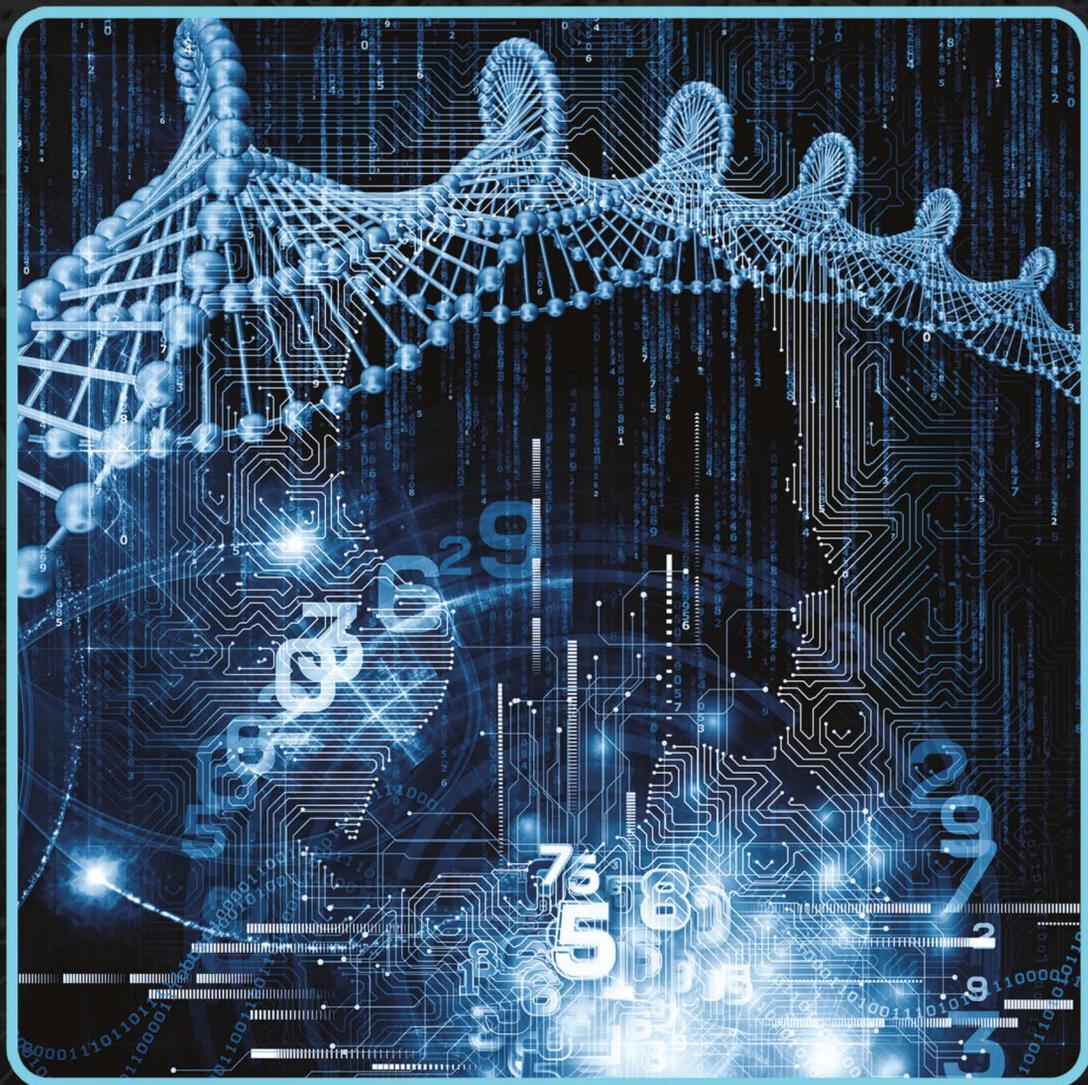


Introduction to Recursive Programming

Manuel Rubio-Sánchez



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK



Introduction to
**Recursive
Programming**



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction to
**Recursive
Programming**

Manuel Rubio-Sánchez



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2018 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20170817

International Standard Book Number-13: 978-1-4987-3528-5 (Paperback)
International Standard Book Number-13: 978-1-138-10521-8 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

To the future generations



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

PREFACE	xv
LIST OF FIGURES	xxi
LIST OF TABLES	xxxix
LIST OF LISTINGS	xxxiii
CHAPTER 1 ■ Basic Concepts of Recursive Programming	1
1.1 RECOGNIZING RECURSION	1
1.2 PROBLEM DECOMPOSITION	7
1.3 RECURSIVE CODE	14
1.4 INDUCTION	20
1.4.1 Mathematical proofs by induction	20
1.4.2 Recursive leap of faith	22
1.4.3 Imperative vs. declarative programming	25
1.5 RECURSION VS. ITERATION	25
1.6 TYPES OF RECURSION	27
1.6.1 Linear recursion	27
1.6.2 Tail recursion	27
1.6.3 Multiple recursion	28
1.6.4 Mutual recursion	28
1.6.5 Nested recursion	29
1.7 EXERCISES	29
CHAPTER 2 ■ Methodology for Recursive Thinking	31
2.1 TEMPLATE FOR DESIGNING RECURSIVE ALGORITHMS	31

2.2	SIZE OF THE PROBLEM	32
2.3	BASE CASES	34
2.4	PROBLEM DECOMPOSITION	37
2.5	RECURSIVE CASES, INDUCTION, AND DIAGRAMS	41
2.5.1	Thinking recursively through diagrams	41
2.5.2	Concrete instances	45
2.5.3	Alternative notations	47
2.5.4	Procedures	47
2.5.5	Several subproblems	49
2.6	TESTING	52
2.7	EXERCISES	55
CHAPTER 3 ■ Runtime Analysis of Recursive Algorithms		57
3.1	MATHEMATICAL PRELIMINARIES	57
3.1.1	Powers and logarithms	58
3.1.2	Binomial coefficients	58
3.1.3	Limits and L'Hopital's rule	59
3.1.4	Sums and products	60
3.1.5	Floors and ceilings	66
3.1.6	Trigonometry	66
3.1.7	Vectors and matrices	67
3.2	COMPUTATIONAL TIME COMPLEXITY	70
3.2.1	Order of growth of functions	71
3.2.2	Asymptotic notation	73
3.3	RECURRENCE RELATIONS	76
3.3.1	Expansion method	80
3.3.2	General method for solving difference equations	89
3.4	EXERCISES	101
CHAPTER 4 ■ Linear Recursion I: Basic Algorithms		105
4.1	ARITHMETIC OPERATIONS	106

4.1.1	Power function	106
4.1.2	Slow addition	110
4.1.3	Double sum	113
4.2	BASE CONVERSION	115
4.2.1	Binary representation of a nonnegative integer	115
4.2.2	Decimal to base b conversion	117
4.3	STRINGS	119
4.3.1	Reversing a string	119
4.3.2	Is a string a palindrome?	120
4.4	ADDITIONAL PROBLEMS	121
4.4.1	Selection sort	121
4.4.2	Horner's method for evaluating polynomials	124
4.4.3	A row of Pascal's triangle	125
4.4.4	Ladder of resistors	127
4.5	EXERCISES	129
CHAPTER 5 ■ Linear Recursion II: Tail Recursion		133
<hr/>		
5.1	BOOLEAN FUNCTIONS	134
5.1.1	Does a nonnegative integer contain a particular digit?	134
5.1.2	Equal strings?	136
5.2	SEARCHING ALGORITHMS FOR LISTS	139
5.2.1	Linear search	139
5.2.2	Binary search in a sorted list	142
5.3	BINARY SEARCH TREES	143
5.3.1	Searching for an item	144
5.3.2	Inserting an item	147
5.4	PARTITIONING SCHEMES	148
5.4.1	Basic partitioning scheme	149
5.4.2	Hoare's partitioning method	150
5.5	THE QUICKSELECT ALGORITHM	155
5.6	BISECTION ALGORITHM FOR ROOT FINDING	157

5.7	THE WOODCUTTER PROBLEM	158
5.8	EUCLID'S ALGORITHM	164
5.9	EXERCISES	167
CHAPTER 6 ■ Multiple Recursion I: Divide and Conquer		171
<hr/>		
6.1	IS A LIST SORTED IN ASCENDING ORDER?	172
6.2	SORTING	173
6.2.1	The merge sort algorithm	174
6.2.2	The quicksort algorithm	177
6.3	MAJORITY ELEMENT IN A LIST	180
6.4	FAST INTEGER MULTIPLICATION	183
6.5	MATRIX MULTIPLICATION	186
6.5.1	Divide and conquer matrix multiplication	187
6.5.2	Strassen's matrix multiplication algorithm	190
6.6	THE TROMINO TILING PROBLEM	191
6.7	THE SKYLINE PROBLEM	196
6.8	EXERCISES	203
CHAPTER 7 ■ Multiple Recursion II: Puzzles, Fractals, and More...		205
<hr/>		
7.1	SWAMP TRAVERSAL	205
7.2	TOWERS OF HANOI	209
7.3	TREE TRAVERSALS	213
7.3.1	Inorder traversal	215
7.3.2	Preorder and postorder traversals	216
7.4	LONGEST PALINDROME SUBSTRING	217
7.5	FRACTALS	220
7.5.1	Koch snowflake	220
7.5.2	Sierpiński's carpet	224
7.6	EXERCISES	226

CHAPTER 8 ■ Counting Problems **235**

8.1	PERMUTATIONS	236
8.2	VARIATIONS WITH REPETITION	238
8.3	COMBINATIONS	240
8.4	STAIRCASE CLIMBING	242
8.5	MANHATTAN PATHS	244
8.6	CONVEX POLYGON TRIANGULATIONS	245
8.7	CIRCLE PYRAMIDS	248
8.8	EXERCISES	250

CHAPTER 9 ■ Mutual Recursion **253**

9.1	PARITY OF A NUMBER	254
9.2	MULTIPLAYER GAMES	255
9.3	RABBIT POPULATION GROWTH	256
9.3.1	Adult and baby rabbit pairs	257
9.3.2	Rabbit family tree	258
9.4	WATER TREATMENT PLANTS PUZZLE	263
9.4.1	Water flow between cities	263
9.4.2	Water discharge at each city	265
9.5	CYCLIC TOWERS OF HANOI	268
9.6	GRAMMARS AND RECURSIVE DESCENT PARSERS	273
9.6.1	Tokenization of the input string	274
9.6.2	Recursive descent parser	279
9.7	EXERCISES	288

CHAPTER 10 ■ Program Execution **291**

10.1	CONTROL FLOW BETWEEN SUBROUTINES	292
10.2	RECURSION TREES	297
10.2.1	Runtime analysis	303
10.3	THE PROGRAM STACK	305

10.3.1	Stack frames	306
10.3.2	Stack traces	309
10.3.3	Computational space complexity	310
10.3.4	Maximum recursion depth and stack overflow errors	312
10.3.5	Recursion as an alternative to a stack data structure	313
10.4	MEMOIZATION AND DYNAMIC PROGRAMMING	317
10.4.1	Memoization	317
10.4.2	Dependency graph and dynamic programming	322
10.5	EXERCISES	325
CHAPTER 11 ■ Tail Recursion Revisited and Nested Recursion		333
<hr/>		
11.1	TAIL RECURSION VS. ITERATION	333
11.2	TAIL RECURSION BY THINKING ITERATIVELY	337
11.2.1	Factorial	337
11.2.2	Decimal to base b conversion	340
11.3	NESTED RECURSION	342
11.3.1	The Ackermann function	342
11.3.2	The McCarthy 91 function	342
11.3.3	The digital root	343
11.4	TAIL AND NESTED RECURSION THROUGH FUNCTION GENERALIZATION	344
11.4.1	Factorial	345
11.4.2	Decimal to base b conversion	348
11.5	EXERCISES	350
CHAPTER 12 ■ Multiple Recursion III: Backtracking		353
<hr/>		
12.1	INTRODUCTION	354
12.1.1	Partial and complete solutions	354
12.1.2	Recursive structure	356
12.2	GENERATING COMBINATORIAL ENTITIES	358

12.2.1	Subsets	359
12.2.2	Permutations	364
12.3	THE N -QUEENS PROBLEM	368
12.3.1	Finding every solution	370
12.3.2	Finding one solution	372
12.4	SUBSET SUM PROBLEM	372
12.5	PATH THROUGH A MAZE	377
12.6	THE SUDOKU PUZZLE	384
12.7	0-1 KNAPSACK PROBLEM	388
12.7.1	Standard backtracking algorithm	389
12.7.2	Branch and bound algorithm	393
12.8	EXERCISES	397
FURTHER READING		403
Index		407



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Preface

Recursion is one of the most fundamental concepts in computer science and a key programming technique that, similarly to iteration, allows computations to be carried out repeatedly. It accomplishes this by employing methods that invoke themselves, where the central idea consists of designing a solution to a problem by relying on solutions to smaller instances of the same problem. Most importantly, recursion is a powerful problem-solving approach that enables programmers to develop concise, intuitive, and elegant algorithms.

Despite the importance of recursion for algorithm design, most programming books do not cover the topic in detail. They usually devote just a single chapter or a brief section, which is often insufficient for assimilating the concepts needed to master the topic. Exceptions include *Recursion via Pascal*, by J. S. Rohl (Cambridge University Press, 1984); *Thinking Recursively with Java*, by E. S. Roberts (Wiley, 2006); and *Practicing Recursion in Java*, by I. Pevac (CreateSpace Independent Publishing Platform, 2016), which focus exclusively on recursion. The current book provides a comprehensive treatment of the topic, but differs from the previous texts in several ways.

Numerous computer programming professors and researchers in the field of computer science education agree that recursion is difficult for novice students. With this in mind, the book incorporates several elements in order to foster its pedagogical effectiveness. Firstly, it contains a larger collection of simple problems in order to provide a solid foundation of the core concepts, before diving into more complex material. In addition, one of the book's main assets is the use of a step-by-step methodology, together with specially designed diagrams, for guiding and illustrating the process of developing recursive algorithms. The book also contains specific chapters on combinatorial problems and mutual recursion. These topics can broaden students' understanding of recursion by forcing them to apply the learned concepts differently, or in a more sophisticated manner. Lastly, introductory programming courses usually focus on the imperative programming paradigm, where students primar-

ily learn iteration, understanding and controlling *how* programs work. In contrast, recursion requires adopting a completely different way of thinking, where the emphasis should be on *what* programs compute. In this regard, several studies encourage instructors to avoid, or postpone, covering how recursive programs work (i.e., control flow, recursion trees, the program stack, or the relationship between iteration and tail recursion) when introducing recursion, since the concepts and abilities learned for iteration may actually hamper the acquisition of skills related to recursion and declarative programming. Therefore, topics related to iteration and program execution are covered towards the end of the book, when the reader should have mastered the design of recursive algorithms from a purely declarative perspective.

The book also includes a richer chapter on the theoretical analysis of the computational cost of recursive programs. On the one hand, it contains a broad treatment of mathematical recurrence relations, which constitute the fundamental tools for analyzing the runtime or recursive algorithms. On the other hand, it includes a section on mathematical preliminaries that reviews concepts and properties that are not only needed for solving recurrence relations, but also for understanding the statements and solutions to the computational problems in the book. In this regard, the text also offers the possibility to learn some basic mathematics along the way. The reader is encouraged to embrace this material, since it is essential in many fields of computer science.

The code examples are written in Python 3, which is arguably today's most popular introductory programming language in top universities. In particular, they were tested on Spyder (Scientific PYthon Development EnviRonment). The reader should be aware that the purpose of the book is not to teach Python, but to transmit skills associated with recursive thinking for problem solving. Thus, aspects such as code simplicity and legibility have been prioritized over efficiency. In this regard, the code does not contain advanced Python features. Therefore, students with background in other programming languages such as C++ or Java should be able to understand the code without effort. Of course, the methods in the book can be implemented in several ways, and readers are encouraged to write more efficient versions, include more sophisticated Python constructs, or design alternative algorithms. Lastly, the book provides recursive variants of iterative algorithms that usually accompany other well-known recursive algorithms. For instance, it contains recursive versions of Hoare's partition method used in the quicksort algorithm, or of the merging method within the merge sort algorithm.

The book proposes numerous exercises at the end of the chapters, whose fully worked-out solutions are included in an instructor's manual available at the book's official website (see www.crcpress.com). Many of them are related to the problems analyzed in the main text, which make them appropriate candidates for exams and assignments.

The code in the text will also be available for download at the book's website. In addition, I will maintain a complementary website related to the book: <https://sites.google.com/view/recursiveprogrammingintro/>. Readers are more than welcome to send me comments, suggestions for improvements, alternative (clearer or more efficient) code, versions in other programming languages, or detected errata. Please send emails to: recursion.book@gmail.com.

INTENDED AUDIENCE

The main goal of the book is to teach students how to think and program recursively, by analyzing a wide variety of computational problems. It is intended mainly for undergraduate students in computer science or related technical disciplines that cover programming and algorithms (e.g., bioinformatics, engineering, mathematics, physics, etc.). The book could also be useful for amateur programmers, students of massive open online courses, or more experienced professionals who would like to refresh the material, or learn it in a different or clearer way.

Students should have some basic programming experience in order to understand the code in the book. The reader should be familiar with notions introduced in a first programming course such as expressions, variables, conditional and loop constructs, methods, parameters, or elementary data structures such as arrays or lists. These concepts are not explained in the book. Also, the code in the book is in accordance with the procedural programming paradigm, and does not use object oriented programming features. Regarding Python, a basic background can be helpful, but is not strictly necessary. Lastly, the student should be competent in high school mathematics.

Computer science professors can also benefit from the book, not just as a handbook with a large collection and variety of problems, but also by adopting the methodology and diagrams described to build recursive solutions. Furthermore, professors may employ its structure to organize their classes. The book could be used as a required textbook in introductory (CS1/2) programming courses, and in more advanced classes on the design and analysis of algorithms (for example, it covers topics such as

divide and conquer, or backtracking). Additionally, since the book provides a solid foundation of recursion, it can be used as a complementary text in courses related to data structures, or as an introduction to functional programming. However, the reader should be aware that the book does not cover data structures or functional programming concepts.

BOOK CONTENT AND ORGANIZATION

The first chapter assumes that the reader does not have any previous background on recursion, and introduces fundamental concepts, notation, and the first coded examples.

The second chapter presents a methodology for developing recursive algorithms, as well as diagrams designed to help thinking recursively, which illustrate the original problem and its decomposition into smaller instances of the same problem. It is one of the most important chapters since the methodology and recursive diagrams will be used throughout the rest of the book. Readers are encouraged to read the chapter, regardless of their previous background on recursion.

[Chapter 3](#) reviews essential mathematical fundamentals and notation. Moreover, it describes methods for solving recurrence relations, which are the main mathematical tools for theoretically analyzing the computational cost of recursive algorithms. The chapter can be skipped when covering recursion in an introductory course. However, it is included early in the book in order to provide a context for characterizing and comparing different algorithms regarding their efficiency, which would be essential in a more advanced course on design and analysis of algorithms.

The fourth chapter covers “linear recursion.” This type of recursion leads to the simplest recursive algorithms, where the solutions to computational problems are obtained by considering the solution to a single smaller instance of the problem. Although the proposed problems can also be solved easily through iteration, they are ideal candidates for introducing fundamental recursive concepts, as well as examples of how to use the methodology and recursive diagrams.

The fifth chapter covers a particular type of linear recursion called “tail recursion,” where the last action performed by a method is a recursive call, invoking itself. Tail recursion is special due to its relationship with iteration. This connection will nevertheless be postponed until [Chapter 11](#). Instead, this chapter focuses on solutions from a purely declarative approach, relying exclusively on recursive concepts.

The advantages of recursion over iteration are mainly due to the use of “multiple recursion,” where methods invoke themselves several times, and the algorithms are based on combining several solutions to smaller instances of the same problem. [Chapter 6](#) introduces multiple recursion through methods based on the eminent “divide and conquer” algorithm design paradigm. While some examples can be used in an introductory programming course, the chapter is especially appropriate in a more advanced class on algorithms. Alternatively, [Chapter 7](#) contains challenging problems, related to puzzles and fractal images, which can also be solved through multiple recursion, but are not considered to follow the divide and conquer approach.

Recursion is used extensively in combinatorics, which is a branch of mathematics related to counting that has applications in advanced analysis of algorithms. [Chapter 8](#) proposes using recursion for solving combinatorial counting problems, which are usually not covered in programming texts. This unique chapter will force the reader to apply the acquired recursive thinking skills to a different family of problems. Lastly, although some examples are challenging, many of the solutions will have appeared in earlier chapters. Thus, some examples can be used in an introductory programming course.

[Chapter 9](#) introduces “mutual recursion,” where several methods invoke themselves indirectly. The solutions are more sophisticated since it is necessary to think about several problems simultaneously. Nevertheless, this type of recursion involves applying the same essential concepts covered in earlier chapters.

[Chapter 10](#) covers how recursive programs work from a low-level point of view. It includes aspects such as tracing and debugging, the program stack, or recursion trees. In addition, it contains a brief introduction to memoization and dynamic programming, which is another important algorithm design paradigm.

Tail-recursive algorithms can not only be transformed to iterative versions; some are also designed by thinking iteratively. [Chapter 11](#) examines the connection between iteration and tail recursion in detail. In addition, it provides a brief introduction to “nested recursion,” and includes a strategy for designing simple tail-recursive functions that are usually defined by thinking iteratively, but through a purely declarative approach. These last two topics are curiosities regarding recursion, and should be skipped in introductory courses.

The last chapter presents backtracking, which is another major algorithm design technique that is used for searching for solutions to com-

putational problems in large discrete state spaces. The strategy is usually applied for solving constraint satisfaction and discrete optimization problems. For example, the chapter will cover classical problems such as the N-queens puzzle, finding a path through a maze, solving sudokus, or the 0-1 knapsack problem.

POSSIBLE COURSE ROAD MAPS

It is possible to cover only a subset of the chapters. The road map for introductory programming courses could be [Chapters 1, 2, 4, 5](#), and [10](#). The instructor should decide whether to include examples from [Chapters 6–9](#), and whether to cover the first section of [Chapter 11](#).

If students have previously acquired skills to develop linear-recursive methods, a more advanced course on algorithm analysis and design could cover [Chapters 2, 3, 5, 6, 7, 9, 11](#), and [12](#). Thus, [Chapters 1, 4](#), and [10](#) could be proposed as readings for refreshing the material. In both of these suggested road maps [Chapter 8](#) is optional. Finally, it is important to cover [Chapters 10](#) and [11](#) after the previous ones.

ACKNOWLEDGEMENTS

The content of this book has been used to teach computer programming courses at Universidad Rey Juan Carlos, in Madrid (Spain). I am grateful to the students for their feedback and suggestions. I would also like to thank Ángel Velázquez and the members of the LITE (Laboratory of Information Technologies in Education) research group for providing useful insights regarding the content of the book. I would also like to express my gratitude to Luís Fernández, computer science professor at Universidad Politécnica de Madrid, for his advice and experience related to teaching recursion. A special thanks to Gert Lanckriet and members of the Computer Audition Laboratory at University of California, San Diego.

Manuel Rubio-Sánchez
July, 2017

List of Figures

1.1	Examples of recursive entities.	2
1.2	Recursive decomposition of lists and trees.	6
1.3	Family tree representing the descendants of a person, which are its children plus the descendants of its children.	7
1.4	Recursive problem solving.	8
1.5	Decompositions of the sum of the first positive integers.	9
1.6	Decompositions of the sum of the elements in a list, denoted as \mathbf{a} , of $(n = 9)$ numbers.	12
1.7	Functions that compute the sum of the first n natural numbers in several programming languages.	15
1.8	Data structures similar to lists, and parameters necessary for defining sublists.	18
1.9	Thought experiment in a classroom, where an instructor asks a student to add the first 100 positive integers. $S(n)$ represents the sum of the first n positive integers.	23
2.1	General template for designing recursive algorithms.	32
2.2	Additional diagrams that illustrate the decomposition of the sum of the first positive integers when the problem size is decreased by a unit.	38
2.3	When thinking recursively we generally do not need to decompose a problem into every instance of smaller size.	39
2.4	Decompositions of the Fibonacci function.	40

- 2.5 General diagram for thinking about recursive cases (when a problem is decomposed into a single self-similar subproblem). 42
- 2.6 Diagram showing a decomposition of the sum of the first n positive integers $S(n)$ that uses two subproblems of half the size as the original. 45
- 2.7 Diagram showing a decomposition of the problem consisting of printing the digits of a nonnegative integer on the console, in reversed order, and vertically. A particular ($n = 2743$) and a general m -digit ($n = d_{m-1} \cdots d_1 d_0$) case are shown in (a) and (b), respectively. 49
- 2.8 General diagram for thinking about recursive cases, when a problem is decomposed into several (N) self-similar subproblems. 50
- 2.9 Alternative diagram showing a divide and conquer decomposition, and the recursive thought process, for the problem of finding the largest value in a list. The thick and thin arrows point to the solutions of the problem and subproblems, respectively. 50
- 2.10 Diagram showing a decomposition of the sum of the first n positive integers $S(n)$ that uses two subproblems of (roughly) half the size as the original, when n is odd. 54
- 3.1 Graphical mnemonic for determining the quadratic formula for the sum of the first n positive integers ($S(n)$). 63
- 3.2 Right triangle used for showing trigonometric definitions. 67
- 3.3 Geometric interpretation of vector addition and subtraction. 69
- 3.4 Rotation matrix (counterclockwise). 70
- 3.5 The highest-order term determines the order of growth of a function. For $T(n) = 0.5n^2 + 2000n + 50000$ the order is quadratic, since the term $0.5n^2$ clearly dominates the lower-order terms (even added up) for large values of n . 71

3.6	Orders of growth typically used in computational complexity.	72
3.7	Graphical illustrations of asymptotic notation definitions for computational complexity.	74
3.8	Sequence of operations carried by the function in Listing 1.1 in the base case.	77
3.9	Sequence of operations carried by the function in Listing 1.1 in the recursive case.	78
3.10	Summary of the expansion method.	81
3.11	An algorithm processes the shaded bits of numbers from 1 to $2^n - 1$ (for $n = 4$).	102
4.1	Conversion of 142 into its representation (1032) in base 5.	118
4.2	Pascal's triangle.	126
4.3	Decomposition and recovery of a row of Pascal's triangle.	126
4.4	Ladder of resistors problem.	127
4.5	Equivalence of circuits with resistors.	127
4.6	Decomposition of the ladder of resistors problem, and derivation of the recursive case through induction.	128
4.7	The product of two nonnegative integers n and m can be represented as the number of unit squares that form an $n \times m$ rectangle.	130
4.8	Step of the insertion sort algorithm.	132
5.1	Decomposition related to the binary search algorithm.	141
5.2	Binary search tree that stores information about a birthday calendar.	145
5.3	Binary search tree in Figure 5.2 and (5.1), where each node is a list of four elements: name (string), birthday (string), left subtree (list), and right subtree (list).	145
5.4	Decomposition associated with several algorithms related to binary search trees.	146
5.5	Partitioning of a list used in the quicksort and quick-select algorithms.	148

5.6	Example of Hoare's partition method.	151
5.7	Decomposition of Hoare's partitioning problem.	153
5.8	Base case and problem decomposition used by the quickselect algorithm.	155
5.9	Steps of the bisection algorithm.	158
5.10	Base case of the bisection algorithm ($b - a \leq 2\epsilon$).	159
5.11	Instance of the woodcutter problem.	160
5.12	Decomposition of the woodcutter problem.	162
5.13	Steps of the binary search algorithm related to an instance of the woodcutter problem, for $w = 10$.	164
5.14	Steps in the counting sort algorithm.	167
5.15	Main idea behind Newton's method.	169
6.1	Merge sort algorithm.	174
6.2	Decomposition of the quicksort algorithm.	178
6.3	Types of trominoes ignoring rotations and reflections.	192
6.4	Tromino tiling problem.	192
6.5	Decomposition of the tromino tiling problem.	192
6.6	L trominoes considering rotations.	195
6.7	The skyline problem.	197
6.8	Base case for the skyline problem with one building.	198
6.9	Recursive case for the skyline problem.	199
6.10	Possible situations when merging skylines that change at the same location x .	200
6.11	Possible situations when merging skylines and $x_1 < x_2$.	202
7.1	Swamp traversal problem.	206
7.2	Decomposition of the swamp traversal problem.	207
7.3	The towers of Hanoi puzzle.	209
7.4	Solution to the towers of Hanoi puzzle for $n = 2$ disks.	210
7.5	Solution to the towers of Hanoi puzzle for $n = 4$ disks.	212
7.6	Decomposition of the towers of Hanoi problem.	213
7.7	Output of Listing 7.3, which represents the solution to the towers of Hanoi puzzle for $n = 4$ disks.	214

7.8	Concrete example of the decomposition of the in-order traversal problem.	215
7.9	Decomposition of the problem of finding the longest palindrome substring.	218
7.10	Koch curve fractal.	221
7.11	Koch snowflake fractal.	222
7.12	Koch curve decomposition.	223
7.13	New endpoints of shorter segments when applying an iteration related to the Koch curve.	224
7.14	Sierpiński's carpet after 0, 1, 2, and 3 iterations.	226
7.15	Sierpiński's carpet decomposition.	227
7.16	Simulation of tick marks on an English ruler.	229
7.17	Rules for the "sideways" variant of the towers of Hanoi puzzle.	230
7.18	Sierpiński's triangle after 0, 1, 2, and 3 iterations.	231
7.19	Hilbert curves of orders 1–6.	232
8.1	Possible permutations (lists) of the first four positive integers.	236
8.2	Decomposition of the problem that counts the number of possible permutations of n different elements, denoted as $f(n)$.	237
8.3	Decomposition of the possible permutations of the first four positive integers.	238
8.4	Example of a k -element variation with repetition of n items.	239
8.5	Decomposition of the problem that counts the number of k -element variations with repetition of n items.	240
8.6	Decomposition of the problem that counts the number of k -element combinations of n items.	241
8.7	A possible way to climb a staircase by taking leaps of one or two steps.	242
8.8	Decomposition of the problem that counts the number of ways to climb a staircase by taking leaps of one or two steps.	243

8.9	Manhattan paths problem.	244
8.10	Decomposition of the Manhattan paths problem.	245
8.11	Two possible triangulations of the same convex polygon containing seven vertices.	246
8.12	Six $(n - 2)$ possible triangles associated with an edge of an octagon ($n = 8$).	246
8.13	Decomposition of the convex polygon triangulation problem for fixed triangles.	247
8.14	Total number of triangulations related to an octagon.	247
8.15	Valid and invalid circle pyramids.	248
8.16	Pyramids of $n = 4$ circles on the bottom row, grouped according to how many circles appear in the row immediately above it.	249
8.17	Decomposition of the circle pyramids problem for subproblems of a fixed size.	250
8.18	Two-element variations with repetition of the four items in $\{a, b, c, d\}$.	250
8.19	Tiling of a 2×10 rectangle with 1×2 or 2×1 domino tiles.	251
8.20	Five different binary trees that contain three nodes.	252
8.21	Pyramids of $n = 4$ circles on the bottom row, grouped according to their height.	252
9.1	Calls of mutually recursive methods.	254
9.2	Illustration of the rabbit population growth rules.	257
9.3	Rabbit family tree after seven months.	259
9.4	Concrete example of the decomposition of the rabbit population growth problem into self-similar subproblems.	260
9.5	Concrete decompositions of the problems that lead to two mutually recursive methods for solving the rabbit population growth problem.	261
9.6	Water treatment plants puzzle.	263
9.7	Decomposition of the water treatment plants puzzle when modeling the water flow between cities.	264
9.8	Three problems for the mutually recursive solution of the water treatment plants puzzle.	266

9.9	Decompositions of the three problems for the mutually recursive solution of the water treatment plants puzzle.	267
9.10	The cyclic towers of Hanoi puzzle.	268
9.11	Illustration of two different problems comprised in the cyclic towers of Hanoi puzzle.	269
9.12	Three operations used to implement the recursive methods for solving the cyclic towers of Hanoi puzzle.	270
9.13	Operations for moving n disks clockwise.	271
9.14	Operations for moving n disks counterclockwise.	272
9.15	Decomposition of a mathematical formula into categories such as expressions, terms, factors, or numbers.	280
9.16	Method calls of the mutually recursive functions that implement the recursive descent parser associated with the calculator program.	282
9.17	Possible decompositions of an expression.	284
9.18	Rules of the Spin-out [®] brainteaser challenge.	289
10.1	Sequence of method calls and returns for the code in Listing 10.1.	293
10.2	Sequence of calls and returns for <code>sum_first_naturals(4)</code> .	294
10.3	Sequence of calls and returns for the procedure <code>mystery_method_1('Word')</code> .	296
10.4	Sequence of calls and returns for the procedure <code>mystery_method_2('Word')</code> .	298
10.5	Recursion trees for <code>sum_first_naturals(4)</code> .	299
10.6	Activation tree for <code>sum_first_naturals(4)</code> .	299
10.7	Activation tree for <code>gcd1(20,24)</code> .	300
10.8	Recursion (a) and activation (b) trees for <code>fibonacci(6)</code> .	301
10.9	Activation tree for the mutually recursive functions in Listing 9.1.	302
10.10	Recursion tree for the mutually recursive functions in (1.17) and (1.18).	303
10.11	Activation tree for the nested recursive function in (1.19).	304

10.12	Recursion tree associated with the recurrence $T(n) = 2T(n/2) + n^2$.	305
10.13	The stack and queue data structures.	306
10.14	Evolution of stack frames when running the code in Listing 10.1, where <code>cos</code> , <code> · </code> , and <code><·,·></code> represent the methods <code>cosine</code> , <code>norm_Euclidean</code> , and <code>dot_product</code> , respectively.	307
10.15	Program stack and data at step 5 in Figure 10.14.	308
10.16	Evolution of stack frames for <code>sum_first_naturals(4)</code> .	309
10.17	Evolution of stack frames for <code>fibonacci(5)</code> .	311
10.18	File system tree example.	313
10.19	State of a stack data structure when running the iterative code in Listing 10.3, for the files and folders in Figure 10.18.	315
10.20	Evolution of stack frames when running the code in Listing 10.4, for the files and folders in Figure 10.18.	317
10.21	Overlapping subproblems when computing Fibonacci numbers through $F(n) = F(n - 1) + F(n - 2)$.	319
10.22	Dependency graph for $F(n) = F(n - 1) + F(n - 2)$.	322
10.23	Dependency graph for Listing 10.6, which solves the longest palindrome substring problem.	323
10.24	Matrix L after running Listing 10.7 with <code>s = 'bcaac'</code> .	325
10.25	Alternative binary search tree that stores information about a birthday calendar.	326
10.26	Transformation of a line segment into five smaller ones for a Koch curve variant.	330
10.27	“Koch square” variant for $n = 4$.	330
11.1	State of the program stack when running the base case relative to a call to <code>gcd1(20, 24)</code> .	334
11.2	Similarities between the iterative and tail-recursive codes that compute the factorial function.	339
11.3	McCarthy 91 function.	343
12.1	One solution to the four-queens puzzle.	354

12.2	Partial solutions within complete solutions that are coded as lists or matrices.	355
12.3	Recursion tree of a backtracking algorithm that finds one solution to the four-queens puzzle.	356
12.4	Binary recursion tree of an algorithm that generates all of the subsets of three items.	359
12.5	Alternative binary recursion tree of an algorithm that generates all of the subsets of three items.	362
12.6	Recursion tree of an algorithm that generates all of the permutations of three items.	364
12.7	Pruning a recursion tree as soon as a partial solution is not valid.	367
12.8	Indexing principal and secondary diagonals on a matrix or chessboard.	369
12.9	Recursion tree of the procedure that solves the subset sum problem for $S = \{2, 6, 3, 5\}$ and $x = 8$.	376
12.10	Problem of finding a path through a maze, and solution through backtracking when searching in a particular order.	378
12.11	Different paths through a maze, depending on the search order.	379
12.12	Decomposition of the problem of finding a path through a maze.	380
12.13	An instance of the sudoku puzzle and its solution.	384
12.14	Recursive cases for the sudoku solver.	385
12.15	Recursion tree of a backtracking algorithm for the 0-1 knapsack problem.	390
12.16	Recursion tree of a branch and bound algorithm for the 0-1 knapsack problem.	395
12.17	Alternative recursion tree of an algorithm that generates all of the subsets of three items.	398
12.18	One solution to the four-rooks puzzle.	398
12.19	A 3×3 magic square.	399
12.20	Chess knight moves.	399

12.21	An instance and solution to the traveling salesman problem.	400
12.22	Two ways to represent a solution for the tug of war problem.	401

List of Tables

3.1	Concrete values of common functions used in computational complexity.	72
11.1	Program state related to the iterative factorial function when computing $4!$.	338
11.2	Program state related to the iterative base conversion function in Listing 11.5, when obtaining the base-5 representation of 142, which is 1032.	340



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

List of Listings

1.1	Python code for adding the first n natural numbers.	16
1.2	Alternative Python code for adding the first n natural numbers.	16
1.3	Python code for computing the n -th Fibonacci number.	17
1.4	Alternative Python code for computing the n -th Fibonacci number.	17
1.5	Recursive functions for adding the elements in a list \mathbf{a} , where the only input to the recursive function is the list.	19
1.6	Alternative recursive functions for adding the elements in a sublist of a list \mathbf{a} . The boundaries of the sublist are specified by two input parameters that mark lower and upper indices in the list.	21
2.1	Misconceptions regarding base cases through the factorial function.	35
2.2	Code for computing the sum of the digits of a non-negative integer.	47
2.3	Code for printing the digits of a nonnegative integer vertically, and in reversed order.	49
2.4	Code for computing the maximum value in a list, through a divide and conquer approach.	51
2.5	Erroneous Python code for determining if a nonnegative integer n is even.	52
2.6	Correct Python code for determining if a nonnegative integer n is even.	52
2.7	Erroneous Python code for adding the first n positive numbers, which produces infinite recursions for most values of n .	53

2.8	Incomplete Python code for adding the first n positive numbers.	54
2.9	Python code for adding the first n positive numbers based on using two subproblems of (roughly) half the size as the original.	55
3.1	Measuring execution times through Python's <code>time</code> module.	70
3.2	Solving a system of linear equations, $\mathbf{Ax} = \mathbf{b}$, in Python.	94
4.1	Power function in linear time for nonnegative exponents.	107
4.2	Power function in linear time.	108
4.3	Power function in logarithmic time for nonnegative exponents.	109
4.4	Inefficient implementation of the power function that runs in linear time.	109
4.5	Slow addition of two nonnegative integers.	111
4.6	Quicker slow addition of two nonnegative integers.	112
4.7	Alternative quicker slow addition of two nonnegative integers.	113
4.8	Recursive functions that compute the double sum in (4.3).	115
4.9	Binary representation of a nonnegative integer.	117
4.10	Conversion of a nonnegative integer n into its representation in base b .	118
4.11	Conversion of a nonnegative integer n into its representation in base b .	120
4.12	Function that determines if a string is a palindrome.	121
4.13	Recursive selection sort algorithm.	123
4.14	Recursive variant of the selection sort algorithm.	123
4.15	Horner's method for evaluating a polynomial.	125
4.16	Function that generates the n -th row of Pascal's triangle.	126
4.17	Function that solves the ladder of resistors problem.	129
5.1	Linear-recursive Boolean function that determines if a nonnegative integer contains a digit.	135
5.2	Tail-recursive Boolean function that determines if a nonnegative integer contains a digit.	136

5.3	Linear-recursive function that determines if two strings are identical.	138
5.4	Tail-recursive function that determines if two strings are identical.	138
5.5	Tail-recursive linear search of an element in a list.	139
5.6	Linear-recursive linear search of an element in a list.	140
5.7	Alternative tail-recursive linear search of an element in a list.	141
5.8	Binary search of an element in a list.	142
5.9	Algorithm for searching an item with a particular key in a binary search tree.	146
5.10	Procedure for inserting an item with a particular key in a binary search tree.	147
5.11	Auxiliary methods for partitioning a list.	150
5.12	Hoare's iterative partitioning algorithm.	152
5.13	Alternative recursive version of Hoare's partitioning scheme.	154
5.14	Tail-recursive quickselect algorithm.	156
5.15	Bisection algorithm.	159
5.16	Function that computes the amount of wood collected when cutting trees at height h .	160
5.17	Binary search algorithm for the woodcutter problem.	163
5.18	Euclid's algorithm for computing the greatest common divisor of two nonnegative integers.	165
6.1	Function that determines whether a list is sorted in ascending order.	172
6.2	Merge sort method.	175
6.3	Method for merging two sorted lists.	177
6.4	Variant of the quicksort algorithm.	179
6.5	In-place quicksort algorithm.	179
6.6	Code for counting the number of times an element appears in a list.	182
6.7	Code for solving the majority element problem.	183

6.8	Karatsuba's fast algorithm for multiplying two non-negative integers.	185
6.9	Divide and conquer matrix multiplication.	188
6.10	Alternative divide and conquer matrix multiplication.	189
6.11	Auxiliary functions for drawing trominoes.	193
6.12	Recursive method for drawing trominoes.	194
6.13	Code for calling the trominoes method.	195
6.14	Main recursive method for computing skylines.	198
6.15	Recursive method for merging skylines.	201
7.1	Function that determines whether there exists a path through a swamp.	207
7.2	Alternative function that determines whether there exists a path through a swamp.	208
7.3	Towers of Hanoi procedure.	214
7.4	Inorder traversal of a binary tree.	216
7.5	Preorder and postorder traversals of a binary tree.	216
7.6	Code for finding the longest palindrome substring (or sublist).	218
7.7	Alternative code for finding the longest palindrome substring (or sublist).	220
7.8	Code for generating Koch curves and the Koch snowflake.	225
7.9	Code for generating Sierpiński's carpet.	228
9.1	Mutually recursive functions for determining the parity of a nonnegative integer n .	255
9.2	Mutually recursive procedures implementing Alice and Bob's strategies when playing a game.	256
9.3	Mutually recursive functions for counting the population of baby and adult rabbits after n months.	258
9.4	Alternative mutually recursive functions for counting the population of rabbits after n months.	262
9.5	Function based on multiple recursion for solving the water treatment plants puzzle.	265
9.6	Mutually recursive procedures for the cyclic towers of Hanoi puzzle.	273

9.7	Mutually recursive functions for tokenizing a mathematical expression.	275
9.8	Function that checks whether a string represents a number.	276
9.9	Function that parses a mathematical expression of additive terms.	283
9.10	Function that parses a term of multiplicative factors.	285
9.11	Function that parses a term of multiplicative factors, where the first one is a parenthesized expression.	286
9.12	Function that parses a mathematical factor.	287
9.13	Function that parses a parenthesized expression.	287
9.14	Basic code for executing the calculator program.	287
10.1	Methods for computing the cosine of the angle between two vectors.	292
10.2	Similar recursive methods. What do they do?	295
10.3	Iterative algorithm for finding a file in a file system.	314
10.4	Recursive algorithm for finding a file in a file system.	316
10.5	Recursive algorithm for computing Fibonacci numbers in linear time, by using memoization.	320
10.6	Memoized version of Listing 7.7.	321
10.7	Code based on dynamic programming that computes the longest palindrome substring within a string s .	324
10.8	Methods that, supposedly, add and count the digits of a nonnegative integer. Are they correct?	327
10.9	Erroneous code for computing the number of times that two adjacent elements in a list are identical.	328
10.10	Code for computing the smallest prime factor of a number n , which is greater than or equal to m .	328
10.11	Erroneous code for computing the floor of a logarithm.	328
10.12	Erroneous code for determining if a list contains an element that is larger than the sum of all of the rest.	329
10.13	Erroneous code for finding the location of the “peak element.”	330

10.14	Code for generating a Koch fractal based on the transformation in Figure 10.26.	331
10.15	Code for computing the length of the longest palindrome subsequence of a list.	332
11.1	Iterative version of Euclid's method (<code>gcd1</code>).	335
11.2	Iterative version of the bisection method.	336
11.3	Iterative factorial function.	338
11.4	Tail-recursive factorial function and wrapper method.	338
11.5	Iterative conversion of a nonnegative integer n into its representation in base b .	340
11.6	Tail-recursive base conversion function and wrapper method.	341
11.7	The Ackermann function implemented in Python.	341
11.8	Nested-recursive method for finding the digital root of a nonnegative integer.	344
12.1	Code for printing all of the subsets of the elements in a list.	360
12.2	Alternative code for printing all of the subsets of the elements in a list.	363
12.3	Code for printing all of the permutations of the elements in a list.	366
12.4	Alternative code for printing all of the permutations of the elements in a list.	368
12.5	Code for finding all of the solutions to the n -queens puzzle.	371
12.6	Code for finding one solution to the n -queens puzzle.	373
12.7	Backtracking code for solving the subset sum problem.	375
12.8	Backtracking code for finding a path through a maze.	381
12.9	Auxiliary code related to the backtracking methods for finding a path through a maze.	383
12.10	Code for solving a sudoku puzzle.	386
12.11	Auxiliary code for solving a sudoku puzzle.	387
12.12	Backtracking code for solving the 0-1 knapsack problem.	391
12.13	Auxiliary code related to the 0-1 knapsack problem.	393

- 12.14 Branch and bound code for solving the 0-1 knapsack problem. 396
- 12.15 Auxiliary code for the branch and bound algorithm related to the 0-1 knapsack problem. 397



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Basic Concepts of Recursive Programming

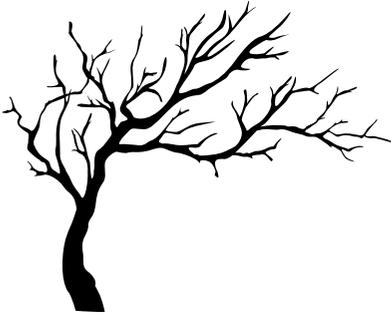
To iterate is human, to recurse divine.

— Laurence Peter Deutsch

RECURSION is a broad concept that is used in diverse disciplines such as mathematics, bioinformatics, or linguistics, and is even present in art or in nature. In the context of computer programming, recursion should be understood as a powerful problem-solving strategy that allows us to design simple, succinct, and elegant algorithms for solving computational problems. This chapter presents key terms and notation, and introduces fundamental concepts related to recursive programming and thinking that will be further developed throughout the book.

1.1 RECOGNIZING RECURSION

An entity or concept is said to be recursive when simpler or smaller self-similar instances form part of its constituents. Nature provides numerous examples where we can observe this property (see [Figure 1.1](#)). For instance, a branch of a tree can be understood as a stem, plus a set of smaller branches that emanate from it, which in turn contain other smaller branches, and so on, until reaching a bud, leaf, or flower. Blood vessels or rivers exhibit similar branching patterns, where the larger structure appears to contain instances of itself at smaller scales. Another related recursive example is a romanesco broccoli, where it is



Tree branches



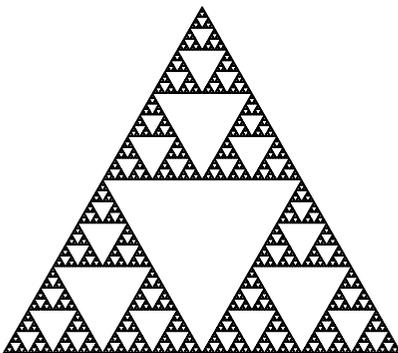
Branching rivers



Romanesco broccoli



Spiral Droste effect



Sierpiński's triangle



Matryoshka dolls

Figure 1.1 Examples of recursive entities.

apparent that the individual florets resemble the entire plant. Other examples include mountain ranges, clouds, or animal skin patterns.

Recursion also appears in art. A well-known example is the Droste effect, which consists of a picture appearing within itself. In theory the process could be repeated indefinitely, but naturally stops in practice when the smallest picture to be drawn is sufficiently small (for example, if it occupies a single pixel in a digital image). A computer-generated fractal is another type of recursive image. For instance, Sierpiński's triangle is composed of three smaller identical triangles that are subsequently decomposed into yet smaller ones. Assuming that the process is infinitely repeated, each small triangle will exhibit the same structure as the original's. Lastly, a classical example used to illustrate the concept of recursion is a collection of matryoshka dolls. In this craftwork each doll has a different size and can fit inside a larger one. Note that the recursive object is not a single hollow doll, but a full nested collection. Thus, when thinking recursively, a collection of dolls can be described as a single (largest) doll that contains a smaller collection of dolls.

While the recursive entities in the previous examples were clearly tangible, recursion also appears in a wide variety of abstract concepts. In this regard, recursion can be understood as the process of defining concepts by using the definition itself. Many mathematical formulas and definitions can be expressed this way. Clear explicit examples include sequences for which the n -th term is defined through some formula or procedure involving earlier terms. Consider the following recursive definition:

$$s_n = s_{n-1} + s_{n-2}. \quad (1.1)$$

The formula states that a term in a sequence (s_n) is simply the sum of the two previous terms (s_{n-1} and s_{n-2}). We can immediately observe that the formula is recursive, since the entity it defines, s , appears on both sides of the equation. Thus, the elements of the sequence are clearly defined in terms of themselves. Furthermore, note that the recursive formula in (1.1) does not describe a particular sequence, but an entire family of sequences in which a term is the sum of the two previous ones. In order to characterize a specific sequence we need to provide more information. In this case, it is enough to indicate any two terms in the sequence. Typically, the first two terms are used to define this type of sequence. For instance, if $s_1 = s_2 = 1$ the sequence is:

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

which is the well-known Fibonacci sequence. Lastly, sequences may also be defined starting at term s_0 .

The sequence s can be understood as a function that receives a positive integer n as an argument, and returns the n -th term in the sequence. In this regard, the Fibonacci function, in this case simply denoted as F , can be defined as:

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

Throughout the book we will use this notation in order to describe functions, where the definitions include two types of expressions or cases. The **base cases** correspond to scenarios where the function's output can be obtained trivially, without requiring values of the function on additional arguments. For Fibonacci numbers the base cases are, by definition, $F(1) = 1$, and $F(2) = 1$. The **recursive cases** include more complex recursive expressions that typically involve the defined function applied to smaller input arguments. The Fibonacci function has one recursive case: $F(n) = F(n-1) + F(n-2)$, for $n > 2$. The base cases are necessary in order to provide concrete values for the function's terms in the recursive cases. Lastly, a recursive definition may contain several base and recursive cases.

Another function that can be expressed recursively is the factorial of some nonnegative integer n :

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

In this case, it is not immediately obvious whether the function can be expressed recursively, since there is not an explicit factorial on the right-hand side of the definition. However, since $(n-1)! = 1 \times 2 \times \cdots \times (n-1)$, we can rewrite the formula as the recursive expression $n! = (n-1)! \times n$. Lastly, by convention $0! = 1$, which follows from plugging in the value $n = 1$ in the recursive formula. Thus, the factorial function can be defined recursively as:

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

Similarly, consider the problem of calculating the sum of the first n positive integers. The associated function $S(n)$ can be obviously defined as:

$$S(n) = 1 + 2 + \cdots + (n-1) + n. \quad (1.4)$$

Again, we do not observe a term involving S on the right-hand side of the definition. However, we can group the $n - 1$ smallest terms in order to form $S(n - 1) = 1 + 2 + \dots + (n - 1)$, which leads to the following recursive definition:

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

Note that $S(n - 1)$ is a self-similar **subproblem** to $S(n)$, but is **simpler**, since it needs fewer operations in order to calculate its result. Thus, we say that the subproblem has a smaller **size**. In addition, we say we have **decomposed** the original problem ($S(n)$) into a **smaller** one, in order to form the recursive definition. Lastly, $S(n - 1)$ is a smaller **instance** of the original problem.

Another mathematical entity for which how it can be expressed recursively may not seem immediately obvious is a nonnegative integer. These numbers can be decomposed and defined recursively in several ways, by considering smaller numbers. For instance, a nonnegative integer n can be expressed as its predecessor plus a unit:

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

Note that n appears on both sides of the equals sign in the recursive case. In addition, if we consider that the *predecessor* function necessarily returns a nonnegative integer, then it cannot be applied to 0. Thus, the definition is completed with a trivial base case for $n = 0$.

Another way to think of (nonnegative) integers consists of considering them as ordered collections of digits. For example, the number 5342 can be the concatenation of the following pairs of smaller numbers:

$$(5, 342), \quad (53, 42), \quad (534, 2).$$

In practice, the simplest way to decompose these integers consists of considering the least significant digit individually, together with the rest of the number. Therefore, an integer can be defined as follows:

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

where $//$ and $\%$ represent the quotient and remainder of an integer division, respectively, which corresponds to Python notation. For example,

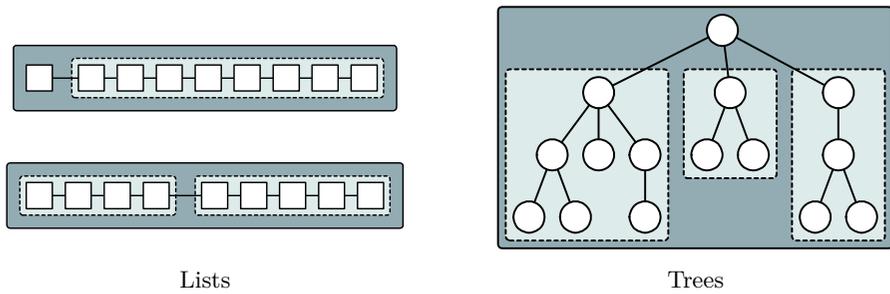


Figure 1.2 Recursive decomposition of lists and trees.

if $n = 5342$ then the quotient is $(n//10) = 534$, while the remainder is $(n\%10) = 2$, which represents the least significant digit of n . Clearly, the number n can be recovered by multiplying the quotient by 10 and adding the remainder. Finally, the base case considers numbers with only one digit that naturally cannot be decomposed any further.

Recursive expressions are abundant in mathematics. For instance, they are often used in order to describe properties of functions. The following recursive expression indicates that the derivative of a sum of functions is the sum of their derivatives:

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

In this case the recursive entity is the derivative function, denoted as $[\cdot]'$, but not the functions $f(x)$ and $g(x)$. Observe that the formula explicitly indicates the decomposition that takes place, where some initial function (which is the input argument to the derivative function) is broken up into the sum of the functions $f(x)$ and $g(x)$.

Data structures can also be understood as recursive entities. [Figure 1.2](#) shows how lists and trees can be decomposed recursively. On the one hand, a list can consist of a single element plus another list (this is the usual definition of a list as an abstract data type), or it can be subdivided into several lists (in this broader context a list is any collection of data elements that are arranged linearly in an ordered sequence, as in lists, arrays, tuples, etc.). On the other hand, a tree consists of a parent node and a set (or list) of subtrees, whose root node is a child of the original parent node. The recursive definitions of data structures are completed by considering empty (base) cases. For instance, a list that contains only one element would consist of that element plus an empty list. Lastly, observe that in these diagrams the darker boxes represent a

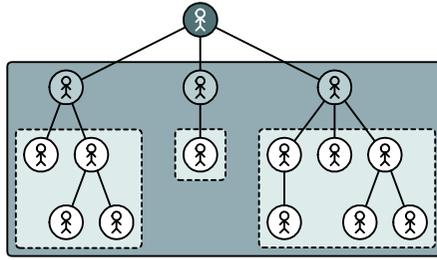


Figure 1.3 Family tree representing the descendants of a person, which are its children plus the descendants of its children.

full recursive entity, while the lighter ones indicate smaller self-similar instances.

Recursion can even be used to define words in dictionaries. This might seem impossible since we are told in school that the description of a word in a dictionary should not contain the word itself. However, many concepts can be defined correctly this way. Consider the term “descendant” of a specific ancestor. Notice that it can be defined perfectly as: someone who is either a child of the ancestor, or a *descendant* of any of the ancestor’s children. In this case, we can identify a recursive structure where the set of descendants can be organized in order to form a (family) tree, as shown in Figure 1.3. The darker box contains all of the descendants of a common ancestor appearing at the root of the tree, while the lighter boxes encompass the descendants of the ancestor’s children.

1.2 PROBLEM DECOMPOSITION

In general, when programming and thinking recursively, our main task will consist of providing our own recursive definitions of entities, concepts, functions, problems, etc. While the first step usually involves establishing the base cases, the main challenge consists of describing the recursive cases. In this regard, it is important to understand the concepts of: (a) problem decomposition, and (b) induction, which we will cover briefly in this chapter.

The book will focus on developing recursive algorithms for solving **computational problems**. These can be understood as questions that computers could possibly answer, and are defined through statements that describe relationships between a collection of known input values or parameters, and a set of output values, results, or solutions. For example,

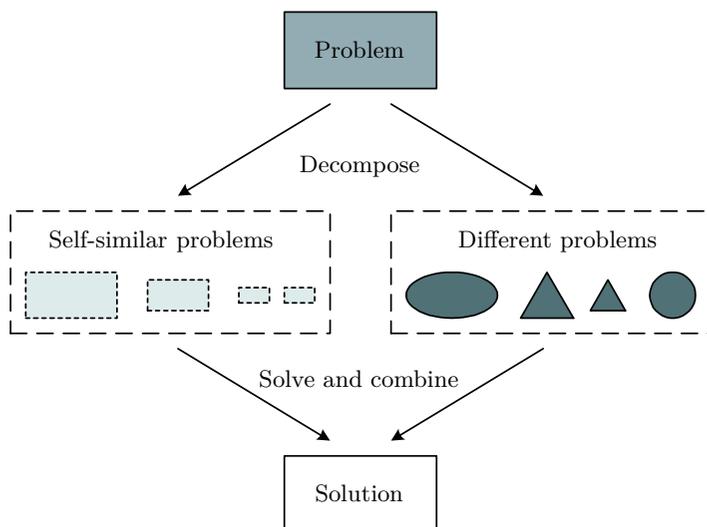


Figure 1.4 Recursive problem solving.

“given some positive integer n , calculate the sum of the first n positive integers” is the statement of a computational problem with one input parameter (n), and one output value defined as $1 + 2 + \dots + (n - 1) + n$. An instance of a problem is a specific collection of valid input values that will allow us to compute a solution to the problem. In contrast, an **algorithm** is a logical procedure that describes a step-by-step set of computations needed in order to obtain the outputs, given the initial inputs. Thus, an algorithm determines how to solve a problem. It is worth mentioning that computational problems can be solved by different algorithms. The goal of this book is to explain how to design and implement recursive algorithms and programs, where a key step involves decomposing a computational problem.

Decomposition is an important concept in computer science and plays a major role not only in recursive programming, but also in general problem solving. The idea consists of breaking up complex problems into smaller, simpler ones that are easier to express, compute, code, or solve. Subsequently, the solutions to the subproblems can be processed in order to obtain a solution to the original complex problem.

