основной теореме (3.28) $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$. Теперь рассмотрим алгоритм Штрассена – известный метод, который сокращает временную сложность до $\Theta(n^{\log_2 7}) = \Theta(n^{2{,}807\ldots})$.

Метод, как и стандартный алгоритм, разбивает каждую из входных матриц на четыре блока. Таким образом, **AB** = **C** можно записать как:

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix}.$$

Цель метода – определить следующие новые матрицы, которые используют всего одну операцию умножения матриц:

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).
\end{aligned} \tag{6.7}$$

После чего эти матрицы могут быть сгруппированы так, чтобы сформировать выходные блочные матрицы:

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6.
\end{aligned} \tag{6.8}$$

Таким образом, в каждом рекурсивном вызове алгоритм выполняет 7 произведений и 18 сложений (или вычитаний). Поэтому оценка времени его выполнения:

$$T(n) = \begin{cases} 1, & \text{если } n \le 1, \\ 7\,T(n/2) + 18\,\Theta(n^2), & \text{если } n > 1, \end{cases} \tag{6.9}$$

где $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2{,}807\ldots})$. Этот алгоритм для больших значений n может быть быстрее стандартного со временем выполнения $\Theta(n^3)$. Однако для малых или средних матриц он может быть медленнее из-за больших постоянных множителей, имеющих значение на практике.

Наконец, теоретически входы этого алгоритма должны быть квадратными матрицами $n \times n$, где n – степень двух. На практике эффективные реализации разбивают матрицы на множество квадратных подматриц и неоднократно применяют этот алгоритм. Более простая, но медленная альтернатива – дополнить (расширить) входные матрицы нулями до размерности $2^k \times 2^k$ (см. упражнение 6.6).

6.6. Задача укладки тримино

Тримино – многоугольник из трёх равных смежных квадратов. Без учёта вращений и зеркальных отражений тримино бывают только двух типов – типа «**I**» и типа «**L**», как показано на рис. 6.3.

Рис. 6.3. Типы тримино без вращений и симметрии

Задача укладки тримино – покрыть L-тримино квадратное поле (возможно, с «дырами») размером $n \times n$, где $n \geqslant 2$ есть степень двух. Рисунок 6.4 поясняет задачу на графическом примере.

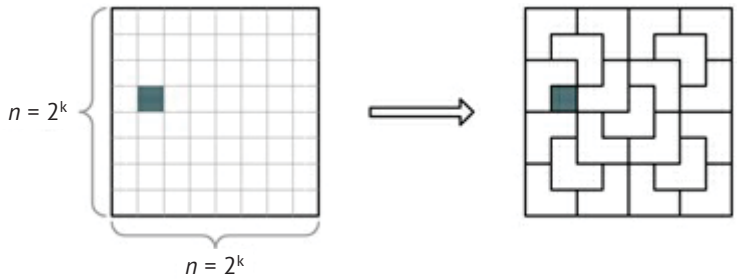


Рис. 6.4. Задача укладки тримино

Размер задачи – очевидно, n. Наименьшие экземпляры задачи соответствуют полям размера 2×2, решения для которых тривиальны. На рис. 6.5 показана декомпозиция методом «разделяй и властвуй», используемая в рекурсивном условии.

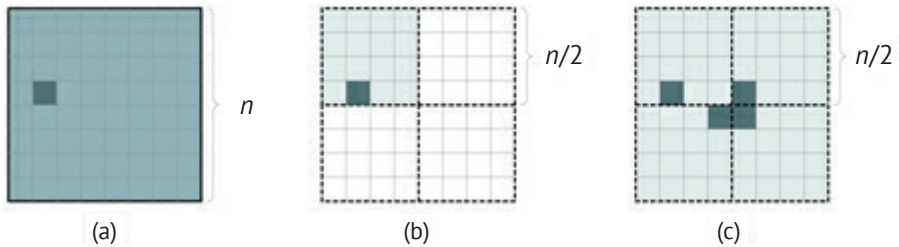


Рис. 6.5. Декомпозиция задачи укладки тримино

Исходное поле (a) поделено на четыре меньших квадратных поля размером $n/2$ (b). Однако только одно из этих меньших полей будет содержать исходную дыру. Поэтому остальные три поля не являются подзадачами, подобными исходной. Но этот вопрос решается размещением тримино в центре поля так, чтобы три его квадрата стали дырами в меньших полях и тем самым обеспечили бы подобие подзадач исходной задаче (c).

Для графического отображения решения задачи будем использовать пакет Matplotlib. В частности, само тримино можно изобразить шестью отрезками прямой. Но поскольку за счёт поворотов возможны четыре варианта L-тримино (см. рис. 6.6), в листинге 6.11 определяются четыре вспомогательные функции для каждого из таких случаев. Функции получают координаты (x, y) нижнего левого угла квадрата 2×2, охватывающего тримино. Команда `plot`([x_1, x_2], [y_1, y_2], '*k*-'), чертит отрезок прямой между точками (x_1, y_1) и (x_2, y_2).

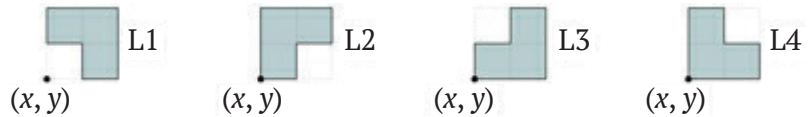


Рис. 6.6. Вращения L-тримино

В листинге 6.12 приводится возможная реализация рекурсивного метода. Процедура должна знать, какую задачу/подзадачу она должна решить. Эта информация задаётся первыми тремя её параметрами. Первые два – это координаты левой нижней точки (x, y) поля, а третий – размер поля (n). Последние два параметра задают положение дырки (в частности, (p, q) определяют левый нижний угол квадрата 1×1). И начальные, и рекурсивные условия используются методом для определения относительного положения дырки и рисования соответствующего тримино. Наконец, в рекурсивном условии метод вызывает себя четырежды с разными параметрами для разных подзадач вместе с новыми дырами в трех из них.

В заключительном листинге 6.13 приводится фрагмент кода, который можно использовать для вызова метода тримино. Строка 7 создает фигуру, строка 8 устанавливает белый цвет её фона, а `ax` в строке 9 фиксирует оси внутри фигуры. После определения размера исходного поля (строка 10) в нём наугад выбирается и рисуется дыра (строка 13). Если используется пакет Matplotlib, то прямоугольник можно нарисовать вызовом метода Rectangle. Он получает координаты левой нижней вершины вместе с шириной, высотой и, возможно, другими параметрами. В строке 14 вызывается основной метод, а последние строки подавляют масштабирование, убирают оси внутри фигуры и рисуют её.

Листинг 6.11. Вспомогательные функции для рисования тримино

```
def draw_L1(x, y):
    plt.plot = ([x, x + 2], [y + 2, y + 2], 'k-')
    plt.plot = ([x, x + 1], [y + 1, y + 1], 'k-')
    plt.plot = ([x + 1, x + 2], [y, y], 'k-')
    plt.plot = ([x, x], [y + 1, y + 2], 'k-')
    plt.plot = ([x + 1, x + 1], [y, y + 1], 'k-')
    plt.plot = ([x + 2, x + 2], [y, y + 2], 'k-')


def draw_L2(x, y):
    plt.plot = ([x, x + 2], [y + 2, y + 2], 'k-')
    plt.plot = ([x, x + 1], [y, y], 'k-')
    plt.plot = ([x + 1, x + 2], [y + 1, y + 1], 'k-')
    plt.plot = ([x, x], [y, y + 2], 'k-')
    plt.plot = ([x + 1, x + 1], [y, y + 1], 'k-')
    plt.plot = ([x + 2, x + 2], [y + 1, y + 2], 'k-')


def draw_L3(x, y):
    plt.plot = ([x, x + 2], [y, y], 'k-')
    plt.plot = ([x, x + 1], [y + 1, y + 1], 'k-')
    plt.plot = ([x + 1, x + 2], [y + 2, y + 2], 'k-')
    plt.plot = ([x, x], [y, y + 1], 'k-')
    plt.plot = ([x + 1, x + 1], [y + 1, y + 2], 'k-')
    plt.plot = ([x + 2, x + 2], [y, y + 2], 'k-')


def draw_L4(x, y):
    plt.plot = ([x, x + 2], [y, y], 'k-')
    plt.plot = ([x, x + 1], [y + 2, y + 2], 'k-')
    plt.plot = ([x + 1, x + 2], [y + 1, y + 1], 'k-')
    plt.plot = ([x, x], [y, y + 2], 'k-')
    plt.plot = ([x + 1, x + 1], [y + 1, y + 2], 'k-')
    plt.plot = ([x + 2, x + 2], [y, y + 1], 'k-')
```

Листинг 6.12. Рекурсивный метод рисования тримино

```
def trominoes(x, y, n, p, q)
    if n == 2:
        if y == q:     # hole in bottom tiles
            if x == p:     # hole in bottom-left tile
               draw_L1(x, y)
            else:     # hole in bottom-right tile
               draw_L2(x, y)
        else:    # hole in top tiles
            if x == p:     # hole in top-left tile
               draw_L3(x, y)
            else:     # hole in top-right tile
               draw_L4(x, y)

    else:
        mid_x = x + n // 2
        mid_y = y + n // 2

        if q < mid_y:     # hole in bottom squares

            if p < mid_x:     # hole in bottom-left square
                draw_L1(mid_x - 1, mid_y - 1)
                trominoes(x, y, n // 2, p, q)
                trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
                trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
                trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)
            else:     # hole in bottom-right square
                draw_L2(mid_x - 1, mid_y - 1)
                trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
                trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
                trominoes(mid_x, y, n // 2, p, q)
                trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)

        else:    # hole in top squares

            if p < mid_x:     # hole in top-left square
                draw_L3(mid_x - 1, mid_y - 1)
                trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
                trominoes(x, mid_y, n // 2, p, q)
                trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
                trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)
            else:     # hole in top-right square
                draw_L4(mid_x - 1, mid_y - 1)
                trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
                trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
                trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
                trominoes(mid_x, mid_y, n // 2, p, q)
```

Листинг 6.13. Вызов метода тримино

```
import random
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# Include tromino methods here

fig = plt.figure()
fig.patch.set_facecolor('white')
ax = plt.gca()
n = 16    # power of 2
p = random.choice([i for i in range(n)])
q = random.choice([i for i in range(n)])
ax.add_patch(Rectangle((p, q), 1, 1, facecolor=(0.5, 0.5, 0.5)))
trominoes(0, 0, n, p, q)
plt.axis('equal')
plt.axis('off')
plt.show()
```

6.7. Задача очертания

Задача очертания заключается в том, чтобы определить контур на фоне неба ряда прямоугольных зданий. Рисунок 6.7 поясняет задачу. Вход задачи – список из $n \geqslant 1$ прямоугольников, изображающих здания (см. рис. 6.7(a)). Нижняя сторона прямоугольников всегда расположена на уровне 0. Таким образом, каждое здание можно определить всего тремя параметрами. В данном случае мы будем использовать кортежи вида (x_1, x_2, h), где x_1 – положение левой стены здания, x_2 – положение его правой стены, а h – его высота, причём $x_1 < x_2$ и $h > 0$. Например, [(1,7,7), (18,20,7), (2,9,5), (17,19,2), (12,24,3), (3,8,8), (11,13,5), (15,21,6)]. Отметим, что нет необходимости сортировать здания каким-либо образом (скажем, по значению x_1).

Контур, выделенный жирной линией на рис. 6.7(b), – это ломаная линия, высота которой в любой точке оси X является высотой максимально высокого здания в этой точке. Так как здания суть прямоугольники, то их контур можно определить набором координат (x, h) на плоскости, которые задают точки x изменения высоты h, как показано на рис. 6.7(c). Таким образом, результат задачи – упорядоченный по возрастанию x список пар координат, которые можно представить кортежами. Для приведённого выше примера результатом будет список [(1,7), (3,8), (8,5), (9,0), (11,5), (13,3), (15,6), (18,7), (20,6), (21,3), (24,0)].

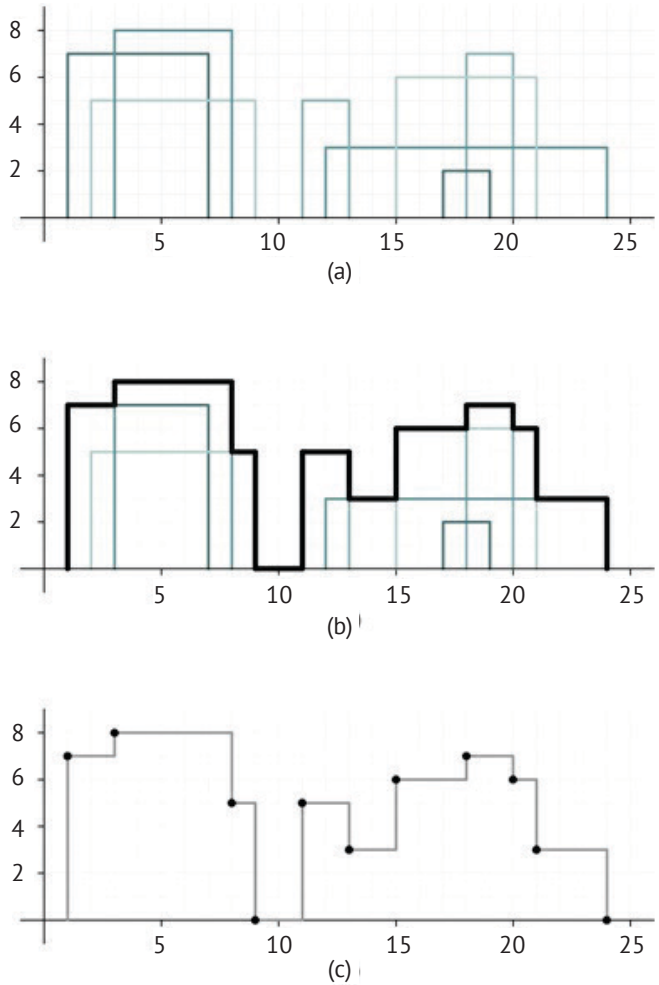


Рис. 6.7. Задача очертания

Размер задачи – количество зданий n. Начальному условию соответствует наименьший экземпляр задачи – одно здание. Если оно задано кортежем (x_1, x_2, h), то результатом будет список $[(x_1, h), (x_2, 0)]$, как показано на рис. 6.8.

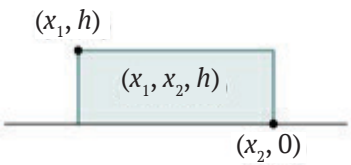


Рис. 6.8. Начальное условие задачи очертания для одного здания

Декомпозицию этой задачи можно осуществить уменьшением её размера на 1. Однако это приводит к алгоритму со временем выполнения в худшем случае $O(n^2)$. Вместо него мы применим подход «разделяй и властвуй», который позволяет решить задачу за время $\Theta(n \log n)$. Рисунок 6.9 иллюстрирует эту идею, которая подобна подходу, применённому в алгоритме сортировки слиянием.

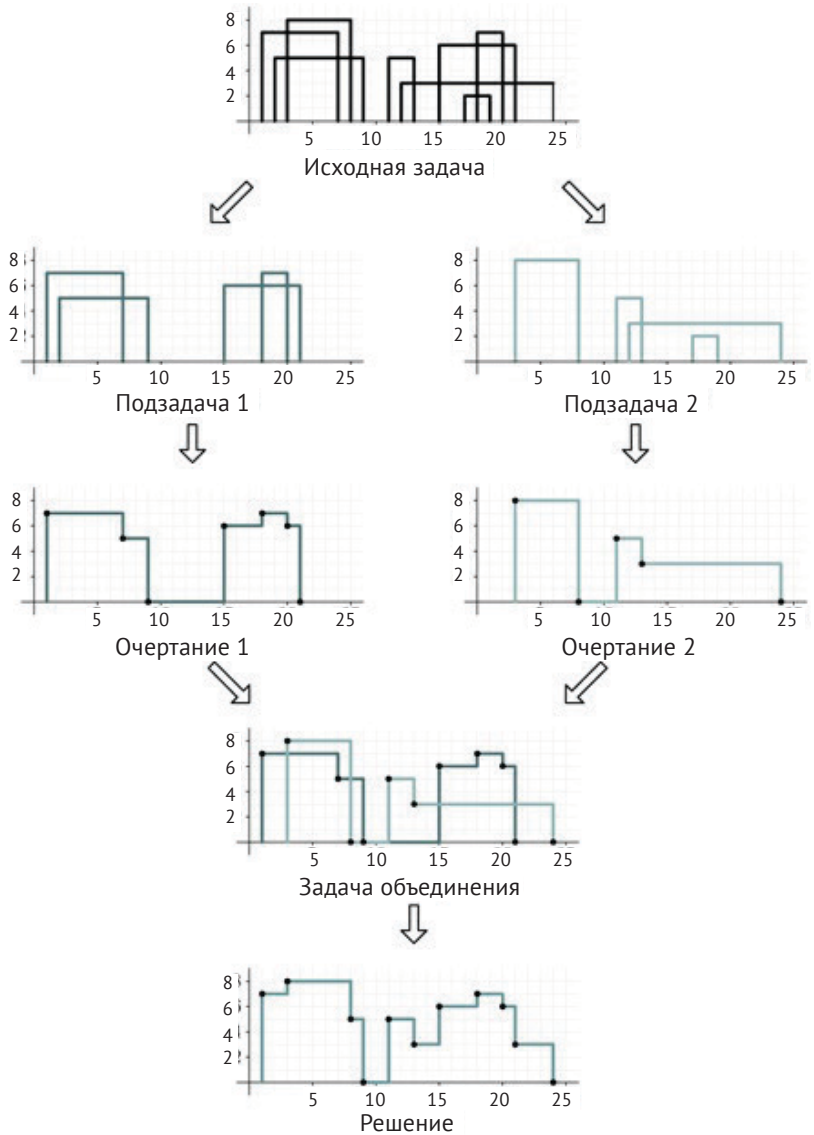


Рис. 6.9. Рекурсивное условие задачи очертания

Шаг декомпозиции заключается в делении входного списка на два меньших подсписка примерно из $n/2$ зданий. Тогда метод выполняет два рекурсивных вызова с этими подсписками, которые возвращают два независимых контура. Предположив на основании индукции, что контуры были созданы правильно, последним и главным для получения результата шагом будет объединение контуров. В листинге 6.14 приводится отвечающий принципу «разделяй и властвуй» метод, структура которого, в сущности, совпадает с таковой в листинге 6.2.

Листинг 6.14. Основной рекурсивный метод вычисления очертания

```
1  def compute_skyline(buildings):
2      n = len(buildings)
3      if n == 1:
4          return ([(buildings[0][0], buildings[0][2]),
5                   (buildings[0][1], 0)])
6      else:
7          skylines1 = compute_skyline(buildings[0:n // 2])
8          skylines2 = compute_skyline(buildings[n // 2:n])
9          return merge_skylines(skyline1, skyline2, 0, 0)
```

Задача объединения контуров – новая отдельная вычислительная задача. В то время как большинство решений в различных источниках является итерационными, мы предложим линейно-рекурсивный метод. Его входы – два сортированных списка кортежей, представляющих собой контур. Кроме того, поскольку кортеж задаёт изменение высоты контура, метод, прежде чем изменить высоту, должен иметь доступ к предыдущей высоте. Более того, поскольку предлагаемый алгоритм обрабатывает список с первого кортежа (до исчерпания списка), последовательно удаляя обработанные кортежи, предыдущих высот не будет во входных списках. Таким образом, метод требует двух дополнительных параметров, скажем p_1 и p_2, для сохранения предыдущих высот контура. Естественно, оба этих параметра должны получить начальное значение 0 при первом вызове метода из основной функции (см. строку 9 листинга 6.14).

Размер задачи зависит от длин входных списков контуров, скажем n_1 и n_2. Можно считать, что начальное условие выполняется, когда один из списков пуст, и метод в таком случае должен тривиально вернуть второй список. В листинге 6.15 приведён код, где начальные условия описаны в строках 2–5.

Листинг 6.15. Рекурсивный метод объединения контуров

```
def merge_skylines(sky1, sky2, p1, p2):
    if sky1 == []:
        return sky2
    elif sky2 == []
        return sky1
    else:
        x1 = sky1[0][0]
        x2 = sky2[0][0]
        h1 = sky1[0][1]
        h2 = sky2[0][1]

        if x1 == x2:
            h = max(p1, p2)
            new_h = max(h1, h2)
            if h == new_h:
                return merge_skylines(sky1[1:], sky2[1:],
                                      h1, h2)
            else:
                return ([(x1, new_h)]
                        + merge_skylines(sky1[1:], sky2[1:],
                                         h1, h2))

        elif x1 < x2
            if h1 > p2:
                return ([(x1, h1)]
                        + merge_skylines(sky1[1:], sky2,
                                         h1, p2))
            elif p1 > p2:
                return ([(x1, p2)]
                        + merge_skylines(sky1[1:], sky2,
                                         h1, p2))
            else:
                return merge_skylines(sky1[1:], sky2,
                                      h1, p2))

        else:
            if h1 > p1:
                return ([(x2, h2)]
                        + merge_skylines(sky1, sky2[1:],
                                         p1, h2))
            elif p2 > p1:
                return ([(x2, p1)]
                        + merge_skylines(sky1, sky2[1:],
                                         p1, h2))
            else:
                return merge_skylines(sky1, sky2[1:], p1, h2))
```

Ключом для определения подходящей декомпозиции является то, что результат объединяющей функции – это новое очертание (контур), кортежи которого отсортированы в порядке возрастания значения x. Таким образом, алгоритм анализирует первые кортежи каждого списка и обрабатывает тот, что имеет меньшее значение x (или оба, если их значения x равны). Значит, в рекурсивных условиях, помимо первых кортежей контуров (x_1, h_1) и (x_2, h_2), нам нужны ещё и их предыдущие высоты p_1 и p_2.

Давайте сначала рассмотрим случай, когда $x_1 = x_2 = x$. Так как кортежи маркируют изменения в горизонте, нам, возможно, придётся включать в решение кортеж с наибольшей высотой. Например, на рис. 6.10(a) точка (x, h_2) должна быть включена в конечный горизонт.

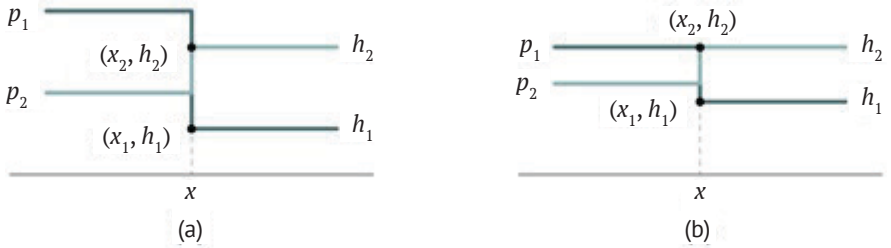


Рис. 6.10. Случаи объединения контуров, когда их высота меняется в одной и той же точке x

Далее рекурсивный вызов будет использовать хвосты обоих входных списков без (x, h_1) и (x, h_2), так как возможные изменения в точке x уже обработаны правильно. Это делается в строках 19–21. Наконец, бывают ситуации, когда новый кортеж не включается в решение. Это происходит, когда новая наибольшая высота совпадает с предыдущей наибольшей высотой контура (то есть когда $\max(h_1, h_2) = \max(p_1, p_2)$), так как в точке x высота не меняется. Рисунок 6.10(b) иллюстрирует этот случай, где точка (x, h_2) не должна включаться в конечный контур (см. строки 16 и 17).

Теперь проанализируем возможные сценарии, когда значения x первых кортежей горизонтов не равны. Без потери общности допустим, что $x_1 < x_2$. В этом случае алгоритм должен решить, включать ли кортеж (x_1, h_1), или (x_1, p_2), или ни тот, ни другой, как показано на рис. 6.11.

Если $h_1 > p_2$, то в точке x_1 первый контур выше второго, и поэтому в объединение должен быть включён кортеж (x_1, h_1) (см. строки 25–27). Если же $h_1 \leqslant p_2$, то алгоритм должен проверить условие $p_1 > p_2$. Если условие истинно, то метод включает кортеж (x_1, p_2) (см. строки 29–31). Об-

ратите внимание, что при $h_1 < p_2$ создаётся новый кортеж, которого нет во входных списках контуров. Случаи $p_2 \geqslant h_1$ и $p_2 \geqslant p_1$ означают, что объединённый контур не меняется в x_1. В итоге, обработав первый кортеж первого контура (x_1, h_1), метод отбрасывает его при вызове самого себя в соответствующих рекурсивных условиях. Кроме этого, аргументами, задающими предыдущие высоты горизонтов, будут h_1 и p_2 (см. строки 27, 31 и 34). Остальная часть кода в строках 36–46 похожа на код в строках 23–34 и обрабатывает случай $x_2 < x_1$.

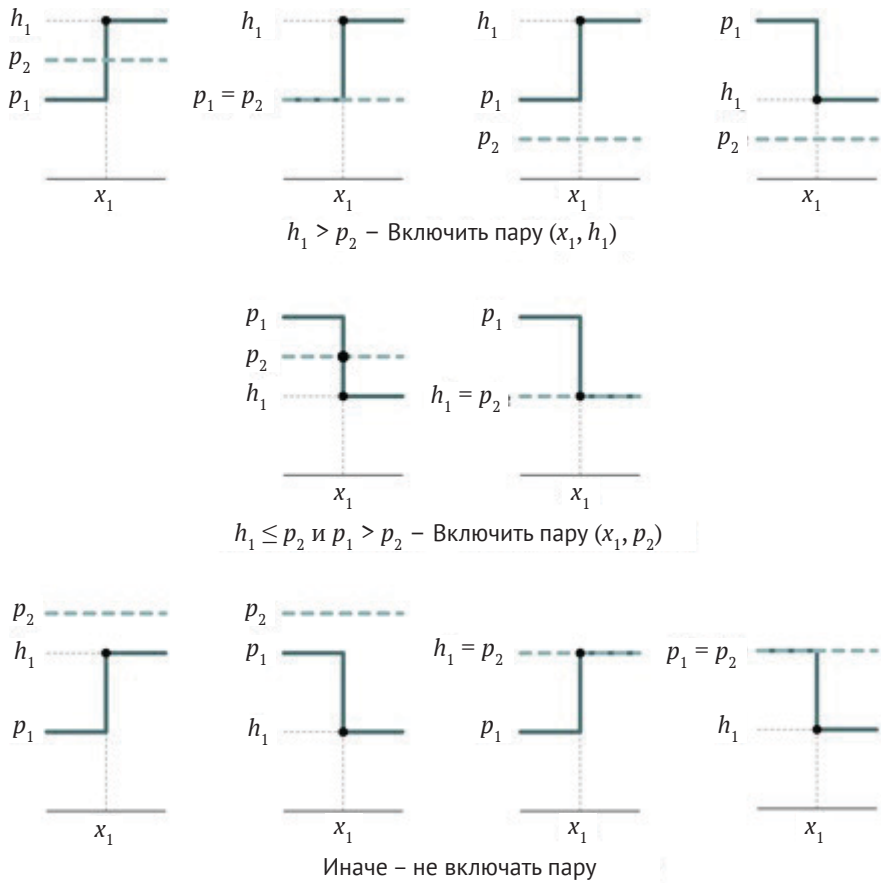


Рис. 6.11. Возможные варианты слияния контуров при $x_1 < x_2$

6.8. Упражнения

Упражнение 6.1. Применив подход «разделяй и властвуй», реализуйте алгоритм, определяющий наличие в списке элемента x.

Упражнение 6.2. Пусть **a** – список из *n* неотрицательных целых чисел. Опираясь на подход «разделяй и властвуй», напишите функцию, возвращающую множество цифр, входящих во все значения элементов **a**, и определите её асимптотическую стоимость вычисления. Например, для **a** = [2**3**48, 1**3**49, 752**3**, **3**215] результатом будет {3}. Функция должна вызывать другую функцию, которая возвращает множество цифр неотрицательного целого числа. Реализуйте и её тоже.

Упражнение 6.3. Задача о максимальном подсписке заключается в поиске внутри списка чисел непрерывной цепочки элементов (подсписка) с наибольшей суммой их значений. Например, для списка [–1, –4, 5, 2, –3, 4, 2, –5] максимальный подсписок – [5, 2, –3, 4, 2] с суммой значений 10. Считая исходный список чисел непустым, реализуйте с применением подхода «разделяй и властвуй» функцию, возвращающую сумму элементов максимального подсписка.

Упражнение 6.4. Разработайте рекурсивный алгоритм быстрого умножения полиномов, основанный на декомпозиции способом «разделяй и властвуй», подобной той, что применялась в алгоритме Карацубы (см. раздел 6.4). Полиномы задаются списками своих коэффициентов, как описано в упражнении 5.2. Метод должен вызвать функции сложения и вычитания полиномов. Реализуйте также и эти функции.

Упражнение 6.5. Реализуйте рекурсивную функцию, которая получает матрицу **A** размерности $n \times m$ и транспонирует её в матрицу размерности $m \times n$ ($\mathbf{A}^{\mathsf{T}}$). Используйте пакет NumPy и следующую декомпозицию способом «разделяй и властвуй», которая разбивает исходную матрицу делением каждой её размерности пополам на четыре блока:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix}.$$

В этом случае транспонирование **A** можно определить как

$$\mathbf{A}^{\mathsf{T}} = \begin{pmatrix} \mathbf{A}_{1,1}^{\mathsf{T}} & \mathbf{A}_{2,1}^{\mathsf{T}} \\ \mathbf{A}_{1,2}^{\mathsf{T}} & \mathbf{A}_{2,2}^{\mathsf{T}} \end{pmatrix}.$$

Упражнение 6.6. Реализуйте алгоритм Штрассена для умножения матриц. Код должен включать функцию-оболочку, которая позволяет умножать матрицы размерности $p \times q$ на матрицы размерности $q \times r$, дополняя их нулями до квадратных.

Упражнение 6.7. Реализуйте алгоритм умножения матриц, разбив их следующим образом:

$$\mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \end{pmatrix} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2.$$

Упражнение 6.8. Реализуйте алгоритм умножения матриц, разбив входные матрицы следующим образом:

$$\begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_1 & \mathbf{B}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 \\ \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2 \end{pmatrix}.$$

Глава 7

Множественная рекурсия II: пазлы, фракталы и прочее

Вычисления, электрические батареи, телефон, паровой двигатель, радио – все эти революционные инновации натолкнулись на массу изобретателей, работающих параллельно и ничего не знающих друг о друге.

– Стивен Джонсон

В предыдущей главе были представлены алгоритмы, опирающиеся на принцип «разделяй и властвуй» и разбивающие задачу на несколько частей. В этой главе предлагаются решения задач повышенной сложности, которые тоже разбиваются на несколько подзадач, но уменьшают их размер на 1 или 2. Предыдущие примеры таких алгоритмов – функция (1.2), определяющая числа Фибоначчи, или (3.2), вычисляющая биномиальные коэффициенты (в упражнении 7.1 предлагается реализовать такую функцию). Настоящая глава включает классическую задачу «Ханойская башня» – излюбленный пример для иллюстрации множественной рекурсии. Кроме того, в главе рассматриваются задачи о фракталах, для рисования которых используется популярный модуль `Matplotlib`.

7.1. Путь через болото

В этой задаче (также известной как «проход по трясине») прямоугольное болото из квадратных участков задаётся матрицей A размерности $n \times m$

и начальной строкой r. Элементы матрицы ($a_{i,j}$) могут иметь только два значения – «суша» («кочка») или «топь». Цель задачи – найти путь через болото (от левого его края до правого), начав с заданного на его левом краю участка суши $a_{r,0}$, согласно следующим правилам (см. рис. 7.1(a)):

- путь через болото может проходить только по суше;
- продвигаться можно только вправо и только по смежным участкам суши. Например, с участка $a_{i,j}$ можно шагнуть только на участки $a_{i-1,\,j+1}$, $a_{i,\,j+1}$ или $a_{i+1,\,j+1}$. Отметим, что движение по вертикали запрещено. Путь не может пересекать нижнего и верхнего краёв болота;
- путь должен заканчиваться на любом участке суши на правом краю болота.

Рисунок 7.1(b) показывает путь через болото, где участки суши и топи изображаются светлыми и темными квадратами соответственно.

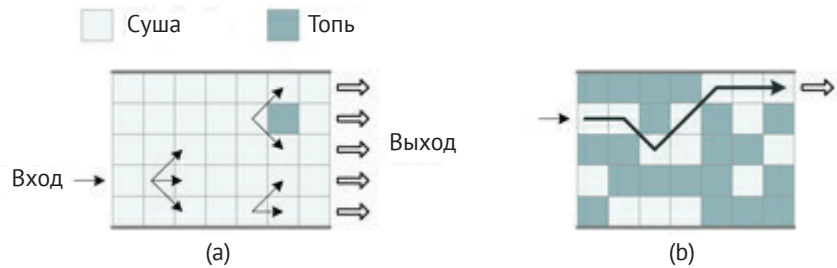


Рис. 7.1. Задача о пути по болоту

Можно предположить, что размер задачи – это ширина болота (m). В первом начальном условии метод может просто вернуть False, если r выходит за пределы болота ($r < 0$ или $r \geqslant n$) или если начальный участок $a_{r,0}$ – топь (отметим, что это последнее условие было бы ненужным, если бы мы сразу условились, что первый участок – это суша). Иначе функция должна будет вернуть True, если ширина болота равна 1 (это обеспечивается предыдущим начальным условием, проверяющим, что участок является сушей).

В листинге 7.1 приведена одна из многих возможных реализаций функции. Матрица (из пакета `NumPy`) содержит символы, где символ 'W' обозначает топь. Строки 2–5 определяют начальные условия. В строках 7, 8 и 9 метод определяет, можно ли перейти к $a_{r-1,1}$ (по диагонали вверх), $a_{r,1}$ (по горизонтали вправо) или $a_{r+1,1}$ (по диагонали вниз) соответственно.

Листинг 7.1. Функция поиска пути через болото

```
def exists_path_swamp(A, r):
    if r < 0 or r >= A.shape[0] or A[r, 0] == 'W':
        return False
    elif A.shape[1] == 1:
        return True
    else:
        return (exists_path_swamp(A[:, 1:], r - 1)
                or exists_path_swamp(A[:, 1:], r)
                or exists_path_swamp(A[:, 1:], r + 1))
```

В рекурсивном условии задача разбивается на три подзадачи размера $m - 1$ (см. рис. 7.2). В частности, можно предположить, что нам известен правильный путь через болото, начинающийся с участков $a_{r-1,1}$, $a_{r,1}$ или $a_{r+1,1}$. Иными словами, можно считать, что нам известны решения трех меньших подзадач, возникающих после отбрасывания первого столбца болота и начинающихся с участков в строках $r - 1$, r и $r + 1$. Таким образом, рекурсивное условие будет состоять из трёх соответствующих подзадачам рекурсивных вызовов.

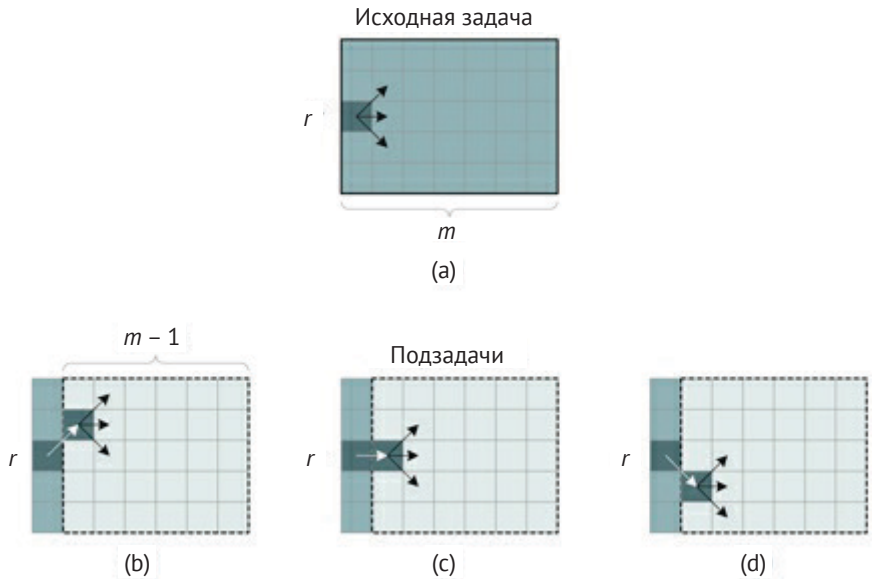


Рис. 7.2. Декомпозиция задачи о пути через болото

Интересно выяснить, как влияют на алгоритм предварительные условия. В листинге 7.2 есть предварительное условие для входа $a_{r,0} \neq$ 'W'.

Поэтому ему не нужна проверка $a_{r,0}$ = 'W' (см. строку 2 в листинге 7.1). Кроме того, любой вызов метода, которому предшествует условный оператор, должен убедиться, что $a_{r,0} \neq$ 'W'. В итоге эффективность кода возросла за счёт включения условных операторов в строках 10 и 17, которые устраняют лишние рекурсивные вызовы, если путь через болото уже найден. Например, если уже существует путь вверх по диагонали, то не нужно проверять другие направления – по горизонтали или вниз по диагонали.

Листинг 7.2. Альтернативная функция поиска пути через болото

```
1  def exists_path_swamp_alt(A, r):
2      if A.shape[1] == 1:
3          return A[r, 0] != 'W'
4      else:
5          if r == 0 or A[r - 1, 1] == 'W':
6              diag_up == False
7          else:
8              diag_up == exists_path_swamp_alt(A[:, 1:], r - 1)
9
10         if not diag_up:
11             if r == A.shape[0] - 1 or A[r + 1, 1] == 'W':
12                 diag_down == False
13             else:
14                 diag_down == exists_path_swamp_alt(
15                     A[:, 1:], r + 1)
16
17             if not diag_down:
18                 if A[r, 1] == 'W':
19                     horizontal == False
20                 else:
21                     horizontal == exists_path_swamp_alt(
22                         A[:, 1:], r)
23
24         return diag_up or diag_down or horizontal
```

Наконец, в листингах 7.1 и 7.2 размер матрицы уменьшается явно – вызовом метода с параметром A[:, 1:]. Другая возможность состоит в том, чтобы всегда передавать матрицу $n \times m$ целиком и управлять размером задачи с помощью входного параметра, задающего столбец c, с которого метод должен начать поиск пути. При таком сценарии отправной точкой пути была бы $a_{r,c}$. В результате код оказался бы более общим, позволяя найти правильный путь через болото не только из его

левого края, но и из любого участка суши $a_{r,c}$. В упражнении 7.2 предлагается реализовать этот вариант.

7.2. Ханойская башня

Игра «Ханойская башня» – одна из самых популярных задач для иллюстрации рекурсии. Несмотря на то что её можно решить с помощью относительно простых итерационных алгоритмов, она также допускает короткое и изящное рекурсивное решение, которое не только подчеркивает роль декомпозиции и индукции в рекурсии, но и демонстрирует их возможности при решении задач.

Игра состоит из трёх вертикальных стержней и n дисков, которые можно насадить на любой стержень, как показано на рис. 7.3 для $n = 7$ дисков. Диски имеют разные радиусы, и можно считать, что радиус диска i равен i.

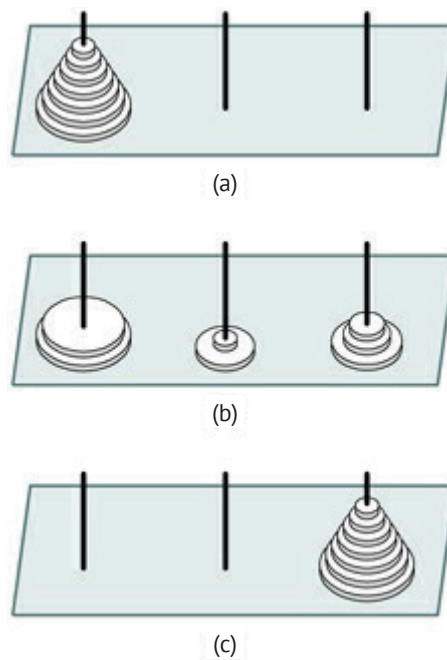


Рис. 7.3. Головоломка «Ханойская башня»

В исходном положении все n дисков насажены на один из стержней в порядке убывания (индекса или радиуса), как показано на рис. 7.3(a). Цель игры состоит в том, чтобы, используя третий (вспомогательный) стержень, переместить всю башню лежащих друг на друге дисков с со-

хранением их порядка следования с первого (начального) стержня на другой (конечный) стержень, руководствуясь следующими правилами:

- только один диск может быть перемещён за один раз;
- при каждом перемещении с одного из стержней снимается верхний диск и насаживается на другой стержень;
- на любом стержне больший диск не может лежать на меньшем.

На рис. 7.3(b) приведено положение дисков на некотором промежуточном шаге после выполнения нескольких перемещений, а на рис. 7.3(c) – заключительное состояние дисков, когда задача решена и все диски находятся на нужном стержне.

Размер задачи определяется количеством дисков. Начальное условие возникает при $n = 1$, когда решение тривиально и заключается в перемещении диска с исходного стержня на заключительный (см. рис. 7.4). На первом шаге меньший диск переносится на вспомогательный стержень, что позволяет переместить больший диск на конечный стержень. На последнем шаге меньший диск просто перемещается на конечный стержень.

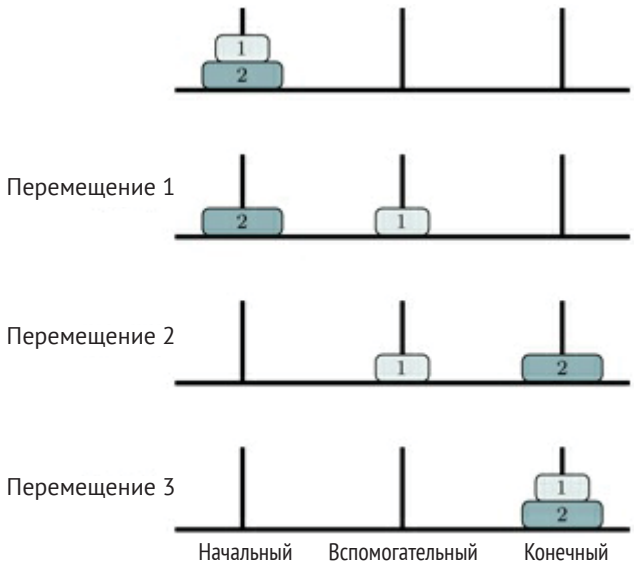


Рис. 7.4. Решение головоломки «Ханойская башня» с количеством дисков $n = 2$

Можно доказать, что минимальное число перемещений, необходимое для решения задачи с n дисками, равно $2^n - 1$. Для $n = 4$ задача может быть решена за 15 перемещений (см. рис. 7.5).

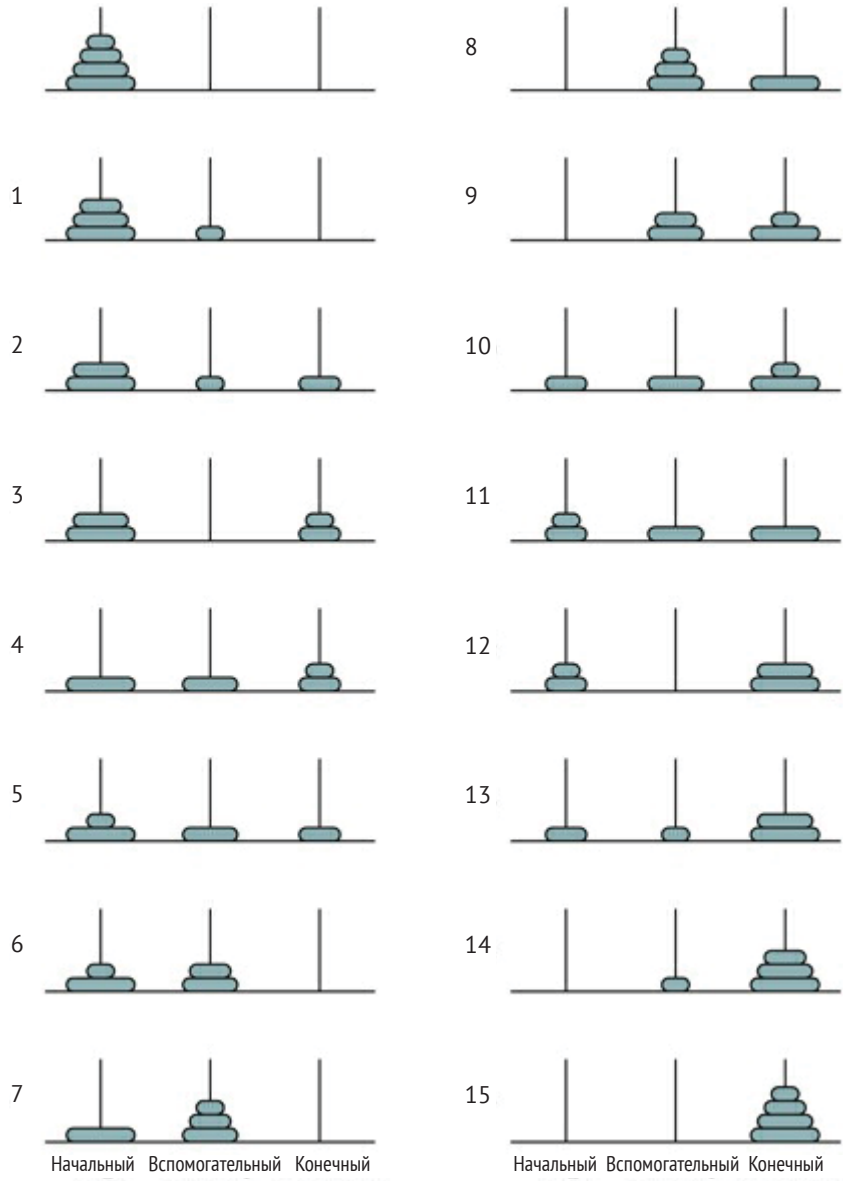


Рис. 7.5. Решение головоломки «Ханойская башня» с количеством дисков $n = 4$

Несмотря на то что число перемещений растёт экспоненциально относительно n, при использовании рекурсии сложность задачи, в сущности, та же, что для случая $n = 2$. Самые интересные конфигурации дисков при $n = 4$ – после перемещений 7 и 8. Заметим, что после 7-го

перемещения на вспомогательном стержне находится (n – 1)-й диск. Эта комбинация нужна для перемещения наибольшего диска на конечный стержень. Таким образом, первые семь шагов состоят из решения задачи для n – 1 = 3 дисков, когда цель заключается в перемещении этих трёх дисков с начального стержня на вспомогательный при посредстве конечного стержня. Другими словами, конечный и вспомогательный стержни меняются ролями. Восьмое перемещение – главная операция по перемещению наибольшего диска. После того как он помещён на конечный стержень, на вспомогательном стержне находится стопка из n – 1 дисков, которую нужно переместить на конечный стержень. Таким образом, семь оставшихся шагов – это решение другой задачи из n – 1 дисков, когда роли вспомогательного и конечного стержней меняются местами. Этот вывод, опирающийся на декомпозицию задачи и индукцию, показан на рис. 7.6.

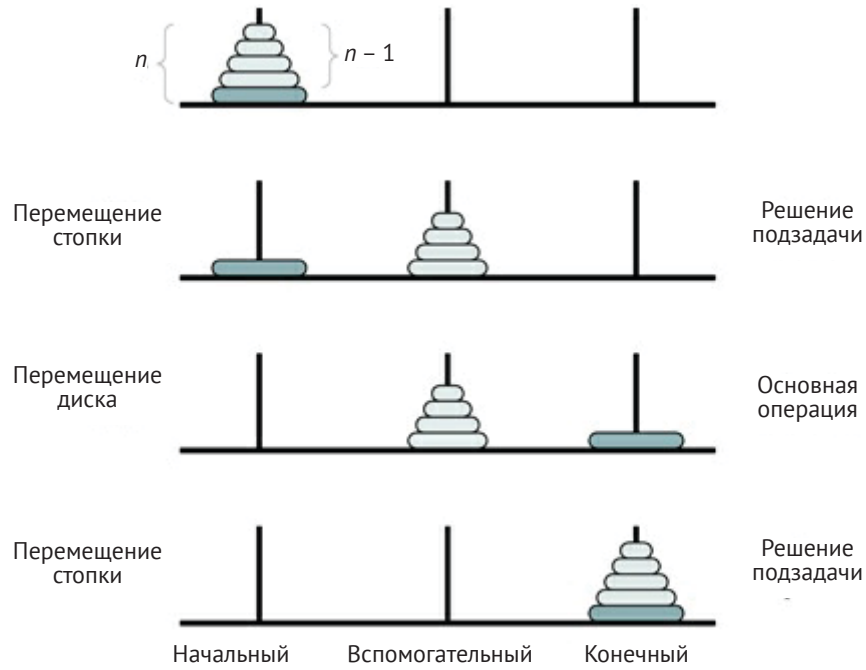


Рис. 7.6. Декомпозиция задачи «Ханойская башня»

В частности, декомпозиция рассматривает две задачи размером n – 1, решение которых состоит из трёх шагов. Если на первом и третьем шагах решается одна из подзадач, то на втором шаге наибольший диск перемещается на конечный стержень. Следовательно, рекурсивное реше-

ние может опираться на индукцию, чтобы предположить, что мы можем переместить всю стопку из $n - 1$ дисков за один шаг (соответствующий рекурсивному вызову). Отметим, что в этом случае решение концептуально подобно случаю $n = 2$. В частности, отметим подобие рис. 7.4 и 7.6. Оба решения состоят из трёх шагов, когда диск 1 заменяется стопкой из $n - 1$ дисков на рис. 7.6 (а диск 2 играет ту же роль, что диск n).

В листинге 7.3 приведена реализация процедуры, которая выводит на экран протокол перемещений дисков.

Листинг 7.3. Процедура решения задачи «Ханойская башня»

```
def towers_of_Hanoi(n, o, d, a):
    if n == 1:
        print ('Move disk', n, 'from rod', o, 'to rod', d),
    else:
        towers_of_Hanoi(n - 1, o, a, d)
        print ('Move disk', n, 'from rod', o, 'to rod', d),
        towers_of_Hanoi(n - 1, a, d, o)


def towers_of_Hanoi_alt(n, o, d, a):
    if n > 0:
        towers_of_Hanoi_alt(n - 1, o, a, d)
        print ('Move disk', n, 'from rod', o, 'to rod', d),
        towers_of_Hanoi_alt(n - 1, a, d, o)


towers_of_Hanoi(4, 'O', 'D', 'A')
```

Последняя строка кода просто вызывает метод, который выводит протокол перемещений (см. рис. 7.7) для $n = 4$. Параметры, относящиеся к стержням, нужны для правильного задания отдельных подзадач (включая исходную). В коде также приведена альтернативная процедура, рассматривающая начальное условие для $n = 0$, когда ничего делать не нужно.

7.3. Обходы дерева

Обход дерева – это процесс однократного посещения каждого из узлов дерева (понимаемого как неориентированный граф) в заданном порядке. В этом разделе мы разберём три способа обхода дерева «в глубину» в заданном порядке. Это значит, что всё поддерево узла обходится до обхода его «братьев» (элементов того же уровня) и их поддеревьев. Кроме

того, мы рассмотрим обходы дерева только слева направо, когда «дети» некоторого родительского узла обрабатываются строго слева направо. Рекурсивные обходы дерева привлекают своей простотой и не требуют ни стека, ни очереди, ни итераций.

```
Переместить диск 1 со стержня O на стержень A
Переместить диск 2 со стержня O на стержень D
Переместить диск 1 со стержня A на стержень D
Переместить диск 3 со стержня O на стержень A
Переместить диск 1 со стержня D на стержень O
Переместить диск 2 со стержня D на стержень A
Переместить диск 1 со стержня O на стержень A
Переместить диск 4 со стержня O на стержень D
Переместить диск 1 со стержня A на стержень D
Переместить диск 2 со стержня A на стержень O
Переместить диск 1 со стержня D на стержень O
Переместить диск 3 со стержня A на стержень D
Переместить диск 1 со стержня O на стержень A
Переместить диск 2 со стержня O на стержень D
Переместить диск 1 со стержня A на стержень D
```

Рис. 7.7. Результат процедуры из листинга 7.3 при решении задачи «Ханойская башня» с количеством дисков $n = 4$

7.3.1. Внутренний обход

Рассмотрим задачу печати узлов двоичного дерева поиска (см. раздел 5.3) в порядке следования его ключей. Рекурсивный алгоритм решения задачи можно разработать следующим образом. Понятно, если дерево пусто, то алгоритм не выполняет никаких действий. Это – начальное условие. Для вывода рекурсивного условия рассмотрим схему на рис. 7.8, где корень дерева отбрасывается, а дерево разбивается на левое поддерево и правое поддерево.

В первую очередь мы уверены, что любой ключ левого поддерева меньше ключа его корня. Кроме того, согласно основному свойству двоичных деревьев поиска левое поддерево – это тоже двоичное дерево поиска. Поэтому согласно принципу индукции можно считать, что рекурсивный вызов для левого поддерева, как и рекурсивный вызов для

правого поддерева, напечатает свои ключи в нужном порядке. Отсюда понятно, что метод просто должен получить результаты обеих подзадач, а между ними напечатать ключ корневого узла.

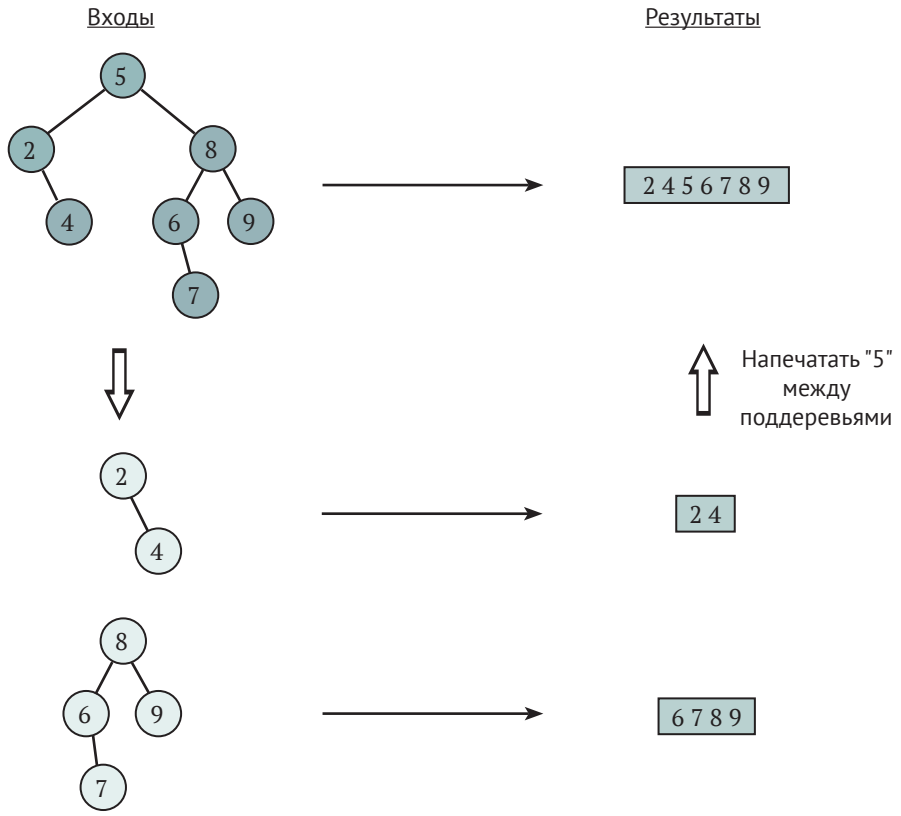


Рис. 7.8. Пример декомпозиции задачи внутреннего обхода дерева

Процедура внутреннего обхода дерева приведена в листинге 7.4, где предполагается, что каждый узел исходного двоичного дерева поиска – это список из четырех элементов (см. раздел 5.3).

Листинг 7.4. Внутренний обход двоичного дерева

```
def inorder_traversal(T):
    if T != []:
        inorder_traversal(T[2])     # process left subtree
        print (T[0], ': ', T[1], sep='')     # print key and item
        inorder_traversal(T[3])     # process right subtree
```

Например, применительно к дереву дней рождения из (5.1) этот код напечатает элементы дерева в алфавитном порядке имён (ключей) людей. Таким образом, мы говорим, что метод выполняет *внутренний* (inorder) обход двоичного дерева поиска.

7.3.2. Прямой и обратный обходы

При *прямом* (preorder) и *обратном* (postorder) порядках обхода двоичного дерева корневой ключ (и сам корневой узел) печатается не между рекурсивными вызовами, а *до* и *после* них соответственно. В листинге 7.5 приводятся методы, осуществляющие такие обходы.

Листинг 7.5. Прямой и обратный обходы двоичного дерева

```
def preorder_traversal(T):
    if T != []:
        print (T[0], ': ', T[1], sep='')     # print key and item
        inorder_traversal(T[2])     # process left subtree
        inorder_traversal(T[3])     # process right subtree


def postorder_traversal(T):
    if T != []:
        postorder_traversal(T[2])     # process left subtree
        postorder_traversal(T[3])     # process right subtree
        print (T[0], ': ', T[1], sep='')     # print key and item
```

Отметим, что процедура прямого обхода печатает ключ корневого узла всего дерева или любого из его поддеревьев перед вызовом метода с непустым деревом. Поэтому прямой обход задаёт порядок, при котором рекурсивные вызовы только предполагается выполнить (один для каждого узла двоичного дерева). В частности, метод `preorder_traversal` для списка (5.1) даст следующий результат:

```
Emma:  2002/05/23
Anna:  1999/12/03
Paul:  2000/01/13
Lara:  1987/08/23
John:  2006/05/08
Luke:  1976/07/31
Sara:  1995/03/14
```

Напротив, последовательность ключей обратного обхода отражает порядок, при котором рекурсивные вызовы уже выполнены. В частнос-

ти, обратный обход двоичного дерева дней рождения даст следующую последовательность ключей: 'Anna', 'John', 'Luke', 'Lara', 'Capa', 'Sara', 'Emma'. Заметьте, что ключ корня исходного дерева ('Emma') оказывается последним, поскольку вызов метода для дерева в целом является последним, завершающим.

7.4. Самый длинный палиндром в строке

В этой задаче в заданной исходной строке $s = s_0 s_1 \ldots s_{n-2} s_{n-1}$ длины n, где s_i – её символы, нужно найти самую длинную подстроку-палиндром. Если таких палиндромов несколько, алгоритм возвращает любой из них. Важно понимать, что *подстрока* – это непрерывная последовательность элементов s, дабы не путать её с понятием *подпоследовательности*, элементы которой не обязательно идут подряд. Кроме того, это *задача оптимизации*, цель которой – в достижении максимального значения некоторой функции от исходной строки. Такие задачи обычно решаются методами «динамического программирования» (см. раздел 10.4), которые применяются к определенному классу задач оптимизации.

