Оглавление

Предисловие ... 12

Целевая аудитория ... 14

Содержание и структура книги ... 15

Примерные учебные курсы ... 17

Благодарности ... 17

Глава 1. Основные понятия рекурсивного программирования ... 19

1.1. Распознание рекурсии ... 19

1.2. Декомпозиция задачи ... 25

1.3. Рекурсивный код ... 33

1.4. Индукция ... 40

1.4.1. Математические доказательства методом индукции ... 40

1.4.2. Рекурсивная убеждённость ... 41

1.4.3. Императивное и декларативное программирование ... 43

1.5. Рекурсия против итерации ... 44

1.6. Типы рекурсии ... 46

1.6.1. Линейная рекурсия ... 46

1.6.2. Хвостовая рекурсия ... 46

1.6.3. Множественная рекурсия ... 47

1.6.4. Взаимная рекурсия ... 47

1.6.5. Вложенная рекурсия ... 48

1.7. Упражнения ... 48

Глава 2. Методика рекурсивного мышления ... 51

2.1. Шаблон проектирования рекурсивных алгоритмов ... 51

2.2. Размер задачи ... 52

2.3. Начальные условия ... 54

2.4. Декомпозиция задачи ... 57

2.5. Рекурсивные условия, индукция и схемы ... 61

2.5.1. Рекурсивное мышление посредством схем ... 61

2.5.2. Конкретные экземпляры задачи ... 65

2.5.3. Альтернативные обозначения ... 66

2.5.4. Процедуры 67
2.5.5. Несколько подзадач 69
2.6. Тестирование 71
2.7. Упражнения 75

Глава 3. Анализ времени выполнения рекурсивных алгоритмов 77

3.1. Предварительные математические соглашения 77
3.1.1. Степени и логарифмы 78
3.1.2. Биномиальные коэффициенты 78
3.1.3. Пределы и правило Лопиталя 79
3.1.4. Суммы и произведения 79
3.1.5. Верхняя и нижняя границы 85
3.1.6. Тригонометрия 86
3.1.7. Векторы и матрицы 87
3.2. Временнáя сложность вычислений 89
3.2.1. Порядок роста функций 90
3.2.2. Асимптотические обозначения 92
3.3. Рекуррентные соотношения 95
3.3.1. Метод расширения 99
3.3.2. Общий метод решения разностных уравнений 107
3.4. Упражнения 119

Глава 4. Линейная рекурсия I: основные алгоритмы 122

4.1. Арифметические операции 123
4.1.1. Степенная функция 123
4.1.2. Медленное сложение 127
4.1.3. Двойная сумма 131
4.2. Системы счисления 132
4.2.1. Двоичное представление неотрицательного целого числа 133
4.2.2. Приведение десятичного числа к другому основанию 135
4.3. Строки 136
4.3.1. Обращение строки 137
4.3.2. Является ли строка палиндромом? 137
4.4. Дополнительные задачи 139
4.4.1. Сортировка выбором 139
4.4.2. Схема Горнера 141
4.4.3. Треугольник Паскаля 143
4.4.4. Резистивная цепь 145
4.5. Упражнения 147

Глава 5. Линейная рекурсия II: хвостовая рекурсия 151
5.1. Логические функции 152
5.1.1. Есть ли в неотрицательном целом числе заданная цифра? 152
5.1.2. Равны ли строки? 155
5.2. Алгоритмы поиска в списке 156
5.2.1. Линейный поиск 157
5.2.2. Двоичный поиск в сортированном списке 159
5.3. Двоичные деревья поиска 161
5.3.1. Поиск элемента 163
5.3.2. Вставка элемента 165
5.4. Схемы разбиения 165
5.4.1. Основная схема разбиения 167
5.4.2. Метод разбиения Хоара 168
5.5. Алгоритм quickselect 173
5.6. Двоичный поиск корня функции 174
5.7. Задача лесоруба 177
5.8. Алгоритм евклида 182
5.9. Упражнения 184
Глава 6. Множественная рекурсия I: «разделяй и властвуй» 188
6.1. Отсортирован ли список? 189
6.2. Сортировка 190
6.2.1. Алгоритм сортировки слиянием 191
6.2.2. Алгоритм быстрой сортировки 194
6.3. Мажоритарный элемент списка 197
6.4. Быстрое целочисленное умножение 200
6.5. Умножение матриц 203
6.5.1. Умножение матриц методом «разделяй и властвуй» 203
6.5.2. Алгоритм Штрассена умножения матриц 207
6.6. Задача укладки тримино 208
6.7. Задача очертания 213
6.8. Упражнения 219
Глава 7. Множественная рекурсия II: пазлы, фракталы и прочее 222
7.1. Путь через болото 222
7.2. Ханойская башня 226