In the context of recursive problem solving and programming, decomposition involves breaking up a computational problem into several subproblems, some of which are self-similar to the original, as illustrated in [Figure 1.4](#). Note that obtaining the solution to a problem may re-

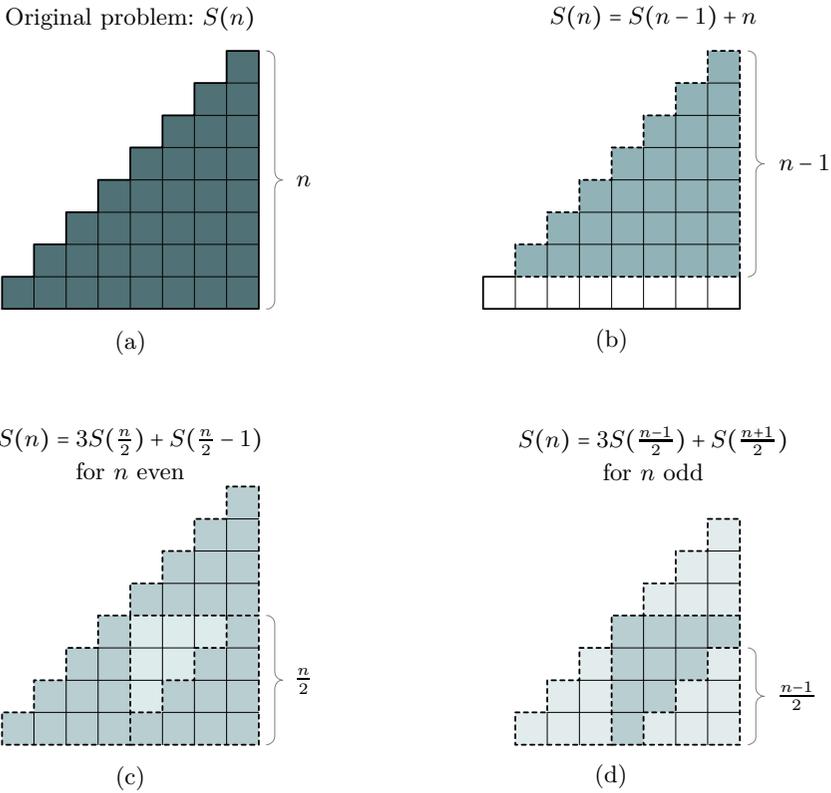


Figure 1.5 Decompositions of the sum of the first positive integers.

quire solving additional different problems that are not self-similar to the original one. We will tackle several of these problems throughout the book, but in this introductory chapter we will examine examples where the original problems will only be decomposed into self-similar ones.

For the first example we will reexamine the problem of computing the sum of the first n positive integers, denoted as $S(n)$, which can be formally expressed as in (1.4). There are several ways to break down the problem into smaller subproblems and form a recursive definition of $S(n)$. Firstly, it only depends on the input parameter n , which also specifies the size of the problem. In this example, the base case is associated with the smallest positive integer $n = 1$, where clearly $S(1) = 1$ is the smallest instance of the problem. Furthermore, we need to get closer to the problem's base case when considering subproblems. Therefore, we have to think of how we can reduce the input parameter n .

A first possibility consists of decreasing n by a single unit. In that case, the goal would be to define $S(n)$ somehow by using the subproblem $S(n - 1)$. The corresponding recursive solution, derived in [Section 1.1](#) algebraically, is given in [\(1.5\)](#). We can also obtain the recursive case by analyzing a graphical description of the problem. For instance, the goal could consist of counting the total number of blocks in a “triangular” structure that contains n blocks in its first level, $n - 1$ in its second, and so on (the n -th level would therefore have a single block), as shown in [Figure 1.5\(a\)](#) for $n = 8$. In order to decompose the problem recursively we need to find self-similar problems. In this case, it is not hard to find smaller similar triangular shapes inside the original. For example, [Figure 1.5\(b\)](#) shows a triangular structure of height $n - 1$ that contains all of the blocks of the original, except for the n blocks on the first level. Since this smaller triangular shape contains exactly $S(n - 1)$ blocks, it follows that $S(n) = S(n - 1) + n$.

Another option that involves using a problem that adds $n - 1$ terms consists of considering the sum $2 + \dots + (n - 1) + n$. However, it is important to note that it is not a self-similar problem to $S(n)$. Clearly, it is not the sum of the first positive integers. Instead, it is a special case of a more general problem consisting of adding all of the integers from some initial value m up to another larger n : $m + (m + 1) + \dots + (n - 1) + n$, for $m \leq n$. The difference between both problems can also be understood graphically. Regarding the illustrations in [Figure 1.5](#), this general problem would define right trapezoid structures instead of triangular shapes. Finally, it is possible to compute the sum of the first n positive integers by using this more general problem, since we could simply set $m = 1$. Nevertheless, its recursive definition is slightly more complex since it requires two input parameters (m and n) instead of one.

Other possibilities concern decreasing n by larger quantities. For example, the input value n can be divided by two in order to obtain the decomposition illustrated in [Figures 1.5\(c\)](#) and [\(d\)](#). When n is even we can fit three triangular structures of height $n/2$ inside the larger one corresponding to $S(n)$. Since the remaining blocks also form a triangular shape of height $n/2 - 1$ the recursive formula can be expressed as $S(n) = 3S(n/2) + S(n/2 - 1)$. Alternatively, when n is odd it is possible to fit three triangular structures of height $(n - 1)/2$, and one of size $(n + 1)/2$. Thus, in that case the recursive formula is

$S(n) = 3S((n-1)/2) + S((n+1)/2)$. The final recursive function is:

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3 & \text{if } n = 2, \\ 3S\left(\frac{n}{2}\right) + S\left(\frac{n}{2} - 1\right) & \text{if } n > 2 \text{ and } n \text{ is even,} \\ 3S\left(\frac{n-1}{2}\right) + S\left(\frac{n+1}{2}\right) & \text{if } n > 2 \text{ and } n \text{ is odd.} \end{cases} \quad (1.6)$$

It is important to note that the definition needs an additional base case for $n = 2$. Without it we would have $S(2) = 3S(1) + S(0)$ due to the recursive case when n is even. However, $S(0)$ is not defined since, according to the statement of the problem, the input to S must be a positive integer. The new base case is therefore necessary in order to avoid using the recursive formula for $n = 2$.

When dividing the size of a problem the resulting subproblems are considerably smaller than the original, and can therefore be solved much faster. Roughly speaking, if the number of subproblems to be solved is small and it is possible to combine their solutions efficiently, this strategy can lead to substantially faster algorithms for solving the original problem. However, in this particular example, code for (1.6) is not necessarily more efficient than the one implementing (1.5). Intuitively, this is because (1.6) requires solving two subproblems (with different arguments), while the decomposition in (1.5) only involves one subproblem. Chapter 3 covers how to analyze the runtime cost of these recursive algorithms.

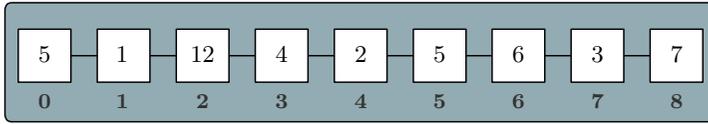
The previous idea for adding the first n positive integers broke up the problem into two subproblems of smaller size, where the new input parameters approach the specified base cases. In general, we can decompose problems into any number of simpler subproblems, as long as the new parameters get closer to the values specified in the base cases. For instance, consider the following alternative recursive definition of the Fibonacci function (it is equivalent to (1.2)):

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

where $\sum_{i=1}^{n-2} F(i)$ is the sum $F(1) + F(2) + \dots + F(n-2)$ (see Section 3.1.4). In this example, for some value n that determines the size of the problem,

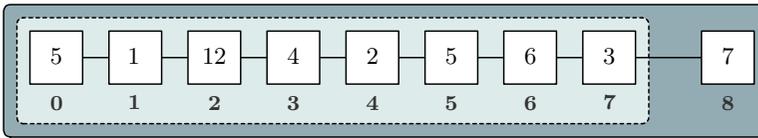
12 ■ Introduction to Recursive Programming

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



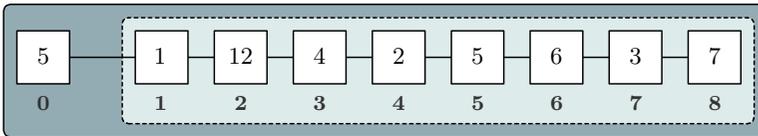
(a)

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



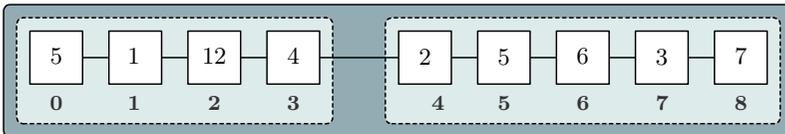
(b)

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



(c)

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



(d)

Figure 1.6 Decompositions of the sum of the elements in a list, denoted as \mathbf{a} , of $(n = 9)$ numbers.

the recursive case relies on decomposing the original problem into every smaller problem of size 1 up to $n - 2$.

For the last example of problem decomposition we will use lists that allow us to access individual elements by using numerical indices (in many programming languages this data structure is called an “array,” while in Python it is simply a “list”). The problem consists of adding

the elements of a list, denoted as \mathbf{a} , containing n numbers. Formally, the problem can be expressed as:

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

where $\mathbf{a}[i]$ is the $(i+1)$ -th element in the list, since the first one is not indexed by 1, but by 0. [Figure 1.6\(a\)](#) shows a diagram representing a particular instance of 9 elements.

Regarding notation, in this book we will assume that a **sublist** of some list \mathbf{a} is a collection of *contiguous* elements of \mathbf{a} , unless explicitly stated otherwise. In contrast, in a **subsequence** of some initial sequence \mathbf{s} its elements appear in the same order as in \mathbf{s} , but they are not required to be contiguous in \mathbf{s} . In other words, a subsequence can be obtained from an original sequence \mathbf{s} by deleting some elements of \mathbf{s} , and without modifying the order of the remaining elements.

The problem can be decomposed by decreasing its size by a single unit. On the one hand, the list can be broken down into the sublist containing the first $n-1$ elements ($\mathbf{a}[0:n-1]$, where $\mathbf{a}[i:j]$ denotes the sublist from $\mathbf{a}[i]$ to $\mathbf{a}[j-1]$, following Python's notation) and a single value corresponding to the last number on the list ($\mathbf{a}[n-1]$), as shown in [Figure 1.6\(b\)](#). In that case, the problem can be defined recursively through:

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

In the recursive case the subproblem is naturally applied to the sublist of size $n-1$. The base case considers the trivial situation when the list is empty, which does not require any addition. Another possible base case can be $s(\mathbf{a}) = \mathbf{a}[0]$ when $n = 1$. However, it would be redundant in this decomposition, and therefore not necessary. Note that if $n = 1$ the function adds $\mathbf{a}[0]$ and the result of applying the function on an empty list, which is 0. Thus, it can be omitted for the sake of conciseness.

On the other hand, we can also interpret that the original list is its first element $\mathbf{a}[0]$, together with the smaller list $\mathbf{a}[1:n]$, as illustrated in [Figure 1.6\(c\)](#). In this case, the problem can be expressed recursively through:

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

Although both decompositions are very similar, the code for each one can be quite different depending on the programming language used.

Section 1.3 will show several ways of coding algorithms for solving the problem according to these decompositions.

Another way to break up the problem consists of considering each half of the list separately, as shown in Figure 1.6(d). This results in two subproblems of roughly half the size of the original's. The decomposition produces the following recursive definition:

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

Unlike the previous definitions, this decomposition requires a base case when $n = 1$. Without it the function would never return a concrete value for a nonempty list. Observe that the definition would not add or return any element of the list. For a nonempty list the recursive case would be applied repeatedly, but the process would never halt. This situation is denoted as **infinite recursion**. For instance, if the list contained a single element the recursive case would add the value associated with an empty list (0), to the result of the same initial problem. In other words, we would try to calculate $s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) \dots$, which would be repeated indefinitely. The obvious issue in this scenario is that the original problem $s(\mathbf{a})$ is not decomposed into smaller and simpler ones when $n = 1$.

1.3 RECURSIVE CODE

In order to use recursion when designing algorithms it is crucial to learn how to decompose problems into smaller self-similar ones, and define recursive methods by relying on induction (see Section 1.4). Once these are specified it is fairly straightforward to convert the definitions into code, especially when working with basic data types such as integers, real numbers, characters, or Boolean values. Consider the function in (1.5) that adds the first n positive integers (i.e., natural numbers). In Python the related function can be coded as shown in Listing 1.1. The analogy between (1.5) and the Python function is evident. As in many of the examples covered throughout the book, a simple **if** statement is the only control flow structure needed in order to code the function. Additionally, the name of the function appears within its body, implementing a **recursive call**. Thus, we say that the function **calls** or **invokes** itself, and is therefore recursive (there exist recursive functions that do not call themselves directly within their body, as explained in Chapter 9).

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

```

Figure 1.7 Functions that compute the sum of the first n natural numbers in several programming languages.

Listing 1.1 Python code for adding the first n natural numbers.

```

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

```

Coding the function in other programming languages is also straightforward. Figure 1.7 shows equivalent code in several programming languages. Again, the resemblance between the codes and the function definition is apparent. Although the code in the book will be in Python, translating it to other programming languages should be fairly straightforward.

An important detail about the function in (1.5) and the associated codes is that they do not check if $n > 0$. This type of condition on an input parameter, which is specified in the statement of a problem or definition of a function, is known as a **precondition**. Programmers can assume that the preconditions always hold, and therefore do not have to develop code in order to detect or handle them.

Listing 1.2 Alternative Python code for adding the first n natural numbers.

```

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))

```

Listing 1.2 shows the recursive code associated with (1.6). The function uses a cascaded `if` statement in order to differentiate between the two base cases (lines 2–5) and the two recursive cases (lines 6–11), which each make two recursive calls to the defined Python function.

It is also straightforward to code a function that computes the n -th Fibonacci number by relying on the standard definition in (1.2). List-

Listing 1.3 Python code for computing the n -th Fibonacci number.

```

1 def fibonacci(n):
2     if n == 1 or n == 2:
3         return 1
4     else:
5         return fibonacci(n - 1) + fibonacci(n - 2)

```

Listing 1.3 shows the corresponding code, where both base cases are considered in the Boolean expression of the `if` statement.

Listing 1.4 Alternative Python code for computing the n -th Fibonacci number.

```

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

```

Implementing the Fibonacci function defined in (1.7) requires more work. While the base cases are identical, the summation in the recursive case entails using a loop or another function in order to compute and add the values $F(1)$, $F(2)$, \dots , $F(n - 2)$. Listing 1.4 shows a possible solution that uses a `for` loop. The result of the summation can be stored in an auxiliary accumulator variable `aux` that can be initialized to 1 (line 5), taking into account the extra unit term in the recursive case. The `for` loop simply adds the terms $F(i)$, for $i = 1, \dots, n - 2$, to the auxiliary variable (lines 6–7). Finally, the function returns the computed Fibonacci number stored in `aux`.

When data types are more complex there may be more variations among the codes of different programming languages due to low-level details. For instance, when working with a data structure similar to a list it is usually necessary to access its length, since recursive algorithms will use that value in order to define sublists. Figure 1.8 illustrates three combinations of lists (or similar data structures) together with parameters that are necessary in order to use sublists in recursive programs. In (a) the data structure (denoted as `a`) allows us to recover its length

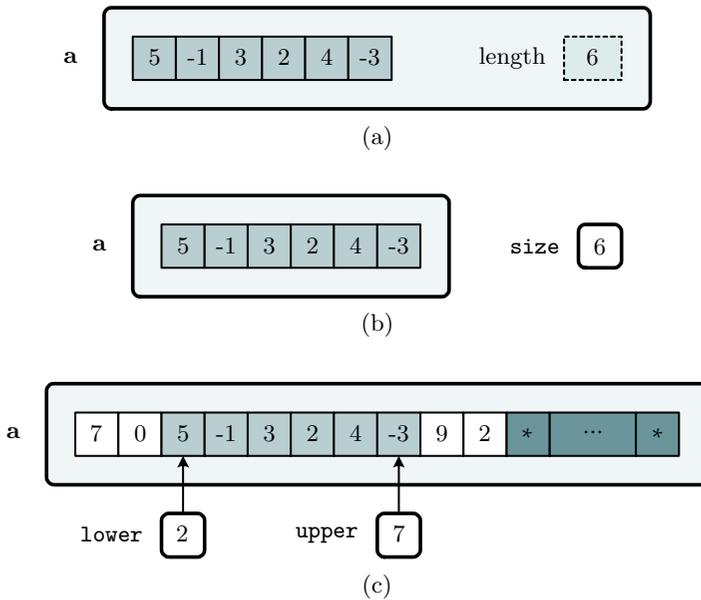


Figure 1.8 Data structures similar to lists, and parameters necessary for defining sublists.

without using any extra variables or parameters. For instance, it could be a property of the list, or it could be recovered through some method. In Python, the length of a list can be obtained through the function `len`. However, it is not possible to access the length of a standard array in other programming languages such as C or Pascal. If the length of the list cannot be recovered directly from the data structure an additional parameter, denoted as `size` (which contains the length of the list) is needed in order to store and access it, as shown in (b). This approach can be used when working with arrays of a fixed size or partially filled arrays. In these cases a more efficient alternative consists of using starting and finishing indices for determining the limits of a sublist, as shown in (c). Note that in this scenario the fixed size of the data structure may be a large constant (enough for the requirements of the application), but the true length of the lists and sublists may be much smaller. The graphic in (c) shows a list where only the first 10 elements are relevant (the rest should therefore be ignored). Furthermore, a sublist of six elements is defined through the `lower` and `upper` index variables, which delimit its boundaries. Note that the elements at these indices are included in the sublist.

Listing 1.5 Recursive functions for adding the elements in a list **a**, where the only input to the recursive function is the list.

```

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))

```

The constructs and syntax of Python allow us to focus on algorithms at a high level, avoiding the need to understand low-level mechanisms such as parameter passing. Nevertheless, its flexibility permits a wide variety of coding possibilities. Listing 1.5 shows three solutions to the problem of adding the elements in a list, corresponding to the three decomposition strategies in Figure 1.6, where the only input parameter is the list,

which is the scenario in [Figure 1.8\(a\)](#). Functions `sum_list_length_1`, `sum_list_length_2`, and `sum_list_length_3` implement the recursive definitions in (1.9), (1.10), and (1.11), respectively. The last lines of the example declare a list `v` and print the sum of its elements using the three functions. Note that the number of elements in the list n is recovered through `len`. Finally, recall that `a[lower, upper]` is the sublist of `a` from index `lower` to `upper-1`, while `a[lower:]` is equivalent to `a[lower:len(a)]`. If the size of the list cannot be obtained from the list directly it can be passed to the functions through an additional parameter, as shown in [Figure 1.8\(b\)](#). The corresponding code is similar to [Listing 1.5](#) and is proposed as an exercise at the end of the chapter.

Alternatively, [Listing 1.6](#) shows the corresponding functions when using large lists and two parameters (`lower` and `upper`) in order to determine sublists within it, as illustrated in [Figure 1.8\(c\)](#). Observe the analogy between these functions and the ones in [Listing 1.5](#). In this case, empty lists occur when `lower` is greater than `upper`. Also, the sublist contains a single element when both indices are equal (recall that both parameters indicate positions of elements that belong to the sublist).

1.4 INDUCTION

Induction is another concept that plays a fundamental role when designing recursive code. The term has different meanings depending on the field and topic where it is used. In the context of recursive programming it is related to mathematical proofs by induction. The key idea is that programmers must assume that the recursive code they are trying to implement already works for simpler and smaller problems, even if they have not yet written a line of code! This notion is also referred to as the recursive “leap of faith.” This section reviews these crucial concepts.

1.4.1 Mathematical proofs by induction

In mathematics, proofs by induction constitute an important tool for showing that some statement is true. The simplest proofs involve formulas that depend on some positive (or nonnegative) integer n . In these cases, the proofs verify that the formulas are indeed correct for every possible value of n . The approach involves two steps:

- a) Base case (basis). Verify that the formula is valid for the smallest value of n , say n_0 .

Listing 1.6 Alternative recursive functions for adding the elements in a sublist of a list **a**. The boundaries of the sublist are specified by two input parameters that mark lower and upper indices in the list.

```

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))

```

- b) Inductive step. Firstly, assume the formula is true for some general value of n . This assumption is referred to as the **induction hypothesis**. Subsequently, by relying on the assumption, show that if the formula holds for some value n , then it will also be true for $n + 1$.

If it is possible to prove both steps, then, by induction, it follows that the statement holds for all $n \geq n_0$. The statement would be true for n_0 , and then for $n_0 + 1$, $n_0 + 2$, and so on, by applying the inductive step repeatedly.

Again, consider the sum of the first n positive numbers (see (1.4)). We will try to show if the following identity (which is the induction hypothesis) involving a quadratic polynomial holds:

$$S(n) = \sum_{i=1}^n i = \frac{n(n+1)}{2}. \quad (1.12)$$

The base case is trivially true, since $S(1) = 1(2)/2 = 1$. For the induction step we need to show whether

$$S(n+1) = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} \quad (1.13)$$

holds, by assuming that (1.12) is true. Firstly, $S(n+1)$ can be expressed as:

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

Furthermore, assuming (1.12) holds we can substitute the summation by the polynomial:

$$S(n+1) \stackrel{\substack{\text{induction} \\ \text{hypothesis}}}{\stackrel{\downarrow}{=}} \frac{n(n+1)}{2} + n+1 = \frac{n^2 + n + 2n + 2}{2} = \frac{(n+1)(n+2)}{2},$$

showing that (1.13) is true, which completes the proof.

1.4.2 Recursive leap of faith

Recursive functions typically call themselves in order to solve smaller subproblems. It is reasonable for beginner programmers to doubt if a recursive function will really work as they code, and to question whether it is legitimate to call the function being written within its body, since it has not even been finished! However, not only can functions call themselves (in programming languages that support recursion), it is crucial to assume that they work correctly for subproblems of smaller size. This assumption, which plays a similar role as the induction hypothesis in proofs by induction, is referred to as the recursive “leap of faith.” It is one of the cornerstones of recursive thinking, but also one of the hardest

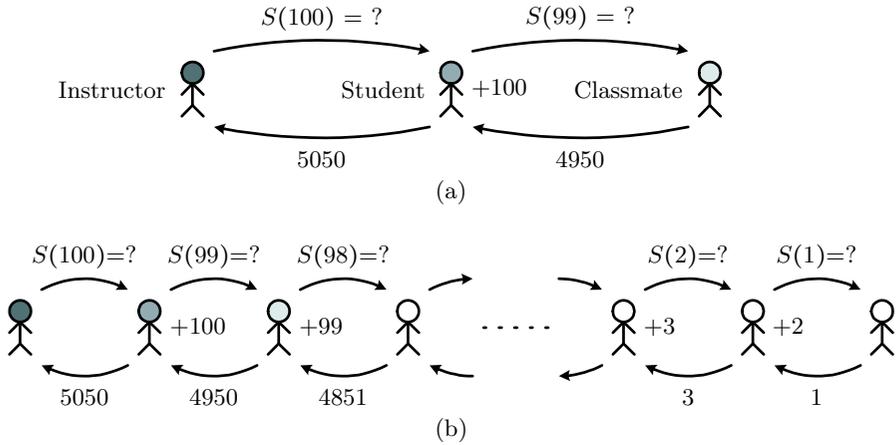


Figure 1.9 Thought experiment in a classroom, where an instructor asks a student to add the first 100 positive integers. $S(n)$ represents the sum of the first n positive integers.

concepts for novice programmers to wrap their heads around. Naturally, it is not a mystical leap of faith that should be accepted without understanding it. The following thought experiment illustrates its role in recursive thinking.

Consider you are in a large classroom and the instructor asks you to add up the integers from 1 to 100. Assuming you are unaware of the formula in (1.12), you would have to carry out the 99 involved additions. However, imagine you are allowed to talk to classmates and there is a “clever” one who claims he can add the first n positive integers, for $n \leq 99$. In that case you could take a much simpler approach. You could simply ask him to tell you the sum of the first 99 positive integers, which is 4950, and all you would need to do is add 100 to your classmate’s response, obtaining 5050. Although it appears that you are cheating or applying some trick, this strategy (illustrated in Figure 1.9(a)) is a valid way to think (recursively) and solve the problem. However, it is necessary to assume that the clever classmate is indeed providing a correct answer to your question. This assumption corresponds to the recursive leap of faith.

There are two possibilities regarding your classmate’s ability to respond to your question. Perhaps he is indeed clever, and can somehow provide the answer without any help (for this problem it is straightforward to apply (1.12)). However, this would be unlikely for complicated

problems. Instead, a second possibility is that he is not as clever as he claims to be, and is also using your strategy (i.e., asking another “clever” classmate) to come up with his answer. In other words, you can interpret that your friend is simply a clone of yourself. In fact, in this last scenario, represented in [Figure 1.9\(b\)](#), all of the students in the class would implement your approach. Note that the process is clearly recursive, where each student asks another one to solve a smaller instance of the problem (decreasing its size by one at each step), until a final student can simply reply 1 in the trivial base case. From there on, each student is able to provide an answer to whoever asked him to compute a full sum of integers by calculating a single addition. This process is carried out sequentially until you receive the sum of the first 99 positive integers, and you will be able to respond correctly to the instructor’s question by adding 100. It is worth mentioning that the approach does not halt when reaching the base case (this is one of students’ major misconceptions). Note that there is a first stage where students successively ask questions to other classmates until one of them responds the trivial 1, and a second phase where they calculate an addition and supply the result to a classmate.

This approach can be formalized precisely by [\(1.5\)](#) and coded as in [Listing 1.1](#), where using $S(n - 1)$ or `sum_first_naturals(n-1)` is equivalent to asking your classmate to solve a subproblem. The recursive leap of faith consists of assuming that $S(n - 1)$ in the definition, or `sum_first_naturals(n-1)` in the code, will indeed return the correct answer.

Similarly to proofs by induction, we can reason that the entire procedure has to be correct. Firstly, your response will be correct as long as your classmate returns the right answer. But this also applies to your classmate, and to the student he is asking to solve a subproblem. This reasoning can be applied repeatedly to every student until finally some student asks another one to add up the integers from 1 to 1. Since the last student returns 1, which is trivially correct, all of the responses from the students must therefore be correct.

All of this implies that programmers can – and *should* – construct valid recursive cases by assuming that recursive function calls, involving self-similar subproblems of smaller size, work correctly. Informally, we must think that we can obtain their solutions “for free.” Therefore, when breaking up a problem as illustrated in [Figure 1.4](#), while we will have to solve the different problems resulting from the decomposition, we will not need to solve the self-similar subproblems, since we can as-

sume that these solutions are already available. Naturally, we will have to process (modify, extend, combine, etc.) them somehow in order to construct the recursive cases. Lastly, recursive algorithms are completed by incorporating base cases that are not only correct, but that also allow the algorithms to terminate.

1.4.3 Imperative vs. declarative programming

Programming paradigms are general strategies for developing software. The imperative programming paradigm focuses on **how** programs work, where the code explicitly describes the control flow and the instructions that modify variable values (i.e., the program state). Iterative code follows this paradigm. In contrast, recursive code relies on the declarative programming paradigm, which focuses on **what** programs should perform without describing the control flow explicitly, and can be considered as the opposite of imperative programming. Functional programming follows this paradigm, which prevents side effects, and where the computation is carried out by evaluating mathematical functions.

Therefore, when designing recursive code, programmers are strongly encouraged to think of **what** functions accomplish, instead of **how** they carry out a task. Note that this is related to functional abstraction and the use of software libraries, where programmers do not need to worry about implementation details, and can focus exclusively on functionality by considering methods as black boxes that work properly. In this regard, programmers can – and *should* – consider calling recursive functions from a high-level point of view as invoking black boxes that return correct results (as in [Figure 1.9\(a\)](#)), instead of thinking about all of the recursive steps taken until reaching a base case (as in [Figure 1.9\(b\)](#)). More often than not, focusing on lower-level details involving the sequence of function calls will be confusing.

1.5 RECURSION VS. ITERATION

The computational power of computers is mainly due to their ability to carry out tasks repeatedly, either through iteration or recursion. The former uses constructs such as **while** or **for** loops to implement repetitions, while recursive functions invoke themselves successively, carrying out tasks repeatedly in each call, until reaching a base case. Iteration and recursion are equivalent in the sense that they can solve the same kinds of problems. Every iterative program can be converted into a re-

cursive one, and vice versa. Choosing which one to use may depend on several factors, such as the computational problem to solve, efficiency, the language, or the programming paradigm. For instance, iteration is preferred in imperative languages, while recursion is used extensively in functional programming, which follows the declarative paradigm.

The examples shown so far in the book can be coded easily through loops. Thus, the benefit of using recursion may not be clear yet. In practice, the main advantage of using recursive algorithms over iterative ones is that for many computational problems they are much simpler to design and comprehend. A recursive algorithm could resemble more closely the logical approach we would take to solve a problem. Thus, it would be more intuitive, elegant, concise, and easier to understand.

In addition, recursive algorithms use the program stack implicitly to store information, where the operations carried out on it (e.g., push and pop) are transparent to the programmer. Therefore, they constitute clear alternatives to iterative algorithms where it is the programmer's responsibility to explicitly manage a stack (or similar) data structure. For instance, when the structure of the problem or the data resembles a tree, recursive algorithms may be easier to code and comprehend than iterative versions, since the latter may need to implement breadth- or depth-first searches, which use queues and stacks, respectively.

In contrast, recursive algorithms are generally not as efficient as iterative versions, and use more memory. These drawbacks are related to the use of the program stack. In general, every call to a function, whether it is recursive or not, allocates memory on the program stack and stores information on it, which entails a higher computational overhead. Thus, a recursive program cannot only be slower than an iterative version, a large number of calls could cause a stack overflow runtime error. Furthermore, some recursive definitions may be considerably slow. For instance, the Fibonacci codes in [Listings 1.3](#) and [1.4](#) run in exponential time, while Fibonacci numbers can be computed in (much faster) logarithmic time. Lastly, recursive algorithms are harder to debug (i.e., analyze a program step by step in order to detect errors and fix them), especially if the functions invoke themselves several times, as in [Listings 1.3](#) and [1.4](#).

Finally, while in some functional programming languages loops are not allowed, many other languages support both iteration and recursion. Thus, it is possible to combine both programming styles in order to build algorithms that are not only powerful, but also clear to understand (e.g., backtracking). [Listing 1.4](#) shows a simple example of a recursive function that contains a loop.

1.6 TYPES OF RECURSION

Recursive algorithms can be categorized according to several criteria. This last section briefly describes the types of recursive functions and procedures that we will use and analyze throughout the book. Each type will be illustrated through recursive functions that, when invoked with certain specific arguments, can be used for computing Fibonacci numbers ($F(n)$). Lastly, recursive algorithms may belong to several categories.

1.6.1 Linear recursion

Linear recursion occurs when methods call themselves only once. There are two types of linear-recursive methods, but we will use the term “linear recursion” when referring to methods that process the result of the recursive call somehow before producing or returning its own output. For example, the factorial function in (1.3) belongs to this category since it only carries out a single recursive call, and the output of the subproblem is multiplied by n in order to generate the function’s result. The functions in (1.5), (1.9), and (1.10) are also clear examples of linear recursion. The following function provides another example:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2, \\ \left\lfloor \Phi \cdot f(n-1) + \frac{1}{2} \right\rfloor & \text{if } n > 2, \end{cases} \quad (1.14)$$

where $\Phi = (1 + \sqrt{5})/2 \approx 1.618$ is a constant also known as the “golden ratio,” and where $\lfloor \cdot \rfloor$ denotes the floor function. In this case, $F(n) = f(n)$ is the n -th Fibonacci number. Chapter 4 covers this type of linear-recursive algorithms in depth.

1.6.2 Tail recursion

A second type of linear recursion is called “tail recursion.” Methods that fall into this category also call themselves once, but the recursive call is the last operation carried out in the recursive case. Therefore, they do not manipulate the result of the recursive call. For example, consider the following tail-recursive function:

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

Observe that in the recursive case the method simply returns the result of a recursive call, which does not appear within a mathematical

or logical expression. Therefore, the recursive case is simply specifying relationships between sets of arguments for which the function returns the same value. As the algorithm carries out recursive calls it modifies the arguments until it is possible to compute the solution easily in a base case. In this example Fibonacci numbers can be recovered through $F(n) = f(n, 1, 1)$. This type of recursive algorithm will be covered in [Chapter 5](#).

1.6.3 Multiple recursion

Multiple recursion occurs when a method calls itself several times in some recursive case (see [Chapter 7](#)). If the method invokes itself twice it is also called “binary recursion.” Examples seen so far include the functions in (1.2), (1.6), (1.7), and (1.11). The following function uses multiple recursion in order to provide an alternative definition of Fibonacci numbers ($F(n) = f(n)$):

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2, \\ [f(\frac{n}{2} + 1)]^2 - [f(\frac{n}{2} - 1)]^2 & \text{if } n > 2 \text{ and } n \text{ even,} \\ [f(\frac{n+1}{2})]^2 + [f(\frac{n-1}{2})]^2 & \text{if } n > 2 \text{ and } n \text{ odd.} \end{cases} \quad (1.16)$$

This type of recursion appears in algorithms based on the “divide and conquer” design strategy, covered in [Chapter 6](#).

1.6.4 Mutual recursion

A set of methods are said to be mutually recursive when they can call each other in a cyclical order. For example, a function f may call a second function g , which in turn can call a third function h , which can end up calling the initial function f . This type of recursion is also called “indirect” since a method might not invoke itself directly within its body, but through a cyclical chain of calls. For example, consider the two following functions:

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

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

It is clear that A is recursive since it calls itself, but the recursion in B is achieved indirectly by calling A , which in turn calls B . Thus, recursion arises since invoking B can produce other calls to B (on simpler instances of the problem). These functions can also be used to generate Fibonacci numbers. In particular, $F(n) = B(n) + A(n)$. Mutual recursion will be covered in [Chapter 9](#).

1.6.5 Nested recursion

Nested recursion occurs when an argument of a recursive function is defined through another recursive call. Consider the following function:

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

The second parameter of the recursive call is an expression that involves yet another recursive function call. In this case, Fibonacci numbers can be recovered through $F(n) = f(n, 0)$. This type of recursion is rare, but appears in some problems and contexts related to tail recursion. Finally, an overview of nested recursion is covered in [Chapter 11](#).

1.7 EXERCISES

Exercise 1.1 — What does the following function calculate?

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

Exercise 1.2 — Consider a sequence defined by the following recursive definition: $s_n = s_{n-1} + 3$. Calculate its first four terms, by considering that: (a) $s_0 = 0$, and (b) $s_0 = 4$. Provide a nonrecursive definition of s_n in both cases.

Exercise 1.3 — Consider a sequence defined by the following recursive definition: $s_n = s_{n-1} + s_{n-2}$. If we are given the initial values of s_1 and s_2 we have enough information to build the entire sequence. Show that it is also possible to construct the sequence when given two arbitrary values s_i and s_j , where $i < j$. Finally, find the elements of the sequence between $s_1 = 1$ and $s_5 = 17$.

Exercise 1.4 — The set “descendants of a person” can be defined recursively as the children of that person, together with the descendants

of those children. Provide a mathematical description of the concept using set notation. In particular, define a function $D(p)$, where D denotes descendants, and the argument p refers to a particular person. Also, consider that you can use the function $C(p)$, which returns the set containing the children of a person p .

Exercise 1.5 — Let $F(n) = F(n - 1) + F(n - 2)$ represent a recursive function, where n is a positive integer, and with arbitrary initial values for $F(1)$ and $F(2)$. Show that it can be written as $F(n) = F(2) + \sum_{i=1}^{n-2} F(i)$, for $n \geq 2$.

Exercise 1.6 — Implement the factorial function, defined in (1.3).

Exercise 1.7 — When viewed as an abstract data type, a list can be either empty, or it can consist of a single element, denoted as the “head” of the list, plus another list (which may be empty), referred to as the “tail” of the list. Let `a` represent a list in Python. There are several ways to check whether it is empty. For example, the condition `a==[]` returns `True` if the list is empty, and `False` otherwise. In addition, the head of the list is simply `a[0]`, while the tail can be specified as `a[1:]`. Write an alternative version of the function `sum_list_length_2` in Listing 1.5 that avoids using `len` by using the previous elements.

Exercise 1.8 — Implement three functions that calculate the sum of elements of a list of numbers using the three decompositions in Figure 1.6. The functions will receive two input parameters: a list, and its size (i.e., length), according to the scenario in Figure 1.8(b). In addition, indicate an example calling the functions and printing their results.

Exercise 1.9 — Show by mathematical induction the following identity related to a geometric series, where n is a positive integer, and $x \neq 1$ is some real number:

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}.$$

Exercise 1.10 — Code the five recursive functions defined in Section 1.6 that compute Fibonacci numbers $F(n)$. Since some require several parameters besides n , or do not compute a Fibonacci number directly, implement additional “wrapper” functions that only receive the parameter n , and call the coded functions in order to produce Fibonacci numbers. Test that they produce correct outputs for $n = 1, \dots, 10$.

Methodology for Recursive Thinking

Science is much more than a body of knowledge. It is a way of thinking.

— Carl Sagan

WHEN thinking declaratively it is possible to design recursive algorithms for solving a wide variety of computational problems by following a systematic approach. This chapter describes a general template for deriving recursive solutions from a declarative perspective that unravels the process of recursive thinking into a sequence of steps. In addition, the chapter introduces several useful diagrams that programmers can utilize when designing recursive cases. These are beneficial since they force us to think about problem decomposition and induction, the two most important concepts underlying recursive (declarative) thinking.

Finally, for some tasks (e.g., debugging) or certain complex problems it may be useful to think about the lower-level details regarding the sequence of operations carried out by recursive procedures. We will nevertheless postpone covering these lower-level aspects until [Chapter 10](#). Instead, this chapter focuses exclusively on higher-level declarative thinking.

2.1 TEMPLATE FOR DESIGNING RECURSIVE ALGORITHMS

The recursive functions introduced in [Chapter 1](#), as well as many of the algorithms that will appear in the rest of the book, can be designed by following the methodology described in [Figure 2.1](#). This template em-

1. Determine the **size** of the problem.
2. Define **bases cases**.
3. **Decompose** the computational problem into self-similar subproblems of smaller size, and possibly additional different problems.
4. Define **recursive cases** by relying on **induction** and **diagrams**.
5. **Test** the code.

Figure 2.1 General template for designing recursive algorithms.

phasizes declarative thinking due to its fourth step, since using induction implies focusing on what an algorithm performs, rather than on how it solves the problem. The following sections explain each step and mention common pitfalls and misunderstandings.

2.2 SIZE OF THE PROBLEM

Naturally, the first step when tackling any computational problem is to understand it. Given the problem statement, the inputs, outputs, and relationships between them should be clearly identified in order to proceed. Regardless of the type of algorithm we intend to design, it is crucial to determine the size of the problem (also denoted as input size). It can be thought of as a mathematical expression involving quantities related to the input parameters that determine the problem's complexity in terms of the number of operations that algorithms need in order to solve it. When using recursion it is especially relevant since the base and recursive cases clearly depend on it.

In many problems the size only depends on one factor, which also happens to be one of the input parameters. For instance, the size of calculating the n -th Fibonacci number, the factorial of n , or the sum of the first n positive integers (see [Chapter 1](#)) is precisely the input parameter n . For these problems it is apparent that solving smaller instances (e.g., for $n - 1$ or $n/2$) requires less operations. In other cases the size of the

problem might not be specified explicitly by the inputs, but the programming language will allow us to retrieve the particular size through its syntax and constructs. For example, when working with lists, their length often determines the size of the problem, which can be recovered through the function `len`.

In other problems the size may be expressed as a function of the input parameters. For instance, consider the problem of computing the sum of the digits of some positive integer n . Although the input parameter is n , the size of the problem is not n , but rather the number of digits of n , which determines the number of operations to carry out. Formally this quantity can be expressed as $\lfloor \log_{10}(n) \rfloor + 1$.

The size of a problem may also depend on several input parameters. Consider the problem of adding the elements of a sublist within a list (see [Listing 1.6](#)), determined by “lower” and “upper” indices. Solving the problem requires adding $n - m + 1$ elements, where n and m are the upper and lower indices, respectively. Thus, it needs to compute $n - m$ additions. The small (unit) difference between both expressions is irrelevant. It is worth mentioning that it is not necessary to know exactly how many operations will be carried out in order to solve a problem. Instead, it is enough to understand when a base case is reached, and how to reduce the size of the problem in order to decompose it into smaller subproblems. For this last problem, the size is decreased by reducing the difference between m and n .

Moreover, the size of a problem does not necessarily need to specify the number of operations that algorithms must perform. Consider the problem of adding the elements of a square $n \times n$ -dimensional matrix. Since it contains n^2 elements, the sum requires $n^2 - 1$ additions, which is a function of n . However, in this case the size of the problem is simply n , and not n^2 . Note that smaller subproblems arise by decreasing n , not n^2 . If the matrix were $n \times m$ -dimensional, the problem size would depend on both parameters n and m . In particular, the size of the problem would be nm , where we could obtain subproblems of smaller size by decreasing n , m , or both.

The size is generally a property of a problem, not of a particular algorithm that solves it. Thus, it is not the actual number of computations carried out by a specific algorithm, since problems can be solved by different algorithms whose runtime may vary considerably. Consider the searching problem that consists of determining whether or not some number appears in a sorted list of n numbers. In the worst case (when the list does not contain the number and the result is `False`) it can be

solved naively in n steps, or more efficiently in $\lfloor \log_2(n) \rfloor + 1$ operations by the “binary search” algorithm. Regardless of the algorithm used, the function that describes the runtime only depends on n . Thus, the size of the problem is n .

Nevertheless, for some problems the size can be defined in several ways. Moreover, the choice of the size affects the rest of the decisions in the recursive design process, and ultimately leads to different algorithms for solving the problem. For example, consider the problem of adding two nonnegative integers a and b by only using unit increments or decrements (see [Section 4.1.2](#)). The size of the problem can be $a+b$, a , b , or $\min(a, b)$, and the final algorithm will be different depending on which problem size we choose to work with.

2.3 BASE CASES

Base cases are instances of problems that can be solved without using recursion; or more precisely, without carrying out recursive calls. The most common type of base case is associated with the smallest instances of a problem, for which the result can be determined trivially, and sometimes may not even require carrying out computations. For instance, the value of the first and second Fibonacci numbers is simply 1, which follows directly from the definition. The “sum” of the first positive integer is evidently 1. Similarly, the “sum” of the digits of a nonnegative number n that contains only one digit (i.e., $n < 10$) is obviously that digit (n). Observe that these functions simply return a value in the base cases, and do not perform any addition or other mathematical operation.

Some methods may require several base cases. For example, [\(1.2\)](#) defines a base case for $n = 1$ and another for $n = 2$. Since the associated value (1) is the same, the Python function in [Listing 1.3](#) can use a single logical expression (`(n==1) or (n==2)`) in order to take into account both cases. Other methods need to use a cascaded `if` structure in order to separate each base case. For instance, the function in [\(1.6\)](#) defines a base case for $n = 1$ and another for $n = 2$ since it returns different values for both inputs. Logically, the associated code in [Listing 1.2](#) separates both cases.

One of the advantages of recursive programming is that the resulting code is often concise, which helps in understanding it. In this regard, consider the factorial function defined in [\(1.3\)](#). One could argue that if base cases correspond to trivial instances of a problem, then the definition of the function could include the case $1! = 1$ as well. However, $1!$ can be

Listing 2.1 Misconceptions regarding base cases through the factorial function.

```

1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return factorial(n - 1) * n
6
7
8 def factorial_redundant(n):
9     if n == 0 or n == 1:
10        return 1
11    else:
12        return factorial_redundant(n - 1) * n
13
14
15 def factorial_missing_base_case(n):
16     if n == 1:
17         return 1
18     else:
19         return factorial_missing_base_case(n - 1) * n
20
21
22 def factorial_no_base_case(n):
23     return factorial_no_base_case(n - 1) * n

```

obtained through the recursive case correctly since $1! = 0! \times 1 = 1 \times 1 = 1$. Therefore, although $1!$ is a valid base case (it is obviously correct), including it would be redundant, and therefore unnecessary. Moreover, these dispensable cases could be misleading, since programmers trying to understand the recursive code could interpret that the functions require them in order to be correct. Thus, for clarity and conciseness, it is often recommended to include the least number of base cases that lead to correct recursive definitions. Lastly, redundant base cases usually do not have a relevant impact on the efficiency of a recursive program. In general, although they can reduce the number of recursive function calls for a given input, the saved computational time is often negligible. For instance, if $1!$ is included in the factorial function as a base case the recursive process will only perform one recursive call less, which barely has any effect on computing time (moreover, the code would need two conditions, and could even be more inefficient if both are evaluated before reaching a recursive case).

[Listing 2.1](#) uses the factorial function to illustrate this and other basic pitfalls regarding bases cases. Firstly, the `factorial` method provides a perfect implementation of the mathematical function. The `factorial_redundant` method is correct since it produces a valid output for any nonnegative integer input argument. However, it contains an extra base case that is unnecessary.

In addition, we should also strive for generality, which is an important desirable software feature. More general functions are those that can operate correctly on a wider range of inputs, and are therefore more applicable and useful. The factorial function in [\(1.3\)](#) receives a nonnegative integer as its input argument, and is defined adequately since it can be applied correctly not only to every positive integer, but also to 0. Replacing the base case $0! = 1$ by $1! = 1$ would lead to a less general function, since it would not be defined for $n = 0$. The method `factorial_missing_base_case` implements this function. If it were called with `n = 0` as its input argument it would fail to produce a valid result since it would run into an infinite recursion. In particular, `factorial_missing_base_case(0)` would call `factorial_missing_base_case(-1)`, which would call `factorial_missing_base_case(-2)`, and so on. In practice the process `textcolorred` would halt producing a runtime error message (e.g., “stack overflow,” or “maximum recursion depth exceeded”) after performing a large number of function calls (see [Chapter 10](#)). Finally, `factorial_no_base_case` always generates an infinite recursion since it does not contain any base case, and can therefore never halt.

For many problems, there are instances for which we can provide a result without using recursion, even if their size is not small. For example, consider the problem of determining whether some element appears in a list. The size of the problem is the number of elements in the list. A first base case occurs when the list is empty (this is the smallest instance of the problem, of zero size), where the result is obviously `False`. In addition, the algorithm can check if the element appears in some position (e.g., the first, middle, or last) of the list, and return a true value immediately if it does, even if the list is very large ([Chapter 5](#) covers this type of searching problems). In both cases there is no need to carry out a recursive call.

2.4 PROBLEM DECOMPOSITION

The next step in the recursive design methodology consists of identifying self-similar problems smaller than the original (and possibly other different problems), as illustrated in [Figure 1.4](#). These smaller subproblems will be used in the fourth step of the template in order to establish the recursive cases. The process involves decreasing the size of the problem in order to consider smaller instances that are closer to the base cases. Therefore, it is essential to correctly establish the size of the problem and its base cases before proceeding with this step.

Many recursive solutions reduce the size of a problem by decreasing it a single unit, or by dividing it by two, as shown graphically in [Figures 1.5](#) and [1.6](#). Representing problems through diagrams is useful and highly recommended since it may help us to recognize self-similar subproblems (i.e., recursion) visually, and can also facilitate designing recursive cases. [Figure 2.2](#) shows additional diagrams that illustrate the decomposition of the sum of the first positive integers when the problem size is decreased by a unit. Firstly, since the recursive method is a mathematical function that can be written as a formula, we can simply expand the expression in order to identify the original problem and the corresponding subproblem, as shown in (a). In (b) the diagram also contains a formula, but uses boxes that enclose the formulas instead of using curly braces. In this case, the darker outer box represents a full problem, while the lighter box contained in it symbolizes a subproblem. In (c) the diagram uses a similar scheme, where a problem consists of adding the small circles arranged in a triangular pattern. This illustration is analogous to that of [Figure 1.5\(b\)](#). Finally, in (d) the numbers to be added are depicted inside rectangles, which are concatenated in order to represent the problem and subproblem.

These apparently simple diagrams are appropriate since they allow us to identify the subproblem within the original one. However, it is crucial to observe that the subproblem is not decomposed into yet smaller ones. For instance, the sum of the positive integers up to $n - 2$ does not appear explicitly as a subproblem in any of the diagrams. In other words, the diagrams avoid illustrating subproblems in a nested manner, or in a tree structure. This is one of the main pitfalls and sources of misunderstandings when thinking recursively. [Figures 2.3\(a\)](#) and (c) illustrate a proper decomposition of the sum of the first positive integers ($S(n)$). Instead, while the nested diagram in (b), or the tree representing the sequence of recursive calls in (d), represent the problem correctly, they

$$\overbrace{1 + 2 + 3 + \dots + (n - 2) + (n - 1)}^{\text{Subproblem}} + n$$

Original problem

(a)

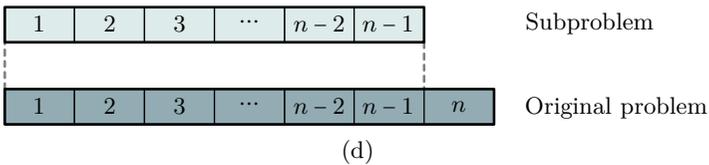
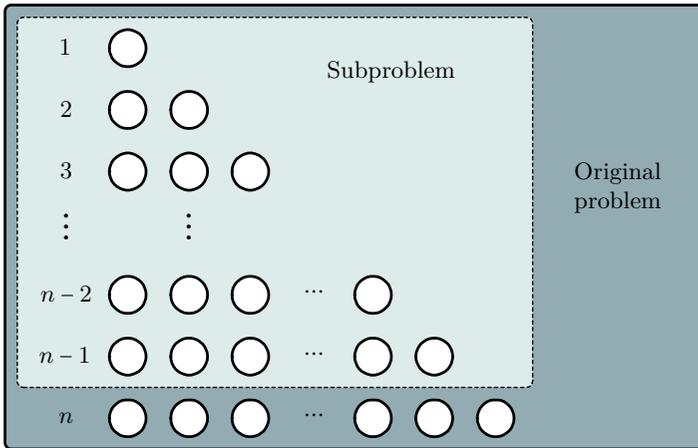
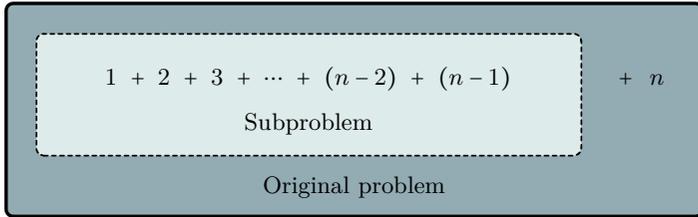


Figure 2.2 Additional diagrams that illustrate the decomposition of the sum of the first positive integers when the problem size is decreased by a unit.

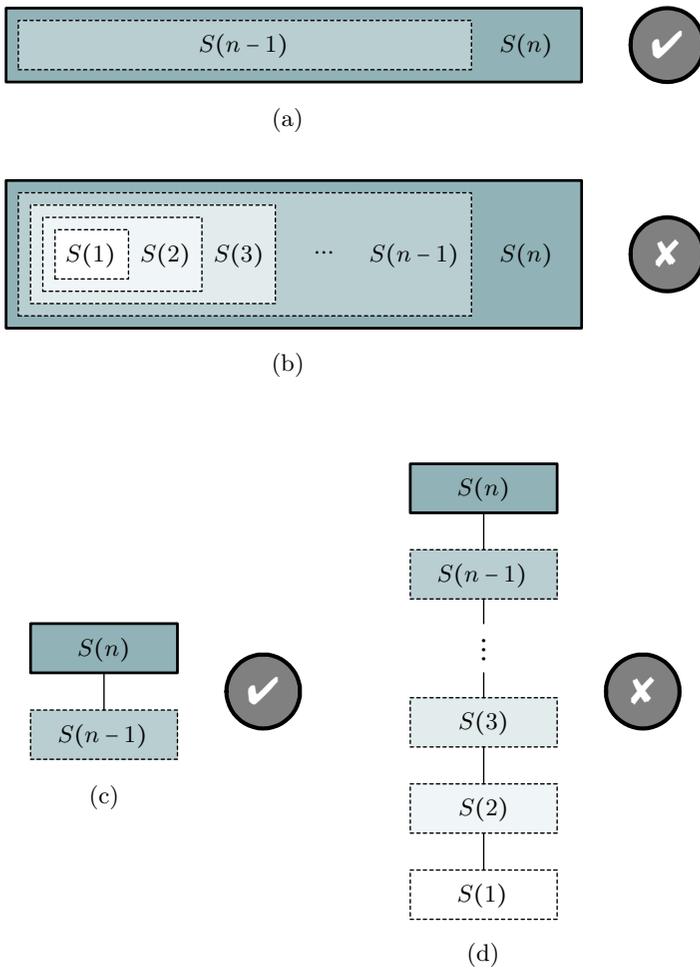


Figure 2.3 When thinking recursively we generally do not need to decompose a problem into every instance of smaller size.

incorporate additional subproblems ($S(1), S(2), \dots, S(n-2)$) that do not appear in the recursive definition in (1.5). Therefore, these subproblems should be omitted from the diagrams, and therefore also from the recursive thinking process, in order to avoid confusion. The nested and tree-structured diagrams will nevertheless be useful for other tasks (see Chapter 10).

Avoiding nested diagrams does not mean that we should only consider one subproblem. Many recursive definitions require decomposing

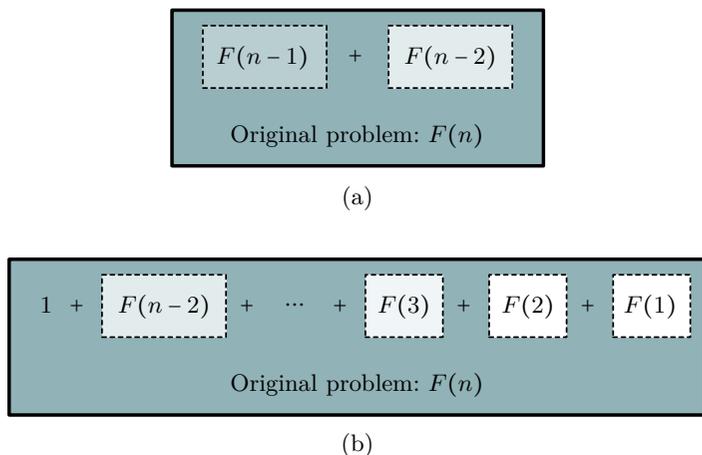


Figure 2.4 Decompositions of the Fibonacci function.

the original problem into numerous subproblems. For instance, [Figure 2.4](#) shows two diagrams for decomposing a Fibonacci number. In particular, (a) is associated with the definition in (1.2), while (b) is related to (1.7). Although the diagram in (b) shows every subproblem from $F(1)$ until $F(n-2)$, it is a valid representation since the boxes that represent subproblems are not nested.

For most problems in this book the size will depend on only one parameter or factor. However, in some cases it is determined by several. For instance, problems on matrices usually depend on their height (number of rows) and width (number of columns). Problems of smaller size can therefore arise from decreasing one or both of these parameters. In particular, many recursive algorithms partition matrices in blocks that correspond to submatrices. These “block matrices” can be interpreted as the original matrix with vertical and horizontal subdivisions, as illustrated in the following example with a 4×7 matrix:

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

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

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

In this case, the number of rows and columns of each submatrix results from performing the integer division of the original height and width of the matrix by two.

Regarding efficiency, dividing the size of the problem by two, instead of decreasing it by a single unit, may lead to faster algorithms. Therefore, this strategy should be considered in general, and especially if a theoretical analysis shows that it is possible to improve the computational cost of an algorithm based on reducing the size of the problem by a unit. However, dividing the size of a problem by two may not lead to a reasonable recursive algorithm. For instance, it is difficult to obtain a simple recursive formula for the factorial function $n!$ by considering the subproblem $(n/2)!$.

2.5 RECURSIVE CASES, INDUCTION, AND DIAGRAMS

The next step in the template for designing recursive algorithms consists of defining the recursive cases, which involves figuring out how to build the full solution to an original problem by using the solutions to the self-similar subproblems provided by the decomposition stage. As mentioned in [Section 1.4](#), we can assume that these simpler solutions are readily available by relying on induction. Thus, the real challenge when thinking recursively consists of determining how to modify, extend, or combine the solutions to subproblems in order to obtain the complete solution to an original problem, which corresponds to the last step in the scheme presented in [Figure 1.4](#).

2.5.1 Thinking recursively through diagrams

It is often useful to represent the thought process or “mental model” of deriving recursive cases by means of diagrams. The general illustration in [Figure 2.5](#) shows a recursive thinking procedure when a problem is decomposed into a single self-similar subproblem. The process starts by considering a general instance of the problem, of a certain size, determined by the input parameters, which are depicted in the top-left box. Applying the recursive method to those parameters would produce the results represented in the top-right box. Note that this top row is simply another way to specify the statement of the problem. In order to define recursive cases we first choose a particular decomposition that leads to a smaller instance of the problem. The new, simpler, parame-

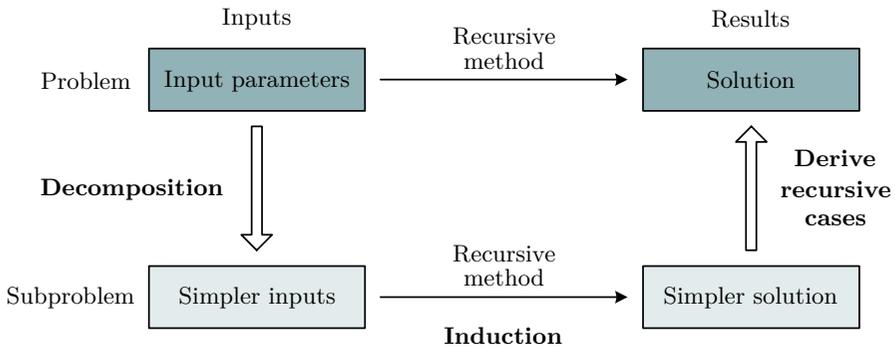
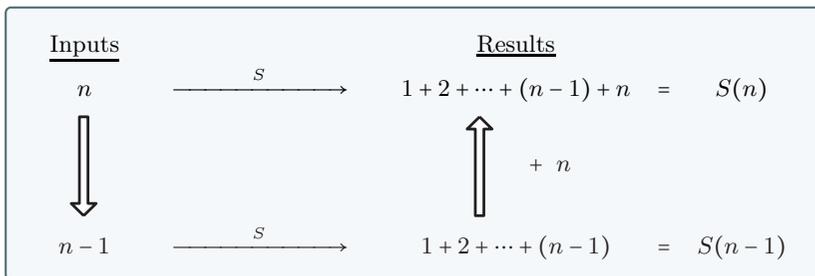


Figure 2.5 General diagram for thinking about recursive cases (when a problem is decomposed into a single self-similar subproblem).

ters for the self-similar subproblem are shown in the bottom-left box. A recursive call to the method with those parameters would therefore produce the results in the bottom-right box, which are obtained by applying the statement of the problem to the simpler inputs. Finally, since we can assume that these results are available by relying on induction, we derive recursive cases by determining how we can modify or extend them in order to obtain or arrive at the solution to the original problem (top-right box).

For the sum of the first n positive integers ($S(n)$) the diagram would be:



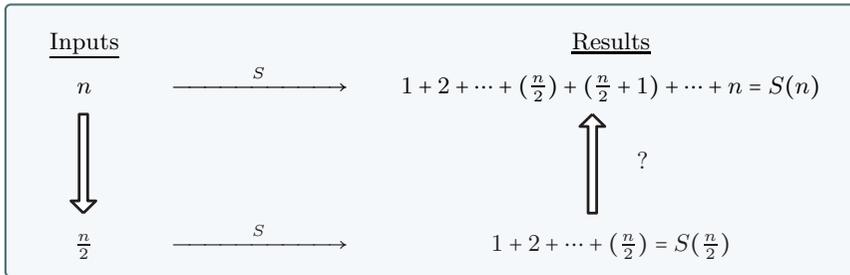
Firstly, we can choose to decompose the problem by reducing its size by a unit. For this particular decomposition our goal is to figure out how we can obtain $S(n)$ by modifying or extending the solution to the subproblem $S(n - 1)$. In this case, it is easy to see that adding n to $S(n - 1)$ yields $S(n)$. Therefore, the recursive case is defined as $S(n) = S(n - 1) + n$.

It is important to understand that we are only required to define the decomposition, and establish how to obtain the results to the original problem given the results of the subproblem. Therefore, we only have to think about the processes associated with the thicker arrows. Alternatively, the thinner arrows simply symbolize the effect of solving particular instances of the problem (on different inputs), which is completely defined by the statement of the problem. Observe that we do not have to make any algorithmic design decisions regarding the relationships between inputs and results, since they are specified in the statement of the problem.

The advantage of the diagram in [Figure 2.5](#) is its generality, since it can be applied to numerous problems for deriving recursive solutions. Therefore, the explanations to many of the examples covered throughout the book will rely on it. Naturally, other graphical representations, such as the diagrams in [Figures 1.5](#), [1.6](#), and [2.2](#) can also be used instead. These alternative visualizations are especially useful since they not only show subproblems, but also their relationship with the original problem, which provides clues about how we can combine, increment, or modify the solutions to the subproblems in order to solve the original one. In other words, they visually suggest how we can define the recursive cases. For instance, it is apparent from the diagrams in [Figure 2.2](#) that the solution to the problem consists of adding n to the result of the subproblem of size $n - 1$.

Indeed, more elaborate diagrams, especially tailored for the particular problem, may help us to find different decompositions that could eventually lead to more efficient algorithms. For instance, representing the sum of the first n positive integers through triangular pyramids of tiled blocks, as shown in [Figure 1.5](#), allows us to derive recursive cases easily when the size of the subproblems is roughly half of the original one. Firstly, recall that the problem can be interpreted as counting the number of blocks in a triangular structure. Furthermore, since the representation allows us to superimpose several subproblems (i.e., smaller triangular structures) simultaneously on top of the original, the visualizations indicate how it is possible to “fill” a larger pyramid with smaller ones. Therefore, the diagrams illustrate how to derive recursive cases by adding the results (in this case, the number of blocks) of smaller subproblems. For example, in [Figure 1.5\(c\)](#) the output of three subproblems of size $n/2$ can be added to the result of the subproblem of size $n/2 - 1$ in order to obtain the total value of the original problem.

Nonetheless, the general diagram in Figure 2.5 can also be used in order to provide recursive solutions for $S(n)$ when the decomposition divides the size of the problem by two. In that case we have:



The question is to figure out how (and if) we can obtain $S(n)$ by modifying or extending $S(n/2)$. A first obvious idea leads to $S(n) = S(n/2) + (n/2 + 1) + \dots + n$. However, although it is correct, its implementation requires either a loop, or using a different recursive function, in order to calculate the sum of the last $n/2$ terms. Observe that it is not possible to obtain $(n/2 + 1) + \dots + n$ by using a recursive call to the function under construction (S), since that sum is not an instance of the problem. Nevertheless, it can be broken up, for example, in the following way:

$$\begin{array}{r}
 \frac{(n/2 + 1) + (n/2 + 2) + \dots + n}{n/2 + n/2 + \dots + n/2} = (n/2)^2 \\
 + \quad 1 + 2 + \dots + n/2 = S(n/2)
 \end{array}$$

This not only simplifies the expression, but it also contains $S(n/2)$, which we can use in order to obtain a much simpler recursive case:

$$\begin{aligned}
 S(n) &= S(n/2) + (n/2 + 1) + \dots + n \\
 &= S(n/2) + S(n/2) + (n/2)^2 \\
 &= 2S(n/2) + (n/2)^2.
 \end{aligned} \tag{2.1}$$

The general diagram could be completed in the following way:

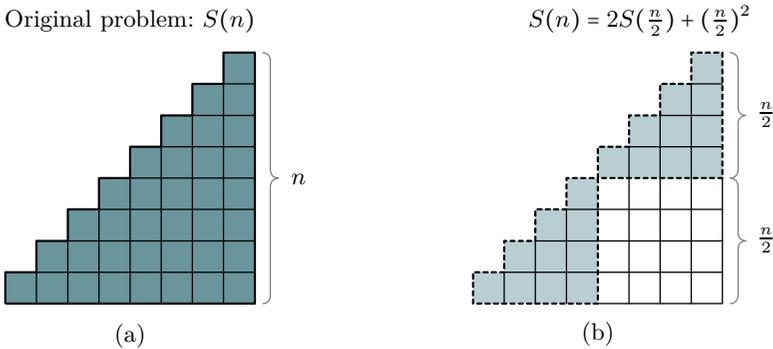
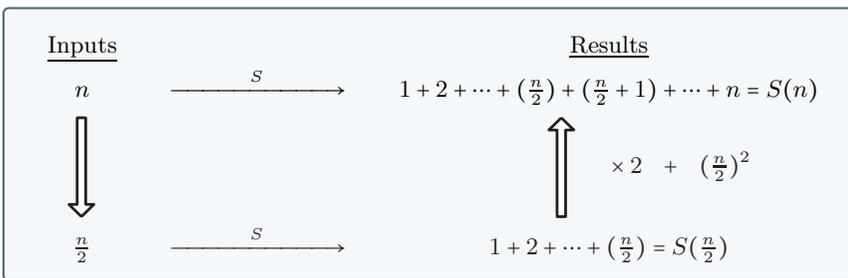


Figure 2.6 Diagram showing a decomposition of the sum of the first n positive integers $S(n)$ that uses two subproblems of half the size as the original.



indicating that $S(n)$ can be obtained by multiplying the result of $S(n/2)$ times 2, and adding $(n/2)^2$. Although the decomposition of the problem is unique ($S(n/2)$), this example shows that we can use the result of the subproblem several times in order to arrive at the solution of the original problem. In this case, we can interpret that the recursive case uses $S(n/2)$ *twice*.

This recursive case can also be obtained through the diagrams that depict $S(n)$ as a triangular pyramid of blocks. In particular, [Figure 2.6](#) illustrates the same reasoning that leads to $S(n) = 2S(n/2) + (n/2)^2$ through these representations. The nature of this particular diagram, where the solution to the problem consists of adding up blocks, facilitates obtaining recursive cases.

2.5.2 Concrete instances

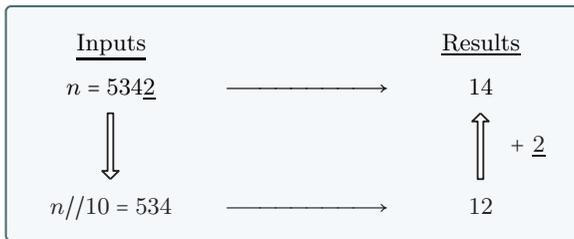
In the previous example involving the sum of the first positive integers it is straightforward to express the result of the recursive method as a

general function of the input parameter n . In other words, it is easy to obtain the expressions on the right-hand side of the general diagram. However, in other problems it can be more complicated to describe the results of the recursive methods (e.g., through some formula), and therefore to figure out how to arrive at the solution given the results to the subproblems. In these cases a useful initial approach consists of analyzing the problem through concrete instances.

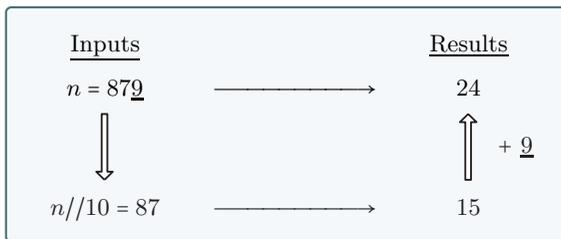
Consider the problem of adding the digits of some nonnegative integer n , and assume that we choose the decomposition that decreases the number of digits by a unit, where the least significant digit is dropped from the original integer. This information provides the elements on the left-hand side of the general diagram. However, the method's output on the right-hand side can be complicated as a function of n . For example, one possible way to describe it is through the following summation:

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

Although we could use the formula to derive the recursive cases, we will proceed by analyzing concrete instances of the problem. For example (we can discard using the method's name in the results column, and when labeling the arrows, for the sake of simplicity):



or



It is easy to see in these diagrams that we can obtain the result of the method by adding the last (underlined) digit of the original number to the output of the subproblem.

Listing 2.2 Code for computing the sum of the digits of a nonnegative integer.

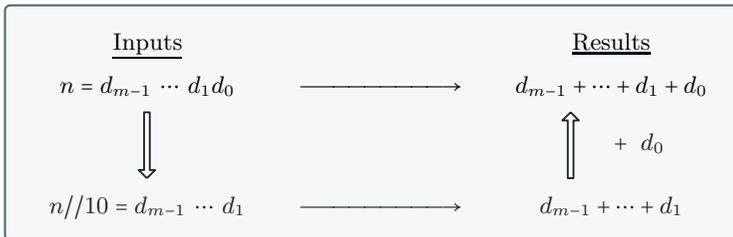
```