Размер задачи – это длина исходной строки n. Наименьшие её экземпляры – это строки длины 0 (пустая строка) и 1 (строка из одного символа), когда алгоритм в начальном условии просто возвращает входную строку. Задачу можно упростить, отказавшись от крайних символов строки. Таким образом, декомпозиция исходной задачи на рис. 7.9(a) предполагает решение двух подзадач на рис. 7.9(b) и 7.9(c). Понятно, что если самая длинная подстрока-палиндром не содержит самый левый символ (s_0) исходной строки, то результатом будет выход метода, применённого к оставшейся подстроке ($s_{1\ldots n-1}$), независимо от того, принадлежит ли s_{n-1} решению. Точно так же, если самая длинная подстрока-палиндром не содержит s_{n-1}, алгоритм должен вернуть самую длинную подстроку-палиндром строки $s_{0\ldots n-2}$. Наконец, вся входная строка может оказаться палиндромом, что проверяется функцией из листинга 10.9, которая, помимо этого, возвращает истинное значение для начальных условий.

В листинге 7.6 приведена реализация рекурсивного метода.

Так как эта задача подобна задаче поиска самого длинного палиндрома-подсписка, код работает так же, как если бы вход был списком. Если вход s – палиндром (см. раздел 4.3.2), то результатом будет просто s, который, естественно, включает и пустую строку, и строку из одного элемента. В противном случае алгоритм ищет самый длинный палиндром в двух подстроках длины $n - 1$ и возвращает самое длинное решение. Оценку времени его выполнения можно определить как

$$T(n) = \begin{cases} 1, & \text{если } 0 \le n \le 1, \\ 2\,T(n-1) + n/2 + 1, & \text{если } n > 1, \end{cases}$$

где $n/2$ – слагаемое, связанное с вызовом `is_palindrome`. Нерекурсивное выражение $T(n) = (7/4)2^n - n/2 - 2$. Таким образом, $T(n) \in \Theta(2^n)$ имеет экспоненциальный рост.

Рис. 7.9. Декомпозиция задачи поиска в строке самого длинного палиндрома

Листинг 7.6. Поиск самого длинного палиндрома в строке

```
def longest_palindrome_substring(s):
    n = len(s)
    if is_palindrome(s):
        return s
    else:
        s_aux_1 = longest_palindrome_substring(s[1:n])
        s_aux_2 = longest_palindrome_substring(s[0:n - 1])
        if len(s_aux_1) > len(s_aux_2):
            return s_aux_1
        else:
            return s_aux_2
```

Понятно, что рекурсивный алгоритм неэффективен, так как задача может быть решена «в лоб» за время $O(n^3)$. Отметим, что существует порядка n^2 подстрок (их можно задать двумя граничными индексами, управляемыми двумя циклами), требующих порядка n операций для определения, является ли строка палиндромом. Неэффективность листинга 7.6 происходит из-за наличия многочисленных идентичных *перекрывающихся подзадач*. Например, оба рекурсивных вызова на рис. 7.9(d) решают подзадачу длины $n - 2$, что влечёт за собой повторение идентичных и потому избыточных действий. Более того, мень-

шие идентичные подзадачи решаются за показательное время. Тем не менее такие рекурсивные решения полезны на практике, так как их разработка может быть первым шагом к проектированию более эффективных алгоритмов. В разделе 10.4 обсуждается, как избежать перекрытия подзадач в рекурсивных решениях с помощью подхода, известного как мемоизация, или применения динамического программирования, которое является выдающимся методом проектирования алгоритмов. Основанные на динамическом программировании решения этой задачи могут работать за время $O(n^2)$. Наконец, было разработано несколько алгоритмов, способных решить эту задачу за линейное время.

Итак, в листинге 7.6 используется функция `is_palindrome` для определения, является ли строка палиндромом. Следующий пример заменяет эту функцию другим рекурсивным вызовом. Очевидно, строка s длины n является палиндромом, если самая длинная её подстрока-палиндром имеет длину n. Таким образом, можно проверить, является ли s палиндромом, оценивая, есть ли у самой длинной подстроки палиндрома $s_{1\ldots n-2}$ длина – 2 с $s_0 = s_{n-1}$. В листинге 7.7 приводится альтернативный метод, вызывающий только себя и использующий решение подзадачи на рис. 7.9(d). Время его выполнения:

$$T(n) = \begin{cases} 1, & \text{если } 0 \le n \le 1, \\ 2\ T(n-1) + T(n-2) + 1, & \text{если } n > 1, \end{cases}$$

чей порядок роста – $\Theta((1 + \sqrt{2})^n) = \Theta((2.41\ldots)^n)$. Так что этот метод ещё менее эффективен, чем метод в листинге 7.6, поскольку обрабатывает гораздо больше перекрывающихся идентичных подзадач.

7.5. Фракталы

Фрактал – это геометрический объект, структура которого повторяется в разных масштабах. В этом разделе мы узнаем, как генерировать снежинку Коха и ковёр Серпиньского – два первых фрактальных рисунка.

7.5.1. Снежинка Коха

Снежинка Коха – это фрактальный рисунок, состоящий из множества прямых отрезков и называемый «кривой Коха». Эта кривая создаётся многократным применением к отрезкам прямой следующего правила. Исходный отрезок прямой длины L (см. рис. 7.10) делится на 3 равных отрезка длины $L/3$, и средний из них «ломается» – заменяется двумя сторонами равностороннего треугольника с длиной стороны $L/3$ (шаг 1).

Листинг 7.7. Альтернативный метод поиска самого длинного палиндрома в строке

```
def longest_palindrome_substring_alt(s):
    n = len(s)
    if n <= 1:
        return s
    else:
        s_aux_1 = longest_palindrome_substring_alt(s[0:n - 1])
        if len(s_aux_1) == n - 2 and s[0] == s[n - 1]:
            return s
        else:
            s_aux_2 = longest_palindrome_substring_alt(s[1:n])
            s_aux_3 = longest_palindrome_substring_alt(
                s[0:n - 1])
            if len(s_aux_1) > len(s_aux_2):
                return s_aux_2
            else:
                return s_aux_3
```

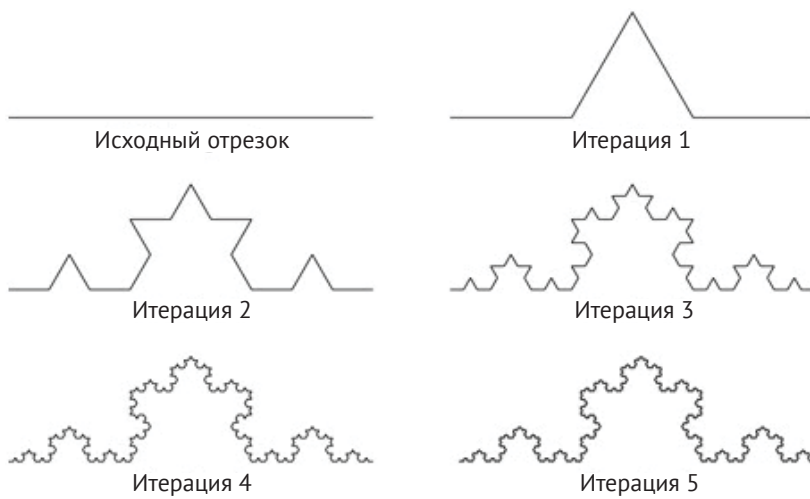


Рис. 7.10. Фрактальная кривая Коха

Таким образом, новая кривая имеет длину $4L/3$. Фрактал Коха формируется применением этого правила к каждому отрезку кривой. Так, например, кривая на шаге 2 получается применением этого правила к каждому из четырёх прямых отрезков шага 1. Новая кривая состоит уже из 16 отрезков длиной $L/9$ каждый, а её полная длина – $(4/3)^2L$. Понятно, что после n итераций длина кривой будет равна $(4/3)^nL$, которая вместе с n стремится к бесконечности.

Снежинка Коха создаётся применением этого правила к каждой из трёх сторон равностороннего треугольника с генерацией трёх кривых Коха). На рис. 7.11 показано несколько первых шагов создания снежинки Коха.

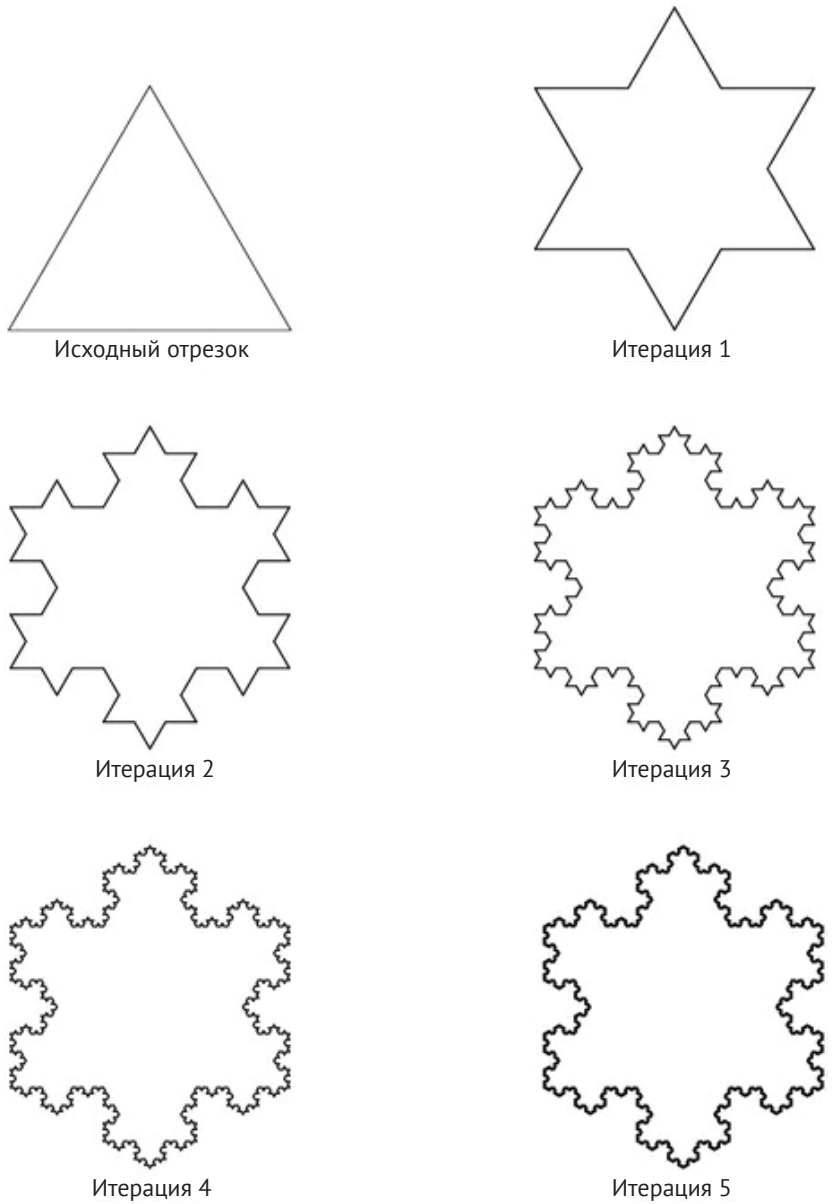


Рис. 7.11. Фрактальная снежинка Коха

С вычислительной точки зрения задача генерации кривой Коха определяется тремя входными параметрами – двумя концевыми точками p и q (координатами на плоскости) исходного отрезка прямой и количеством итераций n. Понятно, что размер задачи – n, так как он определяет количество шагов для генерации рекурсивного изображения. Начальное условие – это простейший случай $n = 0$, когда алгоритм просто чертит линию от p до q.

На рис. 7.12 показана декомпозиция задачи, которую мы будем использовать для вывода рекурсивного условия. Отметим, что кривая Коха после n шагов состоит из четырех более простых кривых Коха, созданных на шаге $n - 1$.

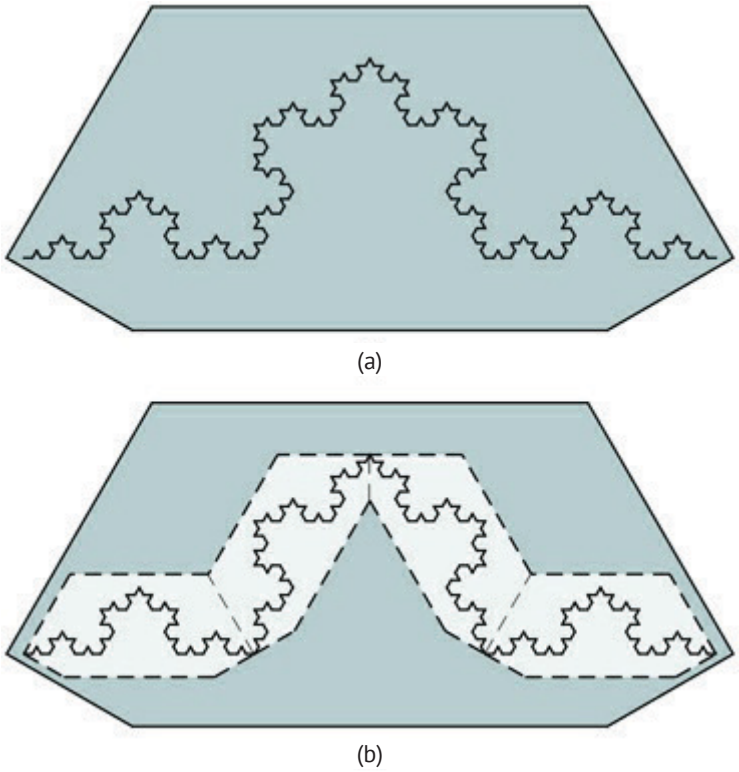


Рис. 7.12. Декомпозиция кривой Коха

Основная математическая задача состоит в том, чтобы определить концевые точки каждой меньшей кривой Коха. На рис. 7.13 показан один из способов их получения с использованием векторов.

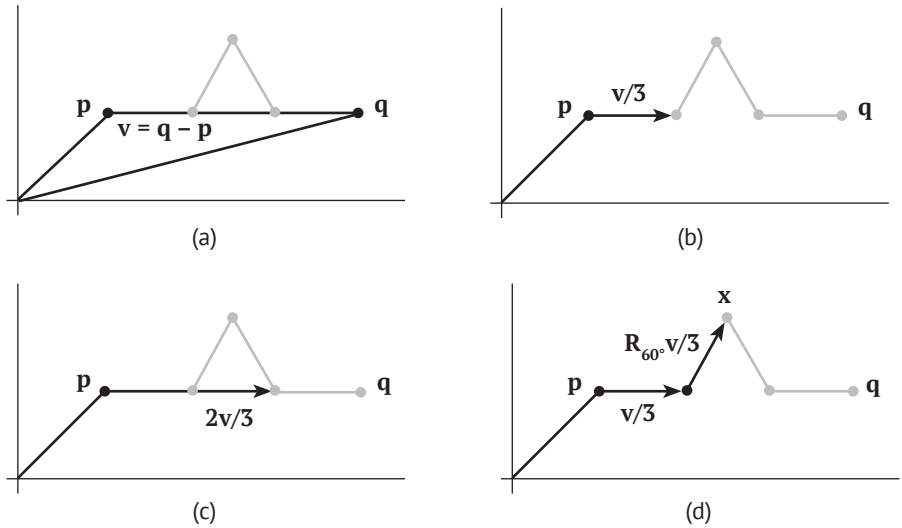


Рис. 7.13. Новые концевые точки на очередном шаге построения кривой Коха

Во-первых, отрезок прямой от точки **p** до точки **q** можно считать вектором на плоскости (a). Кроме того, пусть **v = q – p**, тогда концевые точки на исходном отрезке находятся на расстоянии одной и двух третей его длины от **p**, то есть **p** + **v**/3 и **p** + 2**v**/3 соответственно (b), (c). Последнюю концевую точку **x**, лежащую вне отрезка (**p**, **q**), можно получить как сумму **p** + **v**/3 + $\mathbf{R}_{60^\circ}$**v**/3, где

$$R_{60^\circ} = \begin{pmatrix} \cos 60^\circ & -\sin 60^\circ \\ \sin 60^\circ & \cos 60^\circ \end{pmatrix} = \begin{pmatrix} 1/2 & -\sqrt{3}/2 \\ \sqrt{3}/2 & 1/2 \end{pmatrix}$$

– матрица поворота вектора на 60° против часовой стрелки (d).

Для генерации кривых Коха листинг 7.8 использует пакеты `NumPy` и `Matplotlib`. Начальное условие просто чертит отрезок прямой от **p** до **q**. Рекурсивное условие вызывает метод четырежды с соответствующими концевыми точками, уменьшая количество итераций на 1. Снежинку Коха можно получить, генерируя кривые Коха для трёх начальных отрезков, которые образуют равносторонний треугольник. Наконец, отметим, что код определяет векторы-столбцы согласно большинству источников, где векторы соответствуют векторам-столбцам.

Листинг 7.8. Генерация кривых и снежинки Коха

```
import math
import numpy as np
import matplotlib.pyplot as plt


def koch_curve(p, q, n):
    if n == 0:    # The base case is just a line sigment
        plt.plot([p[0, 0], q[0, 0]], [p[1, 0], q[1, 0]], 'k-'),
    else:

        v = q - p
        koch_curve(p, p + v / 3, n - 1)

        R_60 = np.matrix([[math.cos(math.pi \ 3),
                           -math.sin(math.pi \ 3)],
                          [math.sin(math.pi \ 3),
                           -math.cos(math.pi \ 3)]]),

        x = p + v / 3 + R_60 * v / 3
        koch_curve(p + v / 3, x, n - 1)

        koch_curve(x, p + 2 * v / 3, n - 1)

        koch_curve(p + 2 * v / 3, q, n - 1)


def koch_snowflake(n):
    p = np.array([[0], [0]])
    q = np.array([[1], [0]])
    r = np.array([[0.5], [math.sqrt(3) / 2]])
    koch_curve(p, r, n)
    koch_curve(r, q, n)
    koch_curve(q, p, n)


fig = plt.figure()
fig.patch.set_facecolor('white')
koch_snowflake(3)
plt.axis('equal')
plt.axis('off')
plt.show()
```

7.5.2. Ковёр Серпиньского

Ковёр Серпиньского – второй классический фрактал, который формируется следующим образом. Заданный пустой квадрат с длиной стороны *s* делится на 9 равных квадратов с длиной стороны *s*/3, и центральный из них закрашивается (то есть заполняется некоторым цветом), как показано на рис. 7.14 (шаг 1). Затем то же самое проделывается с восемью меньшими пустыми квадратами, окружающими центральный, что приводит к картинке на рис. 7.14 (шаг 2). Этот процесс можно повторять и дальше вплоть до некоторого *n*-го шага. Добавим только, что квадратные рамки вокруг фрактальных картинок на рис. 7.14 служат лишь для удобства их восприятия, но не являются частью фрактала.

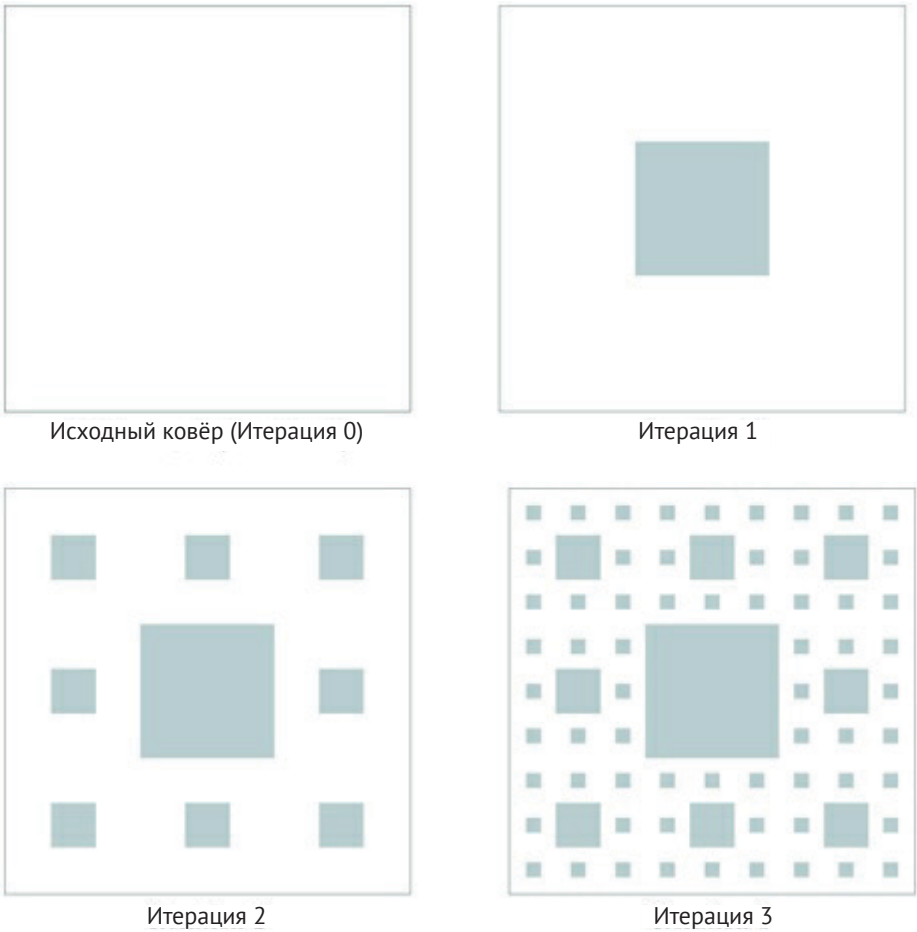


Рис. 7.14. Ковёр Серпиньского после 0-го, 1-го, 2-го и 3-го шагов

Входами задачи могут быть координаты начальной точки p на плоскости – центра пустого исходного квадрата (без краёв), длина его стороны s и количество желаемых шагов n. Ясно, что размер задачи определяется n, а начальное условие возникает при $n = 0$, что не требует никаких действий.

На рис. 7.15 показана декомпозиция задачи для общего рекурсивного условия, когда квадрат делится на девять подквадратов со сторонами $s/3$, а в листинге 7.9 приведена возможная реализация метода. Если $n > 0$, то прежде всего рисуется квадрат с центром в точке p. Его левый нижний угол расположен в точке $p - [s/6, s/6]$, а длина его стороны – $s/3$ (эти значения используются методом `Rectangle` из `Matplotlib`). Затем на шаге $n - 1$ на оставшихся восьми меньших квадратах рекурсивно рисуются восемь ковров Серпиньского размером $s/3$. Координаты центров этих квадратов – $p + s/3$, 0, или $-s/3$ по каждой размерности. Наконец, код в строках 39–41 просто рисует границы пустого исходного квадрата.

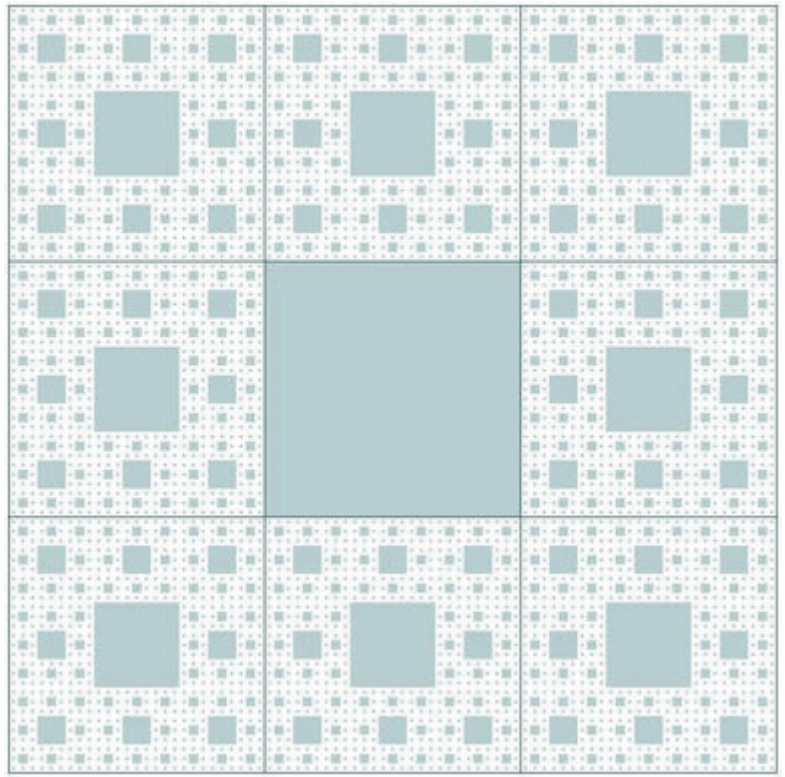


Рис. 7.15. Декомпозиция ковра Серпиньского

Листинг 7.9. Генерация ковра Серпиньского

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle


def sierpinski_carpet(ax, p, n, size):
    if n > 0:
        ax.add_patch(Rectangle((p[0, 0] - size / 6,
                                p[1, 0] - size / 6),
                               size / 3, size / 3,
                               facecolor=(0.5, 0.5, 0.5),
                               linewidth=0))

        q = np.array([[-size / 3], [-size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)
        q = np.array([[-size / 3], [0]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)
        q = np.array([[-size / 3], [size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)

        q = np.array([[0], [-size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)
        q = np.array([[0], [size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)

        q = np.array([[size / 3], [-size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)
        q = np.array([[size / 3], [0]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)
        q = np.array([[size / 3], [size / 3]])
        sierpinski_carpet(ax, p + q, n - 1, size / 3)


fig = plt.figure()
fig.patch.set_facecolor('white')
ax = plt.gca()
p = np.array([[0], [0]])
sierpinski_carpet(ax, p, 4, 1)
ax.add_patch(Rectangle((-1 / 2, -1 / 2), 1, 1,
                       fill=False, edgecolor=(0, 0, 0),
                       linewidth=0.5))
plt.axis('equal')
plt.axis('off')
plt.show()
```

7.6. Упражнения

Упражнение 7.1. Реализуйте функцию из (3.2), вычисляющую биномиальный коэффициент.

Упражнение 7.2. Цель этого упражнения – разработать другое решение задачи о пути по болоту из раздела 7.1. В листингах 7.1 и 7.2 входная матрица уменьшается при каждом новом вызове функции, пока не станет вектором-столбцом. В некоторых языках программирования в каждом рекурсивном вызове проще (а в некоторых случаях просто необходимо) передавать входную матрицу целиком. В таких случаях размер задачи может определяться дополнительными параметрами. Реализуйте вариант такого алгоритма, который выясняет наличие пути по болоту (от правого его края), начиная с участка суши $a_{r,c}$. Метод требует дополнительного параметра c и может в каждом рекурсивном вызове передавать входную матрицу целиком.

Упражнение 7.3. Реализуйте рекурсивный метод, который моделирует печать делений английской дюймовой линейки. Для заданного неотрицательного целого числа n программа должна напечатать на консоли деления от 0 до n дюймов, как показано на рис. 7.16. Кроме того, она должна печатать деления дюйма согласно заданной точности k. В частности, каждый дюйм должен быть разделён на 2^k частей (то есть должны печататься деления в $1/2^k$ дюйма). На рис. 7.16(a) $k = 2$, тогда как на рис. 7.16(b) $k = 3$. Наконец, деления в $1/2^j$ дюйма (для минимально возможного значения j) должны быть представлены $k + 1 - j$ символами дефиса. Например, в (b) деления в 1/4 (3/4, 5/4, 7/4 и т. д.) дюйма состоят из 3 + 1 – 2 = 2 дефисов.

Упражнение 7.4. Цель упражнения – решение иного варианта головоломки «Ханойская башня», которую назовём «узкой» задачей «Ханойская башня». Пусть три стержня упорядочены слева направо. Правила те же, что в исходной задаче, но диски нельзя перемещать между крайними стержнями, как показано на рис. 7.17.

Создайте процедуру, использующую множественную рекурсию для перемещения стопки из n дисков с левого стержня на правый, или наоборот (упражнение включает методы перемещения дисков между средним стержнем и левым или правым стержнями). Кроме того, определите количество перемещений отдельных дисков стопки из n дисков.

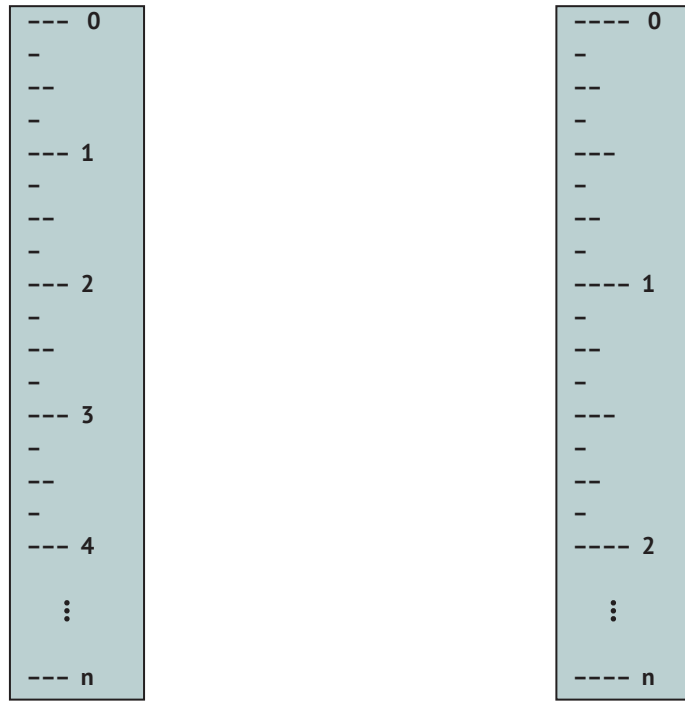


Рис. 7.16. Деления английской дюймовой линейки

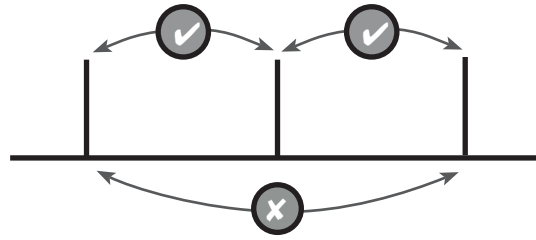


Рис. 7.17. Дополнительное условие «узкой» задачи «Ханойская башня»

Упражнение 7.5. Реализуйте функцию, вычисляющую количество узлов в двоичном дереве поиска. Считайте, что дерево представлено в виде списка четырехкомпонентных элементов, как описано в разделе 5.3.

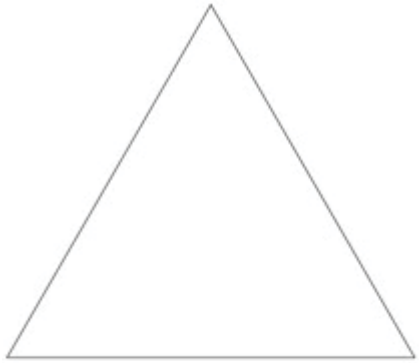
Упражнение 7.6. Реализуйте функцию, которая для заданного входного списка **a** из *n* элементов определяет самый длинный непрерывный

его подсписок одинаковых элементов. Например, для a = [1, 3, 5, 5, 4, 4, 4, 5, 5, 6] результатом будет подсписок [4, 4, 4]. Разбейте задачу так же, как в листинге 7.6, и примените вспомогательную рекурсивную функцию, определяющую одинаковость всех элементов подсписка. Кроме того, реализуйте рекурсивную функцию, которая не вызывает вспомогательный метод, как в листинге 7.7.

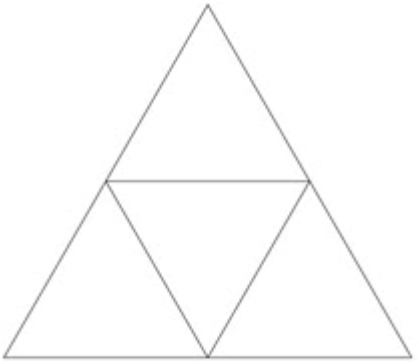
Упражнение 7.7. Разработайте и реализуйте рекурсивный метод, вычисляющий самую длинную подстроку-палиндром в заданном списке из n элементов. Это не та задача, что описана в разделе 7.4, так как подпоследовательность списка не обязательно состоит из смежных элементов. Например, самая длинная подпоследовательность-палиндром списка [1, 3, 4, 4, 6, 3, 1, 5, 1, 3] – это [1, 3, 4, 4, 3, 1], тогда как самый длинный (непрерывный) палиндром – это подсписок [3, 1, 5, 1, 3].

Упражнение 7.8. Реализуйте рекурсивный метод, рисующий треугольник Серпиньского (см. рис. 7.18). Треугольник Серпиньского может быть сгенерирован, начиная с равностороннего треугольника вида △, который делится на четыре меньших равносторонних треугольника со стороной, вдвое меньшей, чем у исходного. Один из них (центральный) будет перевёрнутым ▽, а три других – того же вида △, что исходный. Для этих трёх треугольников процесс повторяется итерационно (см. рис. 7.18). Для решения своей задачи метод должен получать координаты точки на плоскости (например, нижней левой вершины треугольника), длину его стороны и количество итераций n. Используйте пакеты `Numpy` и `Matplotlib`.

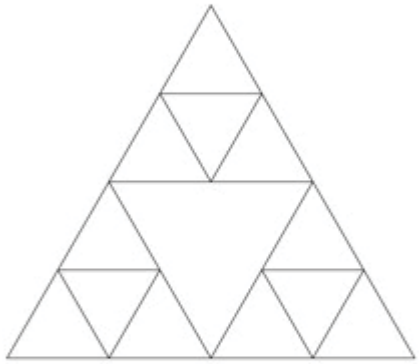
Упражнение 7.9. Кривая Гильберта – это непрерывная фрактальная кривая, заполняющая некоторое пространство. На рис. 7.19 показаны первые шесть кривых Гильберта. Длина кривой растёт экспоненциально с ростом порядка (размерности) задачи, постепенно заполняя всю площадь ограниченного пространства (квадрата). Рекурсивная декомпозиция задачи заключается в разбиении фрактала на четыре кривых меньшего размера с соответствующей ориентацией. Кроме того, так как вся кривая непрерывна, четыре меньшие кривые должны соединяться, и именно это соединение малых кривых в одну фактически завершает процесс формирования всей кривой. Цель этого сложного упражнения – реализовать рекурсивный метод, рисующий кривую Гильберта n-го порядка.



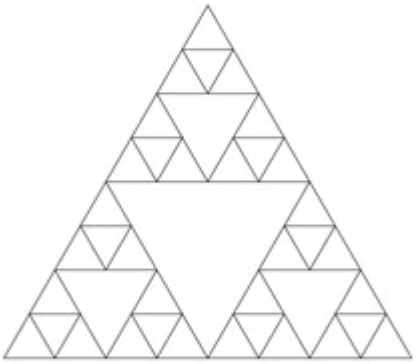
Исходный треугольник



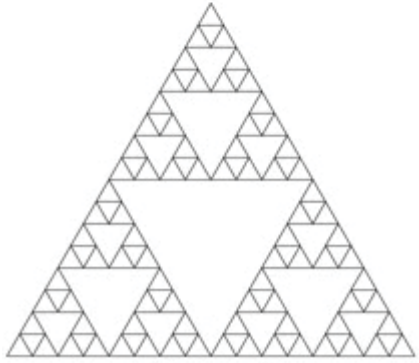
Итерация 1



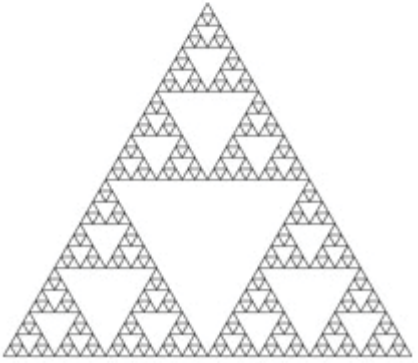
Итерация 2



Итерация 3

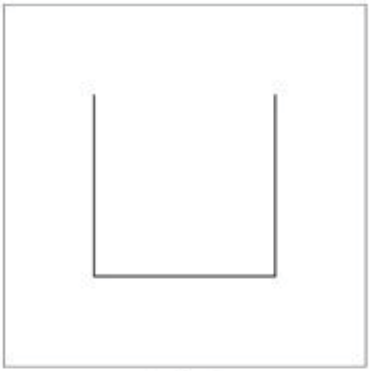


Итерация 4

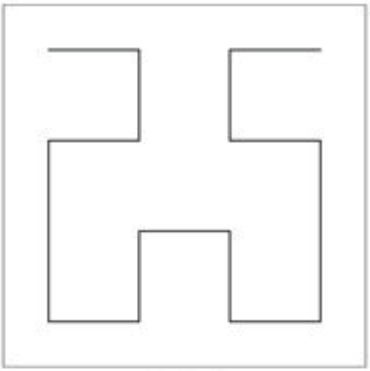


Итерация 5

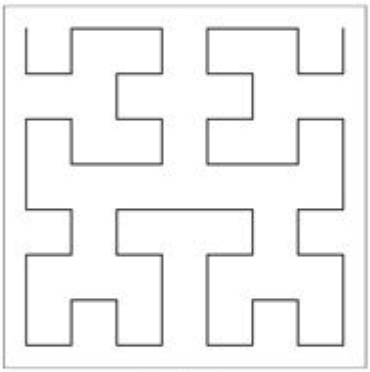
Рис. 7.18. Треугольный ковёр Серпиньского



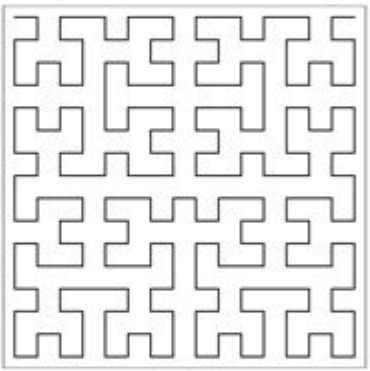
Порядок 1



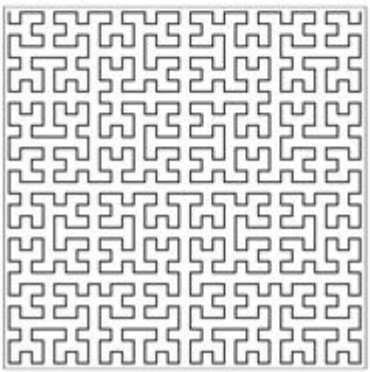
Порядок 2



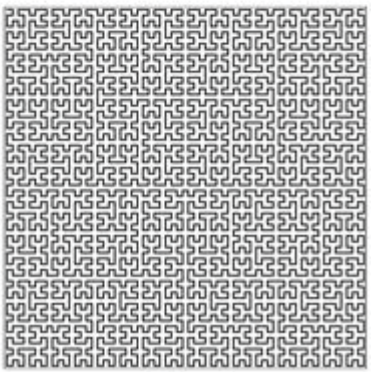
Порядок 3



Порядок 4



Порядок 5



Порядок 6

Рис. 7.19. Первые шесть шагов построения кривой Гильберта

Глава 8

Задачи подсчёта

Музыка – это удовольствие от подсчётов, которое испытывает человеческая душа, не осознающая, что она считает.

– Готфрид Лейбниц

Рекурсия широко используется в комбинаторике – разделе математики, изучающем подсчёт на дискретных множествах объектов и играющем важнейшую роль в анализе алгоритмов. Эта глава посвящена рекурсивным решениям вычислительных задач подсчёта, цель которых – сложение определенного числа элементов, объектов, вариантов, понятий и т. д. Общая стратегия заключается в разбиении подлежащих подсчёту элементов на несколько непересекающихся подмножеств с последующим сложением количеств в каждом из них. С точки зрения рекурсии исходная задача должна разбиваться на несколько меньших подзадач, а её результатом должна быть сумма результатов этих меньших подзадач.

В предыдущих главах мы уже встречались с несколькими из таких задач. Например, вычисление суммы первых положительных целых чисел можно понимать как подсчёт числа квадратных блоков треугольных структур, как показано на рис. 1.5(а). Другой пример – задача подсчёта битов из упражнения. В этой главе мы изучим две альтернативные рекурсивные стратегии подсчёта битов, опирающиеся на две разные декомпозиции.

Мы увидим, что несколько из широко известных рекурсивных функций связано с фундаментальными комбинаторными понятиями. Например, факториал, мощность и биномиальные коэффициенты связаны с перестановками, размещениями (с повторами) и сочетаниями соот-

ветственно. Кроме того, во многих комбинаторных задачах возникают ещё и числа Фибоначчи. В этой главе мы будем выводить эти функции, расчленяя соответствующие задачи подсчёта, а не их математические формулы и определения. Хотя все эти задачи – чисто математические, мы будем решать их, используя те же понятия, приёмы и методики, что были изложены в предыдущих главах.

Наконец, поскольку решения задач подсчёта либо просты для программирования, либо уже появлялись в книге, глава не содержит их кода (некоторые упражнения предлагают реализовать лишь отдельные функции). В этом смысле её основная цель – побудить к рекурсивному мышлению при решении разного рода вычислительных задач.

8.1. Перестановки

Под перестановкой множества элементов понимается определённая последовательность этих элементов (или действие по изменению порядка следования элементов в заданной последовательности). В этом разделе мы разберём перестановки «без повторений», когда все их элементы различны. В частности, рассмотрим задачу подсчета общего количества возможных перестановок из n различных элементов. Для простоты предположим, что эти n элементов – целые числа 1, 2, ..., n, а перестановка – упорядоченный список этих элементов. Для $n = 4$ существует 24 различные перестановки, представленные списками на рис. 8.1 (в разделе 12.2.2 описывается алгоритм генерации всех таких перестановок).