7.3. Обходы дерева ... 230
7.3.1. Внутренний обход ... 231
7.3.2. Прямой и обратный обходы ... 233
7.4. Самый длинный палиндром в строке ... 234
7.5. Фракталы ... 236
7.5.1. Снежинка Коха ... 236
7.5.2. Ковёр Серпиньского ... 242
7.6. Упражнения ... 245

Глава 8. Задачи подсчёта ... 250
8.1. Перестановки ... 251
8.2. Размещения с повторениями ... 253
8.3. Сочетания ... 255
8.4. Подъём по лестнице ... 256
8.5. Путь по Манхэттену ... 259
8.6. Триангуляция выпуклого многоугольника ... 260
8.7. Пирамиды из кругов ... 263
8.8. Упражнения ... 265

Глава 9. Взаимная рекурсия ... 268
9.1. Чётность числа ... 269
9.2. Игры со многими игроками ... 270
9.3. Размножение кроликов ... 271
9.3.1. Зрелые и незрелые пары кроликов ... 272
9.3.2. Родовое дерево кроликов ... 273
9.4. Задача о станциях водоочистки ... 277
9.4.1. Переток воды между городами ... 278
9.4.2. Сброс воды в каждом городе ... 280
9.5. Циклические ханойские башни ... 282
9.6. Грамматики и синтаксический анализатор на основе рекурсивного спуска ... 288
9.6.1. Лексический анализ входной строки ... 288
9.6.2. Синтаксический анализатор на основе рекурсивного спуска ... 293
9.7. Упражнения ... 302

Глава 10. Выполнение программы ... 306
10.1. Поток управления между подпрограммами ... 309
10.2. Деревья рекурсии ... 312

10.2.1. Анализ времени выполнения ... 318
10.3. Программный стек ... 320
10.3.1. Стековые кадры ... 320
10.3.2. Трассировка стека ... 323
10.3.3. Пространственная сложность вычислений ... 325
10.3.4. Ошибки предельной глубины рекурсии и переполнения стека ... 326
10.3.5. Рекурсия как альтернатива стеку ... 327
10.4. Мемоизация и динамическое программирование ... 332
10.4.1 Мемоизация ... 332
10.4.2. Граф зависимости и динамическое программирование ... 336
10.5. Упражнения ... 340
Глава 11. Вложенная рекурсия и снова хвостовая ... 347
11.1. Хвостовая рекурсия и итерация ... 347
11.2. Итерационный подход к хвостовой рекурсии ... 351
11.2.1. Факториал ... 352
11.2.2. Приведение десятичного числа к другому основанию ... 353
11.3. Вложенная рекурсия ... 356
11.3.1. Функция Аккермана ... 356
11.3.2. Функция-91 Маккарти ... 357
11.3.3. Цифровой корень ... 357
11.4. К хвостовой и вложенной рекурсии через обобщённую функцию ... 359
11.4.1. Факториал ... 359
11.4.2. Приведение десятичного числа к основанию b ... 363
11.5. Упражнения ... 365
Глава 12. Множественная рекурсия III: перебор с возвратами ... 366
12.1. Введение ... 367
12.1.1. Частичные и полные решения ... 367
12.1.2. Рекурсивная структура ... 369
12.2. Генерация комбинаторных объектов ... 371
12.2.1. Подмножества ... 372
12.2.2. Перестановки ... 377
12.3. Задача n ферзей ... 381
12.3.1. Поиск всех решений ... 383
12.3.2. Поиск одного решения ... 384
12.4. Задача о сумме элементов подмножества ... 387
12.5. Путь в лабиринте ... 390