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

```

2.5.3 Alternative notations

Another approach consists of using some alternative general notation that facilitates constructing a diagram. Let $d_{m-1} \cdots d_1 d_0$ represent the sequence of digits that define some nonnegative integer n in base 10 (in other words, $n = d_{m-1} \cdot 10^{m-1} + \cdots + d_1 \cdot 10 + d_0$, with $0 \leq d_i \leq 9$ for all i , and $d_{m-1} \neq 0$ if $m > 1$). In that case the general diagram would be:



and the function can be coded as shown in [Listing 2.2](#), where $(n \% 10)$ represents d_0 . Note that the base case for this problem occurs when n contains a single digit (i.e., $n < 10$), for which the result is obviously n .

Lastly, in practice more complicated problems may require analyzing multiple examples and scenarios (e.g., distinguishing when the input parameter is odd or even), that lead to different recursive cases.

2.5.4 Procedures

The methods seen so far correspond to functions that return values, where the results can be defined through formulas. Therefore, they can be used within expressions in other methods, where they would return specific values given a set of input arguments. However, there exist methods, called “procedures” in certain programming languages (e.g., Pascal), which do not return values. Instead, they may alter data structures

passed as inputs to the method, or they can simply print information on a console. In these cases it is also possible to use the general diagram.

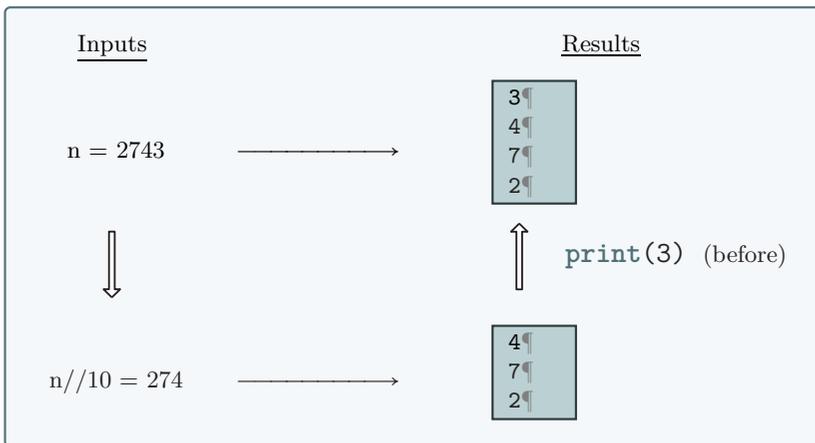
Consider the problem of printing the digits of some nonnegative integer n on the console, in reversed order, and vertically. In other words, the least significant digit will appear on the first line, the second least significant digit on the second line, and so on. For instance, if $n = 2743$ the program will print the following lines on the console:

```

3
4
7
2
    
```

Firstly, the size of this problem is the number of digits of n . The base case occurs when n contains a single digit ($n < 10$), where the algorithm would simply print n . As in the previous example, the simplest decomposition considers $n//10$, where the least significant digit is removed from the original number. Figure 2.7 shows a possible decomposition diagram of the problem.

In addition, the general diagram can also be used for this type of procedure:



In this example the results are no longer numerical values, but a sequence of instructions, which correspond to printed lines on a console. For this problem it is possible to arrive at the solution by printing the least significant digit of the input number, and by calling the recursive method on the remaining digits. However, the order in which these operations are performed is crucial. In particular, the least significant digit



Figure 2.7 Diagram showing a decomposition of the problem consisting of printing the digits of a nonnegative integer on the console, in reversed order, and vertically. A particular ($n = 2743$) and a general m -digit ($n = d_{m-1} \cdots d_1 d_0$) case are shown in (a) and (b), respectively.

Listing 2.3 Code for printing the digits of a nonnegative integer vertically, and in reversed order.

```

1 def print_digits_reversed_vertically(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n % 10)
6         print_digits_reversed_vertically(n // 10)

```

must be printed *before* the rest of the digits. The associated code is shown in [Listing 2.3](#).

2.5.5 Several subproblems

Some algorithms need to decompose a problem into several self-similar subproblems, where the thought process of deriving recursive cases is analogous to the one illustrated in [Figure 2.5](#). The diagrams simply need to include various subproblems and their corresponding solutions, according to the chosen decomposition of the problem, as shown in [Figure 2.8](#). Besides extending and modifying the individual solutions to the subproblems, the recursive cases usually need to combine them as well.

Given a nonempty list of n integers ($n \geq 1$), consider the problem of finding the value of the largest one. For this example, we will decompose the problem by splitting it in two, in order to work with the first and second halves of the list, as shown in [Figure 1.6\(d\)](#). The following diagram illustrates the thought process with a concrete example:

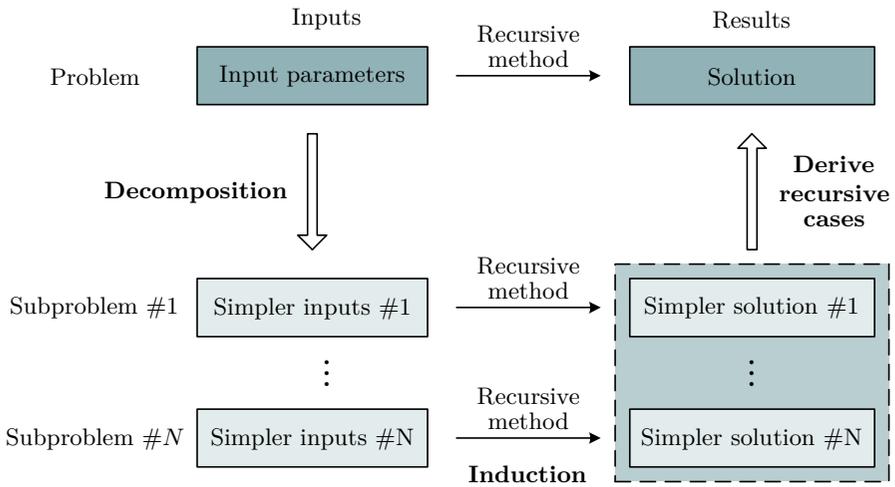


Figure 2.8 General diagram for thinking about recursive cases, when a problem is decomposed into several (N) self-similar subproblems.

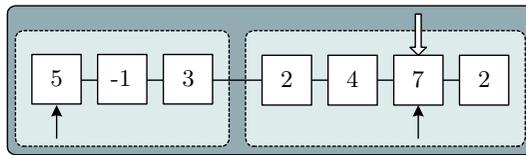


Figure 2.9 Alternative diagram showing a divide and conquer decomposition, and the recursive thought process, for the problem of finding the largest value in a list. The thick and thin arrows point to the solutions of the problem and subproblems, respectively.

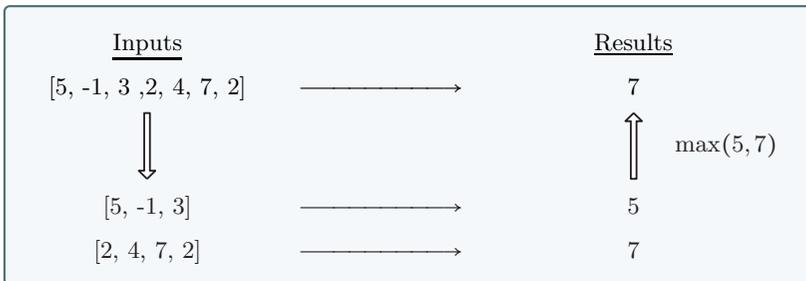


Figure 2.9 shows an alternative diagram that indicates the decomposition and recursive thought process for the problem. Both cases illustrate that each recursive call to one of the halves returns the greatest value in

Listing 2.4 Code for computing the maximum value in a list, through a divide and conquer approach.

```

1 def max_list_length_DaC(a):
2     if len(a) == 1:
3         return a[0]
4     else:
5         middle = len(a) // 2
6         m1 = max_list_length_DaC(a[0:middle])
7         m2 = max_list_length_DaC(a[middle:len(a)])
8         return max(m1, m2)
9
10
11 def max_list_limits_DaC(a, lower, upper):
12     if lower == upper:
13         return a[lower] # or a[upper]
14     else:
15         middle = (upper + lower) // 2
16         m1 = max_list_limits_DaC(a, lower, middle)
17         m2 = max_list_limits_DaC(a, middle + 1, upper)
18         return max(m1, m2)
19
20
21 # Some list:
22 v = [5, -1, 3, 2, 4, 7, 2]
23
24 # Function calls:
25 print(max_list_length_DaC(v))
26 print(max_list_limits_DaC(v, 0, len(v) - 1))

```

the particular half. Therefore, the recursive case can simply return the maximum value in either half. The recursive function (f) is defined as follows:

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

Listing 2.4 shows two ways of coding the function. The version that uses the lower and upper limits is usually faster. Naturally, this problem also allows recursive solutions based on a single subproblem whose size is decreased by a single unit. This approach is as efficient as the divide and conquer strategy. However, in practice it may produce runtime errors for large lists (see [Chapter 10](#)).

Listing 2.5 Erroneous Python code for determining if a nonnegative integer n is even.

```

1 def is_even_incorrect(n):
2     if n == 0:
3         return True
4     else:
5         return is_even_incorrect(n - 2)

```

Listing 2.6 Correct Python code for determining if a nonnegative integer n is even.

```

1 def is_even_correct(n):
2     if n == 0:
3         return True
4     elif n == 1:
5         return False
6     else:
7         return is_even_correct(n - 2)

```

2.6 TESTING

Testing is a fundamental stage in any software development process. In the context of this book its main purpose consists of discovering errors in the code. Testing therefore consists of running the developed software on different instances (i.e., inputs) of a problem in order to detect failures. Novice programmers are strongly encouraged to test their code since the ability to detect and correct errors (e.g., with a debugger) is a basic programming skill. In addition, it teaches valuable lessons in order to avoid pitfalls and code more efficiently and reliably.

Besides checking the basic correctness of the base and recursive cases, when testing recursive code, programmers should pay special attention to possible scenarios that lead to infinite recursions. These usually appear due to missing base cases or by erroneous recursive cases. For example, consider the function in [Listing 2.5](#) whose goal consists of determining whether some nonnegative integer n is even. Both base and recursive cases are correct. Naturally, if a number n is even then so is $n - 2$, and the function must return the same Boolean value for both integers. Nevertheless, `is_even_incorrect` only works for even numbers. Let $f(n)$ represent `is_even_incorrect(n)`. A call to $f(7)$ produces the following

Listing 2.7 Erroneous Python code for adding the first n positive numbers, which produces infinite recursions for most values of n .

```

1 def sum_first_naturals_3(n):
2     if n == 1:
3         return 1
4     else:
5         return 2 * sum_first_naturals_3(n / 2) + (n / 2)**2

```

recursive calls:

$$f(7) \rightarrow f(5) \rightarrow f(3) \rightarrow f(1) \rightarrow f(-1) \rightarrow f(-3) \rightarrow \dots$$

which is an infinite recursion since the process does not halt at a base case. The fact that the function does not contain a base case that returns `False` provides a warning regarding its correction (not all Boolean functions need two base cases in order to return `True` or `False`). Indeed, the method can be fixed by adding that base case. Listing 2.6 shows a function that works for any valid argument ($n \geq 0$).

Another example is the function in Listing 2.7 that uses the recursive case described in (2.1) in order to compute the sum of the first n positive integers ($S(n)$). It is incomplete, and generates infinite recursions for values of n that are not a power of two. Firstly, since Python considers n to be a real number, $n/2$ is also a real number in general. Therefore, if the argument n is an odd integer in any recursive function call then $n/2$ will have a fractional part, and so will the arguments of the following recursive calls. Thus, the algorithm would not halt at the base case $n = 1$ (which is an integer with no fractional part), and would continue to make function calls with smaller and smaller arguments. For example, let $f(n)$ represent `sum_first_naturals_3(n)`. A call to $f(6)$ produces the following recursive calls:

$$f(6) \rightarrow f(3) \rightarrow f(1.5) \rightarrow f(0.75) \rightarrow f(0.375) \rightarrow f(0.1875) \rightarrow \dots$$

never stopping at the base case. The only situation in which the algorithm works is when the first argument n is a power of two, since each of the divisions by two produces an even number, eventually reaching $n = 2$ and afterwards $n = 1$, where the function can finally return a concrete value instead of producing another function call.

The method `sum_first_naturals_3` does not work properly due to real-valued arguments. Thus, we could try replacing the real divi-

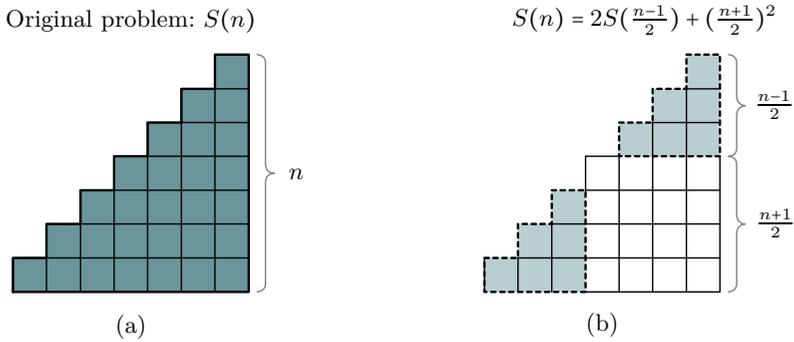


Figure 2.10 Diagram showing a decomposition of the sum of the first n positive integers $S(n)$ that uses two subproblems of (roughly) half the size as the original, when n is odd.

Listing 2.8 Incomplete Python code for adding the first n positive numbers.

```

1 def sum_first_naturals_4(n):
2     if n == 1:
3         return 1
4     else:
5         return 2 * sum_first_naturals_4(n // 2) + (n // 2)**2

```

sions by integer divisions, as shown in Listing 2.8. This forces the arguments to be integers, which prevents infinite recursions. Nevertheless, the function still does not work properly for arguments that are not powers of two. The issue with `sum_first_naturals_4` is that it is incomplete. In particular, although the recursive case is correct, it only applies when n is even. Figure 2.10 shows how to derive the recursive case ($S(n) = 2S((n-1)/2) + ((n+1)/2)^2$) for the problem and decomposition when n is odd. The complete function is therefore:

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2S\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)^2 & \text{if } n > 1 \text{ and } n \text{ is even,} \\ 2S\left(\frac{n-1}{2}\right) + \left(\frac{n+1}{2}\right)^2 & \text{if } n > 1 \text{ and } n \text{ is odd,} \end{cases}$$

and the corresponding code is shown in Listing 2.9. With the new recursive case, every argument to function `sum_first_naturals_5` will also be an integer (given an initial integer input). Finally, replacing the

Listing 2.9 Python code for adding the first n positive numbers based on using two subproblems of (roughly) half the size as the original.

```

1 def sum_first_naturals_5(n):
2     if n == 1:
3         return 1
4     elif n % 2 == 0:
5         return 2 * sum_first_naturals_5(n / 2) + (n / 2)**2
6     else:
7         return (2 * sum_first_naturals_5((n - 1) / 2)
8             + ((n + 1) / 2)**2)

```

real divisions ($/$) by integer divisions ($//$) would also lead to a correct algorithm.

2.7 EXERCISES

Exercise 2.1 — Let n be some positive integer. Consider the problem of determining the number of bits set to 1 in the binary representation of n (i.e., n expressed in base 2). For example, for $n = 25_{10} = 11001_2$ (the subscript indicates the base in which a number is expressed), the result is three bits set to 1. Indicate the size of the problem and provide a mathematical expression for such size.

Exercise 2.2 — Consider the function that adds the first n positive integers in (1.5). Define a more general function that it is also applicable to all nonnegative integers. In other words, modify the function considering that it can also receive $n = 0$ as an input argument. Finally, code the function.

Exercise 2.3 — Use similar diagrams as in Figures 2.6 and 2.10 in order to derive recursive definitions of the sum of the first n positive integers ($S(n)$), where the recursive case will add the result of four subproblems of (roughly) half the size as the original. Finally, define and code the full recursive function.

Exercise 2.4 — Consider the problem of printing the digits of some nonnegative integer n on the console vertically, in “normal” order, where the most significant digit must appear on the first line, the second most significant digit on the second line, and so on. For instance, if $n = 2743$ the program must print the following lines on the console:



Indicate the size of the problem and its base case. Draw a diagram for a general nonnegative input integer $n = d_{m-1} \cdots d_1 d_0$, where m is the number of digits of n , in order to illustrate the decomposition of the problem, and how to recover the solution to the problem given the result of a subproblem. Finally, derive the recursive case and code the method.

Exercise 2.5 — Define a general diagram when using a divide and conquer approach for the problem of calculating the largest value in a list \mathbf{a} of n elements. Use appropriate general notation instead of concrete examples.

Exercise 2.6 — Define a recursive function that calculates the largest value in a list \mathbf{a} of n elements, where the decomposition simply reduces the size of the problem by a unit.

Runtime Analysis of Recursive Algorithms

The faster you go, the shorter you are.

— Albert Einstein

ALGORITHM analysis is the field that studies how to theoretically estimate the resources that algorithms need in order to solve computational problems. This chapter focuses on analyzing the runtime, also denoted as “computational time complexity,” of recursive algorithms that solve problems whose size depends on a single factor (which occurs in the majority of the problems covered in the book). This will provide a context that will enable us to characterize and compare different algorithms regarding their efficiency. In particular, the chapter describes two methods for solving recurrence relations, which are recursive mathematical functions that describe the computational cost of recursive algorithms. These tools are used in order to transform a recurrence relation into an equivalent nonrecursive function that is easier to understand and compare. In addition, the chapter provides an overview of essential mathematical fundamentals and notation used in the analysis of algorithms. Lastly, the analysis of memory (storage) or space complexity of recursive algorithms will be covered in [Chapter 10](#).

3.1 MATHEMATICAL PRELIMINARIES

This section presents a brief introduction to basic mathematical definitions and properties that appear in the analysis of the computational

complexity of algorithms, and in several problems covered throughout the book.

3.1.1 Powers and logarithms

The following list reviews essential properties of powers and logarithms:

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

where a , b , x , and y are arbitrary real numbers, with the exceptions that: (1) the base of a logarithm must be positive and different than 1, (2) a logarithm is only defined for positive numbers, and (3) the denominator in a fraction cannot be 0. For example, $\log_b x = \log_a x / \log_a b$ is only valid for $a > 0$ with $a \neq 1$, $b > 0$ with $b \neq 1$, and $x > 0$.

Logarithms and powers of positive numbers are monotonically increasing functions. Therefore, if $x \leq y$ then $\log_b x \leq \log_b y$, and $b^x \leq b^y$ (for valid values of x , y , and b).

3.1.2 Binomial coefficients

A binomial coefficient, denoted as $\binom{n}{m}$, is an integer that appears in the polynomial expansion of the binomial power $(1+x)^n$. It can be defined as:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = m, \\ \frac{n!}{m!(n-m)!} & \text{otherwise,} \end{cases} \quad (3.1)$$

where n and m are integers that satisfy $n \geq m \geq 0$. In addition, a binomial coefficient can be defined recursively through:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = m \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{otherwise.} \end{cases} \quad (3.2)$$

Binomial coefficients play an important role in combinatorics. In particular, $\binom{n}{m}$ determines the number of ways it is possible to choose m elements from a set of n distinct ones, where the order of selection does not matter.

3.1.3 Limits and L'Hopital's rule

The computational cost of different algorithms can be compared through limits involving a quotient of functions. Firstly,

$$\frac{k}{\infty} = 0, \quad \text{and} \quad \frac{\infty}{k} = \infty,$$

where k is a constant, and where ∞ should be understood as the result of a limit. In addition, since functions that measure computational cost are generally increasing (with the exception of constants), the limit when their input approaches infinity will also be infinity. For example:

$$\lim_{n \rightarrow \infty} \log_b n = \infty, \quad \text{or} \quad \lim_{n \rightarrow \infty} n^a = \infty,$$

for a valid base b for the logarithm, and for $a > 0$. Therefore, the following indeterminate form appears frequently:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{\infty}{\infty}.$$

It is usually solved by simplifying the fraction $f(n)/g(n)$ until it is possible to obtain a result that is not an indeterminate form. A well-known approach for simplifying the fraction is to use L'Hopital's rule:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}, \quad (3.3)$$

where $f'(n)$ and $g'(n)$ are the derivatives of $f(n)$ and $g(n)$, respectively. Formally, L'Hopital's rule is only valid when the limit on the right-hand side of (3.3) exists, which is usually the case.

3.1.4 Sums and products

It is important to be familiarized with sums since they not only appear in function definitions and formulas that we may need to code, but also when analyzing the efficiency of both iterative and recursive algorithms. A sum, or summation, is simply the addition of a collection of mathematical entities (integers, real numbers, vectors, matrices, functions, etc.). When a sum involves adding some function $f(i)$, evaluated at consecutive integer values from an initial integer index m up to a final index n , it can be expressed compactly by using “sigma notation”:

$$\sum_{i=m}^n f(i) = f(m) + f(m+1) + \cdots + f(n-1) + f(n). \quad (3.4)$$

Thus, the result of the sum is simply the addition of terms that arise from substituting every occurrence of i in function $f(i)$ with integers from m up to n . For instance:

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

where $f(i) = ki^2$.

It is important to understand that the result of the sum does not depend on the index variable. For example, the sum in (3.4) logically depends on the function f , and on the particular limits m and n , but not on any specific value of i . For instance, notice that i does not appear on the right-hand side of (3.4). Therefore, the integer index is simply a “dummy” variable that allows us to indicate all of the terms that need to be added with a single expression. In this regard, we can choose any name for the index variable, say j or k :

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

Furthermore, we can express the same sum in different ways by performing a change of variable:

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

In the second sum the limits (and parameter of f) appear shifted. The third sum simply adds the terms in “reverse” order (when $i = m$ it adds $f(n)$, while when $i = n$ the term added is $f(m)$). Finally, if the lower limit m is greater than the upper limit n , then the sum evaluates to 0, by convention.

3.1.4.1 Basic properties of sums

The following basic properties are useful for simplifying and working with sums, and can be derived easily from the addition and multiplication properties of the type of elements to be added. Firstly,

$$\sum_{i=1}^n 1 = 1 + 1 + \cdots + 1 + 1 = n.$$

n times

Notice that $f(i)$ is a constant (1) that does not depend on the index variable i . Similarly,

$$\sum_{i=1}^n k = k + k + \cdots + k + k = kn.$$

n times

The previous example also illustrates that when a multiplicative term in f does not depend on the index variable it can be “pulled out” of the sum:

$$\sum_{i=1}^n k = \sum_{i=1}^n (k \cdot 1) = \sum_{i=1}^n k \cdot 1 = k \cdot \left(\sum_{i=1}^n 1 \right) = k \sum_{i=1}^n 1 = kn. \quad (3.5)$$

In this case the constant term k , which we can consider is multiplied by 1, does not depend on i , and can be pulled out of the sum, where it appears multiplying it (regarding notation, (3.5) shows that it is not necessary to use parenthesis inside a sum if f does not contain additive terms, nor if some term multiplies a sum). In general, the property can be expressed as:

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

which follows from the distributive law of multiplication over addition, where the expression has been simplified by extracting the common factor k from all of the terms being added. Naturally, the factor k can be the product of several terms that do not depend on the index, and may contain the upper and lower limits, as shown in the following example:

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

where a is some constant.

Finally, sums where the function f contains several additive terms can be broken up into simpler individual sums. Formally,

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

3.1.4.2 Arithmetic series

An arithmetic series is the sum of the terms in a sequence s_i , for $i = 0, 1, 2, \dots$, in which each term is equal to the previous one plus a particular constant d (which could be negative). In other words, $s_i = s_{i-1} + d$, for $i > 0$, which is a recursive definition. These terms can also be described nonrecursively as follows:

$$s_i = s_{i-1} + d = s_{i-2} + 2d = \dots = s_0 + id. \quad (3.6)$$

While technically an arithmetic series is an infinite sum:

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

when analyzing the runtime of algorithms the quantities of interest only add the terms of finite sequences:

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

also referred to as partial sums.

The sum of the first n positive integers ($S(n)$) is used frequently when analyzing the efficiency of recursive and iterative algorithms, and can be understood as a partial sum (of n elements) of the arithmetic series with $s_0 = 1$, and $d = 1$:

$$\begin{aligned} S(n) &= \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} (s_0 + id) = \sum_{i=0}^{n-1} (1 + i) \\ &= \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} i = n + \sum_{i=0}^{n-1} i = \sum_{i=1}^n i. \end{aligned}$$

As shown in [Section 1.4.1](#), its result can be expressed as a quadratic polynomial (see (1.12)). A simple way to derive the formula consists of adding two sums $S(n)$, one expressed with terms in increasing order, and the other in decreasing order:

$$\begin{array}{r} S(n) = 1 + 2 + \dots + (n-1) + n \\ + \\ S(n) = n + (n-1) + \dots + 2 + 1 \\ \hline 2S(n) = (n+1) + (n+1) + \dots + (n+1) + (n+1) \end{array}$$

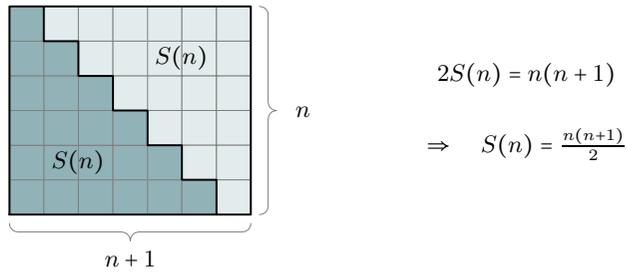


Figure 3.1 Graphical mnemonic for determining the quadratic formula for the sum of the first n positive integers ($S(n)$).

It follows from the result that $2S(n) = n(n+1)$, since there are n terms (each equal to $n+1$) on the right-hand side of the identity. Finally, dividing by 2 yields:

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

The idea can also be understood through the graphical mnemonic in [Figure 3.1](#), where $S(n)$ is the area of a triangular pyramid of square (1×1) blocks. Two of these triangular structures can be joined together in order to form a rectangle of area $n \times (n+1)$. Therefore, it follows that the area ($S(n)$) of each triangular structure is $n(n+1)/2$.

A similar formula can be obtained for the general partial sum of n terms of an arithmetic sequence ($s_i = s_{i-1} + d$, for some initial s_0):

$$\sum_{i=0}^{n-1} s_i = \frac{n}{2}(s_0 + s_{n-1}),$$

which is the average between the first and last elements of the sequence, multiplied by the number of elements in the sequence.

3.1.4.3 Geometric series

A geometric series is the sum of the terms in a sequence s_i , for $i = 0, 1, 2, \dots$, in which each term is equal to the previous one times a particular constant r . In other words, $s_i = r \cdot s_{i-1}$, for $i > 0$. These terms can also be described nonrecursively as follows:

$$s_i = r \cdot s_{i-1} = r^2 \cdot s_{i-2} = \dots = r^i \cdot s_0. \quad (3.7)$$

The partial sum of a geometric series can be obtained through the following formula:

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

for $r \neq 1$. In practice the last equality can be derived easily through the following approach. First, let S represent the value of the sum. Subsequently, create another sum by multiplying S by r . Finally, subtract both sums (where most terms cancel out), and solve for S . The process (ignoring s_0) can be illustrated as follows:

$$\begin{array}{r} S \\ - rS \\ \hline S - rS \end{array} = \begin{array}{r} r^m + r^{m+1} + r^{m+2} + \dots + r^{n-1} + r^n \\ + r^{m+1} + r^{m+2} + \dots + r^{n-1} + r^n + r^{n+1} \\ r^m \qquad \qquad \qquad - r^{n+1} \end{array}$$

Therefore, solving for S yields:

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

Finally, a geometric series converges to a constant value if $|r| < 1$:

$$\sum_{i=0}^{\infty} ar^i = a \frac{1}{1 - r}.$$

3.1.4.4 Differentiation

Another useful sum is:

$$S = \sum_{i=1}^n ir^i = 1r^1 + 2r^2 + 3r^3 + \dots + nr^n, \quad (3.10)$$

which can be interpreted as a hybrid between arithmetic and geometric series. It is possible to derive a formula for the sum as follows. Firstly, consider a general partial sum of a geometric series and its corresponding simplified formula:

$$T = 1 + r + r^2 + \dots + r^n = \frac{r^{n+1} - 1}{r - 1}.$$

We then differentiate with respect to r :

$$\frac{dT}{dr} = 1 + 2r + 3r^2 + \dots + nr^{n-1} = \frac{nr^{n+1} - (n+1)r^n + 1}{(r-1)^2}, \quad (3.11)$$

and multiply by r in order to obtain a formula for (3.10):

$$S = \sum_{i=1}^n ir^i = r + 2r^2 + \dots + nr^n = r \frac{nr^{n+1} - (n+1)r^n + 1}{(r-1)^2}. \quad (3.12)$$

We can use these formulas in order to simplify similar sums. For example, let

$$S = \sum_{i=1}^{N+1} i2^{i-1} = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 8 + \dots + (N+1) \cdot 2^N.$$

It is a special case of (3.11) for $r = 2$ and $n = N + 1$. Substituting, we obtain:

$$\begin{aligned} S &= \frac{(N+1)2^{N+2} - (N+2)2^{N+1} + 1}{1} = 2^{N+1}(2N+2 - N-2) + 1 \\ &= N2^{N+1} + 1. \end{aligned}$$

3.1.4.5 Products

Similarly to the notation used for sums, a product of several terms of some function $f(i)$, evaluated at consecutive integer values from an initial index m up to a final index n , can be written as follows:

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

where by convention the product is 1 if $m > n$. For example, the factorial function can be expressed as:

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

where for mathematical convenience $0! = 1$.

Similarly to sums, in products multiplicative terms that do not depend on the index variable can also be pulled out of the product. However, if the product involves n terms, they have to be raised to the power of n :

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

In addition, the product of a sum is not the sum of the products in general:

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

Lastly, the logarithm of a product is a sum of logarithms:

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

3.1.5 Floors and ceilings

The floor of a real number x , denoted as $\lfloor x \rfloor$, is the largest integer that is less than or equal to x . Similarly, the ceiling of x , denoted as $\lceil x \rceil$, is the smallest integer that is greater than or equal to x . Formally, they can be described as:

$$\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leq x\},$$

$$\lceil x \rceil = \min\{m \in \mathbb{Z} \mid m \geq x\},$$

where \mathbb{Z} represents the set of all integers, and \mid can be read as “such that.” The following list includes several basic properties of floors and ceilings:

- $\lfloor x \rfloor \leq x$
- $\lceil x \rceil \geq x$
- $\lfloor x + n \rfloor = \lfloor x \rfloor + n$
- $\lceil x + n \rceil = \lceil x \rceil + n$
- $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
- $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$
- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$
- $n - 2\lfloor n/2 \rfloor = 0 \Leftrightarrow n$ is even
- $n // 2 = \lfloor n/2 \rfloor$
- $n \gg m = \lfloor n/2^m \rfloor$
- $\lfloor \log_{10} p \rfloor + 1 =$ the number of decimal digits of p
- $\lfloor \log_2 p \rfloor + 1 =$ the number of binary digits (bits) of p

where x is a real number, n is an integer, m is a nonnegative integer, and p is a positive integer. In addition, `//` and `>>` represent operators in Python (and also in other programming languages) that compute the quotient of an integer division, and perform a right bit-shift, respectively. Finally, \Leftrightarrow denotes “if and only if.”

3.1.6 Trigonometry

Consider the right triangle in [Figure 3.2](#). The following list reviews basic trigonometric definitions and properties:

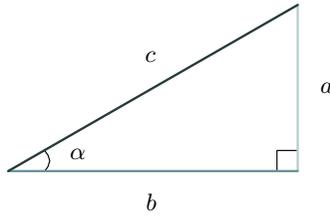


Figure 3.2 Right triangle used for showing trigonometric definitions.

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

where \sin , \cos , \tan , and \cot denote sine, cosine, tangent, and cotangent, respectively. In most programming languages, the arguments of the trigonometric functions are in radians (one radian is equal to $180/\pi$ degrees).

3.1.7 Vectors and matrices

A **matrix** is a collection of numbers (when programming they can be other data types such as characters, Boolean values, etc.) that are arranged in rows and columns forming a rectangular structure. Formally, an $n \times m$ -dimensional matrix \mathbf{A} contains n rows and m columns of numbers, for $n \geq 1$ and $m \geq 1$. Typically, matrices are written within brackets or parentheses:

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

where $a_{i,j}$ denotes the particular element or entry of matrix \mathbf{A} in its i -th row and j -th column.

If one of the dimensions is equal to one the mathematical entity is called a **vector**, while if both dimensions are equal to one the object is a **scalar** (i.e., a number). In this book we will use a standard notation where matrices are represented by capital boldface letters (\mathbf{A}), vectors through boldface lower case letters (\mathbf{a}), and scalars by italic lower case letters or symbols (a).

The transpose of an $n \times m$ -dimensional matrix \mathbf{A} is the $m \times n$ -dimensional matrix \mathbf{A}^T whose rows are the columns of \mathbf{A} (and therefore its columns are the rows of \mathbf{A}). For example, if:

$$\mathbf{A} = \begin{bmatrix} 3 & 4 & 2 \\ 1 & 8 & 5 \end{bmatrix}, \quad \text{then} \quad \mathbf{A}^T = \begin{bmatrix} 3 & 1 \\ 4 & 8 \\ 2 & 5 \end{bmatrix}.$$

It is possible to add and multiply matrices (and vectors). The sum $\mathbf{A} + \mathbf{B}$ is a matrix whose entries are $a_{i,j} + b_{i,j}$. In other words, matrices (and vectors) are added entrywise. Thus, \mathbf{A} and \mathbf{B} must share the same dimensions. For example:

$$\begin{bmatrix} 4 \\ -1 \\ 2 \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \\ -7 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \\ -5 \end{bmatrix},$$

is a basic vector sum.

Instead, multiplication is more complex. Let \mathbf{a} and \mathbf{b} be two n -dimensional (column) vectors. Their **dot product**, expressed as $\mathbf{a}^T \mathbf{b}$ (other notations include $\vec{\mathbf{a}} \cdot \vec{\mathbf{b}}$, or $\langle \mathbf{a}, \mathbf{b} \rangle$), is defined as the sum of their entrywise products:

$$\mathbf{a}^T \mathbf{b} = \sum_{i=1}^n a_i b_i. \quad (3.14)$$

Vectors of n components (i.e., defined in \mathbb{R}^n) also have a geometrical interpretation. It can be shown that:

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

where α is the angle between the vectors \mathbf{a} and \mathbf{b} , and $|\cdot|$ denotes the Euclidean norm:

$$|\mathbf{a}| = \sqrt{a_1^2 + \dots + a_n^2}. \quad (3.16)$$

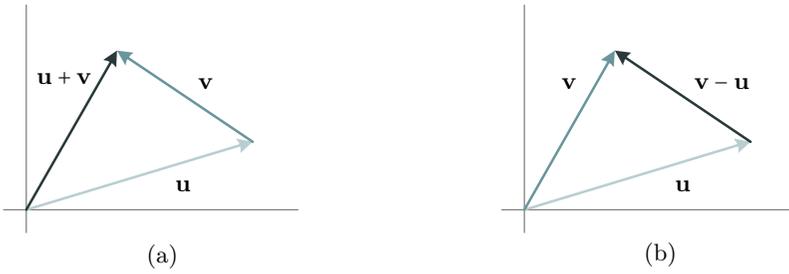


Figure 3.3 Geometric interpretation of vector addition and subtraction.

Alternatively, let \mathbf{A} and \mathbf{B} be $n \times m$ and $m \times p$ -dimensional matrices, respectively. The product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is an $n \times p$ -dimensional matrix whose entries are defined as:

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

In this case, the number of columns of \mathbf{A} (i.e., m) must be the same as the number of rows of \mathbf{B} . The entry $c_{i,j}$ corresponds to the dot product between the i -th row of \mathbf{A} , and the j -th column of \mathbf{B} (which are obviously both vectors). For example:

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

which is a **symmetric** matrix, since it is identical to its transpose.

Vectors can also be regarded as points. Geometrically, adding two vectors \mathbf{u} and \mathbf{v} can be understood as creating a new vector whose endpoint is the result of “concatenating” \mathbf{u} and \mathbf{v} , as shown in Figure 3.3(a). It therefore follows that the vector that begins at the endpoint of \mathbf{u} and ends at the endpoint of \mathbf{v} is the vector $\mathbf{v} - \mathbf{u}$, as illustrated in (b).

Lastly, a 2-dimensional vector can be rotated counterclockwise α degrees (or radians) by multiplying it times the following “rotation” matrix:

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

as shown in Figure 3.4, where \mathbf{u} is a column vector (in most mathematical texts vectors are expressed as column vectors).

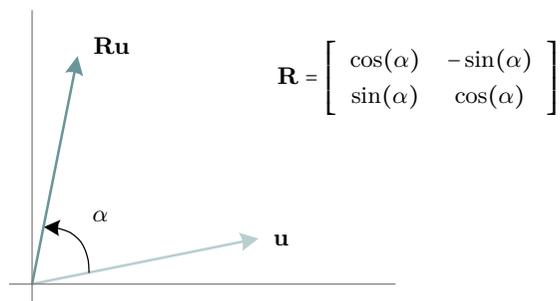


Figure 3.4 Rotation matrix (counterclockwise).

Listing 3.1 Measuring execution times through Python’s `time` module.