```
[1,2,3,4]   [2,1,3,4]   [3,1,2,4]   [4,1,2,3]
[1,2,4,3]   [2,1,4,3]   [3,1,4,2]   [4,1,3,2]
[1,3,2,4]   [2,3,1,4]   [3,2,1,4]   [4,2,1,3]
[1,3,4,2]   [2,3,4,1]   [3,2,4,1]   [4,2,3,1]
[1,4,2,3]   [2,4,1,3]   [3,4,1,2]   [4,3,1,2]
[1,4,3,2]   [2,4,3,1]   [3,4,2,1]   [4,3,2,1]
```

Рис. 8.1. Возможные перестановки (списки) первых четырех положительных целых чисел

Для определения математической рекурсивной функции, обеспечивающей результат для заданного n, можно, как обычно, следовать схеме на рис. 2.1. Для задач подсчёта их размер связан с общим числом складываемых элементов. Размер данной задачи – это просто n, поскольку число перестановок растет, как функция от n. Тривиальное начальное условие выполняется при $n = 1$, когда получается всего одна перестановка единственного элемента. Кроме этого, можно рассмотреть случай, когда $n = 0$. Хотя может показаться, что результатом будет 0 перестановок, математики ради удобства считают, что число перестановок в этом случае равно 1. Например, если перестановке соответствует список, то можно считать, что при $n = 0$ получается один список – пустой.

Для вывода рекурсивного условия можно разбить результат на n подзадач размера $n - 1$, как показано на рис. 8.2, где каждая из подзадач подсчитывает возможные перестановки в разных подмножествах.

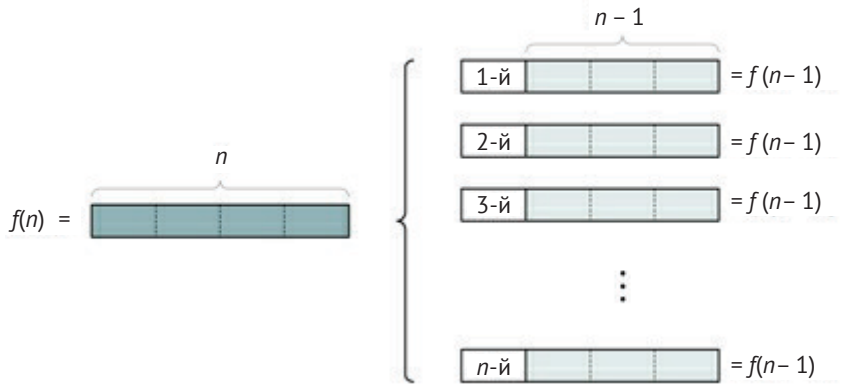


Рис. 8.2. Декомпозиция задачи подсчёта возможных перестановок $f(n)$ для n различных элементов

Пусть $f(n)$ – функция, которая вычисляет количество возможных перестановок n различных элементов. Если мы зафиксировали элемент в первой позиции перестановки, то нам нужно посчитать количество возможных перестановок для оставшихся $n - 1$ элементов. По нашему определению, это количество равно $f(n - 1)$, которое, по принципу индукции, мы считаем известным. Наконец, поскольку в первой позиции перестановки может оказаться каждый из n элементов, $f(n)$ определяется как $f(n - 1)$, сложенное с собой n раз, что, очевидно, можно записать как $n \cdot f(n - 1)$. Таким образом, вместе с начальными условиями определение функции будет

$$f(n) = \begin{cases} 1, & \text{если } n = 0, \\ n \cdot f(n-1), & \text{если } n > 0, \end{cases}$$

и является функцией-факториалом (начальное условие для $n = 1$ избыточно). На рис. 8.3 приведено обоснование декомпозиции задачи для $n = 4$, где вычисляемые перестановки разделены на 4 подмножества (соответствующих подзадачам размера $n - 1$), каждое из которых содержит перестановки для фиксированного первого элемента.

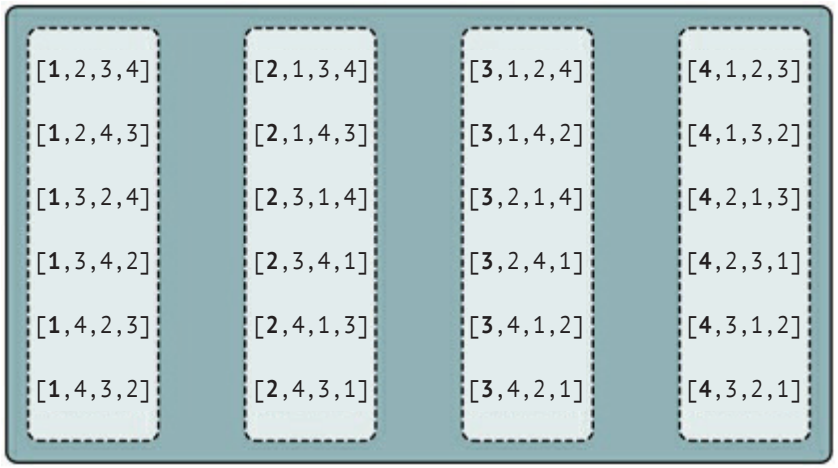


Рис. 8.3. Декомпозиция возможных перестановок первых четырёх положительных целых чисел

Наконец, нетрудно видеть, что нужный результат даёт функция вычисления факториала, поскольку существует ровно n способов выбрать первый элемент перестановки, $n - 1$ способов выбрать второй (зафиксировав первый), $n - 2$ способов выбрать третий и т. д. Это, естественно, приводит к $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$ различным перестановкам. Однако такое рассуждение ближе к итерационному, тогда как представленное выше рекурсивное решение опирается на индукцию и декомпозицию задачи.

8.2. Размещения с повторениями

Рассмотрим некоторое множество из n элементов. В комбинаторике размещением из n элементов множества по k элементов называется последовательность из $k \leqslant n$ элементов, в которой имеет значение поря-

док следования элементов. Если для записи размещений использовать списки, то $[a_1, a_5, a_8]$ и $[a_8, a_5, a_1]$ – это два разных размещения по 3 элемента на множестве $\{a_1, a_2, \ldots, a_{10}\}$. Кроме того, если элементу разрешается появляться в последовательности не один раз, то такое размещение называется размещением «с повторениями» (например, $[a_1, a_5, a_1]$ было бы правильным размещением). В этом разделе мы изучим именно этот вид размещения с повторениями. На рис. 8.4 приведён конкретный пример, где квадратиками обозначены элементы множества, кружочками – положения элементов в размещении, а стрелками – ссылки на конкретные элементы множества, вошедшие в размещение (в упражнении 8.1 используется другое представление).

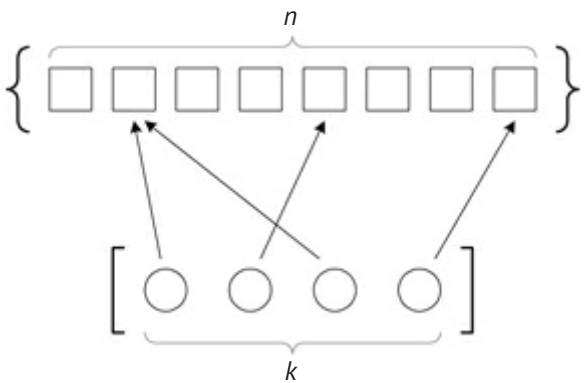
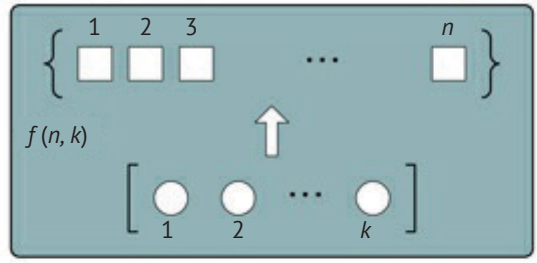


Рис. 8.4. Пример размещения с повторениями по k элементам из n элементов множества

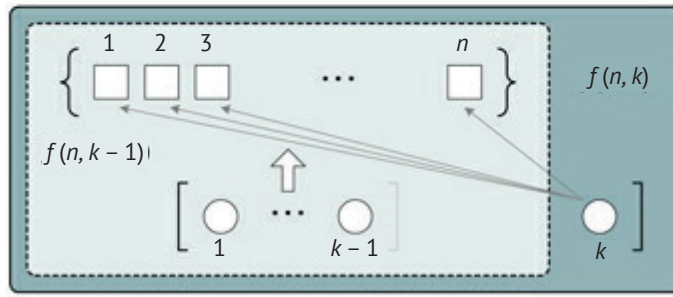
Нетрудно видеть, что согласно «принципу умножения» число размещений с повторениями по k элементов на множестве из n элементов есть n^k, так как на каждом месте в размещении из k элементов может находиться (в том числе повторно) каждый из n элементов множества. Тем не менее мы выведем этот факт, используя рекурсию.

Задача определяется параметрами k и n. Хотя задачу можно свести к меньшим подзадачам путём уменьшения k и/или n, декомпозиция будет проще, если считать размером задачи k. Тривиальное начальное условие выполняется при $k = 1$, когда существует ровно n различных способов выбрать один элемент из множества. Кроме того, как в задаче о перестановках, можно рассмотреть случай, когда $k = 0$, который можно понимать как пустое размещение.

Для вывода рекурсивного условия можно воспользоваться схемой на рис. 8.5.



(a)



(b)

Рис. 8.5. Декомпозиция задачи подсчёта размещений с повторениями *n* элементов множества в *k* позициях

Исходная задача, обозначенная *f*(*n, k*), показана на рис. 8.5(a), где толстая стрелка заключает в себе все возможные отношения между множеством и размещением. На рис. 8.5(b) приведена подзадача с уменьшенным на 1 исходным размером, которая подсчитывает все способы размещения *n* элементов в *k* – 1 позициях. Наконец, поскольку существует *n* способов разместить элементы множества в последней *k*-й позиции, общее количество размещений будет *f*(*n*, *k* – 1), сложенное *n* раз. Таким образом, рекурсивная функция определяется как

$$f(n,k)=\begin{cases}1, & \text{если } k=0,\\ f(n,k-1)\cdot n, & \text{если } k>0,\end{cases}$$

и является степенью n^k (см. листинг 4.1).

8.3. Сочетания

Сочетание из *n* различных элементов по *k* без повторений – это просто подмножество из *k* элементов исходного множества. Сочетания подоб-

ны размещениям, только порядок следования элементов здесь не имеет значения. Поэтому сочетание соответствует (под)множеству, а не последовательности. Например, подмножество $\{a_1, a_2, a_3\}$ – это сочетание элементов на множестве $\{a_1, a_2, ..., a_{10}\}$, причём $\{a_1, a_2, a_3\}$ и $\{a_3, a_1, a_2\}$ не различаются. В этом разделе мы займёмся подсчётом всех сочетаний из n различных элементов по k элементов без повторений.

Размер этой задачи зависит и от k, и от n (где $k \leqslant n$). Как мы вскоре увидим, для разработки рекурсивного решения их необходимо уменьшать вместе. Но мы начнём с определения начальных условий. Тривиальное выполняется при $k = n$, когда существует лишь одно правильное сочетание – это всё множество из n элементов. Другое простое условие выполняется при $k = 1$, когда результат, очевидно, равен n. Кроме того, при $k = 0$ можно считать, что есть только одна правильная комбинация – пустое множество. Как и в предыдущих разделах, если у нас есть начальное условие для $k = 0$, то необходимость в начальном условии для $k = 1$ отпадает.

Для вывода рекурсивного условия разобьём все сочетания на два множества. Рассмотрим исходное множество из n элементов $\{a_1, a_2, ..., a_n\}$ и отдельный его элемент, скажем, a_1. Обратите внимание, что существует два типа сочетаний: (a) содержащие a_1 и (b) не содержащие a_1. Таким образом, исходную задачу $f(n, k)$ можно разбить на две в зависимости от типа сочетания, как показано на рис. 8.6.

Если сочетания содержат a_1, то оставшиеся $k - 1$ элементов – это сочетания из $n - 1$ элементов подмножества $\{a_2, ..., a_n\}$ по $k - 1$ элементов. А этот набор сочетаний определяется функцией $f(n - 1, k - 1)$. Если же a_1 не содержится в сочетаниях, то это – сочетания из того же подмножества $\{a_2, ..., a_n\}$ по k элементов, а количество таких сочетаний – $f(n - 1, k)$. Таким образом, результатом будет сумма обоих типов сочетаний, а рекурсивная формула – $f(n, k) = f(n - 1, k - 1) + f(n - 1, k)$. В итоге вместе с начальными условиями функция определяется как

$$f(n,k) = \begin{cases} 1, & \text{если } k = 0 \text{ или } k = n, \\ f(n-1, k-1) + f(n-1, k) & \text{в противном случае} \end{cases}$$

и определяет биномиальный коэффициент (см. (3.2)).

8.4. Подъём по лестнице

Цель следующей задачи – посчитать количество возможных способов подняться по лестнице из n ступенек с шагом в одну или две ступеньки. На рис. 8.7 показан один из способов подъёма по лестнице из семи ступенек.

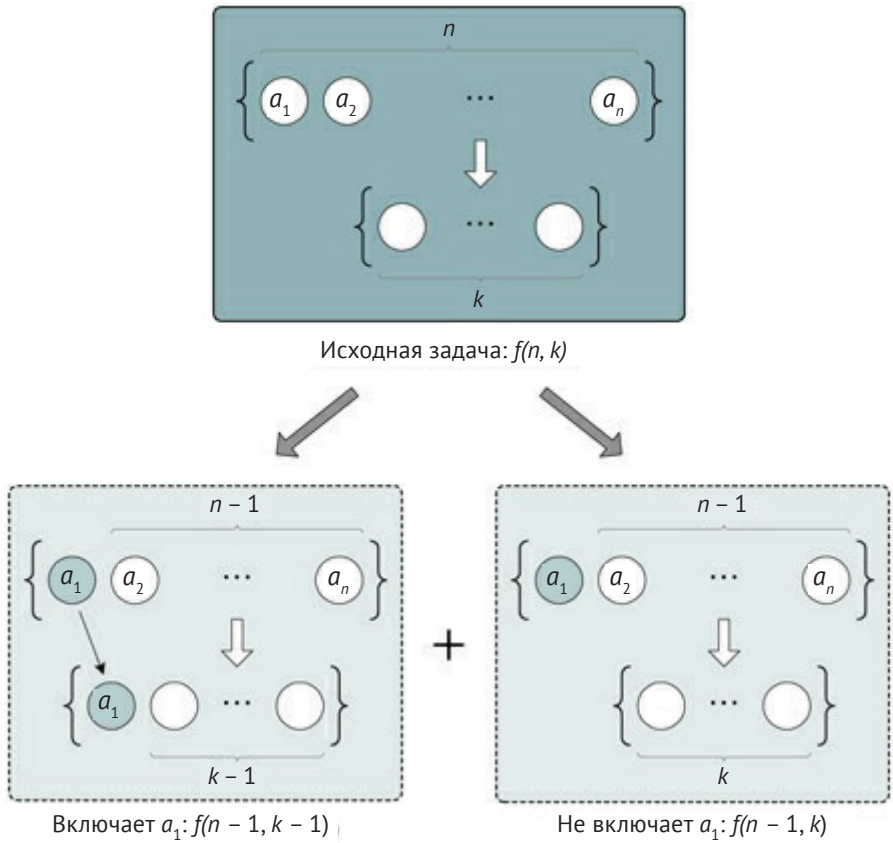


Рис. 8.6. Декомпозиция задачи подсчёта сочетаний из *n* элементов по *k*

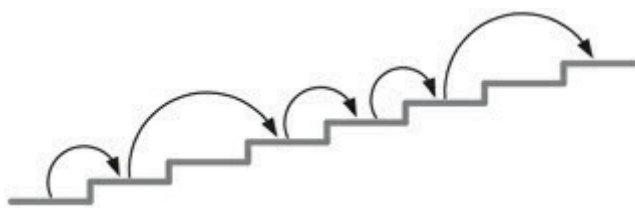


Рис. 8.7. Один из способов подняться по лестнице с шагом в 1 или 2 ступеньки

Понятно, что размер задачи – *n*. Рассмотрим несколько начальных условий. Если $n = 1$, то существует только один способ подняться по лестнице. Если $n = 2$, то можно сделать либо два шага, либо один (через ступеньку). Кроме того, если представлять способ подъёма списком, то можно также считать, что при $n = 0$ результатом будет пустой список.

Для вывода рекурсивного условия задачу можно разбить так, как показано на рис. 8.8.

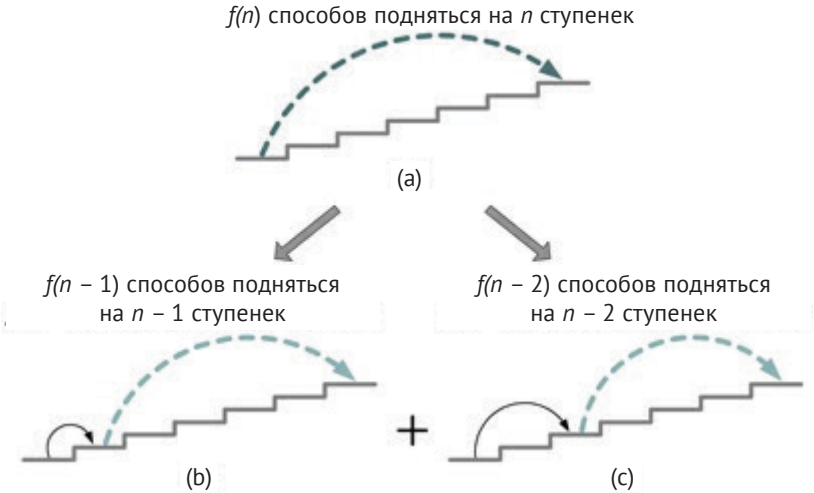


Рис. 8.8. Декомпозиция задачи подсчёта всех способов подняться по лестнице с шагом в 1 или 2 ступеньки

В исходной задаче (рис. 8.8(a)) темная стрелка представляет общее количество способов $f(n)$ подъёма по лестнице из n ступенек. А сами способы подъёма можно разделить на две группы. Одна из них включает последовательность дальнейших шагов после одиночного шага. После одиночного шага будет ещё $f(n-1)$ способов добраться до вершины лестницы, делая одиночные или двойные шаги (рис. 8.8(b)). Поэтому жирная стрелка представляет подзадачу размера $n-1$. Во второй группе – дальнейшая последовательность шагов после двойного шага. В этом случае (рис. 8.8(c)) количество способов подняться по лестнице будет $f(n-2)$, где жирная стрелка указывает на подзадачу размера $n-2$. Наконец, поскольку обе группы последовательностей представляют общее количество способов подняться по лестнице, решение – $f(n-1) + f(n-2)$. Вместе с начальными условиями функция может быть определена следующим образом:

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ 2, & \text{если } n = 2, \\ f(n-1) + f(n-2), & \text{если } n \geq 3, \end{cases}$$

что очень напоминает функцию Фибоначчи $F(n)$ (см. (1.2)), а точнее $f(n) = F(n+1)$.

8.5. Путь по Манхэттену

Во многих городах улицы и авеню перпендикулярны друг другу. Предположим, что они образуют прямоугольную систему координат, как на рис. 8.9.

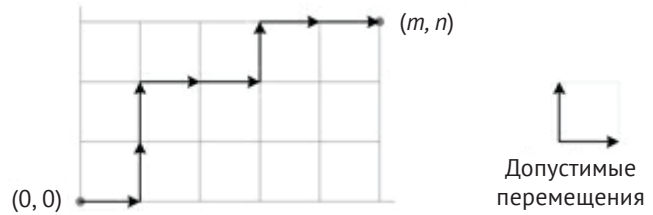


Рис. 8.9. Задача о пути по Манхэттену

Цель задачи – определить количество путей, ведущих к пересечению авеню $m \geqslant 0$ с улицей $n \geqslant 0$ (где m и n – целые числа), начиная с авеню 0 и улицы 0. Чтобы это сделать, предположим, что на каждом шаге разрешается идти только к улице или авеню с бо́льшим номером, то есть вверх или вправо.

Можно считать, что размер задачи – $\min(m, n)$. В этом случае простейшее начальное условие выполняется, когда один из параметров равен 0, поскольку в этом случае к точке (m, n) ведёт только один правильный путь по прямой. Кроме того, если оба параметра равны 0, такой путь можно считать «пустым».

Задачу можно разбить на две отдельные подзадачи, как показано на рис. 8.10.

Пусть $f(m, n)$ – общее количество путей от (0, 0) к (m, n). Обратите внимание, что можно разделить пути на два несвязных множества в зависимости от выбора первого шага. Если мы пошли вправо к точке (1, 0), то от неё до (m, n) будет $f(m - 1, n)$ путей. И наоборот, если мы пошли вверх к точке (0, 1), то от неё до (m, n) будет $f(m, n - 1)$ путей. Сумма этих количеств даст рекурсивное решение. Функция выглядит следующим образом:

$$f(m,n) = \begin{cases} 1, & \text{если } m = 0 \text{ или } n = 0, \\ f(m-1, n) + f(m, n-1) & \text{в противном случае.} \end{cases}$$

Наконец, путь к (m, n) – это последовательность из $m + n$ шагов, где m – количество шагов вправо, а n – количество шагов вверх. Иными словами, весь путь можно представить двоичной последовательностью длины $m + n$. Следовательно, его можно задать последовательностью

направлений движения – вправо (это сочетание из $n + m$ элементов по m) или, аналогично, прямо (это сочетание из $n + m$ элементов по n). Таким образом, функция – это биномиальный коэффициент:

$$f(m,n) = C^n_{m+n} = C^m_{m+n} = (m+n)!/(m! \cdot n!).$$

В итоге можно считать, что формула определяет количество сочетаний из $m + n$ элементов по m или по n элементов, где количество шагов вправо m, как и количество шагов вверх n, неизменно.

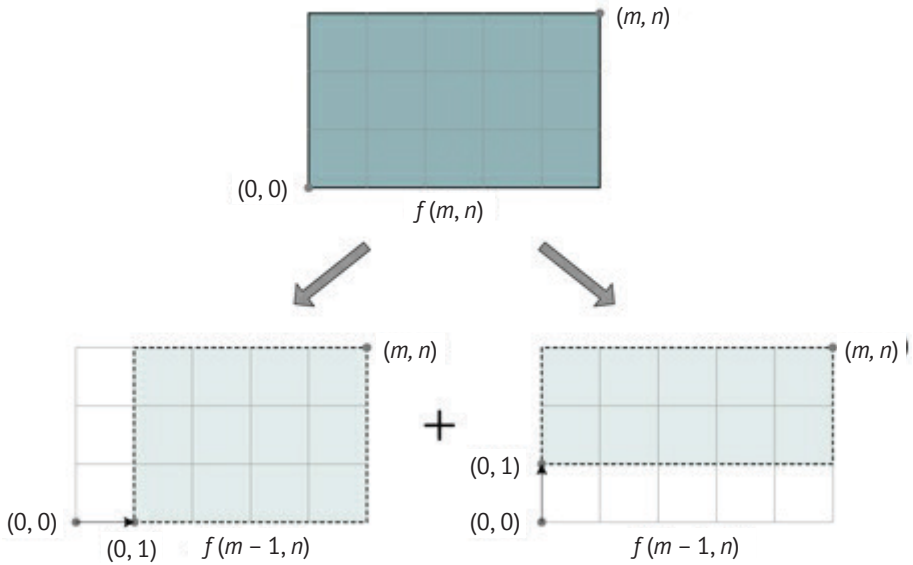


Рис. 8.10. Декомпозиция задачи о пути по Манхэттену

8.6. Триангуляция выпуклого многоугольника

Триангуляция многоугольника – это множество непересекающихся треугольников, которые покрывают (без перекрытий) всю область многоугольника и вершины которых являются вершинами многоугольника. На рис. 8.11 показаны две триангуляции выпуклого 7-угольника (выпуклый многоугольник – это многоугольник, в котором любой отрезок, соединяющий любые две его внутренние точки, лежит внутри многоугольника). Следующая задача состоит в определении функции, вычисляющей все возможные различные способы разбиения выпуклого n-угольника на треугольники.

Рис. 8.11. Две возможные триангуляции одного выпуклого 7-угольника

Размер задачи – n. Тривиальное начальное условие выполняется при $n = 3$, когда результат, очевидно, равен 1. Кроме того, подобно предыдущим задачам, если $n = 2$, то результат тоже равен 1 (что будет уточнено ниже).

Рекурсивное условие здесь сложнее, чем в предыдущих примерах, так как решение потребует рассмотрения нескольких подзадач меньшего размера. Для понимания декомпозиции воспользуемся конкретным примером. Рассмотрим, например, нижнюю сторону 8-угольника на рис. 8.12.

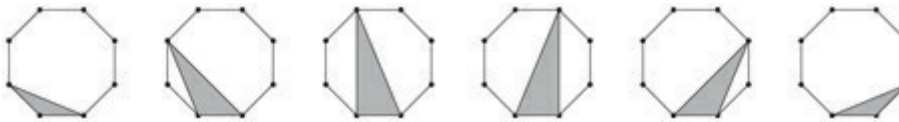


Рис. 8.12. Шесть (n – 2) треугольников на базе нижней стороны восьмиугольника (n = 8)

Естественно, что она должна принадлежать одному из треугольников триангуляции, но таких различных треугольников, включающих эту сторону, – 6 (n – 2) по числу вершин многоугольника, не принадлежащих его нижней стороне. Поэтому триангуляции могут быть разбиты на 6 независимых множеств, отвечающих конкретному треугольнику на базе нижней стороны. Другими словами, затененный треугольник определяет соответствующее ему множество триангуляций, а общее количество триангуляций будет суммой триангуляций в каждом из шести множеств.

Выбрав один из затенённых треугольников, по обе его стороны мы получим два выпуклых многоугольника (один из которых может быть пустым), которые также должны быть разбиты на треугольники. Отметим, что оба многоугольника примыкают к сторонам затенённого треугольника. Согласно принципу умножения, общее количество триангуляций, связанных с выбранным затенённым треугольником, будет равно произведению числа триангуляций каждого из смежных многоугольников, как показано на рис. 8.13.

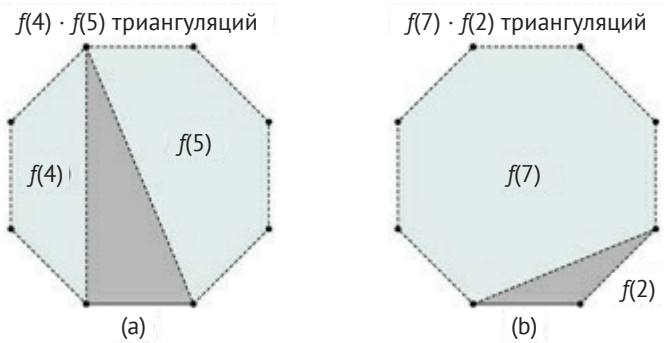


Рис. 8.13. Декомпозиция задачи триангуляции выпуклого многоугольника для конкретного треугольника

Пусть $f(n)$ – целевая функция задачи, тогда на рис. 8.13(a) слева от затенённого треугольника – подзадача размера 4, а справа – подзадача размера 5. Количество триангуляций, связанных с затенённым треугольником, будет $f(4) \cdot f(5)$. В случае (b) одна из подзадач имеет размер 7, а вторая – размер 2, поэтому мы имеем $f(7) \cdot f(2)$ триангуляций, связанных с затенённым треугольником. Несмотря на то что один из многоугольников (связанный с $f(2)$) пуст, количество возможных триангуляций, связанных с затенённым треугольником, даст $f(7)$, что подтверждает использование в качестве начального условия $f(2) = 1$.

В конце концов, нужно сложить все триангуляции, связанные с шестью затененными треугольниками. В приведённом примере, как показано на рис. 8.14, мы получим:

$$f(8) = f(2) \cdot f(7) + f(3) \cdot f(6) + f(4) \cdot f(5) + f(5) \cdot f(4) + f(6) \cdot f(3) + f(7) \cdot f(2).$$

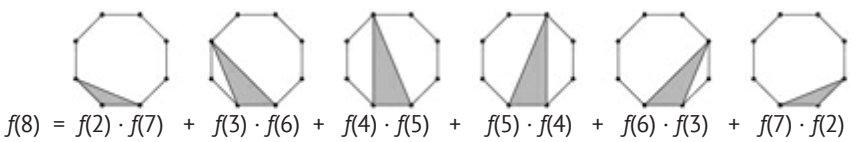


Рис. 8.14. Общее количество триангуляций в восьмиугольнике

В общем виде рекурсивную функцию можно определить как

$$f(n) = \begin{cases} 1, & \text{если} \quad n = 2, \\ \sum_{i=2}^{n-1} f(i) \cdot f(n+1-i), & \text{если} \quad n > 2. \end{cases}$$

Отметим, что дополнительное начальное условие для $n = 3$ не обязательно.

Наконец, $f(n)$ имеет отношение к часто встречающимся в комбинаторике числам Каталана и может быть определена как

$$C(n) = \begin{cases} 1, & \text{если } n = 0, \\ \sum_{i=0}^{n-1} C(i) \cdot C(n-1-i), & \text{если } n > 0. \end{cases} \quad (8.1)$$

В частности, $f(n) = C(n - 2)$.

8.7. Пирамиды из кругов

В следующей задаче нужно определить количество возможных способов укладки кругов рядами внутри пирамиды. Начав с непрерывного ряда кругов в основании, нужно укладывать на него круги согласно следующим правилам:

1) все круги в ряду должны примыкать друг к другу, то есть между ними не должно быть промежутков;
2) каждый круг не из нижнего ряда должен лежать на двух кругах рядом ниже, образуя равносторонний треугольник («пирамиду»).

На рис. 8.15 приведены примеры правильных и неправильных укладок кругов. Задача состоит в том, чтобы определить функцию, которая подсчитывает количество правильных укладок кругов, начиная с n кругов нижнего ряда.

Неправильные

Правильные

Рис. 8.15. Правильные и неправильные пирамиды из кругов

Размер задачи – n. Тривиальное начальное условие выполняется при $n = 1$, когда результат, очевидно, равен 1. Для вывода рекурсивного условия будем делить возможные пирамидальные конфигурации на не-

сколько несвязных групп по количеству кругов непосредственно над нижним рядом. Рисунок 8.16 иллюстрирует эту идею для $n = 4$.

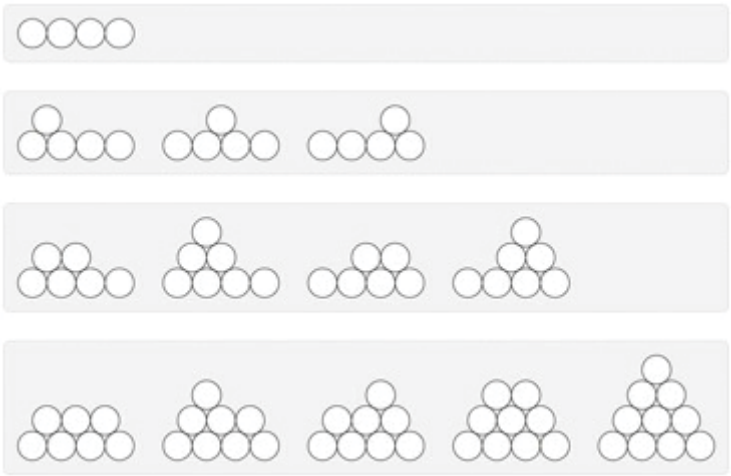


Рис. 8.16. Пирамиды, сгруппированные по количеству кругов непосредственно над нижним рядом из $n = 4$ кругов

Ключ к решению – в том, что меньшие пирамиды (подзадачи) с одинаковым количеством кругов в нижнем ряду можно разместить несколькими способами над нижним рядом кругов исходной задачи (см. рис. 8.17). В частности, для начального входного параметра n существует $n - i$ способов размещения подобной пирамиды размера i. На рисунке $i = 3$ и $n = 7$, тогда существует $n - i = 4$ способа разместить подобные исходной пирамиды размера 3 над нижним рядом из $n = 7$ кругов.

Рис. 8.17. Декомпозиция задачи о пирамиде из кругов на подзадачи конечного размера

В итоге, поскольку размер меньших подзадач варьируется от 1 до $n - 1$, функция должна сложить все возможные конфигурации кругов. Кроме того, поскольку сам полный нижний ряд кругов (без кругов над ним) считается правильной укладкой, к функции следует добавить ещё 1. Ниже приведена функция, решающая задачу:

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ 1 + \sum_{i=1}^{n-1} (n-i) \cdot f(i), & \text{если } n > 1. \end{cases} \tag{8.2}$$

Таким образом, результат представляет собой функцию Фибоначчи. В частности, можно доказать, что $f(n) = F(2n - 1)$, где F – функция Фибоначчи.

8.8. Упражнения

Упражнение 8.1. Пусть $f(n, k)$ – количество размещений с повторениями из n элементов по k элементов. Существует n способов выбрать первый элемент размещения, n способов выбрать второй и т. д. Все эти варианты можно представить полным деревом-списком высоты k, внутренние узлы которого всегда имеют n дочерних узлов, а листья представляют собой размещения. На рис. 8.18 приведён пример всех размещений с повторениями из 4 элементов множества $\{a, b, c, d\}$ по 2 элемента.

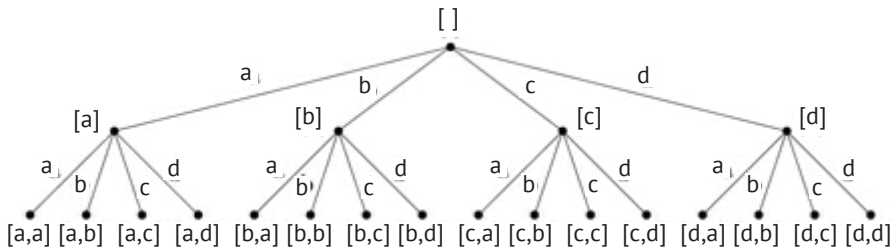


Рис. 8.18. Размещения с повторениями из четырех элементов множества $\{a, b, c, d\}$ по два элемента

Узловые элементы списка – это частичные подпоследовательности в процессе построения размещения, тогда как помеченные ветви – это конкретные элементы, добавляемые к частичной подпоследовательности. Таким образом, каждое размещение в дереве – это путь от корневого узла до листа.

Для данного древовидного представления размещений нарисуйте схему вывода рекурсивного правила $f(n, k) = f(n, k - 1) \cdot n$. Исходная задача и её подзадачи должны представляться деревьями. Для простоты можно опустить метки узлов и ветвей.

Упражнение 8.2. У нас есть n одинаковых костей для игры в «домино» размера 1×2 (или 2×1 при повороте на 90°). Выведите рекурсивную функцию, определяющую количество способов выложить из костей (без наложений и дыр) прямоугольник размером $2 \times n$. На рис. 8.19 приведён конкретный пример для $n = 10$.

Рис. 8.19. Выкладывание прямоугольника размером 2×10 из костей домино размером 1×2 или 2×1

Упражнение 8.3. Баскетбольная команда может набирать очки тремя различными способами: штрафной бросок – одно очко, бросок с игры (в пределах трёхочковой зоны) – два очка и бросок из-за трёхочковой зоны – три очка. Найдите функцию, определяющую количество вариантов набора баскетбольной командой n очков. Например, существует 4 варианта набрать 3 очка: 1 + 1 + 1, 1 + 2, 2 + 1 и 3. Как видим, последовательность получения очков имеет значение.

Упражнение 8.4. Решите задачу из упражнения 3.8, используя вместо формулы сложения битов рекурсию. Нарисуйте исходную задачу, как на рис. 3.11, и определите в ней подзадачи.

Упражнение 8.5. Реализуйте функцию вычисления чисел Каталана $C(n)$ в (8.1).

Упражнение 8.6. В двоичном дереве каждый узел может иметь ни одного, один или два дочерних узла. Определите рекурсивную функцию $f(n)$, вычисляющую количество различных двоичных деревьев, состоящих из n узлов. Учтите разницу между левыми и правыми дочерними узлами. Например, на рис. 8.20 показано 5 возможных двоичных деревьев, состоящих из $n = 3$ узлов.

Рис. 8.20. Пять различных двоичных деревьев из трёх узлов

Упражнение 8.7. Рассмотрим задачу о пирамиде из кругов из раздела 8.7. В этом упражнении для решения задачи предлагается воспользоваться другой декомпозицией. А именно разбивать различные пирамиды по их высоте, как показано на рис. 8.21, где $g(h, n)$ – количество пирамид высоты h, которые можно построить на базе n кругов нижнего ряда. Таким образом, цель этого упражнения – рекурсивное определение функции $g(h, n)$. В заключение убедитесь в правильности функции, закодировав её и вычислив общее количество пирамид, построенных на базе нижнего ряда из n кругов:

$$\sum_{i=1}^{n} g(h,n) = f(n) = F(2n-1), \tag{8.3}$$

где f – функция (8.2), а F – функция Фибоначчи.

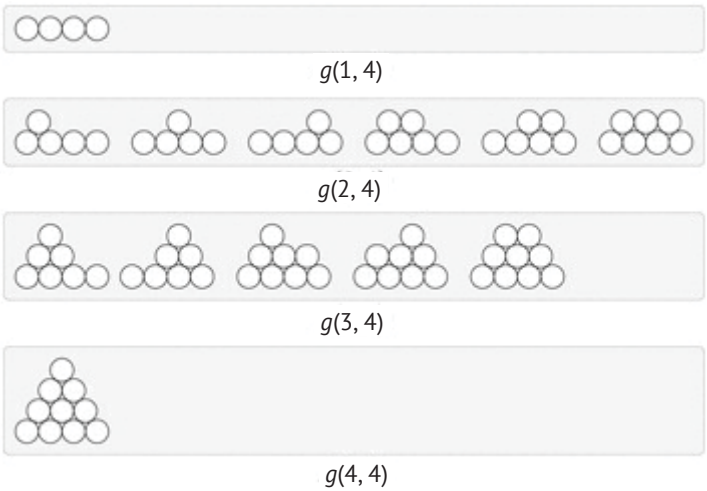


Рис. 8.21. Пирамиды на базе $n = 4$ кругов нижнего ряда, сгруппированные по их высоте

Глава 9

Взаимная рекурсия

В одиночку мы можем так мало, а вместе – так много.

– Хелен Келлер

Метод, явно вызывающий сам себя, рекурсивен по определению. Однако рекурсивный метод не обязательно вызывает себя явно. Он может быть вызван другим методом, который, в свою очередь, вызывает его. Таким образом, вызов метода может повлечь множество последующих обращений к нему. Такой тип рекурсии называют взаимной, или косвенной, рекурсией.

Вообще говоря, несколько методов взаимно-рекурсивны, если они вызывают друг друга циклически. Для примера рассмотрим множество методов $\{f_1, f_2, \ldots, f_n\}$. Если f_1 вызывает f_2, f_2 вызывает f_3 и т. д. и, наконец, f_n вызывает f_1, то говорят, что эти методы взаимно-рекурсивны. При этом они не обязаны вызывать друг друга строго циклически. Они могут вызвать несколько методов, включая себя, как показано на рис. 9.1, где стрелки означают вызов метода (например, f_8 вызывает f_1 и f_4). В этом примере только f_3 вызывает себя явно, а все остальные методы взаимно-рекурсивны.

На первом шаге применения взаимной рекурсии задача разбивается на несколько разных задач (а не просто подзадач). После чего каждая из этих задач решается рекурсивно, опираясь на решения своих подзадач. Для взаимной рекурсии характерно то, что при решении некоторой задачи *P* используются рекурсивные решения подзадач, которые не обязательно являются меньшими экземплярами *P*. Конечно, разбиение исходной задачи на несколько разных задач требует больше труда. Но как только эти задачи определены, процесс рекурсивного проектирования уподобляется работе с единственным методом. Иными словами, мы,

как обычно, применяем декомпозицию и индукцию. Более того, бóльшая сложность задач и характер их декомпозиции только подчеркнут важность этих принципиальных понятий.

В конце главы приводится несколько задач повышенной сложности, некоторые из которых являются вариациями других задач этой книги. Если читатель разберётся в их решениях и сможет самостоятельно разрабатывать алгоритмы, то он приобретёт прочное основание в навыках рекурсивного мышления.

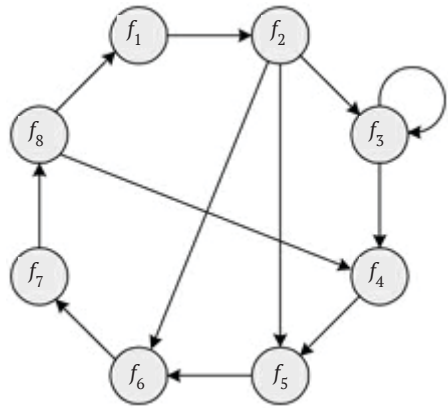


Рис. 9.1. Вызовы взаимно-рекурсивных методов

9.1. Чётность числа

Одна из самых простых и самых популярных задач, используемых для иллюстрации взаимной рекурсии, состоит в определении чётности неотрицательного целого числа n. Мы уже решали задачу определения чётности n (см. листинг 2.6), уменьшая её размер на две единицы. Кроме того, в упражнении 4.1 предлагалось решить задачу с помощью одной линейно-рекурсивной функции. Теперь же мы создадим две взаимозависимые функции – одну, проверяющую n на чётность, и вторую, проверяющую n на нечётность. Таким образом, помимо решения исходной задачи (является ли n чётным), мы решим и другую задачу – является ли n нечётным.

Пусть $f(n)$ проверяет чётность n, а $g(n)$ – нечётность n. Если считать размером задачи n, то начальное условие для $f(n)$ выполняется при $n = 0$ с очевидным результатом True. Напротив, $g(0)$ = False. Для вывода рекурсивных условий можно уменьшать размер задачи на 1. Поскольку мы намерены вывести и $f(n)$, и $g(n)$, на основании индукции можно

предположить, что они применимы к $n - 1$. Крайне важно, что для вывода $f(n)$ нужно использовать $g(n - 1)$, а для вывода $g(n)$ – $f(n - 1)$. С этой целью можно воспользоваться, например, такими простейшими свойствами: $f(n) = g(n - 1)$ и $g(n) = f(n - 1)$. Тогда два взаимно-рекурсивных метода можно закодировать, как в листинге 9.1.

Листинг 9.1. Взаимно-рекурсивные функции для определения чётности неотрицательного целого числа *n*

```
def is_even(n):
    if n == 0:
        return True
    else:
        return is_odd(n - 1)


def is_odd(n):
    if n == 0:
        return False
    else:
        return is_even(n - 1)
```

9.2. Игры со многими игроками

При написании подпрограмм хорошей общепринятой практикой программирования является то, чтобы сделать их по возможности короткими и простыми, что заметно повышает удобочитаемость кода. В этом смысле взаимная рекурсия может быть полезной для реализации различных действий или способов поведения. Например, в играх со многими игроками различные роли игроков можно реализовать отдельными методами. Следующий пример моделирует две различные стратегии поведения игроков в простой игре. Из кучки, состоящей изначально из n камешков, два игрока по очереди убирают один или два камешка. Выигрывает тот, кто уберёт последние камешки. Допустим, первый игрок (Боб) решил всякий раз убирать по одному камешку, тогда как второй игрок (Элис) убирает один камешек, если оставшееся их число нечётно, и два – если чётно.

Эту простую игру можно легко закодировать, используя всего один метод. Однако в листинге 9.2 приводится решение, основанное на двух взаимно-рекурсивных методах, в каждом из которых реализована своя

стратегия игры. В частности, в начальных условиях проверяется, выиграет ли игрок игру, тогда как в рекурсивных условиях очередь передаётся другому игроку. Параметр функции, реализующей стратегию другого игрока, уменьшается на число убранных из кучки камешков. Важно отметить, что каждая стратегия реализована отдельной процедурой, что упрощает понимание и изменение кода. Наконец, более сложная стратегия Элис явно лучше, чем у Боба. В частности, Боб победит только тогда, когда исходная кучка состоит из одного камешка.

Листинг 9.2. Взаимно-рекурсивные процедуры, реализующие игровые стратегии Элис и Боба

```
def play_Alice(n):
    if n <= 2:
        print('Alice wins')
    elif n & 1:
        # Alice removes one pebble
        play_Bob(n - 1)     # Turn switches to Bob
    else:
        # Alice removes two pebbles
        play_Bob(n - 2)     # Turn switches to Bob


def play_Bob(n):
    if n == 1:
        print('Bob wins')
    else:
        # Bob removes one pebble
        play_Alice(n - 1)     # Turn switches to Alice
```

9.3. Размножение кроликов

Следующая задача из разряда умозрительных касается искусственного размножения кроликов, число которых растёт по следующим правилам.

1. Новорождённая пара кроликов, самец и самка, пасётся один месяц до полового созревания.
2. Зрелая пара кроликов спаривается в начале каждого месяца и ровно через месяц рожает новую пару кроликов.
3. Крольчиха всегда рожает одного самца и одну самку, которые спариваются только между собой.
4. Кролики никогда не умирают.

Цель задачи (подсчёта) – вычислить общее количество $R(n)$ кроликов через n месяцев. В следующих подразделах приводится два решения на основе взаимной рекурсии.

9.3.1. Зрелые и незрелые пары кроликов

Вместо прямого подсчёта объёма всей популяции можно подсчитать число зрелых и незрелых пар для каждого месяца в отдельности. Это приводит к двум различным задачам размера n. В частности, пусть $A(n)$ и $B(n)$ обозначают, соответственно, количество зрелых и незрелых пар в каждом месяце n. Тогда начальными условиями будут $A(1) = 0$ и $B(1) = 1$, поскольку вначале есть только одна пара незрелых кроликов. На втором месяце появится пара кроликов, еще не способных производить потомство. Таким образом, $B(2) = 0$ и $A(2) = 1$.

Для рекурсивных условий правила роста популяции показаны на рис. 9.2, где маленькие и большие кролики изображают незрелые и зрелые пары соответственно. С одной стороны, каждая пара незрелых кроликов в данном месяце созревает во взрослую пару в следующем месяце. С другой стороны, каждая зрелая пара появляется и в следующем месяце (поскольку кролики никогда не умирают), и она также спаривается, создавая новую пару потомства.

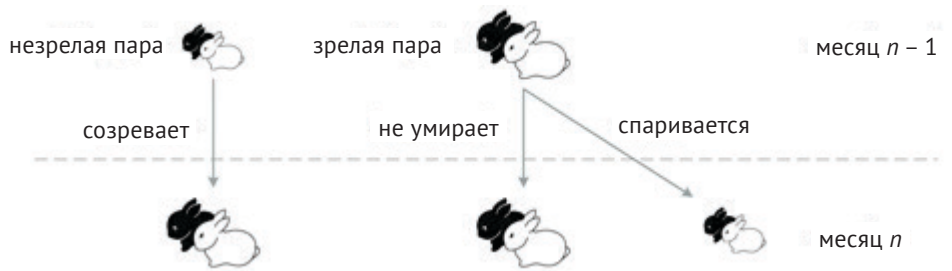


Рис. 9.2. Правила размножения кроликов

Согласно методу индукции, нам известен объём популяции зрелых и незрелых пар в месяце $n - 1$ (то есть $A(n - 1)$ и $B(n - 1)$), откуда довольно просто вывести рекурсивные условия для $A(n)$ и $B(n)$. Во-первых, число незрелых пар в n-м месяце равно числу зрелых пар в прошлом месяце, поскольку зрелой паре нужен один месяц, чтобы спариться и произвести новую пару незрелых кроликов. Поэтому $B(n) = A(n - 1)$. Что касается $A(n)$, то все зрелые пары, жившие в месяце $n - 1$, будут живы и в месяце n, так как кролики не умирают. Кроме того, все незрелые пары в течение месяца $n - 1$ созревают и превращаются в зрелые. Таким образом,

рекурсивное условие – $A(n) = A(n-1) + B(n-1)$. Обе функции можно определить следующим образом:

$$A(n) = \begin{cases} 0, & \text{если} \quad n = 1, \\ A(n-1) + B(n-1), & \text{если} \quad n > 1, \end{cases} \tag{9.1}$$

$$B(n) = \begin{cases} 1, & \text{если} \quad n = 1, \\ A(n-1), & \text{если} \quad n > 1. \end{cases} \tag{9.2}$$

В них не нужны явные начальные условия для $n = 2$, так как они – именно те самые функции, которые были введены в разделе 1.6.4. Ясно, что A – рекурсивная функция, потому что она вызывает себя. Метод B вызывает себя косвенно – через вызов A, который, в свою очередь, вызывает B. В листинге 9.3 приведён код, соответствующий этим функциям.

Листинг 9.3. Взаимно-рекурсивные функции подсчёта популяции незрелых и зрелых пар кроликов спустя *n* месяцев

```
def adults(n):
    if n == 1:
        return 0
    else:
        return adults(n - 1) + babies(n - 1)


def babies(n):
    if n == 1:
        return 1
    else:
        return adults(n - 1)
```

Эти функции можно выразить исключительно через самих себя, как показано в (3.38) и (3.39). Наконец, общее количество пар кроликов – это просто сумма $A(n) + B(n)$, которая оказывается n-м числом Фибоначчи (см. упражнение 9.2)[1].

9.3.2. Родовое дерево кроликов

Совершенно иной взгляд на эту задачу – рассмотреть родовое (генеалогическое) древо кроликов. На рис. 9.3 показано такое дерево спустя

[1] Нелишне напомнить, что автором этой задачи был сам Леонардо Пизанский по прозвищу Фибоначчи. – *Прим. перев.*

7 месяцев, где метки рядом с кроликами указывают на их возраст (в месяцах). Первая пара кроликов – начальная, отправившаяся пастись в 1-й месяц. Заметьте, что у неё есть 5 пар потомков, первая из которых появилась в начале 3-го месяца (поэтому её возраст – 5), а последняя – в начале 7-го месяца (значит, в конце 7-го месяца этим кроликам 1 месяц). В следующих подразделах разбираются решения, основанные на множественной и взаимной рекурсиях.

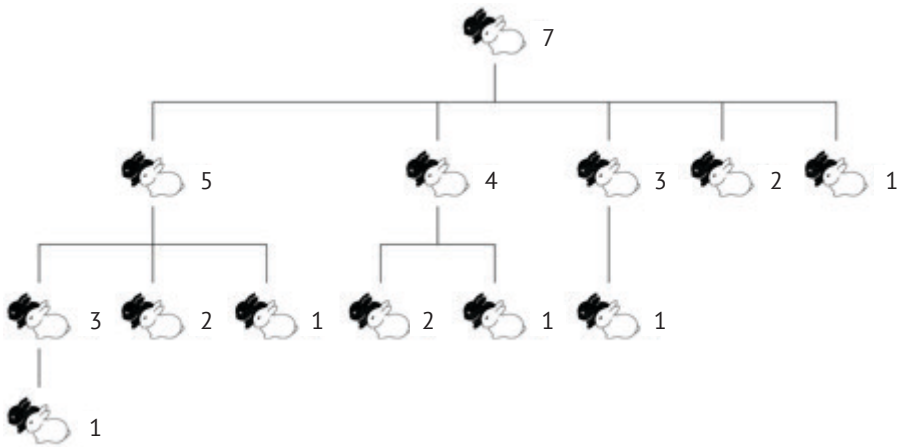


Рис. 9.3. Родовое дерево кроликов спустя 7 месяцев

9.3.2.1. *Решение с множественной рекурсией*

Первая приходящая на ум в связи с этой задачей декомпозиция – разбить задачу так, как показано на рис. 9.4.

В этом конкретном примере задача размера 7 разбита на пять меньших задач размером от 1 до 5. Заметим, что начальная пара кроликов с третьего месяца создаёт новую пару потомков, каждая из которых уже имеет своих потомков. Каждая пара потомков вместе со своими потомками образует меньшую задачу, подобную исходной. Таким образом, в рекурсивном условии искомая функция $f(n)$ – это пара, прожившая n месяцев плюс общее количество её потомков. Формально декомпозиция приводит к следующему определению:

$$f(n) = \begin{cases} 1, & \text{если } n = 1 \text{ или } n = 2, \\ 1 + \sum_{i=1}^{n-1} f(i), & \text{если } n > 2, \end{cases} \tag{9.3}$$

где используется множественная рекурсия, подобная (1.7). Более того, схема декомпозиции задачи на рис. 9.4 пригодна для любого n (напомним, что $f(n) = F(n)$, где F – функция Фибоначчи).

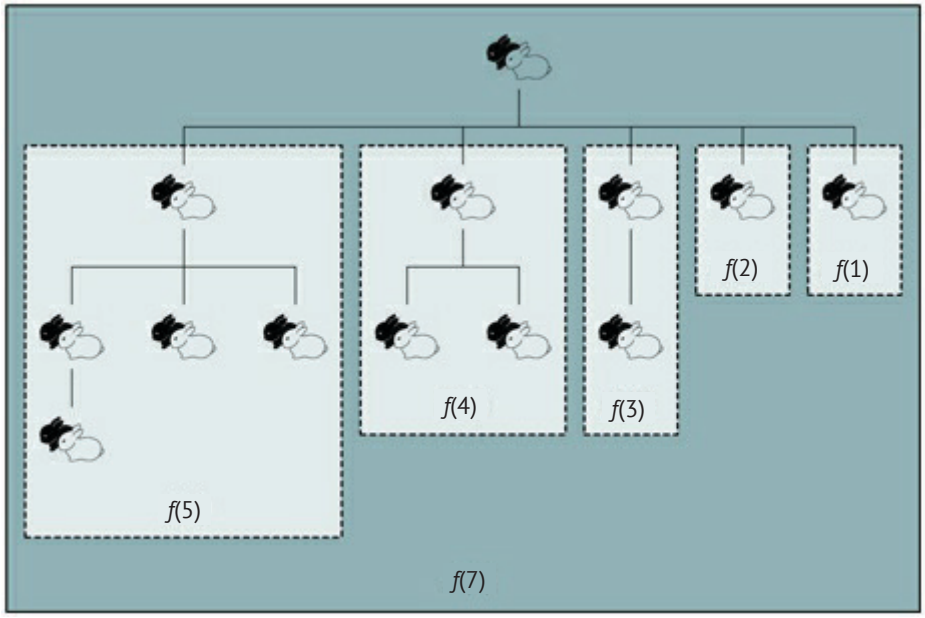


Рис. 9.4. Конкретный пример разбиения задачи о росте популяции кроликов на подобные ей подзадачи

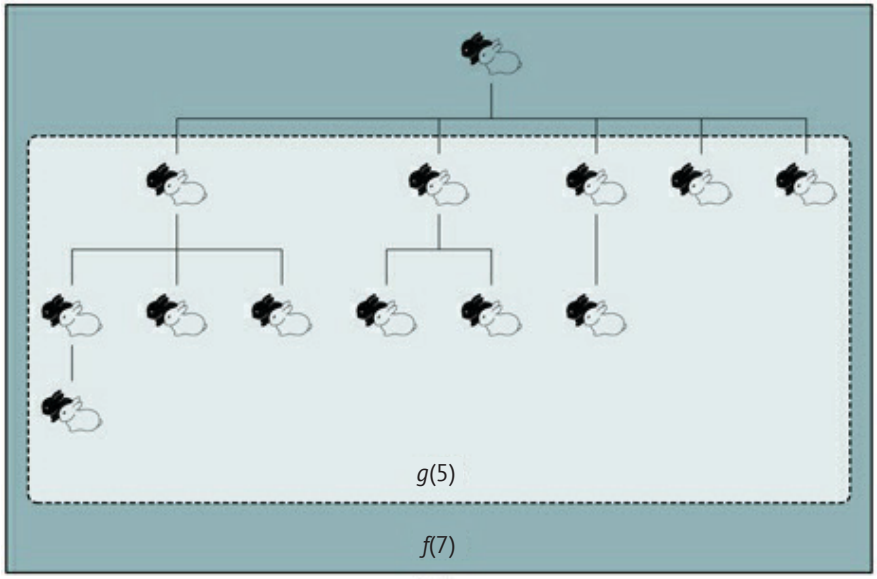
9.3.2.2. *Решение со взаимной рекурсией*

Рекурсивное выражение (9.3) можно вычислить в цикле (см. листинг 1.4). Решение, к которому мы хотим прийти, заменяет суммирование рекурсивной функцией (это важно, если язык программирования не поддерживает циклы).

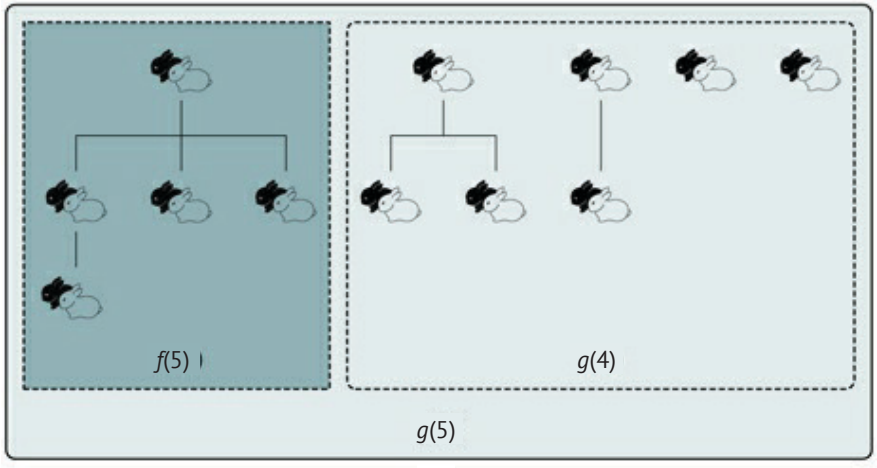
Во-первых, взаимная рекурсия возникает только тогда, когда в исходной задаче можно выделить несколько разных задач. В данном случае, помимо функции $f(n)$, решающей исходную задачу, мы рассмотрим ещё и решение похожей, но иной задачи. А именно пусть $g(n)$ – количество пар детей в возрасте от 1 до n месяцев вместе с их потомками от начальной пары. С помощью этой вспомогательной функции просто определить $f(n)$, а именно: для заданного n она подсчитывает всю семью начальной пары кроликов плюс $g(n - 2)$, поскольку начальная пара всегда на 2 месяца старше самых старших своих детей. Таким образом, $f(n)$ можно выразить так:

$$f(n) = \begin{cases} 1, & \text{если } n = 1 \text{ или } n = 2, \\ 1 + g(n-2), & \text{если } n > 2, \end{cases} \tag{9.4}$$

что соответствует декомпозиции на рис. 9.5(a) при $n = 7$.



(a)



(b)

Рис. 9.5. Декомпозиция задачи о размножении кроликов, приводящая к двум взаимно-рекурсивным функциям

Для функции g размер задачи – n. Тривиальное начальное условие с результатом 1 выполняется при $n = 1$. Но можно также использовать начальное условие $g(0) = 0$. Вывод рекурсивного условия основан на декомпозиции, уменьшающей размер задачи на 1 (см. рис. 9.5(b)). Заметим, что общее количество пар кроликов – это сумма количества пар для некоторой начальной пары спустя n месяцев $f(n)$ и количества детей в возрасте не старше $n - 1$ месяцев вместе с их потомками $g(n - 1)$. Следовательно, $g(n)$ можно записать так:

$$g(n) = \begin{cases} 0, & \text{если } n = 0, \\ f(n) + g(n-1), & \text{если } n > 0. \end{cases} \tag{9.5}$$

В результате мы получили пару взаимно-рекурсивных функций, которые вызывают друг друга. Кроме того, отметим, что $g(n)$ просто равна $\sum_{i=1}^{n} f(i)$. Таким образом, (9.4) и (9.5) равносильны (9.3). В итоге можно без труда закодировать эти функции, как показано в листинге 9.4.

Листинг 9.4. Альтернативные взаимно-рекурсивные функции подсчёта популяции кроликов спустя *n* месяцев

```
def population_rabbits(n):
    if n <= 2:
        return 1
    else:
        return 1 + children_descendants(n - 2)


def children_descendants(n):
    if n == 0:
        return 0
    else:
        return (population_rabbits(n)
                + children_descendants(n - 1))
```

9.4. Задача о станциях водоочистки

В задаче «Станции водоочистки» есть n городов со станциями водоочистки, расположенных по одну сторону реки, как показано на рис. 9.6.

Допустим, что города упорядочены слева направо. Каждый город – источник грязной воды, которую необходимо очистить на станции (не

обязательно своей) и сбросить в реку по трубам, соединяющим соседние города. Работающая станция забирает сточные воды своего и соседних городов и после очистки сбрасывает их в реку. Но станция может не работать, и тогда сточная вода её города плюс вода любого соседнего с ней города направляется в другой город. Учитывая, что вода по трубе, соединяющей города, может течь лишь в одном из направлений, задача состоит в вычислении количества способов сброса очищенной воды в реку для n городов. Кроме того, мы должны быть уверены в том, что хотя бы одна из станций работает, то есть способна очистить и сбросить очищенную воду в реку.

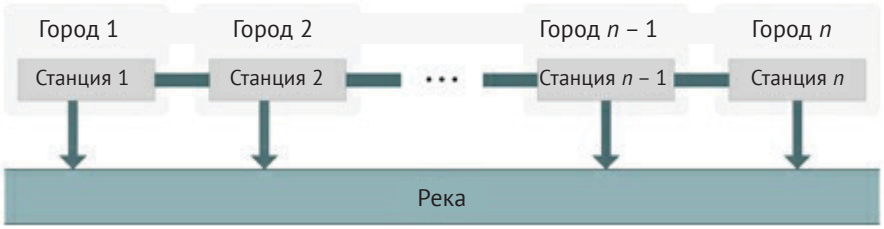


Рис. 9.6. Задача о станциях водоочистки

Мы рассмотрим два рекурсивных решения этой задачи подсчёта. Первое моделирует переток воды между городами и использует множественную рекурсию. Второе рассматривает сброс воды в реку по каждому городу и опирается на три взаимно-рекурсивные функции.

9.4.1. Переток воды между городами

Вода по каждой из соединяющих соседние города n – 1 труб может перетекать в трёх направлениях: '**N**' – перетока нет, '**R**' – направо и '**L**' – налево. Следовательно, сброс воды в реку можно смоделировать строкой длины n – 1, состоящей только из этих символов. Единственное ограничение, касающееся перетока воды, состоит в том, что город не имеет права направить свою воду обоим своим соседям. Таким образом, цель задачи в том, чтобы найти все строки, в которых нет подстроки "**LR**".

Очевидно, что размер задачи – n. Пусть $f(n)$ представляет решение задачи. Если есть только один город, то его водоочистная станция может сбросить в реку лишь свою воду, то есть $f(1) = 1$. Для двух городов существует 3 варианта строки размера 1 с одним из трех символов, то есть $f(2) = 3$. Заметим, что если обе станции работают, они сбросят в реку только воду своих городов, а если какая-то из них не работает, она направит свою воду на соседнюю станцию.

На рис. 9.7 представлена декомпозиция задачи, приводящая к алгоритму с множественной рекурсией.

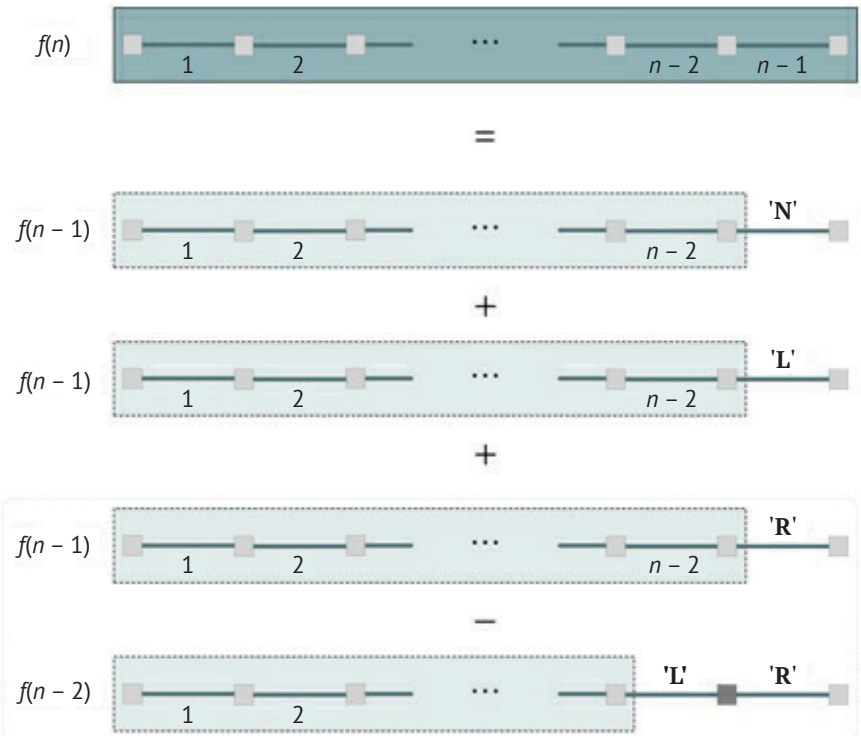


Рис. 9.7. Декомпозиция задачи о станциях водоочистки, моделирующая переток воды между городами

Сначала общее количество возможных строк длины $n - 1$ втрое больше количества правильных строк длины $n - 2$, так как к каждой из них можно добавить один из трёх символов, что даёт слагаемое $3f(n - 1)$. Однако к строке с окончанием '**L**' нельзя добавить '**R**', поскольку тогда станция $n - 1$ направляла бы воду в обоих направлениях, что недопустимо. Поэтому все возможные строки длины $n - 1$ с окончанием "**LR**" необходимо исключить, а их количество равно $f(n - 2)$. Таким образом, рекурсивное правило – $f(n) = 3f(n - 1) - f(n - 2)$. Вместе с начальными условиями функцию можно определить как

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ 3, & \text{если } n = 2, \\ 3f(n-1) - f(n-2), & \text{если } n > 2, \end{cases} \tag{9.6}$$

и можно легко реализовать, как показано в листинге 9.5. Наконец, $f(n) = F(2n)$, где F – функция Фибоначчи (см. упражнение 9.4).

Листинг 9.5. Функция с множественной рекурсией для решения задачи о станциях водоочистки

```
def water_multiple(n):
    if n == 1:
        return 1
    elif n == 2:
        return 3
    else:
        return 3 * water_multiple(n - 1) - water_multiple(n - 2)
```

9.4.2. Сброс воды в каждом городе

Другой подход к решению задачи заключается в моделировании направлений передачи воды станциями. Существует 3 направления передачи воды, которые мы обозначим следующими символами: 'v' – сброс в реку, '>' – переток вправо, '<' – переток влево. В этом случае мы должны подсчитать все строки длины n, которые не начинаются с '<', не заканчиваются на '>' и не содержат подстрок "><".

Для создания алгоритма со взаимной рекурсией будем использовать три похожие, но разные задачи. Рассмотрим правильные строки из трёх символов, начинающихся (слева направо) с непустой подстроки и заканчивающихся подстрокой из n символов. В зависимости от начального символа, предшествующего подстроке длины n (подстрока слева от символа неуместна), существует 3 сценария (подзадачи), которые представлены схемами на рис. 9.8 и заключаются в определении количества правильных подстрок в каждом из трёх случаев. Их решения обозначены функциями $f_v(n)$, $f_>(n)$ и $f_<(n)$, где нижний индекс указывает на символ, предшествующий подстроке длины n.

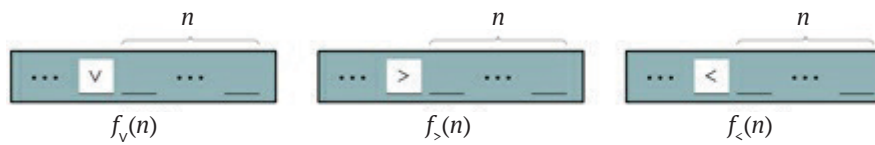


Рис. 9.8. Три сценария взаимно-рекурсивного решения задачи о станциях водоочистки

Очевидно, что размер задач – n, а их начальные условия выполняются при $n = 1$. В частности, $f_v(1) = f_<(1) = 2$, так как существует всего два вари-

анта ('v' и '<') для самого последнего символа. Наоборот, $f_{>}(1) = 1$, так как последняя станция очистки, получившая воду от своего соседа, должна сбросить её в реку.

На рис. 9.9 приведены декомпозиции задач. Для $f_{v}(n)$ и $f_{<}(n)$ первый символ подстроки длины n может быть любым. Поэтому общее количество правильных строк длины n равно количеству правильных строк длины $n - 1$ для каждого из трёх первых символов.

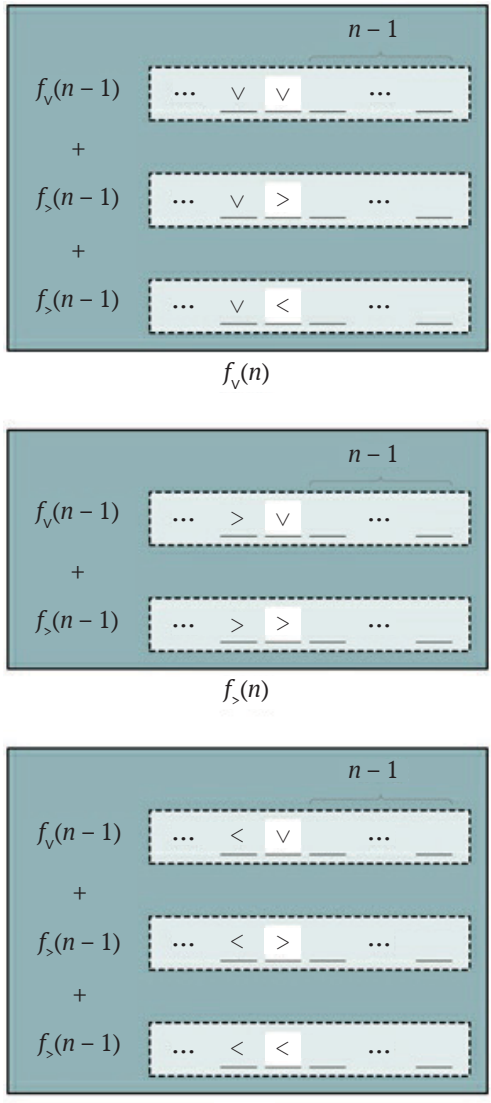


Рис. 9.9. Декомпозиции трех подзадач для взаимно-рекурсивного решения задачи о станциях водоочистки

Таким образом, в правой части рекурсивного правила для обеих функций будет формула $f_{\vee}(n-1) + f_{>}(n-1) + f_{<}(n-1)$. Кроме того, поскольку их начальные условия тоже одинаковы, одинаковыми будут и сами функции. В результате мы имеем:

$$f_{\vee}(n) = f_{<}(n) = \begin{cases} 2, & \text{если } n = 1, \\ f_{\vee}(n-1) + f_{>}(n-1) + f_{<}(n-1), & \text{если } n > 1. \end{cases} \quad (9.7)$$

Наоборот, для $f_{>}(n)$ символ '<' не может быть первым в строке длины n. Поэтому в правой части её рекурсивного правила будет $f_{\vee}(n-1) + f_{>}(n-1)$. А сама функция вместе с ее начальным условием будет такой:

$$f_{>}(n) = \begin{cases} 1, & \text{если } n = 1, \\ f_{\vee}(n-1) + f_{>}(n-1), & \text{если } n > 1. \end{cases} \quad (9.8)$$

Итак, решение $f(n)$ исходной задачи можно выразить двояко:

$$f(n) = f_{\vee}(n-1) + f_{>}(n-1) = f_{>}(n).$$

Первое равенство говорит о том, что первым символом строки может быть или 'v', или '<', и в этом случае мы должны сложить число правильных строк длины $n-1$, начинающихся с этого символа. Второе равенство следует из рекурсивного правила для $f_{>}(n)$. Наконец, можно показать, что $f_{\vee}(n) = f_{<}(n) = F(2n+1)$, а $f_{>}(n) = F(2n)$, где F – функция Фибоначчи (см. упражнение 9.5).

9.5. Циклические ханойские башни

Следующая задача является разновидностью классической задачи «Ханойская башня» (см. раздел 7.2) и известна как «циклические» ханойские башни. Несмотря на то что вводимое в ней ограничение усложняет задачу, принцип её решения и само решение (то есть перемещение отдельных дисков или стопок из нескольких дисков) аналогичны. Таким образом, она представляет собой превосходное дополнение к оригинальной задаче, которое поможет читателю лучше понять её решение. Кроме того, она подтверждает важность рекурсии для решения задач вообще, так как (насколько известно автору) единственный из итерационных алгоритмов решения этой задачи просто моделирует её рекурсивное решение, используя стек (см. раздел 10.3).

Правила циклических ханойских башен идентичны правилам оригинальной задачи с той лишь разницей, что перемещать отдельные диски

можно только циклически (по кругу). Например, если *A*, *B* и *C* обозначают три стержня, то диск можно перемещать только с *A* на *B*, с *B* на *C* или с *C* на *A*. Задача становится наглядней, если стержни расположить в виде треугольника и разрешить перемещения дисков только по кругу в определенном направлении, скажем по часовой стрелке, как на рис. 9.10.

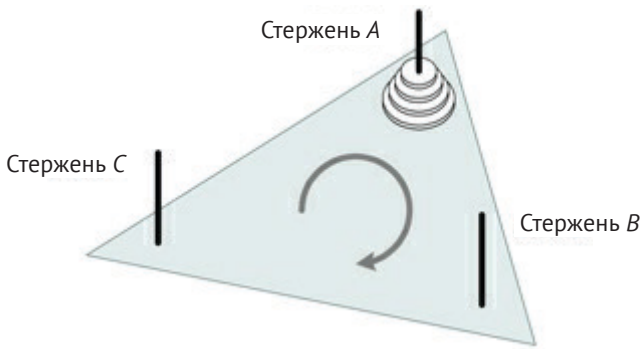
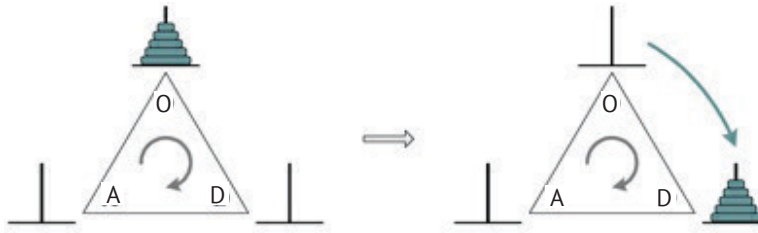


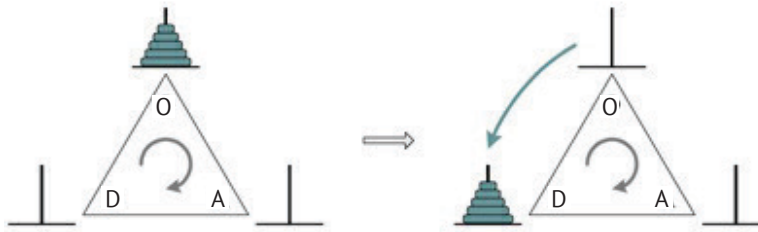
Рис. 9.10. Циклические ханойские башни

Задача заключается в том, чтобы переместить всю стопку из *n* дисков с исходного (начального) стержня, скажем *A*, на другой заданный (конечный) стержень, используя третий вспомогательный (промежуточный) стержень. Обратите внимание, что условие задачи не оговаривает, какой именно стержень, *B* или *C,* должен быть конечным. Несмотря на то что мы условились перемещать диски по часовой стрелке, важно понимать, что на самом деле можно написать разные процедуры для перемещения всей стопки из *n* дисков как по часовой стрелке, так и против часовой стрелки. Иными словами, задача о циклических ханойских башнях состоит из двух разных задач: перемещение *n* дисков *по* часовой стрелке или *против* часовой стрелки, но при этом каждый отдельный диск должен перемещаться по кругу только в одном направлении. Обе задачи изображены на рис. 9.11, где O, D и A обозначают соответственно начальный, конечный и вспомогательный стержни. Заметьте, что конечный и вспомогательный стержни меняются местами в зависимости от варианта задачи.

Понятно, что размер обеих задач – *n*. Первое начальное условие имеет место при $n = 1$. Это тривиальное решение предполагает одно перемещение единственного диска по часовой стрелке или два перемещения того же диска против часовой стрелки. Второе начальное условие, подобно функции `towers_of_Hanoi_alt` в листинге 7.3, предупреждает какие-либо действия при $n = 0$.



Переместить *n* дисков по часовой стрелке



Переместить *n* дисков против часовой стрелки

Рис. 9.11. Две разновидности задачи о циклических ханойских башнях

Для вывода рекурсивных условий можно применить декомпозицию задачи с уменьшением её размера на 1, что приведёт к трём действиям, лежащим в основе рекурсивных методов её решения (см. рис. 9.12).

Во-первых, согласно правилам задачи, отдельный диск можно переместить по часовой стрелке. Во-вторых, согласно индукции, стопку из $n - 1$ дисков можно переместить либо по часовой стрелке, либо против неё.

На рис. 9.13 показаны шаги решения задачи по перемещению стопки из n дисков по часовой стрелке с начального стержня на конечный. Сначала нужно переместить $n - 1$ дисков на вспомогательный стержень, так как иначе невозможно переместить наибольший диск. Добиться этого можно перемещением стопки из $n - 1$ дисков против часовой стрелки с начального стержня на вспомогательный при посредстве конечного стержня (шаг 1). На следующем, основном шаге 2 на конечный стержень переносится наибольший диск. А затем $n - 1$ дисков переносятся со вспомогательного стержня на конечный, что также осуществляется их перемещением против часовой стрелки (шаг 3).

Процедура `clockwise` в листинге 9.6 реализует этот метод. Ход рассуждений при решении данной задачи очень похож на тот, что применялся при решении оригинальной задачи о ханойских башнях. Таким образом, алгоритм почти совпадает с кодом листинга 7.3, за исключением того, что рекурсивные вызовы обращаются ко второму методу. Заметим, что при программировании алгоритма в целом мы предпола-

гаем, что перемещения против часовой стрелки работают правильно, даже если мы не написали для них ни одной строчки кода.

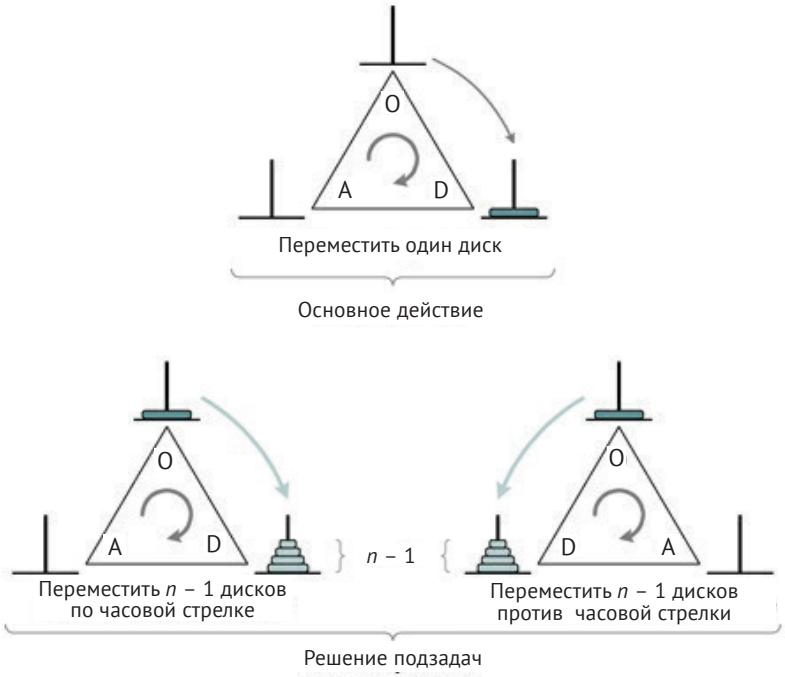


Рис. 9.12. Три действия рекурсивных методов, решающих задачу о циклических ханойских башнях

Листинг 9.6. Взаимно-рекурсивные процедуры задачи о циклических ханойских башнях