12.6. Судоку ... 396
12.7. Задача о рюкзаке 0–1 ... 402
12.7.1. Стандартный алгоритм перебора с возвратами ... 402
12.7.2. Алгоритм ветвей и границ ... 407
12.8. Упражнения ... 411

Что ещё почитать ... 416
Монографии о рекурсии ... 416
Разработка и анализ алгоритмов ... 416
Функциональное программирование ... 417
Язык Python ... 417
Исследования в обучении и изучении рекурсии ... 417

Дополнительная литература ... 418
Список рисунков ... 420
Список таблиц ... 426
Список листингов ... 426
Предметный указатель ... 432

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 не достаточно 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 вызывают 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 применяется 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 Рубьо-Санчес,
июль 2017

Глава 1

Основные понятия рекурсивного программирования

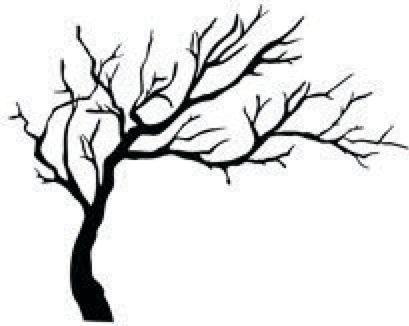
Итерация – от человека, рекурсия – от Бога.

– 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 этой 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 подобна структуре ветвления деревьев, когда бо́льшая 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.

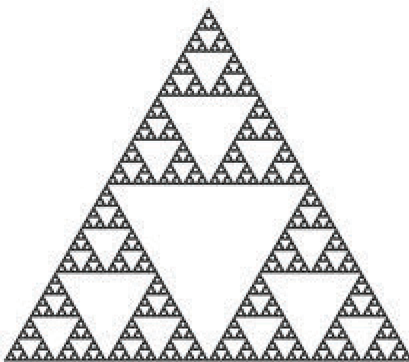


Ветви дерева

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 меньшие. 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:

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

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

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

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.

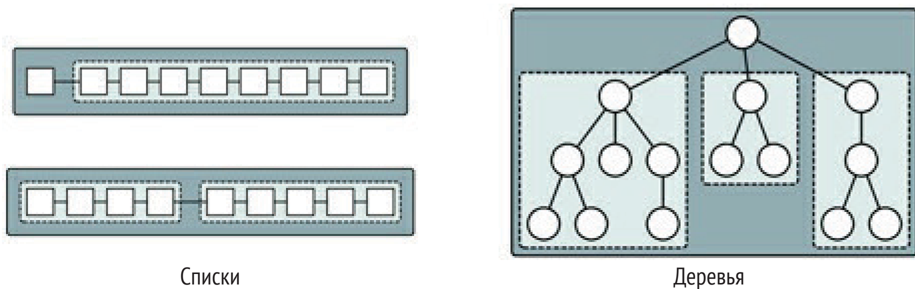


Рис. 1.2. Рекурсивная декомпозиция списков и деревьев

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.

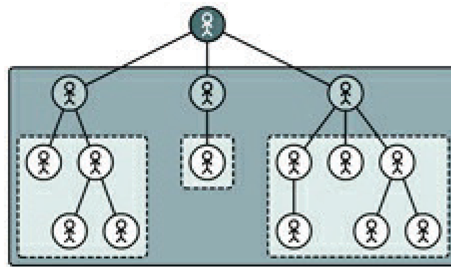


Рис. 1.3. Генеалогическое древо потомков человека – его детей и потомков их детей

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, обозначаемую как $S(n)$ и выраженную 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.