```

1 import time
2
3 t = time.time()
4 # execute some code here
5 elapsed_time = time.time() - t
6 print(elapsed_time)

```

3.2 COMPUTATIONAL TIME COMPLEXITY

The computational time complexity of an algorithm is a theoretical measure of how much time, or how many operations, it needs to solve a problem (Listing 3.1 shows a simple way to measure execution times in practice, through Python’s `time` module). It is determined by analyzing a function, say T , of the input size that quantifies this number of operations for a particular instance. In computer science the efficiency of algorithms is generally studied by contemplating how T behaves when the size of the problem is very large. Moreover, the key factor is the rate at which T grows as the input size tends to infinity. The following subsections explain these ideas and the mathematical notation typically used to characterize computational time complexity. Lastly, while the size of a problem may depend on several factors, in this introductory book we will analyze the computational time complexity of algorithms that solve problems whose size is determined by only one factor. Thus, the runtime cost of the covered algorithms will be determined by a function $T(n)$ of one parameter, where n usually represents the size of the problem.

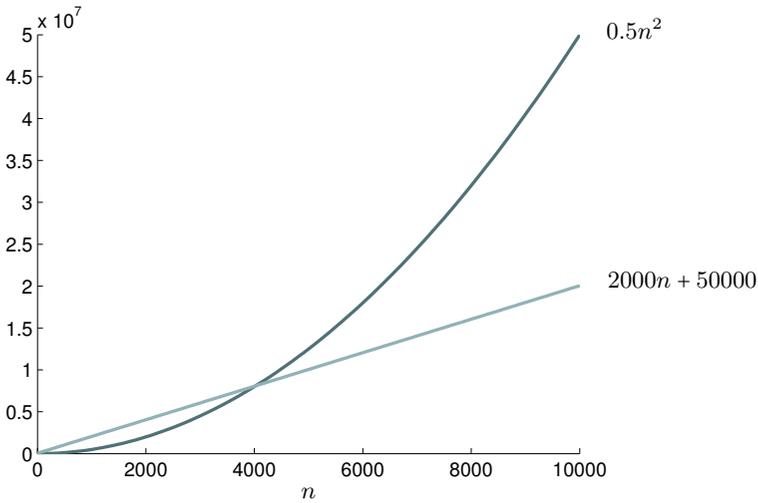


Figure 3.5 The highest-order term determines the order of growth of a function. For $T(n) = 0.5n^2 + 2000n + 50000$ the order is quadratic, since the term $0.5n^2$ clearly dominates the lower-order terms (even added up) for large values of n .

3.2.1 Order of growth of functions

The function $T(n)$ that quantifies the runtime of an algorithm may contain several additive terms that contribute differently to its value for a given input. For example, let $T(n) = 0.5n^2 + 2000n + 50000$. For moderate values of n all of the terms are relevant regarding the magnitude of $T(n)$. However, as n increases the leading term ($0.5n^2$) affects the growth of the function significantly more than the other two (even combined). Therefore, the **order of growth** of the function is characterized by its highest-order term. Figure 3.5 shows a plot of $0.5n^2$ together with $2000n + 50000$. For large inputs it is apparent that the quadratic term dominates the other two, and therefore characterizes the rate of growth of $T(n)$. The coefficients of the polynomial have been chosen in order to illustrate that they do not play a significant role in determining the function's rate of growth.

Figure 3.6 plots several orders of growth that appear frequently in computational complexity. They can be sorted as follows when considering large values of n :

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

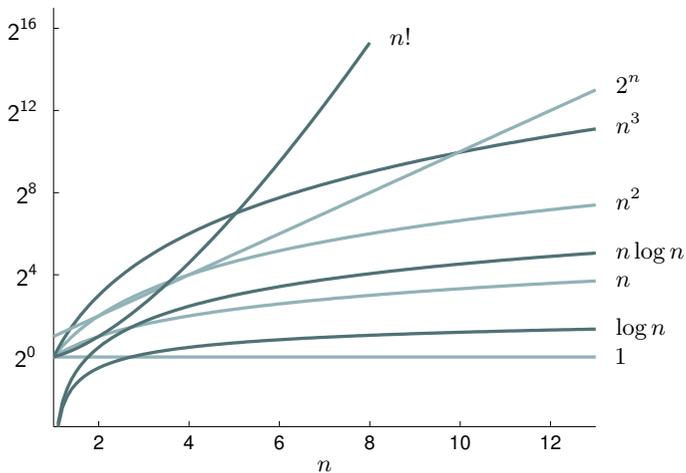


Figure 3.6 Orders of growth typically used in computational complexity.

where informally $f(n) < g(n)$ if:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

The previous orders of growth are called (from left to right) “constant,” “logarithmic,” “linear,” “n-log-n,” “quadratic,” “cubic,” “exponential,” and “factorial.”

Since the scale of the Y axis in Figure 3.6 is logarithmic, the differences between the orders of growth may appear to be much smaller than

Table 3.1 Concrete values of common functions used in computational complexity.

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	40320
1	4	16	64	256	4096	65.536	$2.09 \cdot 10^{13}$
1	5	32	160	1024	32.768	4.295.967.296	$2.63 \cdot 10^{35}$

they actually are (the difference between consecutive tick marks means that an algorithm is 16 times slower/faster). Table 3.1 shows concrete values for the functions, where the fast growth rates of the exponential or factorial functions clearly stand out. Problems that cannot be solved by algorithms in polynomial time are typically considered to be intractable, since it would take too long for the methods to terminate even for problems of moderate size. In contrast, problems that can be solved in polynomial time are regarded as tractable. However, the line between tractable and intractable problems can be subtle. If the runtime of an algorithm is characterized by a polynomial order with a large degree, in practice it could take too long to obtain a solution, or for its intermediate results to be useful.

3.2.2 Asymptotic notation

An important detail about the orders mentioned so far is the lack of constant multiplicative terms. Similarly to lower-order terms, these are omitted since they are less relevant than the actual order of growth when determining the computational efficiency for large inputs. Moreover, it is not worth the effort to specify the efficiency of an algorithm with exact precision, since its runtime depends on numerous factors that include the computer's hardware, the programming language, the compiler or interpreter, and many others. Thus, it is sufficient to assume in practice that it simply takes a constant amount of time to execute a basic instruction, where its value is irrelevant.

The theoretical analysis of algorithms and problems relies on a type of notation called “asymptotic notation,” which allows us to discard the lower-order terms and multiplicative constants when dealing with arbitrarily large inputs. In particular, asymptotic notation provides definitions of sets that we can use in order to specify “asymptotic bounds.”

Big-O notation defines the following set:

$$\mathcal{O}(g(n)) = \left\{ f(n) : \exists c > 0 \text{ and } n_0 > 0 / 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \right\}.$$

If a function $f(n)$ belongs to this set, then $g(n)$ will be an **asymptotic upper bound** for $f(n)$. This means that $g(n)$ will be greater than $f(n)$, but the definition only requires this to be true on an interval from some point $n_0 > 0$ until infinity, where in addition we can multiply $g(n)$ times a sufficiently large positive constant c . Figure 3.7(a) illustrates the idea graphically. If $g(n)$ is an asymptotic upper bound for $f(n)$ then it must be possible to find a positive constant c such that $cg(n) \geq f(n)$, but

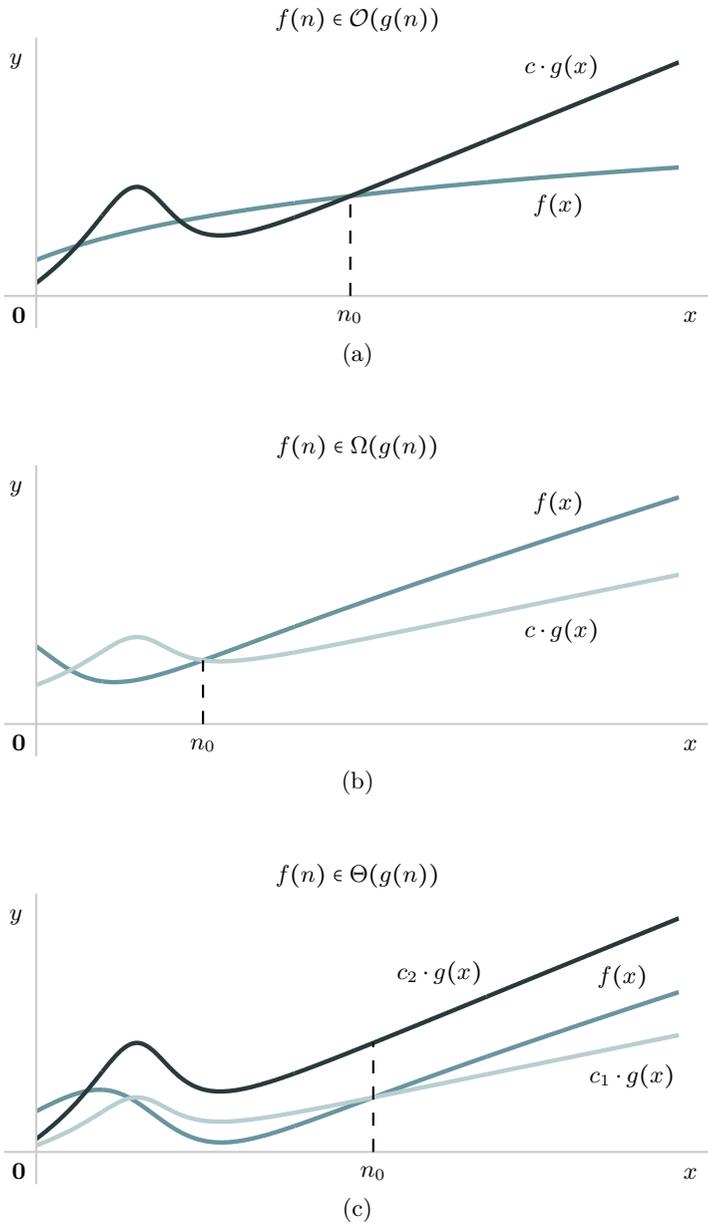


Figure 3.7 Graphical illustrations of asymptotic notation definitions for computational complexity.

from some positive value n_0 until infinity (whatever happens for $n < n_0$ is irrelevant). In order to prove that a function belongs to $\mathcal{O}(g(n))$ it is sufficient to show the existence of a pair of constants c and n_0 that will satisfy the definition, since they are not unique. For instance, if the definition is true for some particular c and n_0 , then it will also be true for larger values of c and n_0 . In this regard, it is not necessary to provide the lowest values of c and n_0 that satisfy the definition \mathcal{O} (this also applies to the notations mentioned below).

Algorithms are often compared according to their efficiency in the **worst case**, which corresponds to an instance of a problem, amongst all that share the same size, for which the algorithm will require more resources (time, storage, etc.). Since Big-O notation specifies asymptotic upper bounds, it can be used in order to provide a guarantee that a particular algorithm will need at most a certain amount of resources, even in a worst-case scenario, for large inputs. For example, the running time for the quicksort algorithm that sorts a list or array of n elements (see [Section 6.2.2](#)) belongs to $\mathcal{O}(n^2)$ in general, since it requires carrying out on the order of n^2 comparisons in the worst case. However, quicksort can run faster (in $n \log n$ time) in the best and average cases.

In contrast, Big-Omega notation defines **asymptotic lower bounds**:

$$\Omega(g(n)) = \left\{ f(n) : \exists c > 0 \text{ and } n_0 > 0 / 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \right\}.$$

[Figure 3.7\(b\)](#) illustrates the idea graphically. Big-Omega notation is useful for specifying a lower bound on the resources needed to solve a problem, no matter which algorithm is applied. For instance, it is possible to prove theoretically that any algorithm capable of sorting a list of n real numbers will require $\Omega(n \log n)$ comparisons in the worst case.

Finally, Big-Theta notation defines **asymptotic tight bounds**:

$$\Theta(g(n)) = \left\{ f(n) : \exists c_1 > 0, c_2 > 0, \text{ and } n_0 > 0 / \right. \\ \left. 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \right\}.$$

If $f(n) \in \Theta(g(n))$ then $f(n)$ and $g(n)$ will share the same order of growth. Thus, by choosing two appropriate constants c_1 and c_2 , $g(n)$ will be both an upper and lower asymptotic bound of $f(n)$, as shown in [Figure 3.7\(c\)](#). In other words, $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$. For example, the merge sort algorithm for sorting a list or array of n elements always requires (in the best and worst case) on the order of $n \log n$ comparisons. Therefore, we say its running time belongs to $\Theta(n \log n)$.

When specifying an asymptotic bound the constants and lower-order terms are dropped from the function $g(n)$. For example, although it is true that $3n^2 + 10n \in O(5n^2 + 20n)$, it is sufficient to indicate $3n^2 + 10n \in O(n^2)$, since these notations indicate orders of growth. In this regard, the reader may have noticed the lack of the base of a logarithm when describing the order of a function. The reason for this is that the difference between two logarithms of different bases is a constant multiplicative term. Let ρ represent some asymptotic bound, and a and b two bases for logarithms. The following identities indicate that all logarithms share the same order, regardless of their base:

$$\rho(\log_a g(n)) = \rho\left(\frac{\log_b g(n)}{\log_b a}\right) = \rho\left(\underbrace{\frac{1}{\log_b a}}_{\text{constant}} \log_b g(n)\right) = \rho(\log_b g(n)).$$

Thus, the base of a logarithm is not specified when indicating its order of growth.

Finally, we can also use limits to determine the order of functions, due to the following equivalent statements that relate them to the definitions of asymptotic bounds:

$$\begin{aligned} f(n) \in \mathcal{O}(g(n)) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \quad (\text{constant or zero}), \\ f(n) \in \Omega(g(n)) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0 \quad (\text{constant} > 0, \text{ or infinity}), \\ f(n) \in \Theta(g(n)) &\Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \text{constant} > 0. \end{aligned}$$

3.3 RECURRENCE RELATIONS

The running time or number of operations carried out by a recursive algorithm is specified through a recurrence relation (or simply, “recurrence”), which is a recursive mathematical function, say T , that describes its computational cost. Consider the code in [Listing 1.1](#) for adding the n first positive integers. Firstly, the number of operations that it needs to perform clearly depends on the input parameter n . Thus, T will be a function of n .

In the base case (when $n = 1$) the method performs the basic operations shown in [Figure 3.8](#). Before carrying out instructions the program needs to store low-level information (e.g., regarding the parameters, or

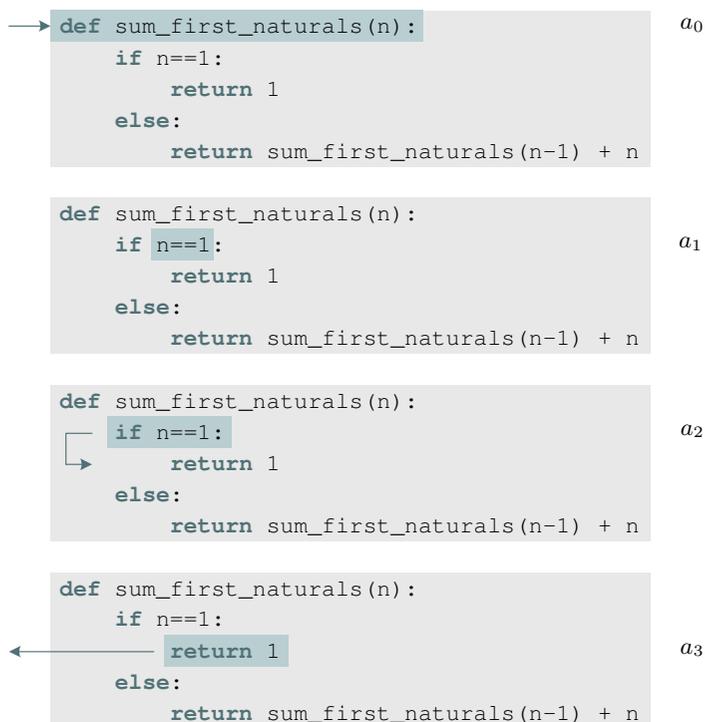


Figure 3.8 Sequence of operations carried by the function in Listing 1.1 in the base case.

the return address). Say this requires a_0 units of computing time, where a_0 is a simple constant. The next basic operation evaluates the condition, taking a_1 units of time. Since the result is `True`, the next operation is a “jump” to the third line of the method, which requires a_2 units of time. Finally, the method can return the value 1 in the last step, requiring a_3 units of time. In total, the method requires $a = a_0 + a_1 + a_2 + a_3$ units of time for $n = 1$. Thus, we can define $T(1) = a$. The exact value of a is irrelevant regarding the asymptotic computational complexity of the method. What is important is that a is a constant quantity that does not depend on n .

Alternatively, Figure 3.9 shows the operations carried out in the recursive case (when $n > 1$). Let $b = \sum_{i=0}^5 b_i$ be the total computing time needed to carry out the basic operations (store low-level information, evaluate the condition, jump to the recursive case, subtract a unit from n , add n to the output of the recursive call, and return the result), which

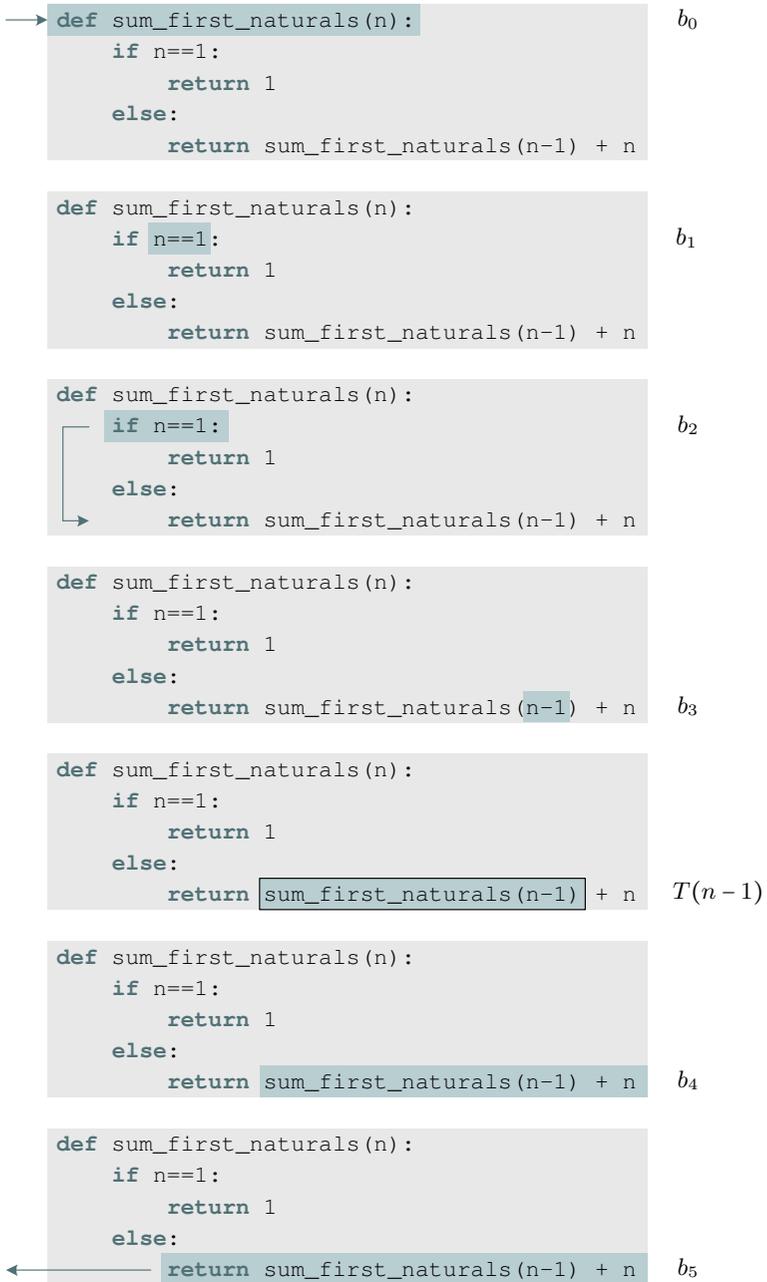


Figure 3.9 Sequence of operations carried by the function in Listing 1.1 in the recursive case.

is also a constant whose exact value is irrelevant regarding asymptotic computational complexity. In addition to b , we need to consider the time required by the recursive call. Since it solves a full problem of size $n - 1$, we can define it as $T(n - 1)$. Altogether, the recurrence relation $T(n)$ can be specified as follows:

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

where, for example, $T(3) = T(2) + b = T(1) + b + b = a + 2b$.

Although T describes the algorithm's computational cost correctly, it is not trivial to figure out its order of growth since it is recursive. Therefore, the next step of the analysis consists of transforming the function into an equivalent nonrecursive form. This process is referred to as “solving” the recurrence relation. In this example, it is not hard to see that $T(n)$ is a linear function of n :

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

The next sections will cover methods for solving common recurrence relations.

In addition, in this introductory text we will simplify the recurrence relations in order to ease the analysis. Consider the code in [Listing 2.9](#). Its associated runtime cost function can be defined as:

$$T(n) = \begin{cases} a & \text{if } n = 1, \\ T\left(\frac{n}{2}\right) + b & \text{if } n > 1 \text{ and } n \text{ is even,} \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c & \text{if } n > 1 \text{ and } n \text{ is odd.} \end{cases} \quad (3.19)$$

This recurrence relation is hard to analyze for two reasons. On the one hand, it contains more than one recursive case. On the other hand, although it is possible to work with the floor function, it is subject to technicalities and requires more complex mathematics (e.g., inequalities). Moreover, the extra complexity that stems from separating the odd and even cases, and dealing with a recurrence of finer detail, is unnecessary regarding the function's order of growth. Since the algorithm is based on solving subproblems of (approximately, in the odd case) half the size, we can work with the following recurrence relation instead:

$$T(n) = \begin{cases} a & \text{if } n = 1, \\ T\left(\frac{n}{2}\right) + b & \text{if } n > 1. \end{cases} \quad (3.20)$$

Using this function is analogous to utilizing (3.19) and assuming that the size of the problem (n) will be a power of two, which can be arbitrarily large.

Therefore, in this book we will cover recurrence relations with only one recursive case, which will not involve the floor or ceiling functions. This will allow us to determine exact nonrecursive definitions of recurrence relations whose order of growth can be characterized by the tight Θ asymptotic bound.

3.3.1 Expansion method

The expansion method, also known as the “iteration” or “backward substitution” method, can be used primarily to solve recurrence relations whose recursive case contains one reference to the recursive function (in some cases it can be applied to recurrences where the recursive function appears several times in the recursive definitions). The idea consists of simplifying the recurrence relation progressively step by step, until noticing a general pattern at some i -th stage. Subsequently, the function can take concrete values by considering that the base case is reached at that i -th step. The following examples illustrate the procedure. Finally, Section 10.2.1 illustrates a related visual approach called the “tree method.”

3.3.1.1 Common recurrence relations

Consider the function defined in (3.18). Its recursive case:

$$T(n) = T(n - 1) + b \quad (3.21)$$

can be applied repeatedly (to arguments of smaller size) in order to “expand” the T term of the right-hand side. For example, $T(n - 1) = T(n - 2) + b$, where all we have done is substitute n with $(n - 1)$ in (3.21). Thus, we arrive at:

$$T(n) = \underbrace{[T(n - 2) + b]}_{T(n-1)} + b = T(n - 2) + 2b,$$

where the expression inside the square brackets is the expansion of $T(n - 1)$. The idea can be applied again expanding $T(n - 2)$, which is $T(n - 3) + b$. Thus, at a third step we obtain:

$$T(n) = \underbrace{[T(n - 3) + b]}_{T(n-2)} + 2b = T(n - 3) + 3b.$$

1. Write down the recursive case of the recurrence relation.
2. Expand the recursive terms (T) on the right-hand side several times until detecting a general pattern for the i -th step.
3. Determine the value of i that allows us to reach a base case.
4. Substitute for i in the general pattern.

Figure 3.10 Summary of the expansion method.

After several of these expansions we should be able to detect a general pattern corresponding to the i -th step. For this function it is:

$$T(n) = T(n - i) + ib. \quad (3.22)$$

Finally, for some value of i the process will reach a base case. For function (3.18) it is defined for $T(1)$. Therefore, the term $T(n - i)$ will correspond to the base case when $n - i = 1$, or equivalently, when $i = n - 1$. Substituting this result in (3.22) allows us to eliminate the variable i from the formula, and provide a full nonrecursive definition of $T(n)$:

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

Thus, Listing 1.1 runs in linear time with respect to n .

Figure 3.10 presents a summary of the expansion method that we will now apply to other common recurrence relations. Consider the function defined in (3.20). The expansion process is:

$$\begin{aligned}
 T(n) &= T(n/2) + b && \text{(step 1)} \\
 &= [T(n/4) + b] + b = T(n/4) + 2b && \text{(step 2)} \\
 &= [T(n/8) + b] + 2b = T(n/8) + 3b && \text{(step 3)} \\
 &= [T(n/16) + b] + 3b = T(n/16) + 4b && \text{(step 4)} \\
 &\vdots
 \end{aligned}$$

where the general pattern for the recurrence relation at step i is:

$$T(n) = T(n/2^i) + ib. \quad (3.23)$$

The base case $T(1)$ is reached when $n/2^i = 1$. Thus, it occurs when $n = 2^i$, or equivalently, when $i = \log_2 n$. By substituting in (3.23) we obtain:

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

Since the order of growth is logarithmic, Listing 2.9 is faster than Listing 1.1, whose order is linear. This makes intuitive sense, since the former decomposes the original problem by dividing its size by two, while the latter is based on decrementing the size of the problem by a unit. Thus, Listing 2.9 needs fewer recursive function calls in order to reach the base case.

A subtle detail about recurrence relations that stem from dividing the size of a problem by an integer constant $k \geq 2$ is that they should not contain a single base case for $n = 0$. From a mathematical point of view, it would never be reached, and we would not be able to find a value for i in the method's third step. The pitfall is that the argument of $T(n)$ must be an integer. Thus, the fraction in $T(n/k)$ actually corresponds to an integer division. Notice that after reaching $T(1)$ the next expansion would correspond to $T(0)$ instead of $T(1/k)$. Therefore, for these recurrence relations we should include an additional base case, usually for $n = 1$, in order to apply the method correctly.

In the previous examples it was fairly straightforward to detect the general recursive pattern for the i -th stage of the expansion process. For the next recurrence relations the step is slightly more complex since it will involve computing sums like the ones presented in Section 3.1.4.

Consider the following recurrence relation:

$$T(n) = \begin{cases} a & \text{if } n = 0, \\ T(n-1) + bn + c & \text{if } n > 0. \end{cases} \quad (3.24)$$

The expansion process is:

$$\begin{aligned} T(n) &= T(n-1) + bn + c \\ &= [T(n-2) + b(n-1) + c] + bn + c \\ &= T(n-2) + 2bn - b + 2c \\ &= [T(n-3) + b(n-2) + c] + 2bn - b + 2c \\ &= T(n-3) + 3bn - b(1+2) + 3c \end{aligned}$$

$$\begin{aligned}
&= [T(n-4) + b(n-3) + c] + 3bn - b(1+2) + 3c \\
&= T(n-4) + 4bn - b(1+2+3) + 4c \\
&\vdots \\
&= T(n-i) + ibn - b(1+2+3+\cdots+(i-1)) + ic,
\end{aligned}$$

where the square brackets indicate particular expansions of terms involving T . In addition, the last step contains the general recursive pattern, which can be written as:

$$T(n) = T(n-i) + ibn - b \sum_{j=1}^{i-1} j + ic = T(n-i) + ibn - b \frac{(i-1)i}{2} + ic. \quad (3.25)$$

Two common misconceptions when using sums in the pattern at the i -th stage are: (1) using i as the index variable of the sum, and (2) choosing n as its upper limit. It is important to note that $i-1$ is the upper limit of the sum, which implies that the index of the sum cannot be i .

Finally, the base case $T(0)$ is reached when $i = n$. Therefore, substituting in (3.25) yields:

$$T(n) = bn^2 - \frac{b}{2}n(n-1) + cn + a = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right)n + a \in \Theta(n^2),$$

which is a quadratic polynomial.

The next recurrence relation appears in divide and conquer algorithms such as merge sort (see [Chapter 6](#)):

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

The expansion process is:

$$\begin{aligned}
T(n) &= 2T(n/2) + bn + c \\
&= 2[2T(n/4) + bn/2 + c] + bn + c \\
&= 4T(n/4) + 2bn + 2c + c \\
&= 4[2T(n/8) + bn/4 + c] + 2bn + 2c + c \\
&= 8T(n/8) + 3bn + 4c + 2c + c
\end{aligned}$$

$$\begin{aligned}
&= 8[2T(n/16) + bn/8 + c] + 3bn + 4c + 2c + c \\
&= 16T(n/16) + 4bn + 8c + 4c + 2c + c \\
&\vdots \\
&= 2^i T(n/2^i) + ibn + c(1 + 2 + 4 + \dots + 2^{i-1}).
\end{aligned}$$

Again, square brackets indicate expansions of terms involving T . In this case they are especially useful since each of the expanded terms needs to be multiplied by two. Students are advised to use them in general, since omitting them is a common source of mistakes.

In this case, the recursive pattern contains a partial sum of a geometric series.

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

Finally, we reach base case $T(1)$ when $n/2^i = 1$, which implies that $i = \log_2 n$. Therefore, substituting in (3.27) yields:

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

whose highest-order term is $bn \log_2 n$.

3.3.1.2 Master theorem

The master theorem is a result that can be used as a quick recipe for determining the computational time complexity of algorithms based on the divide and conquer design strategy. In particular, it can be applied to recurrence relations of the following form:

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

where $a \geq 1$, $b > 1$, $c \geq 0$, and f is an asymptotically positive function. Thus, these algorithms solve a subproblems whose size is equal to the original's divided by b . In addition, the further processing or combination of the subsolutions requires $f(n)$ operations. Depending on the relative contribution of the terms $aT(n/b)$ and $f(n)$ to the algorithm's runtime cost, the master theorem states, in its most general definition, that it is possible to determine an asymptotic tight bound for T in the following three cases:

1. If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then:

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

2. If $f(n) = \Theta(n^{\log_b a}(\log n)^k)$, with $k \geq 0$, then:

$$T(n) \in \Theta(n^{\log_b a}(\log n)^{k+1}).$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition ($af(n/b) \leq df(n)$ for some constant $d < 1$, and for all n sufficiently large), then:

$$T(n) \in \Theta(f(n)).$$

For example, for

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

it is possible to find an $\epsilon > 0$ such that the order of $f(n) = 2^n$ will be greater than that of $n^{\log_2 1 + \epsilon} = n^\epsilon$. Indeed, in this example any value $\epsilon > 0$ would be valid since the exponential order of 2^n is always greater than the order of a polynomial. Thus, this recurrence relation falls in the third case of the master theorem, which implies that $T(n) \in \Theta(2^n)$.

When $f(n)$ is a polynomial of degree k we can apply the following simpler version of the master theorem:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{if } \frac{a}{b^k} < 1 \\ \Theta(n^k \log n) & \text{if } \frac{a}{b^k} = 1 \\ \Theta(n^{\log_b a}) & \text{if } \frac{a}{b^k} > 1 \end{cases} \quad (3.28)$$

This result can be derived through the expansion method. Consider the following recurrence relation:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ aT(n/b) + dn^k & \text{if } n > 1, \end{cases} \quad (3.29)$$

where $a \geq 1$, $b > 1$, $c \geq 0$, and $d \geq 0$. The expansion process is:

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

The base case is $T(1) = c$, which is reached when $i = \log_b n$. Substituting yields:

$$T(n) = ca^{\log_b n} + dn^k \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^j = cn^{\log_b a} + dn^k \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^j,$$

where there are three cases depending on the values of a , b , and k :

1. If $a < b^k$ then $\sum_{j=0}^{\log_b n - 1} (a/b^k)^j$ will be a constant term (it cannot be infinity). Notice that the infinite sum $\sum_{i=0}^{\infty} r^i = 1/(1-r)$ is a constant (i.e., it does not diverge) for $r < 1$. Thus, in this scenario:

$$T(n) = cn^{\log_b a} + dKn^k,$$

for some constant K . In addition, since $a < b^k$ implies that $\log_b a < k$, the highest-order term is n^k . Therefore,

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

2. If $a = b^k$ then $T(n) = cn^k + dn^k \sum_{j=0}^{\log_b n - 1} 1 = cn^k + dn^k \log_b n$, which implies that

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

3. If $a > b^k$, then, solving the geometric series:

$$\begin{aligned} T(n) &= cn^{\log_b a} + dn^k \frac{\left(\frac{a}{b^k}\right)^{\log_b n} - 1}{\left(\frac{a}{b^k}\right) - 1} \\ &= cn^{\log_b a} + dn^k \frac{n^{\log_b a} - 1}{\frac{a}{b^k} - 1} \\ &= \left(c + \frac{d}{K}\right) n^{\log_b a} - \frac{d}{K} n^k, \end{aligned}$$

where $K = a/b^k - 1$ is a constant. Finally, since $a > b^k$ implies that $\log_b a > k$, we have:

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

3.3.1.3 Additional examples

A recurrence relation that captures the runtime of [Listings 1.2](#) and [2.4](#) could be:

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

where we have made the assumptions that multiplicative constants are equal to 1, and that the size of the subproblems is exactly half of the original problem's. The recurrence multiplies $T(n/2)$ times two since the algorithm needs to invoke itself twice in the recursive cases, with different arguments in each call. In addition, it adds the constant 1 since the results of the subproblems need to be added and returned by the method. Lastly, for simplicity, we do not need to specify a base case for $n = 2$. These assumptions will not affect the runtime's order of growth.

The recurrence relation can be solved through the expansion method as follows:

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

The base case $T(1) = 1$ occurs when $n = 2^i$. Substituting yields:

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

where $T(n)$ is a linear function of n . Naturally, this result is in accordance with the master theorem (see (3.28)), since the recurrence relation is a special case of (3.29), where $a = 2$, $b = 2$, and $k = 0$. Therefore, $T(n) \in \Theta(2^{\log_2 n}) = \Theta(n)$.

Consider the functions in Listing 1.5. The first two methods decompose the problem by reducing its size by a unit. Thus, the associated recurrence relation could be:

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

which is a special case of (3.21), where $T(n) \in \Theta(n)$. Instead, the third method decomposes the original problem in two of half the size, where the subsolutions do not need to be processed any further. Therefore, the corresponding recurrence relation can be identical to the one in (3.30).

Finally, we will study the computational time complexity of Listings 2.2 and 2.3. The associated recurrence relation (ignoring multiplicative constants) is:

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

It is different than the previous recurrences since the base case is defined on an interval. Nevertheless, by assuming that the input will be a power of 10, we can use an alternative definition of the recurrence:

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

Both functions are equivalent for $n = 10^p$ (and $p \geq 0$).

The second recurrence is a special case of (3.29), where $a = 1$, $b = 10$, and $k = 0$. Thus, the master theorem indicates that its complexity is logarithmic, since $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$.

We can obtain this result by the expansion method as follows:

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

The base case is obtained when $n/10^i = 1$, or in other words, when $i = \log_{10} n$. Substituting we obtain:

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

3.3.2 General method for solving difference equations

The expansion method is effective at solving recurrence relations in which the recursive function appears only once in the recursive definitions. This section describes a powerful approach, denoted in this book as the “general method for difference equations,” which allows us to solve recurrences that may “call” themselves several times. In particular, the method can be used to solve recurrences of the following form:

$$T(n) = \underbrace{-a_1T(n-1) - \dots - a_kT(n-k)}_{\text{“}T\text{ difference” terms}} + \underbrace{P_1^{d_1}(n)b_1^n + \dots + P_s^{d_s}(n)b_s^n}_{\text{polynomial} \times \text{exponential terms}}, \quad (3.33)$$

where a_i and b_i are constants, and $P_i^{d_i}(n)$ are polynomials (of n) of degree d_i . Terms involving T may appear several times on the right-hand side of the definition. Moreover, their arguments are necessarily n minus some integer constant. Thus, these recurrence relations are also known as “difference equations.” In this book we will call these terms involving T the “ T difference” terms, to emphasize that the arguments cannot take the form n/b , where b is a constant (if these terms appear it is necessary to transform the recurrence). In addition, the right-hand side of the definition may contain several terms that consist of a polynomial times a power of the input n (i.e., an exponential).

Instead of providing a general procedure directly, the following subsections will explain the method progressively, starting with simple recurrences, and introducing new elements as they grow in complexity.

3.3.2.1 Homogeneous recurrences and different roots

Homogeneous recurrence relations only contain T difference terms:

$$T(n) = -a_1T(n-1) - \dots - a_kT(n-k).$$

The first step for solving them consists of passing every term to the left-hand side of the definition:

$$T(n) + a_1T(n-1) + \dots + a_kT(n-k) = 0.$$

Subsequently, we define its associated **characteristic polynomial** by applying the change $x^{k-z} = T(n-z)$ for $z = 0, \dots, k$:

$$x^k + a_1x^{k-1} + \dots + a_{k-1}x + a_k.$$

All we have done is replace $T(n)$ with x^k , $T(n-1)$ with x^{k-1} , and so on. The coefficient (a_k) associated with the T difference term that has the smallest argument will be the polynomial's constant. The next step consists of finding the k roots of the characteristic polynomial. If r_i is its i -th root then it can be expressed in factorized form as:

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

If all of the roots are different then the nonrecursive expression for T will be:

$$T(n) = C_1r_1^n + \dots + C_kr_k^n. \quad (3.34)$$

Finally, the values of the constants will depend on the base cases of T , and will be found by solving a system of k linear equations with k unknown variables (the constants C_i). We will need to find k initial values (base cases, for small values of n) of T in order to construct the k equations.

The following example illustrates the approach applied to the basic Fibonacci function defined in (1.2), and coded in Listing 1.3. It can be rewritten as:

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

where we have included an extra base case for $n = 0$ that will be useful shortly. The first step consists of writing each T difference term on the left-hand side of the recursive identity:

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

It is a homogeneous recurrence since there is a 0 on the right-hand side. We then form the characteristic polynomial:

$$x^2 - x - 1.$$

The roots of this quadratic function are:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad \text{and} \quad r_2 = \frac{1 - \sqrt{5}}{2},$$

which are different. Therefore, we can express the recurrence nonrecursively as:

$$T(n) = C_1 r_1^n + C_2 r_2^n = C_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + C_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n. \quad (3.36)$$

The last step consists of finding values for the constants C_i by solving a system of linear equations. Each equation is formed by choosing a small value for n corresponding to a base case, and plugging it in (3.36). The simplest equation is obtained for $n = 0$, which is the reason why we considered the base case $n = 0$ in (3.35). For the second equation we can use $n = 1$. This provides the following system of linear equations:

$$\left. \begin{aligned} C_1 + C_2 &= 0 = T(0) \\ \left(\frac{1 + \sqrt{5}}{2} \right) C_1 + \left(\frac{1 - \sqrt{5}}{2} \right) C_2 &= 1 = T(1) \end{aligned} \right\}.$$

The solutions are:

$$C_1 = \frac{1}{\sqrt{5}}, \quad \text{and} \quad C_2 = -\frac{1}{\sqrt{5}}.$$

Therefore, the Fibonacci function can be expressed as:

$$T(n) = F(n) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n \in \Theta \left(\frac{1 + \sqrt{5}}{2} \right)^n, \quad (3.37)$$

which is an exponential function. Observe that r_1^n clearly dominates the growth of the function. On the one hand, $r_1 > r_2$. On the other hand, $|r_2| < 1$, which means that r_2^n will approach 0 as n approaches infinity. Finally, despite its complex appearance, the result is obviously an integer.

For the next example consider the mutually recursive functions defined in (1.17) and (1.18). They must be redefined exclusively in

terms of themselves in order to apply the method. On the one hand, $B(n) = A(n-1)$ implies that $B(n-1) = A(n-2)$. In addition, $A(2) = 1$. Thus, we can redefine $A(n)$ as:

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

On the other hand, since $A(n-1) = B(n)$, and $A(n) = B(n+1)$, we can substitute in (1.17) in order to obtain $B(n+1) = B(n) + B(n-1)$. Furthermore, since $B(2) = 0$, we can define $B(n)$ as:

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

Both of these functions have the form in (3.36). Thus, the only difference between them is the value of the constants. In particular:

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

and

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

where it is possible to assume that $A(0) = 1$, and $B(0) = -1$. Lastly, it is not hard to see that $A(n) + B(n)$ are Fibonacci numbers $F(n)$, since adding (3.40) and (3.41) yields (3.37).

3.3.2.2 Roots with multiplicity greater than one

The previous section showed that if the multiplicity of a root r_i is 1, corresponding to a term $(x-r_i)^1$ in the factorization of the characteristic polynomial, then the nonrecursive version of $T(n)$ will contain the term $C_i r_i^n$. In contrast, when the multiplicity m of a root r is greater than 1, resulting from $(x-r)^m$, then $T(n)$ will incorporate m terms of the form: constant \times polynomial $\times r^n$. In particular, the polynomials will be different powers of n , ranging from 1 to n^{m-1} . For example, a term

$(x - 2)^4$ in the factorization of the characteristic polynomial would lead to the four following terms in the nonrecursive version of $T(n)$:

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

for some constants C_i . Therefore, $T(n)$ can be expressed in general as:

$$T(n) = C_1 P_1(n) r_1^n + \dots + C_k P_k(n) r_k^n, \quad (3.42)$$

where $P_i(n)$ is a polynomial of the form n^c , for some c .

For example, consider the following recurrence:

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

with $T(0) = 0$, $T(1) = 2$, $T(2) = 11$, and $T(3) = 28$. It can be expressed as $T(n) - 5T(n - 1) + 9T(n - 2) - 7T(n - 3) + 2T(n - 4) = 0$, leading to following the characteristic polynomial:

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

It can be factorized (for example, by using Ruffini's rule) as:

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

which implies that the recurrence will have the following form:

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

where there are three terms associated with root $r = 1$. Finally, the constants can be recovered by solving the following system of linear equations:

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

The solutions are $C_1 = -1$, $C_2 = -2$, $C_3 = 3$, and $C_4 = 1$ (Listing 3.2 shows how to solve the system of linear equations, expressed as $\mathbf{Ax} = \mathbf{b}$, with the NumPy package). Finally,

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

whose order of growth is exponential.

Listing 3.2 Solving a system of linear equations, $\mathbf{Ax} = \mathbf{b}$, in Python.

```

1 import numpy as np
2
3 A = np.array([[1, 0, 0, 1], [1, 1, 1, 2],
4              [1, 2, 4, 4], [1, 3, 9, 8]])
5 b = np.array([0, 2, 11, 28])
6 x = np.linalg.solve(A, b)

```

3.3.2.3 Nonhomogeneous recurrences

A nonhomogeneous recurrence contains nonrecursive terms on its right-hand side. We will be able to solve these recurrences when such terms consist of a polynomial times an exponential, as shown in (3.33). The procedure is exactly the same, but for every term $P_i^{d_i}(n)b_i^n$ we will need to include $(x - b_i)^{d_i+1}$ in the characteristic polynomial, where d_i is the degree of the polynomial $P_i(n)$.

Consider the following recurrence relation:

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

As in the previous examples, the first step consists of moving the T difference terms to the left-hand side of the recurrence:

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

For the next step it is useful to express the terms on the right-hand side as the product of a polynomial times an exponential. Naturally, if a term only contains an exponential then it can be multiplied by 1, which is a polynomial. Similarly, we can multiply polynomials times 1^n . Therefore, the recurrence can be written as:

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

Furthermore, even though a particular exponential may appear several times, we must consider that it multiplies a single polynomial. Thus, the recurrence should be regarded as:

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

The next step consists of determining the characteristic polynomial. From the left-hand side we have $(x^2 - 2x + 1) = (x - 1)^2$. From the term $(1+n) \cdot 3^n$ we need to include $(x - 3)^2$, where the 3 is the base

of the exponential, and 2 is the degree of the polynomial (1) plus one. Similarly, $(3 + n + n^2) \cdot 1^n$ provides the new term $(x - 1)^3$. Therefore, the characteristic polynomial is;

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

and $T(n)$ has the following form:

$$T(n) = C_1 + C_2n + C_3n^2 + C_4n^3 + C_5n^4 + C_63^n + C_7n3^n.$$

The next example illustrates the approach with the following non-homogeneous recurrence relation:

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

In this case, we can write the recursive case as $T(n) - 2T(n - 1) = n \cdot 1^n + 1 \cdot 2^n$. The corresponding factorized characteristic polynomial is:

$$\underbrace{(x - 2)}_{\text{from } T(n) - 2T(n - 1)} \cdot \underbrace{(x - 1)^2}_{\text{from } n \cdot 1^n} \cdot \underbrace{(x - 2)}_{\text{from } 1 \cdot 2^n} = (x - 1)^2(x - 2)^2,$$

which implies that the recurrence has the following form:

$$T(n) = C_1 + C_2n + C_32^n + C_4n2^n.$$

Since there are four unknown constants we need the values of T evaluated at four different inputs. Starting with the base case $T(0) = 1$, we can compute $T(1)$, $T(2)$, and $T(3)$ by using $T(n) = 2T(n - 1) + n + 2^n$. In particular, $T(1) = 5$, $T(2) = 16$, and $T(3) = 43$. With this information we can build the following system of linear equations:

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

The solutions are $C_1 = -2$, $C_2 = -1$, $C_3 = 3$, and $C_4 = 1$. Therefore, the nonrecursive expression of $T(n)$ is:

$$T(n) = -2 - n + 3 \cdot 2^n + n2^n \in \Theta(n2^n).$$

3.3.2.4 Fractional inputs

The general method for difference equations can only be applied when the inputs of terms involving T are differences of the form $n - b$, where b is some integer constant. Nevertheless, it is possible to solve some recurrence relations that contain terms where the inputs to T are fractions of the form n/b , for some constant b . The key resides in transforming the fraction into a difference through the change of variable $n = b^k$, and therefore assuming that n is a power of b .

Consider the recurrence relation in (3.30), whose recursive case is:

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

and where we can assume that the input is a power of two. Since the argument in the T term of the right-hand side is $n/2$ we need to apply the change of variable $n = 2^k$ in order to obtain:

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

In this new definition $T(2^k)$ is a function of k . Therefore, we can substitute it with the term $t(k)$:

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

which we can solve through the general method for difference equations. In particular, the expression can be written as $t(k) - 2t(k-1) = 1 \cdot 1^n$. Therefore, the characteristic polynomial is $(x-2)(x-1)$, and the function will have the following form:

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

By undoing the change of variable we obtain a general solution to the recurrence in terms of the original variable:

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

The last step consists of determining the constants C_1 and C_2 . This can be done through either (3.43) or (3.44). For T we can use the base cases $T(1) = 1$ and $T(2) = 3$. The analogous base cases for t are: $t(0) = T(2^0) = T(1) = 1$, and $t(1) = T(2^1) = T(2) = 3$. Either way, the system of linear equations is:

$$\left. \begin{aligned} C_1 + C_2 &= 1 = T(1) = t(0) \\ C_1 + 2C_2 &= 3 = T(2) = t(1) \end{aligned} \right\}.$$

The solutions are $C_1 = -1$ and $C_2 = 2$, and the nonrecursive expression of $T(n)$ is:

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

In the next recurrence relation we can assume that the input is a power of four. In particular, let:

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

In this case applying the change of variable $n = 4^k$ leads to:

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

The next step consists of performing the substitution $t(k) = T(4^k)$:

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

We will call this operation a “change of function” in order to emphasize that only the T terms are replaced in the expression (by the t terms). Therefore, the change does not affect the term $2k$. The recurrence can be rewritten as $t(k) - 2t(k-1) = 2k \cdot 1^k$, where the associated characteristic polynomial is $(x-2)(x-1)^2$. Thus, the recurrence has the form:

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

In order to undo the change variable we can use $k = \log_4 n$, but we need an expression for 2^k . In particular, notice that $n = 4^k = (2^2)^k = (2^k)^2$. Therefore, $2^k = \sqrt{n}$, which leads to:

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

Finally, from the base case $T(1) = 1$ we can compute $T(4) = 2T(1) + \log_2 4 = 4$, and $T(16) = 2T(4) + \log_2 16 = 12$. This allows us to compute the constants C_i by solving the following system of linear equations:

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

The solutions are $C_1 = 5$, $C_2 = -4$, and $C_3 = -2$, and the nonrecursive expression of $T(n)$ is:

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

When the argument of T in the recursive expression is a square root, we can use the change of variable $n = 2^{(2^k)}$. Consider the following recurrence relation:

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

where $T(2) = 1$ and $n = 2^{(2^k)}$ (note that $2^{(2^k)} \neq 4^k$). This last restriction on n guarantees that the recursive procedure will terminate at the base case for $n = 2$. Applying the change of variable, we have:

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

By using the change of function $t(k) = T(2^{(2^k)})$, the recurrence is $t(k) = t(k-1) + 2^k$, whose characteristic polynomial is $(x-2)(x-2)$. Therefore, the new recurrence will have the form:

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

In order to undo the change of variable we can use $k = \log_2(\log_2 n)$, and $2^k = \log_2 n$. The recurrence as a function of n is therefore:

$$T(n) = C_1 \log_2 n + C_2 (\log_2(\log_2 n)) \log_2 n. \quad (3.46)$$

Finally, we can use the base cases $T(2) = 1$ and $T(4) = 4$ in order to find the constants. In particular, we need to solve the following system of linear equations:

$$\left. \begin{array}{l} C_1 = 1 = T(2) \\ 2C_1 + 2C_2 = 4 = T(4) \end{array} \right\}.$$

The solutions are $C_1 = C_2 = 1$. Therefore, the final nonrecursive formula for $T(n)$ is:

$$T(n) = \log_2 n + (\log_2(\log_2 n)) \log_2 n \in \Theta((\log(\log n)) \log n). \quad (3.47)$$

3.3.2.5 Multiple changes of variable or function

The previous recurrence in (3.45) can also be solved by applying two consecutive changes of variable (and function). Firstly, the change $n = 2^k$ leads to:

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

Replacing $T(2^k)$ with $t(k)$, we obtain the following recurrence:

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

which still cannot be solved through the method since it is not a difference equation. However, we can apply a new change of variable in order to transform it into one. With the change $k = 2^m$ we obtain:

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

which can be written as a difference equation by performing the change of function $u(m) = t(2^m)$:

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

Its characteristic polynomial is $(x-2)^2$, which implies that the recurrence has the following form:

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

Undoing the changes of variable leads to:

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

and finally:

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

which is identical to (3.46). Thus, the solution is provided in (3.47).

The strategy of using several changes of variables or functions can be used to solve more complex recurrences, such as the following nonlinear one:

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

where n is a power of two. Firstly, we can apply the change of variable $n = 2^k$, which leads to:

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

In addition, with the change of function $t(k) = T(2^k)$ we obtain:

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

which we still cannot solve. However, we can take logarithms on both sides of the definition:

$$\log_2 t(k) = \log_2 \left(2^k [t(k-1)]^2 \right) = k + 2 \log_2 t(k-1),$$

and apply the more complex change of function $u(k) = \log_2 t(k)$, which leads to:

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

and that we can solve through the method. In particular, its characteristic polynomial is $(x-2)(x-1)^2$, which implies that $u(k)$ has the following form:

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

Undoing the changes, we first have:

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

and finally:

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

The last step consists of determining the constants. Using the initial recurrence with $T(1) = 1/3$, we obtain $T(2) = 2[T(1)]^2 = 2/9$, and $T(4) = 4[T(2)]^2 = 4(2/9)^2 = 16/81$. We can use these values to build the following system of (nonlinear) equations:

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

which can be transformed into a system of linear equations by taking logarithms on both sides of the equations:

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

The solutions are: $C_1 = 2 - \log_2 3 = \log_2(4/3)$, $C_2 = -2$, and $C_3 = -1$. Therefore, $T(n)$ can be expressed as:

$$\begin{aligned} T(n) &= 2^{(\log_2(4/3))n - 2 - \log_2 n} = [2^{\log_2(4/3)}]^n \cdot 2^{-2} \cdot 2^{\log_2(1/n)} \\ &= \left[\frac{4}{3} \right]^n \cdot \frac{1}{4n} \in \Theta \left(\left[\frac{4}{3} \right]^n \cdot \frac{1}{n} \right). \end{aligned}$$

3.4 EXERCISES

Exercise 3.1 — Prove the following identity:

$$\left(\frac{a}{b^k}\right)^{\log_b n} = \frac{n^{\log_b a}}{n^k}$$

Exercise 3.2 — By using limits, show that $\log n! \in \Theta(n \log n)$. Hint: when n approaches infinity, $n!$ can be substituted by “Stirling’s approximation”:

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Exercise 3.3 — Show that $n \log n \in \mathcal{O}(n^{1+a})$, where $a > 0$. Use limits, and L’Hopital’s rule.

Exercise 3.4 — Let m and n be some integers. Determine:

$$\sum_{i=m}^n 1.$$

Exercise 3.5 — Write the sum of the first n odd integers in sigma notation, and simplify it (the result should be a well-known polynomial).

Exercise 3.6 — Use the following identity:

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

to provide an expression for the sum of the first n squared integers ($1^2 + 2^2 + \dots + n^2$).

Exercise 3.7 — Show that a general partial sum of n terms of an arithmetic sequence ($s_i = s_{i-1} + d$, for some initial s_0) follows:

$$\sum_{i=0}^{n-1} s_i = \frac{n}{2}(s_0 + s_{n-1}).$$

Exercise 3.8 — An algorithm processes some of the bits of the binary representations of numbers from 1 to $2^n - 1$, where n is the number of bits of each number. In particular, the algorithm processes the least significant bits of each number (from right to left), until it finds a bit set to 1. Given n , determine, using sums, the total number of bits that the algorithm processes. For example, for $n = 4$ the algorithm processes the 26 shaded bits in [Figure 3.11](#).

0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

←

Figure 3.11 An algorithm processes the shaded bits of numbers from 1 to $2^n - 1$ (for $n = 4$).

Exercise 3.9 — Specify recurrence relations that describe the runtime of algorithms that implement functions (1.14), (1.15), (1.16), and (1.19).

Exercise 3.10 — Solve the following recurrences, but without computing multiplicative constants (C_i). Thus, in this exercise the base cases are not needed.

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

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

c) $T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3) + 3 + n^2 + n2^n$

Exercise 3.11 — Define a recurrence relation for the runtime of Listing 2.6, then solve it through the expansion method and the general method for difference equations. Finally, determine its order of growth.

Exercise 3.12 — Solve the following recurrence relation through the expansion method:

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

Exercise 3.13 — Solve the following recurrence relation through the expansion method:

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

Exercise 3.14 — Solve the recurrence relation in (3.18) through the general method for difference equations.

Exercise 3.15 — Define a recurrence relation for the runtime of Listing 2.9, solve it through the master theorem, the expansion method, and the general method for difference equations. Finally, determine its order of growth.

Exercise 3.16 — Solve the following recurrence relation through the master theorem, the expansion method, and the general method for difference equations. Finally, determine its order of growth.

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

where n is a power of 2.

Exercise 3.17 — Solve the recurrence relation in (3.32) through the general method for difference equations.

Exercise 3.18 — Solve the following recurrence relations:

- a) $T(n) = 2T(n-1) + 3n - 2$, with $T(0) = 0$.
- b) $T(n) = T(n/2) + n$, with $T(1) = 1$, and n is a power of 2.
- c) $T(n) = T(n/\alpha) + n$, with $T(1) = 0$, and where $\alpha \geq 2$ is an integer.
- d) $T(n) = T(n/3) + n^2$, with $T(1) = 1$, and n is a power of 3.
- e) $T(n) = 3T(n/3) + n^2$, with $T(1) = 0$, and n is a power of 3.
- f) $T(n) = 2T(n/4) + n$, with $T(1) = 1$, and n is a power of 4.
- g) $T(n) = T(n/2) + \log_2 n$, with $T(1) = 1$, and n is a power of 2.
- h) $T(n) = 4T(n/2) + n$, with $T(1) = 1$, and n is a power of 2.

104 ■ Introduction to Recursive Programming

- i) $T(n) = 2T(n/2) + n \log_2 n$, with $T(1) = 1$, and n is a power of 2.
- j) $T(n) = \frac{3}{2}T(n/2) - \frac{1}{2}T(n/4) - \frac{1}{n}$, with $T(1) = 1$, $T(2) = 3/2$, and n is a power of 2.

Linear Recursion I: Basic Algorithms

Do the difficult things while they are easy and do the great things while they are small. A journey of a thousand miles must begin with a single step.

— Lao Tzu

RECURSIVE methods can be categorized according to the number of times they invoke themselves in the recursive cases. This chapter examines methods that not only call themselves just once, but also process the output of the recursive call before producing or returning their own result. This common type of recursion is known as **linear recursion**, where the single recursive call is not the last operation carried out by the method. This chapter will present numerous problems and respective linear-recursive solutions that we will design by relying on the concepts and methodology introduced in [Chapters 1](#) and [2](#). Finally, if the single recursive call is the method's last action, we say we are using **tail recursion**. This special type of recursion will be covered in [Chapter 5](#).

Linear recursion is the simplest type of recursion, and can be viewed as an alternative way to implement iterative loops that carry out a certain number of operations repeatedly. In this regard, although it is fairly straightforward to solve the problems discussed in this chapter iteratively, linear recursion provides the clearest examples of how to think and program recursively. The chapter will therefore present examples of how to apply recursive concepts (i.e., problem decomposition and induction), which the reader should comprehend before tackling more complex problems and types of recursion.

4.1 ARITHMETIC OPERATIONS

This section presents recursive solutions to several elemental arithmetic computations. We will examine them for illustration purposes, since most correspond to simple operations that can be implemented through basic commands or expressions.

4.1.1 Power function

A classical problem used to illustrate recursion is the power function. The goal is to calculate b to the power of n :

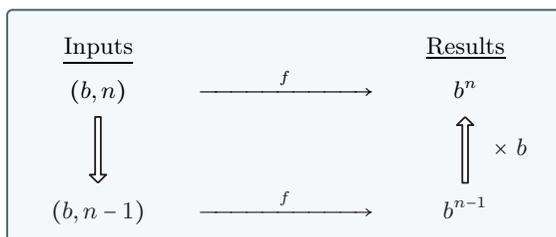
$$b^n = \prod_{i=1}^n b = \underbrace{b \times b \times \dots \times b}_{n \text{ times}},$$

where the base b is a real number, and the exponent n is a nonnegative integer (in Python, powers can be obtained through the `**` operator.) The following subsections examine algorithms that can compute the power in linear and logarithmic time.

4.1.1.1 Powers in linear time

According to the methodology presented in [Figure 2.1](#), the first step consists of determining the size of the problem, which is related to the input parameters. For this problem, only the exponent is relevant for determining the number of operations that algorithms need to solve it. Clearly, the runtime of the algorithms increases as n grows, since the result would require more multiplications. Thus, the exponent n is the size of the problem. The base case corresponds to trivial instances of the problem, with minimum size. Since n is a nonnegative integer, the smallest instance is b^0 , which is equal to one.

In the decomposition step we have to consider smaller and simpler self-similar subproblems that stem from reducing the size of the problem. In this section we will simply decrement it by a single unit, which leads to the following diagram:



Listing 4.1 Power function in linear time for nonnegative exponents.

```

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

```

where $f(b, n) = b^n$ is naturally a function of the two parameters b and n . It is apparent that if we assume to know the solution to $f(b, n-1) = b^{n-1}$, then all we need to do in the recursive case is multiply it times b in order to obtain b^n . Together with the base case, the function can be implemented as shown in [Listing 4.1](#).

It is an example of linear recursion since there is only one function call in the recursive case, and the recursive call is not the last operation that the algorithm performs in the recursive case. Even though the function call is the last element in the expression (in line 5), the method first evaluates it, and then it processes it, multiplying the result times b . Thus, the function is not tail-recursive. It is also important to note that the function does not need a base case when $n = 1$, since it would be redundant. Finally, the computational complexity of this algorithm is linear with respect to n . In particular, the runtime can be defined through:

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

where $T(n) = n + 1 \in \Theta(n)$.

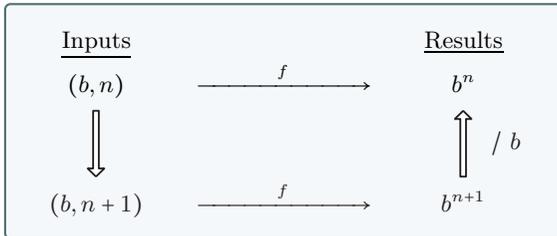
If the problem statement allowed negative integer exponents as well, the size of the problem would be the absolute value of n . In this variant the base case also occurs when $n = 0$, and the recursive case is identical if $n > 0$. When n is negative the decomposition needs to reduce the size of the problem, ensuring that the corresponding subproblem gets closer to the base case. Thus, for this recursive case we need to increment n . The diagram that illustrates the thought process would be:

Listing 4.2 Power function in linear time.

```

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

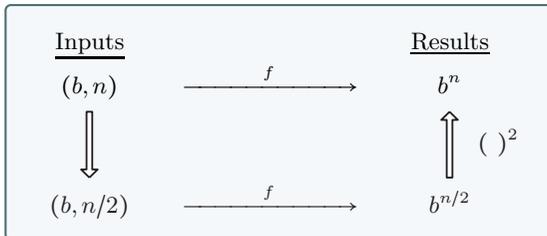
```



Thus, the result of the subproblem must be divided by b in order to obtain the desired power b^n . The recursive case is therefore $f(b, n) = f(b, n + 1)/b$, as shown in Listing 4.2.

4.1.1.2 Powers in logarithmic time

It is possible to design more efficient algorithms that calculate powers, which run in logarithmic time, by considering a subproblem of half the size in the decomposition stage. Assuming that n is a nonnegative integer, we need two diagrams in order to account for the parity of n . If n is even, the recursive design thought process can be visualized as follows:



Thus, the result of the subproblem must be squared in order to obtain b^n . Instead, if n is odd, we can consider using a subproblem of size $(n - 1)/2$ (which is an integer). The diagram would be:

Listing 4.3 Power function in logarithmic time for nonnegative exponents.

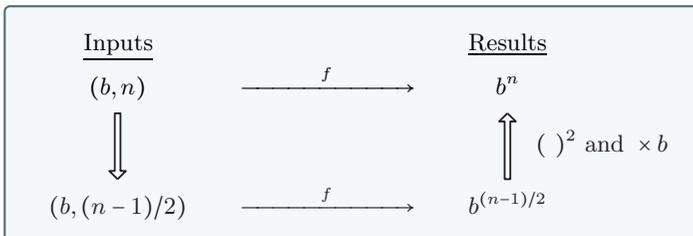
```

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

Listing 4.4 Inefficient implementation of the power function that runs in linear time.

```

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



In this case, the output of the subproblem also needs to be squared, but a final multiplication times the base b is also required. With these recursive cases, and the trivial base case, the recursive function is:

$$b^n = \begin{cases} 1 & \text{if } n = 0, \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even,} \\ (b^{(n-1)/2})^2 \cdot b & \text{if } n > 0 \text{ and } n \text{ is odd.} \end{cases}$$

A proper implementation of the function is shown in [Listing 4.3](#), which again uses linear recursion. The use of the integer division (`//`) instead of a real division `/` is optional, but included in order to emphasize that the second input parameter is an integer. In addition, when n is

odd $(n-1)//2$ is equivalent to $n//2$. The code uses the former expression since it resembles the mathematical definition of the function more closely.

Its runtime can be defined through:

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

where $T(n) = 2 + \log_2 n$ for $n > 0$. Thus, $T(n) \in \Theta(\log n)$. The superior performance stems from dividing the size of the problem by two in the decomposition stage. However, the function must make a single recursive call in each recursive case. For example, the code in [Listing 4.4](#) does not run in logarithmic time even though the decomposition divides the problem size by two. The issue is that it calculates the result of the same subproblem twice by using two identical recursive calls, which is obviously unnecessary. The runtime cost for this function is:

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

where $T(n) = n + 1 \in \Theta(n)$. Notice that it is analogous to the runtime of the function in [Listing 4.1](#), whose runtime is described through (4.1). Thus, the extra performance that could have been gained by halving the size of the input is lost by making two identical recursive calls in the recursive cases.

4.1.2 Slow addition

This problem consists of adding two nonnegative integers a and b , where we cannot use the general arithmetic commands such as addition, subtraction, multiplication, or division. Instead, we are only allowed to add or subtract single units from the numbers (a constant amount of times). This simple problem will help illustrate how the choice of the size can affect the way we decompose a problem, which leads to different recursive algorithms.

In consonance with the template in [Figure 2.1](#), the first step consists of determining the size of the problem. Since we are only allowed to add or subtract 1 from a and b , it is reasonable to think that we will have to perform these simple operations $a + b$ times until a base case condition is met (we will see shortly that it can be solved in only a or b operations). If we choose $a + b$ to be the size of the problem, then a base case occurs

Listing 4.5 Slow addition of two nonnegative integers.

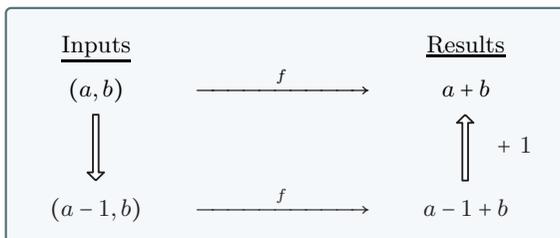
```

1 def slow_addition(a, b):
2     if a == 0:
3         return b
4     elif b == 0:
5         return a
6     else:
7         return slow_addition(a - 1, b) + 1

```

when $a = b = 0$, which corresponds to the instance of smallest size. In addition, we have to consider other possible base cases where we can provide a result easily, without the need to carry out a recursive call. For this problem, when $a = 0$ the result is obviously b , and when $b = 0$ the output should be a . Furthermore, with these two base cases it is not necessary to include the one for $a = 0$ and $b = 0$. Lastly, the base cases guarantee that both a and b will be positive in the recursive case.

We could proceed in the decomposition step by reducing the size of the problem by a unit. Since our choice for the size of the problem is $a + b$, we could subtract a unit from either a or b . The recursive diagram related to decrementing a would be:



If the function we are implementing is f , then clearly $f(a, b) = f(a - 1, b) + 1$ defines the recursive case, and it can be expressed mathematically as:

$$f(a, b) = \begin{cases} b & \text{if } a = 0, \\ a & \text{if } b = 0, \\ f(a - 1, b) + 1 & \text{if } a > 0, b > 0. \end{cases} \quad (4.2)$$

Listing 4.5 shows an implementation of the function in Python. Had we chosen a to be the size of the problem the two base cases would also appear in the definition of the function, and we would have also decremented the value of a when decomposing the problem. Therefore,

Listing 4.6 Quicker slow addition of two nonnegative integers.

```

1 def quicker_slow_addition(a, b):
2     if a == 0:
3         return b
4     elif b == 0:
5         return a
6     elif a < b:
7         return quicker_slow_addition(a - 1, b) + 1
8     else:
9         return quicker_slow_addition(a, b - 1) + 1

```

the resulting function would be identical. Alternatively, choosing b as the size of the problem would imply replacing the recursive case by $f(a, b) = f(a, b - 1) + 1$, since we would have decremented b instead of a .

The function in [Listing 4.5](#) can be slow if a is large. Alternatively, we can build a more efficient algorithm by considering that the size of the problem is the smallest input parameter, i.e., $\min(a, b)$. In this scenario the two base cases $f(0, b) = b$, and $f(a, 0) = a$, correspond to the smallest instances of the problem ($f(0, 0)$ would be redundant). For the recursive cases we must find a decomposition that guarantees reducing the size of the problem. A first approach consists of decrementing the smallest input parameter, where the size of the subproblem would be $\min(a, b) - 1$. For example, if $a < b$ the recursive rule would be $f(a, b) = f(a - 1, b) + 1$, while if $a \geq b$ we would apply $f(a, b) = f(a, b - 1) + 1$. Thus, the mathematical function would be:

$$f(a, b) = \begin{cases} b & \text{if } a = 0, \\ a & \text{if } b = 0 \text{ (and } a \neq 0), \\ f(a - 1, b) + 1 & \text{if } a < b \text{ (and } a \neq 0, b \neq 0), \\ f(a, b - 1) + 1 & \text{if } b \leq a \text{ (and } a \neq 0, b \neq 0), \end{cases}$$

which can be coded as shown in [Listing 4.6](#).

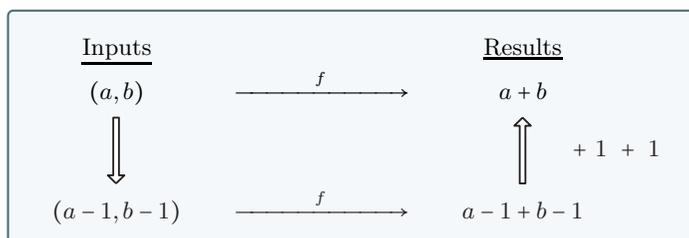
A second idea consists in decrementing both parameters, which also ensures reducing the size of the problem. In particular, note that $\min(a - 1, b - 1) = \min(a, b) - 1$. For this decomposition the recursive diagram would be:

Listing 4.7 Alternative quicker slow addition of two nonnegative integers.

```

1 def quicker_slow_addition_alt(a, b):
2     if a == 0:
3         return b
4     elif b == 0:
5         return a
6     else:
7         return quicker_slow_addition_alt(a - 1, b - 1) + 1 + 1

```



The recursive rule is therefore: $f(a, b) = f(a - 1, b - 1) + 1 + 1$, for $a > 0$ and $b > 0$. According to the problem statement, this is valid if we assume that we can perform increments a *constant* number of times (in this case, $f(a - 1, b - 1)$ is incremented twice), which does not depend on the inputs. Listing 4.7 shows the associated code.

4.1.3 Double sum

The next example involves calculating the particular double sum:

$$f(m, n) = \sum_{i=1}^m \sum_{j=1}^n (i + j). \quad (4.3)$$

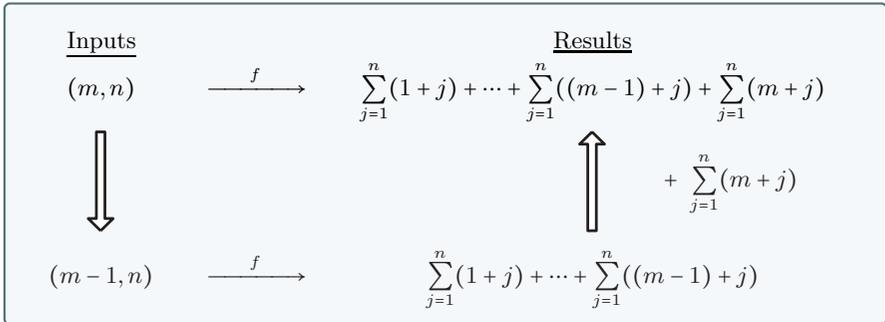
By using the properties of sums it can be simplified to a simple expression:

$$\begin{aligned}
 f(m, n) &= \sum_{i=1}^m \sum_{j=1}^n (i + j) = \sum_{i=1}^m \left(in + \frac{n(n+1)}{2} \right) = \frac{nm(m+1)}{2} + \frac{mn(n+1)}{2} \\
 &= \frac{mn(m+n+2)}{2}.
 \end{aligned}$$

However, we will develop a recursive solution that uses two recursive functions (one for each sum).

Firstly, the outer sum is a function of the parameters m and n , whose size is m . The function returns 0 in its base case when m is smaller than

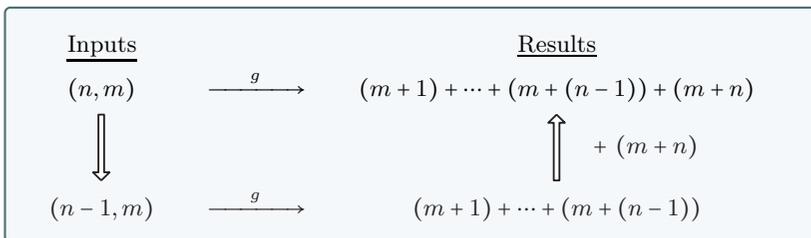
the lower index (i.e., when $m \leq 0$). The general diagram for the recursive case, assuming that the size of the subproblem is $m - 1$, is:



Clearly, all we have to do to the result of $f(m - 1, n)$, in order to obtain $f(m, n)$, is add the inner sum $\sum_{j=1}^n (m + j)$, which is a function g that depends on n and m . Therefore, we can define f as follows:

$$f(m, n) = \begin{cases} 0 & \text{if } m \leq 0, \\ f(m - 1, n) + g(n, m) & \text{if } m > 0. \end{cases}$$

We can also define $g(n, m)$ recursively. Firstly, we need to determine its size, which is n , since it adds n terms. In accordance with other sums, it returns 0 in the base case, which occurs when $n \leq 0$. The general diagram for its recursive case, again decrementing the size of the problem by a unit, is:



In this case, we need to add $(m + n)$ to the result of the subproblem in order to obtain $g(n, m)$. Thus, the function is:

$$g(n, m) = \begin{cases} 0 & \text{if } n \leq 0, \\ g(n - 1, m) + (m + n) & \text{if } n > 0. \end{cases}$$

Finally, [Listing 4.8](#) shows an implementation of both sums.

Listing 4.8 Recursive functions that compute the double sum in (4.3).