```
def clockwise(n, o, d, a):
    if n > 0:
        counterclockwise(n - 1, o, a, d)
        print('Move disk', n, 'from rod', o, 'to rod', d)
        counterclockwise(n - 1, a, d, o)


def counterclockwise(n, o, d, a):
    if n > 0:
        counterclockwise(n - 1, o, d, a)
        print('Move disk', n, 'from rod', o, 'to rod', d)
        clockwise(n - 1, d, o, a)
        print('Move disk', n, 'from rod', a, 'to rod', d)
        counterclockwise(n - 1, o, d, a)
```

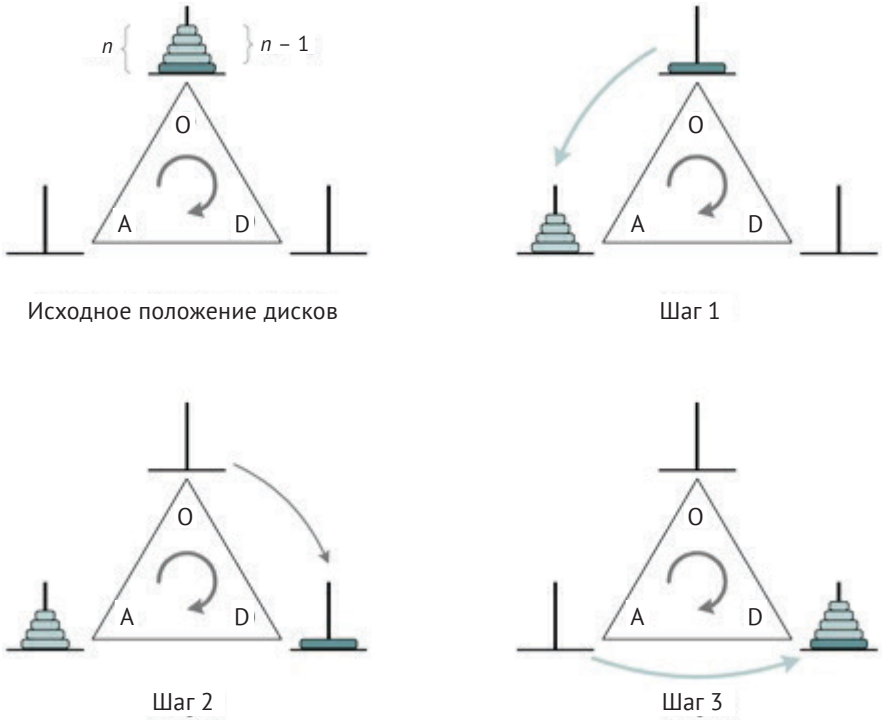
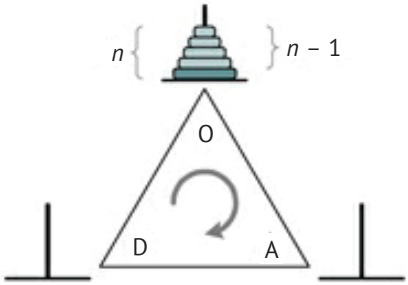


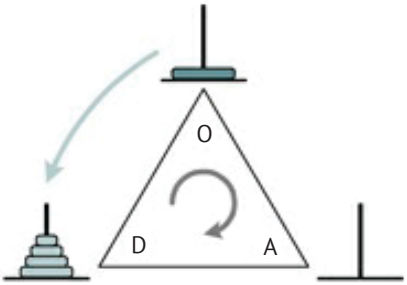
Рис. 9.13. Действия по перемещению *n* дисков по часовой стрелке

Наконец, на рис. 9.14 показаны действия по перемещению стопки из *n* дисков против часовой стрелки, осуществляемые процедурой `counterclockwise` из листинга 9.6.

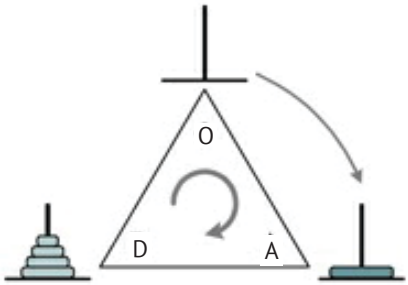
В этом случае требуется больше работы, так как для воспроизведения единственного перемещения наибольшего диска по часовой стрелке здесь необходимо два его перемещения против часовой стрелки. Первый шаг с рекурсивным вызовом – это *n* – 1 перемещений против часовой стрелки, обеспечивающих выполнение основного действия по переносу наибольшего диска на вспомогательный стержень (шаг 2). На шаге 3 стопка из *n* – 1 дисков возвращается по часовой стрелке на начальный стержень (вызывая уже реализованный метод `clockwise`), чтобы тем самым подготовить второе основное действие на шаге 4 – перемещение наибольшего диска на конечный стержень. После этого выполняется заключительное действие – перемещение против часовой стрелки стопки из *n* – 1 дисков на конечный стержень.



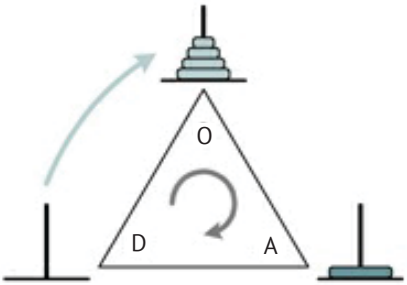
Исходное положение дисков



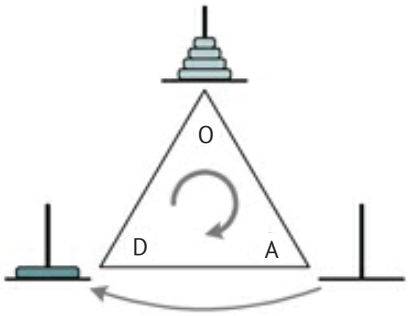
Шаг 1



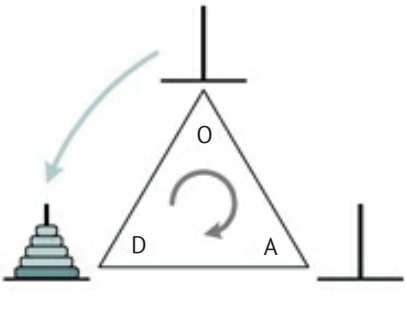
Шаг 2



Шаг 3



Шаг 4



Шаг 5

Рис. 9.14. Действия по перемещению *n* дисков против часовой стрелки

9.6. Грамматики и синтаксический анализатор на основе рекурсивного спуска

В информатике *формальная грамматика* – это набор *порождающих правил* (production rules) или *продукций*, определяющих синтаксис *формального языка*. Иными словами, она описывает, как соединять символы *алфавита* формального языка и составлять из них цепочки (слова, символы, числа, знаки препинания и т. д.), называемые *лексемами*. В этом разделе мы рассмотрим более сложный пример, касающийся понятий, с которыми обычно знакомятся в расширенных курсах по компиляторам и языковым процессорам. В частности, мы реализуем калькулятор, который получает на входе строку, представляющую собой математическое выражение, и выдаёт на выходе её значение. Калькулятор должен уметь складывать, вычитать, умножать и делить числа. Для простоты числа будут целыми (хотя результат может быть вещественным числом). Кроме того, калькулятор должен уметь обрабатывать круглые скобки и унарную (одноместную) операцию минус, которой предшествует левая круглая скобка.

Построение калькулятора состоит из двух этапов. На первом исходная строка *s*, представляющая собой математическое выражение, разбивается на последовательность лексем. На втором, согласно правилам формальной грамматики, выполняется разбор последовательности лексем *методом рекурсивного спуска* с целью вычисления исходного выражения. На практике при решении таких задач применяются мощные инструментальные средства и библиотеки. Однако мы ограничимся низкоуровневыми алгоритмами, чтобы рассмотреть как можно больше примеров взаимной рекурсии. Естественно, полный обзор упомянутых тем и понятий выходит далеко за пределы настоящей книги.

9.6.1. Лексический анализ входной строки

Цель этого этапа заключается в преобразовании входной строки, представляющей собой математическое выражение, в последовательность (список) лексем, которыми могут быть:

- положительные или отрицательные целые числа: последовательность цифр, которой может предшествовать символ '–';
- операции: '+', '–', '*', '/';
- круглые скобки: '(' и ')'.

Например, для входной строки "–(–6 / 3) – (–(4)) + (18–2)" алгоритм на первом этапе должен создать такой список:

['–', '(', '–6', '/', '3', ')', '–', ')', '–', '(', '4', ')', ')', '+', '(', '18', '–', '2', ')'].

Прежде всего алгоритм игнорирует символы пробела и табуляции (пропуски). Кроме того, отметим, что знак минус может выполнять разные роли: помимо обозначения отрицательного числа, он может быть также либо одноместной, либо двуместной (бинарной) операцией. Например, алгоритм должен считать –6 единым числом и потому не должен создавать две отдельные лексемы для (унарного) минуса и числа 6. Пример также показывает, что унарный минус может появиться в начале входной последовательности или между двумя левыми скобками. Взаимно-рекурсивные функции, которые мы вскоре рассмотрим, возникают из-за необходимости обработки унарных минусов в математических выражениях. Наконец, отметим, что лексемой может быть число, состоящее из нескольких цифр, возможно, со знаком минус.

Для решения этой задачи воспользуемся двумя похожими методами `tokenize_unary` и `tokenize` из листинга 9.7, которые получают на входе строку *s* длины *n* и возвращают список лексем. Разница между ними лишь в том, что первый предполагает, что первым непустым символом *s* может быть унарный минус. Оба метода посредством функции `is_number` из листинга 9.8 проверяют также, является ли строка целым числом.

Размер задач – длина входной строки (*n*). Существует несколько начальных условий. Если строка пуста или содержит лишь пропуски, методы должны вернуть пустой список. Кроме того, если строка состоит из единственного непустого символа (предыдущее условие уже «отсекло» все пропуски), то начальное условие должно вернуть список, содержащий этот символ. Оба этих условия гарантируют, что в рекурсивных условиях входная строка будет содержать не менее двух символов. Следовательно, мы можем спокойно обращаться ко второму элементу строки, не опасаясь выхода её индекса за границы строки.

Рекурсивные условия этой задачи гораздо сложнее всех других, так как есть несколько моментов, которые следует принять во внимание. Во-первых, если первый элемент входной строки – пропуск, то им нужно пренебречь. В этом случае оба метода, вызвав самих себя, должны вернуть лишь остаток входной строки ($s_{1\ldots n-1}$).

Листинг 9.7. Взаимно-рекурсивные функции лексического анализа математического выражения

```
def tokenize_unary(s):
    if not s or s == ' ' or s == '\t':
        return []
    elif len(s) == 1:
        return [s]
    elif s[0] == ' ' or s[0] == '\t':
        return tokenize_unary(s[1:])
    elif s[0].isdigit() or s[0] == '-':
         if s[0].isdigit() and (s[1] == ' ' or s[1] == '\t'):
            return [s[0]] + tokenize(s[2:])
         else:
            t = tokenize(s[1:])
            if t == []:
               return [s[0]]
            else:
               if is_number(t[0]):
                  t[0] = s[0] + t[0]
                  return t
               else:
                  return [s[0]] + t
    else:
        if s[0] == '(':
            t = tokenize_unary(s[1:])
        else:
            t = tokenize(s[1:])
        return [s[0]] + t


def tokenize(s):
    if not s or s == ' ' or s == '\t':
        return []
    elif len(s) == 1:
        return [s]
    elif s[0] == ' ' or s[0] == '\t':
        return tokenize(s[1:])
    elif s[0].isdigit():
         if s[1] == ' ' or s[1] == '\t':
            return [s[0]] + tokenize(s[2:])
         else:
            t = tokenize(s[1:])
            if t == []:
               return [s[0]]
            else:
               if is_number(t[0]):
                  t[0] = s[0] + t[0]
                  return t
               else:
                  return [s[0]] + t
    else:
        if s[0] == '(':
            t = tokenize_unary(s[1:])
        else:
            t = tokenize(s[1:])
        return [s[0]] + t
```

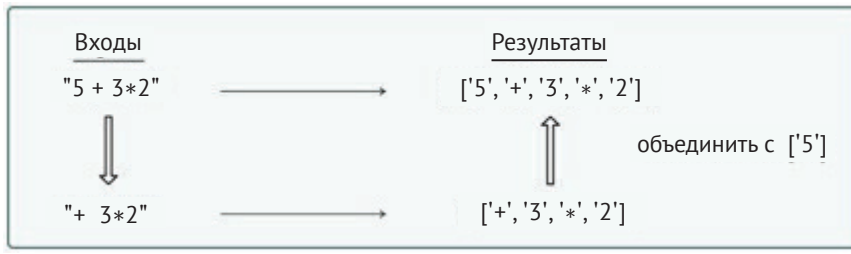
Листинг 9.8. Представляет ли собой строка число?

```
1  def is_number(s):
2      try:
3         float(s)
4          return True
5      exept ValueError:
6          return False
```

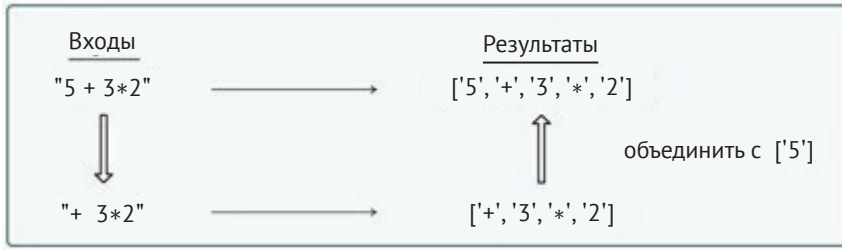
9.6.1.1. *Функция tokenize()*

До рассмотрения прочих рекурсивных условий мы остановимся сначала на более простом лексическом анализе. Следующее рекурсивное условие выполняется, если первый символ входной строки (s_0) является цифрой (строка 36), которая, естественно, относится к некоторому числу. В этом случае алгоритм должен выполнить разные действия в зависимости от следующего символа (s_1) или первой лексемы остальной части строки ($s_{1\ldots n-1}$).

Во-первых, если s_1 – пропуск, то s_0 – лексема. Значит, метод должен вернуть список, первый элемент которого – s_0, плюс список лексем остатка строки $s_{2\ldots n-1}$. Это можно представить следующей схемой:

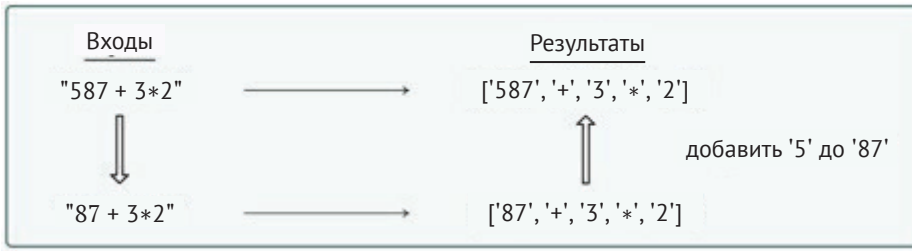


В противном случае метод посредством рекурсивного вызова ищет лексемы в $s_{1\ldots n-1}$ и сохраняет их в списке t (строка 40). Если этот список пуст, то метод просто вернёт лексему, соответствующую s_0 (строка 42). Иначе последующее действие будет зависеть от того, является или нет первая лексема $s_{1\ldots n-1}$ числом. Рассмотрим следующую частную схему:



Поскольку первая лексема *t* ('+') не является числом, алгоритм должен присоединить список, содержащий s_0, к списку *t* (строка 48).

Иначе, если первая лексема *t* – число, метод должен добавить символ s_0 к первой лексеме (строки 45 и 46). Следующая схема иллюстрирует это:



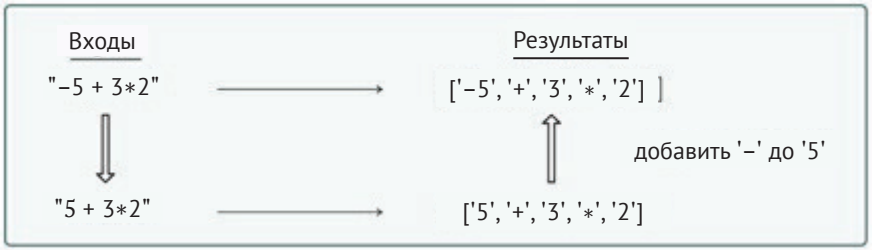
Заметьте, что таким образом алгоритм собирает из отдельных цифр числовую лексему.

Наконец, есть еще два рекурсивных условия, когда первый элемент строки не является цифрой. Поскольку функция уже проверила, что символ не является пропуском (строка 34), он может быть только лексемой – операцией или круглой скобкой. Таким образом, метод добавит эту лексему к результату решения задачи для $s_{1\ldots n-1}$. Однако если символ s_0 является левой круглой скобкой, то первой лексемой оставшейся подстроки может быть унарный минус, и в этом случае метод должен вызвать функцию `tokenize_unary` (строка 51). В любом другом случае унарный минус не может быть первой лексемой, и метод может вызвать просто сам себя (строка 53).

9.6.1.2. *Функция tokenize_unary()*

Функция `tokenize_unary` очень похожа на `tokenize`. Но есть два важных отличия. Во-первых, если в первой позиции строки появляется символ пропуска, то функция вызывает саму себя (строка 7), так как вслед за первой лексемой строки может оказаться унарный минус.

Во-вторых, если первый символ – знак минус, то он может быть только унарной операцией. А если он предшествует числу, то он становится частью этого числа, как будто бы это была цифра. Рассмотрим следующую схему:



Отметим, что '–' добавляется к '5', чтобы получить лексему '–5'. Поэтому в строке 8 проверяется, что s_0 – знак минус, а строки 12–20 идентичны строкам 40–48.

Наконец, между строками 9 и 37 тоже есть тонкое различие. Если вслед за цифрой идёт символ пропуска, то эта цифра становится лексемой (строки 9, 10). Если же после унарного минуса идут пропуски, то это вовсе не значит, что он будет лексемой. Например, для строки "–6" алгоритм должен вернуть именно лексему '–6', а не две лексемы для каждого непустого символа. Таким образом, первое условие в строке 9 до перехода к строке 10, где создаётся лексема, проверяет, что символ – именно цифра. Наконец, при обработке строк, подобных "–6", рекурсивный вызов в строке 12, игнорируя пропуски, возвращает лексему '6'. Таким образом, код в строке 17 продолжает работу и правильно применяет унарный минус к 6, как будто бы он является цифрой.

9.6.2. Синтаксический анализатор на основе рекурсивного спуска

Создав список лексем, мы готовы обработать его, чтобы вычислить исходное математическое выражение. Конечно, лексемы не могут появляться в случайном порядке (например, в списке лексем не может быть двух подряд операций или чисел, незакрытых скобок и т. д.). Они должны подчиняться порождающим правилам формальной грамматики, описывающим правильный порядок следования лексем. Грубо говоря, порождающие правила нужны для того, чтобы сгруппировать лексемы или разложить список лексем на фрагменты, соответствующие определённым грамматическим категориям, и выстроить их иерархию.

Для нашего калькулятора такими категориями будут «Выражение» (Expression), «Слагаемое» (Term), «Множитель» (Factor) и несколько других, которые мы вкратце опишем. Чтобы понять эту идею, рассмотрим следующее математическое выражение:

$$-(2 + 3) + 2 * 8 / 4 - 3 * (4 + 2) / 6 + 7 - (-(2 + 8) * 3).$$

На рис. 9.15 показано, что оно состоит из пяти слагаемых, связанных бинарными операциями плюс или минус. Затем эти слагаемые могут быть разбиты на множители, связанные операциями '*' или '/'. На рисунке показано только разбиение слагаемого 3 * (4 + 2) / 6 на три множителя. Важно отметить, что второй множитель не является слагаемым исходного выражения, так как он заключён в круглые скобки. Кроме того, первый минус в выражении – унарный и, конечно же, не разделяет слагаемые. Унарный минус может также оказаться между двумя левыми круглыми скобками, как в последнем слагаемом.

Из рисунка видно, что множитель может быть выражением, заключённым в круглые скобки («скобочным выражением»). Это значит, что определения выражения, слагаемого и множителя взаимно-рекурсивны. Обратите внимание, что выражение – это последовательность слагаемых, слагаемое – это ряд множителей, а множителем может быть или число, или другое скобочное выражение. Понятно, что и выражение, и слагаемое, и множитель могут быть просто числами.

Рассмотрим определение выражения с рекурсивной точки зрения. Его «размером» можно считать количество его слагаемых. Его начальное условие выполняется, когда оно состоит из единственного слагаемого. Но если в нём более одного слагаемого, то размер такого рекурсивного объекта можно уменьшить на 1 и определить его как подвыражение плюс/минус слагаемое. Такое рекурсивное определение в точности соответствует порождающим правилам формальной грамматики. Если E обозначает выражение, а T – слагаемое, то порождающие правила для выражения могут быть записаны с использованием следующей нотации:

$$E \rightarrow T \mid E + T \mid E - T, \tag{9.9}$$

где знаки плюс и минус – это лексемы (бинарные операции) входного списка лексем, а вертикальная черта | (читается как «или») используется для краткости записи, чтобы одной строкой можно было определить несколько порождающих правил. Иными словами, (9.9) определяет три порождающих правила: $E \rightarrow T$, $E \rightarrow E + T$ и $E \rightarrow E - T$.

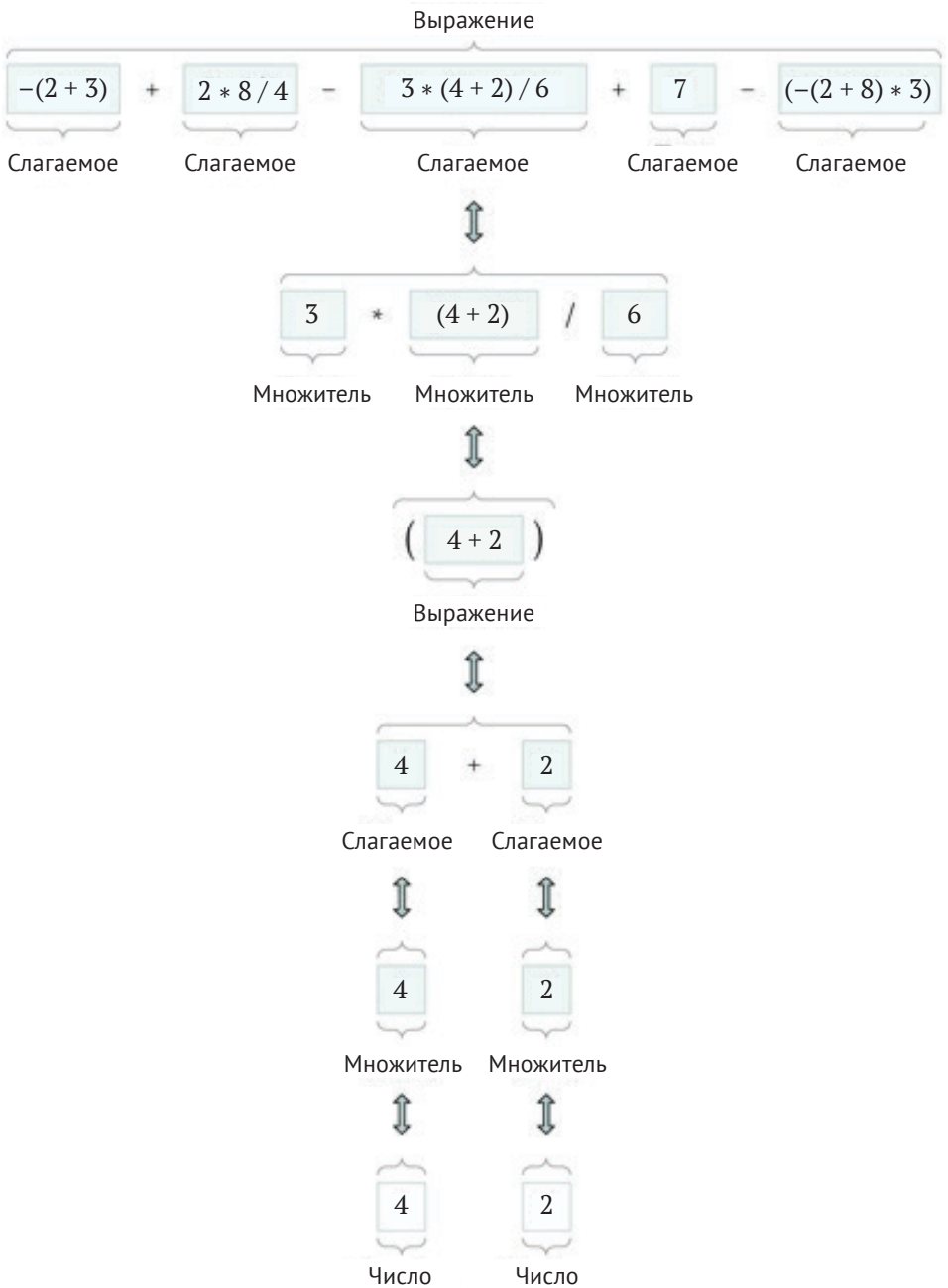


Рис. 9.15. Декомпозиция математической формулы
на категории «Выражение», «Слагаемое», «Множитель» или «Число»

Однако для калькулятора, который мы хотим реализовать, не достаточно лишь одного определения выражения (9.9). В частности, есть ещё одно начальное условие для слагаемого, которое начинается с унарного минуса, продолжается скобочным выражением и заканчивается несколькими множителями. Пусть *I* – это то же самое слагаемое, но с противоположным (после применения унарной операции) знаком. Тогда порождающие правила для E будут такими:

$$E \rightarrow T \mid E + T \mid E - T \mid -I. \tag{9.10}$$

Таким же образом можно построить рекурсивные (или нерекурсивные) определения и для других элементов языка. Пусть *F* – множитель, *P* – скобочное выражение, а *Number* – числовая лексема, тогда полное описание грамматики, которую мы будем использовать:

$$\begin{aligned} E &\rightarrow T \mid E + T \mid E - T \mid -I \\ T &\rightarrow F \mid T * F \mid T / F \\ I &\rightarrow P \mid P * T \mid P / T \\ F &\rightarrow P \mid Number \\ P &\rightarrow (\ E\ ) \end{aligned}$$

где '+', '–', '*', '/', '(' и ')' – возможные входные лексемы. Возможны и другие формальные грамматики, но их разработка выходит далеко за рамки этой книги.

Синтаксический анализатор на основе рекурсивного спуска – это программа проверки списка лексем на соответствие порождающим правилам грамматики, реализованным методами, которые обычно взаимно-рекурсивны. Обычно в таких анализаторах один метод соответствует одному символу[2] в левой части порождающих правил, который предполагает, что входной список лексем имеет тип, определённый этим символом. Для нашего калькулятора мы построим синтаксический анализатор на основе рекурсивного спуска с пятью взаимно-рекурсивными функциями для каждого из символов *E*, *T*, *I*, *F* и *P* (на рис. 9.16 приводятся взаимные вызовы методов). Например, соответствующая *E* функция обрабатывает выражения в предположении, что входной список лексем – именно выражение. В следующих подразделах приводятся возможные реализации этих функций.

9.6.2.1. *Функция expression()*

В листинге 9.9 приведена функция, обрабатывающая выражение (*E*).

[2] Синтаксической переменной. – *Прим. перев.*

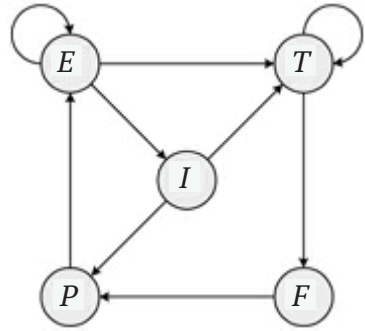


Рис. 9.16. Взаимно-рекурсивные вызовы функций синтаксического анализатора на основе рекурсивного спуска

Листинг 9.9. Синтаксический анализ математического выражения, состоящего из слагаемых

```
def expression(tokens):
    if tokens == []:
       raise SyntaxError('Syntax error')
    else:
       par_counter = 0
       i = len(tokens) - 1
       while i > 0 and ((tokens[i] != '+' and tokens[i] != '-')
                        or par_counter > 0):
           if tokens[i] == ')':
              par_counter = par_counter + 1
           if tokens[i] == '(':
              par_counter = par_counter - 1

           i = i - 1

       if i == 0:
           if tokens[0] == '-':
              return - inverted_term(tokens[1:])
           else:
              return term(tokens)
       else:
           e = expression(tokens[0:i])
           t = term(tokens[i + 1:])
           if tokens[i] == '+':
              return e + t
           else:
              return e - t
```

Сначала все функции синтаксического анализатора на основе рекурсивного спуска проверяют список лексем на пустоту, сообщая при этом о синтаксической ошибке. В противном случае метод выбирает подходящее порождающее правило в предположении, что входной список лексем представляет собой выражение. Варианты выбора приведены на рис. 9.17.

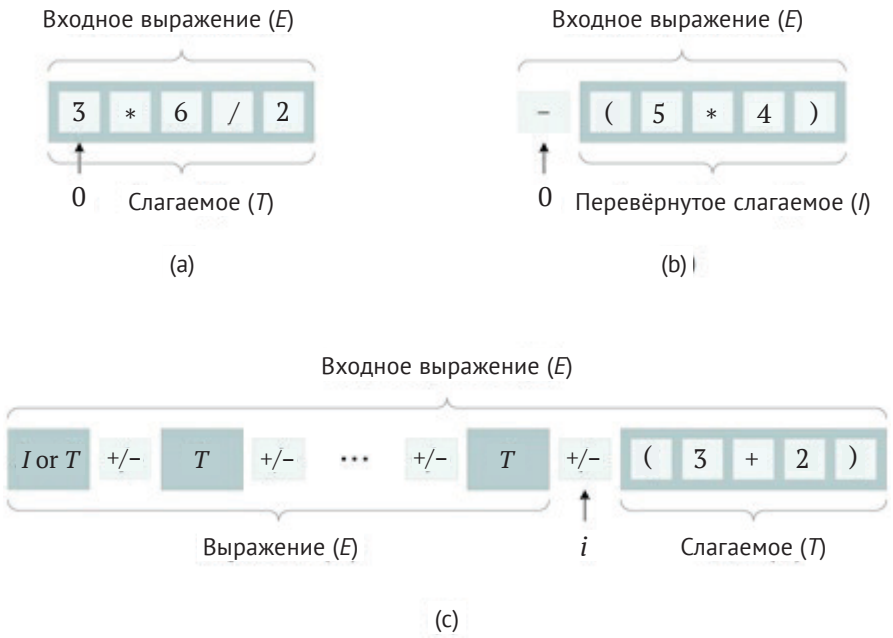


Рис. 9.17. Возможные декомпозиции выражения

С одной стороны, выражение может быть простым (*T*) или «перевёрнутым» (*I*) слагаемым (варианты (a) и (b) на рисунке). С другой стороны, если выражение содержит более одного слагаемого, то оно разбивается на меньшее подвыражение (*E*) плюс/минус слагаемое (*T*), как в случае (c).

Для распознавания этих вариантов функция использует цикл `while` (строки 7–14), который, начиная с конца списка (то есть справа налево), ищет первую бинарную операцию плюс или минус, находящуюся вне любых круглых скобок. Метод подсчитывает левые и правые круглые скобки в переменной-сумматоре `par_counter`. После выхода из цикла переменная `i` (*i* на рис. 9.17(c)) будет содержать либо индекс найденной лексемы '+' или '–', либо 0, если её нет (то есть выражение состоит

из одного слагаемого). В этом случае может быть два варианта. Если первой лексемой входного списка является '–', то это – «перевёрнутое» слагаемое (*I*), и в этом случае функция должна вернуть результат метода `inverted_term(tokens[1:])` с входным списком лексем без первой лексемы '–' (строка 18). В противном случае это будет обычное слагаемое (*T*), и функция может вызвать `term(tokens)` в строке 20. Если же `i` больше 0, то `tokens[i]` будет бинарной операцией ('+' или '–'). В этом случае функция вычисляет выражение слева от этой лексемы, сохраняя результат в `e`, и слагаемое справа от неё, сохраняя результат в `t`. После чего метод в зависимости от операции `token[i]` возвращает сумму или разность `e` и `t`, которые, очевидно, вычисляются взаимно-рекурсивными вызовами функций.

9.6.2.2. *Функция term()*

В листинге 9.10 приведён код метода, который анализирует слагаемое, состоящее из множителей. Он очень похож на код листинга 9.9 тем, что в нём операции '*' и '/' играют ту же роль, что операции '+' и '–' в функции `expression()`. Цикл `while` ищет индекс лексемы-операции над множителями вне круглых скобок. Если этот индекс $i = 0$, слагаемое не содержит лексемы-операции. Следовательно, оно может быть простым множителем (*F*), и функция должна вызвать метод factor со всем входным списком лексем (строка 17). В противном случае алгоритм выполнит бинарную операцию `token[i]` ('*' или '/') над значениями её левого и правого операндов – слагаемого (`t`) и множителя (`f`) соответственно.

9.6.2.3. *Функция inverted_term()*

В листинге 9.11 приведён код, анализирующий «перевёрнутое» слагаемое. Поскольку оно всегда начинается с открывающей круглой скобки '(', то в случае её отсутствия функция выдаст сообщение о синтаксической ошибке. В противном случае функция в цикле ищет индекс (`i`) операции, которая следует непосредственно за закрывающей скобкой, соответствующей открывающей. Если `i` оказалась равной длине входного списка лексем, то всё слагаемое целиком – это выражение в скобках. Следовательно, функция должна вернуть результат вызова метода `parenthesized_expression` со всем входным списком (строка 17). Иначе функция вычисляет выражение в первых круглых скобках (строка 19) и слагаемое из множителей до и после лексемы-операции с индексом *i* (строка 20), а затем применяет к ним эту операцию.

9.6.2.4. *Функция factor()*

В листинге 9.12 приведён код, анализирующий множитель. Если входной список состоит из одной лексемы, то она может быть только числом, и функция возвращает её значение (строка 6). Если же лексема не является числом, то исходное выражение ошибочно. Наконец, если длина входного списка лексем больше 1, множитель может быть только выражением в скобках, и метод должен вызвать соответствующую ему функцию со всем входным списком (строка 10).

Листинг 9.10. Синтаксический анализ слагаемого, состоящего из множителей

```
1   def term(tokens):
2       if tokens == []:
3          raise SyntaxError('Syntax error')
4       else:
5          par_counter = 0
6          i = len(tokens) - 1
7          while i > 0 and ((tokens[i] != '*' and tokens[i] != '/')
8                           or par_counter > 0):
9              if tokens[i] == ')':
10                par_counter = par_counter + 1
11             if tokens[i] == '(':
12                par_counter = par_counter - 1
13
14             i = i - 1
15
16         if i == 0:
17            return factor(tokens)
18         else:
19             t = term(tokens[0:i])
20             f = factor(tokens[i + 1:])
21             if tokens[i] == '*':
22                return t * f
23             else:
24                return t / f
```

Листинг 9.11. Синтаксический анализ слагаемого, состоящего из множителей, первый из которых – скобочное выражение

```
def inverted_term(tokens):
    if tokens == [] or tokens[0] != '(':
       raise SyntaxError('Syntax error')
    else:
       par_counter = 1
       n = len(tokens)
       i = 1
       while i < n and par_counter > 0:
           if tokens[i] == '(':
              par_counter = par_counter + 1
           if tokens[i] == ')':
              par_counter = par_counter - 1

           i = i + 1

       if i == n:
          return parenthesized_expression(tokens)
       else:
           p = parenthesized_expression(tokens[0:i])
           t = term(tokens[i + 1:])
           if tokens[i] == '*':
               return p * t
           else:
               return p / t
```

Листинг 9.12. Синтаксический анализ множителя

```
def factor(tokens):
    if tokens == []:
       raise SyntaxError('Syntax error')
    elif len(tokens) == 1:
        if is_number(tokens[0]):
           return float(tokens[0])
        else:
           raise SyntaxError('Syntax error')
    else:
        return parenthesized_expression(tokens)
```

9.6.2.5. *Функция parenthesized_expression()*

Листинг 9.13 анализирует выражение, заключенное в скобки. Помимо проверки входного списка на пустоту, метод проверяет также наличие открывающей и закрывающей круглых скобок соответственно в начале и в конце выражения. Если ошибок нет, метод продолжает вычисление выражения в круглых скобках (строка 5) и возвращает результат. Таким образом, вызовом функции `expression` этот метод замыкает всю цепь взаимно-рекурсивных функций.

Листинг 9.13. Синтаксический анализ скобочного выражения

```
1  def parenthesized_expression(tokens):
2      if tokens == [] or tokens[0] != '(' or tokens[-1] != ')':
3         raise SyntaxError('Syntax error')
4      else:
5          return expression(tokens[1:-1])
```

И наконец, в листинге 9.14 приводится сам калькулятор в целом.

Листинг 9.14. Основной код программы-калькулятора

```
1  s = input('> ')
2  print(expression(tokenize_unary(s)))
```

9.7. Упражнения

Упражнение 9.1. Мэри и Джон решили сыграть в игру с n камешками. Каждый игрок может удалить один, три или четыре камешка за раз. Выигрывает тот, кто забирает последние камешки. Каждый из игроков может победить, если при его ходе остаётся один, три или четыре камешка. Если же осталось два камешка, то это приводит к проигрышу, так как можно забрать только один из двух камешков. Если камешков больше четырёх, Мэри решила, что будет забирать всякий раз по четыре камешка, если это возможно, тогда как Джон решил забирать по одному камешку. Реализуйте две взаимно-рекурсивные функции, моделирующие игру соперников и возвращающие 1 в случае победы Джона и 0 в случае победы Мэри. Чья стратегия лучше, если $1 \leqslant n \leqslant 100$? Считайте, что начать игру может любой игрок.

Упражнение 9.2. Рассмотрите функции $A(n)$ и $B(n)$, определённые в (3.38) и (3.39). Не решая рекуррентных соотношений, покажите, что $A(n) + B(n)$ соответствует n-му числу Фибоначчи. Иными словами, не используйте (3.40) или (3.41).

Упражнение 9.3. Определите асимптотический порядок роста времени выполнения $A(n)$ и $B(n)$ в формулах (3.38) и (3.39). Иными словами, выведите (3.40) и (3.41).

Упражнение 9.4. Покажите, что $f(n)$, описанная в (9.6), равна $F(2n)$, где F – функция Фибоначчи. Докажите равносильность: (1) математической индукции и (2) решения рекуррентного соотношения в (9.6), получив формулу, подобную (3.37).

Упражнение 9.5. Выразите функции (9.7) и (9.8) через самих себя (без взаимной рекурсии).

Упражнение 9.6. Пусть $A(n)$ и $B(n)$ обозначают количество перемещений дисков, выполненных процедурами из листинга 9.6, соответственно по часовой стрелке и против часовой стрелки для циклической ханойской башни. Приведите рекурсивные формулы для $A(n)$ и $B(n)$ и выразите каждую из них через саму себя без взаимной рекурсии. Затем решите рекуррентные соотношения и укажите порядок их роста.

Упражнение 9.7. Вспомните вариант задачи о «поперечных» ханойских башнях из упражнения 7.4. Разработайте взаимно-рекурсивные процедуры перемещения стопки из n дисков:

1) с левого/правого стержня на средний;
2) со среднего стержня на левый/правый.

Опишите функции, определяющие количество перемещений отдельных дисков при перемещении стопки из n дисков.

Упражнение 9.8. Следующая задача – один из вариантов известной головоломки Spin-out®, изображённой на рис. 9.18.

Изначально у нас есть n фигур (на рисунке их семь), «сидящих» рядком на стержне внутри контейнера с узкой горловиной справа. Фигуры могут находиться в двух положениях: вертикальном – вершиной вверх (a) или горизонтальном – вершиной влево (b). Цель задачи – поворотами фигур привести «запертую» исходную конфигурацию (a) к «незапертой» (b), чтобы извлечь стержень с фигурами через узкую горловину контейнера. Заметьте, что в вертикальном положении фигура не проходит через горловину, не позволяя вынуть и стержень. За один

ход можно повернуть только одну из фигур – из вертикального в горизонтальное положение или наоборот. Более того, их можно поворачивать только в «поворотном круге» (затемнённый круг левее горловины контейнера). На рис. 9.18(c) горизонтально можно повернуть лишь 4-ю фигуру. Кроме того, заметьте, что стенки контейнера не позволяют выдвинуть стержень дальше вправо, так как фигура правее поворотного круга находится в вертикальном положении. Более того, если первую фигуру можно повернуть всегда, остальные можно повернуть только тогда, когда фигура правее неё находится в вертикальном положении, как в конфигурации (c). Таким образом, в конфигурации (d) 4-ю фигуру повернуть уже нельзя.

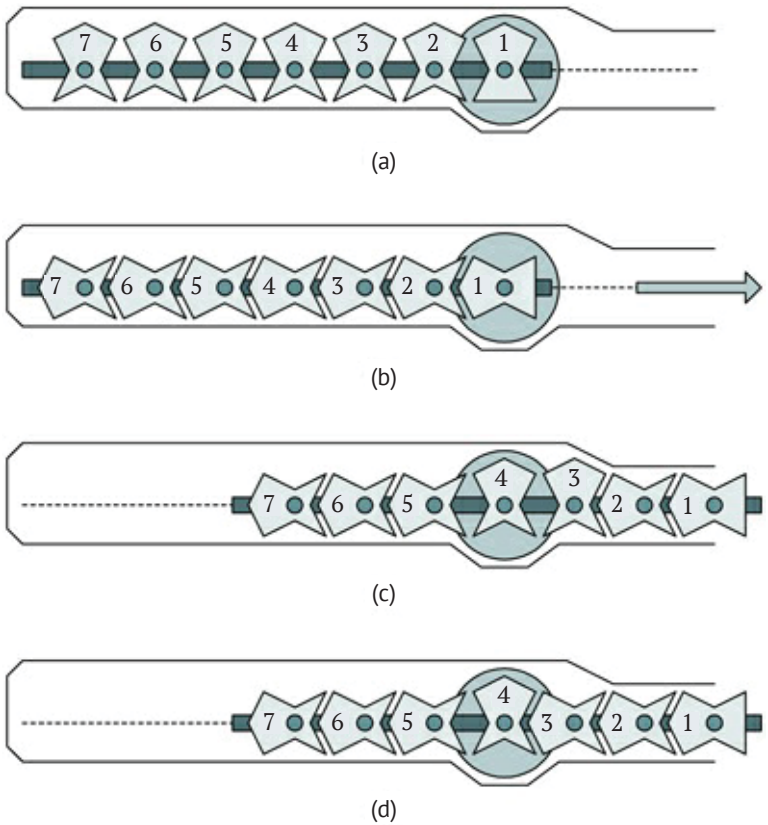


Рис. 9.18. Правила головоломки Spin-out®

Реализуйте две взаимно-рекурсивные процедуры, определяющие последовательность поворотов для *n* фигур для «отпирания» стержня, начиная с конфигурации (a) и заканчивая конфигурацией (b).

Подсказка: задавшись противоположной целью достичь из (b) конфигурации (a), используйте одну процедуру для «отпирания» первых *n* фигур на стержне, а вторую – для их «запирания». Затем определите функцию, определяющую количество поворотов для отпирания стержня с *n* фигурами.

Глава 10

Выполнение программы

На самом деле вы не понимаете того, что понимаете односторонне.

– Марвин Минский

В предыдущих главах на большом количестве примеров разнообразных вычислительных задач и основных типов рекурсии было показано, как нужно мыслить, чтобы создавать рекурсивные программы. Все алгоритмы придерживались высокоуровневой декларативной парадигмы программирования, когда нужно было сосредоточиться на том, *что* вычисляет программа. Эта глава, напротив, представляет низкоуровневые аспекты рекурсивных программ и раскрывает, *как* они работают, вплоть до последовательности выполнения команд (то есть потока управления).

В этой главе процесс выполнения рекурсивных программ рассматривается с разных точек зрения, некоторые из которых являются чисто «умозрительными». В ней вводится понятие дерева рекурсии – распространённое графическое представление последовательности вызовов рекурсивных программ. Среди его достоинств – удобство отладки, помощь в вычислении функций, наглядность оценки вычислительной (временно́й и пространственной) сложности рекурсивных программ, а также покрытие всех рассмотренных ранее разновидностей рекурсии. Кроме этого, в главе описывается тесное взаимодействие рекурсивных программ со стеком – структурой данных, используемой ими при вызовах подпрограмм (методов).

Заканчивается глава кратким введением в мемоизацию – стратегию, которая сокращает временну́ю сложность рекурсивных методов путём

сохранения результатов потенциально затратных вызовов. Мемоизация тесно связана с таким важным методом проектирования алгоритмов, как динамическое программирование.

Независимо от того, рекурсивны или нет выполняемые программой методы, важно понимать правила их вызова. Рассмотрим код из листинга 10.1 для вычисления косинуса угла между двумя n-мерными векторами согласно определениям (3.14), (3.15) и (3.16).

Листинг 10.1. Методы для вычисления косинуса угла между двумя n-мерными векторами

```
import math


def dot_product(u, v):
    s = 0
    for i in range(0, len(u)):
        s = s + u[i] * v[i]
    return s


def norm_Euclidean(v):
    return math.sqrt(dot_product(v, v))


def cosine(u, v):
    return dot_product(u, v) / norm_Euclidean(u) / norm_Euclidean(v)


# Example
a = [2, 1, 1]
b = [1, 0, 1]
print(cosine(a, b))
```

На рис. 10.1 приводится цепочка (нерекурсивных) вызовов подпрограмм (тёмные стрелки) и возвратов из них (светлые стрелки) при выполнении этого кода. Блоки в правых верхних углах функций содержат конкретные параметры, приводящие к конечному результату $\sqrt{3}/2$. Числа в кружочках обозначают последовательность выполнения действий. Очевидно, но крайне важно то, что полное завершение подпрограммы невозможно до полного завершения всех вызвавших её методов.

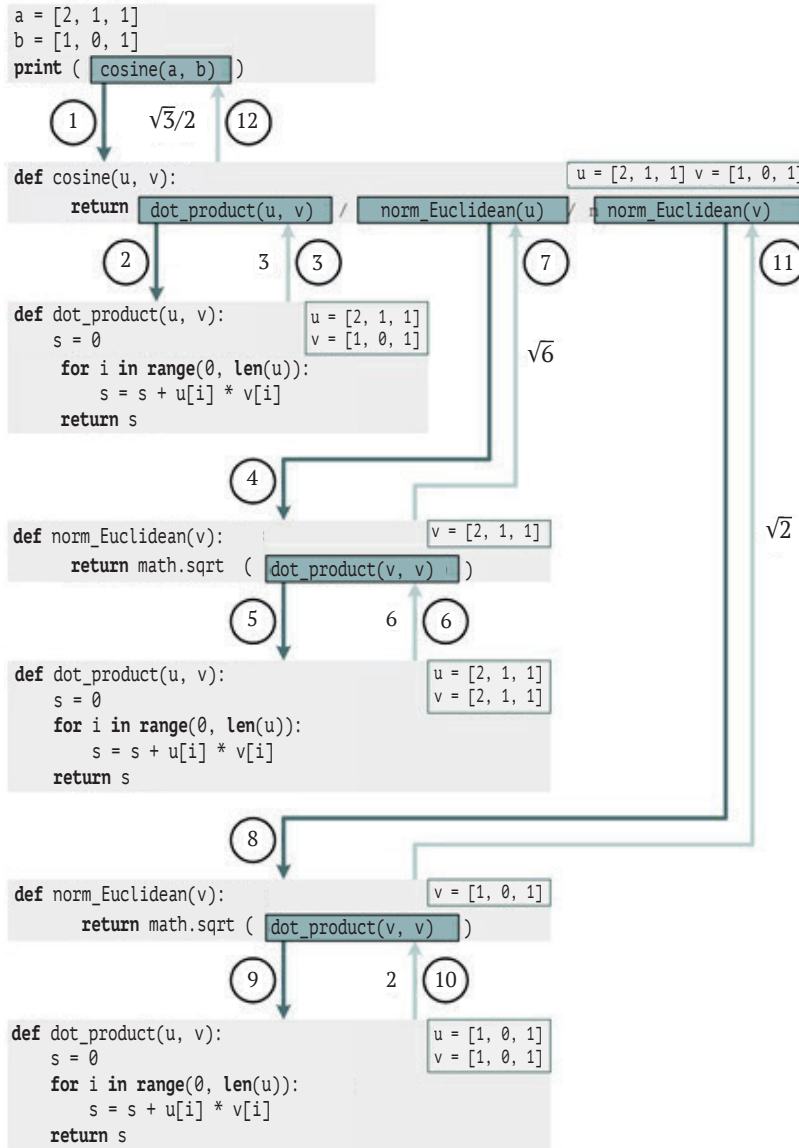


Рис. 10.1. Цепочка вызовов-возвратов методов из листинга 10.1

То же самое – для рекурсивного кода. На рис. 10.2 приведена цепочка рекурсивных вызовов функции `sum_first_naturals` из листинга 1.1 для начального значения $n = 4$. При выполнении рекурсивного условия метод вызывает сам себя (уменьшая исходное значение на 1) и снова переходит к первой своей инструкции. Крайне важно понимать, что процесс

не завершается с выполнением начального условия ($n = 1$). Ведь до этого метод уже вызывал себя несколько раз, поэтому программа должна выполняться до полного завершения каждого из этих рекурсивных вызовов. Непонимание этой последней цепочки действий, известной под названием «пассивный поток» (passive flow), – одна из главных причин искажённого представления о рекурсии. Наконец, вся цепочка действий алгоритма может быть проверена построчно шаг за шагом методами отладчика. Такие инструментальные средства доступны в большинстве интегрированных сред разработки.

10.1. Поток управления между подпрограммами

Код в листинге 10.2 включает две процедуры с входным параметром-строкой, которые не возвращают значений, а просто выводят символы на экран. Они очень похожи и отличаются лишь порядком следования двух своих команд в рекурсивном условии, которое выполняется при непустой входной строке. Первая из них сначала выводит символ, а затем вызывает саму себя, тогда как вторая сначала вызывает себя, а затем выводит символ. Загадочные имена этим методам присвоены для того, чтобы скрыть суть их действий. Читателю предлагается разобраться в этом самостоятельно, прежде чем продолжить чтение.

Листинг 10.2. Похожие рекурсивные методы. Что они делают?