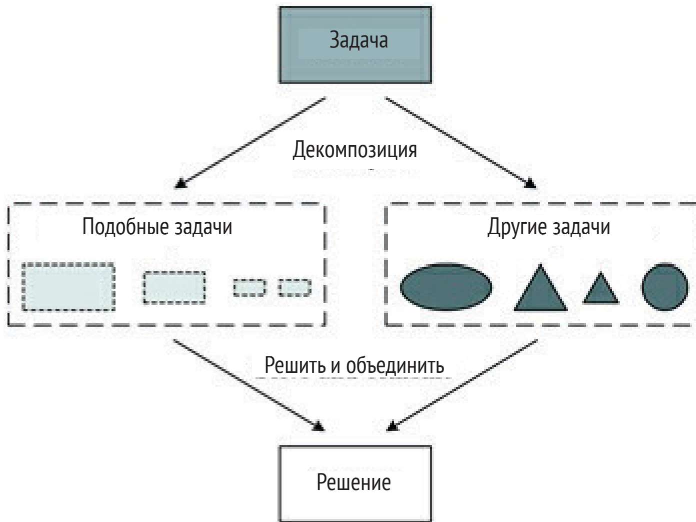
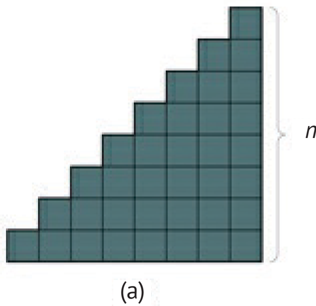


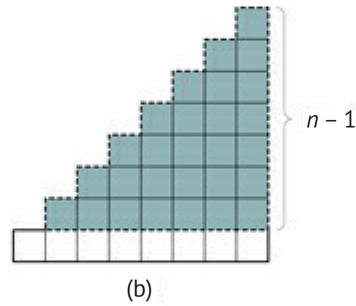
Рис. 1.4. Рекурсивное решение задачи

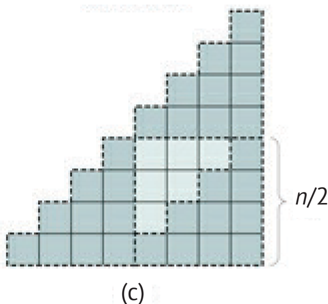
Первый вариант – уменьшать n на единицу. В этом случае наша цель – определить $S(n)$, используя подзадачу $S(n - 1)$. Соответствующее рекурсивное решение, полученное алгебраически в разделе 1.1, приведено в (1.5). Рекурсивное условие можно также вывести из графического представления задачи. Например, 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 $2 + ... + (n - 1) + n$. 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: $m + (m + 1) + ... + (n - 1) + n$, 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.







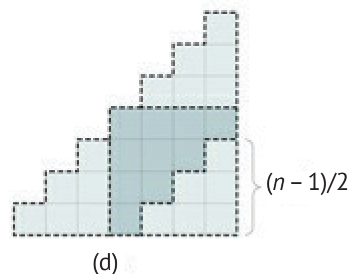


Рис. 1.5. Декомпозиции суммы первых положительных целых чисел

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:

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

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:

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

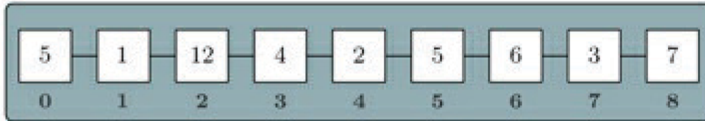
где $a[i]$ – $(i + 1)$-й элемент списка, поскольку 1-й элемент списка имеет индекс 0. На рис. 1.6(a) изображён частный случай списка из 9 элементов.