```

1 def inner_sum(n, m):
2     if n <= 0:
3         return 0
4     else:
5         return inner_sum(n - 1, m) + (m + n)
6
7
8 def outer_sum(m, n):
9     if m <= 0:
10        return 0
11    else:
12        return outer_sum(m - 1, n) + inner_sum(n, m)

```

4.2 BASE CONVERSION

Numbers can be represented in different ways, depending on a particular **base** or radix. Typically, we use a base-10 numeral system where a sequence of digits, say 142, represents the number $1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$. In general, for a particular base b , the value of a number x can be expressed as a unique sequence of m digits $d_{m-1} \cdots d_0$, where:

$$x = \sum_{i=0}^{m-1} d_i b^i, \quad (4.4)$$

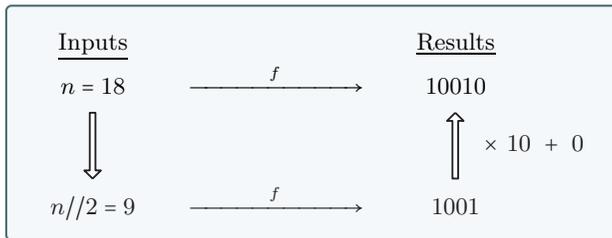
with $0 \leq d_i < b$, and $d_{m-1} \neq 0$ (i.e., we omit writing leading zeros). Therefore, different bases lead to distinct sequences of digits that represent the same number. Regarding notation, the base can be specified through a subscript, which we usually omit when it is 10. For example, $142_{10} = 142$, but $142_5 = 1 \cdot 5^2 + 4 \cdot 5^1 + 2 \cdot 5^0 = 25 + 20 + 2 = 47$. In this section we will examine algorithms for converting numbers expressed in some base to another one.

4.2.1 Binary representation of a nonnegative integer

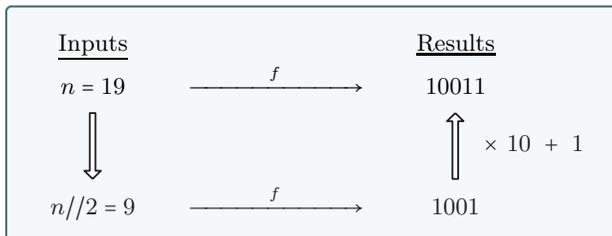
In this example the goal consists of developing a recursive function that, given a particular nonnegative integer n (in base 10), returns a new integer that contains the binary representation of n , which is the sequence of base-2 digits (i.e., bits) associated with the value of n . The output will also be a number in base 10, but its digits will only be either zero or one. Thus, we can interpret that the digits actually correspond to bits.

Since we need to create the sequence of “bits,” the size of the problem is the number of bits in the binary representation of n . Mathematically, this quantity is $\lfloor \log_2 n \rfloor + 1$ (for $n > 0$). However, we do not need the formula in order to design the recursive algorithm. All we require is a clear definition of the size of the problem that will enable us to define base cases and decompose the problem. In particular, the smallest instances of the problem correspond to numbers that contain a single bit, which are 0 and 1. Thus, the base case occurs when $n < 2$, where the output is simply n .

For the recursive case we need to decide how we can reduce the size of the problem. The simplest way consists of decrementing the number of bits by a single unit, which is accomplished by performing an integer division of n by two (this shifts the bits one place to the right, where the least significant bit is discarded). We can start the analysis with concrete instances. For example:



and



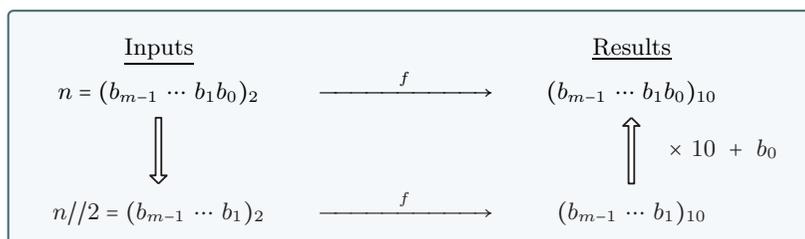
Note that the output of $f(18)$ is ten thousand and ten, since it is expressed in base 10. The diagrams illustrate that we can obtain the desired result by multiplying the output of the subproblem by 10, and then adding 1 if n is odd. Therefore, the recursive case seems to be $f(n) = 10f(n//2) + n\%2$.

We can proceed more rigorously through the following general diagram of the recursive thought process:

Listing 4.9 Binary representation of a nonnegative integer.

```

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



where b_i represents a bit. In particular, $f(n//2)$ is the decimal number that represents the sequence of “bits” of n except for the least significant one (b_0), whose value is $n\%2$ (or $n\&1$, where $\&$ denotes the bitwise AND operator). In order to append the discarded bit we need to multiply $f(n//2)$ by 10 (which shifts the digits one place to the left, and appends a zero), and add it to the result. Therefore, the recursive case is indeed $f(n) = 10f(n//2) + n\%2$, which we obtained by analyzing concrete instances. The complete function is:

$$f(n) = \begin{cases} n & \text{if } n < 2, \\ 10f(n//2) + n\%2 & \text{if } n \geq 2. \end{cases}$$

Listing 4.9 shows the corresponding code. Finally, the binary representation of $142 = 142_{10}$ is $10001110_2 = 128 + 8 + 4 + 2$.

4.2.2 Decimal to base b conversion

The problem in Section 4.2.1 is a particular case of the more general problem that consists of converting a decimal number into another in another base b , where $b \geq 2$. It can be tackled in a similar way by using diagrams. Figure 4.1 shows the steps of a general algorithm that solves the problem. In particular, it illustrates how to obtain the base-5 representation of 142, which is 1032 (i.e., $142_{10} = 1032_5$). The full problem is shown in (a), where the idea consists of performing successive integer divisions of n by b until reaching 0. At each step the remainder of the

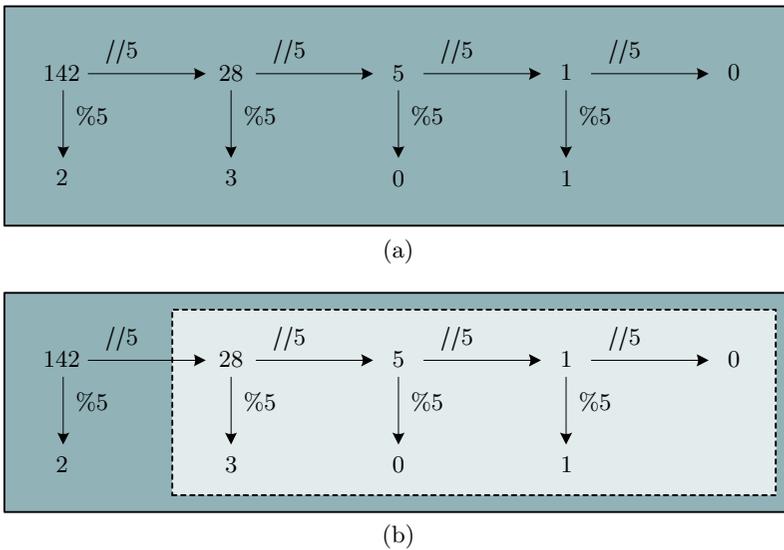


Figure 4.1 Conversion of 142 into its representation (1032) in base 5.

Listing 4.10 Conversion of a nonnegative integer n into its representation in base b .

```

1 def decimal_to_base(n, b):
2     if n < b:
3         return n
4     else:
5         return 10 * decimal_to_base(n // b, b) + (n % b)

```

division by b is also calculated, and will constitute a digit in the new representation in base b , whose position is controlled by a variable associated with a power of 10. The diagram in (b) highlights a subproblem within the original, where $28 = 103_5$.

In order to develop a recursive solution we need to establish the size of the problem. For this problem it is the number of times it is necessary to divide n until reaching 0, which is essentially the number of digits of the input in base b . The base case occurs when n can be represented by a single digit in base b . In other words, when $n < b$, where the result is simply n .

The recursive case requires identifying a subproblem within the original, as shown in Figure 4.1(b). Note that the subproblem is self-similar

to the original, but starts at 28, which is $142//5$. Thus, the decomposition consists of performing the integer division $n//b$, which reduces the size of the problem by a unit. Subsequently, we must determine how it is possible to modify the solution to the subproblem (103), in order to obtain the original (1032). The solution consists of multiplying the result of the subproblem times 10, and adding $n\%b$ (which is 2 in the example). Thus, the function can be coded as shown in [Listing 4.10](#).

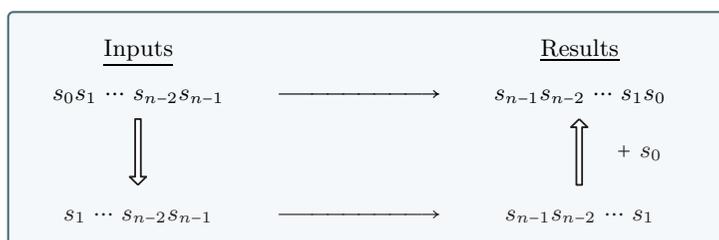
4.3 STRINGS

This section analyzes two problems involving strings, which are essentially sequences of characters, and constitute a fundamental data type in many programming languages.

4.3.1 Reversing a string

Consider the problem of reversing a string. In particular, we will develop a function f that receives an input string and returns its reverse. For example, $f(\text{'abcd'}) = \text{'dcba'}$.

The size of the problem is the length of the input string. A base case occurs when the input string is empty, where the function obviously returns an empty string. In the recursive case we need to discard a character from the input string in order to reduce the size of the problem. The first and last characters of the string are clear candidates. Omitting the first one leads to the following diagram where the input string s is written as the sequence $s_0s_1 \cdots s_{n-2}s_{n-1}$ of characters:



Thus, the function simply has to concatenate the first character to the result of the subproblem associated with $s_1 \cdots s_{n-2}s_{n-1}$ (the + symbol represents string concatenation). Together with the base case, the recursive function in Python is shown in [Listing 4.11](#).

Listing 4.11 Conversion of a nonnegative integer n into its representation in base b .

```

1 def reverse_string(s):
2     if s == '':
3         return ''
4     else:
5         return reverse_string(s[1:]) + s[0]
```

4.3.2 Is a string a palindrome?

The next problem consists of determining whether a string is a palindrome, which is a sequence of characters that reads the same forward as it does backward (e.g., “radar”). Its size is the length of the string, since it determines the number of operations needed in order to obtain a true result. There are two base cases for this problem: (a) when the string is empty, and (b) when it comprises a single character. In both situations the output is `True`. Similarly to the function in Listing 2.6, the second base case is necessary since we will decompose the problem by reducing its size by two units. In particular, we will consider the subproblem that discards the first and last characters of the original string of length $n \geq 2$. In addition, the output of the function for some string $\mathbf{s} = s_0s_1 \cdots s_{n-2}s_{n-1}$ of length n is:

$$(s_0 = s_{n-1}) \wedge (s_1 = s_{n-2}) \wedge \cdots \wedge (s_{\lfloor n/2 \rfloor - 1} = s_{n - \lfloor n/2 \rfloor}),$$

where \wedge denotes logical AND (conjunction), and s_i represents the character from \mathbf{s} at position i (where the first character of the string is located at position 0). Observe that the indices of the characters that need to be compared add up to $n - 1$. Furthermore, we only need to compare $\lfloor n/2 \rfloor$ pairs of characters. The expression can be written more compactly as:

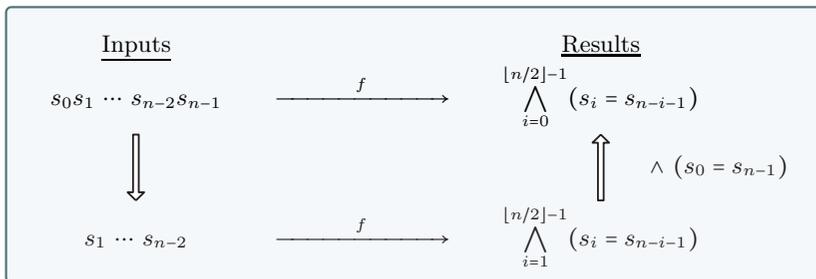
$$\bigwedge_{i=0}^{\lfloor n/2 \rfloor - 1} (s_i = s_{n-i-1})$$

similarly to “sigma” notation. The corresponding recursive diagram is:

Listing 4.12 Function that determines if a string is a palindrome.

```

1 def is_palindrome(s):
2     n = len(s)
3     if n <= 1:
4         return True
5     else:
6         return (s[0] == s[n - 1]) and is_palindrome(s[1:n - 1])
    
```



Thus, the Boolean function can be defined as:

$$f(\mathbf{s}) = \begin{cases} \text{true} & \text{if } n < 2, \\ (s_0 = s_{n-1}) \wedge f(\mathbf{s}_{1..n-2}) & \text{if } n \geq 2, \end{cases}$$

where $\mathbf{s}_{1..n-2}$ simply denotes the substring $s_1 \cdots s_{n-2}$. Finally, Listing 4.12 shows the corresponding code, whose time complexity is $\Theta(n)$.

4.4 ADDITIONAL PROBLEMS

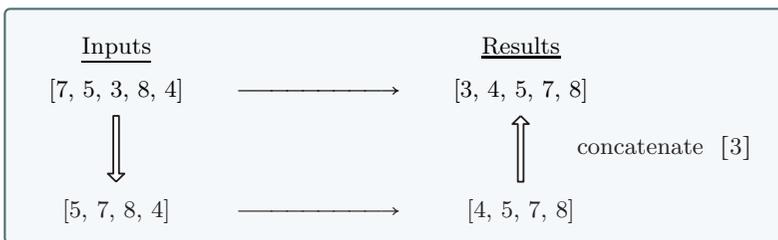
This section contains several classical problems that can be solved elegantly through recursion.

4.4.1 Selection sort

The selection sort algorithm is one of the simplest strategies for sorting the elements of a list. Given an input list \mathbf{a} of n numbers, assume the method must sort it in ascending order. The algorithm begins by placing the smallest element of the list in the first position of the output list. In the second step, the method searches for the minimum element in $\mathbf{a}_{1..n-1}$, and places it in the second position of the output list. This procedure is carried out $n - 1$ times until the array is finally sorted. Clearly, at the end of the k -th step the smallest k numbers are sorted correctly, and after $n - 1$ the entire list is completely sorted.

We will now show a linear-recursive version of the algorithm. Firstly, the size of the problem is the length of the list (n). The base case occurs when the list contains one element, where the output is the input list since it is obviously sorted. In addition, the algorithm can also return an empty list if the input is empty. Thus, the method can return **a** if $n \leq 1$.

For the recursive case we can decompose the problem by reducing its size by a unit, where the main idea is to discard the smallest element from the list. This can be done in two ways. Firstly, the element can be swapped with the first one on the list. This naturally does not change the result (the sorted list), but allows us to discard the first element of the list after having swapped the elements. Therefore, if the input list is $[7, 5, 3, 8, 4]$, it can be modified to $[3, 5, 7, 8, 4]$ by swapping the 3 and the 7. In that case, the input to the subproblem of size $n - 1$ would be $[5, 7, 8, 4]$, and a recursive diagram with this particular example could be:



In order to solve the original problem, the recursive rule must concatenate the discarded smallest element ($[3]$) with the output list of the subproblem (note that the results are sorted lists). [Listing 4.13](#) shows an implementation of the method that uses the function `min` to determine the smallest value in a list, the method `index` that returns the location of an element in a list, and the `+` operator to concatenate lists. An important detail regarding this algorithm is that it needs to make a new copy of the input list (in line 5) in order to not alter it when calling the method. In particular, without this copy the method would swap the first and smallest elements of the list.

Another possibility for reducing the size of the problem consists of discarding the smallest element of the list directly, by calling the `remove` method. In this case, the recursive diagram would be:

Listing 4.13 Recursive selection sort algorithm.

```

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:])

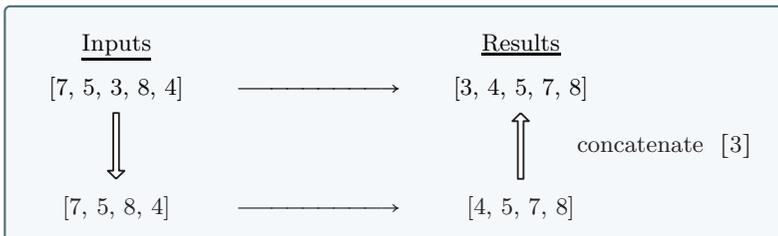
```

Listing 4.14 Recursive variant of the selection sort algorithm.

```

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 [m] + select_sort_rec_alt(b)

```



The only difference with respect to the previous diagram is the order of the elements in the input to the subproblem. This does not affect the recursive rule, where we must also concatenate the discarded smallest element ([3]) with the output list of the subproblem. Listing 4.14 shows a possible implementation of the function, which relies on the `remove` method. Again, the fifth line makes a copy of the input list in order to keep it unaltered when calling the method.

Finally, the runtime cost of these algorithms is characterized by (3.24), since finding (and removing) the smallest element of a list requires on the order of n operations. Thus, the algorithms run in $\Theta(n^2)$.

4.4.2 Horner's method for evaluating polynomials

In this problem the goal consists of evaluating a polynomial of degree n :

$$P(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0, \quad (4.5)$$

at some real value x . The sum contains powers of x that are multiplied by the coefficients c_i . A naive algorithm that computes each power independently would require on the order of n^2 multiplications. Instead, Horner's method only needs to perform n products. Its clever idea is based on expressing the polynomial as:

$$P(x) = c_0 + x(c_1 + x(c_2 + \cdots + x(c_{n-1} + c_n x))).$$

The size of the problem is clearly the degree n . Thus, the base case occurs when $n = 0$, where the result is obviously c_0 . In practice, \mathbf{c} will be a list (or a similar data structure such as an array) of $n+1$ elements that represents the polynomial. Therefore, the base case is reached when the length of \mathbf{c} is one.

In order to apply recursion we need to detect a self-similar subproblem of smaller size. The decomposition of the problem, decrementing its size by a unit, is:

$$P(x) = c_0 + \underbrace{x(c_1 + \overbrace{x(c_2 + \cdots + x(c_{n-1} + c_n x))}^{\text{subproblem}}) \cdots)}_{\text{full problem}}.$$

By discarding coefficient c_0 and the first multiplication times x , the resulting subproblem has exactly the same structure as the original problem. If $p(\mathbf{c}, x)$ is the function that evaluates the polynomial defined by \mathbf{c} at x , the recursive formula would be:

$$p(\mathbf{c}, x) = c_0 + x \cdot p(\mathbf{c}_{1..n}, x),$$

where the list $\mathbf{c}_{1..n}$ is simply \mathbf{c} without its first element. The full function is:

$$p(\mathbf{c}, x) = \begin{cases} c_0 & \text{if } n = 1, \\ c_0 + x \cdot p(\mathbf{c}_{1..n}, x) & \text{if } n > 1, \end{cases}$$

Listing 4.15 Horner's method for evaluating a polynomial.

```

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)

```

and the corresponding code is shown in [Listing 4.15](#).

It is important to note that discarding c_n instead of c_0 does not lead to Horner's method. In that case the recursive formula would be:

$$p(\mathbf{c}, x) = p(\mathbf{c}_{0..n-1}, x) + c_n x^n,$$

which would evaluate the polynomial as in (4.5). If the power x^n is computed in linear or logarithmic time the method would require $\Theta(n^2)$ or $\Theta(n \log n)$ operations, respectively. However, Horner's method is more efficient since it runs in linear time ($\Theta(n)$).

4.4.3 A row of Pascal's triangle

Pascal's triangle is the triangular arrangement of binomial coefficients shown in [Figure 4.2](#), where the triangle in (a) shows the particular binomial coefficients, and the one in (b) shows the actual integer values associated with them. The following problem consists of determining a list containing the binomial coefficients of the n -th row:

$$\binom{n}{0} \quad \binom{n}{1} \quad \cdots \quad \binom{n}{n-1} \quad \binom{n}{n},$$

where the first and last elements are always 1. The problem can be viewed from a recursive point of view by considering the definition in 3.2. Graphically, it means that a binomial coefficient is the sum of the two immediately above it in the previous row of Pascal's triangle. For example, $\binom{4}{3} = \binom{3}{2} + \binom{3}{3}$. Therefore, it is possible to define a row of Pascal's triangle if we know the values of the previous row. [Figure 4.3](#) shows the relationship between a problem (n -th row) and a subproblem ($(n-1)$ -th row).

The size of the problem is n . Thus, the base case corresponds to $n = 0$, where the output is simply a list containing a 1 ($[1]$). For $n = 1$ we cannot apply the recursive rule in 3.2. Thus, initially it appears that we may need an additional base case for $n = 1$, where the output is

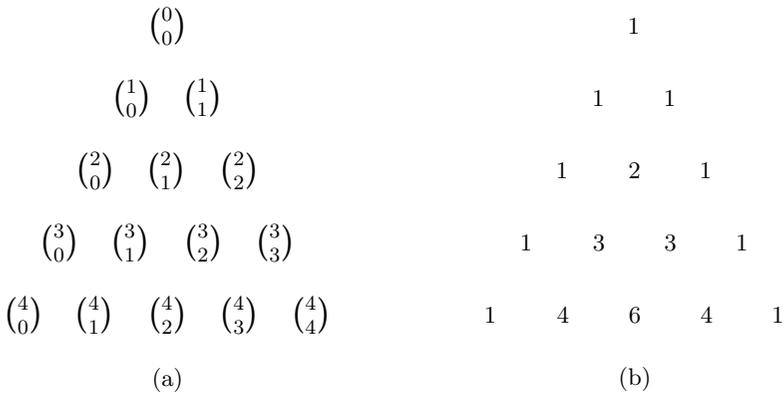


Figure 4.2 Pascal’s triangle.

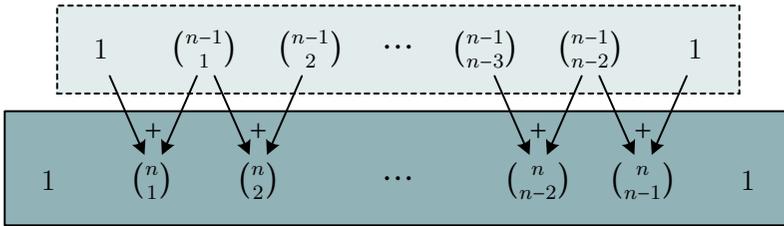


Figure 4.3 Decomposition and recovery of a row of Pascal’s triangle.

Listing 4.16 Function that generates the n -th row of Pascal’s triangle.

```

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

```

$([1, 1])$. However, since all of the rows for $n > 0$ begin and end with a 1, these elements can be incorporated in the recursive case by default. Thus, in this scenario a special base case for $n = 1$ would be unnecessary.

In particular, in the recursive case we assume that we know the solution to the subproblem, which allows us to compute every element

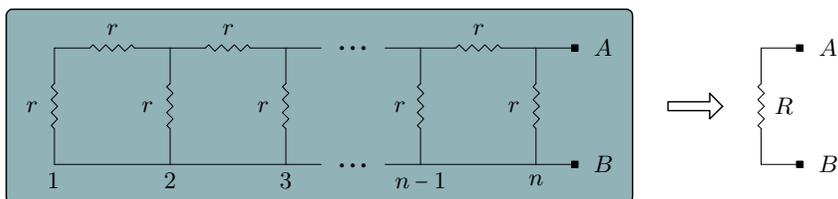


Figure 4.4 Ladder of resistors problem.

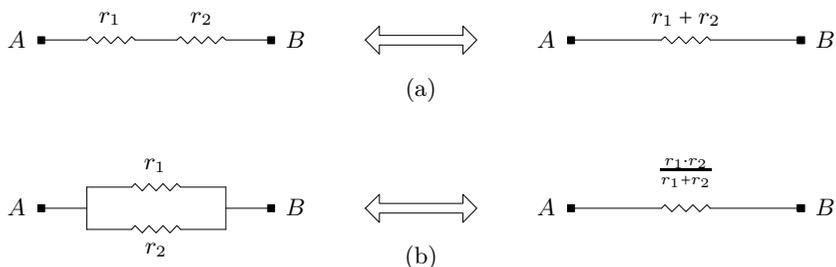


Figure 4.5 Equivalence of circuits with resistors.

of the solution except for the ones at the extremes (these elements can be appended at some point during the process). Listing 4.16 shows a possible solution to the problem. The recursive case starts by inserting a 1 in a list (`row`) that will contain the result. Line 6 computes the result of the subproblem of size $n - 1$, and in lines 7 and 8 the loop adds consecutive integers of the subsolution (as shown in Figure 4.3), appending the sum to the result. Finally, a 1 is inserted at the end, which completes the solution. Exercise 4.14 proposes replacing the loop by a recursive function.

4.4.4 Ladder of resistors

In the next problem the goal is to simplify the electrical circuit in Figure 4.4 resembling a ladder that contains several layers of resistors whose resistance is r . In particular, it is possible to substitute the entire circuit by an equivalent one composed of a single resistor. Thus, the objective consists of determining the value of the resistance R (as a function of the resistance r) that will yield equivalent circuits. Figure 4.5 shows how to transform a circuit with two resistors with resistances r_1 and r_2 when they are connected in series, as shown in (a), or in parallel, as illustrated in (b). When they are connected in series the resulting resistance

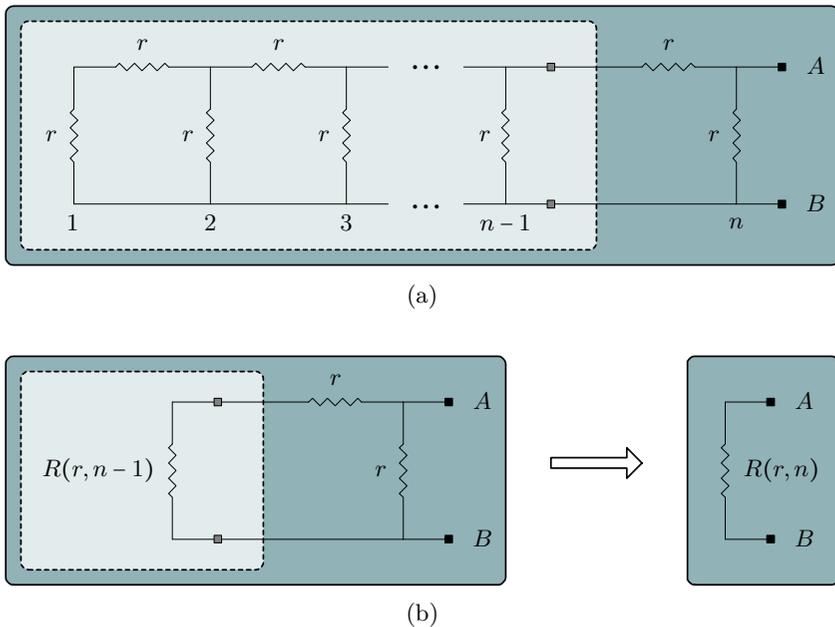


Figure 4.6 Decomposition of the ladder of resistors problem, and derivation of the recursive case through induction.

is $r_1 + r_2$. Instead, when they are connected in parallel the new resistance is $r = (r_1 \cdot r_2) / (r_1 + r_2)$. Alternatively, it can be expressed as:

$$\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}. \tag{4.6}$$

These rules can be applied successively to pairs of resistors until we obtain a circuit that contains a single resistor. However, this process is tedious. Instead, recursion provides a succinct and elegant solution.

The problem has two input parameters: the resistance (r) and the number of rungs (n) of the ladder. The size of the problem is clearly n (r plays a role in the final value, but is not responsible for the runtime of the algorithm). Let $R(r, n)$ denote the recursive function. The base case occurs when $n = 1$, where the initial ladder would only contain a single resistor. Therefore, in that case $R(r, 1) = r$. For the recursive case we need to find a subproblem within the original with exactly the same structure. Figure 4.6(a) shows the decomposition of the problem by decrementing its size by a unit. The circuit associated with the subproblem can be replaced by a single resistor of resistance $R(r, n - 1)$,

Listing 4.17 Function that solves the ladder of resistors problem.

```

1 def circuit(n, r):
2     if n == 1:
3         return r
4     else:
5         return 1 / (1 / r + 1 / (circuit(n - 1, r) + r))

```

as shown in (b), where we can assume that we know its value by using induction. Finally, it is fairly straightforward to simplify the resulting circuit that contains only three resistors. Firstly, the left and top resistors are connected in series. Thus, they can be merged to form a resistor with resistance $R(r, n - 1) + r$. Finally, this new resistor is connected in parallel to the right resistor. By applying (4.6), $R(r, n)$ can be defined through:

$$\frac{1}{R(r, n)} = \frac{1}{r} + \frac{1}{R(r, n - 1) + r}.$$

The recursive function is therefore:

$$R(r, n) = \begin{cases} r & \text{if } n = 1, \\ 1 / \left(\frac{1}{r} + \frac{1}{R(r, n - 1) + r} \right) & \text{if } n > 1. \end{cases}$$

Listing 4.17 shows the associated code. Finally, as a curious note, it is possible to show that $R(r, n) = rF(2n - 1)/F(2n)$, where F is the Fibonacci function.

4.5 EXERCISES

Exercise 4.1 — Listing 2.6 contains a linear-recursive Boolean function that determines whether a nonnegative integer n is even, where the decomposition reduces the size of the problem (n) by two units. Define and code an alternative method based on decrementing the size of the problem by a single unit.

Exercise 4.2 — Implement a recursive function that computes the power b^n in logarithmic time, for a real base b , and an integer n , which can be negative.

Exercise 4.3 — Implement a recursive function that computes the n -th power of a square matrix, which is another square matrix of the same dimensions. Use the NumPy package and its methods:

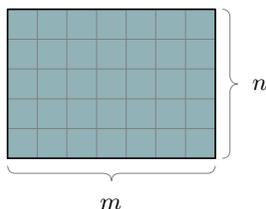


Figure 4.7 The product of two nonnegative integers n and m can be represented as the number of unit squares that form an $n \times m$ rectangle.

- **identity**: returns the identity matrix.
- **shape**: indicates the dimensions of a matrix.
- **dot**: multiplies matrices.

Finally, compute matrices:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

for $n = 1, \dots, 10$, and print the element in their first row and second column. What are these numbers?

Exercise 4.4 — Implement recursive functions that compute a “slow” product of two nonnegative integers m and n . The algorithms are allowed to add and subtract numbers, but they may not use the multiplication ($*$) operator. Use decompositions that decrease one or both of the inputs by a single unit. In addition, illustrate the recursive design thought process through diagrams (for example, by using the general diagram in Figure 2.5; or through diagrams that represent the product of n and m as the area of a rectangle of height n and width m , where the result is the number of square blocks of unit area needed to build an $n \times m$ rectangle, as shown in Figure 4.7).

Exercise 4.5 — Implement a more efficient “slow” product recursive function by using the decomposition that divides both input parameters by two. Use rectangular diagrams like the one described in Figure 4.7.

Exercise 4.6 — In Python and other programming languages methods can also be parameters to other methods. Define and code a general recursive function that computes the sum:

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

where m and n are integers, and f is a function. Use it in order to calculate and print:

$$\sum_{i=1}^n i^3,$$

for $n = 0, \dots, 4$.

Exercise 4.7 — Define and code a function that computes the number of digits of a nonnegative integer n .

Exercise 4.8 — Define and code a function that, given a decimal number n whose digits are either zero or one, returns the number whose binary representation is precisely the sequence of zeros and ones in n . For example, if $n = 10110_{10}$ the function returns 22, since $10110_2 = 22$.

Exercise 4.9 — Derive the recursive case of the “decimal to base b conversion” problem in [Section 4.2.2](#) by using the general diagram for thinking about recursive cases in [Figure 2.5](#).

Exercise 4.10 — Recall the problem in 3.8 that processes (binary) numbers of n bits (from right to left). Derive and code a recursive algorithm that, given a particular number x of n bits, determines the position of its least significant bit set to 1. In particular, consider that the least-significant bit is located at position 1. For example, for $x = 01110\underline{1}00$ the rightmost bit set to 1 is located at position 3.

Exercise 4.11 — Write a recursive method that uses the function developed in [Exercise 4.10](#) in order to solve [Exercise 3.8](#) computationally. Print the solutions for numbers expressed with $n = 1, \dots, 5$ bits.

Exercise 4.12 — Code a recursive function that returns the number of vowels in a given string.

Exercise 4.13 — A binomial coefficient $\binom{n}{m}$ is a function of two integer parameters n and m . Define three linear-recursive functions for calculating binomial coefficients by considering three decompositions: one that decrements n , another that decrements m , and one that decrements both. Use the definition in [\(3.1\)](#) to develop the recursive cases.

Exercise 4.14 — Replace the loop in [Listing 4.16](#) with a recursive function. It should receive a row of Pascal’s triangle, and return a list with the sum of its consecutive terms. For example, if the input is $[1, 3, 3, 1]$ the result should be $[4, 6, 4]$.

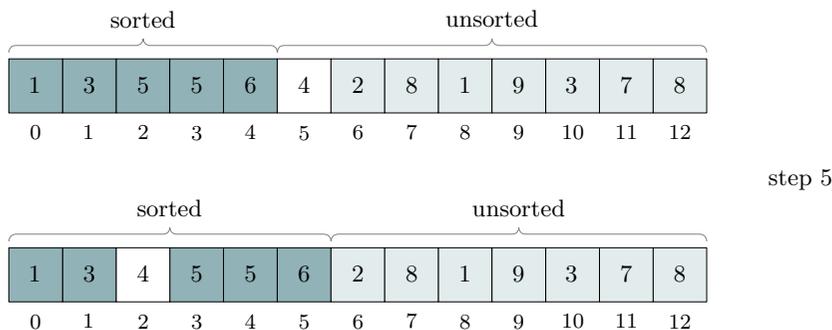


Figure 4.8 Step of the insertion sort algorithm.

Exercise 4.15 — Consider a nonnegative integer n whose digits always appear in ascending order from left to right, such as 24667. In other words, if $d_{m-1} \cdots d_1 d_0$ represents the sequence of (m) digits of n , then $d_i \leq d_j$ for $i < j$. Given an additional digit $0 \leq x \leq 9$, write a function that returns the integer that results from inserting x in n , such that its digits also appear in ascending order from left to right. For instance, if $n = 24667$ and $x = 5$, the function should return 245667. Avoid redundant base cases.

Exercise 4.16 — This exercise is similar to [Exercise 4.15](#). Given a sorted list of numbers in ascending order (**a**), and a number x , implement a function that inserts x in a location in **a** such that the resulting list will remain sorted.

Exercise 4.17 — The “insertion sort” method sorts a list by repeatedly applying the procedure shown in [Figure 4.8](#). At the i -th step the elements from indices 0 to $i - 1$ will be sorted, while the rest of the list will not be (necessarily) sorted. In order to proceed, the method inserts the element at index i into the sorted sublist in a location where the resulting list (from index 0 to i) will remain sorted. If this operation is performed from index (step) 1 (or 0) until $n - 1$ the list will be sorted after the last step. Implement a recursive variant of the method that uses the solution to [Exercise 4.16](#) to sort a list.

Linear Recursion II: Tail Recursion

You are only as beautiful as your last action.

— Stephen Richards

TAIL recursion, also known as “final” recursion, is a form of linear recursion in the sense that the recursive cases only invoke the method once. However, in tail recursion a recursive call is the last action performed by the method. For example, the function in (1.15) is tail-recursive since it does not manipulate the result of the function call in the recursive case. This implies that it will return a value directly obtained in a base case. In addition, functions may require more parameters in order to store intermediate results that will be used in the base cases for providing the final output. The tail-recursive methods seen so far include Listings 2.3, 2.5, and 2.6.

In the previous chapter all of the methods were based on decrementing the size of the problem by one or two units, except for the ones in Section 4.1.1.2. In contrast, this chapter will present more algorithms that divide the size of the problem by two. In these cases the recursive calls will solve a single problem of half the size of the original’s, since the methods only invoke themselves once. Some authors refer to this strategy as “divide and conquer.” However, we will only use this term when the algorithms need to solve *several* independent subproblems whose size is a fraction of that of the original’s. These algorithms will be covered in Chapter 6.

Finally, tail recursion is also special due to its relationship with iteration. Chapter 11 will analyze this connection.

5.1 BOOLEAN FUNCTIONS

This introductory section examines how the choice of base cases can lead to linear or tail-recursive algorithms for some Boolean functions.

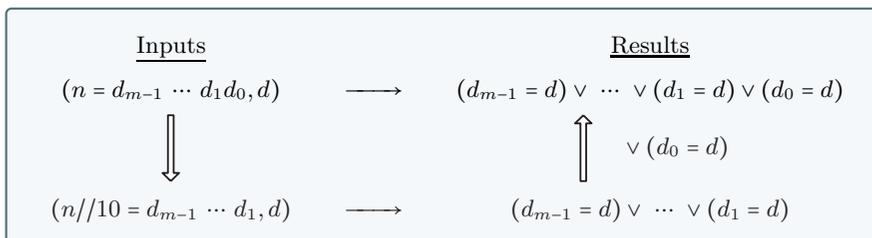
5.1.1 Does a nonnegative integer contain a particular digit?

Given a nonnegative integer n , and a digit $0 \leq d \leq 9$ (which is also an integer), in this problem the goal consists of determining whether d is present in n . The recursive method will therefore be a Boolean function with parameters n and d . Additionally, we will assume that n does not contain any leading zeros (for example, 358 cannot be written as 0358). However, we will consider that the number zero contains the digit 0.

5.1.1.1 Linear-recursive algorithm

The first step consists of establishing the size of the problem, which is the number (m) of digits of n . A base case occurs when n only contains a single digit (i.e., when $n < 10$), where the result is simply whether $n = d$. Let us assume for the time being that this is the only base case.

For the recursive case, it is cumbersome to express the value of a digit in terms of n . Therefore, we can use an alternative notation, as discussed in [Section 2.5.3](#). In particular, let $d_{m-1} \cdots d_1 d_0$ represent the sequence of m digits of n in base 10. We can form the following diagram when decrementing the size of the problem by a unit (i.e., using $n//10$, which discards the least significant digit of n):



where \vee denotes logical OR (disjunction). The diagram clearly illustrates that if d is a digit of n , then it is either present in $n//10$ (this is the result of the recursive call), or it should be equal to d_0 .

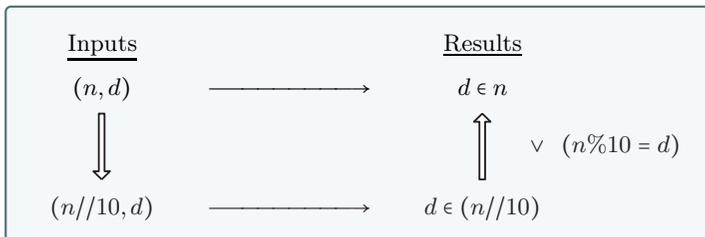
Other notations are also possible. We can simply choose words or expressions, even if they are not technically correct logically or mathematically. For instance,

Listing 5.1 Linear-recursive Boolean function that determines if a non-negative integer contains a digit.

```

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)

```



introduces a clear abuse of notation, since the membership symbol \in should be used with a set. Thus, $d \in n$ actually represents whether d belongs to the set of digits that form n . Nevertheless, the diagram serves its purpose, accurately indicating the function’s recursive case.

Listing 5.1 shows the code that implements the function, where $n\%10$ is the least significant digit of n . Observe that the method is linear-recursive, since the function invokes itself once, and needs to process the output of the subproblem (i.e., compute the result of the expression involving the `or` operator). Additionally, if the condition $(n\%10==d)$ is `True` then a compiler or interpreter that uses “short-circuit” evaluation would automatically return a true value, avoiding the call to `contains_digit(n//10,d)`. In particular, short-circuit evaluation allows us to avoid unnecessary computations by considering the properties: `True` \vee `b` = `True`, and `False` \wedge `b` = `False`, where `b` is some Boolean expression. Since these hold regardless of the value of `b`, there is no need to evaluate `b`.

5.1.1.2 Tail-recursive algorithm

Most programming languages support short-circuit evaluation. However, we can force it by considering an additional base case, which leads to a tail-recursive algorithm. In particular, for this problem an algorithm can return `True` as soon as it finds a digit in n that is equal to d . Therefore, we can return `True` if the last digit of n is equal to d (i.e., if $n\%10 = d$).

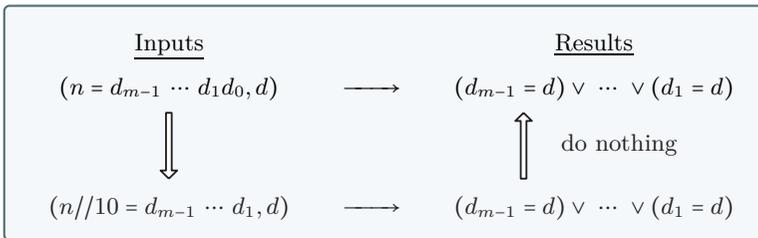
Listing 5.2 Tail-recursive Boolean function that determines if a nonnegative integer contains a digit.

```

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)

```

By considering this base case we can be sure that $n \% 10 = d_0 \neq d$ in the recursive case. Therefore, the recursive diagram is now:



This implies that the result of the subproblem is exactly the output of the original problem, and the function can simply return the result of the recursive call, without processing it. Together with the base cases, this leads to a tail-recursive algorithm that can be coded as shown in Listing 5.2.

Lastly, it is important to understand that although the chosen decomposition divides the input by a constant (10), the size of the problem is only reduced by a unit. The time complexity for the algorithm is logarithmic with respect to the input n , but it is linear with respect to the number of digits (m) of n .

Finally, this problem has analogous counterparts that rely on data structures such as lists, arrays, etc., since these also represent sequences of elements. For example, it is very similar to deciding if a string contains a character, or if an element is present in a list. Although the codes may be different, the underlying reasoning is essentially identical.

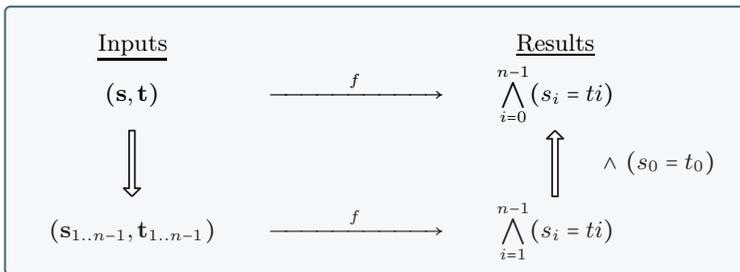
5.1.2 Equal strings?

The following problem consists of determining whether two strings are equal (of course, in Python we can simply use the `==` operator). A

Boolean function that solves the problem will therefore have two string input parameters. If their lengths are different the algorithm can return `False` immediately in a base case. Thus, the challenge lies in solving the problem when they have the same length, which would constitute its size.

5.1.2.1 Linear-recursive algorithm

The smallest instance therefore corresponds to two empty strings, where the result is obviously `True`. In this section we will assume that there are no additional base cases, which leads to a linear-recursive function. In particular, the decomposition can decrease the size of the problem by a unit. This implies discarding one character (located at the same position) from each input string. Omitting the first one leads to the following recursive diagram, where \mathbf{s} and \mathbf{t} are two input strings of length n :



Clearly, the method has to check that the first characters are the same, *and* that the remaining substrings are also identical through a recursive call. The Boolean function is therefore:

$$f(\mathbf{s}, \mathbf{t}) = \begin{cases} \text{false} & \text{if } \text{length}(\mathbf{s}) \neq \text{length}(\mathbf{t}), \\ \text{true} & \text{if } n = 0, \\ (s_0 = t_0) \wedge f(\mathbf{s}_{1..n-1}, \mathbf{t}_{1..n-1}) & \text{if } n > 0, \end{cases}$$

[Listing 5.3](#) shows the corresponding code, which describes a linear-recursive function.

5.1.2.2 Tail-recursive algorithm

Similarly to the example in [Section 5.1.1](#), we can force a short-circuit evaluation by considering an additional base case. In particular, an algorithm can return `False` as soon as it detects two different characters

Listing 5.3 Linear-recursive function that determines if two strings are identical.

```

1 def equal_strings(s, t):
2     if len(s) != len(t):
3         return False
4     elif s == '':
5         return True
6     else:
7         return s[0] == t[0] and equal_strings(s[1:], t[1:])

```

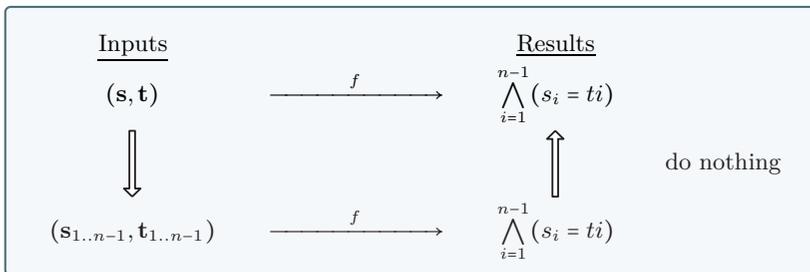
Listing 5.4 Tail-recursive function that determines if two strings are identical.

```

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:])

```

at the same position in the strings. Thus, we can incorporate a base case that checks if $s_0 \neq t_0$. In that case, the algorithm would automatically return `False`. With this base case we can be sure that $s_0 = t_0$ in the recursive case. Therefore, the new recursive diagram would be:



Again, the result of the subproblem is exactly the output of the original problem, which leads to a tail-recursive algorithm. Finally, together with the base cases, the function can be coded as in [Listing 5.4](#).

Listing 5.5 Tail-recursive linear search of an element in a list.

```

1 def linear_search_tail(a, x):
2     n = len(a)
3     if a == []:
4         return -1
5     elif a[n - 1] == x:
6         return n - 1
7     else:
8         return linear_search_tail(a[:n - 1], x)

```

5.2 SEARCHING ALGORITHMS FOR LISTS

This section will cover algorithms that solve the classical problem of finding the position i of an element x within a list $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$. In other words, given \mathbf{a} and x , the goal is to find an index i for which $a_i = x$. If the element x appears several times in the list the methods that solve the problem can return any valid index that indicates a location where x is found in the list.

For a list of length n , a method that solves the problem will return an integer index ranging from 0 until $n - 1$, when the element is present in the list. In contrast, if the list does not contain x the algorithms must return some value outside of that interval. Some Python implementations simply return a Boolean false value. However, this option is not valid in many other languages (C, Java, Pascal, etc.), where methods can only return values of a single data type. Since the indices are necessarily integers, in these languages the value that indicates that an element is not contained in the list must also be an integer. Thus, we will use the integer value -1 to indicate that an element is not present in a list, which is compatible with other programming languages.

5.2.1 Linear search

Firstly, assume that the list \mathbf{a} is not (necessarily) sorted. Any algorithm would require n comparisons in the worst case, which occurs when the element does not belong to the list. Note that any method would have to analyze every single element of the list in order to conclude that the element is missing. In this situation the algorithms would perform a “linear search” for an element in a list, which runs in $\mathcal{O}(n)$.

The size of the problem is clearly n . If the list is empty the method can trivially return -1 . Another base case can return the location of

Listing 5.6 Linear-recursive linear search of an element in a list.

```

1 def linear_search_linear(a, x, n):
2     if a == []:
3         return -n - 1
4     elif a[0] == x:
5         return 0
6     else:
7         return 1 + linear_search_linear(a[1:], x, n)
8
9
10 def linear_search_linear_wrapper(a, x):
11     return linear_search_linear(a, x, len(a))

```

the element if it is found. This base case will depend on the choice of decomposition. If we reduce the size of the problem by discarding the last element of the list, then the base case will need to check whether x is located at the last position. If indeed $a_{n-1} = x$, then the method can simply return $n - 1$. Otherwise, it will need to carry out a recursive call in order to solve the subproblem of size $n - 1$, and the final result will be exactly that of the subproblem's, which leads to a tail-recursive solution. Listing 5.5 shows a possible implementation in Python of the described linear search function.

Finally, it is worth noticing that with the previous decomposition the locations of the elements in the sublist are the same as those in the original list. Alternatively, if the decomposition omitted the first element of the list, all of the indices in the sublist would be decremented by a unit with respect to those in the original list. For example, if the list were $[2, 8, 5]$, the 8 is also located at position 1 in $[2, 8]$, but it appears at location 0 in $[8, 5]$. This leads to more complex methods that require an additional parameter. For example, Listing 5.6 shows a solution that adds a unit to the result of each recursive call. The method trivially returns 0 if $x = a_0$, but the base case for an empty list is more complicated. Note that if that base case is reached the algorithm will have added n units in the n previous recursive calls. Thus, it needs to return $-n - 1$, where n is the length of the initial list (not the particular length of the input argument since it is 0 in that case) in order to return -1 , indicating that x does not appear in the list. Since n cannot be obtained from an empty list it has to be passed as an extra parameter in every function call in order to be recovered in the base case. The code

Listing 5.7 Alternative tail-recursive linear search of an element in a list.

```

1 def linear_search_tail_alt(a, x, index):
2     if a == []:
3         return -1
4     elif a[0] == x:
5         return index
6     else:
7         return linear_search_tail_alt(a[1:], x, index + 1)
8
9
10 def linear_search_alt_tail_wrapper(a, x):
11     return linear_search_tail_alt(a, x, 0)

```

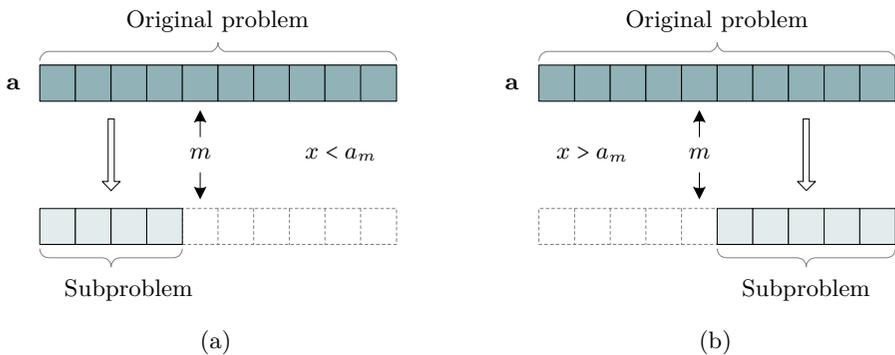


Figure 5.1 Decomposition related to the binary search algorithm.

therefore requires a wrapper method, where the third parameter of the recursive function is initialized to n .

The last decomposition implies that the algorithm will search for the element starting at index 0, and will progressively advance until reaching index $n - 1$. Another solution consists of explicitly specifying the index of the element that will be compared to x each time the method invokes itself. This introduces a new parameter that can be viewed as an accumulator variable that is incremented with each function call, and leads to the tail-recursive function in Listing 5.7, which correctly returns -1 if the list is empty. The function must be called with the accumulator `index` parameter set to 0 from a wrapper method. Lastly, observe that the result is stored in that parameter.

Listing 5.8 Binary search of an element in a list.

```

1 def binary_search(a, x, lower, upper):
2     if lower > upper: # empty list
3         return -1
4     else:
5         middle = (lower + upper) // 2
6
7         if x == a[middle]:
8             return middle
9         elif x < a[middle]:
10            return binary_search(a, x, lower, middle - 1)
11        else:
12            return binary_search(a, x, middle + 1, upper)
13
14
15 def binary_search_wrapper(a, x):
16     return binary_search(a, x, 0, len(a) - 1)

```

5.2.2 Binary search in a sorted list

When the input list is sorted it is possible to search for an element by using a faster approach. Assume that the list \mathbf{a} is sorted in ascending order, where $a_i \leq a_{i+1}$, for $i = 0, \dots, n - 2$. The problem can be solved in $\mathcal{O}(\log n)$ running time by “binary search” algorithms. The main idea behind these methods (there are several variants) consists of dividing the size of the problem progressively by two. [Figure 5.1](#) illustrates the decomposition of the problem. Firstly, the algorithms compute the middle index m . If $x = a_m$ they can terminate in a base case simply returning m . Otherwise, since the list is sorted, it is possible to keep searching for x in only one of the sublists to the right or left of m . In particular, if $x < a_m$ then x cannot appear in the sublist $[a_m, \dots, a_{n-1}]$, and the methods can continue to search for x in $[a_0, \dots, a_{m-1}]$, as shown in (a). Analogously, if $x > a_m$ then x can only appear in $[a_{m+1}, \dots, a_{n-1}]$, as illustrated in (b). This approach essentially divides the size of the problem (n) by two. Therefore, the runtime will be characterized by (3.20), which implies that the methods will run in logarithmic time with respect to n in the worst case. Lastly, the functions will return -1 (at a base case) when the list is empty.

Similarly to the linear search functions in [Listings 5.6](#) and [5.7](#), the binary search methods need to include additional input parameters in order to indicate the original indices of the list. For example, consider

searching for an 8 in the list $[1,3,3,5,6,8,9]$. If we simply pass the sublist $[6, 8, 9]$ to the function in a recursive call, the 8 is now at index 1, while it was located at position 5 in the original list. Thus, we need to indicate that the sublist's first index should really be 5. In other words, we need to specify an extra parameter that contains the position in the original list where the sublist begins. Listing 5.8 shows an implementation of the method that not only includes this parameter (`lower`), but also uses an additional one (`upper`), which is the index of the original list in which a sublist ends. Thus, `lower` and `upper` simply specify the boundaries of a sublist within `a`. The `upper` parameter is not strictly necessary in Python since we can use the length of the list in order to determine the sublist's upper index (Exercise 5.4 proposes implementing a method that only uses the `lower` parameter). However, the code in Listing 5.8 is simpler in the sense that it requires less arithmetic operations, does not invoke itself with sublists (the recursive calls use the entire list `a`), and shows how the method can be implemented in other programming languages where the length of lists is not directly available. In particular, if `lower` is greater than `upper` then the list is empty and the method returns `-1`. Otherwise, the function checks the base case associated with the condition $x = a_m$. If x is not at index m , the method carries out one of the two recursive calls, on the appropriate sublist, depending on the condition $x < a_m$. Finally, the wrapper method is used to initialize the `lower` and `upper` indices to 0 and $n - 1$, respectively.

5.3 BINARY SEARCH TREES

A binary search tree is a data structure used for storing data items, each of which is associated with a certain key (we will consider that it has to be unique), similarly to the entries in a Python dictionary. It is an important data structure that allows us to carry out certain operations efficiently, such as searching, inserting, or removing elements. The data in each node of the binary tree is a pair (key, item), and the nodes are arranged according to the values of the keys, which can be compared through the `<` operator, or some equivalent function that allows us to determine if a key is less than another one (i.e., a total preorder binary relation). In particular, given any node in the tree, every key in its left subtree must be less than its key, while all of the keys in its right subtree must be greater than its key. This is also known as the “binary search tree property,” which implies that every subtree of a binary search tree is also a binary search tree.

For example, this data structure can be used to store the information of a birthday calendar. [Figure 5.2](#) shows a binary tree of seven persons that allows us to retrieve their birthdays according to their names, which are strings that we can assume are unique (naturally, we can include last names or tags in order to identify each person uniquely). Keys can be strings since they can be compared, for example, according to the lexicographic order (in Python we can simply use the `<` operator). Therefore, the names in the binary search tree are sorted as they would appear in a regular dictionary. For instance, consider the root node associated with “Emma”. Observe that all of the names contained in its left subtree would appear before “Emma” in a dictionary, and all of the names in the right subtree would be found after “Emma”. Furthermore, this property holds for every node of the binary search tree.

In order to implement a binary search tree, we first have to decide how to code a binary tree. There are several possibilities (one of the most common approaches consists of using object oriented features in order to declare a class associated with the nodes of the tree), but in this book we will simply use lists. In particular, every node of the tree will consist of a list of four elements: the key, the item, the left subtree, and the right subtree, where the subtrees are also binary search trees. Thus, the binary search tree in [Figure 5.2](#) would correspond to the following list:

```
[ '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)

which is illustrated graphically in [Figure 5.3](#). Observe that the left and right subtrees are lists, which are empty at the leaf nodes.

5.3.1 Searching for an item

The goal of the next problem is to search for some item with a known key k in a binary search tree, and retrieve its associated item. We can assume that the size of the problem is the height of the tree. A trivial base case occurs when the list that represents the binary tree is empty. In that case the algorithm can simply return `None`. There is another situation where the algorithm can provide a result without carrying out a recursive call. If the key of the root node is k , the method will have

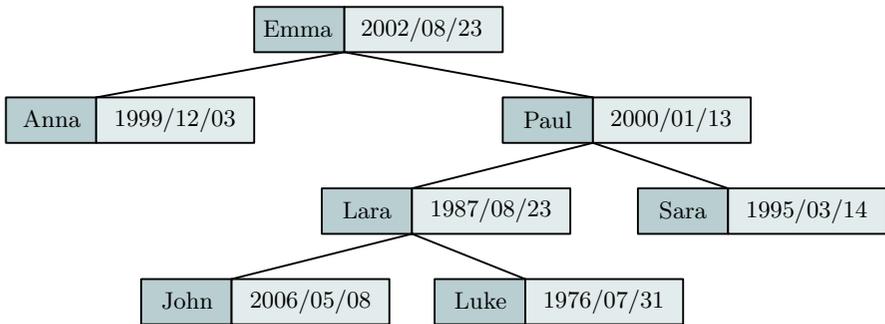


Figure 5.2 Binary search tree that stores information about a birthday calendar.

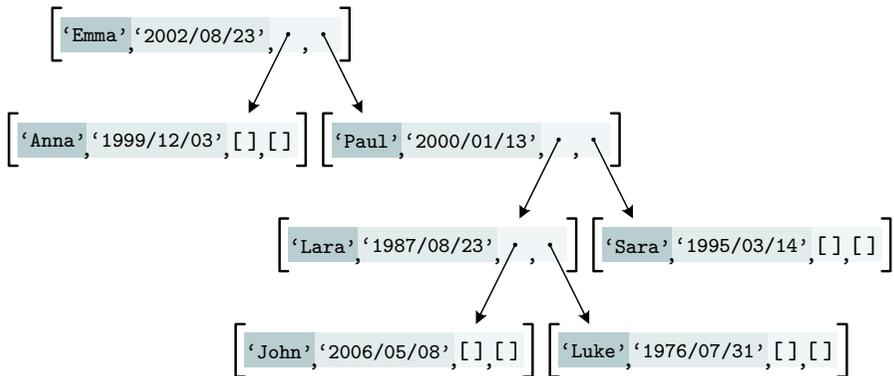


Figure 5.3 Binary search tree in Figure 5.2 and (5.1), where each node is a list of four elements: name (string), birthday (string), left subtree (list), and right subtree (list).

found the item, and can therefore return it. Thus, in the recursive cases we can be sure that the root node does not contain the searched item.

In the next step our goal is to find an appropriate decomposition of the problem that reduces its size. We have already seen that trees are composed recursively of subtrees. Thus, we could consider searching for the item in the two subtrees of the binary search tree. This guarantees reducing the size of the problem by a unit, since it discards the root node. Nevertheless, it is easy to see that in this problem we can also avoid searching in an entire subtree. If the key k is less than the key of the root node (k_{root}), then we can be sure that the item we are looking for will not be in the right subtree, due to the binary tree search property.

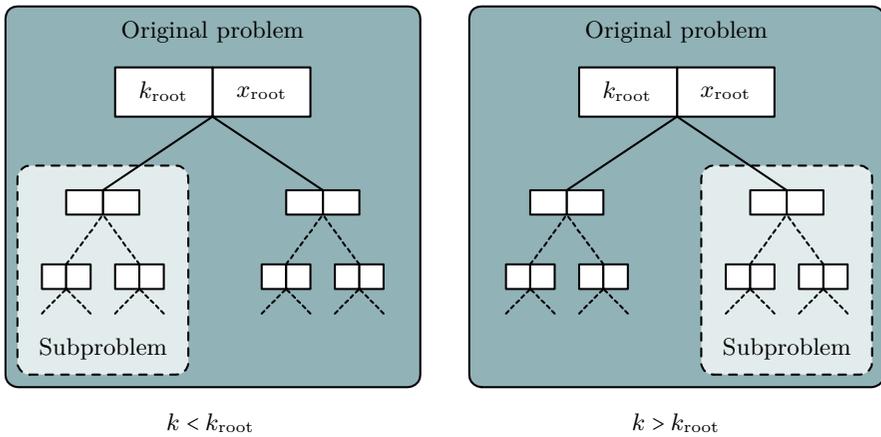


Figure 5.4 Decomposition associated with several algorithms related to binary search trees.

Listing 5.9 Algorithm for searching an item with a particular key in a binary search tree.