```
def mystery_method_1(s):
    if s:
        print(s[0], end='')
        mystery_method_1(s[1:])


def mystery_method_2(s):
    if s:
        mystery_method_2(s[1:])
        print(s[0], end='')


s = 'Word'
mystery_method_1(s)
print()
mystery_method_2(s)
```

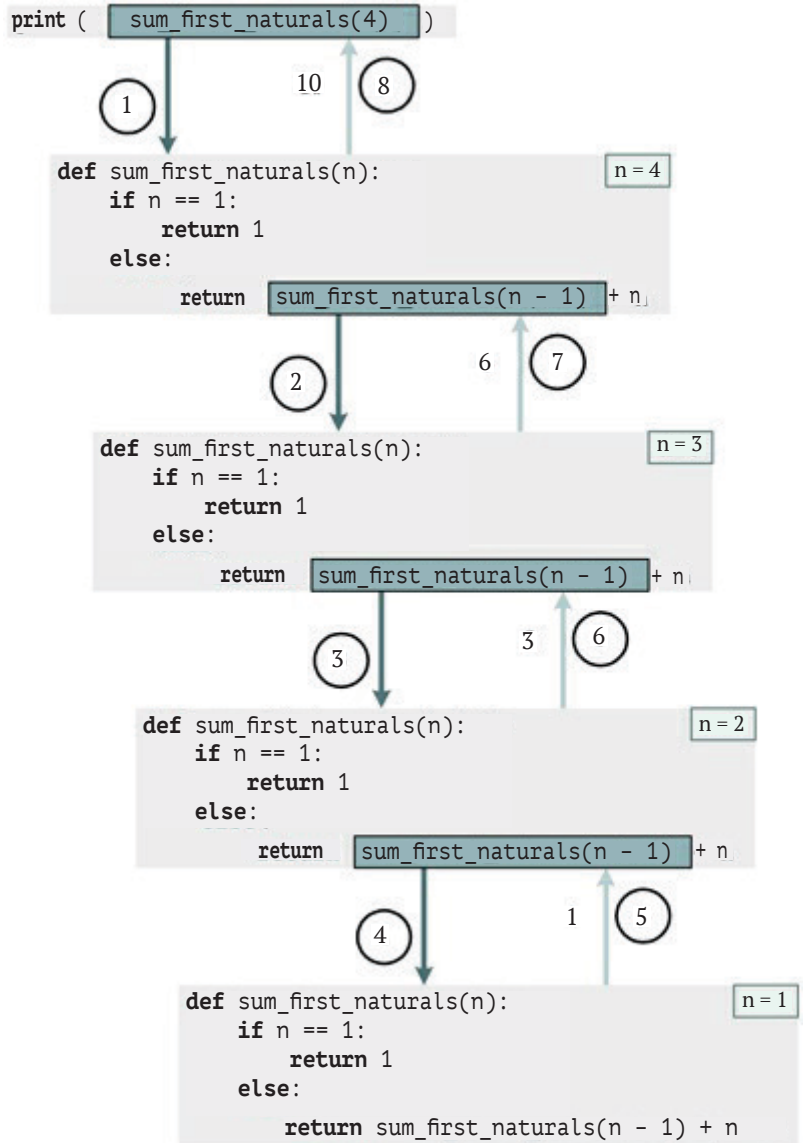


Рис. 10.2. Цепочка вызовов-возвратов для `sum_first_naturals(4)`

Подобно примерам на рис. 10.1 и 10.2, разберём эти процедуры, проследив цепочку выполняемых ими действий. На рис. 10.3 приведена цепочка действий `mystery_method_1` для входной строки `s = 'Word'`, где в полях рядом со стрелками показывается состояние экрана при вызове и по завершении рекурсивной процедуры.

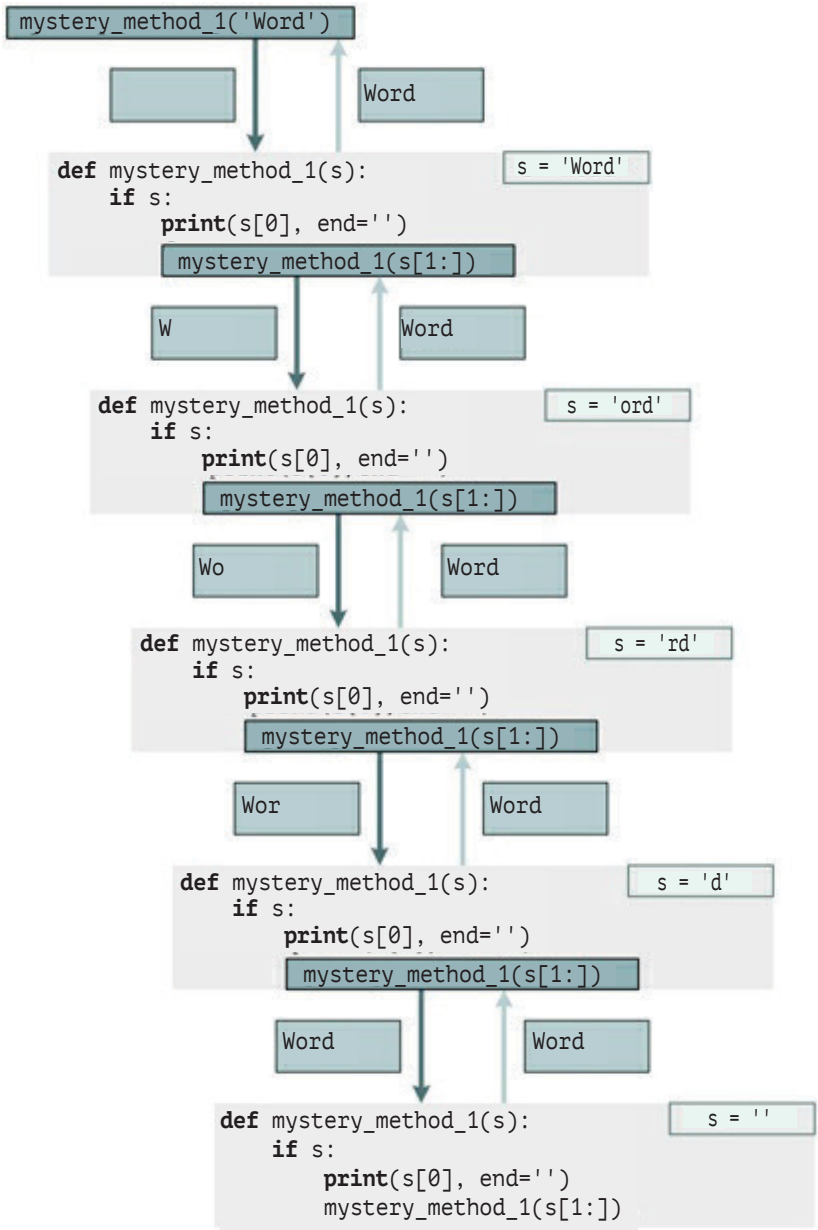


Рис. 10.3. Цепочка вызовов-возвратов для рекурсивной процедуры `mystery_method_1("Word")`

При первом вызове метода экран пуст. Затем, поскольку *s* не пустая строка, выводится её первый символ (`'W'`) и вызывается тот же метод

с остатком строки (`'ord'`). Эти два шага повторяются, пока аргумент метода не станет пустой строкой. Достигнув этого начального условия, программа выведет на экран символ за символом всю исходную строку `'Word'`. После чего метод завершается и возвращает управление предыдущему вызову, который также завершается. Это повторяется, пока не завершится каждый из вызовов. В итоге данная процедура просто выводит на экран исходную строку. В заключение заметьте, что в методе реализована хвостовая рекурсия, так как рекурсивный вызов – это последняя операция в рекурсивном условии. Таким образом, алгоритм решает задачу при выполнении начального условия. Отметьте, что экран не меняется по завершении рекурсивных вызовов.

Напротив, метод `mystery_method_2` тоже выводит на экран исходную строку, но в обратном порядке. На рис. 10.4 приведена цепочка вызовов метода. В данном случае метод – линейно-рекурсивный, что подразумевает выполнение действия (вывода первого символа исходной строки) по завершении рекурсивного вызова.

По достижении начального условия алгоритм ничего не выводит, так как ни одна из команд вывода на экран не выполняется. То есть начальное условие не выполняет никаких действий. Однако по завершении метода управление передаётся первой после рекурсивного вызова инструкции, которая является выводом на экран и, в конце концов, выводит один за другим символы исходной строки, но в обратном порядке.

10.2. Деревья рекурсии

Наиболее простой и наглядный способ представления цепочки рекурсивных вызовов – *дерево рекурсии* (recursion tree). Его узлы соответствуют вызовам метода с определенными параметрами. Если какой-то метод вызывает другие подпрограммы, то его узел становится родителем узлов-вызовов соответствующих подпрограмм. Таким образом, листья дерева соответствуют начальным условиям, а внутренние узлы – рекурсивным условиям. Кроме того, дочерние узлы родителя следуют слева направо в том же порядке, в котором они вызываются. Таким образом, прямой (preorder) обход дерева рекурсии задаёт последовательность вызовов подпрограмм, тогда как обратный (postorder) обход – последовательность возвратов из рекурсивных подпрограмм (см. раздел 7.3).

Как правило, параметры вызова отображаются рядом с соответствующим узлом. На рис. 10.5(а) приведено дерево рекурсии для вызова `sum_first_naturals(4)`, где число в каждом узле – это значение *n* для данного рекурсивного вызова.

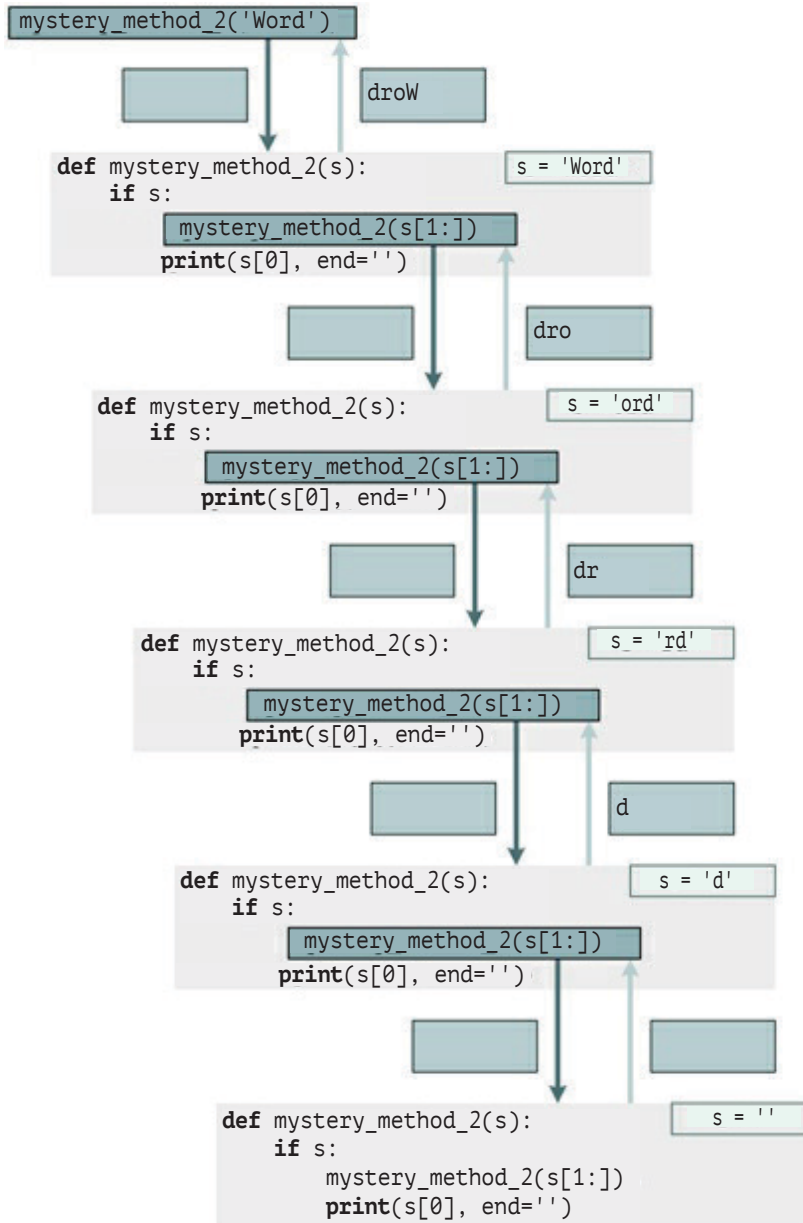


Рис. 10.4. Цепочка вызовов-возвратов для процедуры `mystery_method_2("Word")`

Хотя дерево является неориентированным, важно помнить, что процесс вычисления `sum_first_naturals(4)` сначала спускается вниз по

дереву (рекурсивных вызовов) к начальному условию, а затем поднимается к корню вверх по дереву (рекурсивных возвратов). Поэтому дерево рекурсии можно снабдить стрелками, обозначающими вызовы и возвраты, как показано на рис. 10.5(b). Наконец, по рисунку видно, что `sum_first_naturals(4)` возвращает своё значение методу, вызвавшему её (то есть на экран).

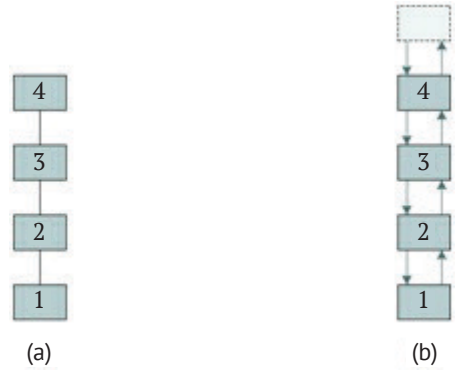


Рис. 10.5. Деревья рекурсии для `sum_first_naturals(4)`

Деревья рекурсии можно дополнить значениями функций по завершении их вызовов. В частности, эти значения можно указать в узлах **дерева активации** (activation tree) вслед за параметрами метода, как показано на рис. 10.6 для вызова `sum_first_naturals(4)`. Деревья активации могут быть полезными для оценки и отладки методов, так как помогают отслеживать возвращаемые значения.

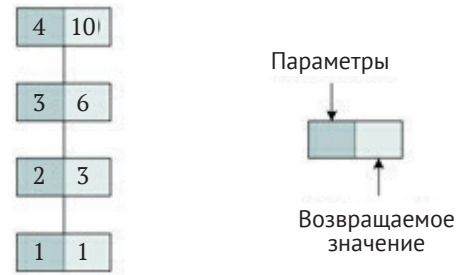


Рис. 10.6. Дерево активации для `sum_first_naturals(4)`

И линейно-рекурсивные методы, и методы с хвостовой рекурсией порождают деревья рекурсии линейной структуры, так как вызывают себя в рекурсивном условии лишь однажды. При этом если линейно-рекурсивная функция при каждом вызове возвращает разные значения,

то функция с хвостовой рекурсией всегда возвращает одно и то же значение. Рассмотрим функцию `gcd1` из листинга для алгоритма Евклида (см. (5.6)). На рис. 10.7 приведено дерево активации для вызова `gcd1(20,24)`, из которого следует, что метод возвращает значение, полученное в начальном условии. Однако вычисление не заканчивается в начальном условии. Наоборот, каждый вызов обязан завершиться, причём с возвратом одного и того же значения (4). Аналогично, в приведённом выше примере (см. рис. 10.3) процедура с хвостовой рекурсией `mystery_method_1` приходит к своему результату по достижении начального условия.

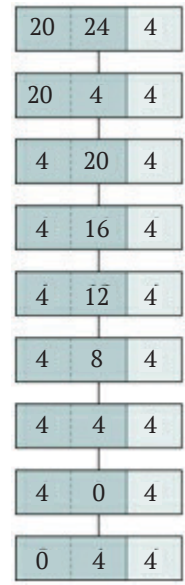


Рис. 10.7. Дерево активации для `gcd1(20,24)`

Рассмотрим ещё одну функцию – функцию Фибоначчи из листинга 1.3. На рис. 10.8(a) приведено её дерево рекурсии для вычисления 6-го числа Фибоначчи. Оно сложнее из-за множественной рекурсии. В частности, оно – двоичное, так как рекурсивное условие содержит два рекурсивных вызова. Дополнительно на рис. 10.8(b) приведено соответствующее дерево активации. Результат метода – количество листьев в этих деревьях, так как по определению метод возвращает 1 в начальном условии и сумму двух вызовов в рекурсивном условии. Таким образом, алгоритм просто подсчитывает, сколько раз он достигает начального условия.

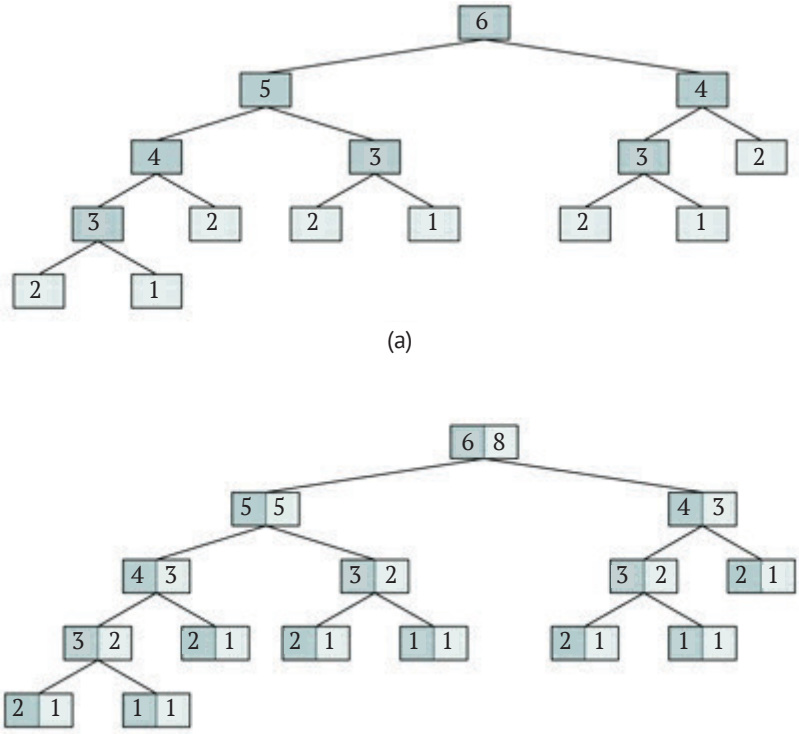


Рис. 10.8. Деревья рекурсии (a) и активации (b) для `fibonacci(6)`

Наконец, если метод – это процедура, не возвращающая значения, то в дерево рекурсии рядом со стрелками, указывающими последовательность вызовов, можно включить дополнительную информацию о действиях этой процедуры. Например, рис. 10.3 и 10.4 похожи на деревья рекурсии (где каждый узел включает ещё и исходный текст), но отображают ещё и состояние экрана до и после вызова (см. упражнение 10.2).

Деревья рекурсии можно генерировать и для других типов рекурсии. Во-первых, взаимно-рекурсивные методы тоже могут порождать деревья рекурсии с линейной структурой. Самый простой пример – методы из листинга 9.1, определяющие чётность неотрицательного целого числа. На рис. 10.9 приведено дерево активации для `is_even(5)`, чей результат – False. Обратите внимание, что методы относятся к хвостовой рекурсии. Таким образом, логическое значение, полученное в начальном условии, просто «всплывает» вверх по дереву в неизменном виде как результат вызовов функций.

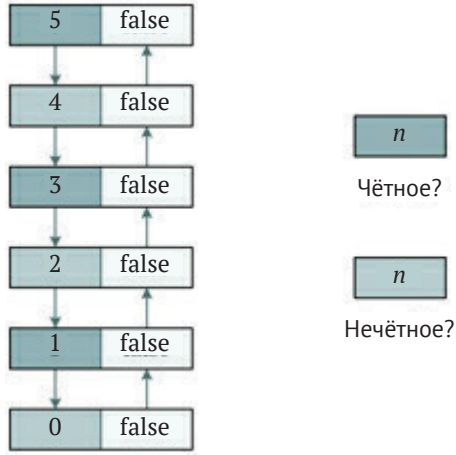


Рис. 10.9. Дерево активации для взаимно-рекурсивных функций из листинга 9.1

Деревья рекурсии могут быть ещё сложнее, если алгоритм включает множественную рекурсию. На рис. 10.10 приводится пример дерева рекурсии для взаимно-рекурсивных функций (1.17) и (1.18), где корневой узел соответствует $A(6)$.

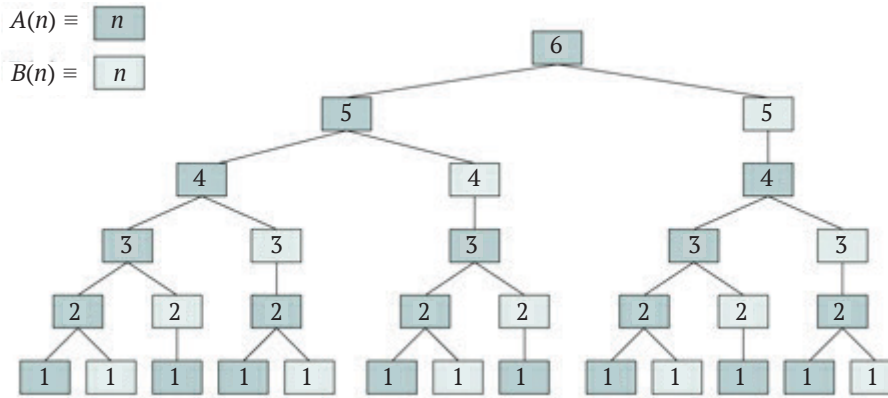


Рис. 10.10. Дерево рекурсии для взаимно-рекурсивных функций (1.17) и (1.18)

Создание дерева рекурсии для вложенных рекурсивных функций ещё сложнее, так как для определения параметров функции необходимо знать результаты всех рекурсивных вызовов. Таким образом, для вложенных рекурсивных функций больше подходят деревья активации, чем обычные деревья рекурсии. На рис. 10.11(a) приведено дерево активации вызова $f(6, 0)$ функции (1.19), а на рис. 10.11(b) – детализация

вложенных рекурсивных вызовов. Левое поддерево отвечает вызову $f(4, 0)$, возвращающему некоторое значение x. Правое поддерево, отвечающее вызову $f(5, 0 + x)$, невозможно детализировать, пока не получено конкретное значение x – результат вычисления $f(4, 0)$ (левое поддерево корневого узла). В данном случае $f(4, 0) = x = 3$, поэтому правое поддерево корневого узла соответствует вызову $f(5, 3)$.

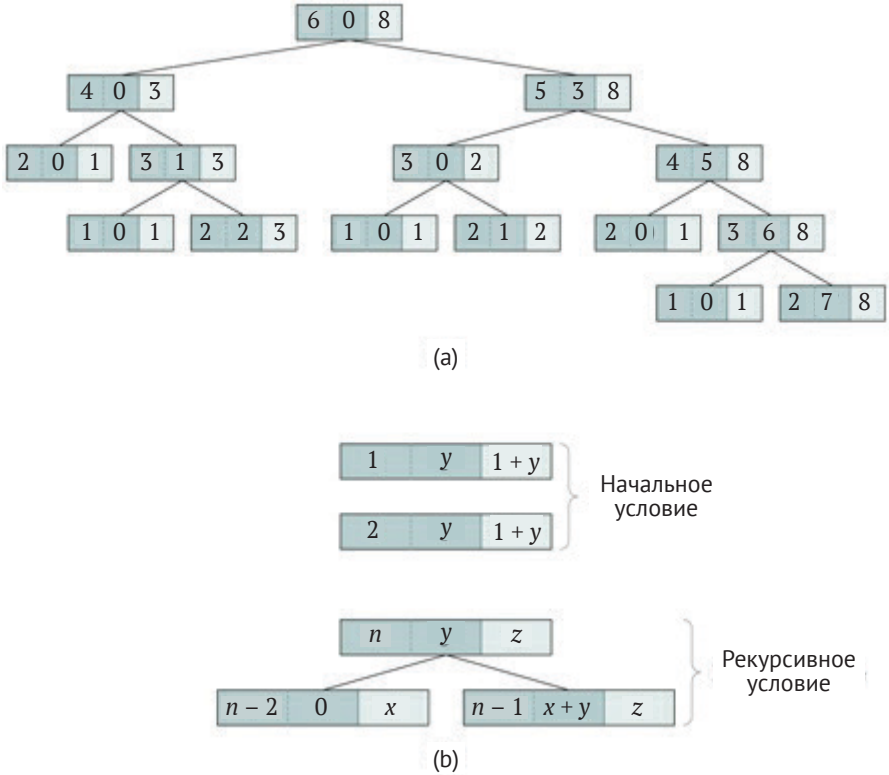


Рис. 10.11. Дерево активации вложенной рекурсивной функции (1.19)

10.2.1. Анализ времени выполнения

Число узлов (то есть рекурсивных вызовов) дерева рекурсии позволяет также оценить вычислительную сложность рекурсивного алгоритма. Если дерево линейно, то понятно, что время выполнения метода прямо пропорционально высоте дерева (стоимость узла умножается на высоту дерева). Но можно также определить количество узлов и для нелинейного дерева. Например, следующая функция определяет количество рекурсивных вызовов в методе Фибоначчи:

$$Q(n) = \begin{cases} 1, & \text{если } n = 1, \\ 1, & \text{если } n = 2, \\ 1 + Q(n-1) + Q(n-2), & \text{если } n \geq 3. \end{cases}$$

Она похожа на саму функцию Фибоначчи. Но, помимо подсчета листьев дерева рекурсии, она добавляет единицу для рекурсивного условия, чтобы учесть также и его внутренние рекурсивные вызовы. В этом случае можно доказать, что $Q(n) = 2F(n) - 1$, где $F(n)$ – n-е число Фибоначчи. Согласно (3.37), количество (приблизительное) вызовов имеет значение порядка $1{,}618^n$. Стоит заметить, что количество узлов полного двоичного дерева высоты n можно оценить как 2^n. Двоичное дерево рекурсии для чисел Фибоначчи содержит несколько меньше узлов, поскольку оно – неполное. Таким образом, вполне объяснимо, что основание соответствующей экспоненты (1,618...) меньше двух.

Кроме того, деревья рекурсии могут использоваться для оценки времени выполнения алгоритмов типа «разделяй и властвуй» с помощью подхода, называемого «метод дерева», который связан с методом расширения и основной теоремой. Идея заключается в формировании другого дерева рекурсии, узлы которого включают стоимость операций, выполняемых в пределах вызова рекурсивного метода, за исключением предстоящих вызовов. На рис. 10.12 приводится такое дерево для рекуррентного соотношения $T(n) = 2T(n/2) + n^2$.

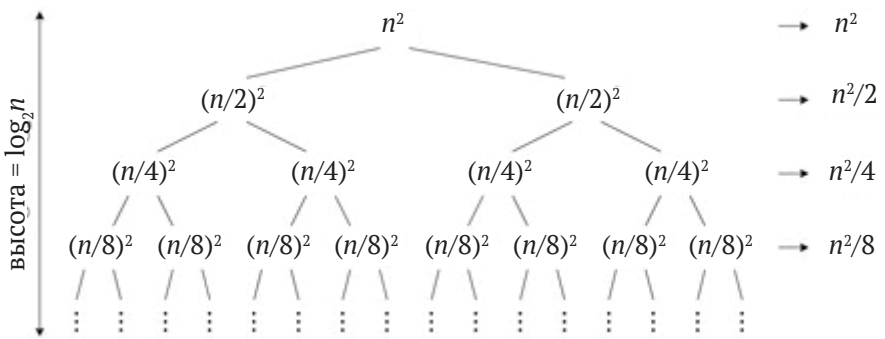


Рис. 10.12. Дерево рекурсии для рекуррентного соотношения $T(n) = 2T(n/2) + n^2$

Первый вызов метода требует n^2 операций плюс стоимость $2T(n/2)$ будущих вызовов. Таким образом, первый узел просто содержит n^2. Узлы второго уровня дерева содержат количество операций, соответствующих $T(n/2)$, то есть $(n/2)^2 = n^2/4$. Поскольку на этом уровне – два узла, общая его стоимость – $n^2/2$. То же рассуждение применимо и для

каждого из последующих уровней, суммарная стоимость которых указана на рисунке справа от соответствующего уровня, а их сумма даст в итоге стоимость алгоритма. Если предположить, что n – степень двух и начальное условие выполняется при $n = 1$, то высота дерева будет $\log_2 n$. В этом примере, предположив, что $T(1) = 1$ (последний уровень будет состоять из n слагаемых, равных 1) – окончательное нерекурсивное выражение, рекуррентное соотношение можно будет выразить так:

$$T(n) = n^2 \sum_{i=0}^{\log_2 n} 1/2^i = n^2(2 - 1/n) = 2n^2 - n.$$

10.3. Программный стек

Рекурсия очень полезна и удобна, когда для реализации алгоритма требуется стек – линейная структура данных, моделирующая кучу объектов, которые могут добавляться или удаляться в строго определённом порядке. А именно: при добавлении элемент помещается на вершину стека, тогда как единственным элементом, который можно удалить из стека, является тот, что находится на вершине стека, как показано на рис. 10.13(a). Поэтому такая структура данных называется магазином или структурой, отвечающей правилу LIFO[1]. Операции добавления и удаления элементов стека, в нашем случае растущего вниз, называются *push* (втолкнуть) и *pop* (вытолкнуть) соответственно. В отличие от стека, родственная ему структура данных *очередь* (queue) отвечает правилу FIFO[2], моделируя очередь ожидания, как показано на рис. 10.13(b). Новый элемент добавляется в конец очереди, а удаляется из неё самый первый элемент. Эти операции обозначаются как *enqueue* (в очередь) и *dequeue* (из очереди) соответственно.

На нижнем уровне исполнения кода вызовы подпрограмм и возвраты из них реализуются в памяти посредством специального стека, именуемого *программным стеком* (program stack), *управляющим стеком* (control stack), стеком вызовов (call stack) или просто стеком. И хотя он скрыт в языках программирования высокого уровня, представлять его устройство чрезвычайно важно для понимания механизма рекурсии.

10.3.1. Стековые кадры

Каждый вызов подпрограммы в первую очередь создает в стеке *стековый кадр* (stack frame), или фрейм, – блок данных, размещаемый на

[1] Last In, First Out – последним вошёл, первым вышел. – *Прим. перев.*

[2] First In, First Out – первым вошёл, первым вышел. – *Прим. перев.*

вершине программного стека. Помимо адреса возврата, он содержит, в частности, параметры и локальные переменные метода или ссылки на них, а также другую низкоуровневую информацию. Адрес возврата указывает на машинную команду, которой метод должен передать управление по завершении своих действий. По завершении метода его запись активации удаляется из программного стека, а освобождаемая память может использоваться для вызовов других методов. Таким образом, в любой момент программный стек содержит информацию только об «активных» подпрограммах, то есть вызванных, но еще не завершённых (стековые кадры называют также активными кадрами, или записями активации).

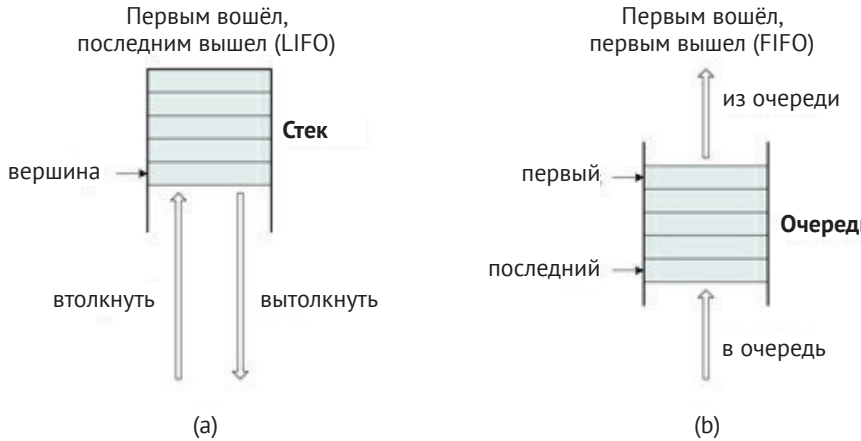


Рис. 10.13. Структуры данных стек и очередь

На рис. 10.14 приведены состояния программного стека при выполнении кода из листинга 10.1, а по существу – последовательность вызовов-возвратов методов на рис. 10.1.

После инициализации списков **a** и **b** в строках 16 и 17 вызов `cosine(u,v)` создаёт в программном стеке первый стековый кадр (тёмный прямоугольник на рисунке), в котором сохраняются ссылки на два входных списка (шаг 1). После этого управление передаётся вызываемому методу, первая операция которого (строка 13) – вычисление скалярного произведения его входных параметров `dot_product(u,v)`. Следовательно, вызов этого нового метода добавляет в стек новый стековый кадр (шаг 2) с выделением памяти для хранения двух ссылок на входные списки и двух локальных переменных `s` и `i`. По завершении этого метода весь его стековый кадр удаляется из программного стека (шаг 3). На следующем шаге в той же строке 13 вычисляется `norm_Euclidean(u)`.

Вызов этой функции помещает в программный стек свой стековый кадр (шаг 4). Метод вычисляет евклидову норму вектора – своего входного списка (в данном случае **a**), вызывая `dot_product(v,v)`, который, в свою очередь, создаёт новый стековый кадр (шаг 5). На рис. 10.15 показаны сохранённые в программном стеке параметры и локальные переменные для вычисления этого скалярного произведения, а также данные двух предшествующих вызовов из приведённого примера кода.

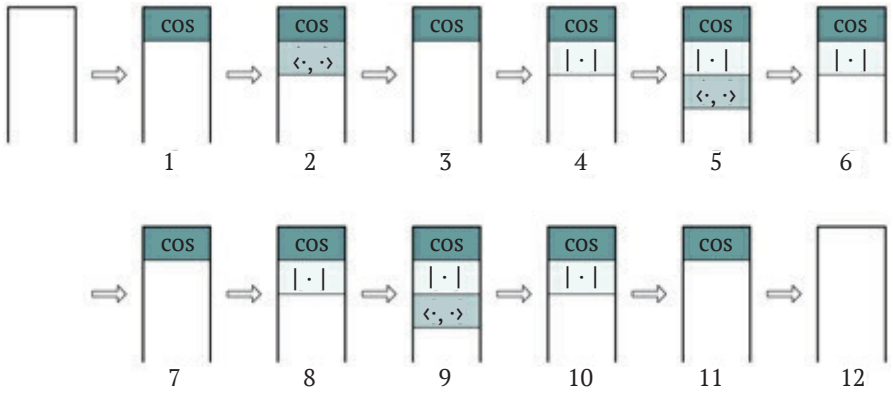


Рис. 10.14. Выделение стековых кадров при выполнении кода из листинга 10.1, где cos, | · | и ‹·, ·› обозначают методы `cosine`, `norm_Euclidean` и `dot_product` соответственно

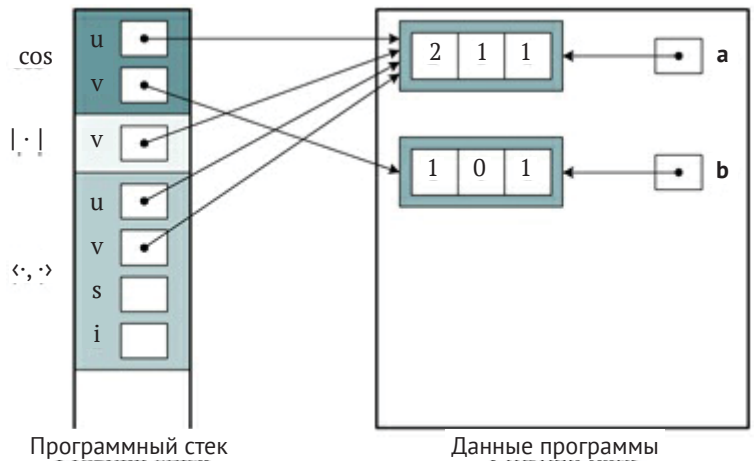


Рис. 10.15. Программный стек и данные на шаге 5 из рис. 10.14

На двух следующих шагах 6 и 7 из стека последовательно удаляются стековые кадры полностью завершённых функций `dot_product` и `norm_Euclidean`. Затем алгоритм вычисляет евклидову норму второго

вектора – входного списка **b**, поэтому шаги 8–11 аналогичны шагам 4–7. В итоге функция `cosine` завершается, возвращая свой результат и освобождая стек программы.

Программный стек – это структура данных, наилучшим образом приспособленная для реализации последовательности вложенных вызовов подпрограмм, включая рекурсию. Рассмотрим рекурсивные вызовы функции `sum_first_naturals` на рис. 10.2. Глядя на него, может показаться, что при каждом вызове функции в памяти создаётся ещё одна копия кода, но это не так: в памяти находится только один экземпляр кода, а весь механизм каждого конкретного вызова и отслеживания цепочек вызовов сосредоточен исключительно в программном стеке. В данном случае каждый новый стековый кадр очередного вызова хранит значение входного параметра *n* для этого вызова. Таким образом, при выполнении кода конкретное значение *n* будет именно тем, что хранится в кадре, расположенном на вершине программного стека. На рис. 10.16 приведено состояние программного стека при выполнении вызова `sum_first_naturals(4)`, где каждый кадр, очевидно, связан с одним и тем же методом.

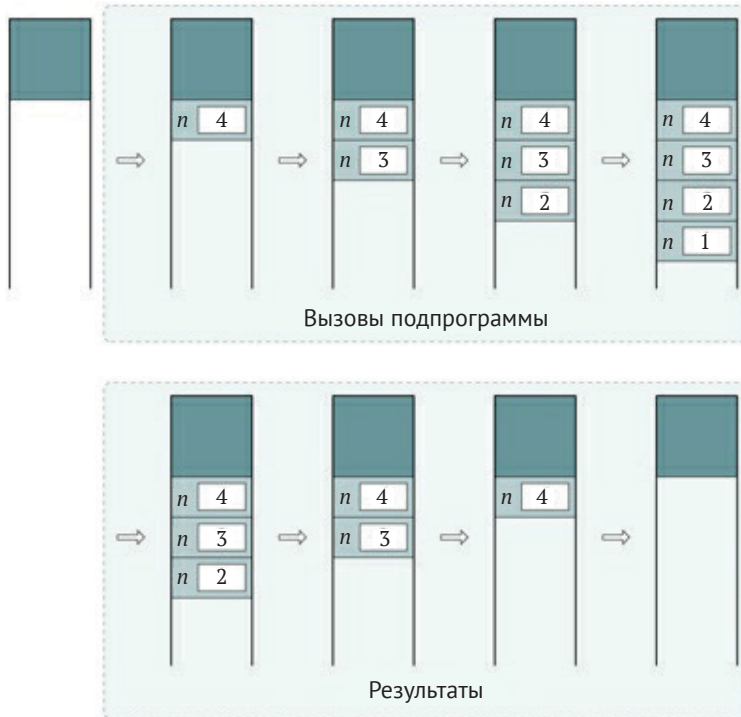


Рис. 10.16. Выделение стековых кадров для `sum_first_naturals(4)`

Выделение стековых кадров отражает цепочку вызовов-возвратов, как в дереве рекурсии (см. рис. 10.5). Затемнённый прямоугольник представляет собой цепочку стековых кадров тех методов, которые могли быть вызваны до вызова `sum_first_naturals(4)`.

10.3.2. Трассировка стека

Ещё одно полезное и важное для отладки понятие – *трассировка стека* (stack trace), или отчёт о состоянии программного стека на определённый момент выполнения программы. Большинство языков программирования обладает функционалом для отображения трассировки стека. Например, в языке Python трассировку стека можно выполнить с помощью модуля `Traceback`. Помимо этого, «трасса» стека печатается при выходе программы на ошибку времени выполнения.

Давайте заменим операцию сложения в строке 6 из листинга 10.1 ошибочным вычитанием `s = s - u[i]*v[i]` (прочие функции тоже будут ошибочными, поскольку и они вызывают `dot_product`). Тогда, например, вычисление евклидовой нормы (длины) отличного от нуля вектора привело бы к ошибке времени выполнения программы, поскольку она попыталась бы извлечь квадратный корень из отрицательного числа. При выполнении такого кода стандартный интерпретатор языка Python выдал бы такое сообщение:

```
Traceback (most recent call last):
    File "traceback_example.py", line 18, in <module>
      print(cosine(a,b))
    File "traceback_example.py", line 13, in cosine
      return dot_product(u, v) / norm_Euclidean(u) / norm_Euclidean(v)
    File "traceback_example.py", line 10, in norm_Euclidean
      return math.sqrt(dot_product(v, v))
ValueError: math domain error
```

Оно говорит о том, что пакет `math` столкнулся с ошибкой (времени выполнения) в строке 10 при выполнении метода `norm_Euclidean`, который был вызван в строке 13 функцией `cosine`. В сообщении также указывается строка 18, в которой вызывается метод `cosine` (в некоторых языках программирования в программный стек помещается стековый кадр основной программы `main`). В заключение отметим, что структура сообщения точно соответствует состоянию программного стека на шаге 5 из рис. 10.14.

10.3.3. Пространственная сложность вычислений

Требования к памяти рекурсивного алгоритма зависят от наибольшего числа, скажем, h стековых кадров, помещенных в программный стек в произвольный момент вычислений. Эту величину можно также понимать как высоту дерева рекурсии. Естественно, для линейных или хвостовых рекурсивных методов h – это просто число узлов дерева рекурсии, поскольку оно линейно. В предположении, что каждый вызов рекурсивного метода требует одинакового объёма памяти M (это не всегда так), полный объём необходимой алгоритму памяти был бы $M \cdot h$, поскольку в какой-то момент в программном стеке окажется h стековых кадров некоторого метода. Например, каждый стековый кадр функции `sum_first_naturals` требует постоянного объёма памяти c (то есть пространства памяти порядка $\Theta(1)$), поскольку он хранит входной параметр и фиксированное количество информации нижнего уровня. Кроме того, так как высота дерева рекурсии – n, то полный объём необходимой памяти – $c \cdot n$, и мы можем заключить, что алгоритм требует пространства памяти порядка $\Theta(n)$.

Для алгоритмов с нелинейным деревом рекурсии важно понимать, что необходимая рекурсивному алгоритму память – это высота дерева рекурсии, а не количество его узлов. Рассмотрим дерево рекурсии для функции `fibonacci` на рис. 10.8(а). Количество его узлов – показательная функция от его высоты n, но его высота – просто $n - 1$, и именно она определяет потребность метода в памяти. Это видно из анализа различных состояний программного стека при выполнении `fibonacci(5)`, как показано на рис. 10.17. Так как по завершении метода стековые кадры удаляются из стека, их максимальное количество равно 4 (шаги 4 и 6). В итоге, поскольку каждый вызов требует одинакового объёма памяти для своих данных, пространственная сложность вычислений для этого алгоритма линейна относительно n.

Так как рекурсия реализуется посредством программного стека, она всегда требует, по крайней мере, h блоков памяти (то есть $\Omega(h)$). В этом соотношении рекурсивные алгоритмы, вообще говоря, используют больше памяти, чем их итерационные версии. Например, основные итерационные алгоритмы вычисления суммы первых положительных целых чисел или чисел Фибоначчи используют постоянный объём памяти, тогда как рекурсивные версии обязательно требуют порядка n блоков памяти. Кроме этого, к дополнительным вычислительным расходам приводят стековые операции вталкивания и выталкивания. Таким образом, среди алгоритмов с одинаковой временно́й сложностью

итерационные, вообще говоря, будут (несколько) эффективнее, чем рекурсивные (см. раздел 11.1).

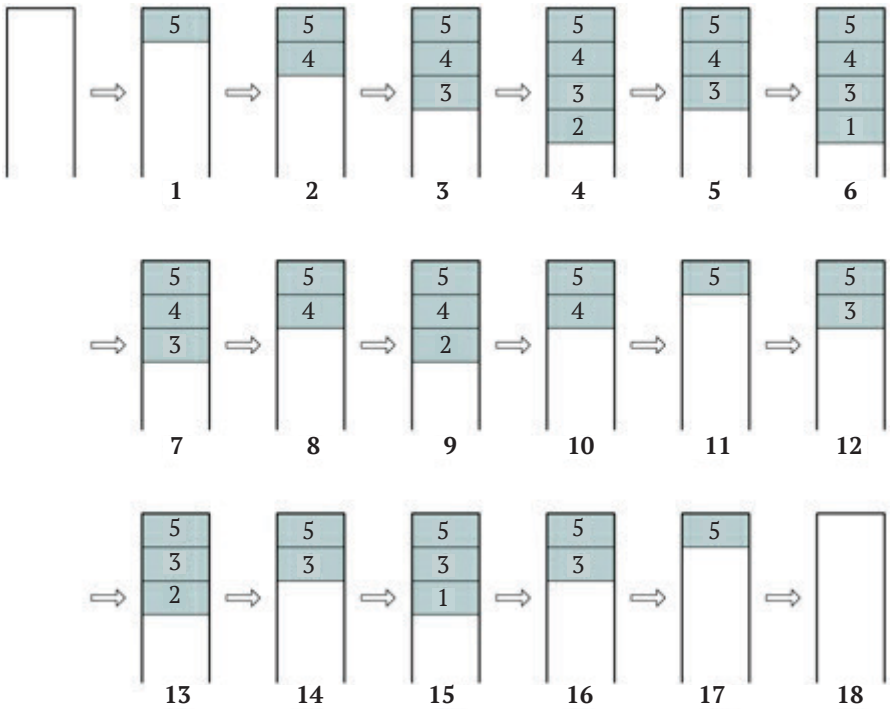


Рис. 10.17. Выделение стековых кадров для `fibonacci(5)`

10.3.4. Ошибки предельной глубины рекурсии и переполнения стека

При выполнении программ существует предел на объем доступной им памяти, зависящий от ряда факторов, включая язык программирования, машинную архитектуру или многопоточность. В частности, существует ограничение на объём данных или количество стековых кадров, сохраняемых в программном стеке. Например, в некоторых реализациях языка Python в программном стеке по умолчанию допускается до 1000 кадров (то есть вложенных рекурсивных вызовов). Конкретную величину, скажем, *M* можно задать вызовом `sys.getrecursionlimit()` так, чтобы максимальная высота дерева рекурсии была *M*. Превысившая этот лимит программа «рушится» с сообщением «RecursionError: превышена предельная глубина рекурсии».

Такой тип ошибки при выполнении программы называют также «переполнением стека». В других языках вместо ограничения числа рекурсивных вызовов ограничивают объём доступной памяти программного стека и «кучи» (области для динамического выделения памяти под данные). Поскольку динамическая память и программный стек разрастаются, программа может исчерпать доступную память, что приведёт к ошибке времени выполнения программы. Таким образом, если динамическая память велика, переполнение стека может произойти даже при небольшом количестве рекурсивных вызовов. В связи с этим программисты должны стремиться уменьшить размер каждого стекового кадра, избегая параметров и локальных переменных большого объёма (например, размещая в динамически распределяемой памяти вместо самых данных указатели (ссылки) на них, как на рис. 10.15).

Наиболее частая причина ошибки переполнения стека – бесконечная рекурсия, возникающая в рекурсивном коде, который не достигает должным образом начального условия, как, например, метод `factorial_no_base_case` из листинга 2.1. Однако даже для правильно закодированных алгоритмов ограничение на количество стековых кадров может иметь серьёзные последствия при использовании рекурсии, особенно для линейных деревьев рекурсии. Например, большинство линейно-рекурсивных методов, использующих списки, не будет работать, если их размер превышает предельную глубину рекурсии (M). Отметим, что при декомпозиции задачи размера n путём его уменьшения на 1 она обычно генерирует n рекурсивных вызовов, пока не достигнет начального условия (будем считать, что это происходит при $n = 1$ или $n = 0$). В таких случаях, если $n > M$, выполнение программы прервётся из-за переполнения стека. В этом – одна из основных проблем использования линейных или хвостовых рекурсивных алгоритмов. В тех случаях, когда размер задачи большой, лучше применять итерационные методы.

10.3.5. Рекурсия как альтернатива стеку

Рекурсия – очевидная альтернатива итерации, когда программисту приходится явно отвечать за управление стеком или подобной ему структурой данных. Рассмотрим задачу поиска файла в файловой системе, которая состоит в том, чтобы по заданным именам файла f и начальной папки F напечатать все пути к файлам с именем f, находящимся в F или в любой из вложенных в неё папках. Например, рассмотрим файлы и папки на рис. 10.18.

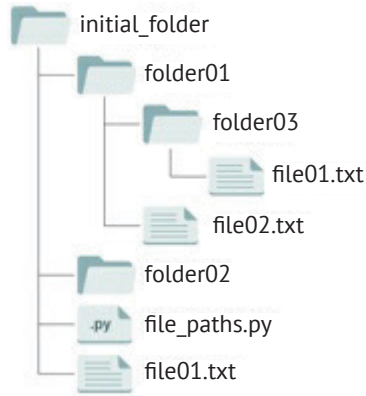


Рис. 10.18. Пример дерева файловой системы

Допустим, что файл `file_paths.py` содержит метод решения этой задачи. Тогда он, будучи вызванным с параметрами 'file01.txt' и начальной папкой '.' (на рисунке обозначена как 'initial_folder'), в которой находится `file_paths.py`, должен вывести что-то вроде:

```
.\file01.txt
.\folder01\folder03\file01.txt
```

Поскольку файловая система имеет древовидную структуру, итерационному алгоритму придётся использовать структуру данных типа стек или очередь для поиска файла *f* в глубину или в ширину. В листинге 10.3 приведена возможная реализация поиска в глубину с использованием стека. Сначала он подключает модуль `os`, чтобы воспользоваться методом `os.scandir`, который позволяет получить описание содержимого конкретной папки. В частности, он возвращает (неупорядоченное) множество объектов, находящихся в данной папке. Оно используется в цикле для проверки (по атрибуту `name`) имён файлов и папок в указанном пути, а также их свойств такими, например, методами, как `is_file` (является ли объект файлом) или `is_dir` (является ли объект папкой). Наконец, атрибут `path` используется для печати пути к найденному в файловой системе файлу.

Определённый нами метод прежде всего объявляет пустой список, который будет представлять собой стековую структуру данных и содержать объекты-папки. После этого он используется в цикле для проверки всех файлов и папок в каталоге, заданном параметром `folder`. Для каждого объекта множества `os.scandir` алгоритм проверяет, является ли он файлом, и печатает его путь при совпадении его имени со значением

параметра file_name. В противном случае процедура посредством метода is_dir определяет, является ли объект папкой. Если это так, то он помещает объект в стек (метод append). После чего алгоритм выполняет те же действия над объектами, находящимися в стеке. Этот процесс повторяется, пока стек не опустеет.

Листинг 10.3. Итерационный алгоритм поиска файла в файловой системе

```
1   import os
2
3
4   def print_path_file_iterative(file_name, folder):
5       stack = []
6       for entry in os.scandir(folder):
7           if entry.is_file() and file_name == entry.name:
8               print(entry.path)
9           elif entry.is_dir():
10              stack.append(entry)
11
12      while len(stack) > 0:
13          entry = stack.pop()
14          for entry in os.scandir(entry.path):
15              if entry.is_file() and file_name == entry.name:
16                  print(entry.path)
17              elif entry.is_dir():
18                  stack.append(entry)
19
20
21  print_path_file_iterative('file01.txt', '.'):
```

На рис. 10.19 приводится несколько состояний стека при выполнении кода из листинга 10.3 для файлов и папок на рис. 10.18.

Сначала стек пуст (шаг 0). Затем в первом цикле алгоритма в стек помещаются два каталога, находящихся непосредственно в начальной папке (шаги 1 и 2), и печатается путь .\file01.txt. Потом метод переходит ко второму циклу (строки 11–17). Сначала он выталкивает каталог folder02 (шаг 3) и проверяет его. Так как он пуст, метод не печатает новых путей и добавляет новые подпапки в стек. На втором шаге цикла while процедура выталкивает folder01 (шаг 4). Эта папка содержит файл, но он не file01.txt. Однако, кроме него, она содержит подпапку, которая помещается на вершину стека (шаг 5). На следующем шаге цикла метод выталкивает folder03 (шаг 6), находит файл под названием file01.txt и печатает его полный путь. При этом он не добавляет новые

объекты в стек, так как `folder03` не содержит подкаталоги. Таким образом, стек становится пустым, и метод завершается. Читатель может проверить алгоритм на больших файловых системах, чтобы убедиться, что метод действительно выполняет поиск в глубину. В этом небольшом примере алгоритм начинает поиск файла с папки `folder02`. Если бы в ней были подпапки, то метод продолжил бы поиск в них, прежде чем искать в `folder01`.

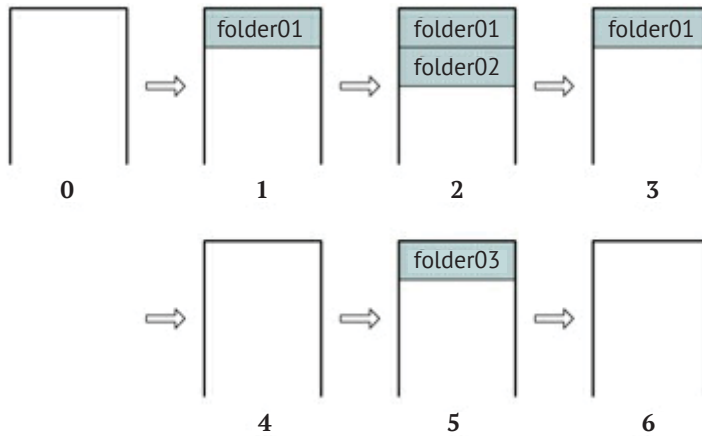


Рис. 10.19. Состояние стека при выполнении итерационного кода из листинга 10.3 для файлов и папок на рис. 10.18

В листинге 10.4 приведёно иное, более простое и естественное, рекурсивное решение задачи. Вся его суть заключается в проверке наличия файла в некоторой папке (без учёта вложенных папок) и во всех вложенных в неё папках. Обратите внимание, что поиск в этих подпапках – это просто меньший экземпляр исходной задачи. Метод тоже использует цикл для проверки каждого объекта данной папки (и потому использует множественную рекурсию). Однако при обнаружении папки он вызывает сам себя для поиска заданного файла в этой папке или в любой из вложенных в неё подпапок.

На рис. 10.20 показано состояние программного стека при выполнении кода из листинга 10.4 для файлов и папок на рис. 10.18.

Имя в стековом кадре – это имя папки при вызове метода (имя файла – одно и то же для каждого вызова), где первый вызов метода использует начальный каталог. Как только метод обнаруживает папку *F*, он снова вызывает сам себя, чтобы напечатать все пути к файлу `file_name` в этой папке и ко всем вложенным в неё папкам. Таким образом, в алгоритме реализован тот же поиск файла в файловой системе в глубину,

но при этом нет явного управления стеком. Наоборот, все действия с программным стеком скрыты от программиста.

Листинг 10.4. Рекурсивный алгоритм поиска файла в файловой системе

```
import os


def print_path_file_recursive(file_name, folder):
    for entry in os.scandir(folder):
        if entry.is_file() and file_name == entry.name:
            print(entry.path)
        elif entry.is_dir():
            print_path_file_recursive(file_name, entry.path)


print_path_file_recursive('file01.txt', '.'):
```

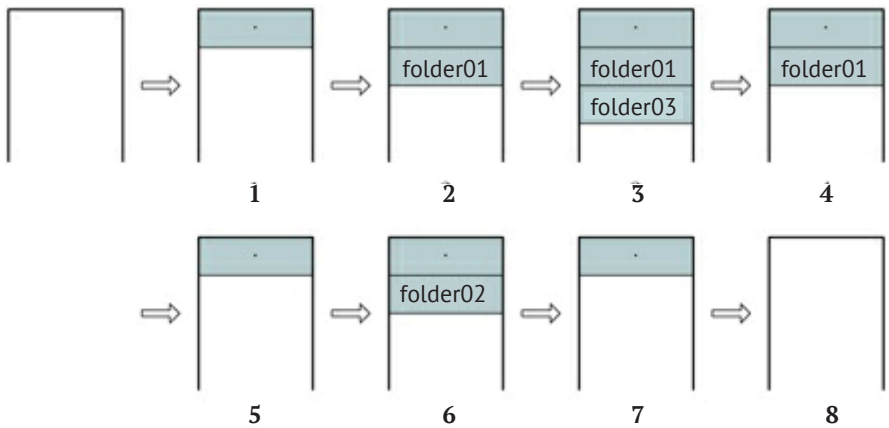


Рис. 10.20. Выделение стековых кадров при выполнении кода из листинга 10.4 для файлов и папок на рис. 10.18

На этом примере мы показали, что для определённых задач рекурсивное решение может быть короче, проще (поскольку не использует стек) и легче для понимания, так как отвечает естественному ходу рассуждений при решении задачи. Кроме того, в этом примере итерационный алгоритм лишь немного эффективнее рекурсивного. Читатели могут оценить время выполнения обоих алгоритмов на своих компьютерах. Наконец, благодаря удобочитаемости и простоте рекурсивные

алгоритмы легче поддерживать и отлаживать. Для определённых задач эти качества могут оказаться весомее временно́й и пространственной эффективности.

Наконец, любая рекурсивная программа может быть преобразована в итерационную (и наоборот). Суть этого преобразования – в явном использовании стека для моделирования действий с программным стеком. Например, он мог бы содержать параметры, передаваемые рекурсивным функциям. Существуют стандартные методы преобразования рекурсивного кода в итерационный, но эта тема выходит далеко за рамки данной книги.

10.4. Мемоизация и динамическое программирование

В этом разделе рассматривается методика, известная как **мемоизация** (memoization), которая используется для существенного ускорения некоторых рекурсивных алгоритмов. Подход связан с **динамическим программированием**, которое является важным и передовым методом проектирования алгоритмов.

10.4.1. Мемоизация

Некоторые рекурсивные алгоритмы несколько раз выполняют одни и те же рекурсивные вызовы (см. раздел 7.4). Другими словами, им приходится неоднократно решать одинаковые *перекрывающиеся подзадачи*. Мемоизация – это подход, позволяющий избежать повторных вычислений при решении таких подзадач и тем самым существенно ускорить алгоритмы. Суть подхода заключается в сохранении результатов рекурсивных вызовов в поисковых таблицах. В частности, до рекурсивного вызова метод проверяет, был ли такой вызов ранее и сохранён ли его результат в таблице. Если это так, то алгоритм вместо повторного рекурсивного вызова просто возвращает его ранее вычисленный и сохранённый в таблице результат. Если же результат не был вычислен, алгоритм продолжает работу, как обычно, вызывая рекурсивный метод.

Одна из самых простых иллюстраций подхода – алгоритм вычисления функции Фибоначчи $F(n) = F(n - 1) + F(n - 2)$. На рис. 10.21 – его дерево рекурсии для $F(6)$.

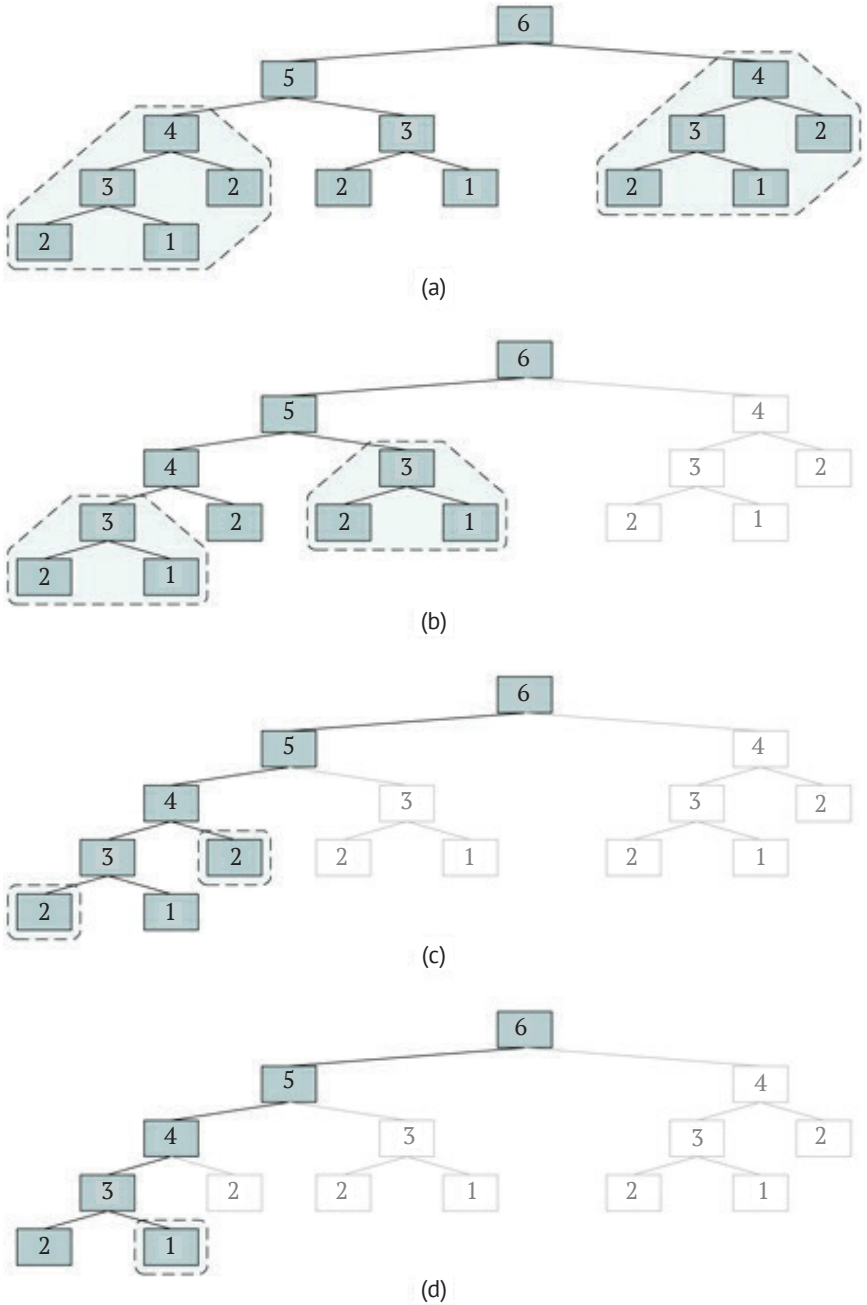


Рис. 10.21. Перекрывающиеся подзадачи при вычислении чисел Фибоначчи по формуле $F(n) = F(n - 1) + F(n - 2)$

На рис. 10.21(a) мы имеем два одинаковых поддерева `fibonacci(4)` (корневой узел соответствует $n = 4$), то есть две перекрывающиеся подзадачи, которые выполняют одно и то же вычисление и возвращают один и тот же результат. Таким образом, по завершении первого вызова `fibonacci(4)` алгоритм может сохранить его результат в поисковой таблице и воспользоваться им впоследствии, чтобы не вызывать метод снова и не выполнять лишних вычислений. Эта стратегия позволяет нам исключить все вызовы, связанные с правым поддеревом дерева рекурсии, как показано на рис. 10.21(b). Более того, усечение дерева рекурсии можно продолжить, если обратить внимание и на другие перекрывающиеся подзадачи. Как видно по рис. 10.21(b), в оставшемся дереве есть ещё два одинаковых поддерева для $n = 3$ (отметим, что всего таких поддеревьев было три). Поэтому можно сохранить результат самого первого вычисления `fibonacci(3)` и при необходимости использовать его значение.

Применение такого подхода ко всему исходному дереву рекурсии приводит к алгоритму с рекурсивным деревом (рис. 10.21(d)), содержащим всего n узлов. Таким образом, за счет увеличения объёма памяти для хранения n промежуточных результатов $F(i)$, где $i = 1, \ldots, n$, новому алгоритму требуется для выполнения линейное время. Увеличение скорости просто поражает, если вспомнить, что первоначальный алгоритм выполнялся за показательное время.

В листинге 10.5 приведён один из способов реализации нового подхода для функции Фибоначчи. Помимо параметра n, метод получает заполненный нулями список (таблицу) поиска a. Его длина – $n + 1$, так как индексы в языке Python начинаются с 0. Метод начинается с проверки начального условия. Поскольку оно всегда возвращает только константу 1, проще проверить начальное условие и вернуть его значение, чем искать его в списке. Таким образом, эта версия алгоритма не сохраняет значения начальных условий в списке поиска. В случае рекурсивного условия, если $a_n > 0$, значение $F(n)$ уже сохранено в a_n (то есть $a_n = F(n)$). Если результат проверки рекурсивного условия – True, метод просто возвращает a_n, не выполняя рекурсивного вызова. В противном случае алгоритм работает, как функция `fibonacci`, но прежде чем вернуть результат, он сохраняет его в a_n. В итоге метод позволяет вычислить числа Фибоначчи за линейное время, но требует дополнительного списка размера n. Однако это не влияет на пространственную сложность вычислений, поскольку функция `fibonacci` тоже требует порядка n блоков памяти в программном стеке, как описано в разделе 1.7.

Листинг 10.5. Рекурсивный алгоритм вычисления чисел Фибоначчи за линейное время с применением мемоизации