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 ($\mathbf{a}[0{:}\ n-2]$, где $\mathbf{a}[i : j]$ – подсписок от $\mathbf{a}[i]$ до $\mathbf{a}[j-1]$ в обозначениях языка Python) и единственный элемент ($\mathbf{a}[n-1]$), соответствующий последнему числу в списке (см. рис.1.6(b)). В этом случае задачу можно определить рекурсивно таким образом:

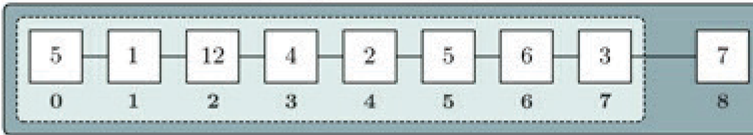
$$s(\mathbf{a}) = \begin{cases} 0, & \text{если } n = 0, \\ s(\mathbf{a}[0{:}n-2]) + \mathbf{a}[n-1], & \text{если } n > 0. \end{cases} \tag{1.9}$$

Исходная задача: $s(\mathbf{a}) = \mathbf{a}[0] + \mathbf{a}[1] + \ldots + \mathbf{a}[n-1]$



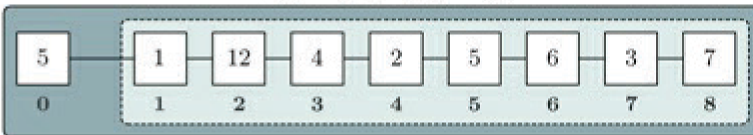
(a)

$s(\mathbf{a}) = s(\mathbf{a}[0 : n-1]) + \mathbf{a}[n-1]$



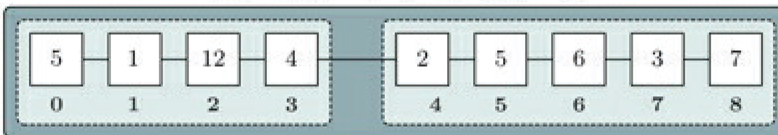
(b)

$s(\mathbf{a}) = \mathbf{a}[0] + s(\mathbf{a}[1 : n])$



(c)

$s(\mathbf{a}) = s(\mathbf{a}[0 : n//2]) + s(\mathbf{a}[n//2 : n])$



(d)

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 вызывает саму рекурсивную функцию.

C, Java:

```
1 int sum_first_naturals(int n)
2 {
3     if (n==1)
4         return 1;
5     else
6         return sum_first_naturals(n-1) + n;
7 }
```

Pascal:

```
1 function sum_first_naturals(n: integer): integer;
2 begin
3     if n=1 then
4         sum_first_naturals := 1
5     else
6         sum_first_naturals := sum_first_naturals(n-1) + n;
7 end;
```

MATLAB®:

```
1 function result = sum_first_naturals(n)
2     if n==1
3         result = 1;
4     else
5         result = sum_first_naturals(n-1) + n;
6     end
```

Scala:

```
1 def sum_first_naturals(n: Int): Int = {
2     if (n==1)
3         return 1
4     else
5         return sum_first_naturals(n-1) + n
6 }
```

Haskell:

```
1 sum_first_naturals 1 = 1
2 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

```
1 def sum_first_naturals_2(n):
2     if n == 1:
3         return 1
4     elif n == 2:
5         return 3
6     elif n % 2:
7         return (3 * sum_first_naturals_2((n - 1) / 2)
8                 + sum_first_naturals_2((n + 1) / 2))
9     else:
10        return (3 * sum_first_naturals_2(n / 2)
11                + 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

```
1 def fibonacci(n):
2     if n == 1 or n == 2:
3         return 1
4     else:
5         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 $F(1)$, $F(2)$, …, $F(n - 2)$. В листинге 1.4 приводится возможное решение с использованием цикла **`for`**.

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

```
1 def fibonacci_alt(n):
2     if n == 1 or n == 2:
3         return 1
4     else:
5         aux = 1
6         for i in range(1, n - 1):
7             aux += fibonacci_alt(i)
8         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.