```

1 def bst_search(T, key):
2     if T == []:
3         return None
4     elif T[0] == key:
5         return T[1] # return the root item
6     elif key < T[0]:
7         return bst_search(T[2], key) # search in left subtree
8     else:
9         return bst_search(T[3], key) # search in right subtree

```

Analogously, if $k > k_{\text{root}}$, the item will not appear in the left subtree. Figure 5.4 illustrates this idea, where x_{root} is the item stored in the root node. Clearly, for this particular problem there are two recursive cases. If $k < k_{\text{root}}$ the method must keep searching in the left subtree through a recursive call, while if $k > k_{\text{root}}$ it will search for the item in the right subtree. Listing 5.9 shows an implementation of the searching algorithm, where each node of the binary tree is coded as a list of four components, as described in Section 5.3.

Finally, the height of the binary tree determines the cost of the algorithm in the worst case. If the tree is balanced (i.e., it has approximately the same number of nodes on the left and right subtrees of nodes that

Listing 5.10 Procedure for inserting an item with a particular key in a binary search tree.

```

1 def insert_binary_tree(x, T):
2     if T == []:
3         T.extend([x[0], x[1], [], []])
4     else:
5         if x[0] < T[0]:
6             if T[2] == []:
7                 T[2] = [x[0], x[1], [], []]
8             else:
9                 insert_binary_tree(x, T[2])
10        else:
11            if T[3] == []:
12                T[3] = [x[0], x[1], [], []]
13            else:
14                insert_binary_tree(x, T[3])

```

appear on the same level) it runs in $\Theta(\log n)$, where n is the number of nodes in the tree, since it would discard approximately half of the nodes with each recursive call. However, the tree could have a linear structure, in which case the algorithm would run in $\mathcal{O}(n)$.

5.3.2 Inserting an item

The goal in this problem is to insert an item with a given key k in a binary search tree, such that the resulting tree also satisfies the binary search tree property. The method will be a procedure that receives two parameters: (1) a tuple x containing the key and the item, and (2) the binary search tree T . In particular, it will extend the list (T) by adding a new leaf to the tree.

Again we can consider that the size of the problem is the height of the tree. The simplest base case occurs when the tree is empty. In that situation the algorithm must extend the tree in order to incorporate a single node, which obviously will not have any children. The diagrams in [Figure 5.4](#) also illustrate the decomposition that we will follow for solving this problem. Clearly, if the tree is not empty, then the procedure will insert a node in the left subtree if k is smaller than the key of the root node. Otherwise, it will insert it in the right subtree. Firstly, assume that $k < k_{\text{root}}$. There are two possible scenarios that lead to a base case and a recursive case. If the root node does not have a left subtree, then the procedure can simply replace the associated empty list by the node

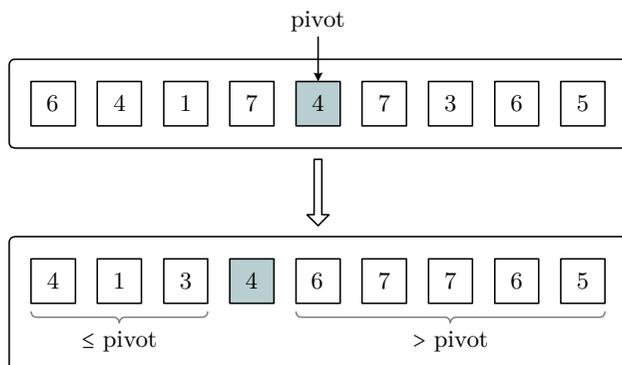


Figure 5.5 Partitioning of a list used in the quicksort and quickselect algorithms.

containing the key and the item (and two empty subtrees). However, if the root node does have a nonempty subtree then the method must insert the new node in that subtree, which naturally leads to a recursive call. The reasoning regarding the right subtree is analogous. Finally, [Listing 5.10](#) shows a possible implementation of the procedure.

5.4 PARTITIONING SCHEMES

A well-known problem in computer science consists of partitioning a list in the following manner:

1. Choose a “pivot” element from the list. Typical choices include the first or middle element of the list, or simply some element selected at random.
2. Construct a new list with the same elements as the input list, or simply permute the input list, where the first elements are smaller than or equal to the pivot, and the last ones are greater than the pivot.

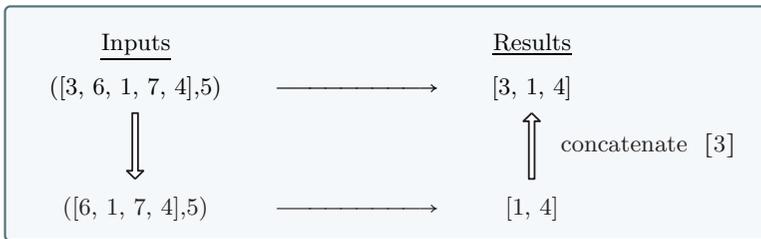
[Figure 5.5](#) illustrates the idea through a concrete example. The problem is relevant since the partitioning is key in other eminent algorithms such as quickselect (see [Section 5.5](#)) or quicksort (see [Section 6.2.2](#)). There are several well-known and efficient algorithms, also denoted as “partitioning schemes,” which solve the problem. The most popular is Hoare’s partitioning method, developed by Tony Hoare, which we will analyze

shortly. Nevertheless, first we will examine a simpler approach for the sake of introducing more intuitive recursive algorithms.

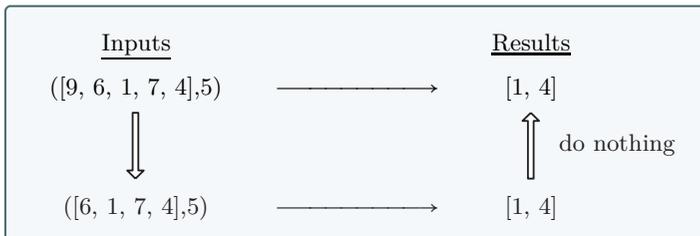
5.4.1 Basic partitioning scheme

A simple idea consists of scanning the entire input list in order to build: (1) a new list that contains the elements that are smaller than or equal to the pivot, and (2) another list formed by the elements that are greater than the pivot. Afterwards, they can be concatenated together with the pivot element, which would obviously be placed in between them.

Since the problems are very similar we will explain only the first one. In particular, the inputs to the problem are a list \mathbf{a} of length n , and some value x that plays the role of the pivot. Its size is clearly the length of the list. The base case occurs when the input list is empty, where algorithms can simply return an empty list. For the recursive case we can decompose the problem by discarding the first element of the list, which reduces its size by a unit. In that case, we can form the following recursive diagrams, depending on whether the first element of the list is smaller than or equal to the pivot. If it is, a concrete diagram could be:



Clearly, the algorithm must concatenate the first element of the list to the output of the subproblem. In contrast, if the first element is greater than x , the diagram is:



In this case, the algorithm simply has to return the solution (list) to the subproblem. [Listing 5.11](#) shows the linear-recursive codes that solve

Listing 5.11 Auxiliary methods for partitioning a list.

```

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)

```

both problems (notice that in the recursive cases that perform the concatenation the function call is not the last action of the method). The runtime cost of the methods can be characterized by:

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

which implies that their runtime is linear with respect to n . Lastly, the functions can be substituted with Python expressions that implement “filters.” In particular, `get_smaller_than_or_equal_to(a,x)` and `get_greater_than(a,x)` return the same lists as the expressions `[y for y in a if y <= x]` and `[y for y in a if y > x]`, respectively.

5.4.2 Hoare's partitioning method

[Listing 5.12](#) contains an iterative version of Hoare's partitioning algorithm. In particular, it partitions a sublist within the input list (**a**), which is specified through lower and upper indices. Firstly, the method chooses a pivot (in this case, the element in the middle of the sublist) and swaps it with the first element of the sublist (lines 3–6). Afterwards, it sets a left index parameter at the second position of the sublist, and a right index at the last location of the sublist (lines 8 and 9). Subsequently the method enters the main loop of the algorithm. Inside it, it

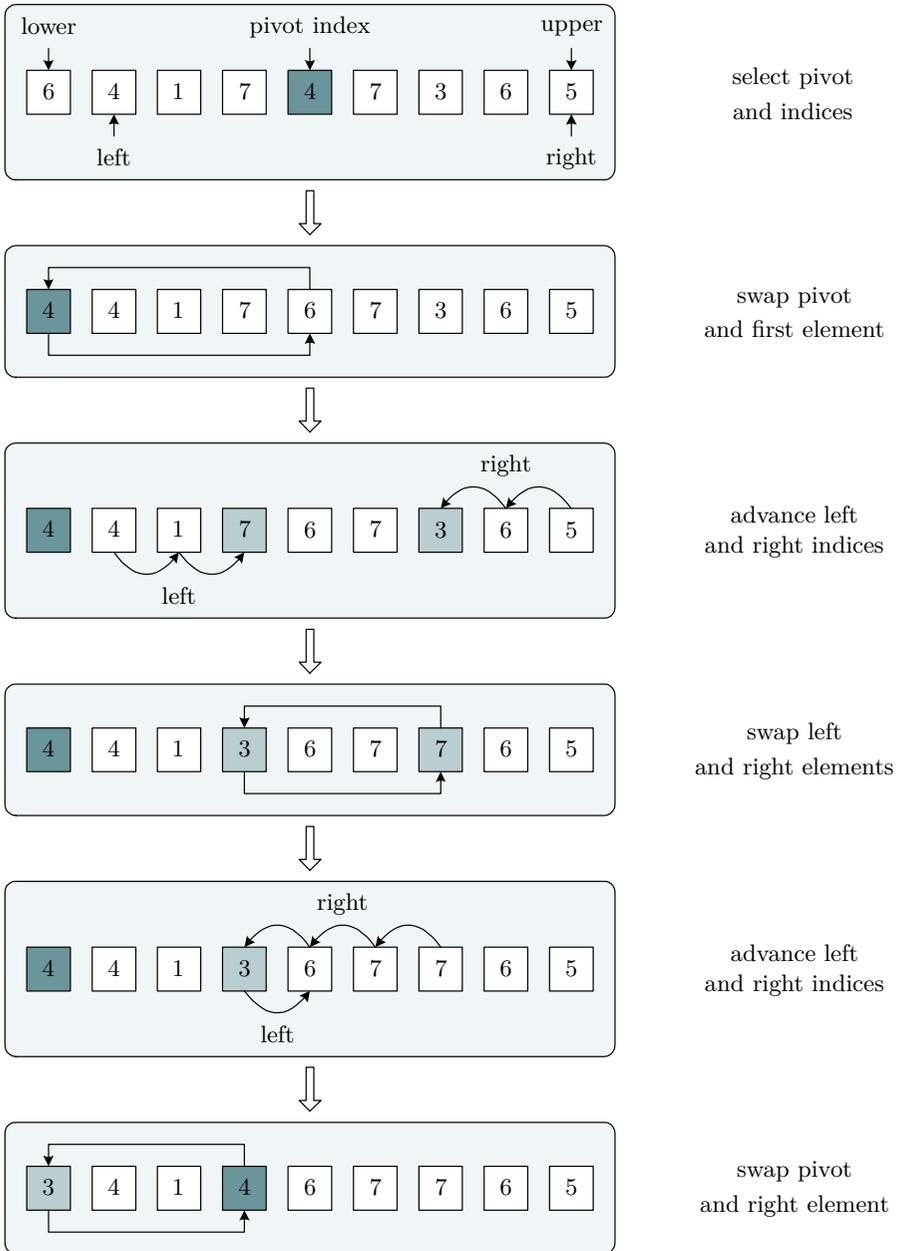


Figure 5.6 Example of Hoare's partition method.

Listing 5.12 Hoare's iterative partitioning algorithm.

```

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

```

iteratively increases the left index until it references an element that is larger than the pivot (lines 13 and 14). Analogously, it decreases the right index until it references an element that is less than or equal to the pivot (lines 16 and 17). The algorithm then swaps the elements referenced by the left and right index (if the left index is smaller than the right one) in lines 19–22. This process is repeated until the indices cross (i.e., until the left index is greater than the right one, which is checked in line 24). At the end of the loop the elements located up to the right index will be smaller than or equal to the pivot, and the elements from the left index until the end of the sublist will be greater than the pivot. Finally, the procedure completes the partition by swapping the element referenced by the right index with the pivot (lines 26 and 27), and re-

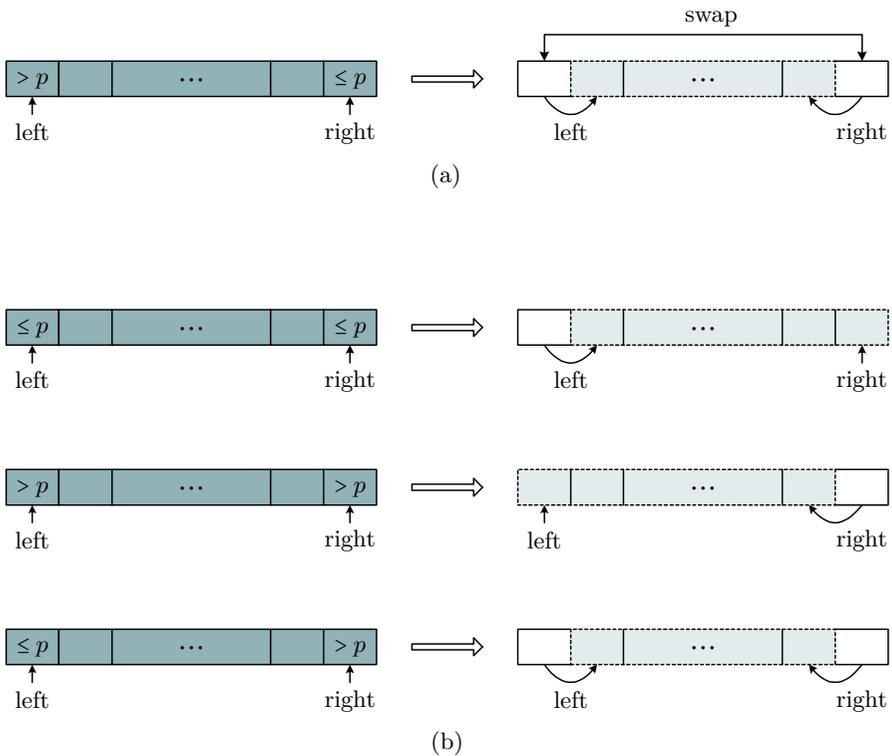


Figure 5.7 Decomposition of Hoare's partitioning problem.

turns the final location of the pivot. Figure 5.6 shows a concrete example of the partitioning method.

We will now see a tail-recursive algorithm that can substitute the main loop in Hoare's partitioning scheme. Given the input list \mathbf{a} , initial left and right indices, as well as the value of the pivot, the method should partition the list analogously as the loop in Hoare's method, returning the final location of the right index. The size of the problem is the difference between the left and right indices, since it determines the number of increment/decrement operations to be performed on the indices until they cross. The base case occurs precisely when they cross (when the left index is greater than the right one), where the method can simply return the right index.

For the recursive case we must reduce the size of the problem by incrementing the left index and/or decrementing the right one. There are two different scenarios, illustrated in Figure 5.7, where p denotes

Listing 5.13 Alternative recursive version of Hoare's partitioning scheme.

```

1 def partition_Hoare_rec(a, left, right, pivot):
2     if left > right:
3         return right
4     else:
5         if a[left] > pivot and a[right] <= pivot:
6             aux = a[left]
7             a[left] = a[right]
8             a[right] = aux
9             return partition_Hoare_rec(a, left + 1,
10                                     right - 1, pivot)
11        else:
12            if a[left] <= pivot:
13                left = left + 1
14            if a[right] > pivot:
15                right = right - 1
16            return partition_Hoare_rec(a, left, right, pivot)
17
18
19 def partition_Hoare_wrapper(a, lower, upper):
20     if upper >= 0:
21         middle = (lower + upper) // 2
22         pivot = a[middle]
23         a[middle] = a[lower]
24         a[lower] = pivot
25
26         right = partition_Hoare_rec(a, lower + 1, upper, pivot)
27
28         a[lower] = a[right]
29         a[right] = pivot
30
31     return right

```

the pivot. If $a_{\text{left}} > p$ and $a_{\text{right}} \leq p$ the method first needs to swap the elements referenced by the indices, and afterwards can perform a function call by advancing both indices, as shown in (a). This leads to a first recursive case that reduces the size of the problem by two units. If the previous condition is **False**, then the method will not swap elements, and will advance at least one of the indices (thus, reducing the size of the problem), as shown in (b). If $a_{\text{left}} \leq p$ it will increment the left index, and if $a_{\text{right}} > p$ it must decrement the right one. Subsequently, the method can invoke itself with the updated indices. [Listing 5.13](#) shows the corresponding code, together with a wrapper method that completes the partitioning algorithm. Note that it is identical to [Listing 5.12](#), where

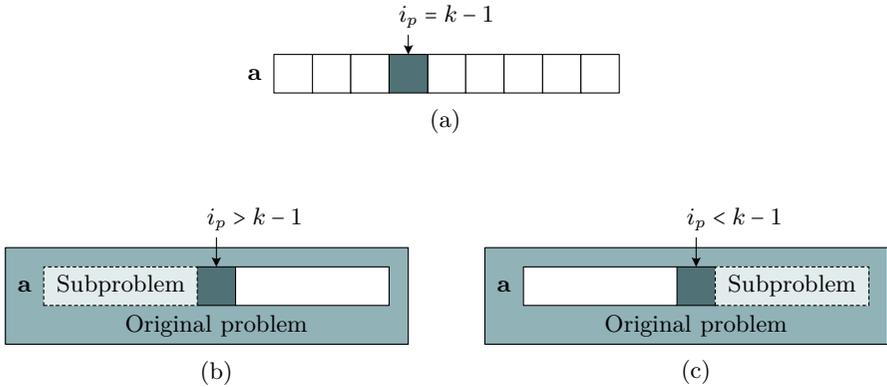


Figure 5.8 Base case and problem decomposition used by the quickselect algorithm.

the code associated with the loop has been substituted by a call to the recursive function. Lastly, the method runs in $\mathcal{O}(n)$ since the runtime of the recursive method can be characterized by the recurrence $T(n) = T(n - 1) + 1$ in the worst case.

5.5 THE QUICKSELECT ALGORITHM

The quickselect algorithm, also developed by Tony Hoare, is a selection method based on Hoare’s partitioning scheme (see [Section 5.4.2](#)). In particular, it finds the k -th smallest number (also called the k -th order statistic) in a list that is not sorted. Assume that an instance of the problem is defined through a numerical input list \mathbf{a} , lower and upper indices defining a sublist within it, and a positive integer k . The algorithm will therefore search for the k -th smallest number in the specified sublist. The size of the problem is the length of the sublist. The smallest instances of the problem correspond to lists that contain a single element. Therefore, the base case occurs when the lower and upper indices are identical, where the method can simply return the element of the list.

For larger lists the method first applies Hoare’s partition scheme, which arranges the list in three parts. A first sublist contains the elements of \mathbf{a} that are less than or equal to a chosen “pivot” element from the list. This list is followed by another one that only contains the pivot. Finally, a third list contains the elements of \mathbf{a} that are greater than the pivot.

Listing 5.14 Tail-recursive quickselect algorithm.

```

1 def quickselect(a, lower, upper, k):
2     if lower == upper:
3         return a[lower]
4     else:
5         pivot_index = partition_Hoare_wrapper(a, lower, upper)
6
7         if pivot_index == k - 1:
8             return a[pivot_index]
9         elif pivot_index < k - 1:
10            return quickselect(a, pivot_index + 1, upper, k)
11        else:
12            return quickselect(a, lower, pivot_index - 1, k)

```

Let i_p denote the index of the pivot. Since the function that implements Hoare's partition scheme returns i_p , we can check whether it is equal to $k - 1$ (since indices start at location 0, the j -th element appears at position $j - 1$). If it is, then the pivot will be the k -th smallest element in \mathbf{a} , and the method can terminate at this base case. This scenario is illustrated in [Figure 5.8\(a\)](#).

In the recursive case the algorithm reduces the size of the problem by considering either the sublist to the left, or to the right, of the pivot. If $i_p > k - 1$, then the location of the searched element will be smaller than i_p . Thus, the algorithm can focus on the sublist to the left of the pivot, as shown in [Figure 5.8\(b\)](#). Instead, if $i_p < k - 1$ then the algorithm will proceed by solving the subproblem related to the list to the right of the pivot, as illustrated in (c). Finally, [Listing 5.14](#) contains an implementation of the tail-recursive function.

The runtime cost of the algorithm depends on where the pivot is located after performing the decomposition. If it is always located at the middle position of the list, the running time can be characterized by:

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

since the algorithm needs to solve a subproblem of approximately half the size of the original's, while the methods that perform the partition run in linear time. This is the best case scenario, where $T(n) \in \Theta(n)$. However, if the pivot is always located at an extreme of the list, then

the runtime cost is determined by:

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

which is a quadratic function (i.e., $T(n) \in \Theta(n^2)$). This situation corresponds to the worst case scenario for the algorithm.

5.6 BISECTION ALGORITHM FOR ROOT FINDING

Consider a continuous real-valued function $f(x)$ defined over some interval $[a, b]$. The bisection algorithm (also known as the interval halving, binary search, or dichotomy method) is a strategy for finding an approximation \hat{z} to a root of $f(x)$ in the (a, b) interval. Recall that a root, or zero, of $f(x)$ is a value r for which $f(r) = 0$.

The inputs to the problem are the extremes of the interval a and b , where $a \leq b$, the function f , and a small value $\epsilon > 0$ that determines the accuracy of the approximation. In particular, the output of the function \hat{z} will satisfy $|\hat{z} - r| \leq \epsilon$. Thus, the smaller ϵ is, the better the approximation will be. In addition, another prerequisite over the inputs is that $f(a)$ and $f(b)$ must have opposite signs. This guarantees that there will exist a root inside the interval (a, b) . Lastly, the approximation \hat{z} is defined as the midpoint between a and b . In other words, $\hat{z} = (a + b)/2$.

The method works by progressively dividing the interval by two where the root must exist. [Figure 5.9](#) illustrates a few steps of the process. In the initial setting the root is located to the left of the approximation ($r < \hat{z}$). Therefore, in the next step b is updated to \hat{z} , which halves the interval. Similarly, at step 1, $\hat{z} < r$, and a is assigned the value of \hat{z} in the following step. This procedure is applied repeatedly until the interval is small enough, guaranteeing that \hat{z} meets the required accuracy.

The size of the problem depends on the length of the interval $[a, b]$, and the base case occurs when $b - a \leq 2\epsilon$, where the method returns $\hat{z} = (a + b)/2$. Observe that this condition guarantees $|\hat{z} - r| \leq \epsilon$, as illustrated in [Figure 5.10](#). In particular, if \hat{z} is the midpoint of the interval $[a, b]$, then the distance between \hat{z} and r cannot be greater than ϵ . In addition, the method can also return \hat{z} trivially if $f(\hat{z}) = 0$ (or if $f(\hat{z})$ is sufficiently small).

The decomposition of the problem involves dividing its size by two. This is accomplished by replacing one of the extremes of the interval $[a, b]$ by its midpoint \hat{z} . Since the signs of the new extremes must be

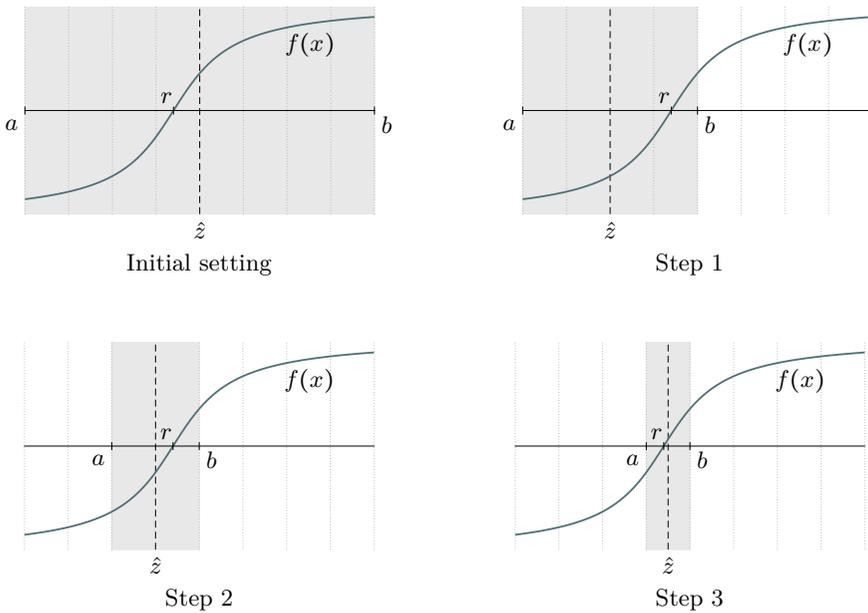


Figure 5.9 Steps of the bisection algorithm.

different, \hat{z} should replace b when $f(a)$ and $f(\hat{z})$ have opposite signs. Otherwise, \hat{z} will replace a . This leads to the tail-recursive method in Listing 5.15, which contains two recursive cases. The code uses the function $f(x) = x^2 - 2$ that allows us to find an approximation of $\sqrt{2}$, since it is a root of $f(x)$. Note that the initial interval $[0, 4]$ contains $\sqrt{2}$. Finally, the error of the approximation will be less than or equal to 10^{-10} (the result will be accurate up to nine decimal digits to the right of the decimal point).

5.7 THE WOODCUTTER PROBLEM

In this problem a woodcutter needs to collect w units of wood from a forest composed of n trees. He has a special machine that is able to cut the trees at some height h , where he would gather the wood from the trees above h . Assume that w is an integer (not greater than all of the wood in the forest), and that the heights of the n trees, which are also integers, are specified through a list \mathbf{t} . The goal of the problem is to find the largest value of h that will allow him to collect at least w units of wood. Figure 5.11 shows an instance of the problem for $n = 20$

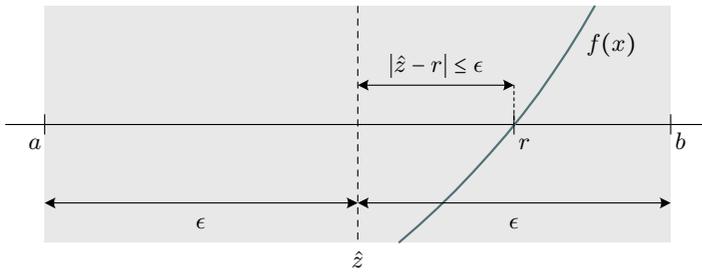


Figure 5.10 Base case of the bisection algorithm ($b - a \leq 2\epsilon$).

Listing 5.15 Bisection algorithm.

```

1 def f(x):
2     return x * x - 2
3
4
5 def bisection(a, b, f, epsilon):
6     z = (a + b) / 2
7
8     if f(z) == 0 or b - a <= 2 * epsilon:
9         return z
10    elif (f(a) > 0 and f(z) < 0) or (f(a) < 0 and f(z) > 0):
11        return bisection(a, z, f, epsilon)
12    else:
13        return bisection(z, b, f, epsilon)
14
15
16 # Print an approximation of the square root of 2
17 print(bisection(0, 4, f, 10**(-10)))

```

trees, where the tallest tree has height $H = 12$. If the goal is to collect 10 units of wood, then the woodcutter should set the height of the cutting machine to $h = 8$, where the total wood collected would be exactly 10 units. If the goal were to collect 7, 8, or 9 units of wood the optimal height h would also be 8. Even though the woodcutter would obtain more wood than initially required, h cannot be higher since cutting at height 9 only provides 6 units of wood.

The problem is interesting from an algorithmic point of view since it can be solved in several ways. For example, the trees can be initially sorted in decreasing order by their height, and subsequently processed from the highest to the lowest until obtaining the optimal height. This

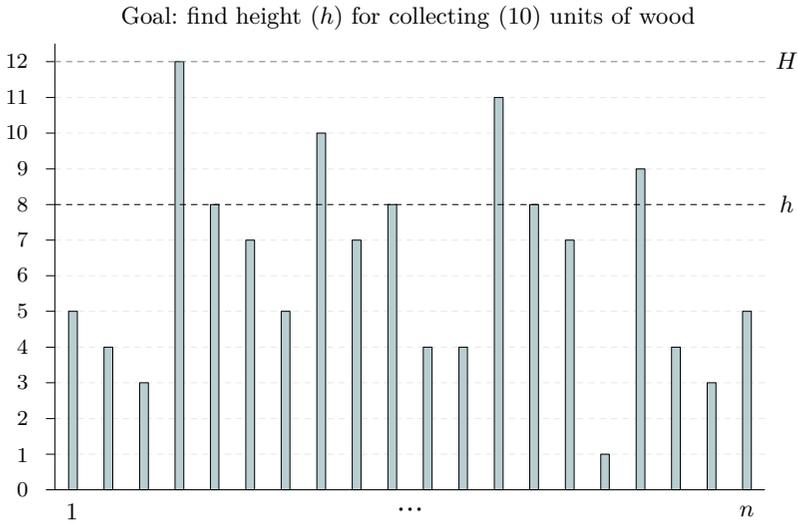


Figure 5.11 Instance of the woodcutter problem.

Listing 5.16 Function that computes the amount of wood collected when cutting trees at height h .

```

1 def compute_wood(t, h):
2     if t == []:
3         return 0
4     else:
5         if t[0] > h:
6             return t[0] - h + compute_wood(t[1:], h)
7         else:
8             return compute_wood(t[1:], h)

```

approach would run in $O(n \log n)$ if we apply general sorting algorithms like merge sort or quicksort (see [Chapter 6](#)). Since the heights are integers it is possible to apply linear time sorting algorithms like the counting sort method (see [Exercise 5.3](#)), which would lead to a solution that runs in $O(n+H)$. Thus, if H were small this approach could be more efficient.

We will now present a “binary search” solution that runs in $O(n \log H)$, which does not rely on sorting the heights of the trees. Firstly, note that a basic approach consists of starting with $h = H - 1$, and then decreasing h unit by unit until obtaining the required amount of wood.

For simplicity, assume that the method calculates the amount of wood obtained at each new height independently of the amount gathered for other heights. This implies that computing the amount of wood collected from n trees requires on the order of n operations. Therefore, in total, the algorithm for solving the problem would require $O(nH)$ operations in the worst case (at each of the possible H heights the algorithm would need to carry out n computations). In particular, the amount of wood collected at a certain height h can be obtained through the linear-recursive function in Listing 5.16 (Exercise 11.2 proposes implementing a tail-recursive version of the function). The method returns zero in the base case if the list of tree heights is empty. The recursive case processes the first tree height from the list. If it is greater than h the tree will be cut, and the function must return the difference between the height of the tree t_0 and h , plus the wood collected from the rest of the trees (through the recursive call). If the tree's height is less than or equal to h , the tree will not be cut, and the method can simply return the total wood gathered from the remaining trees.

Instead of decreasing h one unit at a time, the following algorithm uses a strategy similar to a binary search of an element in a list, or to the bisection method. The idea is to start searching for the height h in the middle of the interval $[0, H]$, and progressively halving it until a decision can be taken. Thus, the method can receive the tree heights and specified wood as parameters, together with lower and upper limits that indicate where the solution h can be found.

The size of the problem is the difference between the upper and lower limits (which is initially H). A first base case occurs if they share the same value. In that case, the output height h will be precisely that value. Otherwise the algorithm will compute a “middle” height, as the (integer) average of the lower and upper limits. If the total amount of wood that would be gathered from cutting the trees at this middle height is equal to the goal w , then the method can return such middle height in another base case.

Figure 5.12 shows the decomposition of the problem used by the algorithm. The original problem is shown in the top image, given the lower and upper indices. After computing the wood that would be collected at the middle height (h_m), there are two possible scenarios. Firstly, if it is greater than w we can clearly rule out all of the heights below h_m . Thus, we can continue searching for the optimal height (through a recursive call) by replacing the lower limit with h_m . It is important to note that h_m could still be the solution, since the wood gathered for $h_m + 1$ may

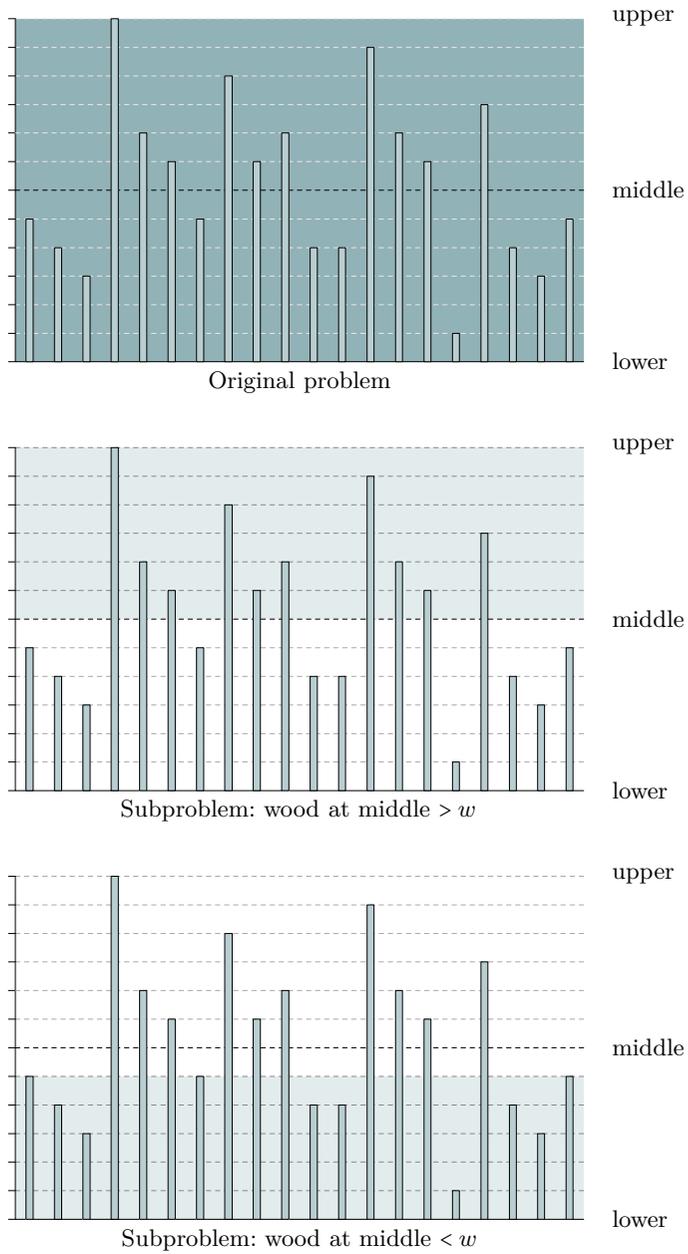


Figure 5.12 Decomposition of the woodcutter problem.

Listing 5.17 Binary search algorithm for the woodcutter problem.

```

1 def collect_wood(t, wood, lower, upper):
2     middle_h = (lower + upper) // 2
3     wood_at_middle = compute_wood(t, middle_h)
4
5     if wood_at_middle == wood or lower == upper:
6         return middle_h
7     elif lower == upper - 1:
8         if compute_wood(t, upper) >= wood:
9             return upper
10        else:
11            return lower
12    elif wood_at_middle > wood:
13        return collect_wood(t, wood, middle_h, upper)
14    else:
15        return collect_wood(t, wood, lower, middle_h - 1)

```

be less than the required amount w . A second scenario occurs when the wood collected at h_m is less than w . In that case we can discard all of the heights that are greater than or equal to h_m , since the woodcutter would not obtain enough wood. The associated recursive case would therefore invoke the method by replacing the upper limit with $h_m - 1$.

Lastly, in the previous decomposition, when the upper limit is just a unit greater than the lower one, the middle height is equal to the lower one. In that situation the first recursive case would not reduce the size of the problem (the limits would not vary). Thus, the algorithm needs an additional base case in order to work properly. When this situation occurs the method has to return either the lower or the upper limit. In particular, if the amount of wood associated with the upper limit is greater than or equal to w , then the upper limit is the correct result. Otherwise, it will be the lower one.

Listing 5.17 shows the code related to the mentioned base and recursive cases. Lastly, Figure 5.13 shows the steps it takes in order to solve the instance in Figure 5.11. Step 1 shows an initial situation where the lower limit is zero and the upper one is $H = \max\{t_i\}$, for $i = 1, \dots, n$. Step 2 and 3 are associated with applying the first and second recursive cases, respectively. Finally, step 4 applies the additional base case since the upper limit (8) is just a unit over the lower one (7). The solution is $h = 8$, since for that height the woodcutter obtains the requested amount of wood (10 units).

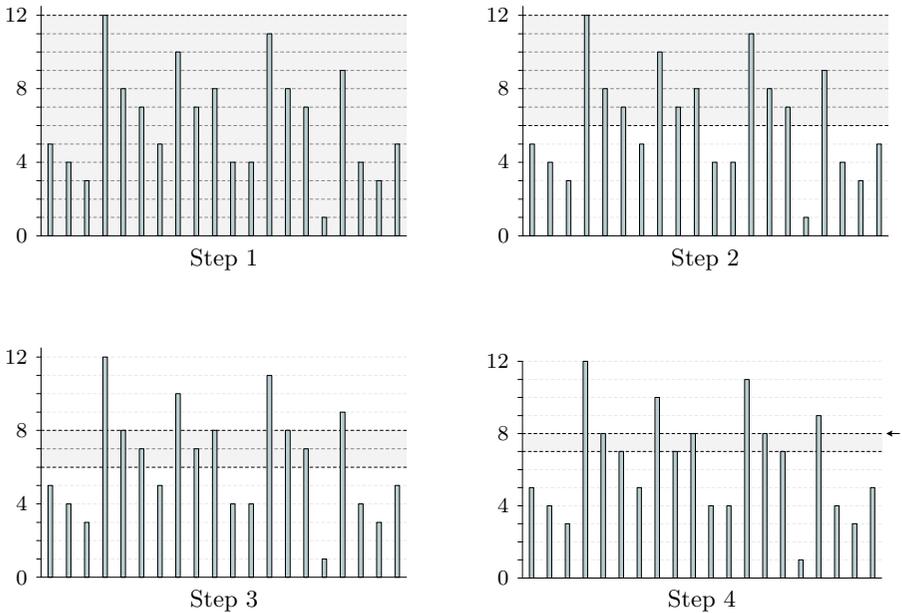


Figure 5.13 Steps of the binary search algorithm related to an instance of the woodcutter problem, for $w = 10$.

5.8 EUCLID'S ALGORITHM

One of the first algorithms developed throughout history is known as Euclid's algorithm, named after the ancient Greek mathematician Euclid, who first described it in his *Elements* (c. 300 BC). Its purpose is finding the greatest common divisor (gcd) of two nonnegative integers m and n (where both cannot be zero), which is the largest positive integer k that divides both m and n without a remainder (i.e., m/k and n/k are integers). For example, the greatest common divisor of 20 and 24 is 4. It can also be understood as the product of the common prime factors of m and n . For $m = 20 = 2 \cdot 2 \cdot 5$, and $n = 24 = 2 \cdot 2 \cdot 2 \cdot 3$, the product of common factors is $2 \cdot 2 = 4$.

There are several versions of the method. The original one corresponds to the following function:

$$gcd_1(m, n) = \begin{cases} n & \text{if } m = 0, \\ gcd_1(n, m) & \text{if } m > n, \\ gcd_1(m, n - m) & \text{otherwise,} \end{cases} \quad (5.2)$$

Listing 5.18 Euclid's algorithm for computing the greatest common divisor of two nonnegative integers.

```

1 def gcd1(m, n):
2     if m == 0:
3         return n
4     elif m > n:
5         return gcd1(n, m)
6     else:
7         return gcd1(m, n - m)

```

which can be coded as shown in Listing 5.18. Firstly, it is tail-recursive since it returns the value of a function call in the recursive cases. In addition, observe that the method returns the value of a parameter in the base case, which is common in many tail-recursive functions. In particular, the greatest common divisor of $n > 0$ and 0 is clearly n , since n divides both n and 0 without a remainder. The first recursive case simply switches the order of the arguments. This guarantees that the first one will be larger than or equal to the second one when reaching the second recursive case. In this last recursive case reducing the size of the problem (which depends on m and n) by a unit, or dividing it by two, does not work. Instead, the recursive case decreases the second argument by subtracting m (observe that $n - m \geq 0$, since $n \geq m$). It implements the mathematical property:

$$\gcd(m, n) = \gcd(m, n - m), \quad (5.3)$$

which is not immediately obvious. It can be shown to be true as follows. Assume $n \geq m$, and let $m = az$, and $n = bz$, where $z = \gcd(m, n)$ represents the product of the common prime factors in n and m , and $a \leq b$. This implies that a and b do not share common prime factors. In addition, let $b = a + c$. In that case we can express n and m as:

$$n = bz = (a + c)z = (a_1 \cdots a_k + c_1 \cdots c_l)z, \quad (5.4)$$

$$m = (a_1 \cdots a_k)z, \quad (5.5)$$

where a_1, \dots, a_k and c_1, \dots, c_l are the prime factors of a and c , respectively. The key to the proof is the fact that a and c cannot share prime factors (i.e., $a_i \neq c_j$), because if they did they could be pulled out of the parenthesis in (5.4), implying that a and b share those common prime factors, which is a contradiction. If a and c do not share prime factors

then $z = \text{gcd}(az, cz)$, and we can conclude that:

$$\text{gcd}(az, bz) = z = \text{gcd}(az, cz) \Rightarrow \text{gcd}(m, n) = \text{gcd}(m, n - m).$$

In addition, it can also be shown that the algorithm is guaranteed to reach the base case, since the values of the arguments decrease until one of them is 0. For $m = 20$ and $n = 24$ the method carries out the following recursive calls:

$$\begin{aligned} & \text{gcd}_1(20, 24) \\ &= \text{gcd}_1(20, 4) \\ &= \text{gcd}_1(4, 20) \\ &= \text{gcd}_1(4, 16) \\ &= \text{gcd}_1(4, 12) \\ &= \text{gcd}_1(4, 8) \\ &= \text{gcd}_1(4, 4) \\ &= \text{gcd}_1(4, 0) \\ &= \text{gcd}_1(0, 4) = 4. \end{aligned} \tag{5.6}$$

Since tail-recursive functions do not manipulate the result of recursive calls, all of them return the same value, which is obtained at the base case ($\text{gcd}_1(0, 4) = 4$). Therefore, tail-recursive functions essentially define relationships between sets of arguments. For example, in (5.6) the pairs $(20, 24)$, $(20, 4)$, and so on, are all related to the value 4. Lastly, a more efficient version that is still used today is:

$$\text{gcd}_2(m, n) = \begin{cases} n & \text{if } m = 0, \\ \text{gcd}_2(n \% m, m) & \text{if } m \neq 0. \end{cases} \tag{5.7}$$

The proof of the property in the recursive case is similar to the one for (5.3). Assume $n \geq m$, and let $m = az$, and $n = bz$, where z is the product of the common prime factors in n and m , and $a \leq b$. This implies that a and b do not share common prime factors. In addition, let $b = qa + r$, where q and r are the quotient and remainder of the division b/a . The key in this proof is that a and r cannot share common prime factors, since otherwise they would also be prime factors of b , and this is not possible since a and b do not share common prime factors. This implies that $z = \text{gcd}(rz, az)$. Thus, we can conclude that:

$$\text{gcd}(az, bz) = z = \text{gcd}(rz, az) \Rightarrow \text{gcd}(m, n) = \text{gcd}(n \% m, m).$$

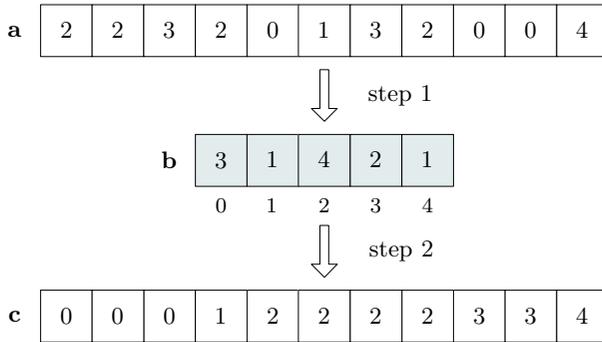


Figure 5.14 Steps in the counting sort algorithm.

5.9 EXERCISES

Exercise 5.1 — Define and code Boolean linear and tail-recursive functions that determine whether a nonnegative integer n contains an odd digit.

Exercise 5.2 — It is more pleasant to display polynomials in a formatted style rather than through a list. Write a method that receives a list of length n that represents a polynomial of degree $n - 1$, and prints it in a formatted style. For example, given the input list $\mathbf{p} = [3, -5, 0, 1]$, corresponding to $x^3 - 5x^1 + 3$ (i.e., p_i is the coefficient associated with the term x^i), the method should print a line similar to: $+ 1x^3 - 5x^1 + 3$. Assume that $p_{n-1} \neq 0$, except if the polynomial is the constant 0. Finally, specify its asymptotic computational cost.

Exercise 5.3 — The “counting sort” algorithm is a method for sorting a list \mathbf{a} of n integers in the range $[0, k]$, where k is usually small. The method runs in $\mathcal{O}(n + k)$ time, which implies that it runs in $\mathcal{O}(n)$ time if $k \in \mathcal{O}(n)$.

Given \mathbf{a} , the method creates a new list \mathbf{b} that contains the number of occurrences of each integer in \mathbf{a} , as shown in step 1 of Figure 5.14. For example, $b_2 = 4$, since the integer 2 appears four times in \mathbf{a} . Implement a tail-recursive procedure that receives lists \mathbf{a} and \mathbf{b} (initialized with zeros), and fills list \mathbf{b} with the occurrences of the integers in \mathbf{a} .

In addition, implement a linear-recursive function that receives the list of integer occurrences \mathbf{b} , and returns a new list (\mathbf{c}) that is the sorted version of \mathbf{a} , as shown in step 2 of Figure 5.14.

Finally, implement a function that calls the previous two methods in order to implement a version of the counting sort algorithm, and specify its asymptotic computational cost.

Exercise 5.4 — Implement an alternative version of [Listing 5.8](#) that does not use the `upper` parameter. Finally, specify its asymptotic computational cost.

Exercise 5.5 — Implement a Boolean “binary search” function in Python that simply indicates whether an element x appears in a list \mathbf{a} . The function will have only two input parameters: x and \mathbf{a} , and will be based on decomposing the problem into another one of half its size. Finally, specify its asymptotic computational cost.

Exercise 5.6 — Write a function that searches for the item with the smallest key in a binary search tree T , defined as a list of four components, as described in [Section 5.3](#).

Exercise 5.7 — Let \mathbf{a} be a sorted list (in ascending order) of *different* integers. The goal of this problem is to *efficiently* search for an element that is equal to its position (i.e., index) along the list. In other words, we want to find an element i for which $a_i = i$. For example, if $\mathbf{a} = [-3, -1, 2, 5, 6, 7, 9]$, then the output of the method will be 2, since $a_2 = 2$. Note that the first element of the list is located at position 0. For simplicity, assume that \mathbf{a} will have at most one element that satisfies $a_i = i$. If the list does not contain any element that satisfies that property the function will return the value -1 . Finally, specify its asymptotic computational cost.

Exercise 5.8 — Let \mathbf{a} be a list of n integers, arranged so that even numbers appear before odd numbers (the index of any even number will be smaller than the index of any odd number), like, for example, $\mathbf{a} = [2, -4, 10, 8, 0, \mathbf{12}, 9, 3, -15, 3, 1]$. The goal of the following problem is to develop an efficient recursive algorithm for determining the largest index associated with the even numbers. In other words, the index i for which a_i is even, but a_j will be odd for any $j > i$. For the example the method would return $i = 5$, since $a_5 = 12$ is even, but $a_6 = 9$ is odd (and so are a_7, a_8 , and so on). If \mathbf{a} does not contain even numbers the result will be -1 , while if it is composed entirely of even numbers the output will be $n - 1$. Finally, specify its asymptotic computational cost.

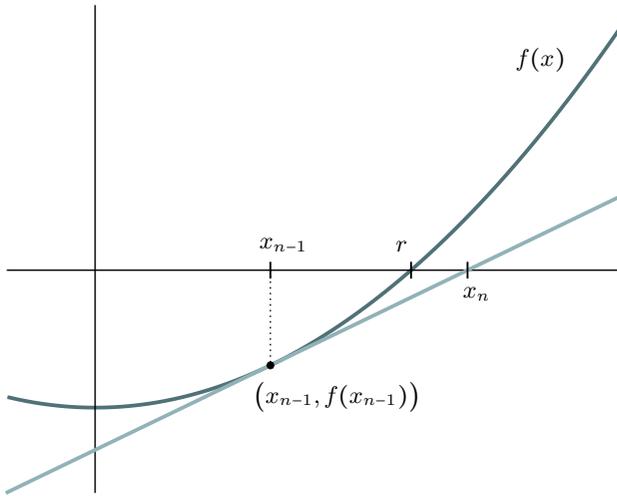


Figure 5.15 Main idea behind Newton's method.

Exercise 5.9 — Newton's method is an approach for finding successively better approximations to the roots of a real-valued function $f(x)$ that is differentiable. It is based on the following recursive rule:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})},$$

which is explained geometrically in [Figure 5.15](#). Say we have an initial approximation x_{n-1} to the root r of function $f(x)$. The procedure finds the tangent line to $f(x)$ at x_{n-1} , and computes a new approximation (x_n) to r that is the value of x where the line crosses the X -axis (note that x_n is closer to r). Thus, starting with some initial value x_0 , $\hat{z} = x_n$ will be the approximation of the root of $f(x)$ after applying the recursive rule n steps.

This procedure can be used, for example, to obtain very accurate approximations to the square root of a number, in only a few steps. Say we want to calculate \sqrt{a} , which is some unknown value x . In that case we have the following identities: $x = \sqrt{a}$, where $x^2 = a$, which implies that $x^2 - a = 0$. Therefore, the root of the function $x^2 - a$ will be the square root of a . In that case, the recursive formula associated with Newton's method is:

$$x_n = x_{n-1} - \frac{x_{n-1}^2 + a}{2x_{n-1}}. \quad (5.8)$$

Implement linear and tail-recursive functions that receive the value a , an initial positive estimate x_0 of \sqrt{a} , and a certain number of steps n , and return the final estimate $\hat{z} = x_n$ of \sqrt{a} by applying (5.8) n times. Finally, specify their asymptotic computational cost.

Multiple Recursion I: Divide and Conquer

Διαίρει και βασίλευε (divide and conquer).

— Philip II of Macedon

THE advantages of recursion over iteration, such as code clarity or avoiding managing a stack explicitly (see [Section 10.3.5](#)), are mainly due to the possibility of using multiple recursion. Methods based on this type of recursion invoke themselves several times in at least one recursive case. Therefore, these algorithms solve more than one simpler self-similar problem, and must combine, extend, and/or modify their results in order to obtain the solution to the original problem.

The book devotes three chapters to multiple recursion, and contains additional examples throughout it as well. In this chapter we will cover an important class of algorithms based on multiple recursion that decompose problems by dividing their size by some constant. These algorithms are said to follow the “divide and conquer” approach, which is one of the most important algorithm design paradigms. It can be viewed as a general problem solving strategy that consists of breaking up a problem into several self-similar subproblems whose size is a fraction of the original problem’s size. The approach is therefore closely related to recursion (iterative algorithms can also follow the divide and conquer strategy), since it relies on recursive problem decomposition (see [Figure 1.4](#)). The function in [Listing 1.2](#) and the third method of [Listing 1.5](#) constitute examples of this algorithm design approach.

In some references the term “divide and conquer” is also applied to algorithms where the decomposition is based on a single subproblem

Listing 6.1 Function that determines whether a list is sorted in ascending order.

```

1 def is_list_sorted(a):
2     n = len(a)
3     if n <= 1:
4         return True
5     else:
6         return (is_list_sorted(a[0:n // 2])
7                 and a[n // 2 - 1] <= a[n // 2]
8                 and is_list_sorted(a[n // 2:n]))

```

of half the size of the original's (the method would invoke itself only once). However, many authors consider that the term should only be used when the solution relies on breaking up a problem into two or more subproblems. This book adopts this last convention. Thus, the methods that invoke themselves once, dividing the size of the problem by two, have been covered in earlier chapters (mostly in [Chapter 5](#)). The following sections describe classical recursive divide and conquer algorithms.

6.1 IS A LIST SORTED IN ASCENDING ORDER?

In this problem the input is a list \mathbf{a} of n items that can be sorted through the \leq operator (or some function that allows us to compare elements by implementing a total order binary relation), and the output is a Boolean value that can be expressed as:

$$f(\mathbf{a}) = \bigwedge_{i=0}^{n-2} (a_i \leq a_{i+1}). \quad (6.1)$$

Thus, the result is **True** when the elements of the list appear in (non-strictly) increasing or nondecreasing order.

The size of this problem is the number of elements in the list. If the list contains one element the result is trivially **True**. In addition, we can consider that an empty list is also ordered.

The problem can be solved by a linear-recursive method that reduces the size of the problem one unit in the decomposition stage. However, we will present a solution that decomposes the problem by dividing the input list in two halves, and which leads to the following partition of

(6.1):

$$\begin{aligned}
 f(\mathbf{a}) = & \underbrace{(a_0 \leq a_1) \wedge \cdots \wedge (a_{\lfloor n/2 \rfloor - 2} \leq a_{\lfloor n/2 \rfloor - 1})}_{\text{Subproblem 1}} \\
 & \wedge (a_{\lfloor n/2 \rfloor - 1} \leq a_{\lfloor n/2 \rfloor}) \wedge \underbrace{(a_{\lfloor n/2 \rfloor} \leq a_{\lfloor n/2 \rfloor + 1}) \wedge \cdots \wedge (a_{n-2} \leq a_{n-1})}_{\text{Subproblem 2}}.
 \end{aligned}$$

Clearly, if a list is sorted in ascending order then both halves must also be sorted in ascending order. Thus, the method can invoke itself twice with the two corresponding sublists, and perform a logical AND operation with their result. Finally, the combination of the results of the subproblems needs an additional step. In particular, the last element of the first sublist ($a_{\lfloor n/2 \rfloor - 1}$) must be less than or equal to the first element of the second sublist ($a_{\lfloor n/2 \rfloor}$). This condition also requires another AND operation in the recursive case. Listing 6.1 shows a possible implementation of the method, which runs in $O(n)$ time, since its cost is characterized by:

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

6.2 SORTING

Sorting a general list (or array, sequence, etc.) is one of the most studied problems in computer science. It can be solved by numerous algorithms that can be used to introduce essential concepts related to computational complexity and runtime analysis, algorithm design paradigms, or data structures. The sorting algorithms covered in this chapter will assume that the input to the problem is a list \mathbf{a} of n real numbers. The output is a rearrangement (or permutation) of the elements that provides another list \mathbf{a}' where $a'_i \leq a'_{i+1}$, for $i = 0, \dots, n - 2$ (\leq can be thought of as Boolean function that allows us to determine if an element precedes another, implementing a total order binary relation). The choice of real numbers for the type of the elements of the list implies that the algorithms need to carry out comparisons through \leq . It can be shown that any algorithm that solves this problem requires $\Omega(n \log n)$ comparisons (i.e., at least on the order of $n \log n$ decisions using \leq). Lastly, there exist algorithms for sorting lists that do not need to compare elements, which require $\Omega(n)$ operations. However, the elements of the list must satisfy certain conditions in order to apply them. For example, the “counting

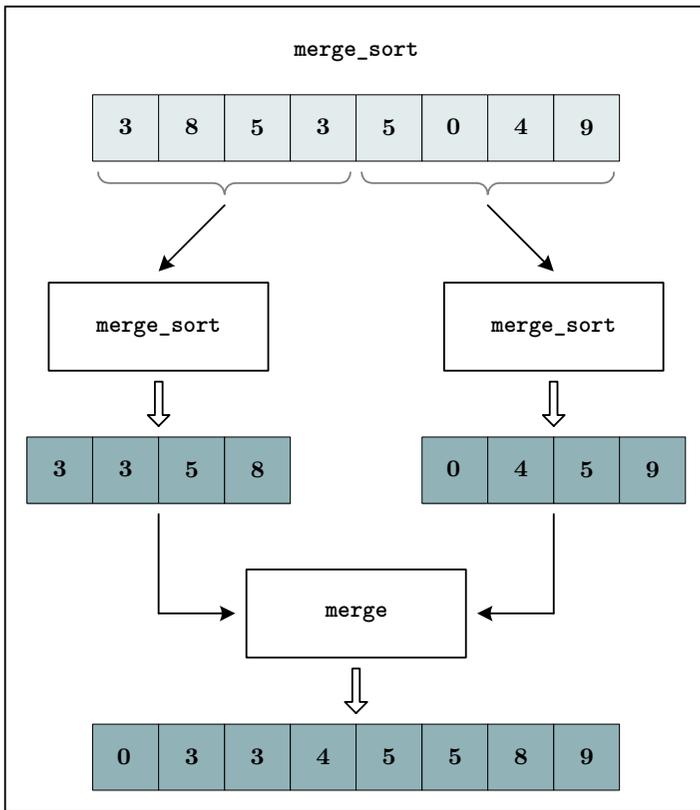


Figure 6.1 Merge sort algorithm.

sort” algorithm sorts integers that belong to a small interval $[0, k]$ (see [Exercise 5.3](#)).

6.2.1 The merge sort algorithm

The merge sort algorithm is one of the most extensively used examples to show the potential of the divide and conquer approach. While many sorting algorithms run in $\mathcal{O}(n^2)$ time in the worst case (see [Section 4.4.1](#) and [Exercise 4.17](#)), the merge sort algorithm runs in $\Theta(n \log n)$. In other words, its efficiency is optimal from a computational complexity perspective.

The size of the problem is the length of the list (n). The smallest instances occur when $n \leq 1$, where the algorithm can simply return the input list. In the recursive case the algorithm decomposes the problem by

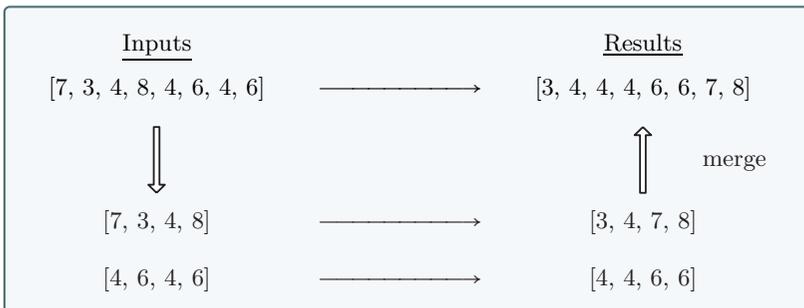
Listing 6.2 Merge sort method.