```
1   def fib_memoization(n, a):
2       if n == 1 or n == 2:
3           return 1
4       elif a[n] > 0:
5           return a[n]
6       else:
7           a[n] = fib_memoization(n - 1, a)
8                  + fib_memoization(n - 2, a)
9           return a[n]
10
11
12  n = 100
13  print(fib_memoization(n, [0] * (n + 1)))
```

В разделе 7.4 описаны два рекурсивных алгоритма решения задачи о самой длинной подстроке-палиндроме, время выполнения которой – показательная функция от длины исходной строки. Рассмотрим теперь основанное на мемоизации решение, выполняющееся за квадратичное время. В листинге 10.6 приведена версия листинга 7.7, но с мемоизацией. Помимо начальной строки s длины n, метод имеет три дополнительных параметра. Во-первых, он использует нижний (i) и верхний (j) индексы, задающие подстроку поиска самой длинной подстроки-палиндрома. Кроме того, он использует матрицу L размерности $n \times n$ для хранения самых длинных подстрок-палиндромов, найденных методом (конечно, можно реализовать более эффективную версию, которая вместо матрицы сохраняет в любой момент времени только самую длинную подстроку-палиндром, найденную методом). В частности, элемент $L_{i,j}$ будет самой длинной подстрокой-палиндромом, найденной в подстроке $s_{i...j}$, где $i \leqslant j$. Первоначально метод вызывается со строкой s, матрицей L с пустыми элементами, а также верхним и нижним индексами 0 и $n - 1$ соответственно.

Сначала метод проверяет, решалась ли ранее такая задача. В этом случае элемент $L_{i,j}$ будет содержать непустую строку, которую метод может сразу возвратить (строки 2 и 3). Затем проверяется условие $i > j$, возвращающее в случае True пустую строку. В строках 6–8 алгоритм проверяет, состоит ли заданная своими индексами подстрока из единственного символа. В этом случае он сохраняется в таблице L и возвращается в качестве результата. Остальная часть кода относится к рекурсивным условиям. В строках 10–14 метод ищет самую длинную подстроку-палин-

дром в $s_{i+1\ldots j-1}$, но вызывает рекурсивный метод, только если решение еще не было вычислено и сохранено в L. Затем, если подстрока $s_{i+1\ldots j-1}$ является палиндромом (`len(s_aux_1) == j - i - 1`), метод проверяет условие $s_i = s_j$ (строка 16). Если оно выполняется, то $s_{i\ldots j}$ будет самым длинным палиндромом, и метод сохранит и вернёт его (строки 17 и 18). Если же $s_{i\ldots j}$ не является палиндромом, метод выполнит два рекурсивных вызова, аналогичных вызовам в строках 10 и 11 из листинга 7.7, и сохранит их результаты, но только если соответствующие подзадачи не были решены ранее (строки 20–30). В заключение метод возвращает самую длинную из двух строк-палиндромов, соответствующих этим двум подзадачам.

Алгоритм имеет квадратичную сложность вычисления, так как, по сути дела, создаёт двумерную матрицу L и решает подзадачу не более одного раза. Поэтому он гораздо эффективнее метода из листинга 7.7, требующего экспоненциального времени.

10.4.2. Граф зависимости и динамическое программирование

Динамическое программирование – парадигма проектирования алгоритмов, применяемая для решения задач оптимизации, которые можно разбить рекурсивно на меньшие подзадачи так, что разбиение приводит к перекрывающимся подзадачам. Метод может быть нисходящим (сверху вниз) с применением рекурсии и мемоизации. Но более общий подход заключается в применении восходящей (снизу вверх) стратегии. Суть её в том, чтобы реализовать восходящий итерационный алгоритм, который заполняет поисковую таблицу (такую же, как в рекурсивном алгоритме с мемоизацией) снизу вверх, начиная с простейших экземпляров задачи, и последовательно сохраняет получаемые результаты, которые позволят нам решить исходную задачу.

Даже при том, что восходящий подход является итерационным, рекурсивное решение даёт представление о том, как следует заполнять поисковую таблицу. В частности, аналитики могут построить *граф зависимости* (dependency graph), называемый также «графом подзадач», указывающий на зависимости между различными подзадачами (то есть вызовами методов), которые необходимо решить с помощью рекурсивного алгоритма. Безусловно, этот граф – ориентированный, где каждый узел представляет собой конкретную подзадачу, а рёбра – зависимости между ними. Будем считать, что если подзадача A требует решения (зависима от) подзадачи B, то соединяющее их ребро будет направлено от A к B. На рис. 10.22 приведён граф зависимости для рекурсивного

алгоритма на базе рекуррентного соотношения $F(n) = F(n - 1) + F(n - 2)$ для $F(6)$ и начальных условий $F(2)$ и $F(1)$.

Листинг 10.6. Версия листинга 7.7 с мемоизацией

```
def lps_memoization(s, L, i, j):
    if len(L[i][j]) > 0:
        return L[i][j]
    elif i > j:
        return ''
    elif i == j:
        L[i][j] = s[i]
        return s[i]
    else:
        if len(L[i + 1][j - 1]) > 0:
            s_aux_1 = L[i + 1][j - 1]
        else:
            s_aux_1 = lps_memoization(s, L, i + 1, j - 1)
            L[i + 1][j - 1] = s_aux_1

        if len(s_aux_1) == j - i - 1 and s[i] == s[j]:
            L[i][j] = s[i:j + 1]
            return s[i:j + 1]
        else:
            if len(L[i + 1][j]) > 0:
                s_aux_2 = L[i + 1][j]
            else:
                s_aux_2 = lps_memoization(s, L, i + 1, j)
                L[i + 1][j] = s_aux_2

            if len(L[i][j - 1]) > 0:
                s_aux_3 = L[i][j - 1]
            else:
                s_aux_3 = lps_memoization(s, L, i, j - 1)
                L[i][j - 1] = s_aux_3

            if len(s_aux_2) > len(s_aux_3):
                return s_aux_2
            else:
                return s_aux_3

s = 'bcaac'
L = [['' for i in range(len(s))] for j in range(len(s))]
print(lps_memoization(s, L, 0, len(s) - 1))
```

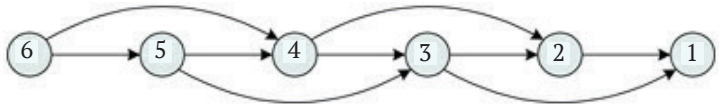


Рис. 10.22. Граф зависимости для $F(n) = F(n - 1) + F(n - 2)$

Обратите внимание на зависимость узлов n от узлов $n - 1$ и $n - 2$. Теперь для реализации решающего задачу итерационного алгоритма граф зависимости нужно проанализировать. Очевидно, что для чисел Фибоначчи алгоритм может получить результат за n шагов при наличии сохранённых решений подзадач $F(1)$, $F(2)$, ..., $F(n - 1)$. Более того, из графа зависимости видно, что нет необходимости в списке длины n для хранения всех промежуточных результатов, поскольку на каждом шаге i нам нужны только два значения – $F(i - 1)$ и $F(i - 2)$, которые можно хранить в двух переменных.

Что касается задачи о самой длинной подстроке-палиндроме, то на рис. 10.23 приводится граф зависимости, относящийся к листингу 10.6, для исходной входной строки из пяти символов.

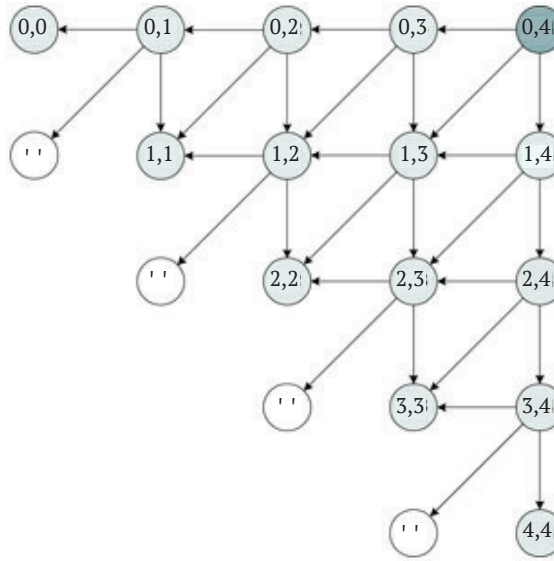


Рис. 10.23. Граф зависимости для листинга 10.6, решающего задачу о самой длинной подстроке-палиндроме

В его узлах указаны только нижний и верхний индексы i и j (если $i > j$, то результатом является пустая строка, а сам узел помечен как ''). Заметьте, что узел (i, j) зависит лишь от решений для пар $(i + 1, j)$, $(i, j - 1)$ и $(i + 1, j - 1)$, что следует из декомпозиции задачи. Каждый узел (i, j) свя-

зан с разными подзадачами, поэтому решение исходной задачи может быть получено только по завершении вызова, связанного с (затемнённым) узлом $(0, n-1)$.

Если промежуточные результаты сохраняются в некоторой матрице L, то из графа зависимости на рис. 10.23 следует, что $L_{i,j}$, связанный с узлом (i, j), может быть получен после вычисления и сохранения $L_{i+1,j}$, $L_{i,j-1}$ и $L_{i+1,j-1}$. Таким образом, граф зависимости задаёт порядок или направление (к верхнему правому углу), в котором нужно решать подзадачи и сохранять их решения. В частности, можно разработать алгоритм, который начинается с решения подзадач, расположенных на главной диагонали, с узлами (i, i) для $i = 0, \ldots, n-1$. Затем на следующей диагонали с узлами $(i, i+1)$ для $i = 0, \ldots, n-2$ и т. д. до достижения узла $(0, n-1)$. В листинге 10.7 приведена возможная итерационная реализация алгоритма.

Листинг 10.7. Поиск самой длинной подстроки-палиндрома в строке *s* с использованием динамического программирования

```
def lps_dynamic_programming(s):
    n = len(s)
    L = [['' for i in range(n)] for j in range(n)]

    for i in range(n):
       L[i][j] = s[i]

    k = 1
    while k < n:
        i = 0
        j = k
        while j < n:
            if (len(L[i + 1][j - 1]) == j - i - 1
                    and s[i] == s[j]):
               L[i][j] = s[i:j + 1]
            elif len(L[i][j - 1]) >= len(L[i + 1][j]):
               L[i][j] = L[i][j - 1]
            else:
               L[i][j] = L[i + 1][j]

            i = i + 1
            j = j + 1

        k = k + 1
    return L[0][n - 1]
```

Заметьте, что он решает подзадачи снизу вверх, начиная с меньших экземпляров, пока не вернёт решение всей исходной задачи. В частности, сначала он заполняет матрицу L пустыми строками (строка 3), а также сохраняет s_i в $L_{i,i}$ для $i = 0, \ldots, n-1$ (строки 5 и 6). Два следующих цикла последовательно, диагональ за диагональю, заполняют матрицу L, пока не достигнут верхнего правого узла графа зависимости. Строки 13–15 проверяют, что подстрока $s_{i..j}$ является палиндромом, и, если это так, сохраняют ее. В противном случае следующие четыре строки сохраняют в $L_{i,j}$ самую длинную из подстрок $L_{i+1,j}$ и $L_{i,j-1}$. На рис. 10.24 приведено состояние матрицы L по завершении метода со всеми решениями подзадач для исходной входной строки s = 'bcaac'. Алгоритм работает за время $\Theta(n^2)$, так как создаёт квадратную матрицу L, каждый элемент которой требует времени $\Theta(1)$ (можно разработать более эффективные алгоритмы, не использующие матрицу).

i \ *j*	0	1	2	3	4
0	'b'	'b'	'b'	'aa'	'caac'
1	' '	'c'	'c'	'aa'	'caac'
2	·	' '	'a'	'aa'	'aa'
3	·	·	' '	'a'	'a'
4	·	·	·	' '	'c'

Рис. 10.24. Матрица L по завершении метода из листинга 10.7 для s = 'bcaac'

10.5. Упражнения

Упражнение 10.1. Догадайтесь, что вычисляет каждая из следующих функций f, подставляя в них различные (неотрицательные целые) значения входных параметров. Кроме того, убедитесь, что найденная вами функция, скажем g, верна, разработав для неё рекурсивный алгоритм с применением декомпозиции к соответствующему описанию f. Рекурсивное определение g должно быть идентично f.

1. $$f(n) = \begin{cases} \text{True}, & \text{если } n = 0, \\ \neg f(n-1), & \text{если } n > 0. \end{cases}$$

2. $$f(n) = \begin{cases} 3, & \text{если } n = 0, \\ n \cdot f(n-1), & \text{если } n > 0. \end{cases}$$

3. $f(n) = \begin{cases} 0, & \text{если } n = 0, \\ f(n-1) + 2n - 1, & \text{если } n > 0. \end{cases}$

4. $f(m,n) = \begin{cases} 0, & \text{если } n = 0 \text{ или } m = 0, \\ f(m-1, n-1) + m + n - 1 & \text{иначе.} \end{cases}$

Упражнение 10.2. Нарисуйте дерево рекурсии, как на рис. 10.5(b), для загадочных методов из листинга 10.2, вызываемых с параметром-строкой 'Word'. Рядом с каждой из стрелок укажите состояние экрана.

Упражнение 10.3. Рассмотрим список кортежей вида (*ключ*, *элемент*). Методы, использующие процедуру `insert_binary_tree` из листинга 5.10, сохраняют их в двоичном дереве поиска в виде четырехкомпонентного списка, как описано в разделе 5.3. Реализуйте различные методы, создающие двоичные деревья поиска, приведённые на рис. 5.2 и 10.25, для исходного списка **a** = [('John', '2006/05/08'), ('Luke', '1976/07/31'), ('Lara', '1987/08/23'), ('Sara', '1995/03/14'), ('Paul', '2000/01/13'), ('Anna', '1999/12/03'), ('Emma', '2002/08/23')].

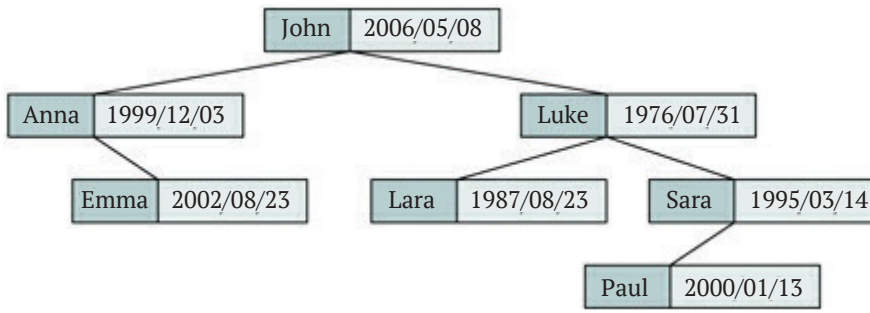


Рис. 10.25. Альтернативное двоичное дерево поиска, хранящее информацию о днях рождения

Упражнение 10.4. Нарисуйте дерево рекурсии для `towers_of_Hanoi(3, 'O', 'D', 'A')` из листинга 7.3. Укажите в узлах все параметры и перемещения дисков.

Упражнение 10.5. Нарисуйте дерево активации вычисления 2^7 для каждого из листингов 4.1, 4.3 и 4.4.

Упражнение 10.6. Рассмотрите функцию синтаксического анализа методом рекурсивного спуска для калькулятора из раздела 9.6.2. Нарисуйте дерево активации для вычисления выражения (5 – 3) * 2 + (–(–7)).

Укажите в нём наименования методов (*E*, *T*, *I*, *F*, *P*) и выразите список лексем в виде математических выражений.

Упражнение 10.7. В листинге 10.8 приведены две функции – сложения и подсчёта цифр неотрицательного целого числа *n*. Одна из них – неправильная. Объясните, в чём ошибка и какое проектное решение привело к ней.

Листинг 10.8. Методы, якобы складывающие и подсчитывающие цифры неотрицательного целого числа. Верны ли они?

```
def add_digits(n):
    if n == 0:
        return 0
    else:
        return add_digits(n // 10) + (n % 10)


def count_digits(n):
    if n == 0:
        return 0
    else:
        return count_digits(n // 10) + 1
```

Упражнение 10.8. Функция в листинге 10.9 подсчитывает количество двойных символов (Например, 'ee', 'oo'. – *Прим. перев.*) в списке длины $n \geqslant 1$. Но в ней есть ошибка. Найдите и устраните её.

Листинг 10.9. Ошибочный код для вычисления количества двойных символов в списке

```
def count_consecutive_pairs(a):
    if len(a)==2:
        return int(a[0]==a[1])
    else:
        return int(a[0]==a[1]) + count_consecutive_pairs(a[1:])
```

Упражнение 10.9. Код в листинге 10.10 вычисляет наименьший простой множитель числа *n*, не меньший *m*. Предусловия: $n \geqslant m$, $n \geqslant 2$ и $m \geqslant 2$. Если он возвращает *n* при $m = 2$, то *n* – простое число. С какой проблемой мы столкнёмся, пытаясь вычислить первые 200 простых чисел?

Листинг 10.10. Вычисление наименьшего простого множителя числа *n*, не меньшего *m*

```
# Preconditions: n >= m, n >= 2, m >= 2
def smallest_prime_factor(n, m):
    if n % m == 0:
        return m
    else:
        return smallest_prime_factor(n, m + 1)
```

Упражнение 10.10. Код в листинге 10.11 предположительно вычисляет нижнюю границу логарифма: $\lfloor \log_b x \rfloor$, где $x \geqslant 1$ – вещественное число, а $b \geqslant 2$ – целое число. Суть его состоит в подсчете количества возможных делений числа x на b. Однако этот код содержит ошибки. Найдите и устраните их, изменив код.

Листинг 10.11. Ошибочный код для вычисления нижней границы логарифма

```
def floor_log(x, b):
    if r == 1:
        return 0
    else:
        return 1 + floor_log(x / b, b)
```

Упражнение 10.11. Рассмотрим список **a** из n чисел. Будем считать, что он содержит «наибольший» элемент, если один из его элементов больше суммы всех остальных. А именно a_i – наибольший элемент, если

$$a_i > \sum_{j=1, j\neq i}^{n} a_j = \sum_{j=1}^{n} a_j - a_i,$$

что равносильно

$$2a_i > \sum_{j=1}^{n} a_j,$$

где в правой части соотношения – сумма элементов списка. В листинге 10.12 приведено возможное решение, но оно неверно. Найдите ошибку и исправьте функцию.

Листинг 10.12. Ошибочный код, проверяющий наличие в списке элемента, большего суммы всех остальных

```
def contains_greatest_element(a):
    if a == []:
        return False
    else:
        return (2 * a[0] > sum(a)
                or contains_greatest_element(a[1:]))
```

Упражнение 10.12. Рассмотрим непустой список **a** из n чисел, которые увеличиваются до определенного индекса i ($0 \leqslant i \leqslant n - 1$), а затем уменьшаются до конца списка. Таким образом, элемент a_i (назовём его «пиковым» и будем считать единственным) будет наибольшим в списке. Код в листинге 10.13 пытается найти этот индекс, но содержит ошибку. Найдите ошибку и исправьте функцию.

Листинг 10.13. Ошибочный код для поиска позиции «пикового» элемента

```
def peak_element(a, lower, upper):
    if lower == upper:
        return lower
    else:
        half = (lower + upper) // 2

        if a[half] > a[half + 1]:
            return peak_element(a, 0, half)
        else:
            return peak_element(a, half, upper)
```

Упражнение 10.13. Рассмотрим вариант кривой Коха, когда линейный отрезок заменяется пятью меньшими отрезками длиной в одну треть исходного, как показано на рис. 10.26.

Рис. 10.26. Вариант разбиения линейного отрезка кривой Коха на пять меньших отрезков

В листинге 10.14 приведена функция, создающая фрактал Коха, основанный на том же преобразовании, что и снежинка Коха (см. раз-

дел 7.5.1), и начинающийся с четырех линейных отрезков, образующих квадрат. Метод должен создать изображение на рис. 10.27 для $n = 4$ и квадрата с вершинами (0, 0), (1, 0), (1, 1) и (0, 1). Однако он этого не делает. Найдите ошибку и исправьте код.

Листинг 10.14. Создание фрактала Коха с помощью преобразования на рис. 10.26

```
import numpy as np
import matplotlib.pyplot as plt


def koch_curve(p, q, n):
    if n == 0:     # The base case is just a line segment
        plt.plot([p[0, 0], q[0, 0], p[0, 0], p[1, 0], q[1, 0], 'k-')
    else:
        v = q - p
        koch_curve(p, p + v / 3, n - 1)
        R_90 = np.matrix([[0, -1], [1, 0]])
        x = p + v / 3 + R_90 * v / 3
        koch_curve(p + v / 3, x, n - 1)
        y = x + v / 3
        koch_curve(x, y, n - 1)
        koch_curve(y, p + 2 * v / 3, n - 1)
        koch_curve(p + 2 * v / 3, q, n - 1)


def koch_square(n):
    p = np.array([0], [0]])
    q = np.array([1], [0]])
    r = np.array([1], [1]])
    s = np.array([0], [1]])

    koch_curve(p, q, n)
    koch_curve(q, r, n)
    koch_curve(r, s, n)
    koch_curve(s, p, n)


fig = plt.figure()
fig.patch.set_facecolor('white')
koch_square(4)
plt.axis('equal')
plt.axis('off')
plt.show()
```

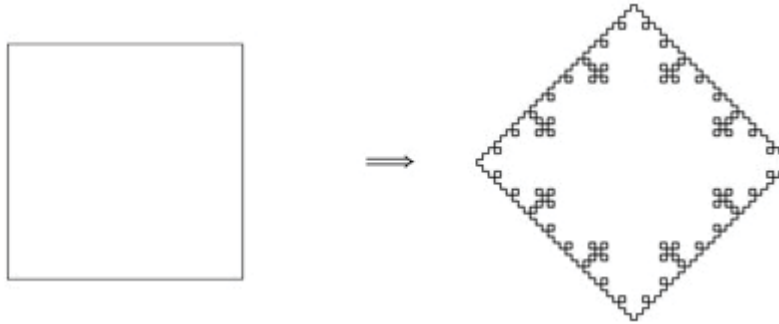


Рис. 10.27. Вариант «квадрата Коха» для $n = 4$

Упражнение 10.14. Реализуйте эффективную версию функции (3.2) для вычисления биномиального коэффициента с мемоизацией, а также нарисуйте соответствующий граф зависимости, скажем, для $n \leqslant 4$.

Упражнение 10.15. В листинге 10.15 приведена функция, вычисляющая длину самой длинной подпоследовательности-палиндрома в списке **a**. Напомним, что в отличие от подстроки, где символы исходной строки должны идти подряд, подпоследовательность не обязательно состоит из подряд идущих элементов списка. Реализуйте эффективную версию алгоритма с мемоизацией, как в листинге 10.6.

Листинг 10.15. Вычисление длины самой длинной подпоследовательности-палиндрома в списке

```
def length_longest_palindrome_subseq(a):
    n = len(a)
    if n <= 1:
        return n
    else:
        if a[0] == a[n - 1]:
            return 2 + length_longest_palindrome_subseq(
                a[1:n - 1])
        else:
            l1 = length_longest_palindrome_subseq(a[1:n])
            l2 = length_longest_palindrome_subseq(a[0:n - 1])
            return max(l1, l2)
```

Глава 11

Вложенная рекурсия и снова хвостовая

Шерлок Холмс: «Всегда есть второе решение задачи, и надо его искать».

– Артур Конан Дойл

Особенность хвостовой рекурсии – её тесная связь с итерацией. Прежде всего её довольно просто преобразовать в аналогичные итерационные версии – более эффективные и не подверженные ошибкам переполнения стека. По этой причине некоторые программисты склонны считать хвостовую рекурсию порочной практикой. Более того, нередко встречаются хвостовые рекурсивные алгоритмы, разработанные с использованием императивного подхода, который ближе к итерации, чем к рекурсии с её декомпозицией и индукцией. В этих случаях итерационные версии явно превосходят рекурсивные. Эта глава исследует взаимосвязь между хвостовой рекурсией и итерацией. Кроме того, она коротко знакомит со вложенной рекурсией (см. (1.19)) и стратегией разработки простых хвостовых рекурсивных алгоритмов, зачастую подобных итерационным, но с использованием декларативного подхода.

11.1. Хвостовая рекурсия и итерация

Итерационные программы могут быть преобразованы в рекурсивные, и наоборот. Более того, связь между итерацией и хвостовой рекурсией особенно очевидна. Этот раздел исследует эту связь и объясняет, почему хвостовой рекурсии обычно предпочитают итерацию.

Вообще говоря, любые рекурсивные алгоритмы могут быть преобразованы в равносильные итерационные версии с использованием стековой структуры данных, исполняющей роль программного стека. Однако для хвостовых рекурсивных методов это преобразование гораздо проще, поскольку оно не нуждается в стеке. Всё дело в том, что хвостовые рекурсивные алгоритмы, в сущности, не обязаны хранить информацию в программном стеке, так как они не нуждаются в стеке при возврате из подпрограммы (метода).

Рассмотрим ещё раз линейно-рекурсивную функцию `sum_first_naturals` из листинга 1.1. Она требует сохранения в программном стеке значений аргументов (см. рис. 10.16), которые для получения конечного результата используются и обрабатываются затем вместе с результатами рекурсивных вызовов. Например, `sum_first_naturals(4)` выталкивает из стека аргумент 4 и добавляет его к результату `sum_first_naturals(4)`, открывая тем самым аргумент `sum_first_naturals(3)`. Напротив, функция `gcd1` в листинге 5.18 получает свой конечный результат после обработки начального условия, так как здесь имеет место хвостовая рекурсия. В частности, на рис. 11.1 приводится программный стек при вызове `gcd1(20,24)` на момент проверки начального условия.

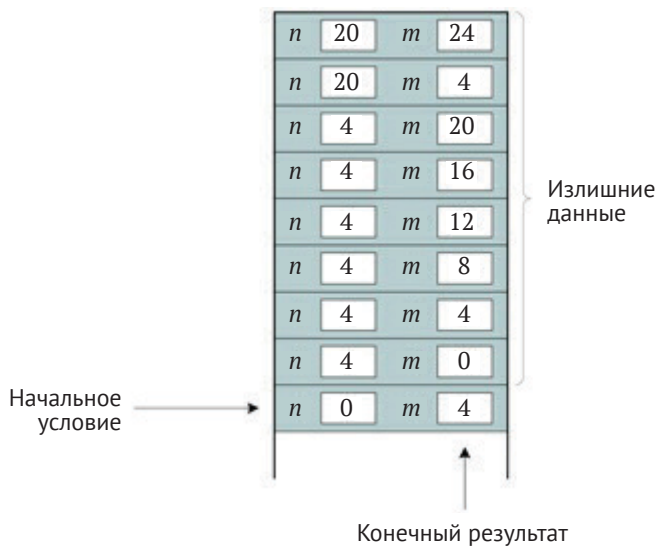


Рис. 11.1. Состояние программного стека при вызове `gcd1(20,24)` на момент проверки начального условия

Этот рисунок похож на дерево активации на рис. 10.7 или на (5.6), где результат метода (4) достигается в начальном условии и без измене-

ний возвращается каждым рекурсивным вызовом. Это значит, что все хранящиеся в программном стеке данные никогда не используются и потому не нужны. Действительно, получив результат в начальном условии, программа просто передаёт его без изменений самому первому рекурсивному вызову, последовательно удаляя каждый стековый кадр по завершении каждого рекурсивного вызова.

Этот пример показывает, что можно создать равносильный итерационный алгоритм вычисления наибольшего общего делителя без использования стека. В частности, функция `gcd1` нуждается в единственном цикле, в котором нужно изменять значения параметров n и m, когда она вызывает саму себя. Другими словами, цикл должен смоделировать изменение параметров при переходе от очередного стекового кадра к следующему, как на рис. 11.1. Следовательно, на каждом шаге цикла, в сущности, должны выполняться те же самые обновления параметров, что выполняются при вызове хвостовой рекурсивной функции (отметим, что рекурсивное условие хвостового рекурсивного метода задаёт лишь изменение параметров от одного вызова до следующего). Наконец, условие выхода из цикла должно быть таким же, как для начального условия.

В листинге 11.1 приведён этот альтернативный итерационный вариант. Начальное условие в `gcd1` достигается при $m = 0$.

Листинг 11.1. Итерационная версия алгоритма Евклида (gcd1)

```
1   def gcd1_iterative(m, n):
2       while m > 0:
3           if m > n:
4                aux = n
5                n = m
6                m = aux
7           else:
8                n = n - m
9
10      return n
```

Таким образом, рекурсивные условия выполняются при $m > 0$. Поэтому мы можем использовать цикл, который выполняется, пока $m > 0$, и который моделирует все выполняемые `gcd1` рекурсивные вызовы вплоть до достижения начального условия. Рекурсивных условий здесь два. Если $m > n$, то рекурсивный вызов просто меняет параметры местами (строки 4–6) с использованием вспомогательной переменной. При

выполнении второго рекурсивного условия $m \leqslant n$ значение n просто меняется на $n - m$ (строка 8). Наконец, когда $m = 0$, метод просто возвращает значение n, согласно начальному условию `gcd1`.

Итерационные аналоги хвостовых рекурсивных алгоритмов обладают двумя преимуществами. С одной стороны, они более эффективны, так как не нуждаются в программном стеке и могут вернуть результат, как только он вычислен. Отметим, что итерационный алгоритм возвращает результат только один раз, тогда как хвостовой рекурсивный возвращает одно и то же значение многократно. С другой стороны, они не приводят к переполнению стека даже для больших значений входных параметров. Если количество рекурсивных вызовов ограничено, то количество итераций цикла (который, в сущности, выполняет те же вычисления, что и рекурсивные вызовы) не ограничено. Для большинства приложений потеря эффективности от применения хвостовых рекурсивных методов не имеет значения. Однако ограничение на количество рекурсивных вызовов – важный фактор масштабируемости (универсальности) рекурсивных алгоритмов, представляющий гораздо более серьезную проблему.

Однако если рекурсивный алгоритм делит размер задачи на два (или на другую константу), ограничение на количество рекурсивных вызовов обычно не представляет проблемы. Для таких алгоритмов высота дерева рекурсии является логарифмической функцией размера задачи. Таким образом, для решения задачи вряд ли потребуется большое количество вложенных рекурсивных вызовов (например, 1000). Если же эффективность алгоритма тоже не важна, то хвостовой рекурсивный алгоритм – вполне приемлемое решение. Рассмотрим, например, хвостовую рекурсивную функцию `bisection` из листинга 5.15, относящуюся к методу деления пополам. Если она используется для вычисления квадратного корня, то количество выполняемых ею рекурсивных вызовов относительно мало. Например, вызов `bisection(0, 4, f, 10**(-10))` требует всего 36 рекурсивных вызовов, чтобы приблизиться к значению $\sqrt{4} = 2$ с точностью до девяти десятичных цифр после запятой. Более того, при таком небольшом количестве рекурсивных вызовов разница во времени выполнения, по сравнению с аналогичным итерационным алгоритмом из листинга 11.2, будет совсем незначительной.

Для моделирования рекурсивных вызовов метод также использует цикл `while`, в условии которого – инвертированное начальное условие рекурсивной функции. Условный оператор `if` различает два сценария, соответствующих двум рекурсивным условиям. В обоих случаях обновляется только один параметр. Наконец, в каждой итерации значение

переменной *z* обновляется для сохранения окончательного результата. Поскольку оба алгоритма практически одинаковы, предпочтение можно отдать тому, который проще читается. В данном случае оба алгоритма просты в понимании той задачи, которую они решают. Однако хвостовые рекурсивные методы больше опираются на декомпозицию задачи.

Листинг 11.2. Итерационная версия метода деления пополам

```
def bisection_iterative(a, b, f, epsilon):
    z = (a + b) / 2

    while f(z) != 0 and b - a > 2 * epsilon:
        if (f(a) > 0 and f(z) < 0) or (f(a) < 0 and f(z) > 0):
            b = z
        else:
             a = z

        z = (a + b) / 2

    return z
```

Наконец, для повышения эффективности многие современные компиляторы в состоянии обнаружить хвостовые рекурсивные методы и автоматически преобразовать их в итерационные версии. К сожалению, в языке Python нет такой возможности, которую часто называют «устранением хвостовой рекурсии» (tail-recursion elimination). Один из аргументов автора языка Python Гуидо ван Россума состоит в том, что это усложняет отладку. Поэтому он рекомендует использовать вместо хвостовой рекурсии итерацию. По мнению автора книги, если эффективность и ограничение на размер программного стека актуальны, то, очевидно, нужно пользоваться итерационными методами вместо хвостовых рекурсивных. Если же они не актуальны, то программистам следует предпочесть реализацию, которая больше соответствует их пониманию решения задачи. Короче говоря, предпочтение отдаётся той версии, которую проще понять и легче сопровождать.

11.2. Итерационный подход к хвостовой рекурсии

Очевидно, что хвостовые рекурсивные алгоритмы – это, как показано в главе 5, результат применения рекурсивного подхода. Но их можно

разрабатывать и с применением итерационного подхода. Этот подход рассматривается в данном разделе на примере задач вычисления факториала и преобразования системы счисления.

11.2.1. Факториал

Рассмотрим итерационную версию функции вычисления факториала из листинга 11.3. Метод вычисляет искомое значение, последовательно повторяя одни и те же действия в цикле `while`. В этом алгоритме начальный параметр *n* задаёт количество итераций цикла, тогда как переменную *p* можно считать накопителем промежуточных результатов, которая по завершении цикла будет содержать искомый факториал.

Листинг 11.3. Итерационная функция вычисления факториала

```
1  def factorial_iterative(n):
2      p = 1
3      while n > 0:
4          p = p * n
5          n = n - 1
6
7      return p
```

В табл. 11.1 приведены значения переменных и параметров метода или состояние памяти программы для каждой итерации цикла при вычислении 4! (иными словами, каждый вход таблицы отображает значения *n* и *p* при проверке программой условия в строке 3).

Таблица 11.1. Состояния программы при вычислении факториала 4! итерационной функцией

n	*p*
4	1
3	4
2	12
1	24
0	**24**

Для вычисления факториала можно написать равнозначную хвостовую рекурсивную функцию, которая выполняет те же действия, что итерационный алгоритм. А именно: метод (скажем, *f*) требует обоих пара-

метров n и p и должен вызывать функцию со значениями параметров из табл. 11.1. Другими словами, хвостовой рекурсивный метод должен генерировать следующие рекурсивные вызовы:

$$f(4, 1) \rightarrow f(3, 4) \rightarrow f(2, 12) \rightarrow f(1, 24) \rightarrow f(0, 24) = 24.$$

Фактически они задают соответствие между рекурсивным хвостовым вызовом и шагом итерационного алгоритма. В частности, очевидно рекурсивное правило $f(n, p) = f(n - 1, p \cdot n)$, выполняющее, в сущности, все те действия по обновлению переменных, которые производятся в теле цикла итерационной версии. Отметим, что n задаёт количество вызовов функции, а p хранит промежуточные результаты, включая окончательный. Последний вызов соответствует начальному условию (при $n = 0$), когда метод возвращает свой второй параметр. Наконец, факториал n вычисляется при вызове $f(n, 1)$, где параметры n и p достигают начальных (до входа в цикл) значений одноимённых переменных итерационной версии. В листинге 11.4 приведена возможная реализация хвостовой рекурсии, где функция-оболочка вызывает рекурсивный метод со вторым аргументом, равным единице.

Листинг 11.4. Хвостовая рекурсивная функция вычисления факториала и функция-оболочка

```
1  def factorial_tail(n, p):
2          if n == 0:
3               return p
4          else:
5               return factorial_tail(n - 1, p * n)
6
7
8  def factorial_tail_recursive_wrapper(n):
9      return factorial_tail(n, 1)
```

Итерационные и хвостовые рекурсивные функции очень похожи. Рисунок 11.2 показывает сходство обоих методов (рекурсивное условие проверяется до начального, чтобы использовать условие $n > 0$ в рекурсивном методе).

11.2.2. Приведение десятичного числа к другому основанию

Итерационный подход может быть применён для создания рекурсивных хвостовых алгоритмов для многих задач. Рассмотрим более

сложную задачу преобразования системы счисления из раздела 4.2.2. Представим себе итерационный алгоритм, выполняющий шаги по приведению десятичного числа 142 к основанию 5 (1032_5) согласно табл. 11.2.

Таблица 11.2. Состояния итерационной программы приведения числа 142 к основанию 5 (1032_5) из листинга 11.5

n	*b*	*p*	*s*
142	5	1	0
28	5	10	2
5	5	100	32
1	5	1000	32
0	**5**	**10000**	**1032**

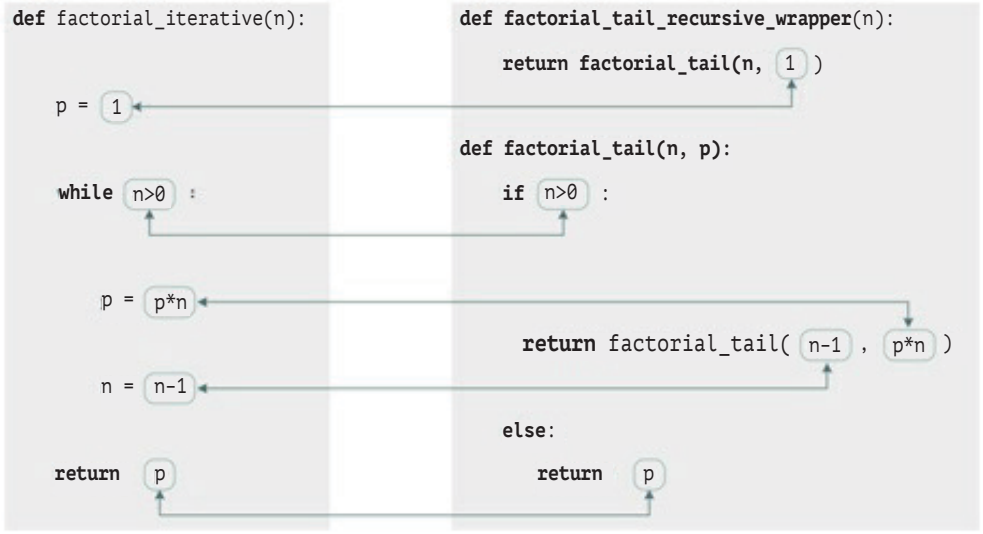


Рис. 11.2. Сходство между итерационным и хвостовым рекурсивным кодами, вычисляющими факториал

Помимо параметров, методу необходима переменная *s*, в которой будет храниться промежуточный результат, а в итоге – результат. Кроме того, дополнительная переменная *p* содержит степень 10, необходимую для обновления значения *s*. На каждом шаге *n* изменяется путём целочисленного деления на основание *b* (*n* // *b*), *p* умножается на 10, а пере-

менная-накопитель *s* увеличивается на $(n \% b) \cdot p$. Этот итерационный алгоритм приведён в листинге 11.5, где изначально $p = 1$ и $s = 0$. Наконец, когда *n* достигает значения 0, метод может возвратить результат, находящийся в *s*.

Листинг 11.5. Итерационный алгоритм приведения неотрицательного целого десятичного числа *n* к основанию *b*

```
1  def decimal_to_base_iterative(n, b):
2      s = 0
3      p = 1
4      while n > 0:
5          s = s + (n % b) * p
6          p = p * 10
7          n = n // b
8      return s
```

Аналогичная хвостовая рекурсивная функция должна моделировать итерации цикла вызовами самой себя. Переменные (и параметры) итерационной версии должны стать параметрами рекурсивных вызовов, а сама рекурсивная функция должна воспроизвести их обновления в теле итерационного цикла (строки 5–7). Наконец, начальное условие выполняется при $n = 0$, когда метод может вернуть значение переменной *s*, так как она будет содержать окончательный результат. В листинге 11.6 приведена хвостовая рекурсивная функция вместе с методом-оболочкой, который инициализирует *p* единицей, а *s* – нулём. Обратите внимание на то, как аргументы рекурсивного вызова отражают изменение переменных итерационного алгоритма в теле цикла.

Листинг 11.6. Хвостовая рекурсивная функция приведения числа к основанию *b* и метод-оболочка

```
1  def decimal_to_base_tail(n, b, p, s):
2      if n == 0:
3          return s
4      else:
5          return decimal_to_base_tail(n // b, b, p * 10,
6                                      s + (n % b) * p)
7
8
9  def decimal_to_base_tail_wrapper(n, b):
10     return decimal_to_base_tail(n, b, 1, 0)
```

Хотя хвостовые рекурсивные функции в листингах 11.4 и 11.6, вне всяких сомнений, рекурсивны, такой способ их разработки соответствует парадигме императивного программирования, когда всё внимание сосредоточено на переменных и параметрах, определяющих состояние программы. Обратите внимание, что при разработке рекурсивного метода данный подход не опирается на декомпозицию задачи или индукцию. Вместо этого рекурсивные алгоритмы просто воспроизводят итерационные версии. По этой причине получаемые алгоритмы могут сбить с толку тех программистов, которые попытаются понять код с точки зрения декомпозиции задачи. Кроме того, из-за упомянутых в разделе 1.1 недостатков такой способ разработки, безусловно, следует избегать. (Для листингов 11.4 и 11.6 ошибки переполнения стека довольно редки, так как исходное значение факториала обычно небольшое, а высота дерева рекурсии для алгоритма преобразования к другому основанию является логарифмической функцией от *n*.) Таким образом, если к решению задачи применяется итерационный подход, то целесообразнее разрабатывать итерационный алгоритм.

11.3. Вложенная рекурсия

Вложенная рекурсия – довольно редкий тип рекурсии, когда рекурсивная функция вызывается в аргументе рекурсивного метода. Подобно хвостовым рекурсивным функциям во многих функциях с вложенной рекурсией (в частности, во всех, представленных в этой книге), рекурсивный вызов – это последнее действие, выполняемое алгоритмом. Тем не менее методы с вложенной рекурсией обязательно должны вызывать себя несколько раз хотя бы в одном из рекурсивных условий. Следовательно, их нельзя отнести к хвостовой рекурсии. В этом разделе описываются две известные функции с вложенной рекурсией и одна задача, которая решается таким же образом. В разделе 11.4 приводятся дополнительные примеры.

11.3.1. Функция Аккермана

Один из самых популярных примеров вложенной рекурсии – функция Аккермана, определяемая следующим образом:

$$A(m,n) = \begin{cases} n+1, & \text{если } m = 0, \\ A(m-1,1), & \text{если } m > 0 \text{ и } n = 0, \\ A(m-1,A(m,n-1)), & \text{если } m > 0 \text{ и } n > 0, \end{cases} \qquad (11.1)$$

где второй аргумент в последнем рекурсивном условии – рекурсивный вызов. Функция растёт чрезвычайно быстро даже при малых значениях аргументов и используется для проверки способности компиляторов оптимизировать рекурсию. В листинге 11.7 приводится код функции, который совсем нетрудно реализовать.

Листинг 11.7. Функция Аккермана

```
1  def ack(m, n):
2      if n == 0:
3          return n + 1
4      elif n == 0:
5          return ack(m - 1, 1)
6      else:
7          return ack(m - 1, ack(m , n - 1))
```

11.3.2. Функция-91 Маккарти

Ещё одна известная функция с вложенной рекурсией – «загадочная» функция-91 Маккарти, определяемая следующим образом:

$$f(n)=\begin{cases} n-10, & \text{если} \quad n>100, \\ f(f(n+11)), & \text{если} \quad n\le 100, \end{cases}$$

где n – положительное целое число. Загадочность её в том, что совсем не просто представить её поведение при $n \leqslant 100$. На рис. 11.3 приводится график этой функции для первых 110 положительных целых чисел. Поэтому функцию можно переопределить как

$$f(n)=\begin{cases} n-10, & \text{если} \quad n>100, \\ 91, & \text{если} \quad n\le 100. \end{cases}$$

11.3.3. Цифровой корень

Цифровой корень неотрицательного целого числа n, обозначаемый здесь как $d(n)$, вычисляется следующим образом. Сначала вычисляется сумма цифр исходного числа n, затем – сумма цифр полученной суммы и так далее до тех пор, пока очередная сумма не станет однозначным числом. Например, $d(79868) = (7 + 9 + 8 + 6 + 8) = 38 \rightarrow d(38) = (3 + 8) = 11 \rightarrow d(11) = (1 + 1) = 2 \rightarrow d(2) = 2$. Размер задачи зависит от количества цифр n, а к начальному условию задачи мы, очевидно, приходим, когда n состоит из одной цифры.

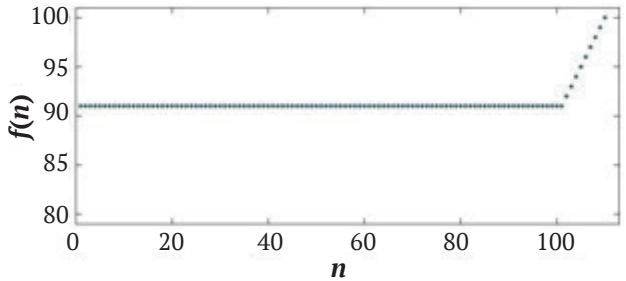


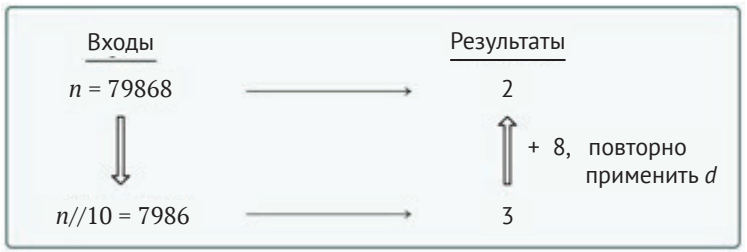
Рис. 11.3. Функция-91 Маккарти

Для рекурсивного условия можно предложить два варианта. Во-первых, по условию задачи n заменяется суммой своих цифр, что приводит к следующему определению цифрового корня:

$$d(n) = \begin{cases} n, & \text{если } n < 10, \\ d(s(n)), & \text{если } n \geq 10, \end{cases}$$

где $s(n)$ – функция суммирования цифр неотрицательного целого числа (см. листинг 2.2). Заметьте, что это – хвостовая рекурсивная функция.

Второй вариант подразумевает уменьшение размера задачи путём отбрасывания одной из цифр n. Рассмотрим следующую конкретную рекурсивную схему:



Рекурсивное условие требует добавления последней цифры n к результату подзадачи и повторного применения того же метода. Таким образом, функцию можно определить исключительно через саму себя:

$$d(n) = \begin{cases} n, & \text{если } n < 10, \\ d(d(n//10) + n\%10), & \text{если } n \geq 10, \end{cases}$$

что является примером вложенной рекурсии. Такую функцию легко закодировать (см. листинг 11.8). Наконец, функция d обладает многими свойствами, которые включают в себя вложенную рекурсию. Например, $d(n + m) = d(d(n) + d(m))$ или $d(n \cdot m) = d(d(n) \cdot d(m))$.

Листинг 11.8. Метод с вложенной рекурсией для вычисления цифрового корня неотрицательного целого числа