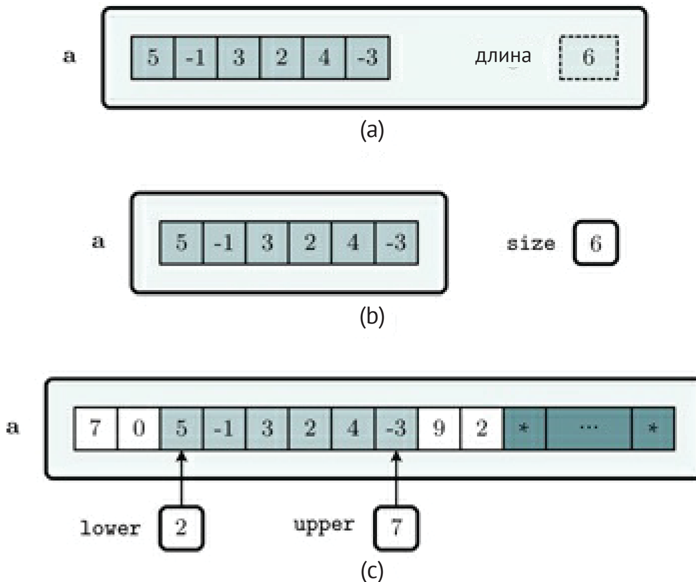


Рис. 1.8. Списочные структуры данных с параметрами, необходимыми для определения подсписков

В случае (a) 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 С или Паскаль, получить длину стандартного массива невозможно. Если длину списка нельзя извлечь непосредственно из структуры данных, то для хранения и получения длины списка нужен до-

полнительный параметр, например, size, как в случае (b). Такой подход может применяться при работе с массивами – фиксированного размера или частично заполненными. Правда, в таких 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**

```
1  # Decomposition: s(a) => s(a[0:n-1]), a[n-1]
2  def sum_list_length_1(a):
3      if len(a) == 0:
4          return 0
5      else:
6          return (sum_list_length_1(a[0:len(a) - 1])
7                  + a[len(a) - 1])
8
9
10 # Decomposition: s(a) => a[0], s(a[1:n])
11 def sum_list_length_2(a):
12     if len(a) == 0:
13         return 0
14     else:
15         return a[0] + sum_list_length_2(a[1:len(a)])
16
17
18 # Decomposition: s(a) => s(a[0:n//2]), s(a[n//2:n])
19 def sum_list_length_3(a):
20     if len(a) == 0:
21         return 0
22     elif len(a) == 1:
23         return a[0]
24     else:
25         middle = len(a) // 2
26         return (sum_list_length_3(a[0:middle])
27                 + sum_list_length_3(a[middle:len(a)]))
28
29
30 # Some list:
31 a = [5, -1, 3, 2, 4, -3]
32
33 # Function calls:
34 print(sum_list_length_1(a))
35 print(sum_list_length_2(a))
36 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**

```
1  # Decomposition: s(a) => s(a[0:n-1]), a[n-1]
2  def sum_list_limits_1(a, lower, upper):
3      if lower > upper:
4          return 0
5      else:
6          return a[upper] + sum_list_limits_1(a, lower, upper - 1)
7
8
9  # Decomposition: s(a) => a[0], s(a[1:n])
10 def sum_list_limits_2(a, lower, upper):
11     if lower > upper:
12         return 0
13     else:
14         return a[lower] + sum_list_limits_2(a, lower + 1, upper)
15
16
17 # Decomposition: s(a) => s(a[0:n//2]), s(a[n//2:n])
18 def sum_list_limits_3(a, lower, upper):
19     if lower > upper:
20         return 0
21     elif lower == upper:
22         return a[lower]  # or a[upper]
23     else:
24         middle = (upper + lower) // 2
25         return (sum_list_limits_3(a, lower, middle)
26                 + sum_list_limits_3(a, middle + 1, upper))
27
28
29 # Some list:
30 a = [5, -1, 3, 2, 4, -3]
31
32 # Function calls:
33 print(sum_list_limits_1(a, 0, len(a) - 1))
34 print(sum_list_limits_2(a, 0, len(a) - 1))
35 print(sum_list_limits_3(a, 0, len(a) - 1))
```