```

1 def merge_sort(a):
2     n = len(a)
3     if n <= 1:
4         return a
5     else:
6         a1 = merge_sort(a[0:n // 2])
7         a2 = merge_sort(a[n // 2:n])
8         return merge(a1, a2)

```

dividing the input list in two halves, giving rise to two different subproblems of (roughly) half the size of the original's. The following diagram shows a concrete example of the recursive thought process associated with the algorithm:



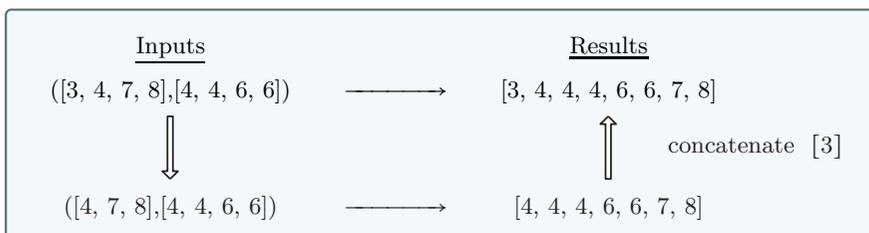
The recursive case is more complicated than in previous examples, since the final sorted list cannot be obtained through a simple operation (e.g., addition, computing the maximum of two numbers, concatenating two lists, etc.). In this case, it is necessary to solve a new problem in order to combine the solutions to the subproblems. In particular, the recursive case needs to “merge” two sorted lists (the outputs of the subproblems) in order to produce the final sorted list. Assuming we have implemented a function `merge` that solves this problem (see below), the succinct code in [Listing 6.2](#) implements the merge sort method. Note the readability of the code and the simplicity of the algorithm, which conforms perfectly to the divide and conquer strategy.

The runtime cost for this algorithm is:

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

where we have also assumed that the partition of the list can be obtained in a constant number of operations. In other words, we have considered that it is possible to obtain $a[0:n//2]$ and $a[n//2:n]$ in $\Theta(1)$ time. In addition, $f(n)$ measures the number of operations needed by the `merge` method in order to combine two sorted lists of (approximately) $n/2$ elements. In this case, due to the master theorem (see (3.28)), if $f(n)$ were a linear function, the merge sort algorithm would run in the (optimal) order of $\Theta(n \log n)$. We will see shortly that indeed it is possible to solve the merge problem in linear time. In general, when faced with a similar problem, algorithm designers focus their efforts on developing the most efficient combination method, which may reduce the order of growth of the divide and conquer algorithm.

The inputs to the merging problem are two sorted lists \mathbf{a} and \mathbf{b} , of lengths n_a and n_b , respectively. The output is another sorted list of length $n = n_a + n_b$. We can interpret that the size of the problem is the number of operations needed by the algorithm until it can return a trivial answer. In this scenario, the size of the problem is $m = \min(n_a, n_b)$, since the solution to the problem if one of the lists is empty is obviously the other list. This constitutes the base cases. For the recursive case we can use the following diagram with the sublists used in the previous example:



The decomposition reduces the size of the problem by a unit, by discarding the smallest element in either list. Naturally, it must appear at the initial position of the lists, since they are sorted in ascending order. In the example, the smallest element (3) is found in the first input list. Clearly, the recursive case simply needs to concatenate this element with the result of the subproblem. Listing 6.3 shows the corresponding code. Its runtime cost is characterized by the following function:

$$T(m) = \begin{cases} 1 & \text{if } m = 0, \\ T(m-1) + 1 & \text{if } m > 0, \end{cases}$$

assuming that the tail of a list (i.e., $\mathbf{x}[1:]$) can be obtained in constant time. Since its nonrecursive expression is $T(m) = m + 1$, the merge func-

Listing 6.3 Method for merging two sorted lists.

```

1 # lists a and b are sorted in ascending order
2 def merge(a, b):
3     if a == []:
4         return b
5     elif b == []:
6         return a
7     else:
8         if a[0] < b[0]:
9             return [a[0]] + merge(a[1:], b)
10        else:
11            return [b[0]] + merge(a, b[1:])

```

tion runs in linear time with respect to m , and also n , which implies that the merge sort algorithm runs in $\Theta(n \log n)$.

6.2.2 The quicksort algorithm

The quicksort algorithm is another method developed by Tony Hoare. It is based on the divide and conquer approach and receives its name due to its remarkable efficiency. Unlike the merge sort algorithm, its running time in the worst case is $O(n^2)$. However, in the best and average cases it runs in $\Theta(n \log n)$ time, and can be several times quicker than the merge sort algorithm in practice.

To understand the difference between both methods, observe that the decomposition in merge sort is easy. In particular, the input list can be divided in two halves by simply using appropriate index ranges ($0:n//2$ and $n//2:n$). However, combining the results of the subproblems requires solving yet another problem (merge) that is not immediately straightforward. In contrast, in the quicksort algorithm the decomposition is hard, but the combination stage is not only trivial, but may not even be necessary in some implementations.

Specifically, instead of simply dividing the list in two halves, quicksort's decomposition transforms the input by using a partitioning scheme like the ones presented in [Section 5.4](#). [Figure 6.2](#) illustrates this type of decomposition, where the sublists within the original list are separated by a pivot (the sublist to the right of the pivot contains elements that are less than or equal to the pivot, and the sublist to the left is composed of numbers that are greater than the pivot). A concrete recursive diagram based on the previous decomposition could be:

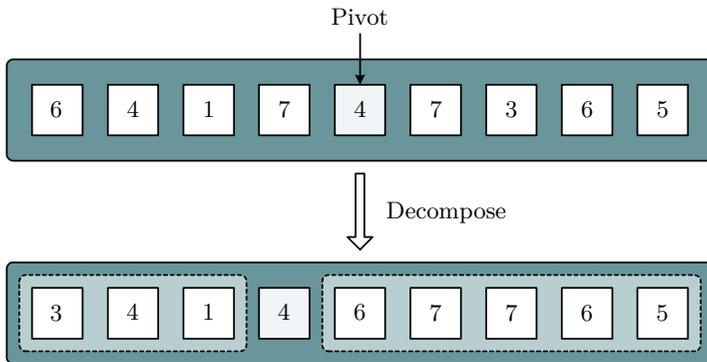
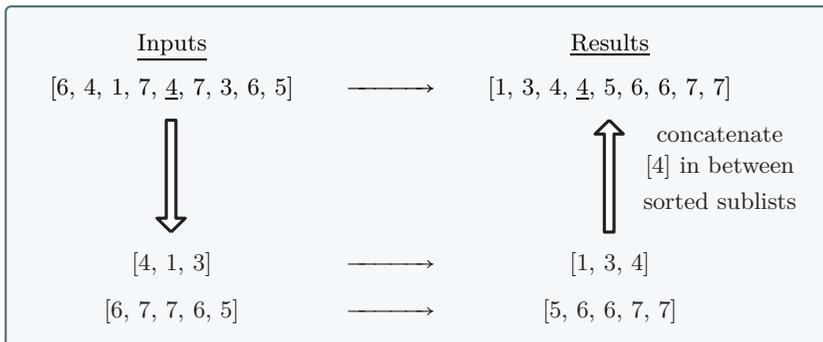


Figure 6.2 Decomposition of the quicksort algorithm.



The diagram clearly shows that sorting the original list simply requires solving the two subproblems through recursive calls, and concatenating the sorted sublists while maintaining the pivot in between them. Thus, after solving the subproblems recursively, the combination of the respective solutions is straightforward. Lastly, an important detail regarding this decomposition is that the pivot is removed from the list that contains the elements that are smaller than or equal to it. This is necessary in order to guarantee that the size of the subproblem is indeed smaller than the size of the original problem.

Listing 6.4 implements a slower variant of the method that is based on the basic partitioning schemes described in Section 5.4.1. Firstly, it checks whether the input corresponds to the base case (which is the same as in the merge sort method). In the recursive case a common strategy is to consider that the pivot is the first element of the list. With this choice the worst case occurs when the input list is already sorted. Moreover, the algorithm would also perform poorly if the input

Listing 6.4 Variant of the quicksort algorithm.

```

1 def quicksort_variant(a):
2     n = len(a)
3     if n <= 1:
4         return a
5     else:
6         pivot = a[n // 2]
7         v1 = get_smaller_than_or_equal_to(a, pivot)
8         v1.remove(pivot)
9         v2 = get_greater_than(a, pivot)
10        return (quicksort_variant(v1) + [pivot]
11                + quicksort_variant(v2))

```

Listing 6.5 In-place quicksort algorithm.

```

1 def quick_sort_inplace(a, lower, upper):
2     if lower < upper:
3         pivot_index = partition_Hoare_wrapper(a, lower, upper)
4
5         quick_sort_inplace(a, lower, pivot_index - 1)
6         quick_sort_inplace(a, pivot_index + 1, upper)

```

is almost sorted. Since these situations may occur frequently in practice, the function uses the middle element of the input list instead as the pivot. Subsequently, the method obtains the two sublists associated with the decomposition (removing the pivot from the sublist that contains it), and finally concatenates the results of the subproblems, leaving the pivot in between them.

The merge sort method in [Listing 6.2](#), and the quicksort function in [Listing 6.4](#), do not alter the input list and return their result in a new list, which requires twice as much storage space. These types of algorithms are denoted as “out-of-place.” Instead, it is possible to implement variants that overwrite the input list, which do not need to allocate storage space for a full list of length n . These algorithms are known as “in-place.” [Listing 6.5](#) shows an in-place variant of the quicksort algorithm that uses lower and upper limits to indicate the boundaries of a sublist within the original one. The code includes the recursive version of Hoare’s partitioning method described in [Listing 5.13](#) (naturally, the iterative version, which is more efficient, can be used as well), which carries out the decomposition of the problem.

The runtime cost analysis of the algorithm is similar to that of the quickselect algorithm. The main difference is that, instead of solving one subproblem, quicksort invokes itself twice. The best case occurs when the pivot is always located at the middle position of the list. In that case, the running time can be characterized by:

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

where $T(n) \in \Theta(n \log n)$. Instead, the worst case occurs when the pivot is always located at an extreme of the list. In that case the runtime cost is determined by:

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

which is a quadratic function ($T(n) \in \Theta(n^2)$).

6.3 MAJORITY ELEMENT IN A LIST

A list is said to have a “majority element” if *more* than half of its entries are identical. In this classical problem the goal consists of determining whether a list **a** of length n has a majority element, and, if so, to find that element. In addition, the problem assumes that the elements can only be compared in $\Theta(1)$ (constant) time by using the equality (`==`) operator. In other words, we are not allowed to compare them through relations such as `<`, `>`, `<=`, or `>=` (the examples will use integers, but not those operators).

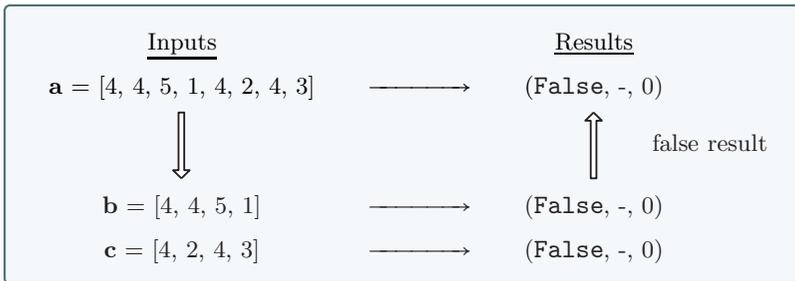
There are several divide and conquer variants that solve the problem. We will consider a method that returns a tuple (it could also be a list) containing three values that indicate:

- a) Whether the list contains a majority element (Boolean).
- b) The majority element if it exists. Otherwise this value is irrelevant (the function could simply return `None`).
- c) The number of occurrences of the majority element (0 if it does not exist).

The size of the problem is n . A first base case can correspond to an empty list, where the result would be the tuple (`False`, x , 0). The value

x would be irrelevant. Additionally, if the list contains one element then the function would return $(\text{True}, a_0, 1)$.

The recursive case can rely on decomposing the list in two halves, and solving the corresponding subproblems. We can form initial recursive diagrams with concrete examples in order to understand how we can combine the solutions to the subproblems. For example:

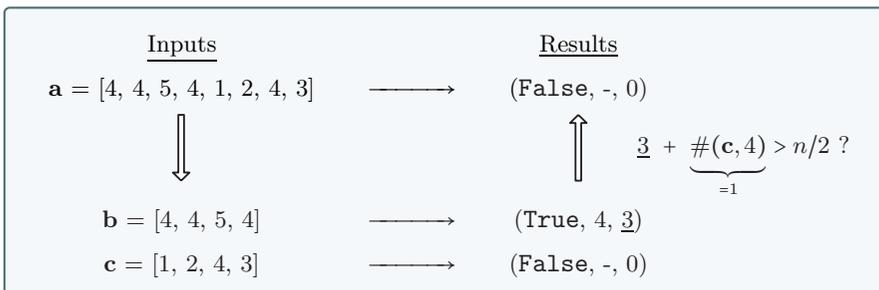


In this case the result of both subproblems is **False**, which implies that the input list cannot contain a majority element (even though there are exactly $n/2$ occurrences of the element 4 in the example, it is not enough to produce a true result). In general, this can be shown as follows. Firstly, the initial list is divided into a sublist \mathbf{b} of length $\lfloor n/2 \rfloor$, and another sublist \mathbf{c} of length $\lceil n/2 \rceil$, regardless of whether n is even or odd, since $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If the result for both subproblems is **False** then an element can appear at most $\lfloor \lfloor n/2 \rfloor / 2 \rfloor$ times in \mathbf{b} , and $\lfloor \lceil n/2 \rceil / 2 \rfloor$ times in \mathbf{c} . Adding these quantities yields:

$$\left\lfloor \frac{\lfloor n/2 \rfloor}{2} \right\rfloor + \left\lfloor \frac{\lceil n/2 \rceil}{2} \right\rfloor \leq \left\lfloor \frac{\lfloor n/2 \rfloor + \lceil n/2 \rceil}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor \leq \frac{n}{2}.$$

Therefore, we can conclude that an element cannot appear more than $n/2$ times in the initial list.

Another concrete diagram could be:



Listing 6.6 Code for counting the number of times an element appears in a list.

```

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)

```

In this case the element 4 appears three times in the first sublist, and is therefore a majority element since $3 > n/2 = 2$. The algorithm must therefore count the number of occurrences of 4 in the second list (denoted through $\#(c, 4)$) in order to determine whether it is also a majority element of the initial list \mathbf{a} . This can be computed through a simple linear-recursive function (see Listing 6.6) that receives an input list \mathbf{a} and an element x . If \mathbf{a} is empty the result is obviously 0. Otherwise, the output can consist of the method applied to the tail of \mathbf{a} ($\mathbf{a}_{1..n-1}$) and x , plus a unit only if $a_0 = x$.

Listing 6.7 shows a possible implementation of the function that solves the majority element problem. Lines 3–6 code the base cases. Lines 8 and 9 decompose the input list into two halves. Line 11 invokes the method on the first sublist, and if there exists a majority element (line 12), then line 13 computes the number of occurrences of the element in the second sublist. If the total number of occurrences of the element (in both sublists) is greater than $n/2$ (line 14), then the method returns a tuple in line 15 with the values: `True`, the majority element, and the number of times it appears in the input list. Lines 17–21 are analogous, but switch the roles of the sublists. Finally, if the function has not returned, then the list does not contain a majority element (line 23).

In the recursive cases the method needs to invoke itself twice with one half of the input list, and also needs to compute the occurrences of an element on two sublists of length $n/2$ (approximately). Since this last auxiliary function runs in linear time, the time complexity of the method can be characterized by:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

Listing 6.7 Code for solving the majority element problem.

```

1 def majority_element_in_list(a):
2     n = len(a)
3     if n == 0:
4         return (False, None, 0)
5     elif n == 1:
6         return (True, a[0], 1)
7     else:
8         b = a[0:n // 2]
9         c = a[n // 2:n]
10
11        t = majority_element_in_list(b)
12        if t[0]:
13            occurrences = occurrences_in_list(c, t[1])
14            if t[2] + occurrences > n / 2:
15                return (True, t[1], t[2] + occurrences)
16
17        t = majority_element_in_list(c)
18        if t[0]:
19            occurrences = occurrences_in_list(b, t[1])
20            if t[2] + occurrences > n / 2:
21                return (True, t[1], t[2] + occurrences)
22
23        return (False, None, 0)

```

Therefore, the order of growth of the algorithm is $\Theta(n \log n)$ (see (3.28)). Lastly, this problem can be solved through the Boyer–Moore majority vote algorithm in linear time.

6.4 FAST INTEGER MULTIPLICATION

The classical algorithm taught in school to multiply two nonnegative n -digit integers requires n^2 digit-times-digit multiplications. In this section we will analyze Karatsuba’s algorithm, which is a faster approach. The method can be applied to numbers expressed in any base, but we will focus on multiplying binary numbers. In particular, let x and y be two nonnegative integers represented by b_x and b_y bits, respectively. Applying a divide and conquer approach, each binary number can be partitioned in two as follows:

$$\begin{aligned}
 x &= a \cdot 2^m + b, \\
 y &= c \cdot 2^m + d,
 \end{aligned}
 \tag{6.2}$$

where $m = \min(\lfloor b_x/2 \rfloor, \lfloor b_y/2 \rfloor)$. For example, for $x = 594$, and $y = 69$, the decomposition is:

$$\begin{aligned} x &= 1001010010_2 = \underbrace{1001010}_{a=74} \underbrace{010}_{b=2} = 74 \cdot 2^3 + 2, \\ y &= 1000101_2 = \underbrace{1000}_{c=8} \underbrace{101}_{d=5} = 8 \cdot 2^3 + 5, \end{aligned}$$

where $b_x = 10$, $b_y = 7$, $m = 3$, $a = 74$, $b = 2$, $c = 8$, and $d = 5$. Thus, the smaller number (in this case, y) is partitioned in two parts with (roughly) the same number of bits, and the numbers associated with the lower-significant parts (b and d) are expressed with the same number of bits.

In many programming languages it is possible to compute the values of a , b , c , and d by relying on bit shift operations. In Python they can be carried out through the `<<` and `>>` operators. On the one hand, $(x \ll m)$ is equivalent to $x2^m$, which shifts the bits of x m times towards the left, appending m least-significant zeros. On the other hand, $(x \gg m)$ performs $\lfloor x/2^m \rfloor$, shifting the bits of x m times towards the right, which discards those m bits. These bitwise operations are therefore useful for multiplying (or performing an integer division) by a power of two, and can be used as follows to decompose x and y according to (6.2):

$$\begin{aligned} a &= x \gg m, \\ b &= x - (a \ll m), \\ c &= y \gg m, \\ d &= y - (c \ll m), \end{aligned}$$

where the parentheses are necessary due to operator precedence rules.

According to the decomposition, the product of x and y can be written as:

$$xy = (a \cdot 2^m + b)(c \cdot 2^m + d) = ac2^{2m} + (ad + bc)2^m + bd. \quad (6.3)$$

This initial (naive) approach can break up the original problem (a multiplication) into four smaller subproblems: ac , ad , bc , and bd , which can be computed through four recursive calls (we can ignore the cost of multiplications times powers of two, since these can be implemented very efficiently as bit shifts). However, it is not more efficient than the

Listing 6.8 Karatsuba's fast algorithm for multiplying two nonnegative integers.

```

1 def number_of_bits(n):
2     if n < 2:
3         return 1
4     else:
5         return 1 + number_of_bits(n >> 1)
6
7
8 def multiply_karatsuba(x, y):
9     if x == 0 or y == 0:
10        return 0
11    elif x == 1:
12        return y
13    elif y == 1:
14        return x
15    else:
16        n_bits_x = number_of_bits(x)
17        n_bits_y = number_of_bits(y)
18
19        m = min(n_bits_x // 2, n_bits_y // 2)
20
21        a = x >> m
22        b = x - (a << m)
23        c = y >> m
24        d = y - (c << m)
25
26        ac = multiply_karatsuba(a, c)
27        bd = multiply_karatsuba(b, d)
28        t = multiply_karatsuba(a + b, c + d) - ac - bd
29
30    return (ac << (2 * m)) + (t << m) + bd

```

“school” method, since it also requires n^2 bit-times-bit multiplications for two n -bit numbers. The algorithm proposed by Karatsuba is able to reduce such quantity to approximately $n^{1.585}$ by rearranging the terms and including more addition/subtraction operations, which turn out to be negligible regarding asymptotic computational complexity. In particular, the product xy can be reformulated as:

$$xy = ac2^{2m} + [(a + b)(c + d) - ac - bd]2^m + bd, \quad (6.4)$$

which requires only three simpler products: ac , bd , and $(a + b)(c + d)$, leading to a faster algorithm that only carries out three recursive calls.

Listing 6.8 implements Karatsuba's method, which contains an auxiliary function that computes the number of bits of the binary representation of a nonnegative integer n (which is equal to $\lfloor \log_2 n \rfloor + 1$, for $n \geq 1$). The size of the problem can be $\min(b_x, b_y)$. Thus, the base cases can check if the inputs are equal to zero or one, which lead to trivial results. The recursive case implements (6.4), using three recursive calls.

Regarding its efficiency, consider that both inputs have the same number of bits n , which is a power of two (i.e., $n = 2^k$). We can make this assumption since it is always possible to append 0 leading bits to the input integers until the assumption is satisfied. In that case, the running time of the algorithm is characterized by:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 3T(n/2) + cn + d & \text{if } n > 1. \end{cases}$$

The term $cn + d$ is due to additions, subtractions, shift operations, and calls to `number_of_bits`, which are carried out in linear time with respect to n . Applying the master theorem, $T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.585\dots})$. Thus, the method is more efficient than the approach related to (6.3), whose computational cost is described by:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 4T(n/2) + en + f & \text{if } n > 1, \end{cases}$$

where $T(n) \in \Theta(n^2)$.

Karatsuba's algorithm computes less multiplications (which correspond to recursive calls) than the school method, but performs more additions and subtractions. In practice, the algorithm will be quicker for large values of n . However, if n is small, the extra operations may make it run slower than the traditional approach. In any case, the method is noteworthy since it is able to reduce the number of multiplication operations, which are considerably more costly than additions or subtractions.

6.5 MATRIX MULTIPLICATION

Two matrices can be multiplied by partitioning them into block matrices as described in [Section 2.4](#). We will now examine a straightforward divide and conquer recursive method that requires n^3 elementary (number) multiplications in order to compute the product of two $n \times n$ matrices (defined through the NumPy package). In addition, we will cover Strassen's algorithm, which is able to obtain the result using approximately $n^{2.8}$ basic scalar multiplications.

6.5.1 Divide and conquer matrix multiplication

Let \mathbf{A} and \mathbf{B} be $p \times q$ and $q \times r$ -dimensional matrices, respectively. The size of the problem depends on the three dimensions p , q , and r . A trivial base case occurs when $p = q = r = 1$, where the result is a simple scalar number. Additionally, some implementations may require considering situations where a dimension is 0. In those cases the output should be an empty matrix, as will be addressed shortly.

One fairly straightforward way to decompose the problem consists of partitioning each matrix into four block matrices (forming a 2×2 array of block matrices). In that case their product can be defined as follows:

$$\begin{aligned} \mathbf{AB} &= \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{bmatrix}. \end{aligned} \tag{6.5}$$

Notice that the formula is analogous to multiplying two 2×2 matrices. For example, the top-left block of the result ($\mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$) can be viewed as the product between the first (block) row of \mathbf{A} and the first (block) column of \mathbf{B} .

The decomposition involves computing eight simpler matrix products. Thus, the method will invoke itself eight times in the recursive case. The results of each product need to be added and stacked appropriately in order to form the output matrix. Listing 6.9 shows a possible implementation. The recursive case first defines each of the smaller block matrices, adds the simpler products, and builds the output matrix through the methods `vstack` and `hstack`. One of the base cases computes a simple product when $p = q = r = 1$. In addition, the code also considers the possibility of receiving empty input matrices, since they appear when partitioning the matrices in the recursive case if one of the dimensions is equal to one (obviously, a vector cannot be partitioned into four vectors as described in (6.5)). Thus, if any of the dimensions is 0 a special base case returns an empty matrix of dimensions $p \times r$, which can be handled appropriately in Python.

The previous method creates 1×1 matrices (in a base case) and progressively stacks them together to form the final $p \times r$ matrix. In addition, note that the dimensions of the input matrices to the methods are not fixed.

Another more efficient alternative consists of passing the entire matrices \mathbf{A} and \mathbf{B} in each call, and specifying the blocks that need to be

Listing 6.9 Divide and conquer matrix multiplication.

```

1 import numpy as np
2
3
4 def matrix_mult(A, B):
5     p = A.shape[0]
6     q = A.shape[1]
7     r = B.shape[1]
8
9     if p == 0 or q == 0 or r == 0:
10        return np.zeros((p, r))
11    elif p == 1 and q == 1 and r == 1:
12        return np.matrix([[A[0, 0] * B[0, 0]])]
13    else:
14        A11 = A[0:p // 2, 0:q // 2]
15        A21 = A[p // 2:p, 0:q // 2]
16        A12 = A[0:p // 2, q // 2:q]
17        A22 = A[p // 2:p, q // 2:q]
18
19        B11 = B[0:q // 2, 0:r // 2]
20        B21 = B[q // 2:q, 0:r // 2]
21        B12 = B[0:q // 2, r // 2:r]
22        B22 = B[q // 2:q, r // 2:r]
23
24        C11 = matrix_mult(A11, B11) + matrix_mult(A12, B21)
25        C12 = matrix_mult(A11, B12) + matrix_mult(A12, B22)
26        C21 = matrix_mult(A21, B11) + matrix_mult(A22, B21)
27        C22 = matrix_mult(A21, B12) + matrix_mult(A22, B22)
28
29        return np.vstack([np.hstack([C11, C12]),
30                           np.hstack([C21, C22])])
31
32
33 A = np.matrix([[2, 3, 1, -3], [4, -2, 1, 2]])
34 B = np.matrix([[2, 3, 1], [4, -1, -5], [0, -6, 3], [1, -1, 1]])
35 print(matrix_mult(A, B))

```

multiplied through appropriate limits passed as parameters, similarly to [Listing 1.6](#). In addition, the result can be stored in a $p \times r$ matrix parameter **C** (passed by reference). [Listing 6.10](#) shows a possible implementation of this alternative solution. The method `matrix_mult_limits` always passes the entire matrices **A** and **B** in each call, storing the result in a $p \times r$ matrix (its third parameter). Additionally, it specifies the submatrices that it will actually multiply through the rest of the param-

Listing 6.10 Alternative divide and conquer matrix multiplication.

```

1 import numpy as np
2
3
4 def add_matrices_limits(A, B, C, lp, up, lr, ur):
5     for i in range(lp, up + 1):
6         for k in range(lr, ur + 1):
7             C[i, k] = A[i, k] + B[i, k]
8
9
10 def matrix_mult_limits(A, B, C, lp, up, lq, uq, lr, ur):
11     mp = (lp + up) // 2
12     mq = (lq + uq) // 2
13     mr = (lr + ur) // 2
14
15     if lp == up and lq == uq and lr == ur:
16         C[mp, mr] = A[mp, mq] * B[mq, mr]
17     elif lp <= up and lq <= uq and lr <= ur:
18
19         M1 = np.zeros((A.shape[0], B.shape[1]))
20         M2 = np.zeros((A.shape[0], B.shape[1]))
21
22         matrix_mult_limits(A, B, M1, lp, mp, lq, mq, lr, mr)
23         matrix_mult_limits(A, B, M2, lp, mp, mq + 1, uq, lr, mr)
24         add_matrices_limits(M1, M2, C, lp, mp, lr, mr)
25
26         matrix_mult_limits(A, B, M1, lp, mp, lq, mq, mr + 1, ur)
27         matrix_mult_limits(
28             A, B, M2, lp, mp, mq + 1, uq, mr + 1, ur)
29         add_matrices_limits(M1, M2, C, lp, mp, mr + 1, ur)
30
31         matrix_mult_limits(A, B, M1, mp + 1, up, lq, mq, lr, mr)
32         matrix_mult_limits(
33             A, B, M2, mp + 1, up, mq + 1, uq, lr, mr)
34         add_matrices_limits(M1, M2, C, mp + 1, up, lr, mr)
35
36         matrix_mult_limits(
37             A, B, M1, mp + 1, up, lq, mq, mr + 1, ur)
38         matrix_mult_limits(
39             A, B, M2, mp + 1, up, mq + 1, uq, mr + 1, ur)
40         add_matrices_limits(M1, M2, C, mp + 1, up, mr + 1, ur)
41
42
43 def matrix_mult_limits_wrapper(A, B):
44     C = np.zeros((A.shape[0], B.shape[1]))
45     matrix_mult_limits(A, B, C, 0, A.shape[0] - 1,
46                       0, A.shape[1] - 1, 0, B.shape[1] - 1)
47     return C

```

eters, which indicate lower and upper limits related to the dimensions p , q , and r . The base case of `matrix_mult_limits` occurs when both of the submatrices correspond to scalar numbers, say $a_{i,j}$ and $b_{j,k}$. In that case the method simply stores their product in row i and column k of \mathbf{C} . Lastly, the method is not a function, since it does not *return* a matrix. Instead, it is a procedure that modifies the parameter \mathbf{C} , where it stores the result. Finally, the iterative method `add_matrices_limits` adds the elements of submatrices passed as the first two matrix input parameters, and stores the result in its third parameter (the submatrices are specified through parameter limits).

6.5.2 Strassen's matrix multiplication algorithm

The most expensive arithmetic operation carried out by the previous algorithms is the scalar multiplication in the base cases. In particular, both require pqr of these multiplications, similarly to the straightforward iterative version that uses three loops. It is nevertheless interesting to analyze the time complexity assuming that the input matrices are $n \times n$. In that case the runtime can be specified through the following function:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 8T(n/2) + 4\Theta(n^2) & \text{if } n > 1, \end{cases} \quad (6.6)$$

since the methods invoke themselves eight times, and need to perform four matrix additions whose cost is quadratic with respect to n . Therefore, according to the master theorem (see (3.28)), $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$. We will now describe Strassen's algorithm, which is a well-known method that can reduce the time complexity to $\Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$.

The method also decomposes each of the input matrices into four block matrices as in the standard algorithm. Thus, $\mathbf{AB} = \mathbf{C}$ can be expressed as:

$$\begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

The key to the method is the definition of the following new matrices that involve one matrix multiplication operation:

$$\begin{aligned}
 \mathbf{M}_1 &= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}), \\
 \mathbf{M}_2 &= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}, \\
 \mathbf{M}_3 &= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}), \\
 \mathbf{M}_4 &= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}), \\
 \mathbf{M}_5 &= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}, \\
 \mathbf{M}_6 &= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}), \\
 \mathbf{M}_7 &= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).
 \end{aligned} \tag{6.7}$$

Finally, these matrices can be combined as follows to form the output's block matrices:

$$\begin{aligned}
 \mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7, \\
 \mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5, \\
 \mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4, \\
 \mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6.
 \end{aligned} \tag{6.8}$$

The algorithm therefore computes seven products and 18 additions (or subtractions) in every recursive call. Thus, its runtime cost is described through:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ 7T(n/2) + 18\Theta(n^2) & \text{if } n > 1, \end{cases} \tag{6.9}$$

which implies that $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$. The algorithm can be faster than the standard method that runs in $\Theta(n^3)$ for large values of n . Nevertheless, for small or medium-sized matrices it may be slower due to the larger multiplicative constants that play a role in practice.

Finally, from a theoretical point of view, the inputs for this algorithm need to be square $n \times n$ matrices, where n is a power of two. Nevertheless, in practice efficient implementations split the matrices into numerous square submatrices, and apply the algorithm repeatedly. A simpler, but slower, alternative consists of padding (i.e., extending) the input matrices with zeros, in order for them to be $2^k \times 2^k$ matrices (see [Exercise 6.6](#)).

6.6 THE TROMINO TILING PROBLEM

A tromino is a polygon formed by connecting three equal-sized squares by their edges. Without considering rotations and reflections, there are



Figure 6.3 Types of trominoes ignoring rotations and reflections.

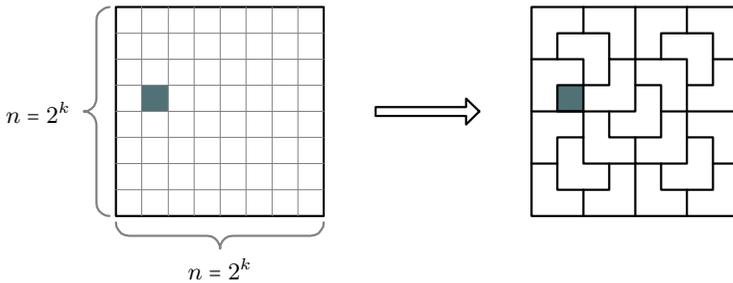


Figure 6.4 Tromino tiling problem.

only two types of trominoes: the “I” and the “L” trominoes, as illustrated in Figure 6.3. The following problem consists of covering a square $n \times n$ board, where $n \geq 2$ is a power of two, which contains a “hole” that cannot be covered, with L trominoes. Figure 6.4 explains the problem graphically with an example.

The size of the problem is clearly n . The smallest instances of the problem correspond to 2×2 boards, whose solutions are trivial. Figure 6.5 illustrates the divide and conquer decomposition used in the recursive case. The initial board in (a) is divided into four smaller square boards of size $n/2$, as shown in (b). However, only one of these smaller boards will contain the initial hole. Therefore, the other three boards do not

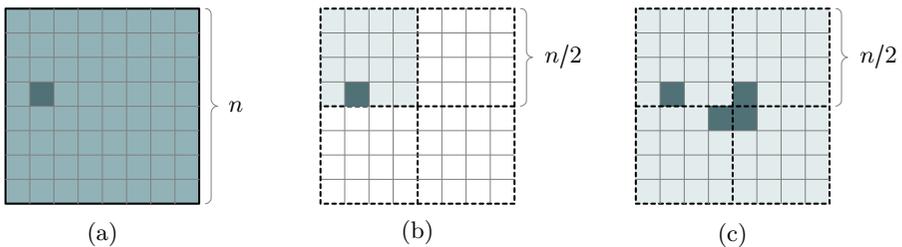


Figure 6.5 Decomposition of the tromino tiling problem.

Listing 6.11 Auxiliary functions for drawing trominoes.

```

1 def draw_L1(x, y):
2     plt.plot([x, x + 2], [y + 2, y + 2], 'k-')
3     plt.plot([x, x + 1], [y + 1, y + 1], 'k-')
4     plt.plot([x + 1, x + 2], [y, y], 'k-')
5     plt.plot([x, x], [y + 1, y + 2], 'k-')
6     plt.plot([x + 1, x + 1], [y, y + 1], 'k-')
7     plt.plot([x + 2, x + 2], [y, y + 2], 'k-')
8
9
10 def draw_L2(x, y):
11     plt.plot([x, x + 2], [y + 2, y + 2], 'k-')
12     plt.plot([x, x + 1], [y, y], 'k-')
13     plt.plot([x + 1, x + 2], [y + 1, y + 1], 'k-')
14     plt.plot([x, x], [y, y + 2], 'k-')
15     plt.plot([x + 1, x + 1], [y, y + 1], 'k-')
16     plt.plot([x + 2, x + 2], [y + 1, y + 2], 'k-')
17
18
19 def draw_L3(x, y):
20     plt.plot([x, x + 2], [y, y], 'k-')
21     plt.plot([x, x + 1], [y + 1, y + 1], 'k-')
22     plt.plot([x + 1, x + 2], [y + 2, y + 2], 'k-')
23     plt.plot([x, x], [y, y + 1], 'k-')
24     plt.plot([x + 1, x + 1], [y + 1, y + 2], 'k-')
25     plt.plot([x + 2, x + 2], [y, y + 2], 'k-')
26
27
28 def draw_L4(x, y):
29     plt.plot([x, x + 2], [y, y], 'k-')
30     plt.plot([x, x + 1], [y + 2, y + 2], 'k-')
31     plt.plot([x + 1, x + 2], [y + 1, y + 1], 'k-')
32     plt.plot([x, x], [y, y + 2], 'k-')
33     plt.plot([x + 1, x + 1], [y + 1, y + 2], 'k-')
34     plt.plot([x + 2, x + 2], [y, y + 1], 'k-')

```

constitute self-similar subproblems. This can be solved by placing a tromino in the center of the board, where its three squares will constitute holes in the smaller boards, creating valid subproblems, as illustrated in (c).

We will use the package Matplotlib in order to generate images showing the solutions to the problem. In particular, the trominoes can be drawn by plotting six line segments. Since there are four possible L trominoes considering rotations (see [Figure 6.6](#)), we can use the four auxil-

Listing 6.12 Recursive method for drawing trominoes.

```

1 def trominoes(x, y, n, p, q):
2     if n == 2:
3         if y == q: # hole in bottom tiles
4             if x == p: # hole in bottom-left tile
5                 draw_L1(x, y)
6             else: # hole in bottom-right tile
7                 draw_L2(x, y)
8         else: # hole in top tiles
9             if x == p: # hole in top-left tile
10                draw_L3(x, y)
11            else: # hole in top-right tile
12                draw_L4(x, y)
13
14     else:
15         mid_x = x + n // 2
16         mid_y = y + n // 2
17
18         if q < mid_y: # hole in bottom squares
19
20             if p < mid_x: # hole in bottom-left square
21                 draw_L1(mid_x - 1, mid_y - 1)
22                 trominoes(x, y, n // 2, p, q)
23                 trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
24                 trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
25                 trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)
26             else: # hole in bottom-right square
27                 draw_L2(mid_x - 1, mid_y - 1)
28                 trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
29                 trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
30                 trominoes(mid_x, y, n // 2, p, q)
31                 trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)
32
33         else: # hole in top squares
34
35             if p < mid_x: # hole in top-left square
36                 draw_L3(mid_x - 1, mid_y - 1)
37                 trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
38                 trominoes(x, mid_y, n // 2, p, q)
39                 trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
40                 trominoes(mid_x, mid_y, n // 2, mid_x, mid_y)
41             else: # hole top-right square
42                 draw_L4(mid_x - 1, mid_y - 1)
43                 trominoes(x, y, n // 2, mid_x - 1, mid_y - 1)
44                 trominoes(x, mid_y, n // 2, mid_x - 1, mid_y)
45                 trominoes(mid_x, y, n // 2, mid_x, mid_y - 1)
46                 trominoes(mid_x, mid_y, n // 2, p, q)

```

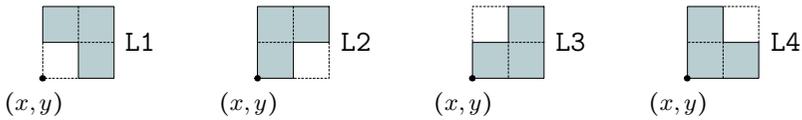


Figure 6.6 L trominoes considering rotations.

Listing 6.13 Code for calling the trominoes method.

```

1 import random
2 import matplotlib.pyplot as plt
3 from matplotlib.patches import Rectangle
4
5 # Include tromino methods here
6
7 fig = plt.figure()
8 fig.patch.set_facecolor('white')
9 ax = plt.gca()
10 n = 16 # power of 2
11 p = random.choice([i for i in range(n)])
12 q = random.choice([i for i in range(n)])
13 ax.add_patch(Rectangle((p, q), 1, 1, facecolor=(0.5, 0.5, 0.5)))
14 trominoes(0, 0, n, p, q)
15 plt.axis('equal')
16 plt.axis('off')
17 plt.show()

```

ary functions in [Listing 6.11](#) to draw each one. The functions receive the coordinates (x, y) of the bottom-left corner corresponding to the square surrounding the tromino. The command `plot([x_1, x_2], [y_1, y_2], 'k-')` draws a black line segment with endpoints (x_1, y_1) and (x_2, y_2) .

[Listing 6.12](#) shows a possible implementation of the recursive method. The procedure needs to know which problem/subproblem it should solve. This information is provided by the first three parameters. The first two indicate the bottom-left coordinates (x, y) of the board, while the third is the size of the board (n) . The last two indicate the location of the hole (in particular, (p, q) specifies the bottom-left corner of the 1×1 square). In both base and recursive cases the method uses conditions in order to determine the relative position of the hole, and draws the appropriate tromino. Finally, the method invokes itself four times in the recursive case, with different parameters indicating the new subproblems, together with the new holes on three of them.

Finally, [Listing 6.13](#) shows a fragment of code that can be used to call the `trominoes` method. Line 7 creates a figure, line 8 sets its background color to white, and `ax` captures the axes of the figure in line 9. After defining the size of the initial board (line 10), the hole is chosen within it at random, and then drawn in line 13. When using the Matplotlib package a rectangle can be formed by calling the method `Rectangle`. It receives the coordinates of the bottom-left vertex, together with the width and height, and other possible arguments. Line 14 calls the main method, and the last lines are included to avoid scaling factors, to eliminate the axes in the final image, and to draw it.

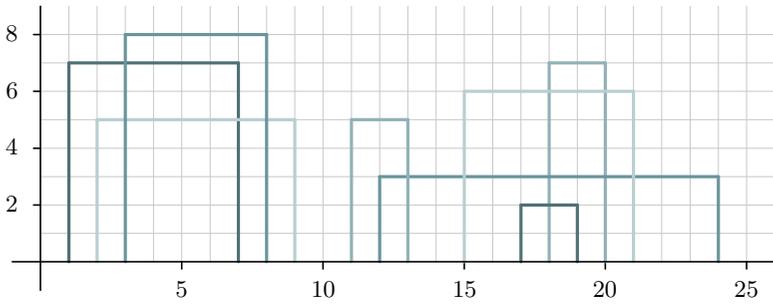
6.7 THE SKYLINE PROBLEM

The skyline problem consists of finding the outline of a set of rectangular buildings against the sky. [Figure 6.7](#) explains the idea. The input to the problem is a list of $n \geq 1$ rectangles that represent buildings, as shown in (a). The bottom side of the rectangles is always located at level 0. Thus, each building can be specified by using only three parameters. In particular, we will use tuples of the form (x_1, x_2, h) , where x_1 marks the location of left side of the building, x_2 indicates the right side, and h contains its height. Thus, $x_1 < x_2$, and $h > 0$. The buildings in the example are: $[(1,7,7), (18,20,7), (2,9,5), (17,19,2), (12,24,3), (3,8,8), (11,13,5), (15,21,6)]$. Note that it is not necessary to sort them in any way (e.g., according to the value of x_1).

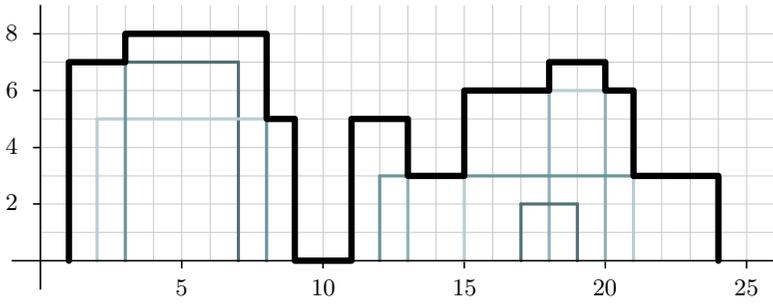
The skyline consists of a curve whose height at any point on the X axis is the maximum height of the buildings at such point, as illustrated through the thick dark segments in (b). Since the buildings are rectangular the skyline can be specified through a set of coordinates on the plane (x, h) that mark the location x when its height changes to h , as illustrated in (c). The output to the problem will therefore be a list of these coordinates, which can be coded as tuples as well, and will appear sorted in ascending order. The output for the example is: $[(1,7), (3,8), (8,5), (9,0), (11,5), (13,3), (15,6), (18,7), (20,6), (21,3), (24,0)]$.

The size of the problem is the number of buildings n . For the base case, the smallest instance occurs when there is only one building. If it is specified through the tuple (x_1, x_2, h) then the output is the list $[(x_1, h), (x_2, 0)]$, as shown in [Figure 6.8](#).

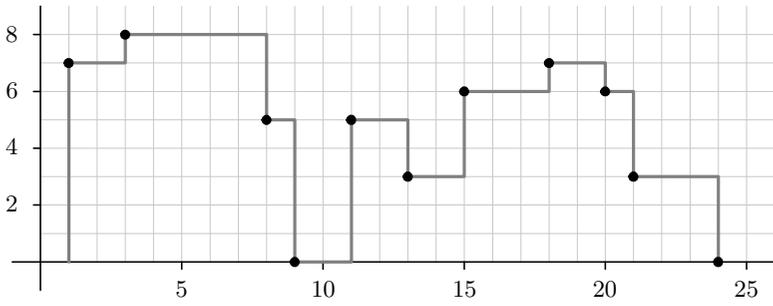
This problem can be decomposed by reducing the size of the problem by a unit. However, this leads to an algorithm whose runtime is $\mathcal{O}(n^2)$ in the worst case. Instead, we will examine a divide and conquer approach



(a)



(b)



(c)

Figure 6.7 The skyline problem.

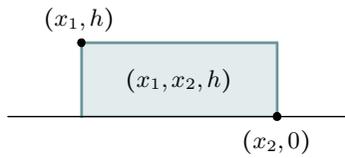


Figure 6.8 Base case for the skyline problem with one building.

Listing 6.14 Main recursive method for computing skylines.

```

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         skyline1 = compute_skyline(buildings[0:n // 2])
8         skyline2 = compute_skyline(buildings[n // 2:n])
9         return merge_skylines(skyline1, skyline2, 0, 0)

```

that is able to solve the problem in $\Theta(n \log n)$ time. The idea, illustrated in Figure 6.9, is similar to the approach used in the merge sort algorithm. The decomposition step consists of dividing the input list in two smaller lists of approximately $n/2$ buildings. The method then carries out two recursive calls on those sublists that return two independent skylines. Assuming that the skylines have been constructed correctly (by applying induction), the final, and challenging, step consists of merging the skylines in order to produce a final one. Listing 6.14 shows the associated divide and conquer method, whose structure is essentially identical to that of Listing 6.2.

The skyline merging problem is a new computational problem in its own right. While the majority of solutions in texts are iterative, we will now examine a linear-recursive method. The inputs are the two input lists of sorted tuples representing skylines. In addition, since a tuple indicates a change in the height of a skyline, the method needs to access the previous height before such change. Moreover, since the proposed algorithm will process the first tuples from the lists (until one list is empty), but progressively discard them as they are analyzed, these previous heights will not be contained in the input lists. Thus, the method will need two additional parameters, say p_1 and p_2 , in order to store the previous heights of the skylines. Naturally, both of these parameters will

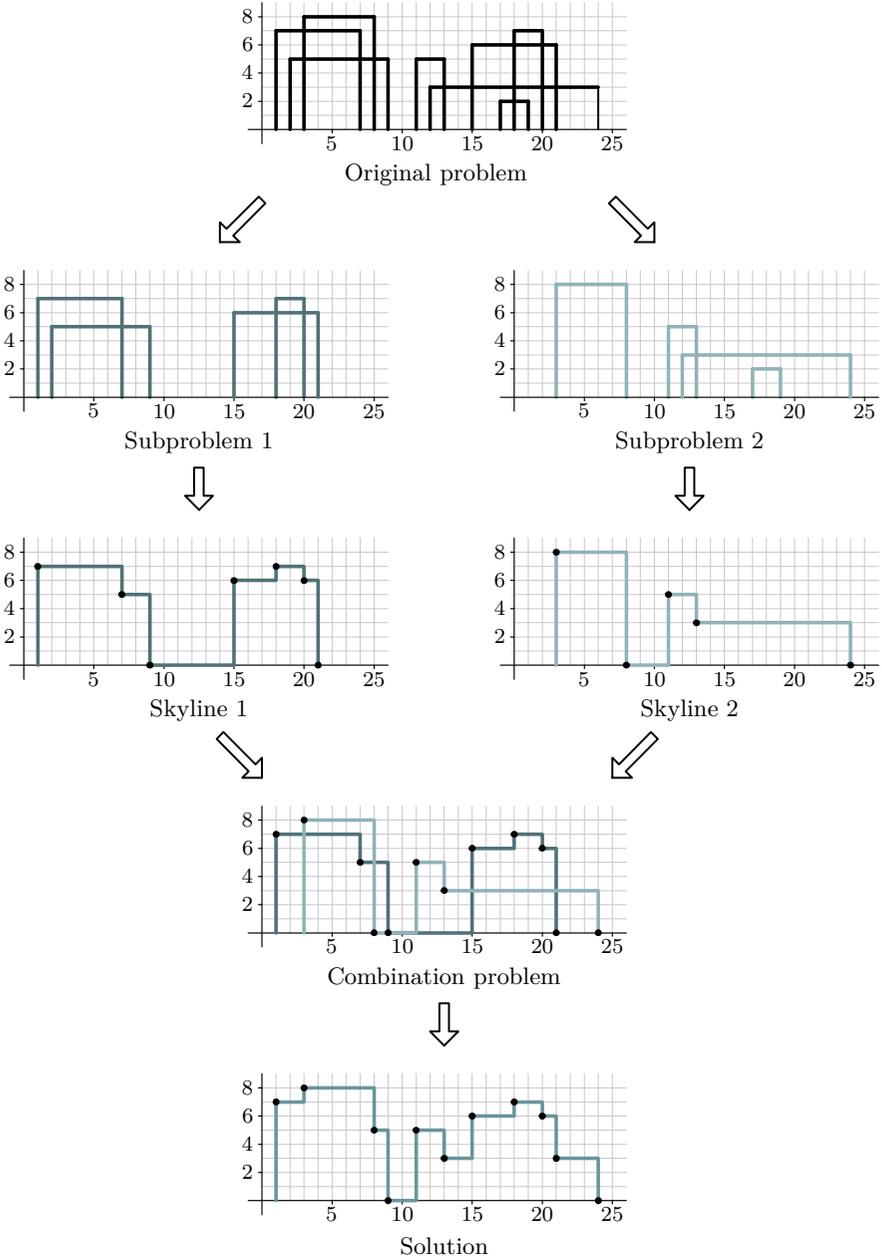


Figure 6.9 Recursive case for the skyline problem.

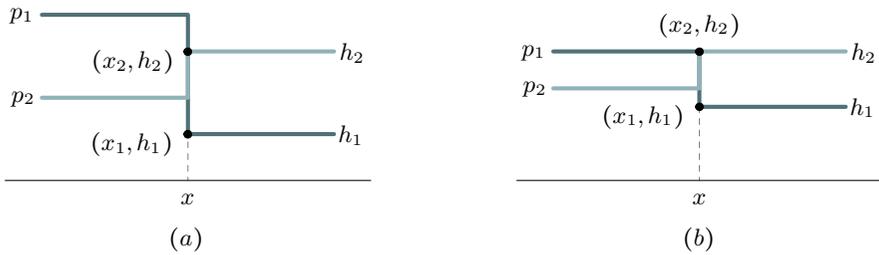


Figure 6.10 Possible situations when merging skylines that change at the same location x .

be initialized to zero when calling the method within the main skyline function (see line 9 of [Listing 6.14](#)).

The size of the problem depends on the lengths of the input skyline lists, say n_1 and n_2 . We can consider that the base case occurs when one of the lists is empty, where the method must trivially return the other list. [Listing 6.15](#) codes the method, where the base cases are described in lines 2–5.

A key observation for determining an appropriate decomposition is that the output of the merging function produces a new skyline whose tuples are sorted in ascending order according to their x value. Thus, the algorithm will analyze the first tuples of each list, and process the one with a smaller x value (or both if their x values are the same). Thus, in the recursive case we need the first tuples of the skylines (x_1, h_1) and (x_2, h_2) , together with their previous heights p_1 and p_2 .

Firstly, let us consider the situation when $x_1 = x_2 = x$. Since the tuples mark changes in the skyline, we may need to include in the solution the one with larger height. For example, in [Figure 6.10\(a\)](#) the point (x, h_2) would be included in the final skyline. Furthermore, the recursive call will use the tails of both input lists, discarding (x, h_1) and (x, h_2) , since the possible changes at x will have been processed correctly. This is accomplished in lines 19–21. Lastly, there is a situation where a new tuple is not included in the solution. This occurs when the largest new height is equal to the largest previous height of the skyline (i.e., when $\max(h_1, h_2) = \max(p_1, p_2)$), since there would be no change of heights at x . [Figure 6.10\(b\)](#) illustrates this case, where the point (x, h_2) would not be included in the final skyline (see lines 16 and 17).

We will now analyze possible scenarios when the x values of the first tuples of the skylines are not equal. Without loss of generality, assume

Listing 6.15 Recursive method for merging skylines.