```
def digital_root(n):
    if n < 10:
        return n
    else:
        return digital_root(digital_root(n // 10) + n % 10)
```

11.4. К хвостовой и вложенной рекурсии через обобщённую функцию

Мы видели, что хвостовые рекурсивные функции можно вывести, опираясь на итерационный подход или на готовую итерационную версию. В этом разделе мы увидим, что для некоторых простых вычислительных задач, содержащих формулы, можно создать те же хвостовые рекурсивные функции, но с использованием чисто декларативного подхода. Эта стратегия связана с формальными методами разработки программного обеспечения, которые выходят за рамки данной книги. Но суть её в том, что некоторые хвостовые рекурсивные функции реализуются с помощью обобщённых (расширенных) функций, решающих ту же задачу. Например, метод `factorial_tail(n,p)` из листинга 11.4 – обобщение функции факториала. Очевидно, при $p = 1$ метод вычисляет факториал n, но при других значениях p он вычислит нечто иное. В данном случае нетрудно видеть, что метод вычисляет $p \cdot n!$. Стратегия разработки алгоритма, описанная в этом разделе, заключается в разработке таких обобщённых хвостовых рекурсивных функций с использованием рекурсивных понятий и элементарной алгебры. Кроме того, подобная методология может привести к неэффективным, но правильным вложенным рекурсивным алгоритмам.

11.4.1. Факториал

Естественное рекурсивное определение факториала $n! = f(n) = n \cdot f(n-1)$ приводит к линейно-рекурсивной функции. Для создания хвостовой рекурсивной функции ей нужны дополнительные параметры, которые будут содержать информацию, необходимую для того, чтобы вернуть конечный результат в начальном условии. Например, можно ввести новую функцию $g(n, p)$, которая включает новый параметр p и будет определяться своей формулой, зависящей от параметра p, но при этом

позволит нам вычислить $n!$. Несколько вариантов обобщения функции факториала:

$$g(n, p) = p \cdot n!, \tag{11.2}$$

$$g(n, p) = p + n!, \tag{11.3}$$

$$g(n, p) = (n!)^p. \tag{11.4}$$

Очевидно, что из каждой формулы легко получить факториал $f(n)$, задав $p = 1$ в (11.2) и (11.4) или $p = 0$ в (11.3).

11.4.1.1. *Приемлемые обобщения*

Первый шаг состоит в выборе конкретного обобщения, с которым мы будем работать. Из нескольких вариантов обобщения функции факториала сначала разберём $g(n, p) = p \cdot n!$, так как он приводит к простому хвостовому рекурсивному алгоритму. Этот выбор объясняется ещё и тем, что другие обобщения обычно приводят к неэффективным и сложным вложенным рекурсивным функциям.

Для построения рекурсивного алгоритма для $g(n, p)$ можно действовать так же, как в предыдущих главах. Размер задачи – n. Поэтому начальное условие выполняется при $n = 0$, когда функция просто возвращает p (то есть $g(0, p) = p$). Для рекурсивного условия можно попытаться уменьшить n на 1, что привело бы к следующей рекурсивной схеме:

Однако такая декомпозиция приводит не к хвостовому рекурсивному алгоритму, а к линейному, поскольку результат подзадачи здесь должен умножаться на n. В этом случае предыдущая схема бесполезна, потому что она не учитывает тот факт, что цель состоит в разработке хвостового рекурсивного метода, когда результат подзадачи должен совпадать с результатом исходной задачи. Поскольку $g(n, p) \neq g(n - 1, p)$, то прежде всего должен быть изменён способ декомпозиции. Если мы хотим уменьшить размер задачи на 1, то должны использовать другое выражение для второго параметра. Например, $g(n, p) = g(n - 1, q)$, где q – фиктивная переменная, представляющая собой выражение от вход-

ных параметров n и p. Теперь наша цель – определить q, которую можно найти, построив другую, но похожую схему:

Во-первых, следует обратить внимание, что все элементы схемы равны: $g(n, p) = p \cdot n!$ – определение обобщённой функции, тогда как $g(n, p) = g(n - 1, q)$ должно быть хвостовым рекурсивным правилом. Применение определения $g(n - 1, q)$ естественно приводит к $g(n - 1, q) = q \cdot (n - 1)!$. Наконец, формулы в правой части схемы тоже должны быть равными и могут использоваться для получения q. В данном случае мы имеем:

$$p \cdot n! = q \cdot (n - 1)!,$$

откуда следует, что $q = p \cdot n$. В итоге функция определяется следующим образом:

$$g(n,p) = \begin{cases} p, & \text{если } n = 0, \\ g(n-1,p \cdot n), & \text{если } n > 0, \end{cases}$$

что соответствует методу `factorial_tail(n, p)` в листинге 11.4 (а $f(n) = g(n, 1)$ становится функцией-оболочкой). В этом случае параметр p – это результат использования обобщённой функции, а не переменная-накопитель, как в итерационном алгоритме.

11.4.1.2. *Неприемлемые обобщения*

Вообще говоря, если определение функции содержит произведение, то обобщение тоже должно включать произведение, умноженное на новый параметр. Точно так же, если функция содержит сумму, то обобщение должно добавлять дополнительный параметр. Это облегчает нахождение дополнительного параметра (в предыдущем примере – q) в подзадаче.

Но не все обобщения приводят к хвостовым рекурсивным алгоритмам. Например, рассмотрим $g(n, p) = p + n!$, использующую вместо произведения сумму. Во-первых, начальное условие было бы $g(0, p) = p + 0! = p + 1$. Для рекурсивного условия мы можем сформировать следующую диаграмму:

Рекурсивное условие		Формулы		
$g(n, p)$	=	$p + n!$		
$\|\|$		$\|\|$	$\Rightarrow$	$q = ?$
$g(n-1, q)$	=	$g + (n-1)!$		

В этом случае определить q сложнее. Начав с $q + (n-1)! = p + n!$, мы получим:

$$q = p + n! - (n-1)! = p + (n-1) \cdot (n-1)! = p + (n-1) \cdot g(n-1, 0),$$

где последнее равенство следует из определения обобщения. Таким образом, функцию можно определить как

$$g(n,p) = \begin{cases} p+1, & \text{если } n = 0, \\ g(n-1, p + (n-1) \cdot g(n-1, 0)), & \text{если } n > 0, \end{cases} \tag{11.5}$$

что является примером вложенной рекурсии. Хотя алгоритм – правильный, он не только более сложный, но ещё и менее эффективный. Его дерево рекурсии является полным двоичным деревом, из чего следует, что его временна́я сложность порядка $\Theta(2^n)$.

Вариант $g(n, p) = (n!)^p$ еще более затруднителен. Чтобы решить уравнение $((n-1)!)^q = (n!)^p$, нужно сначала прологарифмировать (основание логарифма не имеет значения) обе его части. Это приводит к $q \log((n-1)!) = p \log(n!)$. Таким образом, мы имеем:

$$q = p \log(n!) / \log((n-1)!) = p (\log(n) + \log((n-1)!)) / \log((n-1)!) =$$
$$= p (\log(n) / \log((n-1)!) + 1) = p (\log(n) / \log(g(n-1, 1)) + 1).$$

Это выражение верно только для $n > 2$ (обратите внимание, что невозможно получить q для $(1!)^q = (2!)^p$). Таким образом, для вывода правильной функции нам потребуется несколько начальных условий, а саму функцию можно определить следующим образом:

$$g(n,p) = \begin{cases} 1, & \text{если } n \le 1, \\ 2p, & \text{если } n = 2, \\ g(n-1, p \cdot (\log(n)/\log(g(n-1, 1)) + 1), & \text{если } n > 2. \end{cases}$$

Как видим, она тоже содержит вложенную рекурсию. Как и (11.5), она выполняется за экспоненциальное от n время. Кроме того, присутствие логарифмов означает, что она работает с вещественными числами

вместо целых, и поэтому её результат должен округляться до ближайшего целого числа.

11.4.2. Приведение десятичного числа к другому основанию

Нам уже известны два способа создания алгоритма по преобразованию числа из одной системы счисления в другую. Листинги 4.10 и 11.6 содержат линейное и хвостовое рекурсивные решения соответственно. Теперь мы разработаем тот же хвостовой рекурсивный алгоритм, но с использованием обобщённой функции. Решение будет более сложным, поскольку нам потребуется два дополнительных параметра. Кроме того, гораздо сложнее будет и сама функция преобразования системы счисления, которую можно записать, например, следующим образом:

$$f(n,b) = \sum_{i=0}^{m-1} d_i \cdot 10^i,$$

где n – приводимое к основанию b десятичное число. Кроме того, m – количество цифр по основанию b в новом представлении n, а d_i ($i = 0, \ldots, m - 1$) – цифры этого представления. К счастью, более точного определения этих цифр не требуется.

Поскольку формула представляет собой сумму, мы можем начать с добавления к ней нового параметра s, что приводит к следующей обобщённой функции:

$$g(n,b,s) = s + \sum_{i=0}^{m-1} d_i \cdot 10^i.$$

Для уменьшения размера задачи на 1 в декомпозиции используется целочисленное деление n на b, что приводит к следующей схеме:

Рекурсивное условие		Формулы		
$g(n, b, s)$	=	$s + \sum_{i=0}^{m-1} d_i \cdot 10^i$		
$\|\|$		$\|\|$	$\Rightarrow$	$q = ?$
$g(n//b, b, q)$	=	$q + \sum_{i=0}^{m-1} d_i \cdot 10^{i-1}$		

Из неё следует, что:

$$s + \sum_{i=0}^{m-1} d_i \cdot 10^i = q + \sum_{i=0}^{m-1} d_i \cdot 10^{i-1}.$$

При попытке найти q сразу видно, что решение получается сложным:

$$q = s + d_0 + \sum_{i=0}^{m-1} d_i \cdot (10^i - 10^{i-1}). \tag{11.6}$$

Но нетрудно заметить, что, умножив сумму подзадачи на 10, можно избавиться от суммы в формуле (11.6) и тем самым значительно упростить её. С этой целью можно добавить ещё один параметр – множитель суммы, – чтобы получить более общую функцию

$$h(n,b,p,s) = s + p \cdot \sum_{i=0}^{m-1} d_i \cdot 10^i. \tag{11.7}$$

Для этой новой функции можно создать следующую схему:

Рекурсивное условие		Формулы		
$h(n, b, p, s)$	=	$s + p \cdot \sum_{i=0}^{m-1} d_i \cdot 10^i$		
$\|\|$		$\|\|$	$\Rightarrow$	$q = ?, \quad r = ?$
$h(n//b, b, r, q)$	=	$q + r \cdot \sum_{i=0}^{m-1} d_i \cdot 10^{i-1}$		

В этом случае мы имеем:

$$s + p \cdot \sum_{i=0}^{m-1} d_i \cdot 10^i = q + r \cdot \sum_{i=0}^{m-1} d_i \cdot 10^{i-1},$$

где есть две фиктивные переменные r и q. Если задать $r = 10p$, то выражение сокращается до

$$s + p \cdot \sum_{i=0}^{m-1} d_i \cdot 10^i = q + p \cdot \sum_{i=0}^{m-1} d_i \cdot 10^i,$$

откуда теперь можно легко найти q. А именно: $q = s + p \cdot d_0 = s + p \cdot (n \,\%\, b)$.

Наконец, с учётом того, что начальное условие выполняется при $n < b$, где $f(n, b) = n$, начальным условием для $h(n, b, p, s)$ должно быть $s + p \cdot n$ (это следует из (11.7) при $m = 1$ и $d_0 = n$). Таким образом, функцию h можно определить так:

$$h(n,b,p,s) = \begin{cases} s + p \cdot n, & \text{если } n < b, \\ h(n//b,\ b,\ 10p,\ s + p \cdot (n\%b)), & \text{если } n \geq b, \end{cases}$$

где $f(n, b) = h(n, b, 1, 0)$. Второе начальное условие выполняется при $n = 0$, что приводит к функции из листинга 11.6.

11.5. Упражнения

Упражнение 11.1. Отталкиваясь от итерационного решения, реализуйте хвостовой рекурсивный алгоритм, вычисляющий сумму первых n ($n \geqslant 0$) положительных целых чисел. В частности, рассмотрите, как итерационный алгоритм решил бы задачу с использованием переменной-накопителя промежуточных сумм и окончательного результата. Наконец, выведите хвостовой рекурсивный алгоритм с использованием обобщённой функции.

Упражнение 11.2. Реализуйте хвостовую рекурсивную версию функции из листинга 5.16 и напишите аналогичную итерационную функцию.

Упражнение 11.3. Рассмотрите задачу сложения элементов списка **a** из n чисел. Создайте хвостовую рекурсивную функцию, применив обобщения. После чего преобразуйте её в итерационную.

Упражнение 11.4. Применив обобщение, создайте хвостовую рекурсивную функцию, вычисляющую степень b^n, где b – вещественное число, а n – неотрицательное целое число. После этого преобразуйте её в итерационную.

Упражнение 11.5. Создайте хвостовую рекурсивную функцию, вычисляющую n-е число Фибоначчи (для $n > 0$), применив обобщение. В частности, используйте обобщённую функцию $f(n, a, b) = G(n)$, которая задаёт последовательность, отвечающую рекурсивному правилу $G(n) = G(n - 1) + G(n - 2)$, где $G(1) = a$ и $G(2) = b$. После этого преобразуйте код в итерационную версию.

Упражнение 11.6. Создайте функцию, вычисляющую n-е число Фибоначчи $F(n)$ (для $n > 0$), используя обобщённую функцию $f(n, s) = s + F(n)$. Декомпозиция задачи должна уменьшать её размер на единицу.

Упражнение 11.7. Для функции Аккермана (11.1) нарисуйте её дерево активации на примере вызова $A(2, 1)$.

Глава 12

Множественная рекурсия III: перебор с возвратами

Кто хочет найти жемчуг, должен нырнуть глубже.

– Джон Драйден

Перебор с возвратами (backtracking) – одна из самых важных парадигм разработки алгоритмов. Её можно считать «интеллектуальной» стратегией грубой силы (решением «в лоб»), выполняющей исчерпывающий поиск решений задач с заданными ограничениями и задач дискретной оптимизации. Подход может использоваться для решения огромного числа головоломок и задач, включая задачи о восьми ферзях, поиска пути в лабиринте, судоку, оптимизации рюкзака 0–1 и многие другие.

В самом общем случае методы перебора с возвратами сочетают в себе рекурсию и итерацию, имеют несколько параметров и обычно разрабатываются без строгого следования парадигмам декомпозиции и индукции. Поэтому тем, кто изучает материал впервые, они могут показаться сложными. К счастью, все алгоритмы перебора с возвратами зачастую имеют схожую структуру, что позволяет облегчить их разработку. Эта структура зафиксирована в так называемых «шаблонах перебора с возвратами», которые зависят только от языка и стиля программирования. В этой книге все методы обладают вполне определённой структурой с большим множеством параметров, что придаёт им достаточно большую степень свободы и лишь изредка требует дополнительных методов. При этом читатель должен отдавать себе отчёт в том, что существуют и другие варианты их реализации. Но в любом случае студенты могут относительно легко освоить перебор с возвратами, изучив приведённые ниже примеры и применив подобные алгоритмы к другим задачам.

12.1. Введение

В этом разделе излагаются основы метода перебора с возвратами и приводится его краткий обзор на примере простой задачи о четырёх ферзях. Её цель – разместить четыре ферзя на шахматной доске размером 4×4 так, чтобы они не угрожали друг другу. Поскольку ферзи могут ходить по горизонтали, вертикали и диагонали, на любой из них не может находиться более одного ферзя. На рис. 12.1 приведено одно из двух возможных решений задачи. Естественно, задачу можно обобщить до размещения n ферзей на шахматной доске размером $n \times n$ (см. раздел 12.3).

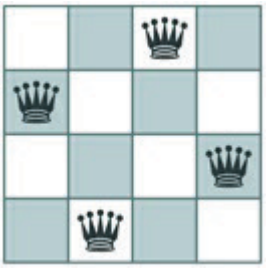


Рис. 12.1. Одно из решений задачи четырёх ферзей

12.1.1. Частичные и полные решения

Перебор с возвратами – это общая стратегия решения вычислительных задач, состоящая в выборе одного из конечного дискретного множества вариантов (элементов). Таким образом, формально говоря, перебор с возвратами – это стратегия поиска решения в «дискретном (конечном) пространстве состояний». Кроме того, это – метод решения «в лоб» в том смысле, что поиск является исчерпывающим. Другими словами, если решение существует, то алгоритм перебора с возвратами обязательно его найдёт.

Метод ведёт поиск решения с созданием и обновлением *частичного решения* (partial solution), которое в итоге может стать правильным *полным решением* (complete solution) задачи. Частичные решения создаются путём пошагового наращивания количества элементов-*кандидатов* (candidate) в последовательных рекурсивных вызовах. Таким образом, неявно подразумевается некий порядок следования кандидатов в решении. В этом смысле частичное решение можно считать «префиксом» полного решения. Если полное решение задачи – список, то частичное решение – это просто его подсписок из нескольких первых элементов (светлая область на рис. 12.2(а)). Если же полное решение представляет

собой двумерную матрицу, то её элементы необходимо линейно упорядочить каким-то образом. Два самых естественных способа – упорядочить элементы матрицы по строкам или по столбцам, как показано на рис. 12.2(b) и рис. 12.2(c) соответственно. В этом случае частичное решение будет представлять собой множество из нескольких первых элементов матрицы, согласно выбранному линейному порядку.

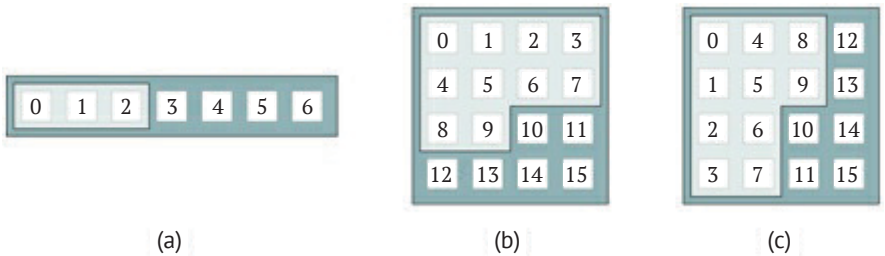


Рис. 12.2. Частичные решения внутри полных решений для списка или матрицы

Ключевой момент перебора с возвратами – эффективная проверка частичного решения на соответствие заданным ограничениям задачи. Если она успешна, то появляется возможность довести частичное решение до полного правильного решения. Если же правильное частичное решение после добавления к нему нового элемента оказывается недопустимым, алгоритм должен вернуться («откатиться») к другому, предыдущему, правильному частичному решению и продолжить перебор других, ещё не рассмотренных вариантов. Таким образом, алгоритмы перебора с возвратами выполняют исчерпывающий перебор вариантов, но для некоторых задач могут оказаться довольно эффективными за счёт сокращения тех частичных решений, которые определённо не могут привести к правильным полным решениям.

Первый вопрос при решении задач методом перебора с возвратами – выбор конкретной формы или структуры данных для решений. Как правило, они реализуются в виде списка или матрицы. В частности, для задачи размещения n ферзей первое, что приходит в голову, – это представить решения задачи в виде логической матрицы $n \times n$, в которой n истинных значений соответствуют тем клеткам шахматной доски, где расположены ферзи. Частичные решения будут иметь ту же структуру, но включать меньше, чем n ферзей (разместив n-го ферзя, мы получим либо неправильное, либо полное решение задачи). Однако, несмотря на естественность и простоту матричного представления, такой выбор структуры данных нельзя признать удачным, поскольку он приводит к неэффективному алгоритму. Так как методы перебора с возвратами

выполняют исчерпывающий перебор, алгоритму для размещения очередного ферзя в одной из клеток шахматной доски придётся просматривать всю матрицу по строкам или столбцам. Это займёт слишком много времени, ведь размещать ферзя, скажем, в столбце, где уже стоит ферзь, алгоритму нет необходимости. Вместо этого алгоритм должен просто перейти к следующему столбцу.

Поэтому более подходящая структура данных для представления решений этой задачи – простой список длины n. В этом случае индексы списка могут представлять номера вертикалей шахматной доски, а значения элементов списка – номера горизонталей, где стоят ферзи. Другими словами, если **x** – список, представляющий решение, и ферзь стоит в столбце (вертикали) i и строке (горизонтали) j, то $x_i = j$. Например, если столбцы пронумерованы слева направо, а строки – сверху вниз, начиная с нуля, то решение на рис. 12.1 соответствует списку [1, 3, 0, 2]. Использующий такой список алгоритм, (правильно) разместив ферзя в строке для определённого столбца (i), может попытаться разместить ферзя в следующем столбце (i + 1). Кроме того, он может отслеживать строки, где уже стоит ферзь, чтобы не размещать в них следующих ферзей.

12.1.2. Рекурсивная структура

Чтобы понять принцип работы алгоритма перебора с возвратами, рассмотрим его дерево рекурсии. На рис. 12.3 приведено дерево рекурсии для метода перебора с возвратами, который находит одно решение задачи четырёх ферзей.

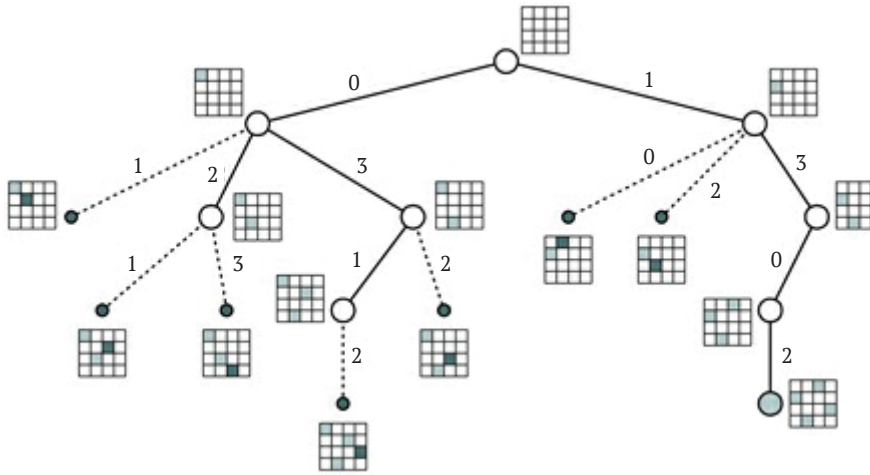


Рис. 12.3. Дерево рекурсии алгоритма перебора с возвратами, который находит одно из решений задачи четырёх ферзей

Корневой узел дерева представляет собой первый вызов метода, где частичное решение в виде квадратной доски размером 4×4 (но представленное списком) пусто. Все последующие вызовы метода выполняются в порядке прямого (preorder) обхода дерева рекурсии. Первый рекурсивный вызов, связанный с левым потомком корня, размещает ферзя (затенённая клетка доски) в первой строке первого столбца. Метка «0» над ветвью дерева обозначает номер строки, на которую поставлен ферзь. Поскольку это частичное решение удовлетворяет ограничениям задачи, метод может продолжить размещение ферзей во втором столбце. В частности, он начинается с исключения попытки размещения ферзя в первой строке, поскольку он там уже есть. Затем алгоритм проверяет, можно ли разместить ферзя во второй строке. Очевидно, что этого сделать нельзя, потому что два ферзя оказались бы на одной диагонали. Значит, такое частичное решение неверно, и алгоритм не должен рассматривать его дальше. Другими словами, он не выполняет последующих рекурсивных вызовов с таким (недопустимым) частичным решением. Такие ветви обозначены на рисунке пунктиром.

Дальше метод ставит ферзя в третью строку, что приводит к правильному частичному решению. Следовательно, алгоритм может попытаться разместить ферзя в третьем столбце. Очередная свободная строка – вторая, но эта попытка приводит к недопустимому частичному решению. Размещение ферзя в четвёртой строке также нарушает условия задачи. Поскольку других вариантов нет, алгоритм «откатывается» к узлу с одним ферзём в первом столбце первой строки. Затем метод рассматривает все возможные частичные решения, которые возникают после размещения ферзя в 4-й строке 2-го столбца. Поскольку и в этом случае ни одного правильного решения нет, алгоритм вынужден будет вернуться к начальному узлу. Отметим, что на данном этапе метод выполняет полный перебор решений, когда ферзь стоит в первой строке первого столбца. На следующем этапе алгоритм продолжает перебор вариантов, начав с размещения ферзя на второй строке первого столбца. Та же самая процедура повторяется до тех пор, пока в конце концов метод не сможет разместить четырёх ферзей, которые не угрожают друг другу. В этом случае он может либо прекратить поиск, если требовалось найти только одно решение, либо продолжить его, чтобы найти все решения.

Дерево рекурсии демонстрирует также несколько особенностей алгоритма. Во-первых, узлы соответствуют рекурсивным вызовам метода, для которых частичное решение (входной параметр метода) правильно. Маленькие темные узлы, «висящие» на пунктирных ветвях, обознача-

ют возможные вызовы метода в тех случаях, когда соответствующие им частичные решения оказываются правильными. Кроме того, обратите внимание, что для каждого такого неосуществлённого рекурсивного вызова указан новый элемент, добавляемый к частичному решению. Другими словами, частичные решения из k элементов связаны с узлами глубины k. Следовательно, полные решения достигаются в листовых узлах глубины n. Таким образом, путь от корня до узла определяет частичное решение или, если узлом является лист, полное решение. Например, найденное алгоритмом решение [1, 3, 0, 2] – это список меток тех ветвей, по которым пролегает путь от корня до листа-решения. Кроме того, поскольку значения в списке всегда представляют только различные строки, решение задачи четырёх ферзей – это фактически одна из перестановок номеров строк, то есть первых n неотрицательных целых чисел. Поэтому полное дерево перебора всех вариантов имело бы $n!$ листьев, а это довольно большая величина даже для малых значений n. Однако на рис. 12.3 видно, что дерево рекурсии можно существенно сократить за счёт исключения из рассмотрения недопустимых частичных решений, что является ключевым моментом в построении эффективных алгоритмов перебора.

Подобные моменты будут выявляться и в других задачах и примерах этой главы. В частности, в задаче о размещении n ферзей, которая вместе с кодом алгоритма её решения подробно разбирается в разделе 12.3. Или, например, дерево на рис. 8.18 может соответствовать дереву рекурсии алгоритма поиска размещения с повторениями из четырёх элементов по два. Списки рядом с узлами можно считать частичными или полными решениями, а метки над ветвями – новыми элементами, добавляемыми к частичному решению. На практике сокращение деревьев рекурсии ради повышения эффективности алгоритмов перебора с возвратами достигается за счёт отбрасывания недопустимых решений.

12.2. Генерация комбинаторных объектов

Мы только что узнали, что задача об n ферзях сводится к задаче о перестановке первых n неотрицательных целых чисел, которая будет возникать во многих задачах. Кроме этого, большое число других задач сводится к нахождению различных подмножеств множества из n элементов. Поэтому важно знать, как генерировать перестановки и подмножества множества из n различных элементов. В этом разделе приводятся алгоритмы, генерирующие такие комбинаторные объекты. Польза их в том,

что они могут послужить отправными точками при разработке алгоритмов перебора с возвратами для многих других задач.

Отметим, что эти алгоритмы не будут создавать списков или иных структур данных для хранения всех подмножеств или перестановок. Это было бы непрактично из-за большого количества вариантов (для множества из n различных элементов существует $n!$ перестановок и 2^n подмножеств). Вместо этого методы будут использовать простой список, который содержит только одно частичное или полное решение (подмножество или перестановку) и обновляется в процессе выполнения алгоритма. Таким образом, методы будут выдавать все решения, не сохраняя их в единой структуре данных, а просто печатая их или подсчитывая их количество во время выполнения.

12.2.1. Подмножества

В этом разделе представлены две стратегии генерации всех подмножеств множества из n (различных) элементов, которое будет задаваться входным списком. В обоих методах дерево рекурсии будет двоичным, а решения (подмножества) будут представляться его листьями. В одном методе частичные решения имеют фиксированную длину n, тогда как в другом их длина меняется при выполнении рекурсивного вызова.

12.2.1.1. *Частичные решения фиксированной длины*

На рис. 12.4 приведено двоичное дерево рекурсии алгоритма, генерирующего восемь подмножеств, которые можно создать на множестве из трёх различных элементов.

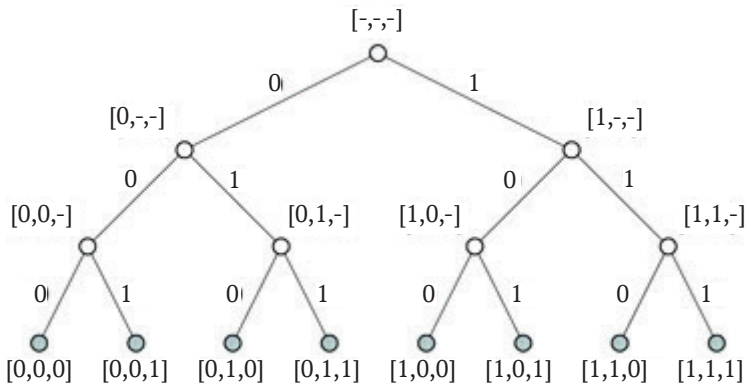


Рис. 12.4. Дерево рекурсии алгоритма генерации всех подмножеств множества из трёх элементов

Каждое подмножество задаётся двоичным списком из n нулей или единиц (конечно, допускаются и логические значения). Например, для исходного списка элементов [*a*, *b*, *c*] список [1, 0, 1] будет обозначать подмножество {*a*, *c*}. Этот двоичный список – частичное решение, которое становится полным по достижении методом листа дерева рекурсии, а сами двоичные цифры – это элементы-кандидаты в процессе его формирования. При выполнении рекурсивного вызова метод просто добавляет нового кандидата (двоичная цифра над ветвью дерева) к частичному решению (указано рядом с узлом дерева), передаваемому методу в качестве аргумента. Исходная длина списка – n, но его элементы не определены. По достижении начального условия метод генерирует одно из 2^n возможных подмножеств в одном из листьев дерева рекурсии. В этом случае список содержит n уже определённых элементов и представляет собой полное решение.

В листинге 12.1 приведён алгоритм, соответствующий дереву рекурсии на рис. 12.4 и печатающий каждое подмножество с помощью метода `print_subset_binary`. Кроме исходного множества (списка) из n элементов, процедура метода получает также список `sol` (частичное решение) из n элементов со значениями `None`, присвоенными ему в методе-оболочке `generate_subsets_wrapper`. Обратите внимание, что на каждом уровне метод, спускаясь вниз по дереву рекурсии, включает в частичное решение нового кандидата. По этой причине он использует третий параметр `i` – индекс элемента списка (частичного решения), куда будет помещена новая цифра 0 или 1. Кроме того, `i` также соответствует уровню узла рекурсивного вызова, отвечающего за добавление нового элемента в частичное решение. Поскольку рекурсивный метод перебора с возвратами `generate_subsets` добавляет новую цифру 0 или 1, начиная с индекса 0 списка частичного решения, в `generate_subsets_wrapper` индексу `i` присваивается начальное значение 0.

Метод `generate_subsets` в строке 3 проверяет, является ли частичное решение `sol` полным, то есть совпадает ли значение `i` с n. Эта проверка отвечает начальному условию, когда метод должен просто напечатать решение (строка 5). В рекурсивном условии процедура должна дважды выполнить рекурсивный вызов, чтобы включить новых кандидатов в частичное решение. Это достигается простым циклом, в котором переменная `k` принимает значения 0 или 1 (строка 8), а элемент частичного решения `sol[i]` получает значение `k` (строка 11). Все алгоритмы перебора с возвратами в этой книге также будут использовать цикл для пе-

ребора всех возможных кандидатов, которых можно добавить к частичному решению. После этого процедура вызывает саму себя (строка 14) с изменённым частичным решением и увеличенным на 1 индексом `i`, чтобы добавить нового кандидата (если частичное решение ещё не стало полным).

Листинг 12.1. Печать всех подмножеств множества, заданного списком

```
1  def generate_subsets(i, sol, elements):
2      # Base case
3      if i == len(elements):
4          # Print complete solution
5          print_subset_binary(sol, elements)
6      else:
7          # Generate candidate elements
8          for k in range(0, 2):
9
10             # Include candidate in partial solution
11             sol[i] = k
12
13             # Expand partial solution at position i+1
14             generate_subsets(i + 1, sol, elements)
15
16             # Remove candidate from partial solution
17             sol[i] = None   # optional
18
19
20 def generate_subsets_wrapper(elements):
21     sol = [None] * (len(elements))
22     generate_subsets(0, sol, elements)
23
24
25 def print_subset_binary(sol, elements):
26     no_elements = True
27     print('{', end='')
28     for i in range(0, len(sol)):
29         if sol[i] == 1:
30            if no_elements:
31               print(elements[i], sep='', end='')
32               no_elements = False
33         else:
34               print(', ', elements[i], sep='', end='')
35     print('{')
```

Строка 17 не обязательна, поскольку метод и без неё работает правильно. Она включена в процедуру на тот случай, если такое действие понадобится в других алгоритмах перебора с возвратами. Обратите внимание, что когда определённый рекурсивный вызов в узле дерева завершается, управление передаётся его родителю, для которого позиция `i` частичного решения уже не имеет значения, и строка 17 явно отражает этот факт. Однако метод работает и без неё, потому что по достижении начального условия (когда частичное решение становится полным) он всего лишь печатает подмножество, а присваивание в строке 11 просто обнуляет этот элемент при `k = 1`. Чтобы проследить изменения частичного решения, читателю рекомендуется выполнить программу шаг за шагом с помощью отладчика (со строкой 17 или без неё).

Например, вызов `generate_subsets_wrapper(['a','b','c'])` даёт следующий результат:

```
{}
{c}
{b}
{b, c}
{a}
{a, c}
{a, b}
{a, b, c}
```

12.2.1.2. *Частичные решения переменной длины*

В предыдущем алгоритме частичные решения имели фиксированную длину n. Такая процедура может работать со списками или массивами, размер которых не меняется. В этом случае для указания индекса добавляемого к частичному решению элемента был необходим параметр `i`. Если же вместо частичных решений фиксированной длины использовать частичные решения переменного размера, то необходимость в параметре `i` отпадает. На рис. 12.5 приведено другое рекурсивное дерево с той же, как на рис. 12.4, структурой, однако метки (частичные решения) рядом с узлами (вызовами рекурсивного метода) теперь не содержат неопределённостей.

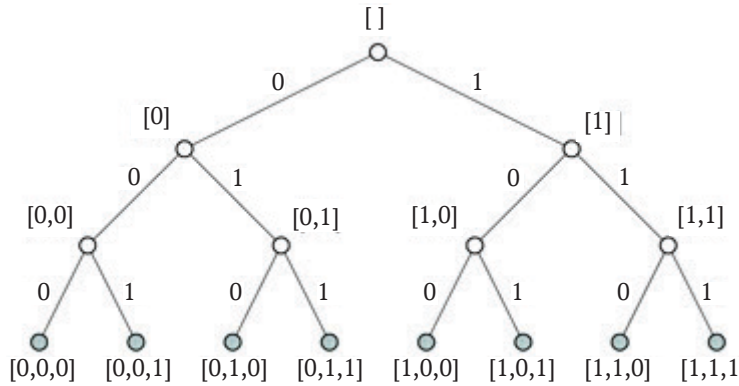


Рис. 12.5. Двоичное дерево рекурсии альтернативного алгоритма генерации всех подмножеств множества из трёх элементов

В листинге 12.2 приведён соответствующий этому подходу метод, который очень похож на код листинга 12.1.

Листинг 12.2. Альтернативная печать всех подмножеств множества, заданного списком

```
def generate_subsets_alt(sol, a):
    # Base case
    if len(sol) == len(a):
        # Print complete solution
         print_subset_binary(sol, a)
    else:
        # Generate candidate elements
        for k in range(0, 2):

            # Include candidate in partial solution
            sol = sol + [k]

            # Expand partial solution at position i+1
            generate_subsets_alt(sol, a)

            # Remove candidate from partial solution
            del sol[-1]


def generate_subsets_alt_wrapper(elements):
    sol = []
    generate_subsets_alt(sol, elements)
```

В начальном условии он просто печатает решение, если оно содержит n элементов. В рекурсивном условии он использует тот же цикл, в котором в конец частичного решения он добавляет 0 или 1, а затем вызывает сам себя. В этом случае код в строке 17 становится необходимым, так как метод должен «отменить» все изменения в частичном решении до рекурсивного вызова самого себя. Если в листинге 12.1 алгоритм обнуляет единицу в частичном решении, то в данном случае метод должен сначала удалить 0 из списка, чтобы затем правильно добавить в него 1. Таким образом, вызов рекурсивного метода подразумевает добавление нового элемента к частичному решению (строка 14) и поэтому должен удалить старый элемент (строка 17) по возвращении из рекурсивного вызова. Строка 17 из листинга 12.1 имеет тот же смысл, но в ней нет необходимости. Что касается эффективности, то динамическое добавление элементов в структуру данных и их удаление из неё может занимать значительное время. Таким образом, зачастую лучше выделить для частичного решения фиксированный объём памяти и обновлять его, не меняя его размера. В этом отношении код в листинге 12.1 выполняется быстрее, чем метод `generate_subsets_alt`. Наконец, для этого альтернативного алгоритма метод-оболочка должен инициализировать частичное решение пустым списком.

12.2.2. Перестановки

В этом разделе рассматриваются два схожих алгоритма, которые печатают все возможные перестановки n различных элементов заданного списка (множества). В первом частичное решение (перестановка) представлено списком индексов от 0 до $n - 1$, указывающих на позиции элементов в исходном списке. Например, для исходного списка [a, b, c] частичное решение [1, 2, 0] означает перестановку [b, c, a]. Во всех последующих алгоритмах частичное решение (перестановка) будет состоять просто из элементов исходного списка (и пустых значений `None`). На рис. 12.6 показана структура дерева рекурсии для такого алгоритма на примере списка [a, b, c].

Заметьте, что список с частичным решением имеет длину n, но только первые его элементы значимы. При первом вызове метода частичное решение пусто, то есть все его элементы установлены в `None`. Кроме того, процедуры получают (первоначально установленный в ноль) параметр `i`, который задаёт количество уже включённых в частичное решение элементов-кандидатов. Таким образом, он указывает место нового кандидата в частичном решении и уровень узла в дереве рекурсии или глубину рекурсивного вызова.

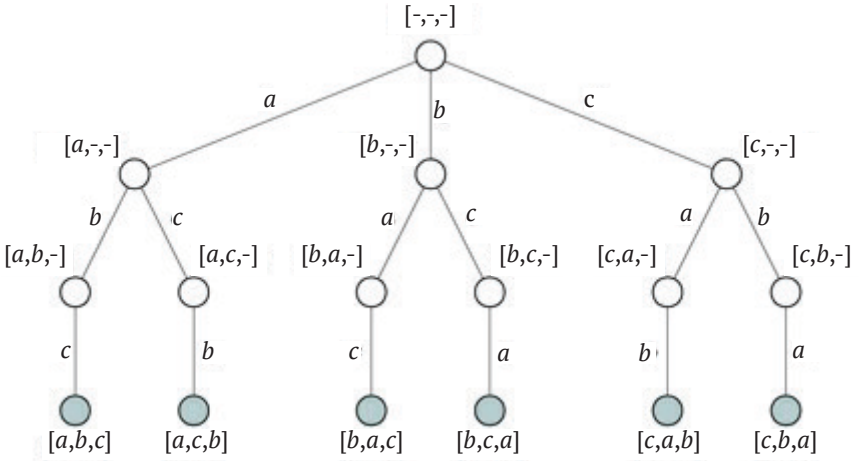


Рис. 12.6. Дерево рекурсии алгоритма генерации всех перестановок множества из трёх элементов

Метод начинается с добавления одного из n элементов-кандидатов исходного списка в первую позицию частичного решения. Поскольку вариантов всего n, корневой узел имеет n дочерних узлов. На следующем уровне дерева методы вызывают себя (n – 1) раз, так как элемент, помещённый в первую позицию, уже не может снова появиться в перестановке. Точно так же на следующем уровне частичные решения содержат по два элемента, а методы вызывают себя (n – 2) раза. И так повторяется до тех пор, пока метод не достигает начального условия в листе дерева рекурсии, где частичные решения становятся полными перестановками.

В следующих двух подразделах описываются способы проверки правильности частичного решения в каждом из двух методов, в чём и заключается их основное различие.

12.2.2.1. *Проверка правильности частичных решений без использования дополнительных структур данных*

В листинге 12.3 приводится первый метод, соответствующий дереву рекурсии на рис. 12.6. Поскольку `i` – это число элементов в частичном решении `sol`, первый условный оператор `if` выполняет проверку начального условия. Если результат – `True`, процедура просто печатает перестановку, сохранённую в `sol`. Иначе алгоритм в цикле начинает добавлять в частичное решение новых кандидатов из исходного списка. Для каждого значения от 0 до n – 1 счётчика цикла `k` в роли индекса

элемента списка алгоритм выполняет проверку на отсутствие кандидата в частичном решении (строка 10). Стоит отметить, что, несмотря на лаконичность записи условия в строке 10, оно может потребовать в самом худшем случае `i` проверок значений кандидатов. Если кандидата ещё нет в частичном решении, метод добавляет его (строка 13) и выполняет рекурсивный вызов с увеличением `i` на единицу (строка 16). Следующая за рекурсивным вызовом строка 19 с «отклонением» кандидата не обязательна, так как метод всё равно перезаписывает значение `sol[i]`. Иными словами, нет нужды присваивать `sol[i]` значение `None`. Тем не менее эта строка включена в код, чтобы он точно соответствовал дереву рекурсии на рис. 12.6.

Листинг 12.3. Печать всех перестановок элементов множества из списка

```
1  def generate_permutations(i, sol, elements):
2      # Base case
3      if i == len(elements):
4          print_permutations(sol)   # complete solution
5      else:
6          # Generate candidate elements
7          for k in range(0, len(elements)):
8
9              # Check candidate validity
10             if not elements[k] in sol[0:i]:
11
12                 # Include candidate in partial solution
13                 sol[i] = elements[k]
14
15                 # Expand partial solution at position i+1
16                 generate_permutations(i + 1, sol, elements)
17
18                 # Remove candidate from partial solution
19                 sol[i] = None   # not necessary
20
21
22 def generate_permutations_wrapper(elements):
23     sol = [None] * (len(elements))
24     generate_permutations(0, sol, elements)
25
26
27 def print_permutations(sol):
28     for i in range(0, len(sol)):
29          print(sol[i], ' ', end='')
30     print()
```

Метод `generate_permutations_wrapper` – это процедура-оболочка, необходимая для инициализации частичного решения *n* значениями `None` и вызова рекурсивного метода `generate_permutations` с нулевым значением `i`. В заключение код содержит процедуру печати через пробел элементов списка (перестановки) `sol`.

Итак, подобно остальным алгоритмам этой главы, метод использует цикл для подбора возможных кандидатов на включение в частичное решение. Без учёта проверок ограничений во внутренних узлах дерева данный алгоритм может привести к полному *n*-арному дереву рекурсии (у каждого внутреннего узла дерева *n* дочерних узлов) с n^n листьями. Конечно, можно проверять правильность частичных решений в каждом из этих n^n листьев. Но гораздо лучше проверять допустимость кандидата во внутреннем узле и отказываться от ненужных рекурсивных вызовов, если алгоритм обнаруживает совершенно недопустимое частичное решение. Условие в строке 6 обеспечивает такую проверку и резко ускоряет алгоритм, сокращая дерево рекурсии, как показано на рис. 12.7. Более светлые узлы и ветви дерева – это те вызовы метода полного троичного дерева, которые не выполняются (сокращаются) благодаря условию в строке 6. Получающееся дерево рекурсии (изображенное более тёмными узлами и ветвями) становится идентичным изображённому на рис. 12.6 и содержит *n*! листьев. Хотя эта величина тоже внушительна даже для относительно малых значений *n*, она всё же значительно меньше n^n.

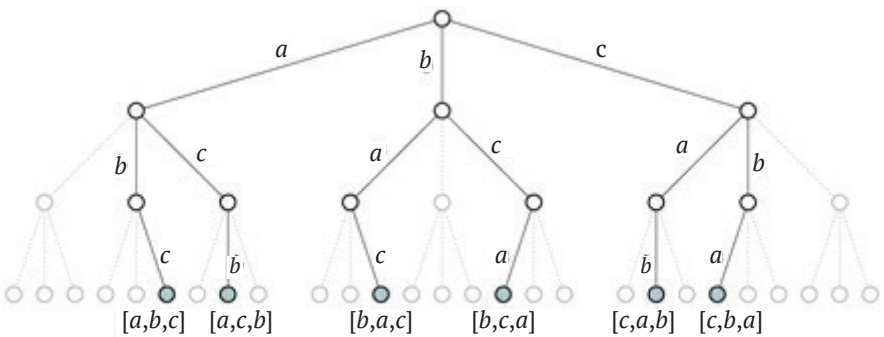


Рис. 12.7. Сокращение дерева рекурсии за счёт исключения недопустимых частичных решений

12.2.2.2. *Проверка правильности частичных решений с использованием дополнительных структур данных*

Вместо проверки правильности частичного решения путём перебора его первых `i` значений (они должны отличаться от `elements[k]`) код в

листинге 12.4 использует дополнительный параметр – логический список `available` размера *n*, в котором хранятся признаки доступности элементов исходного множества для включения их в частичное решение. Поэтому в методе-оболочке всем его *n* элементам изначально присваивается значение `True`. В этом случае условие в строке 10 становится гораздо проще и может быть оценено по времени как Θ(1). Таким образом, если элемент `k` действительно доступен, процедура сначала включает его в частичное решение (строка 13) и делает его недоступным (строка 16), а затем уже вызывает саму себя с увеличением параметра `i` (строки 19 и 20). По завершении вызова выбранный кандидат снова должен стать доступным для включения в частичное решение (строка 23). Обратите внимание, что по завершении рекурсивного вызова метод может выполнить несколько больше итераций цикла, чтобы вставить другие элементы в позицию `i` частичного решения, но перед этим список `available` должен быть приведён к своему начальному состоянию. В частности, `available[k]` должен быть установлен в `True`, так как алгоритму ещё придётся включать `elements[k]` в следующие позиции частичного решения. В итоге, несмотря на дополнительный входной параметр, этот алгоритм эффективнее алгоритма из листинга 12.3.

12.3. Задача *n* ферзей

Задача *n* ферзей была представлена ранее в разделе 12.1. Напомним, что это задача удовлетворения ограничений (constraint satisfaction) или соблюдения заданных условий, решение которой сводится к перестановкам строк, в которых расположены ферзи. Поэтому для решения данной задачи методом перебора с возвратами можно использовать в качестве отправной точки код из листинга 12.4, генерирующий перестановки. Хотя конечный алгоритм потребует некоторых модификаций, его структура будет очень похожа на метод генерации перестановок.

Рассмотрим метод `generate_permutations_alt` и его входные параметры. Он переставляет элементы списка, которые могут содержать произвольные числа, символы или другие типы данных. В данном случае перестановке подлежат горизонтали (строки) шахматной доски. Поскольку их номера – это просто целые числа от 0 до *n* – 1, можно написать код без применения списка. Но мы всё же будем использовать для частичного решения список `sol` номеров строк размера *n*, параметр `i` для номера столбца, в котором размещается очередной ферзь, а также логический список размера *n* `available`, в котором помечены свободные строки – кандидаты для включения в частичные решения.

Мы поменяем имя этого списка на `free_rows`, так как даже в отсутствие ферзя в строке она может быть *недоступной* для включения в частичное решение из-за дополнительного ограничения, связанного с диагональю.

Листинг 12.4. Альтернативная печать всех перестановок элементов множества в списке

```
def generate_permutations_alt(i, available, sol, elements):
    # Base case
    if i == len(elements):
        print_permutations(sol)   # complete solution
    else:
        # Generate candidate elements
        for k in range(0, len(elements)):

            # Check candidate validity
            if available[k]:

                # Include candidate in partial solution
                sol[i] = elements[k]

                # k-th candidate no longer available
                available[k] = False

                # Expand partial solution at position i+1
                generate_permutations_alt(i + 1, available,
                                          sol, elements)

                # k-th candidate available again
                available[k] = True


def generate_permutations_alt_wrapper(elements):
    available = [True] * (len(elements))
    sol = [None] * (len(elements))
    generate_permutations_alt(0, available, sol, elements)
```

В дополнение к этим параметрам мы будем использовать ещё два логических списка для указания присутствия ферзя на диагоналях шахматной доски. Есть два вида диагоналей: a) основные, параллельные главной диагонали от верхнего левого угла до нижнего правого угла шахматной доски, и b) побочные, перпендикулярные основным.

Их ровно $2n - 1$ каждого вида, как показано на рис. 12.8 для $n = 4$. Таким образом, мы будем использовать логические списки `free_pdiags` и `free_sdiags` – оба длиной $2n - 1$, которые будут содержать `True`, если нет ферзей на основных и побочных диагоналях соответственно. На рисунке также показан способ нумерации диагоналей, где пара чисел внутри клетки задаёт номера её строки и столбца на доске. Заметим, что, с одной стороны, сумма номера столбца и номера строки клетки в каждой побочной диагонали постоянна. Она меняется в диапазоне от 0 до $2n - 2$ и может служить индексом в `free_sdiags`. С другой стороны, для каждой основной диагонали разность номера строки и номера столбца постоянна и меняется в диапазоне от $-(n - 1)$ до $n - 1$. Поскольку индексы должны быть неотрицательными, можно просто увеличить эту разность на $n - 1$ и получить правильный индекс для основной диагонали, который после этого будет находиться в диапазоне от 0 до $2n - 2$. Эти простые операции позволят нам по номеру строки и столбца быстро определить диагональ, на которой расположена клетка.

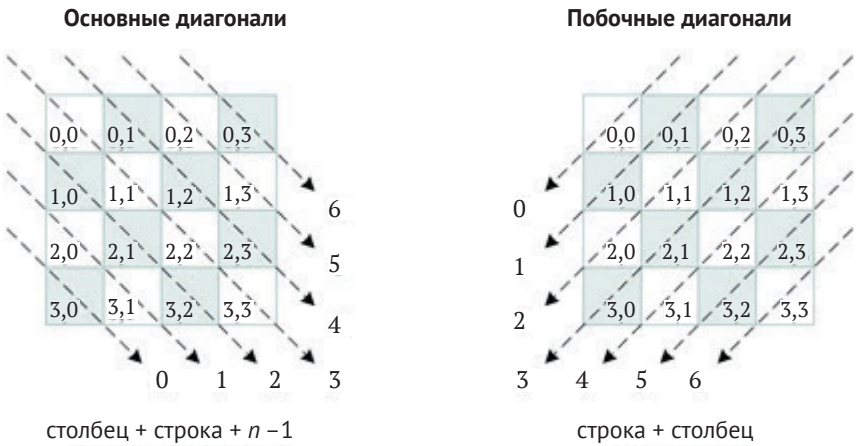


Рис. 12.8. Индексированные основные и побочные диагонали матрицы или шахматной доски

12.3.1. Поиск всех решений

В листинге 12.5 приведён рекурсивный алгоритм перебора с возвратами, который находит все решения задачи n ферзей. Для $n = 8$ возможно 92 решения, хотя только 12 из них действительно различны в том смысле, что не могут быть получены путём поворотов и отражений других решений.

В начальном условии метод обрабатывает правильную перестановку, если она является полным решением (строки 6 и 7). Например, он мог бы печатать это решение (см. метод `print_chessboard` в листинге 12.6) или изображение шахматной доски с расположенными на ней ферзями. Цикл в строке 12 служит для перебора всех кандидатов на включение в частичное решение, а переменная цикла `k` представляет номера строк шахматной доски. Условный оператор в строках 16 и 17 проверяет, является ли строка `k` правильным кандидатом для столбца `i`. В частности, он проверяет, что строка и соответствующие диагонали свободны (то есть не содержат ферзя). Если результат проверки – `True`, алгоритм может включить строку `k` в частичное решение (строка 20). Поскольку это подразумевает размещение нового ферзя на шахматной доске, необходимо обновить логические структуры данных, чтобы отразить тот факт, что строка и две диагонали теперь уже не свободны (строки 24–26). После этого метод может вызвать себя (строки 30 и 31) с изменёнными списками, чтобы продолжить размещение ферзей в следующем столбце ($i + 1$). Наконец, после рекурсивного вызова метод должен на следующем шаге подготовить цикл для проверки возможности размещения ферзя в следующей строке. Для этого ему необходимо вернуть логические списки в то состояние, в котором они находились до включения строки `k` в частичное решение, то есть изменить значения в списках так, чтобы и сама строка `k`, и обе диагонали клетки в столбце `i` и строке `k` стали свободными от ферзей (строки 37–39).

12.3.2. Поиск одного решения

Во многих случаях представляет интерес найти только одно решение задачи, тем более что некоторые из них часто имеют единственное решение (например, задача судоку). Поэтому имеет смысл разработать методы перебора с возвратами, которые прекращают поиск решений, как только одно из них найдено. В языке Python есть возможность применить оператор `return` (возврат) сразу после обнаружения решения. Например, мы можем включать его сразу после вызова `print_chessboard(sol)` в методе `nqueens_all_sol` из листинга 12.5. Однако во многих языках программирования этого делать нельзя.

Существует множество способов реализовать программу поиска и обработки одного решения. В листинге 12.6 приводится один из таких способов, где рекурсивный метод представляет собой логическую функцию, возвращающую `True`, если решение найдено.

Листинг 12.5. Поиск всех решений задачи *n* ферзей

```
def nqueens_all_sol(i, free_rows, free_pdiags,
                    free_sdiags, sol):
    n = len(sol)

    # Test if the partial solution is a complete solution
    if i == n:
        print_chessboard(sol)   # process the solution
    else:

        # Generate all possible candidate that could
        # be introduced in the partial solution
        for k in range(0, n):

            # Check if the partial solution with the
            # k-th candidate would be valid
            if (free_rows[k] and free_pdiags[i - k + n - 1]
                    and free_sdiags[i + k]):

                # Introduce candidate k in the partial solution
                sol[i] = k

                # Update data structures, indicating that
                # candidate k in the partial solution
                free_rows[k] = False
                free_pdiags[i - k + n - 1] = False
                free_sdiags[i + k] = False

                # Perform a recursive call in order to include
                # more candidates in the partial solution
                nqueens_all_sol(i + 1, free_rows, free_pdiags,
                                free_sdiags, sol)

                # Eliminate candidate k from the partial
                # solution, and restore the data structures,
                # indicating that candidate k is no longer
                # in the partial solution
                free_rows[k] = True
                free_pdiags[i - k + n - 1] = True
                free_sdiags[i + k] = True


def nqueens_wrapper(n):
    free_rows = [True] * n
    free_pdiags = [True] * (2 * n - 1)
    free_sdiags = [True] * (2 * n - 1)
    sol = [None] * n
    nqueens_all_sol(0, free_rows, free_pdiags, free_sdiags, sol)
```

Листинг 12.6. Поиск одного решения задачи *n* ферзей

```
def nqueens_one_sol(i, free_rows, free_pdiags,
                    free_sdiags, sol):
    n = len(sol)
    sol_found = False

    if i == n:
        return True
    else:
        k = 0
        while not sol_found and k < n:
            if (free_rows[k] and free_pdiags[i - k + n - 1]
                    and free_sdiags[i + k]):

                sol[i] = k

                free_rows[k] = False
                free_pdiags[i - k + n - 1] = False
                free_sdiags[i + k] = False

                sol_found = nqueens_one_sol(i + 1, free_rows,
                                            free_pdiags,
                                            free_sdiags, sol)

                free_rows[k] = True
                free_pdiags[i - k + n - 1] = True
                free_sdiags[i + k] = True

            k = k + 1

        return sol_found


def nqueens_one_sol_wrapper(n):
    free_rows = [True] * n
    free_pdiags = [True] * (2 * n - 1)
    free_sdiags = [True] * (2 * n - 1)
    sol = [None] * n

    if nqueens_one_sol(0, free_rows, free_pdiags,
                       free_sdiags, sol):
        print_chessboard(sol)


def print_chessboard(sol):
    for i in range(0, len(sol)):
         print(sol[i], ' ', end='')
    print()
```

Он очень похож на код для поиска всех решений. С одной стороны, метод-оболочка `nqueens_one_sol_wrapper` вызывает в условном операторе `if` рекурсивную функцию перебора с возвратами и выдаёт решение, как только она смогла его найти. Отметим, что список `sol` меняется (его можно считать параметром, который передается по ссылке) и по завершении вызова метода будет содержать решение задачи, если оно существует. Другой вариант – выдать решение в начальном условии рекурсивной функции перед возвратом из неё. С другой стороны, функция `nqueens_one_sol` сначала определяет логическую переменную `sol_found` с начальным значением `False` – признак того, что решение найдено. Цикл `for` заменён циклом `while`, чтобы завершить цикл, как только решение найдено. Тело цикла – то же, что в процедуре `nqueens_all_sol`, только результат рекурсивного вызова присваивается переменной `sol_found`, значение которой возвращается по завершении цикла `while`.

12.4. Задача о сумме элементов подмножества

Цель этой задачи – в том, чтобы для заданного множества S из n положительных целых чисел и заданного целого числа x найти в S такое подмножество, сумма элементов которого равна x. Формально, если s_i – i-е целое число множества S, то задача состоит в том, чтобы найти такое подмножество $T \subseteq S$, что:

$$\sum_{s_i \in T} s_i = x. \tag{12.1}$$

В данном случае мы создадим алгоритм перебора с возвратами, печатающий все такие подмножества. Например, если $S = \{1, 2, 3, 5, 6, 7, 9\}$ и $x = 13$, то метод должен вывести на экран следующие пять подмножеств: {6, 7}, {2, 5, 6}, {1, 5, 7}, {1, 3, 9} и {1, 2, 3, 7}.

Простейшее решение задачи «в лоб» – это генерация всех возможных подмножеств множества S с проверкой истинности требования (12.1). Такой полный перебор можно осуществить процедурой, подобной `generate_subsets` из листинга 12.1. Множество S было бы входным списком элементов, а метод имел бы дополнительный параметр, в котором хранится значение x. Кроме этого, процедура в начальном условии перед выводом подмножества на экран должна будет проверить ограничение (12.1). Иными словами, она должна проверять ограничение (12.1) в каждом листе дерева рекурсии.

Однако даже при использовании перебора с возвратами существует возможность ускорить поиск. В листинге 12.7 приводится одна из реализаций, где частичное решение (sol) представляет собой подмножество *T*.

Листинг 12.7. Решение задачи о сумме подмножества перебором с возвратами

```
def print_subset_sum(i, sol, psum, elements, x):
    # Base case
    if psum == x:
        print_subset_binary(sol, elements)
    elif i < len(elements)):
        # Generate candidates
        for k in range(0, 2):

            # Check if recursion tree can be pruned
            if psum + k * elements[i] <= x:

                # Expand partial solution
                sol[i] = k

                # Update sum related to partial solution
                psum = psum + k * elements[i]

                # Try to expand partial solution
                print_subset_sum(i + 1, sol, psum, elements, x)

                # not necessary:
                # psum = psum + k * elements[i]

        # Make sure a 0 indicates the absence of an element
        sol[i] = 0


def print_subset_sum_wrapper(elements, x):
    sol = [0] * (len(elements))
    print_subset_sum(0, sol, 0, elements, x)
```

Чтобы избежать вычисления суммы элементов частичного решения в каждом рекурсивном вызове, используется дополнительный параметр – переменная-накопитель суммы psum, которой, очевидно, в методе-оболочке нужно присвоить начальное значение 0. Ключевая идея алгоритма состоит в том, что дерево рекурсии можно сократить, если сумма элементов частичного решения равна *x*, когда выполнено тре-

бование (12.1), или больше x, когда добавление нового положительного целого числа из множества S к частичному решению только увеличивает его сумму (12.1).

Прежде всего метод проверяет начальное условие (строки 3 и 4), когда на экран выводится подмножество (частичное решение), удовлетворяющее ограничению (12.1). Это позволяет сокращать дерево рекурсии в его внутренних узлах глубины i. В начальном условии в T могут входить только первые i элементов S. Другими словами, значимыми являются лишь первые i элементов частичного решения. Если $n - i$ последних элементов могут принимать произвольные значения, то методу печати следовало бы передать значение i и указать «истинный» размер частичного решения. Но вместо этого алгоритм вызывает метод `print_subset_binary` из листинга 12.1, который требует, чтобы эти последние значения были всегда равны нулю, указывая на то, что они не относятся к частичному решению T. Это требование обеспечивается в методе-оболочке `print_subset_sum_wrapper` инициализацией частичного решения n нулями, что равносильно пустому множеству, и обновлением частичного решения таким образом, чтобы последние его $n - i$ элементов всегда оставались нулевыми при достижении начального условия (это реализуется присваиванием в строке 25).

Если частичное решение из n элементов не отвечает требованию (12.1), метод просто завершается, не выполняя никаких действий. В противном случае при выполнении условия $i < n$ (строка 5) он продолжает пополнять частичное решение. В этом случае он использует цикл, где проверяет два варианта – не включать (`k=0`) или включать (`k=1`) элемент s_i в частичное решение. Так как `k` – число, его можно использовать для вычисления суммы `psum + k*elements[i]` частичного решения. Обратите внимание, что при `k=0` она не меняется, тогда как при `k=1` она увеличивается на s_i. Поэтому перед включением элемента в частичное решение и очередным рекурсивным вызовом в строке 10 проверяется, не превысила ли новая сумма значения x. Если это так, то в частичное решение включается новый кандидат (строка 13), `psum` обновляется (строка 16) и выполняется рекурсивный вызов (строка 19). После всех этих действий нет необходимости восстанавливать значение `psum` (в строке 22), так как в первой итерации цикла при `k=0` значение `psum` не меняется. В завершение алгоритм обнуляет значение позиции `i` частичного решения, чтобы правильно вывести на экран частичное решение в начальном условии (последние его $n - i$ элементов всегда должны быть нулями).

На рис. 12.9 приведено дерево рекурсии метода `print_subset_sum` для множества $S = \{2, 6, 3, 5\}$ и $x = 8$. Частичные решения создаются на каж-

дом уровне, как в методах из раздела 12.2.1. Спускаясь по левой ветви дерева, алгоритм не включает элемент s_i в T, но включает его, спускаясь по правой ветви. Здесь числа рядом с узлами обозначают сумму включенных в частичные решения элементов (psum) при вызове метода. Обратите внимание, что дерево сокращается, как только найдено решение, удовлетворяющее условию (12.1), или сумма элементов в T превысила x.

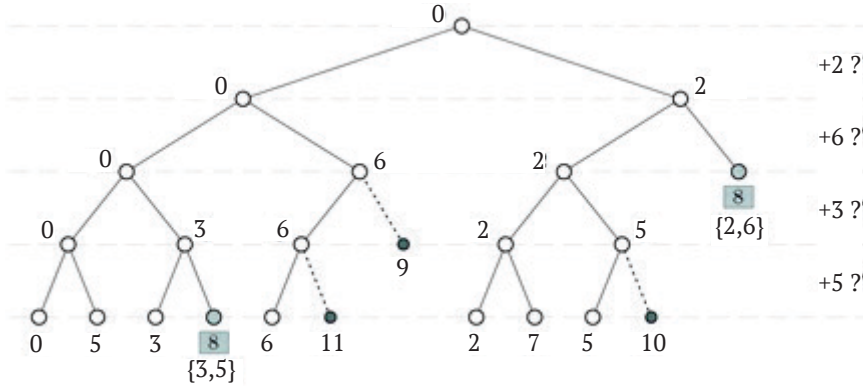


Рис. 12.9. Дерево рекурсии процедуры, решающей задачу суммы подмножества для множества $S = \{2, 6, 3, 5\}$ и $x = 8$

В итоге вызов `print_subset_sum_wrapper([2,6,3,5],8)` приводит к правильному результату:

```
{3, 5}
{2, 6}
```

В заключение отметим, что без обнуления в строке 25 частичное решение {2, 6} попадёт в начальное условие с единицей в своей третьей позиции, и метод выведет на экран {2, 6, 3}.

12.5. Путь в лабиринте

В этом разделе описывается алгоритм перебора с возвратами для поиска пути в лабиринте (рис. 12.10(а)), который представляет собой прямоугольный массив клеток – пустых и непустых. Пустые клетки – это проходы по лабиринту, непустые – это его стены или перегородки. Мы же представим его в виде списка списков **M** (состоящего из символов), где первый список в списке – это верхний ряд (слева направо) клеток лабиринта, второй список – следующий под ним ряд и т. д. Чтобы задать лабиринт, пользователю нужно записать каждый ряд клеток в виде строки

разделённых пробелом символов (см. рис. 12.10(b)), где 'E' обозначает пустую клетку, по которой можно пройти, а 'W' – стену. Поскольку большие лабиринты вводить вручную трудно, можно хранить массив символов в файле, который можно загрузить автоматически при выполнении программы (что также удобно для отладки). Кроме самого лабиринта, методу нужны параметры, задающие клетки входа в лабиринт и выхода из него. На рисунке это верхняя левая и нижняя правая клетки соответственно. Таким образом, в данном примере начальная клетка – это M[0][0].

Алгоритм перебора с возвратами выполняет полный перебор путей, начинающихся в начальной клетке и заканчивающихся в конечной клетке. Достигнув на каком-то промежуточном шаге некоторой клетки, алгоритм должен рассмотреть каждое из четырёх возможных направлений продолжения пути к соседней клетке – вверх, вниз, вправо или влево. Поскольку конечная клетка находится в нижнем правом углу лабиринта, алгоритм может начать поиск пути с попытки пойти сначала вниз, затем вправо, потом вверх и, наконец, влево. При таком порядке поиска пути метод выполнит шаги, показанные на рис. 12.10(c). Отметим, что, достигнув пустой клетки, метод сначала пытается продолжить путь вниз. Если это невозможно, он пытается сделать шаг вправо, затем вверх и, наконец, влево. Если это невозможно ни в одном из этих четырёх направлений, он должен «откатиться» назад к предыдущей клетке и попытаться найти другой путь. В нашем примере начальная клетка находится в позиции (0, 0) и соответствует строке 0 и столбцу 0. Метод помечает эту клетку как часть текущего пути, присваивая M[0][0] значение 'P'. Первый шаг возможен только в клетку (1, 0), которая тоже помечается символом 'P'. Из неё метод пытается продвинуться вниз, но натыкается на стену в клетке (2, 0). Поскольку дальше пути нет, алгоритм делает попытку шагнуть вправо, которая возможна, так как клетка (1, 1) пуста. Достигнув в какой-то момент клетки (6, 0), метод продолжит поиск пути с попытки пойти вниз. Этот путь приведёт его к тупиковой клетке (9, 0): снизу и справа – стены, слева – граница лабиринта, за пределы которого выходить нельзя, а сверху – клетка, заведшая метод в этот тупик. Поскольку алгоритм не может продвинуться ни в одном из направлений, он «откатывается» к клетке (8, 0), где он еще не пытался двигаться вправо, вверх или влево. Однако эти варианты тоже оказываются недопустимыми. То же самое происходит и после «отката» к клетке (7, 0). И только после возврата к клетке (6, 0) и исчерпывающего анализа всех возможных путей вниз от неё появляется возможность исследовать новые пути, ведущие от неё вправо. Этот процесс повторяется до тех пор, пока алгоритм не найдёт конечную клетку выхода из

лабиринта. Решение приведено на рис. 12.10(d), где все исследованные в процессе поиска выхода клетки затенены.

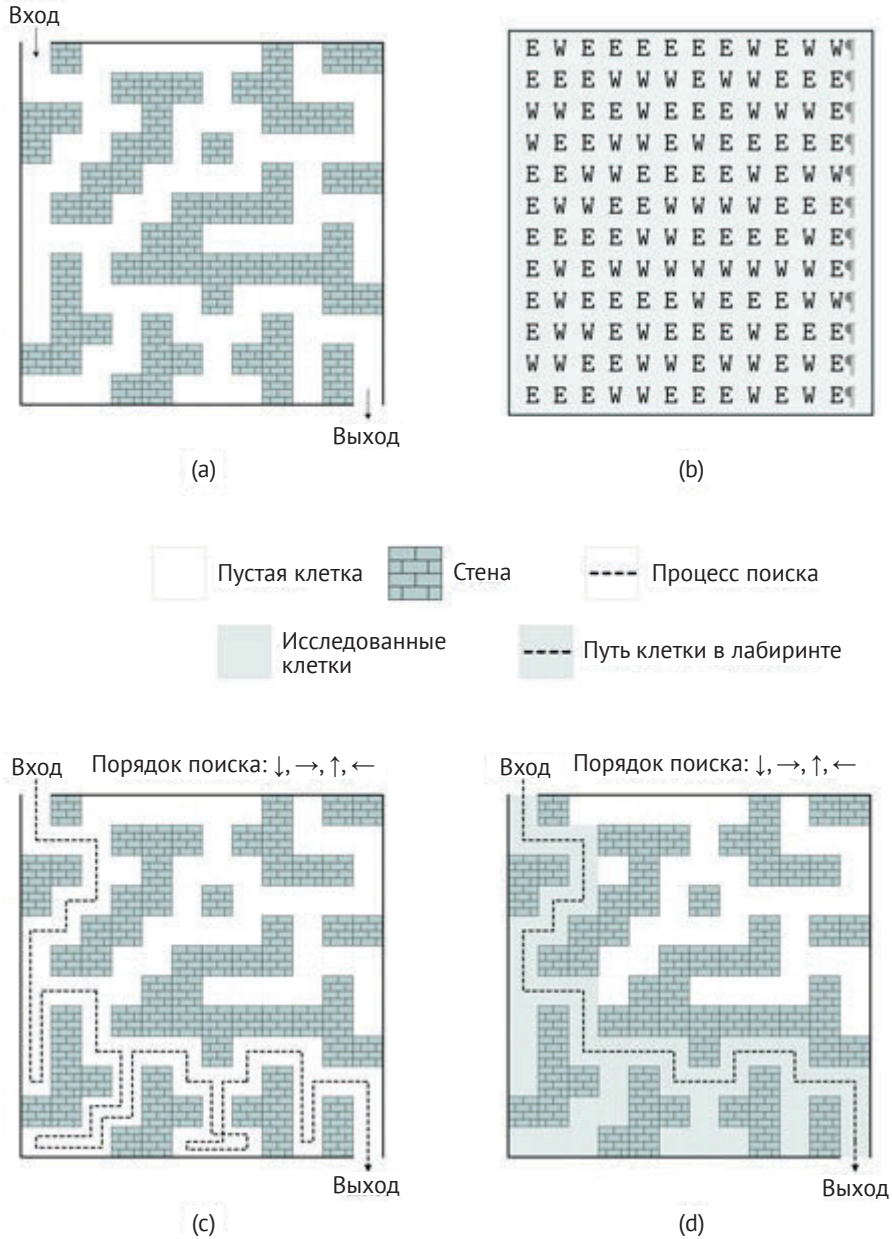


Рис. 12.10. Задача поиска пути в лабиринте и её решение перебором с возвратами при определённом порядке поиска

Алгоритм перебора с возвратами может либо прекратить поиск решений, когда найдено одно из них, либо продолжить поиск других возможных решений. Мы рассмотрим алгоритм, который останавливается, как только находит путь в лабиринте. В связи с этим нужно иметь в виду, что порядок рассмотрения возможных направлений движения может приводить к различным решениям, как показано на рис. 12.11. Обратите внимание, что множества исследуемых в каждом случае клеток различны.

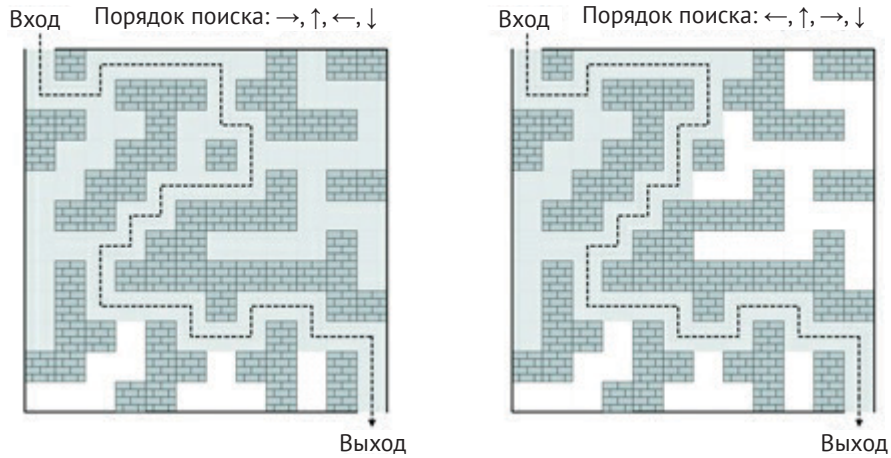


Рис. 12.11. Различные пути в лабиринте в зависимости от порядка поиска

Очевидно, что решение этой задачи не сводится к перестановкам клеток. Это видно хотя бы из того, что длина пути (то есть решение) непостоянна. Более того, хотя путь состоит из подмножества множества клеток лабиринта, эти подмножества являются упорядоченными. Значит, свести решение к генерации подмножеств множества тоже не удастся. По этим причинам мы представляем решение в виде символьной матрицы, в которой клетки определённого пути помечаются символом 'P'. Таким образом, частичное решение будет представлять собой исходный лабиринт, включающий путь от начальной клетки до некоторой пустой клетки, по достижении которой метод должен выбрать одну из четырёх клеток-кандидатов на включение её в частичное решение. Следовательно, мы приходим к рекурсивной декомпозиции задачи, показанной на рис. 12.12, где *r* и *c* обозначают конкретные строку и столбец соответственно. Каждый узел дерева рекурсии будет иметь четыре дочерних узла, соответствующих четырём возможным направлениям движения. При этом алгоритм, прежде чем выполнить рекурсивный

вызов с расширенным частичным решением, должен убедиться, что новая клетка пуста.

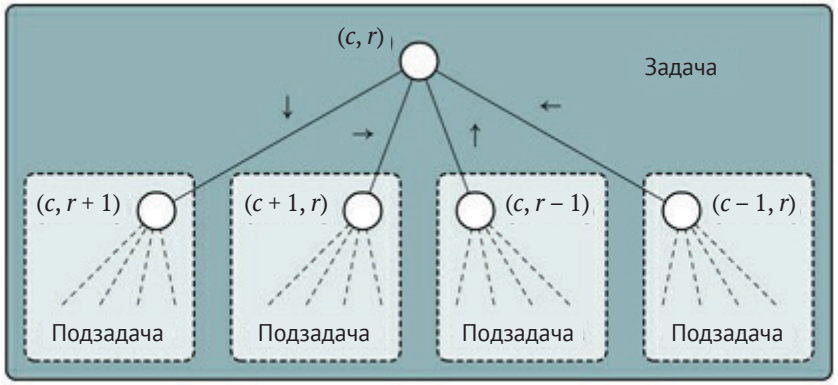


Рис. 12.12. Декомпозиция задачи поиска пути в лабиринте

В листинге 12.8 приводится одна из реализаций алгоритма решения задачи методом перебора с возвратами. Параметры метода-оболочки – исходный лабиринт и четыре целых числа, которые задают координаты начальной (`enter_row`, `enter_col`) и конечной (`exit_row`, `exit_col`) клеток лабиринта. Кроме того, в нём объявляется список `incr`, определяющий направления движения при поиске пути, а также порядок выбора этих направлений. Каждая пара (*x*, *y*) в списке задаёт приращение номера столбца *c* и номера строки *r* клетки для перемещения из неё в новую соседнюю клетку с координатами (*c* + *x*, *r* + *y*). Порядок следования пар в списке определяет последовательность выбора направления поиска в этом алгоритме: вниз, вправо, вверх и влево. Заметьте, что добавление единицы к номеру строки означает движение вниз (поскольку первая строка расположена наверху лабиринта), тогда как добавление единицы к столбцу – движение вправо. В заключение метод-оболочка включает пустую начальную клетку в искомый путь в качестве первого шага к цели и возвращает (логическое) значение результата вызова рекурсивного метода перебора с возвратами.

Первый параметр `M` рекурсивного метода `find_path_maze` – это одновременно и исходный лабиринт, и частичное решение. Следующие два параметра – координаты последней клетки частичного пути в `M`, из которой алгоритм попытается расширить частичный путь перемещением в одну из соседних клеток последней клетки. Последние три параметра метода – это `incr` и координаты конечной клетки выхода из лабиринта.

Листинг 12.8. Перебор с возвратами для поиска пути в лабиринте

```
def find_path_maze(M, row, col, incr, exit_row, exit_col):
    # Base case: check if path found
    if row == exit_row and col == exit_col:
        return True    # Solution found
    else:
        sol_found = False
        # Generate candidates
        k = 0
        while not sol_found and k < 4:

            # New candidate cell
            new_col = col + incr[k][0]
            new_row = row + incr[k][1]

            # Test candidate validity
            if (new_row >= 0 and new_row < len(M)
                and new_col >= 0 and new_col < len(M[0])
                and M[new_row][new_col] == 'E'):

                # Add to path (partial solution)
                M[new_row][new_col] = 'P'

                # Try to expand path starting at new cell
                sol_found = find_path_maze(
                            M, new_row, new_col, incr,
                            exit_row, exit_col)

                # Mark as empty if new cell not in solution
                if not sol_found:
                     M[new_row][new_col] = 'E'

            k = k + 1

        return sol_found


def find_path_maze_wrapper(M, enter_row, enter_col,
                           exit_row, exit_col):
    # search directions
    incr = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    M[enter_row][enter_col] = 'P'
    return find_path_maze(M, enter_row, enter_col, incr,
                           exit_row, exit_col)
```

Рекурсивная функция `find_path_maze` возвращает `True`, как только находит путь в лабиринте `M`. Она объявляет переменную `sol_found` с начальным значением `False`, которое меняется на `True` в начальном условии, когда алгоритм находит полное решение. В противном случае (случае рекурсивного условия) для генерации четырёх клеток-кандидатов она использует условный цикл `while`, который заканчивается, как только решение найдено. Переменные `new_col` и `new_row` определяют нового кандидата (строки 12 и 13), который сразу проверяется на правильность. В частности, он не должен выходить за пределы лабиринта (строки 16 и 17) и должен быть пустым (строка 18)[1]. Если клетка не нарушает ограничений задачи, метод присоединяет её к пути (строка 21) и в строках 24–26 вызывает сам себя, сохраняя результат вызова в `sol_found`. Если решение не найдено, метод возвращает ей прежнее пустое значение 'E', исключая её таким образом из пути (строка 30). Это действие необходимо, так как в полном решении пометку 'P' могут иметь только клетки, образующие путь через лабиринт. Без этого действия все исследуемые клетки содержали бы символ 'P'. По завершении цикла метод просто возвращает значение `sol_found` (строка 34).

Листинг 12.9 служит необходимым дополнением к основному листингу 12.8. В нём приведён вспомогательный код, запускающий и выполняющий основной код, а также заполняющий и рисующий лабиринт. Метод `read_maze_from_file` считывает текстовый файл с определением лабиринта и возвращает соответствующий ему список списков. Бóльшую часть кода занимает итерационная процедура `draw_maze`, которая рисует лабиринт, используя пакет `Matplotlib`. В последних строках кода лабиринт считывается из файла и рисуется, если путь от его начальной (верхней левой) до конечной (нижней правой) клетки найден.

12.6. Судоку

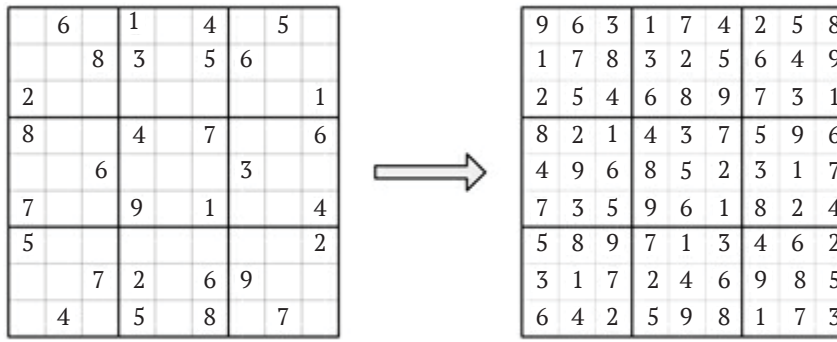
Головоломка судоку – из разряда так называемых задач удовлетворения ограничений. Её цель состоит в том, чтобы заполнить сетку 9×9 цифрами от 1 до 9 (или любыми другими девятью различными символами, так как их числовые значения не важны). Каждая из цифр может появиться только однажды в каждой строке, каждом столбце и каждой из девяти неперекрывающихся подсеток размером 3×3, называемых также блоками, которые покрывают сетку 9×9. На рис. 12.13 показан экземпляр задачи с частично заполненной исходными цифрами сеткой. Правиль-

[1] Если лабиринт по всему его периметру, за исключением входа и выхода, охватить стеной из непустых клеток, то строки 16 и 17 станут ненужными. – *Прим. перев.*

но поставленная задача должна иметь только одно решение (начальная сетка должна содержать не меньше 17 цифр).

Листинг 12.9. Вспомогательный код для поиска пути в лабиринте методом перебора с возвратами

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

def read_maze_from_file(filename):
    file = open(filename, 'r')
    M = []
    for line in file.readlines():
        M.append([x[0] for x in line.split(' ')])
    file.close()
    return M

gray = (0.75, 0.75, 0.75)
black = (0, 0, 0)
red = (0.75, 0, 0)
green = (0, 0.75, 0)

def draw_maze(M, enter_row, enter_col, exit_row, exit_col):
    nrows = len(M)
    ncols = len(M[0])
    fig = plt.figure()
    fig.patch.set_facecolor('white')
    ax = plt.gca()

    if enter_row is not None and enter_col is not None:
        ax.add_patch(Rectangle((enter_col, nrows - enter_row),
                               1, -1, linewidth=0, facecolor=green,
                               fill=True))
    if exit_row is not None and exit_col is not None:
        ax.add_patch(Rectangle((exit_col, nrows - exit_row),
                               1, -1, linewidth=0, facecolor=red,
                               fill=True))

    for row in range(0, nrows):
        for col in range(0, ncols):
            if M[row][col] == 'W':
                ax.add_patch(Rectangle((col, nrows - row), 1, -1,
                                       linewidth=0, facecolor=gray))
            elif M[row][col] == 'P':
                circ = plt.Circle((col + 0.5, nrows - row - 0.5),
                                  radius=0.15, color=black, fill=True)
                ax.add_patch(circ)

    ax.add_patch(Rectangle((0, 0), ncols, nrows, edgecolor=black,
                           fill=False))
    plt.axis('equal')
    plt.axis('off')
    plt.show()

M = read_maze_from_file('maze_01.txt')     # some file
# Enter at top-left, exit bottom-right
if find_path_maze_wrapper(M, 0, 0, len(M) - 1, len(M[0]) - 1):
    draw_maze(M, 0, 0, len(M) - 1, len(M[0]) - 1)
```



	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Задача судоку

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Решение

Рис. 12.13. Задача судоку и ее решение

Для представления сетки судоку мы будем использовать список списков цифр, где цифра 0 будет обозначать пустую клетку. Частичные решения будут частично заполненными сетками, а алгоритм должен неким образом пронумеровать все клетки сетки, чтобы можно было расширять частичные решения новыми цифрами. Приводимый ниже алгоритм будет расширять частичные решения в порядке следования строк, начав добавлять цифры-кандидаты с верхнего ряда сетки слева направо и так далее для каждой последующей строки. Таким образом, алгоритм найдёт решение, если сумеет поместить правильную цифру в последнюю клетку последней строки.

В рекурсивном условии метод генерирует девять возможных кандидатов для заполнения пустой клетки. Таким образом, каждый узел дерева рекурсии мог бы иметь до девяти дочерних узлов, как показано на рис. 12.14(a). Однако у этой задачи есть особенность, которая не встречалась ни в одной из предыдущих задач: присутствие в частичном решении (сетке) заранее заданных фиксированных элементов (клеток с цифрой). Это значит, что при обработке клетки с цифрой алгоритм должен просто пропустить её (без расширения частичного решения) и выполнить рекурсивный вызов, обрабатывающий следующую клетку. Это требует включения второго рекурсивного условия, которое приведено на рис. 12.14(b).

В листингах 12.10 и 12.11 приводится решающая задачу процедура перебора с возвратами и несколько вспомогательных функций. Рекурсивная процедура `solve_sudoku_all_solutions` допускает в том числе и неправильную постановку задачи, когда она может иметь несколько решений или даже ни одного. Таким образом, процедура выдаёт все правильные решения задачи для заданной сетки судоку.

Например, если верхний ряд судоку на рис. 12.13 заменить пустой строкой, то существует 10 различных способов заполнения цифрами сетки судоку.

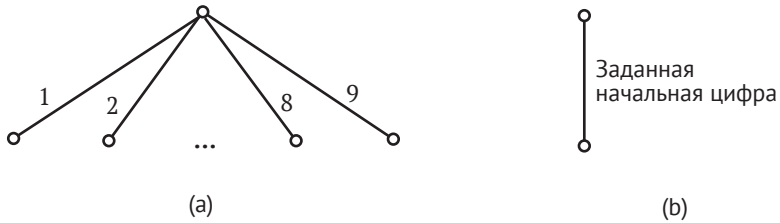


Рис. 12.14. Рекурсивные условия при решении судоку

Параметры рекурсивной процедуры – пара координат (`row` и `col`) клетки, куда она пытается внести цифру для расширения частичного решения, а также само частичное решение *S*, представленное списком списков цифр. Так как процедура расширяет частичное решение построчно, начиная с клетки (0, 0), правильное решение достигается при `row = 9`. При выполнении начального условия она просто выводит сетку судоку (строка 4). В противном случае метод проверяет, не заполнена ли текущая клетка, то есть не содержит ли она одну из заранее заданных фиксированных цифр. Если да, то такая клетка пропускается и процедура вызывает себя (строка 14) с координатами (вычисленными в строке 11) следующей по порядку клетки.

Во втором рекурсивном условии процедура использует цикл для генерации девяти возможных кандидатов на включение в пустую клетку. Затем в строке 20 она проверяет, можно ли включить кандидата `k` в клетку (`row`, `col`) частичного решения `S`. Если можно, то процедура включает кандидата в частичное решение (строка 23) и продолжает работу, выполняя рекурсивный вызов со следующей клеткой (строка 29). По завершении цикла процедура должна отменить произведённые в клетке изменения, сделав её пустой (строка 32). Это необходимо для последующего анализа решений. Отметим, что клетка должна быть пустой, чтобы алгоритм мог снова обработать её после отката к одной из предыдущих клеток.

Вспомогательные функции: a) функция, возвращающая пару координат следующей по порядку клетки (см. листинг 12.10); b) функция, определяющая, нарушает ли размещённая в некоторой строке и столбце цифра ограничения задачи, где переменные `box_row` и `box_col` задают верхнюю левую клетку блока 3×3; c) функция, считывающая сетку судоку из текстового файла, каждая строка которого содержит девять

строк цифр исходного судоку, разделённых символом пробела; d) метод для вывода сетки; e) код, считывающий и решающий судоку.

Листинг 12.10. Решение задачи судоку

```
def solve_sudoku_all_sols(row, col, S):
    # Check if sudoku is complete
    if row == 9:
        print_sudoku(S)   # print the completed sudoku
        print()
    else:
        # Check if digit S[row][col] is an initial fixed symbol
        if S[row][col] != 0:

           # Advance to a new cell in row-major order
           (new_row, new_col) = advance(row, col)

           # Try to expand the partial solution
           solve_sudoku_all_sols(new_row, new_col, S)
       else:
           # Generate candidate digits
           for k in range(1, 10):

               # Check if digit k is a valid candidate
               if is_valid_candidate(row, col, k, S):

                  # Include digit in cell (row, col)
                  S[row][col] = k

                  # Advance to a new cell in row-major order
                  (new_row, new_col) = advance(row, col)

                  # Try to expand the partial solution
                  solve_sudoku_all_sols(new_row, new_col, S)

           # Empty cell (i, j)
           S[row][col] = 0


# Compute the next cell in row-major order
def advance(row, col):
    if col == 8:
        return (row + 1, 0)
    else:
        return (row, col + 1)
```

Листинг 12.11. Вспомогательные функции для решения задачи судоку

```
import math

# Check if the digit at cell (row, col) is valid
def is_valid_candidate(row, col, digit, S):
    # Check conflict in column
    for k in range(0, 9):
        if k != col and digit == S[row][k]:
            return False

    # Check conflict in row
    for k in range(0, 9):
        if k != row and digit == S[k][col]:
            return False

    # Check conflict in box
    box_row = math.floor(row / 3)
    box_col = math.floor(col / 3)
    for k in range(0, 3):
        for n in range(0, 3):
            if (row != 3 * box_row + k
                    and col != 3 * box_col + m):
                if digit == S[3 * box_row + k][3 * box_col + m]:
                    return False

    return True

# Read a sudoku grid from a text file
def read_sudoku(filename):
    file = open(filename, 'r')
    S = [[None] * 9] * 9
    i = 0
    for line in file.readlines():
        S[i] = [int(x) for x in line.split(' ')]
        i = i + 1
    file.close()
    return S

# Print a sudoku grid on the console
def print_sudoku(S):
    for s in S:
        print (*s)

S = read_sudoku('sudoku01_input.txt')       # Some file
solve_sudoku_all_sols(0, 0, S)
```

12.7. Задача о рюкзаке 0–1

Алгоритмы перебора с возвратами применяются также для решения задач оптимизации. В задаче о рюкзаке 0–1 у нас есть множество из n предметов со стоимостями v_i и весами w_i, где $i = 0, \ldots, n-1$. Цель задачи – найти такое подмножество предметов, чтобы их суммарный вес не превышал грузоподъёмности C рюкзака, а суммарная стоимость была максимальной. Формально задачу оптимизации можно записать так: максимизировать $\sum_{i=0}^{n-1} x_i v_i$ при условии, что $\sum_{i=0}^{n-1} x_i w_i \leqslant C$, где $x_i \in \{0, 1\}$ для $i = 0, \ldots, n-1$.

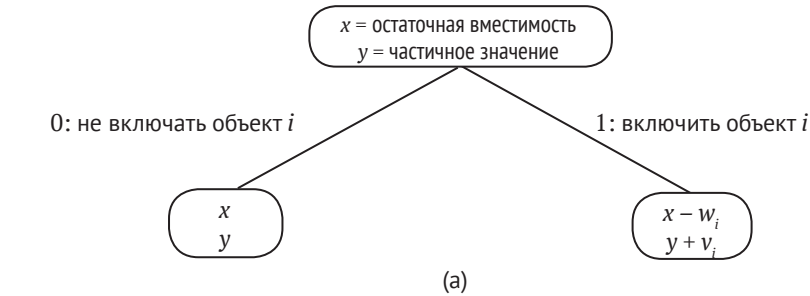
Вектор или список $\mathbf{x} = [x_1, \ldots, x_n]$ – переменная величина задачи (её частичное решение), компоненты которой могут принимать значения 0 или 1. В частности, $x_i = 1$ означает, что предмет под номером i помещён в рюкзак. Таким образом, **x** играет ту же роль, что и двоичный список в задаче генерации подмножеств или в задаче о сумме подмножества. Сумма $\sum_{i=0}^{n-1} x_i v_i$ называется целевой функцией и просто суммирует стоимости находящихся в рюкзаке предметов, а ограничение $\sum_{i=0}^{n-1} x_i w_i \leqslant C$ означает, что сумма их весов не может превышать грузоподъёмности C рюкзака.

В следующих подразделах описываются два подхода, выполняющих полный перебор для поиска *оптимального решения*. Первым является стандартный алгоритм перебора с возвратами, который сокращает дерево рекурсии, когда частичное решение не удовлетворяет ограничениям задачи (то есть сумма весов превышает грузоподъёмность рюкзака). Второй подход использует широко известный *метод ветвей и границ*, тесно связанный с методом перебора с возвратами. Метод также улучшает поиск за счёт сокращения дерева рекурсии, но только тогда, когда обнаруживает, что не в состоянии улучшить путём расширения некоторого частичного решения лучшее решение, найденное на предыдущих шагах.

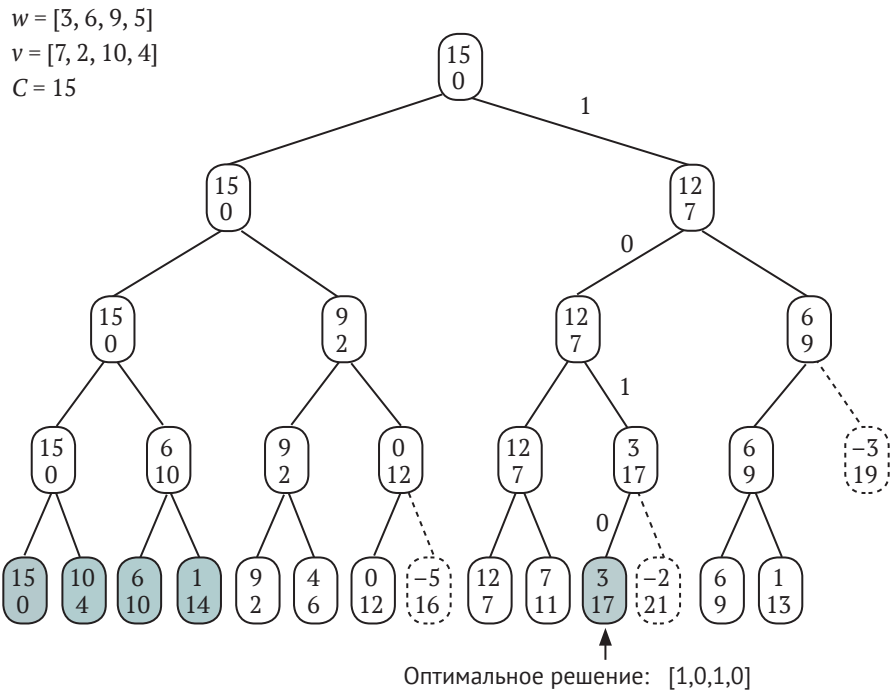
12.7.1. Стандартный алгоритм перебора с возвратами

Поскольку решением задачи является подмножество множества из n предметов, можно разработать алгоритм, дерево рекурсии которого подобно двоичным деревьям рекурсии алгоритмов генерации подмножеств или сумм подмножеств. На рис. 12.15 приводится дерево рекурсии

для весов **w** = [3, 6, 9, 5], стоимостей **v** = [7, 2, 10, 4] и грузоподъёмности рюкзака C = 15, которые будут входными параметрами рекурсивного метода.



(a)



(b)

Рис. 12.15. Дерево рекурсии алгоритма перебора с возвратами для задачи о рюкзаке 0–1

Узлы дерева – это вызовы метода с конкретным частичным решением в качестве параметра, а числа в узлах – это весовой остаток рюк-

зака и суммарная стоимость уже помещённых в него предметов (см. рис.12.15(a)). Для каждого индекса i в частичном решении алгоритм либо не помещает (левый потомок узла), либо помещает (правый потомок узла) предмет в рюкзак. Если предмет не помещён в рюкзак, то обе величины в дочернем узле не меняются. Если же предмет помещён в рюкзак, то весовой остаток рюкзака уменьшается на w_i, а его суммарная стоимость увеличивается на v_i.

Поскольку алгоритм решает задачу оптимизации, он должен хранить найденное на предыдущих шагах лучшее решение, чтобы обновлять его при обнаружении ещё лучшего решения.

Полное дерево рекурсии приведено на рис. 12.15(b), где затемнённые узлы обозначают вызовы метода, которые обновляют это лучшее решение. Если оптимальной суммарной стоимости рюкзака сразу присвоить отрицательное начальное значение, то метод обновит оптимальное решение уже в первом листе дерева. В этом случае частичное решение будет пустым множеством, а суммарная стоимость – нулевой. Продолжив обход дерева, метод обновит лучшее решение после добавления в рюкзак самого последнего предмета ($w_3 = 5$ и $v_3 = 4$) так, что весовой остаток рюкзака в листовом узле будет равен 10, а общая стоимость частичного решения – 4. В процессе решения задачи лучшее решение будет обновлено ещё трижды для суммарных стоимостей 10, 14 и 17. Последнее обновление, естественно, даст оптимальное решение задачи. В данном случае это подмножество, состоящее из первого и третьего предметов, а максимальная сумма стоимостей – 17. В заключение важно отметить, что метод сокращает двоичное дерево рекурсии в тех случаях, когда весовой остаток рюкзака для частичного решения (и узла) – отрицательный.

В листинге 12.12 приведена возможная реализация алгоритма перебора с возвратами. Во-первых, подобно другим методам, его входные параметры включают частичное решение `sol` и индекс `i` предмета, который можно поместить в рюкзак.

Метод получает ещё списки весов `w` и стоимостей `v`, а также грузоподъёмность рюкзака `C`. По этим параметрам можно на любом шаге вычислить и весовой остаток, и суммарную стоимость предметов в рюкзаке. Но гораздо эффективнее ввести для хранения этих величин два дополнительных параметра `w_left` – для весового остатка и `current_v` – для суммарной стоимости предметов. Их преимущество в том, что они легко и быстро обновляются, не требуя дополнительных вычислений.

Листинг 12.12. Алгоритм решения задачи о рюкзаке 0–1 методом перебора с возвратами

```
def knapsack_0_1(i, w_left, current_v, sol,
                 opt_sol, opt_v, w, v, C):
    # Check base case
    if i == len(sol):
        # Check if better solution has been found
        if current_v > opt_v:
            # Update optimal value and solution
            opt_v = current_v
            for k in range(0, len(sol)):
                opt_sol[k] = sol[k]
    else:
        # Generate candidates
        for k in range(0, 2):

            # Check if recursion tree can be pruned
            if k * w[i] <= w_left:

                # Expand partial solution
                sol[i] = k

                # Update remaining capacity and partial value
                new_w_left = w_left - k * w[i]
                new_current_v = current_v + k * v[i]

                # Try to expand partial solution
                opt_v = knapsack_0_1(i + 1, new_w_left,
                                     new_current_v, sol,
                                     opt_sol, opt_v, w, v, C):

    # return value of optimal solution found so far
    return opt_v


def knapsack_0_1_wrapper(w, v, C):
    sol = [0] * (len(w))
    opt_sol = [0] * (len(w))
    total_v = knapsack_0_1(0, C, 0, sol, opt_sol, -1, w, v, C)
    print_knapsack_solution(opt_sol, w, v, C, total_v)
```

Кроме того, метод во время выполнения сохраняет лучшее из найденных решений. Оно хранится в параметре `opt_sol`, который являет-

ся копией того частичного решения, которое получено по достижении листа дерева рекурсии. Наконец, `opt_v` содержит оптимальную суммарную стоимость `opt_sol`. Этот параметр тоже избыточен в том смысле, что его значение можно вычислить по `opt_sol`, но ради дополнительной эффективности его значение лучше передавать как параметр.

Как обычно, рекурсивная функция начинается с проверки начального условия. Если метод достиг полного решения (строка 4), он в строке 6 сравнивает стоимости текущего `sol` и лучшего `opt_sol` решений, полученных на предыдущих шагах. Если стоимость `sol` превышает стоимость `opt_sol`, метод обновляет `opt_v` и `opt_sol`. Позже, в строке 31, метод вернёт в качестве своего результата `opt_v`. Значения списка `opt_sol` тоже будут доступны по завершении метода, поскольку этот список передаётся в метод по ссылке и, следовательно, меняется в процессе поиска решения.

В рекурсивном условии метод в строке 13 генерирует кандидатов (со значениями 0 или 1) для включения в двоичный список частичного решения. Затем, с целью сократить дерево рекурсии, он проверяет условие `k*w[i] <= w_left`. Если $k = 0$, предмет не добавляется в рюкзак, и условие всегда истинно, так как `w_left` – неотрицательная величина. Если же $k = 1$, алгоритм добавляет предмет в рюкзак, если это возможно, то есть если w_i не превосходит весового остатка рюкзака. После чего он, предварительно обновив частичное решение, весовой остаток и суммарную стоимость предметов рюкзака, выполняет рекурсивный вызов с этими новыми параметрами и, сохранив результат в `opt_v`, завершается в строке 31 возвратом этого результата.

Фнкция-оболочка `knapsack_0_1_wrapper` инициализирует i значением 0, `w_left` – грузоподъёмностью рюкзака `C` и `current_v` – значением 0, поскольку рюкзак изначально пуст. Максимальное значение `opt_v` может быть нулём или некоторым отрицательным числом. На рис. 12.15 мы посчитали его отрицательным с расчётом на то, что метод всё равно обновит его даже при пустом оптимальном решении (в крайнем левом листе).

В заключение в листинге 12.13 приведён простой итерационный код, который может использоваться для печати решения задачи.

Стоимость решения можно вычислить по частичному решению и списку стоимостей предметов. Однако метод получает это значение в качестве параметра, так как оно возвращается функцией `knapsack_0_1` и, следовательно, доступно в `knapsack_0_1_wrapper`. В последних строках листинга алгоритму ставится конкретная задача, и он находит её решение.

Листинг 12.13. Вспомогательный код для задачи о рюкзаке 0–1

```
def print_knapsack_solution(sol, w, v, C, opt_value):
    n = len(sol)
    k = 0
    while k < n and sol[k] == 0:
         k = k + 1

    total_weight = 0
    if k < n:
        print('(', w[k], ', ', v[k], ')', sep='', end='')
        total_weight = total_weight + w[k]

            for i in range(k + 1, n):
                if sol[i] == 1:
                   total_weight = total_weight + w[i]
                   print(' + ', sep='', end='')
                   print('(', w[i], ', ', v[i], ')', sep='', end='')

    print(' => ', '(', total_weight,
          ', ', opt_value, ')', sep='')


w = [3, 6, 9, 5]      # List of object weights
v = [7, 2, 10, 4]      # List of object values
C = 15      # Weight capacity of the knapsack
knapsack_0_1_wrapper(w, v, C)
```

12.7.2. Алгоритм ветвей и границ

Алгоритм ветвей и границ можно считать разновидностью алгоритма перебора с возвратами, который выполняет такой же полный, но более эффективный перебор вариантов решения дискретных и комбинаторных задач оптимизации. Его идея заключается в использовании границ для сокращения дерева рекурсии не только тогда, когда частичное решение не удовлетворяет ограничениям задачи, но и когда ясно, что расширение частичного решения не улучшит решения, найденного на предыдущих шагах.

Суть метода ветвей и границ, который мы применим к задаче о рюкзаке 0–1, – в использовании дополнительного параметра для хранения максимальной суммарной стоимости, которая получается при расширении частичного решения. Если при вызове некоторого метода это значение хуже лучшего, найденного на предыдущих шагах, то алгоритм

не будет расширять частичное решение, поскольку не сможет улучшить его. Это позволяет сократить ещё больше узлов дерева рекурсии, что может привести к значительно более эффективному поиску.

На рис. 12.16 приведено дерево рекурсии для тех же весов, стоимостей и грузоподъёмности рюкзака, что на рис. 12.15.

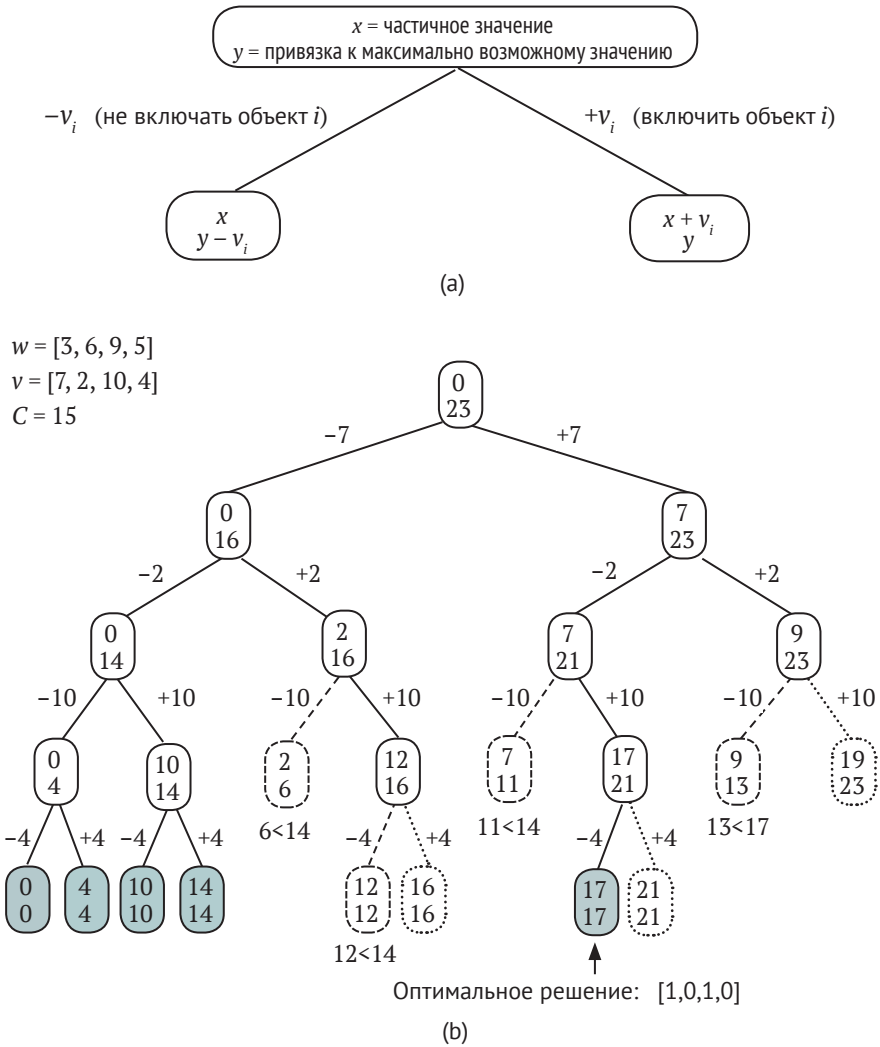


Рис.12.16. Дерево рекурсии алгоритма ветвей и границ для решения задачи о рюкзаке 0–1

Но в этом случае числа в узлах указывают частичную стоимость и границу максимально возможной стоимости, которую мы могли бы полу-

чить, расширив связанное с ней частичное решение, как показано на рис. 12.16(a). Первоначально частичная стоимость равна 0, а граница, как показано на рис. 12.16(b), равна сумме всех стоимостей предметов (то есть 23 = 7 + 2 + 10 + 4), так как сначала мы должны рассмотреть случай, когда все предметы помещаются в рюкзаке.

Каждый внутренний узел глубины i может иметь два дочерних узла. Спуск по левой ветви означает, что предмет i не добавляется в рюкзак. Поэтому частичная стоимость не меняется, но граница при этом уменьшается на v_i, поскольку стоимость этого предмета уже не может входить в суммарную стоимость полного решения. И наоборот, спуск по правой ветви означает, что предмет добавляется в рюкзак и его стоимость v_i добавляется к частичной стоимости, но граница при этом остаётся неизменной.

Подобно примеру на рис. 12.15, метод обновляет лучшее решение, найденное ранее (затемнённые листья дерева). Кроме того, обведённые крупным пунктиром узлы обозначают вызовы метода, которые не выполняются, так как сумма весов предметов превышает грузоподъёмность рюкзака. Помимо этого, на рисунке есть обведённые мелким пунктиром узлы, которые алгоритм также не вызывает, поскольку граница их стоимости меньше лучшей из встречавшихся ранее. Например, лучшая стоимость по достижении четвёртого листа – 14. После этого нужно рассмотреть узел с частичной стоимостью 2 и границей 16. Отказавшись от третьего предмета, алгоритм должен уменьшить границу на 10, а это означает, что максимально возможная суммарная стоимость, которую можно получить при расширении частичного решения, равна 16 – 10 = 6. Поскольку 6 < 14 (это отношение изображено под узлом), алгоритм не вызывает метода, сокращая тем самым дерево рекурсии. Обратите внимание, что дерево рекурсии этого алгоритма имеет ту же структуру, что на рис. 12.15, но содержит меньше узлов, так как сокращает дерево в большем количестве случаев. На практике такое усовершенствование может оказать существенное влияние на эффективность.

В листинге 12.14 приведён код решения задачи методом ветвей и границ, который во многом схож с кодом листинга 12.12. Основное различие между ними – новый (четвёртый) параметр `max_v`, в котором сохраняется граница. Отметим, что он инициализируется в методе-оболочке суммарной стоимостью всех предметов. В рекурсивном условии метод, проверив правильность частичного решения с новым кандидатом (строка 17), вычисляет новое значение границы `new_max_v` (строка 20). Обратите внимание, что при $k = 0$ новая граница уменьшается на v_i, тог-

да как при $k = 1$ она остаётся неизменной. После этого метод проверяет, можно ли сократить дерево с новой границей и вычисленной ранее оптимальной стоимостью решения (строка 24). Остальная часть кода аналогична алгоритму перебора с возвратами.

Листинг 12.14. Алгоритм ветвей и границ для решения задачи рюкзака 0–1

```
1  def knapsack_0_1_bnb(i, w_left, current_v, max_v, sol,
2                       opt_sol, opt_v, w, v, C):
3      # Check base case
4      if i == len(sol):
5          # Check if better solution has been found
6          if current_v > opt_v:
7               # Update optimal value and solution
8               opt_v = current_v
9               for k in range(0, len(sol)):
10                  opt_sol[k] = sol[k]
11     else:
12         # Generate candidates
13         for k in range(0, 2):
14
15             # Check if recursion tree can be pruned
16             # according to (capacity) constraint
17             if k * w[i] <= w_left:
18
19                 # Update maximum possible value
20                 new_max_v = max_v - (1 - k) * v[i]
21
22                 # Check if recursion tree can be pruned
23                 # according to optimal value
24                 if new_max_v > opt_v:
25
26                     # Expand partial solution
27                     sol[i] = k
28
29                     # Update remaining capacity
30                     # and partial value
31                     new_w_left = w_left - k * w[i]
32                     new_current_v = current_v + k * v[i]
33
34                     # Try to expand partial solution
35                     opt_v = knapsack_0_1_bnb(i + 1, new_w_left,
36                                              new_current_v,
37                                              new_max_v, sol,
38                                              opt_sol, opt_v,
39                                              w, v, C):
40
41     # return value of optimal solution found so far
42     return opt_v
```

Наконец, в листинге 12.15 приводится метод-оболочка, решающий задачу для конкретных значений **w**, **v** и *C*.

Листинг 12.15. Метод-оболочка алгоритма ветвей и границ для решения конкретной задачи рюкзака 0–1

```
def knapsack_0_1_branch_and_bound_wrapper(w, v, C):
    sol = [0] * (len(w))
    opt_sol = [0] * (len(w))
    total_v = knapsack_0_1_bnb(0, C, 0, sum(v) sol,
                               opt_sol, -1, w, v, C)
    print_knapsack_solution(opt_sol, w, v, C, total_v)


w = [3, 6, 9, 5]      # List of object weights
v = [7, 2, 10, 4]      # List of object values
C = 15      # Weight capacity of the knapsack
knapsack_0_1_branch_and_bound_wrapper(w, v, C)
```

12.8. Упражнения

Упражнение 12.1. Есть разные способы построения алгоритмов генерации всех подмножеств множества. Описанные в разделе 12.2.1 методы опирались на двоичные деревья рекурсии. Цель этого упражнения состоит в том, чтобы реализовать альтернативные процедуры, которые также выводят все подмножества множества из n элементов, представленного списком. Однако они должны генерировать подмножество в каждом узле дерева рекурсии, приведённого на рис. 12.17.

Отметим, что дерево имеет ровно 2^n узлов, а метки 0, 1 и 2 представляют собой индексы элементов исходного списка [a, b, c]. В связи с этим для указания вхождения элементов в подмножество частичное решение должно быть не двоичным списком, а списком индексов, входящих в подмножество элементов. Например, подмножество {a, c} должно представляться списком [0, 2]. В этом случае каждое частичное решение будет ещё и полным.

Упражнение 12.2. Реализуйте алгоритм перебора с возвратами, который выводит все решения задачи n ладей. Он похож на задачу n ферзей, но вместо ферзей используются ладьи. Поскольку ладьи могут ходить только по вертикали и горизонтали, в каждой строке и каждом столбце может находиться только одна ладья. При этом на одной диа-

гонали может стоять несколько ладей. На рис. 12.18 приведено одно из решений задачи.

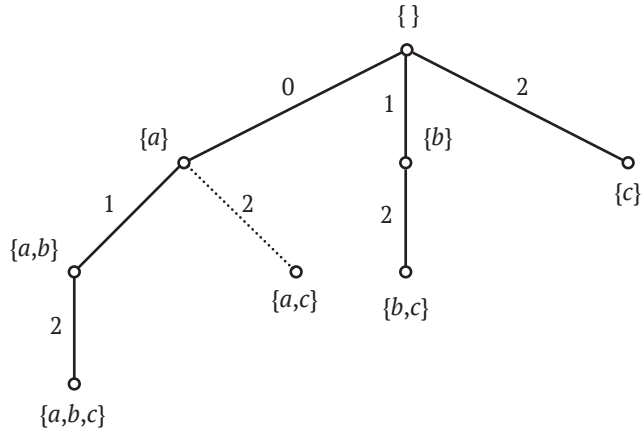


Рис. 12.17. Дерево рекурсии альтернативного алгоритма генерации всех подмножеств множества из трёх элементов

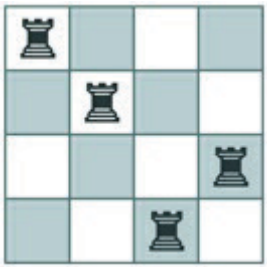


Рис. 12.18. Одно из решений задачи четырёх ладей

Упражнение 12.3. Реализуйте функцию перебора с возвратами, подсчитывающую количество правильных решений неточно поставленной задачи судоку (то есть задачи с одним и более решениями или вовсе без них).

Упражнение 12.4. Магический квадрат – это сетка размером $n \times n$, заполненная первыми n^2 положительными целыми числами так, что суммы чисел в каждой строке, столбце и диагонали одинаковы. Эта сумма называется «магической константой» и равна $n(n^2 + 1)/2$ для любого n. На рис. 12.19 приведён пример магического квадрата размера 3×3 с магической константой 15.

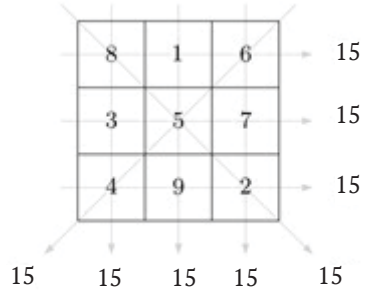
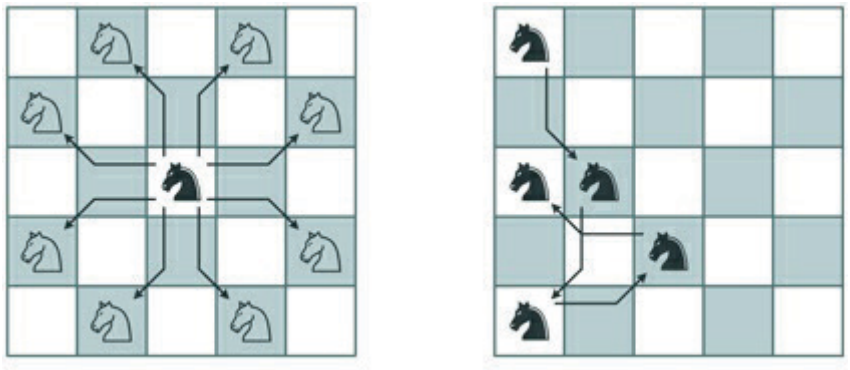


Рис. 12.19. Магический квадрат 3×3

Разработайте алгоритм перебора с возвратами, который выводит все возможные магические квадраты размера 3×3. Примечание: разрабатывать алгоритм решения этой задачи для любого n не нужно. С какой проблемой можно столкнуться при решении этой задачи для бóльших значений n?

Упражнение 12.5. Шахматный конь ходит с одного поля на другое буквой Г, как показано на рис. 12.20(a). На рис. 12.20(b) приведены 4 последовательных хода коня, начиная с левой верхней клетки.



(a) (b)

Рис. 12.20. Ходы шахматного коня

Задача «обхода конём» заключается в определении маршрута шахматного коня из заданного поля по всем остальным полям шахматной доски размера $n \times n$ с однократным их посещением. Реализуйте алгоритм перебора с возвратами, осуществляющий такой обход, и проверьте его для $n = 5$ и $n = 6$. Кроме n, метод должен получить координаты начального поля. Маршрут не обязательно должен быть «замкнутым», то есть заканчивающимся в начальном поле.

Упражнение 12.6. «Задача коммивояжёра» – это классическая дискретная задача оптимизации. Её цель – найти кратчайший замкнутый маршрут по n городам из заданного отправного пункта с однократным посещением каждого города. На рис. 12.21 показан кратчайший путь для 10 городов.

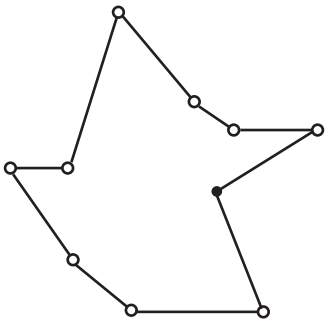


Рис. 12.21. Пример задачи коммивояжёра и её решение

Реализуйте алгоритм перебора с возвратами для решения этой задачи. Будем считать, что местоположения городов задаются в текстовом файле, каждая строка которого содержит координаты x и y города на плоской карте. Этот файл можно прочитать методом `loadtxt` из пакета `NumPy`. Отправным пунктом будем считать первый город в файле. Кроме того, будем считать, что коммивояжёр перемещается от города к городу строго по прямой линии. В этом случае можно (используя, например, метод `pdist` из пакета `SciPy`) заранее вычислить матрицу расстояний между городами, которая будет передаваться в рекурсивный метод перебора с возвратами, делая ненужными местоположения городов. В заключение проверьте полученный код для $n \leqslant 10$.

Упражнение 12.7. Следующая оптимизационная задача разбиения множества чисел известна под названием «перетягивание каната». В ней непустое множество из n чисел, где n – чётное, нужно разбить на два подмножества по $n/2$ элементов так, чтобы разность между суммами чисел в каждом из подмножеств была минимальной. Например, для множества {3, 5, 9, 14, 20, 24} оптимальное разделение приводит к двум подмножествам {5, 9, 24} и {3, 14, 20}, суммы элементов которых, соответственно, – 38 и 37, а разность между этими суммами – 1.

Для решения данной задачи реализуйте алгоритм перебора с возвратами, основанный на генерации подмножеств. Исходное множество представлено списком из n чисел, а решение – двоичным списком длины n с $n/2$ нулями и $n/2$ единицами. Позиции нулей/единиц в списке со-

ответствуют позициям элементов подмножества в исходном множестве. Для приведённого выше примера решением будет список [0, 1, 1, 0, 0, 1], представляющий подмножество {5, 9, 24}, или равносильный ему список [1, 0, 0, 1, 1, 0]), представляющий подмножество {3, 14, 20}.

Дополнительно реализуйте более эффективную стратегию, где решение представляется списком **s** длины $n/2$, значения элементов которого располагаются в возрастающем порядке и являются позициями элементов подмножества в исходном множестве (списке). Например, подмножеству {5, 9, 24} будет соответствовать список [1, 2, 5]. На рис. 12.22 приведены оба способа представления решения.

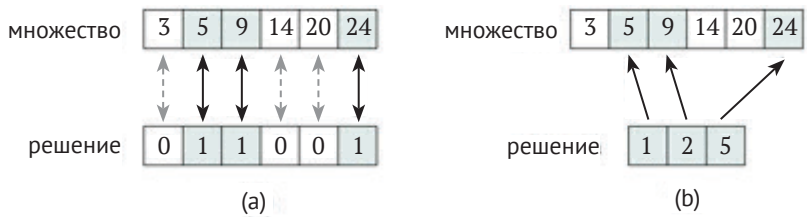


Рис. 12.22. Два способа представления решения задачи «перетягивание каната»

Упражнение 12.8. В рассмотренной в разделе 12.4 задаче нужно было для заданного множества S из n положительных целых чисел вывести все подмножества T, сумма элементов которых равна некоторому целому значению x. Разработайте альтернативную рекурсивную функцию, которая вместо этого выводит (правильное) подмножество T с наименьшей мощностью (количеством элементов). В частности, она должна сохранить в списке оптимальное подмножество и вернуть оптимальное количество элементов. Например, для $S = \{1, 2, 3, 5, 6, 7, 9\}$ и $x = 13$ функция должна сохранить в списке подмножество {6, 7} и вернуть значение 2. Отметим, что в этой оптимизационной задаче оптимальное подмножество может быть не единственным, причём с оптимальным количеством элементов. Например, для $S = \{2, 6, 3, 5\}$ и $x = 8$ оптимальными будут два подмножества – {2, 6} и {3, 5}. Частичное (и оптимальное) решение представьте двоичным списком, как в методе из листинга 12.7. Кроме того, реализуйте сокращение дерева рекурсии, исключив недопустимые подмножества, а также подмножества с мощностью, превышающей оптимальную. После этого напишите метод-оболочку, который находит оптимальное подмножество и выводит его, если оно найдено (не исключено, что искомого подмножества T в S вообще нет).

Что ещё почитать

Монографии о рекурсии

Наша книга охватывает довольно широкий круг вопросов, связанных с рекурсивным программированием, включая вопросы разработки рекурсивных алгоритмов. Однако читатель может найти дополнительные аспекты, примеры и детали реализации в других источниках. В частности, в [14] содержатся многочисленные примеры рекурсивных алгоритмов на Паскале, работающих со структурами данных (связанные списки, деревья, графы), реализованными посредством указателей. В нашей книге мы, напротив, обошлись без них, поскольку в языке Python их явное использование не предусмотрено. Предлагаемая книга включает алгоритмы перебора с возвратами для генерации таких новых комбинаторных объектов, как сочетания, композиции и разбиения. А заключительная её глава посвящена низкоуровневым проблемам преобразования рекурсивных программ в эквивалентные итерационные.

Книги [13] с примерами на Паскале и [14] с примерами на Java также целиком посвящены рекурсии. Обе они содержат главы о рекурсивных типах данных и о низкоуровневой реализации рекурсии.

Java-программисты могут найти много интересного в книге [11] с примерами связанных списков, связанных деревьев и графических задач.

Разработка и анализ алгоритмов

Настоящая книга включает задачи, которые могут быть решены методами «разделяй и властвуй» и перебора с возвратами. Но существует множество превосходных книг, рассматривающих более сложные задачи и иные алгоритмы их решения – «жадный», динамическое программирование или опирающиеся на более сложные структуры данных. Книги [3], [5], [6], [19] – это лишь малая часть обширной литературы по методам разработки и анализа алгоритмов.

Любители сложных задач могут обратиться к книгам, посвящённым соревнованиям по программированию (например, Международный университетский чемпионат ACM по программированию или Международная олимпиада по информатике). Наиболее известные источники – [21], [22].

Учебники [9], [20] посвящены исключительно анализу алгоритмов. Второй из них гораздо полнее и доступен в массовом открытом сетевом ресурсе *coursera.org*. Превосходные курсы по алгоритмам предлагают и другие сетевые учебные платформы.

Функциональное программирование

Всё функциональное программирование основано на рекурсии. Поэтому для освоения этой парадигмы программирования программистам нужно основательно проштудировать нашу книгу. Популярные книги по этой теме – [1], [2], [10]. Последнюю из них (см. сайт *coursera.org*) можно рекомендовать в качестве лучшего обучающего курса по этой дисциплине.

Язык Python

Настоящая книга предполагает, что её читатель уже имеет некоторый опыт программирования. Популярные книги [7], [12], [24] будут полезны тем, кто хочет познакомиться со всеми возможностями языка Python, включая детали его реализации и разработку альтернативных и/или более эффективных рекурсивных решений задач, проходящих через всю нашу книгу.

Исследования в обучении и изучении рекурсии

Настоящая книга содержит несколько идей, возникших в процессе обучения и изучения рекурсии. В частности, она делает упор не на вычислительную (например, пошаговую мысленную) модель или на итерационный/императивный подход, а на абстрактную декларативную декомпозицию задачи и индукцию. Этому способствовали такие, например, источники, как [4], [23].

Используемые в главе 2 методика и схемы (они подобны приведённым в статье [23]) были введены для того, чтобы сделать упор именно на декларативном мышлении. Да и само построение книги преследует ту же цель. Такие понятия, как трассировка, деревья рекурсии, программный стек или связь итерации с хвостовой рекурсией, не вводятся вплоть до глав 10 и 11.

Книга включает также примеры из моих статей [17] и [18], посвящённых обучению взаимной, хвостовой (и вложенной) рекурсии через обобщённые функции.

Наконец, есть множество источников по исследованию обучения и изучения рекурсии. Недавние обзоры [8], [16] содержат сотни источников по данной теме.

Дополнительная литература

1. *Abelson Harold and Sussman Gerald J*. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.
2. *Bird Richard*. Introduction to Functional Programming Using Haskell. Prentice Hall Europe, April 1998.
3. *Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., and Stein Clifford*. Introduction to Algorithms. The MIT Press, 3rd edition, 2009.
4. *Ginat D. and Shifroni E.* Teaching recursion in a procedural environment – how much should we emphasize the computing model? *SIGCSE Bull.,* 31(1):127–131, 1999.
5. *Goodrich Michael T., Tamassia Roberto, and Goldwasser Michael H*. Data Structures and Algorithms in Python. Wiley Publishing, 1st edition, 2013.
6. *Levitin Anany V*. Introduction to the Design and Analysis of Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2012.
7. *Lutz Mark*. Learning Python. O'Reilly, 5th edition, 2013.
8. *McCauley Renée, Grissom Scott, Fitzgerald Sue, and Murphy Laurie*. Teaching and learning recursive programming: a review of the research literature. Computer Science Education, 25(1):37–66, 2015.
9. *McConnell Jeffrey J*. Analysis of Algorithms: An Active Learning Approach. Jones and Bartlett Publishers, Inc., USA, 1st edition, 2001.
10. *Odersky Martin, Spoon Lex, and Venners Bill*. Programming in Scala: A Comprehensive Step-by-Step Guide. Artima Incorporation, USA, 1st edition, 2008.
11. *Pevac Irena*. Practicing Recursion in Java. CreateSpace Independent Publishing Platform, 1st edition, April 2016.
12. *Pilgrim Mark.* Dive Into Python 3. Apress, Berkely, CA, USA, 2009.
13. *Roberts Eric S*. Thinking Recursively. Wiley, 1st edition, January 1986.
14. *Roberts Eric S*. Thinking Recursively with Java. Wiley, 1st edition, February 2006.
15. *Rohl Jeffrey Soden*. Recursion Via Pascal. Cambridge Computer Science Texts. Cambridge University Press, 1st edition, August 1984.

16. *Rinderknecht Christian*. A survey on teaching and learning recursive programming. Informatics in Education, 13(1):87-119, 2014.
17. *Rubio-Sánchez M., J. Urquiza-Fuentes, and Pareja-Flores C*. A gentle introduction to mutual recursion. SIGCSE Bull., 40(3):235–239, 2008.
18. *Rubio-Sánchez Manuel*. Tail recursive programming by applying generalization. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '10, pages 98–102. ACM, 2010.
19. *Sedgewick Robert and Wayne Kevin*. Algorithms. Addison-Wesley Professional, 4th edition, 2011.
20. *Sedgewick Robert and Flajolet Philippe*. An Introduction to the Analysis of Algorithms. Addison-Wesley Professional, 2nd edition, 2013.
21. *Skiena Steven S. and Revilla Miguel*. Programming Challenges: The Programming Contest Training Manual. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
22. *Skiena Steven S*. The Algorithm Design Manual. Springer Publishing Company, Incorporated, 2nd edition, 2008.
23. *Sooriamurthi R*. Problems in comprehending recursion and suggested solutions. SIGCSE Bull., 33(3):25–28, 2001.
24. *Summerfield Mark*. Programming in Python 3: A Complete Introduction to the Python Language. Addison-Wesley Professional, 2nd edition, 2009.

Список рисунков

Список таблиц

Список листингов

Предметный указатель

И

К

Л

М

Н

О

П

Р

С

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: тел**. +7(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Мануэль Рубио-Санчес

Введение в рекурсивное программирование

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Борисов Е. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 35,43. Тираж 200 экз.

Веб-сайт издательства: *www.dmkpress.com*