```

1 def merge_skylines(sky1, sky2, p1, p2):
2     if sky1 == []:
3         return sky2
4     elif sky2 == []:
5         return sky1
6     else:
7         x1 = sky1[0][0]
8         x2 = sky2[0][0]
9         h1 = sky1[0][1]
10        h2 = sky2[0][1]
11
12        if x1 == x2:
13            h = max(p1, p2)
14            new_h = max(h1, h2)
15            if h == new_h:
16                return merge_skylines(sky1[1:], sky2[1:],
17                                       h1, h2)
18            else:
19                return [(x1, new_h)]
20                        + merge_skylines(sky1[1:], sky2[1:],
21                                       h1, h2))
22
23        elif x1 < x2:
24            if h1 > p2:
25                return [(x1, h1)]
26                        + merge_skylines(sky1[1:], sky2,
27                                       h1, p2))
28            elif p1 > p2:
29                return [(x1, p2)]
30                        + merge_skylines(sky1[1:], sky2,
31                                       h1, p2))
32            else:
33                return merge_skylines(sky1[1:], sky2,
34                                       h1, p2)
35
36        else:
37            if h2 > p1:
38                return [(x2, h2)]
39                        + merge_skylines(sky1, sky2[1:],
40                                       p1, h2))
41            elif p2 > p1:
42                return [(x2, p1)]
43                        + merge_skylines(sky1, sky2[1:],
44                                       p1, h2))
45            else:
46                return merge_skylines(sky1, sky2[1:], p1, h2)

```

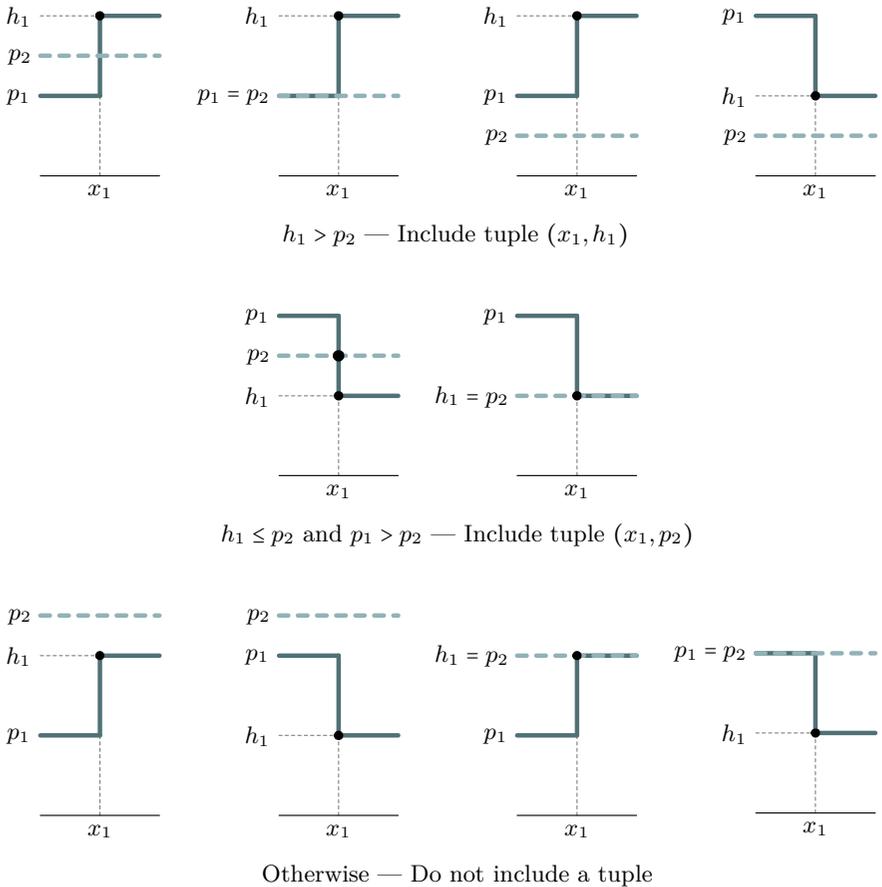


Figure 6.11 Possible situations when merging skylines and $x_1 < x_2$.

$x_1 < x_2$. In that case, the algorithm must decide whether to include the tuple (x_1, h_1) , or (x_1, p_2) , or none at all, as illustrated in Figure 6.11. If $h_1 > p_2$ then the first skyline is above the second one at location x_1 , and therefore must include the tuple (x_1, h_1) as part of the merged skyline (see lines 25–27). If $h_1 \leq p_2$ then the algorithm must check if $p_1 > p_2$. If the result is `True` then the method includes the tuple (x_1, p_2) (see lines 29–31). Notice that when $h_1 < p_2$ this produces a new tuple that does not appear in the input skyline lists. Lastly, in other situations, $p_2 \geq h_1$ and $p_2 \geq p_1$, which implies that the merged skyline will not change at x_1 . Finally, having processed the first tuple from the first skyline (x_1, h_1) , the method discards it when invoking itself in the corresponding recursive

cases. In addition, the arguments that specify the previous heights of the skylines will be h_1 and p_2 (see lines 27, 31, and 34). The rest of the code at lines 36–46 is analogous to the code at lines 23–34, and handles the case when $x_2 < x_1$.

6.8 EXERCISES

Exercise 6.1 — Implement a divide and conquer algorithm that determines whether a list \mathbf{a} contains an element x .

Exercise 6.2 — Let \mathbf{a} be a list of n nonnegative integers. Write a function based on the divide and conquer technique that returns the set of digits shared amongst all of the elements in \mathbf{a} , and specify its asymptotic computational cost. For example, for $\mathbf{a} = [2348, 1349, 7523, 3215]$, the solution is $\{3\}$. The function should call another one that provides the set of digits in a nonnegative integer. Code this function as well.

Exercise 6.3 — The maximum sublist problem consists of finding the sublist of contiguous elements within a list of numbers that has the largest sum of its components. For example, given the list $[-1, -4, 5, 2, -3, 4, 2, -5]$, the optimal sublist is $[5, 2, -3, 4, 2]$, whose elements sum up to 10. Given a nonempty input list of numbers \mathbf{a} , implement a divide and conquer function that returns the sum of the elements of its maximum sublist.

Exercise 6.4 — Design a fast recursive polynomial multiplication algorithm based on a divide and conquer decomposition analogous to the one used in Karatsuba’s algorithm (see [Section 6.4](#)). Code polynomials through lists, as described in [Exercise 5.2](#). The method will need to call functions that add and subtract polynomials. Code these functions as well.

Exercise 6.5 — Implement a recursive function that receives an $n \times m$ matrix (\mathbf{A}) and returns its $m \times n$ transpose (\mathbf{A}^T). Use the NumPy package, and the following divide and conquer decomposition that breaks up the input matrix \mathbf{A} into four block matrices by dividing each dimension by two:

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \hline \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{array} \right].$$

In that case, the transpose of \mathbf{A} can be defined as:

$$\mathbf{A}^T = \left[\begin{array}{c|c} \mathbf{A}_{1,1}^T & \mathbf{A}_{2,1}^T \\ \hline \mathbf{A}_{1,2}^T & \mathbf{A}_{2,2}^T \end{array} \right].$$

Exercise 6.6 — Implement Strassen’s matrix multiplication algorithm. The code should include a wrapper function that will allow multiplying a $p \times q$ matrix times a $q \times r$ matrix, by padding the input matrices with zeros.

Exercise 6.7 — Implement a matrix multiplication algorithm by decomposing the input matrices as follows:

$$\mathbf{A} \cdot \mathbf{B} = \left[\begin{array}{c|c} \mathbf{A}_1 & \mathbf{A}_2 \end{array} \right] \cdot \left[\begin{array}{c} \mathbf{B}_1 \\ \mathbf{B}_2 \end{array} \right] = \left[\begin{array}{c} \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2 \end{array} \right].$$

Exercise 6.8 — Implement a matrix multiplication algorithm by decomposing the input matrices as follows:

$$\mathbf{A} \cdot \mathbf{B} = \left[\begin{array}{c} \mathbf{A}_1 \\ \mathbf{A}_2 \end{array} \right] \cdot \left[\begin{array}{c|c} \mathbf{B}_1 & \mathbf{B}_2 \end{array} \right] = \left[\begin{array}{cc} \mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 \\ \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2 \end{array} \right].$$

Multiple Recursion III: Backtracking

He who would search for pearls must dive below.

— John Dryden

BACKTRACKING is one of the most important algorithm design paradigms. It can be regarded as an “intelligent” brute-force strategy that performs exhaustive searches for solutions to constraint satisfaction and discrete optimization problems. The approach can be used to solve numerous puzzles and problems, including the eight-queens problem, finding paths through mazes, the sudoku puzzle, the 0-1 knapsack optimization problem, and many more.

Backtracking methods generally combine recursion and iteration, contain several parameters, and are not usually designed by strictly thinking about problem decomposition and induction. Thus, they appear to be complex to students who study the material for the first time. Fortunately, backtracking algorithms often share a similar structure, which can be exploited in order to ease their design. This structure is captured in what are known as “backtracking templates,” which vary depending on coding styles and programming languages. In this book the methods will share a particular structure where they receive a relatively large number of parameters, and barely rely on additional methods. However, the reader should be aware that there are other possibilities. In any case, students can master backtracking with relative ease by studying examples, and by applying very similar algorithms to different problems.

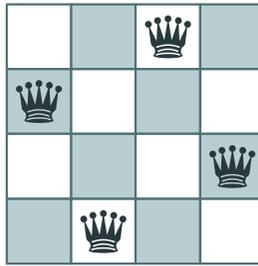


Figure 12.1 One solution to the four-queens puzzle.

12.1 INTRODUCTION

This section introduces fundamental concepts related to backtracking, and provides an overview of how it works by examining the simple four-queens puzzle. Its goal consists of placing four chess queens on a 4×4 chessboard so that they do not threaten each other. Since queens can move horizontally, vertically, and diagonally, two (or more) queens cannot appear in the same row, column, or diagonal on the board. [Figure 12.1](#) illustrates one of the two possible solutions to the puzzle. Naturally, the problem can be generalized to placing n queens on an $n \times n$ chessboard (see [Section 12.3](#)).

12.1.1 Partial and complete solutions

Backtracking is a general strategy for finding solutions to computational problems among a finite set of possibilities. These solutions are collections of discrete items. Thus, technically we say that backtracking is a strategy that searches for solutions in a “discrete state space.” It is also a brute-force method in the sense that the search is exhaustive. In other words, if a solution exists, a backtracking algorithm is guaranteed to find it.

The methods search for solutions by constructing and updating a **partial solution**, which could eventually become a valid **complete solution** to the problem. Partial solutions are built by incorporating **candidate** elements incrementally, step by step, in successive recursive calls. Thus, there is an implicit order in which the candidates appear in the solutions. In this regard, partial solutions can be understood as “prefixes” of complete solutions. When a solution is a list, a partial solution is simply a sublist that contains the first elements, as illustrated in [Figure 12.2\(a\)](#) through the lighter-shaded region. Instead, if it is a two-

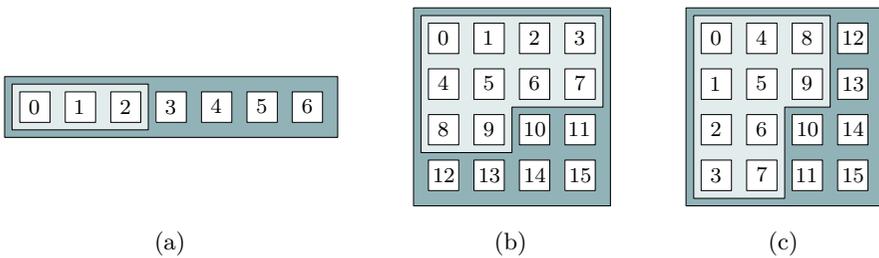


Figure 12.2 Partial solutions within complete solutions that are coded as lists or matrices.

dimensional matrix, we need to consider some linear arrangement of its elements. The two most natural choices consist of ordering the entries of a matrix in row or column-major order, as shown in (b) and (c), respectively. In those cases a partial solution is the collection of the first elements of the matrix according to one of these linear arrangements.

The key to using backtracking effectively is the possibility of efficiently analyzing whether a partial solution satisfies the constraints of the problem. If it does, then it could potentially be expanded into a complete valid solution. However, if it is not possible to create a valid partial solution by including a new item, the algorithm “backtracks” to another previous valid partial solution, and continues exploring other possibilities not contemplated yet. Backtracking algorithms therefore perform exhaustive searches for solutions, but can solve some problems efficiently by avoiding incrementing partial solutions that do not lead to valid complete solutions.

The first task when tackling a problem through backtracking consists of selecting a particular form or data structure for the solutions. Typically, they are implemented as lists or matrices. For the n -queens puzzle a first idea that could come to mind is to use a Boolean $n \times n$ matrix to encode the solutions, where (n) **True** values mark the squares where the queens lie on the board. A partial solution would have the same structure, but it would contain less than n queens (as soon as you place the n -th queen the solution is either not valid, or it is a complete solution to the problem). However, although it is possible to use a matrix, this choice of data structure does not lead to an efficient algorithm. Since backtracking methods perform exhaustive searches, an algorithm would traverse the entire matrix in row or column-major order, deciding whether to place a queen in each of the chessboard’s squares. This would

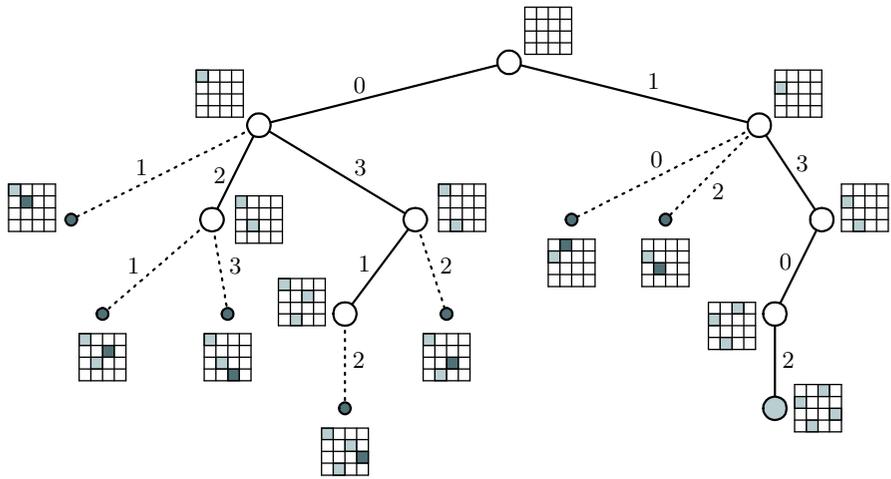


Figure 12.3 Recursion tree of a backtracking algorithm that finds one solution to the four-queens puzzle.

be time-consuming, since, for example, if a queen is placed in a particular column, then the algorithm does not need to try to place other queens on that same column. Instead, the method could simply move on to the next column.

A more appropriate choice consists of using a simple list of length n . In that case, the indices of the list could represent the columns of the chessboard, and the particular value could denote the row where a queen is placed. In other words, if \mathbf{x} is a list that represents a solution, and there is a queen in column i and row j , then we have $x_i = j$. For example, if columns are numbered from left to right, and rows from top to bottom, both starting at 0, the solution in Figure 12.1 corresponds to the list $[1, 3, 0, 2]$. By using a list, after placing a (valid) queen in a row for a particular column (i), the algorithm can continue to try to place a queen in the next column ($i + 1$). In addition, it can keep track of the rows where there are queens, and avoid placing further queens in them.

12.1.2 Recursive structure

We can examine how a backtracking algorithm works by analyzing its recursion tree. Figure 12.3 illustrates it for a backtracking method that finds one solution to the four-queens puzzle. The root node represents the first call to the method, where the partial solution, drawn as a 4×4

board (but coded as a list), is empty. Subsequently, the order of the recursive calls is in accordance with a preorder traversal of the tree. The first recursive call, associated with the root's left child, places a queen (depicted as a shaded square) in the first row of the first column. The label "0" on the corresponding edge indicates the row where a queen is placed. Since this does not violate the constraints of the problem it can continue to place queens in the second column. In particular, it starts by avoiding placing a queen in the first row, since there is already a queen on it. Subsequently, the algorithm analyzes whether it can place a queen on the second row. Clearly it cannot, since the two queens would lie on the same diagonal. The associated partial solution would not be valid, and the algorithm does not proceed with expanding it. In other words, it will not carry out any further recursive calls with that (invalid) partial solution. This is illustrated in the diagram through dotted edges.

Subsequently, the method proceeds by placing a queen in the third row, which leads to a valid partial solution. Thus, the algorithm can continue to try to place a queen in the third column. The first free row is the second one, but this leads to an invalid partial solution. Similarly, placing a queen on the fourth row violates the constraints of the problem. Since there are no more options left, the algorithm "backtracks" to the node where there is only one queen on the first column and first row. Afterwards, the method explores all of the possible partial solutions that could be obtained when placing a queen on the fourth row of the second column. Since there are no valid solutions for this option either, the algorithm will end up backtracking to the initial node. Note that at this stage the method will have carried out an exhaustive search for solutions in which a queen appears on the first row of the first column. Subsequently, the algorithm continues exploring possibilities when starting by placing a queen on the second row of the first column. The same procedure is repeated until eventually the method is able to place four queens that do not threaten each other. In that case it can stop searching if the goal consists of finding only one solution, or it can continue to search for all of the solutions.

The recursion tree also illustrates several features of the algorithm. Firstly, the nodes correspond to recursive method calls where the partial solution (which will be an input parameter) is valid. The small dark nodes attached through dotted edges simply show how the tree would have been expanded had its corresponding partial solution been valid. Also, observe that an item is added to the partial solution at each recursive call. In other words, partial solutions that contain k items will

be associated with nodes of depth k . Therefore, complete solutions are found when reaching leaf nodes, of depth n . Furthermore, a path from the root to a node specifies the partial solution, or a complete solution if the node is a leaf. For example, the solution found by the algorithm is the list $[1, 3, 0, 2]$, which is the sequence of labels associated with the edges of the path from the root to the leaf where the solution is found. In addition, since the values in the list represent different rows, the solution to the four-queens puzzle is actually a permutation of the rows, which are the first n nonnegative integers. A tree that explores all possible permutations would have $n!$ leaves, which can be very large, even for small values of n . However, the figure also illustrates that the recursive tree can be pruned considerably by not expanding invalid partial solutions, which is key to obtaining efficient algorithms.

Similar details will appear in other problems and examples in the chapter. In particular, [Section 12.3](#) will cover the n -queens puzzle in depth, showing a specific coded algorithm that solves it. Lastly, the tree shown in [Figure 8.18](#) could correspond to a recursion tree of an algorithm that searches for two-element variations with repetition of four items. The lists next to the nodes would be partial or complete solutions, while the labels next to the edges would indicate the element that is introduced in a partial solution. In practice, backtracking algorithms prune the recursion trees in order to gain efficiency by discarding invalid solutions.

12.2 GENERATING COMBINATORIAL ENTITIES

We have just seen that the solution to the n -queens problem is a permutation (of the first n nonnegative integers). This will also be the case in numerous problems. In addition, the goal in many other problems consists of finding subsets of n elements. Therefore, it is important to understand how to generate all of the permutations, and all of the subsets, of n distinct elements. This section presents specific algorithms in Python for generating these combinatorial entities. They are particularly beneficial since they can be used as starting points to build backtracking algorithms for many problems.

Lastly, these algorithms will not generate lists or other data structures in order to store all of the subsets or permutations. This would not be practical due to the large number of possibilities (there are $n!$ possible permutations, and 2^n different subsets, of n distinct elements). Instead, the methods will use a simple list (the partial solution) that rep-

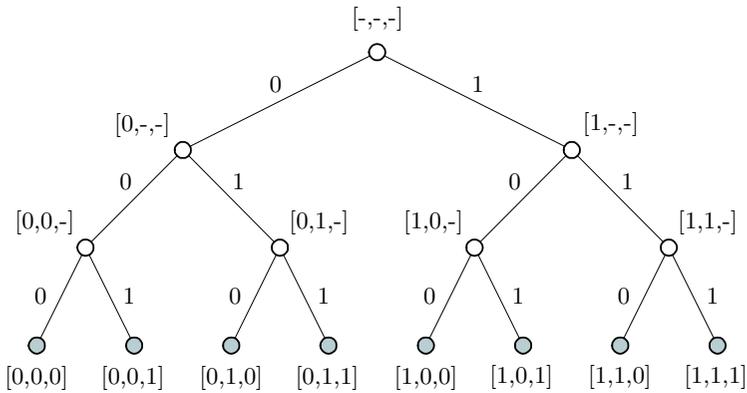


Figure 12.4 Binary recursion tree of an algorithm that generates all of the subsets of three items.

resents only one of the subsets/permutations, but which will be modified in order to contain all of them as the algorithms are executed. Thus, the methods will be able to process every subset/permutation (we will simply print them or count them), but will not return all of them in a single data structure.

12.2.1 Subsets

This section presents two strategies for generating all of the subsets of n (distinct) elements, which will be provided through an input list. In both methods the recursion tree will be binary, and the solutions (subsets) will be represented at its leaves. One method uses partial solutions of (fixed) length n , while in the other their length varies as the procedure carries out recursive calls.

12.2.1.1 Partial solutions of fixed length

Figure 12.4 shows the binary recursion tree of an algorithm that generates the eight subsets that it is possible to create with three distinct elements. Each subset is described through a binary list of n zeros or ones (naturally, Boolean values can be used as well). For example, given the list of elements $[a, b, c]$, the list $[1, 0, 1]$ denotes the subset $\{a, c\}$. This binary list is the partial solution, which is complete when the method reaches a leaf. The binary digits are therefore the candidate elements that form it. As the method carries out recursive calls it simply appends

Listing 12.1 Code for printing all of the subsets of the elements in a list.

```

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('}')

```

a new candidate to the partial solution (the concrete binary value is shown on the edges of the tree), which is passed as a parameter to the method, and shown next to the nodes of the recursion tree. Initially that list has length n , but its entries are meaningless. When the algorithm reaches a base case it will have generated one of the 2^n possible subsets, at one of the leaves of the recursion tree. In those cases the lists contain n meaningful elements and will represent complete solutions.

Listing 12.1 shows an algorithm that complies with the recursion tree in Figure 12.4, and which prints each subset through the method `print_subset_binary`. Besides the initial list `elements` that represents the set of n items that it will group into subsets, the procedure also receives an input list `sol` that represents the partial solutions, and which has n components initialized to `None` in the wrapper method `generate_subsets_wrapper`. In addition, observe that the method includes a new candidate in the partial solution, at every level, as it descends through the recursion tree. Therefore, it uses a third parameter `i` that indicates the index in the partial solution where a new zero or one will appear. Furthermore, `i` also corresponds to the depth of the node associated with the recursive call responsible for introducing the new element in the partial solution. Since the recursive backtracking method `generate_subsets` begins introducing a zero or a one at index 0, `i` is initialized to 0 in `generate_subsets_wrapper`.

The method `generate_subsets` checks in line 3 if the partial solution in `sol` is really a complete solution, which occurs if `i` is equal to n . This would correspond to the base case, where the method simply prints the solution (line 5). In the recursive case the procedure has to perform two recursive calls in order to include more candidates in the partial solution. It accomplishes this through a simple loop where the variable `k` takes either the value 0 or 1 (line 8), and where the i -th component of the partial solution (`sol`) receives the value of `k` (line 11). The backtracking algorithms included in this book will also use a loop in order to consider every possible candidate that could (potentially) be included in the partial solution. The procedure then invokes itself (line 14) with the modified partial solution, and incrementing `i` in order to include more candidates (if the partial solution is not complete).

The code in line 17 is optional, since the method works correctly without it. Nevertheless, the procedure includes it for illustrative purposes, since similar instructions may be necessary in other backtracking algorithms. Observe that when a particular node in the recursion tree finishes executing, the control passes to its parent, where the value of the partial solution at the i -th position is meaningless. Therefore, we can express this explicitly through the code in line 17. However, the code works without it since this particular method only prints a subset when reaching a base case (where partial solutions are complete), and because the assignment in line 11 simply overwrites a zero when `k = 1`. The reader is encouraged to execute the program step by step with a

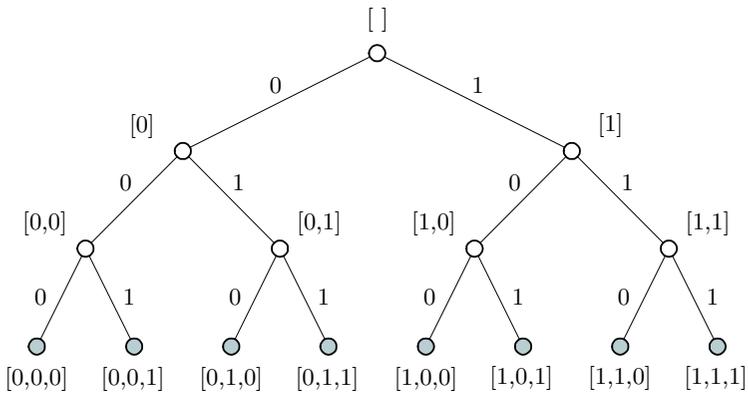


Figure 12.5 Alternative binary recursion tree of an algorithm that generates all of the subsets of three items.

debugger, with or without line 17, in order to examine the evolution of the partial solution.

Finally, a call to `generate_subsets_wrapper(['a', 'b', 'c'])` produces the following output:

```
{
  {}
  {c}
  {b}
  {b, c}
  {a}
  {a, c}
  {a, b}
  {a, b, c}
```

12.2.1.2 Partial solutions of variable length

In the previous algorithm the partial solutions always had the same length (n). Thus, the procedure can be used with lists or arrays that do not vary in size. In that case the parameter `i` was useful in order to specify the indices of the elements to be added to the partial solution. Instead, another possibility is to use partial solutions of variable size, where `i` is no longer needed. Figure 12.5 shows another recursive tree that has the same structure as the one in Figure 12.4, but the labels next to the nodes specify partial solutions at every method call where every element is meaningful.

Listing 12.2 Alternative code for printing all of the subsets of the elements in a list.

```

1 def generate_subsets_alt(sol, a):
2     # Base case
3     if len(sol) == len(a):
4         # Print complete solution
5         print_subset_binary(sol, a)
6     else:
7         # Generate candidate elements
8         for k in range(0, 2):
9
10            # Include candidate in partial solution
11            sol = sol + [k]
12
13            # Expand partial solution at position i+1
14            generate_subsets_alt(sol, a)
15
16            # Remove candidate from partial solution
17            del sol[-1]
18
19
20 def generate_subsets_alt_wrapper(elements):
21     sol = []
22     generate_subsets_alt(sol, elements)

```

Listing 12.2 shows a method that is in accordance with this approach, which is very similar to the code in Listing 12.1. Firstly, it prints the solution if it contains n elements (base case). In the recursive case it uses the same loop, appends a zero or a one at the end of the partial solution, and subsequently invokes itself. In this case, the code in line 17 is necessary, where the method has to “undo” the change to the partial solution prior to invoking the method recursively. While the algorithm in Listing 12.1 overwrites a zero in the partial solution when incorporating a one, in this case the method needs to remove the zero from the list in order to include a one properly. Thus, calling the recursive method implies adding an element to the partial solution (line 14), and the algorithm deletes it (line 17) after returning from the recursive call. The code on line 17 of Listing 12.1 plays a similar role, but is not necessary in that method. Regarding efficiency, appending and removing elements from a data structure dynamically can be time-consuming. Thus, it is often more efficient to allocate a fixed amount of memory for the partial solution, and update it without altering its size. In this regard, the code

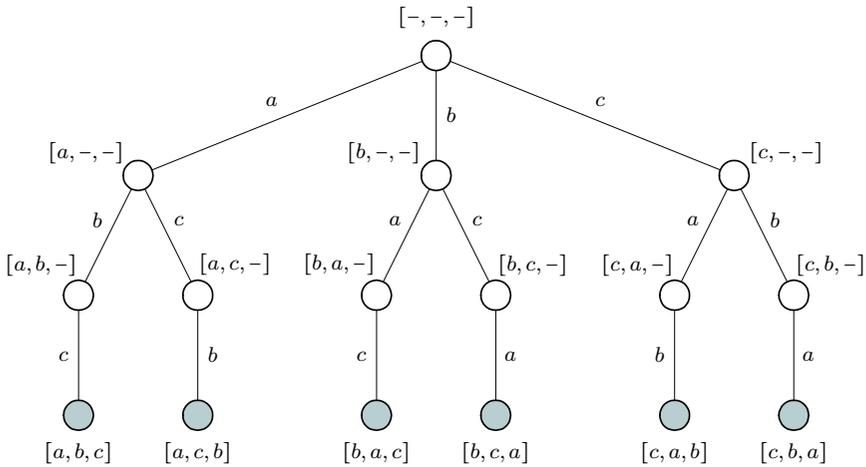


Figure 12.6 Recursion tree of an algorithm that generates all of the permutations of three items.

in [Listing 12.1](#) runs faster than the method `generate_subsets_alt`. Finally, for this alternative algorithm the wrapper method must initialize the partial solution to an empty list.

12.2.2 Permutations

This section examines two similar algorithms that print all of the possible permutations of the n distinct elements in a given list. One way to represent a permutation is through a list of indices from 0 to $n - 1$, which reference the locations of the elements in the input list. For example, given the list $[a, b, c]$, the partial solution $[1, 2, 0]$ would denote the permutation $[b, c, a]$. However, the following algorithms will simply use partial solutions formed by the items of the input list (and `None` values). [Figure 12.6](#) shows the structure of their recursion trees, for the list $[a, b, c]$. Observe that the partial solutions have length n , but only the first items are meaningful. In the first call to the methods the partial solution is “empty,” where all of its elements are set to `None`. The procedures also receive a parameter `i`, initially set to zero, which specifies how many candidate elements have been included in the partial solution. Thus, it indicates the position in the partial solution where the methods introduce a new candidate, and which is also equivalent to the depth of the node related to a method call in the recursion tree.

The methods begin by introducing one of the n candidate elements of the input list at the first position of the partial solution. Since there are n possibilities, the root node has n children. In the next level of the tree the methods invoke themselves $n - 1$ times, since the element that appears in the first position cannot appear again in the permutation. Similarly, in the next level the partial solutions contain two elements and the methods invoke themselves $n - 2$ times. This is repeated in general until they reach a base case, at a leaf of the recursion tree, where the partial solutions are complete permutations.

The next subsections describe both methods, where the main difference between them resides in how they test the validity of a partial solution.

12.2.2.1 *Validity of partial solutions without additional data structures*

Listing 12.3 shows a first method that complies with the recursion tree in Figure 12.6. Since `i` indicates the number of elements in the partial solution `sol`, the first `if` statement checks if the method has reached a base case. If the result is `True` the procedure simply prints the permutation stored in `sol`. If the algorithm has not reached a base case it uses a loop in order to try to include every candidate item of the initial input list `elements` in the partial solution. The loop's counter `k`, which serves as an index of `elements`, therefore receives values from 0 to $n - 1$. Subsequently, the algorithm tests whether the `k`-th candidate has already been included in the partial solution (line 10). It is important to note that although the statement can be written in a single line, it requires examining `i` values in the worst case. If the `k`-th candidate is not in the partial solution the method includes it (line 13), and performs a recursive call incrementing the value of `i` in line 16. Lastly, the method does not require “undoing” the assignment in line 19 after carrying out the recursive call, since it simply overwrites the value `sol[i]`. In other words, it is not necessary to reset `sol[i]` to `None` after the recursive call. Nevertheless, the code includes the instruction so that its recursion tree conforms exactly to the one shown in Figure 12.6.

The method `generate_permutations_wrapper` is a wrapper procedure needed to initialize the partial solution with n `None` values, and call `generate_permutations` with `i` set to zero. Lastly, the code includes a basic procedure that prints a list, separating its elements by space bar characters.

Listing 12.3 Code for printing all of the permutations of the elements in a list.

```

1 def generate_permutations(i, sol, elements):
2     # Base case
3     if i == len(elements):
4         print_permutation(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_permutation(sol):
28     for i in range(0, len(sol)):
29         print(sol[i], ' ', end='')
30     print()

```

Finally, similarly to the rest of the algorithms included in the chapter, the method uses a loop in order to consider including every possible candidate in the partial solution. Without considering constraints in the internal nodes the resulting algorithm would lead to a full n -ary recursion tree (every internal node would have n children), which would contain n^n leaves. Naturally, we could test whether the partial solutions at those n^n leaves are valid. However, it is much more efficient to examine the validity of a candidate in an internal node, and discard unnecessary recursive calls as soon as the algorithm detects that a partial solution will never be valid. Thus, in line 6 the condition verifies that the candidate is

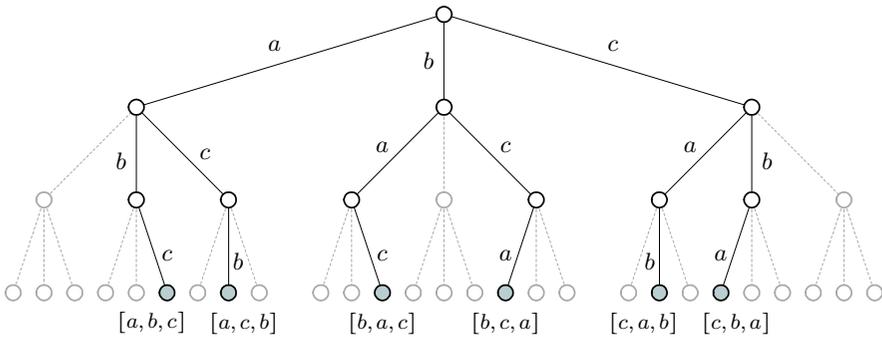


Figure 12.7 Pruning a recursion tree as soon as a partial solution is not valid.

valid, according to the constraints of the problem. This condition speeds up the algorithm dramatically by pruning the recursion tree, as shown in Figure 12.7. The lighter gray nodes and edges indicate method calls of the full ternary tree that are not executed when pruning the tree with the condition in line 6. Finally, the resulting recursion tree (depicted through dark nodes and edges) is identical to the one in Figure 12.6, and contains $n!$ leaves. Although it is a huge quantity even for relatively small values of n , it is much smaller than n^n .

12.2.2.2 Validity of partial solutions with additional data structures

Instead of examining the validity of a partial solution by analyzing its first i values (they have to be different than `elements[k]`), the code in Listing 12.4 uses the additional Boolean parameter `available`, of size n . The idea consists of storing whether a particular item of the input list `elements` is not yet in the partial solution. The list is therefore initialized to n `True` values in a wrapper method. In this case, the condition in line 10 is much simpler, and can be evaluated in $\Theta(1)$ time. Subsequently, if the k -th element is indeed available the procedure first includes it in the partial solution (line 13), marks it as unavailable (line 16), and invokes itself incrementing the parameter i (lines 19 and 20). In addition, in this case it is necessary to undo the change to `available` in line 23. Observe that when the method returns from the recursive call it can carry out more iterations of the loop, inserting other elements at position i in the partial solution, but the list `available` has to be restored to its initial values. In particular, `available[k]` has to be set to

Listing 12.4 Alternative code for printing all of the permutations of the elements in a list.

```

1 def generate_permutations_alt(i, available, sol, elements):
2     # Base case
3     if i == len(elements):
4         print_permutation(sol) # complete solution
5     else:
6         # Generate candidate elements
7         for k in range(0, len(elements)):
8
9             # Check candidate validity
10            if available[k]:
11
12                # Include candidate in partial solution
13                sol[i] = elements[k]
14
15                # k-th candidate no longer available
16                available[k] = False
17
18                # Expand partial solution at position i+1
19                generate_permutations_alt(i + 1, available,
20                                         sol, elements)
21
22                # k-th candidate available again
23                available[k] = True
24
25
26 def generate_permutations_alt_wrapper(elements):
27     available = [True] * (len(elements))
28     sol = [None] * (len(elements))
29     generate_permutations_alt(0, available, sol, elements)

```

True, since the algorithm has to include `elements[k]` at some point in a further position in the partial solution. Finally, although the method requires an extra input parameter, it is more efficient than the algorithm in [Listing 12.3](#).

12.3 THE N-QUEENS PROBLEM

The n -queens puzzle was previously introduced in [Section 12.1](#). Recall that it is a constraint satisfaction problem whose solutions correspond to permutations of the rows where the queens are placed. Therefore, we can use the code that generates permutations in [Listing 12.4](#) as a start-

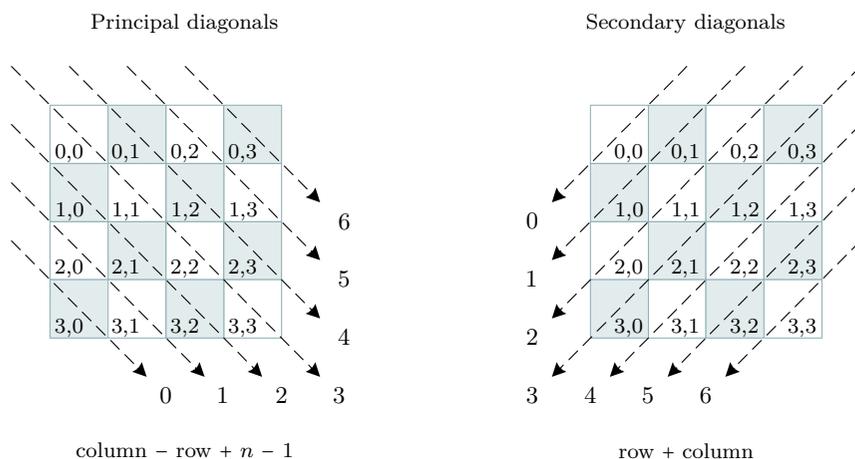


Figure 12.8 Indexing principal and secondary diagonals on a matrix or chessboard.

ing point for building the backtracking method that solves the problem. Although the final algorithm will introduce a few modifications, its structure will be very similar to the method that generates permutations.

Consider the method `generate_permutations_alt` and its input parameters. It permutes the items in the list `elements`, which can contain arbitrary numbers, characters, or other types of data. In this case the elements that we have to permute are the rows of the chessboard. Since they are simply the integers from 0 to $n - 1$, we will be able to write the code omitting the list `elements`. However, we will use the list `sol` that represents a partial solution containing rows, the parameter `i` that indicates the column where a new queen will be placed, and the Boolean list `available` that specifies which rows (candidates) are free to be incorporated in the partial solutions. We will change the name of this list to `free_rows`, since even though there might not be a queen in a particular row, the row might not be *available* for inclusion in the partial solution if there is a conflict related to a diagonal.

In addition to those parameters, we can use two more Boolean lists in order to indicate whether there are queens on the diagonals of the chessboard. There are two types of diagonals: (a) principal diagonals, which are parallel to the main diagonal that runs from the top-left to the bottom-right corners of the board; and (b) secondary diagonals, which are perpendicular to the principal diagonals. There are exactly $2n - 1$

of each kind, as illustrated in [Figure 12.8](#) for $n = 4$. Thus, we can use the Boolean lists `free_pdiags` and `free_sdiags`, both of length $2n - 1$, which will contain `True` values if there are no queens on the principal and secondary diagonals, respectively. The figure also shows how we can enumerate the diagonals, where the pair of numbers in a square indicate its row and column on the board. On the one hand, note that the sum of the column and row for squares along a specific secondary diagonal is constant. These sums range from 0 to $2n - 2$, and can be used as indices to `free_sdiags`. On the other hand, we can subtract the row from the column in order to enumerate the principal diagonals. However, this provides values from $-(n - 1)$ to $n - 1$. Since we need to use nonnegative indices, we can simply add $n - 1$ in order to obtain a valid index for a principal diagonal that will range from 0 to $2n - 2$. These operations allow us to quickly determine the diagonals in which a square lies according to its row and column.

12.3.1 Finding every solution

[Listing 12.5](#) shows a recursive backtracking algorithm that finds all of the solutions to the n -queens puzzle (for $n = 8$ there are 92 solutions, although only 12 are truly distinct in the sense that they cannot be obtained from others through rotations and reflections). In the base case the method processes a valid permutation if it is a complete solution (lines 6 and 7). For example, it could print it (see method `print_chessboard` in [Listing 12.6](#)), or draw queens on a chessboard image. The loop in line 12 is used to iterate through all of the possible candidates that could be appended to the partial solution. The variable `k` therefore represents rows of the chessboard. Subsequently, the `if` statement in lines 16 and 17 examines whether the `k`-th row is a valid candidate for the `i`-th column. In particular, it makes sure that the row and the corresponding diagonals are free (i.e., that they do not contain a queen). If the result is `True` the algorithm can proceed by incorporating the `k`-th row into the partial solution (line 20). Since this implies placing a queen on the chessboard, it is necessary to update the Boolean data structures in order to indicate that a row and two diagonals will no longer be free (lines 24–26). The method can then call itself (in lines 30 and 31) with the modified lists in order to continue placing queens on the next column (`i+1`). Lastly, after the recursive call the method needs to get ready to test whether it can place a queen in the next row, which would occur in the next iteration of the loop. Therefore, it is necessary

Listing 12.5 Code for finding all of the solutions to the n -queens puzzle.

```

1 def nqueens_all_sol(i, free_rows, free_pdiags,
2                     free_sdiags, sol):
3     n = len(sol)
4
5     # Test if the partial solution is a complete solution
6     if i == n:
7         print_chessboard(sol) # process the solution
8     else:
9
10        # Generate all possible candidates that could
11        # be introduced in the partial solution
12        for k in range(0, n):
13
14            # Check if the partial solution with the
15            # k-th candidate would be valid
16            if (free_rows[k] and free_pdiags[i - k + n - 1]
17                and free_sdiags[i + k]):
18
19                # Introduce candidate k in the partial solution
20                sol[i] = k
21
22                # Update data structures, indicating that
23                # candidate k is in the partial solution
24                free_rows[k] = False
25                free_pdiags[i - k + n - 1] = False
26                free_sdiags[i + k] = False
27
28                # Perform a recursive call in order to include
29                # more candidates in the partial solution
30                nqueens_all_sol(i + 1, free_rows, free_pdiags,
31                                free_sdiags, sol)
32
33                # Eliminate candidate k from the partial
34                # solution, and restore the data structures,
35                # indicating that candidate k is no longer
36                # in the partial solution
37                free_rows[k] = True
38                free_pdiags[i - k + n - 1] = True
39                free_sdiags[i + k] = True
40
41
42 def nqueens_wrapper(n):
43     free_rows = [True] * n
44     free_pdiags = [True] * (2 * n - 1)
45     free_sdiags = [True] * (2 * n - 1)
46     sol = [None] * n
47     nqueens_all_sol(0, free_rows, free_pdiags, free_sdiags, sol)

```

to restore the values of the Boolean lists, as if the k -th row had not been included in the partial solution. Finally, the algorithm modifies the lists specifying that the k -th row, and the diagonals related to the square at column i and row k , will be free of queens (lines 37–39).

12.3.2 Finding one solution

In many situations we are only interested in finding one solution to a problem. Moreover, some problems only have one solution (for example, the sudoku puzzle). Therefore, we can code backtracking methods that stop searching for solutions as soon as they find one. In Python it is possible to insert a `return` statement after finding a solution. For example, we can include it after the `print_chessboard(sol)` instruction in the `nqueens_all_sol` method of Listing 12.5. However, in many programming languages this is not possible.

There are many ways to implement a program that finds and processes one solution. Listing 12.6 shows one approach where the recursive method is a Boolean function, which returns `True` if it is able to find a solution. It is very similar to the code that finds all of the solutions. On the one hand, the method `nqueens_one_sol_wrapper` calls the recursive backtracking function with the `if` statement, and only prints the solution if the function is able to find it. Note that the list `sol` is mutable (it can be understood as a parameter passed by reference) and will end up storing the solution after the method call if it exists. Naturally, another option is to print the solution in the base case of the recursive function before returning. On the other hand, the function `nqueens_one_sol` first defines a Boolean variable `sol_found`, initialized to `False`, which indicates if the method has found a solution. The `for` loop is replaced by a `while` loop in order to stop iterating as soon as a solution is found. The body of the loop is identical to the one in the procedure `nqueens_all_sol`, but the result of the recursive call is assigned to the variable `sol_found`. Finally, the function can return the value in `sol_found` after executing the `while` loop.

12.4 SUBSET SUM PROBLEM

Given some set S of n positive integers, and a particular integer x , the goal of this problem is to determine whether the sum of some subset of elements of S is equal to x . Formally, if s_i denotes the i -th integer in S , the idea consists of checking whether there exists a subset $T \subseteq S$ such

Listing 12.6 Code for finding one solution to the n -queens puzzle.

```

1 def nqueens_one_sol(i, free_rows, free_pdiags,
2                     free_sdiags, sol):
3     n = len(sol)
4     sol_found = False
5
6     if i == n:
7         return True
8     else:
9         k = 0
10        while not sol_found and k < n:
11            if (free_rows[k] and free_pdiags[i - k + n - 1]
12                and free_sdiags[i + k]):
13
14                sol[i] = k
15
16                free_rows[k] = False
17                free_pdiags[i - k + n - 1] = False
18                free_sdiags[i + k] = False
19
20                sol_found = nqueens_one_sol(i + 1, free_rows,
21                                            free_pdiags,
22                                            free_sdiags, sol)
23
24                free_rows[k] = True
25                free_pdiags[i - k + n - 1] = True
26                free_sdiags[i + k] = True
27
28            k = k + 1
29
30        return sol_found
31
32
33 def nqueens_one_sol_wrapper(n):
34     free_rows = [True] * n
35     free_pdiags = [True] * (2 * n - 1)
36     free_sdiags = [True] * (2 * n - 1)
37     sol = [None] * n
38
39     if nqueens_one_sol(0, free_rows, free_pdiags,
40                       free_sdiags, sol):
41         print_chessboard(sol)
42
43
44 def print_chessboard(sol):
45     for i in range(0, len(sol)):
46         print(sol[i], ' ', end='')
47     print()

```

that:

$$\sum_{s_i \in T} s_i = x. \quad (12.1)$$

In particular, we will design a backtracking algorithm that prints every subset T of S that satisfies (12.1). For example, if $S = \{1, 2, 3, 5, 6, 7, 9\}$ and $x = 13$, the method should print the following five subsets on the console: $\{6, 7\}$, $\{2, 5, 6\}$, $\{1, 5, 7\}$, $\{1, 3, 9\}$, and $\{1, 2, 3, 7\}$.

A naive brute-force solution to the problem consists of generating every possible subset T of S , and testing whether (12.1) is `True`. This exhaustive search can be carried out through a procedure similar to `generate_subsets` in Listing 12.1. The set S could be represented by the input list `elements`, and the method would contain an additional parameter that stores the value of x . Furthermore, the procedure would have to verify (12.1) in the base case before printing a subset. In other words, it would have to check (12.1) in each leaf of the recursion tree.

However, it is possible to speed up the search by using a backtracking approach. Listing 12.7 shows a particular implementation where a partial solution (`sol`) represents the subset T . In order to avoid calculating the sum of the elements in the partial solution in every recursive call, it uses an additional accumulator parameter `psum` that contains the sum, and which is obviously initialized to zero in the wrapper method. The key observation that leads to the algorithm is that we can prune the recursion tree as soon as the sum of elements related to the partial solution is equal to x (i.e., when (12.1) is satisfied); or when it is greater than x , where it will not be possible to satisfy (12.1) by expanding the partial solution, since the elements of S are positive.

Firstly, the method includes a base case (in lines 3 and 4) in order to print the subset associated with the partial solution if it satisfies (12.1). This allows us to prune the tree at an internal node, of depth `i`, of the recursion tree. In the base case T can only contain the first `i` elements of S . In other words, only the first `i` components of the partial solution are meaningful. If the $n-i$ last entries can take arbitrary values, it would be necessary to pass the value of `i` to the printing method, and specify the “true” size of the partial solution. Instead, the algorithm calls the method `print_subset_binary` in Listing 12.1, since these last values will always be zero, indicating that the elements are not in the partial solution (T). This is achieved by initializing the partial solution with n zeros (in `print_subset_sum_wrapper`), denoting the empty set; and by updating the partial solution appropriately so that its $n-i$ last items

Listing 12.7 Backtracking code for solving the subset sum problem.

```

1 def print_subset_sum(i, sol, psum, elements, x):
2     # Base case
3     if psum == x:
4         print_subset_binary(sol, elements)
5     elif i < len(elements):
6         # Generate candidates
7         for k in range(0, 2):
8
9             # Check if recursion tree can be pruned
10            if psum + k * elements[i] <= x:
11
12                # Expand partial solution
13                sol[i] = k
14
15                # Update sum related to partial solution
16                psum = psum + k * elements[i]
17
18                # Try to expand partial solution
19                print_subset_sum(i + 1, sol, psum, elements, x)
20
21                # not necessary:
22                #psum = psum - k*elements[i]
23
24            # Make sure a 0 indicates the absence of an element
25            sol[i] = 0
26
27
28 def print_subset_sum_wrapper(elements, x):
29     sol = [0] * (len(elements))
30     print_subset_sum(0, sol, 0, elements, x)

```

are always zeros when reaching a base case (this is implemented through the assignment in line 25).

If the partial solution does not satisfy (12.1) the method can simply terminate if the partial solution has n elements, without carrying out any operation. Thus, with the condition in line 5 the algorithm only continues expanding partial solutions if $i < n$. In that case it uses a loop in order to process the two possibilities of ignoring ($k=0$) or including ($k=1$) the i -th element of S in the partial solution. Since k is a number we can use it within the expression $psum + k*elements[i]$ to indicate the new sum associated with the partial solution. Notice that when $k=0$ it does not change, while the method adds the i -th element of S when $k=1$.

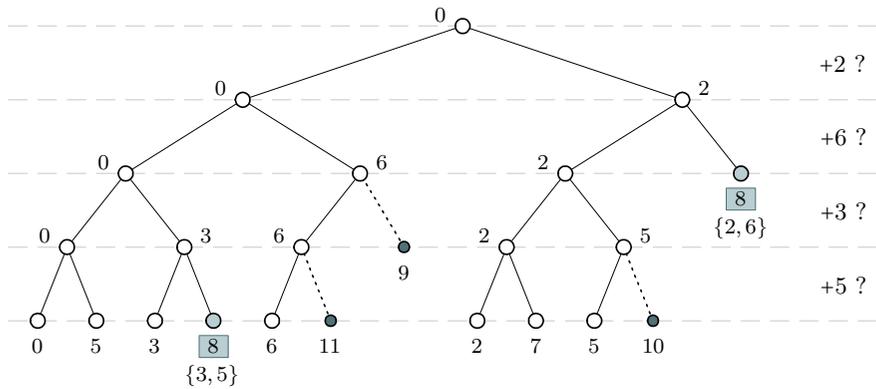


Figure 12.9 Recursion tree of the procedure that solves the subset sum problem for $S = \{2, 6, 3, 5\}$ and $x = 8$.

Therefore, the condition in line 10 makes sure that the new sum is less than or equal to x before including the element, and proceeding to make additional recursive calls. If the condition is **True** the method expands the partial solution with the corresponding candidate (line 13), updates `psum` (line 16), and carries out the recursive call (line 19). Afterwards, it is not necessary to restore the value of `psum` (in line 22) since in the first iteration of the loop $k=0$, which implies that the value of `psum` is not modified. Lastly, the algorithm sets the value of the partial solution to zero at position i when terminating, which is necessary in order to print the partial solutions correctly in the base case (its $n-i$ last items must always be zero).

Finally, Figure 12.9 shows the recursion tree of the method `print_subset_sum`, for $S = \{2, 6, 3, 5\}$ and $x = 8$. The partial solutions are constructed at each level as in the methods in Section 12.2.1. When descending through a left branch the algorithm does not include the i -th element of S in T , but it incorporates it when advancing down a right branch. In this case the numbers next to the nodes indicate the sum of the items included in the partial solutions (i.e., `psum`) when invoking the method. Observe that the tree is pruned as soon as it finds a solution that satisfies (12.1), or if it detects that the sum of the elements in T is greater than x . Lastly, a call to `print_subset_sum_wrapper([2, 6, 3, 5], 8)` produces the correct result:

```
{3, 5}
{2, 6}
```

Finally, observe that without the assignment in line 25, the partial solution would hold a 1 in its third position when reaching the base case associated with the subset $\{2, 6\}$, and the method would print $\{2, 6, 3\}$.

12.5 PATH THROUGH A MAZE

This section describes a backtracking algorithm for finding a path through a maze, such as the one illustrated in [Figure 12.10\(a\)](#). A maze is defined as a rectangular array of cells that can either be empty or constitute a wall. We will code it by using a list of lists (of characters), denoted as M , where the first list will contain the cells of the top row of the maze (from left to right), the second list will represent the row below the first one, and so on. In order to define a maze a user can write each row as a line of characters, separated by a space bar character, as shown in (b). Specifically, an ‘E’ indicates an empty cell, which can be traversed by a path, and a ‘W’ indicates a wall. Since it is time consuming to enter large mazes manually, it is convenient to store the array of characters in a file that can be loaded automatically when running the program (this is also beneficial for debugging). The method will also receive parameters that indicate the cells where a path enters and exits the maze. In the figure these cells are the top-left and bottom-right ones, respectively. Thus, the initial cell in the example is $M[0][0]$.

The backtracking algorithm carries out an exhaustive search for paths that start at the initial cell and arrive at the final cell. Having reached some particular cell in an intermediate step, the algorithm needs to analyze every possible path when advancing to a neighboring cell going upwards, downwards, rightwards, or leftwards. Since the final cell is in the bottom-right corner of the maze the algorithm can begin by searching paths that start going downwards, then towards the right, subsequently upwards, and finally towards the left. For this particular order of search the method takes the steps shown in [Figure 12.10\(c\)](#). Observe that when arriving at an empty cell the method first tries to expand the path downwards. If it cannot advance it will continue trying to move towards the right, then upwards, and finally towards the left. If it cannot proceed in any of the four directions it must backtrack to a previous cell, and continue to search for other alternative paths. In the example the initial cell is at position $(0, 0)$, corresponding to row 0 and column 0. The method will mark that cell as being part of the current path, and update $M[0][0]$ to ‘P’. The first move advances to position $(1, 0)$, and also marks the corresponding cell with a ‘P’. The method

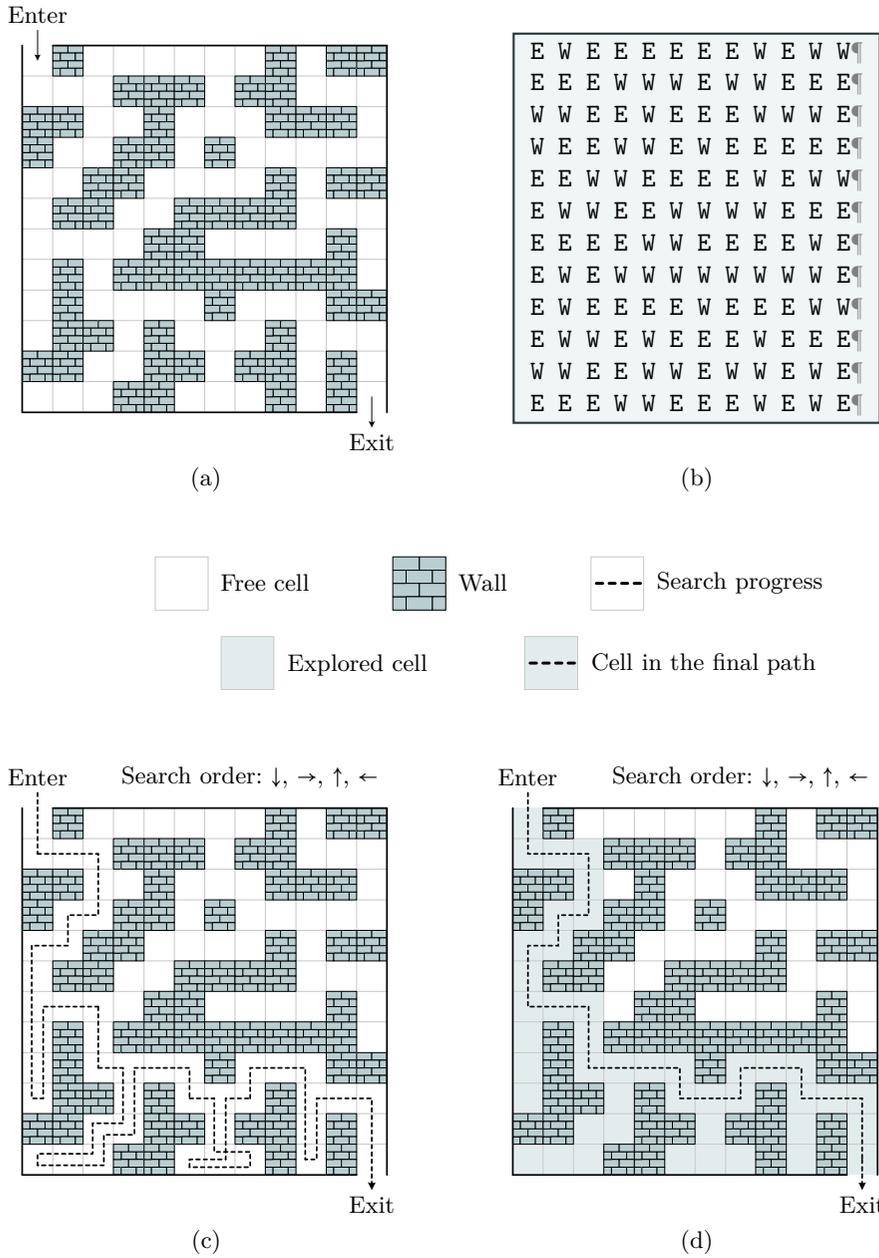


Figure 12.10 Problem of finding a path through a maze, and solution through backtracking when searching in a particular order.

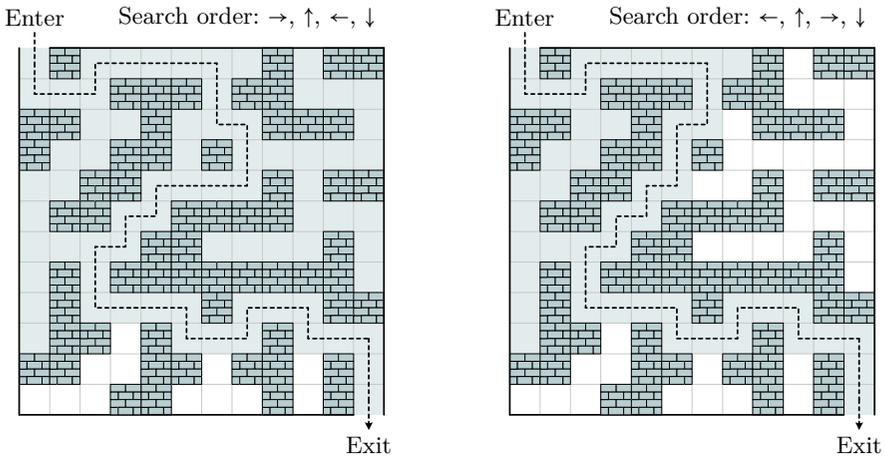


Figure 12.11 Different paths through a maze, depending on the search order.

then tries to advance downwards, but hits a wall located at $(2, 0)$. Since a path with that cell would not be valid, the algorithm continues by trying to advance towards the right, which it can do since the cell at $(1, 1)$ is empty. Eventually the path reaches the cell at $(6, 0)$, where the method will also begin by trying to advance downwards. The path ends up advancing to the cell at $(9, 0)$, where it hits a dead end. In that case, not only are there walls below and to the right, but going upwards is not allowed because the cell on top is already part of the path. Furthermore, going towards the left would mean exiting the limits of the maze. Since the algorithm cannot advance in any direction, it backtracks to the cell at $(8, 0)$, where it has not yet tried to advanced towards the right, upwards, or leftwards. However, these possibilities are also not allowed. This also occurs when returning to the cell at $(7, 0)$. After backtracking to the cell at $(6, 0)$, it explores new paths that go towards the right, after having exhaustively explored all of the possible paths going downwards. This process is repeated until the algorithm finds the exit cell. The solution is shown in (d), where the explored cells appear with a shaded background.

Backtracking algorithms can stop searching for solutions when they find the first one, or they can continue computing every solution. We will see an algorithm that halts as soon as it finds a path through the maze. In this regard, the order in which it examines the possible paths

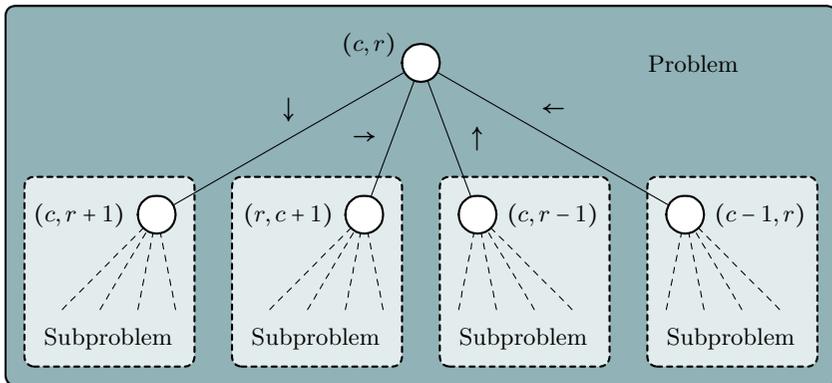


Figure 12.12 Decomposition of the problem of finding a path through a maze.

can lead to different solutions, as illustrated in [Figure 12.11](#). Observe that the explored cells are different in each case.

The solution to this problem is clearly not a permutation of cells. This can be seen, for instance, since the length of a path (i.e., a solution) is not fixed. Furthermore, although a path consists of a subset of cells, these appear ordered in a solution. Therefore, it is not appropriate to model a solution as a subset. Instead, we will use a matrix of characters to code a solution, where the cells in a particular path will be marked with a ‘P’ character. Thus, a partial solution will be the input maze that also contains a path from the initial cell to some other free cell. Having arrived at this last cell, the method has to consider including four candidate cells in the maze. Therefore, the problem can be decomposed recursively as shown in [Figure 12.12](#), where r and c denote a particular row and column, respectively. Every node of the recursion tree will have four possible children, which correspond to the four directions in which it is possible to advance. However, the algorithm will have to check that the new cell is free before expanding the path through a recursive call.

[Listing 12.8](#) shows a possible implementation of a backtracking algorithm that solves the problem. Firstly, the parameters of the wrapper method are the initial maze, and four integers that indicate the coordinates of the entering (`enter_row` and `enter_col`) and exiting (`exit_row` and `exit_col`) cells. It also declares the list `incr`, which specifies the search directions. Each tuple (x, y) in the list indicates how to increment the column c and row r of a cell in order to advance to a neighboring one. In other words, the new cell would be $(c + x, r + y)$. The order of

Listing 12.8 Backtracking code for finding a path through a maze.

```

1 def find_path_maze(M, row, col, incr, exit_row, exit_col):
2     # Base case: check if path found
3     if row == exit_row and col == exit_col:
4         return True # Solution found
5     else:
6         sol_found = False
7         # Generate candidates
8         k = 0
9         while not sol_found and k < 4:
10
11             # New candidate cell
12             new_col = col + incr[k][0]
13             new_row = row + incr[k][1]
14
15             # Test candidate validity
16             if (new_row >= 0 and new_row < len(M)
17                 and new_col >= 0 and new_col < len(M[0])
18                 and M[new_row][new_col] == 'E'):
19
20                 # Add to path (partial solution)
21                 M[new_row][new_col] = 'P'
22
23                 # Try to expand path starting at new cell
24                 sol_found = find_path_maze(
25                     M, new_row, new_col, incr,
26                     exit_row, exit_col)
27
28                 # Mark as empty if new cell not in solution
29                 if not sol_found:
30                     M[new_row][new_col] = 'E'
31
32             k = k + 1
33
34         return sol_found
35
36
37 def find_path_maze_wrapper(M, enter_row, enter_col,
38                             exit_row, exit_col):
39     # search directions
40     incr = [(0, 1), (1, 0), (0, -1), (-1, 0)]
41
42     M[enter_row][enter_col] = 'P'
43     return find_path_maze(M, enter_row, enter_col, incr,
44                             exit_row, exit_col)

```

search for this algorithm is therefore: down, right, up, and left. Observe that adding a unit to a row implies going down (since the first row is located at the top of the maze), while adding one to a column signifies moving towards the right. Lastly, the wrapper method indicates that the initial cell will form part of the solution path, and returns the (Boolean) result of calling the recursive backtracking method. The parameter `M` of `find_path_maze` is both the initial maze and the partial solution. The first two parameters of the recursive function indicate the coordinates of the last cell of the partial path `M`, and the algorithm will try to expand it advancing to one of its neighboring cells. Finally, the method receives `incr`, and the coordinates of the exiting cell.

The recursive function `find_path_maze` returns `True` if it finds a path through the maze, where the solution would be stored precisely in `M`. It declares the variable `sol_found`, initialized as `False`, which is set to `True` at the base case if the algorithm finds a complete solution. Otherwise, it uses a `while` loop to generate the four candidate cells, but one that terminates as soon as a solution is found. The variables `new_col` and `new_row` constitute the new candidate (lines 12 and 13). Subsequently, the algorithm examines whether it is valid. In particular, it has to be within the limits of the maze (lines 16 and 17), and it has to be empty (line 18). If the cell does not violate the constraints of the problem the method incorporates it to the path (line 21), and calls the recursive method in lines 24–26, where the output is stored in `sol_found`. If a solution is not found the cell will not belong to the path. Therefore, it sets its value back to ‘E’, indicating that it is empty (line 30). This is necessary since the path in the final maze is determined through the cells marked as ‘P’. Without this condition all of the explored cells would contain a ‘P’ character. Lastly, the method can simply return the value of `sol_found` after exiting the loop (line 34).

Finally, [Listing 12.9](#) shows additional code that can be used in order to execute the program and draw the maze. The method `read_maze_from_file` reads a text file defining a maze and returns the corresponding list of lists. The basic iterative procedure `draw_maze` uses the Matplotlib package to depict a maze. Finally, the last lines of the code read a maze from a file, and draw it only if there exists a path through it, where the initial and final cells are the top-left and bottom-right ones, respectively.

Listing 12.9 Auxiliary code related to the backtracking methods for finding a path through a maze.

```

1 import matplotlib.pyplot as plt
2 from matplotlib.patches import Rectangle
3
4 def read_maze_from_file(filename):
5     file = open(filename, 'r')
6     M = []
7     for line in file.readlines():
8         M.append([x[0] for x in line.split(' ')])
9     file.close()
10    return M
11
12 gray = (0.75, 0.75, 0.75)
13 black = (0, 0, 0)
14 red = (0.75, 0, 0)
15 green = (0, 0.75, 0)
16
17 def draw_maze(M, enter_row, enter_col, exit_row, exit_col):
18     nrows = len(M)
19     ncols = len(M[0])
20     fig = plt.figure()
21     fig.patch.set_facecolor('white')
22     ax = plt.gca()
23
24     if enter_row is not None and enter_col is not None:
25         ax.add_patch(Rectangle((enter_col, nrows - enter_row),
26                               1, -1, linewidth=0, facecolor=green,
27                               fill=True))
28     if exit_row is not None and exit_col is not None:
29         ax.add_patch(Rectangle((exit_col, nrows - exit_row),
30                               1, -1, linewidth=0, facecolor=red,
31                               fill=True))
32
33     for row in range(0, nrows):
34         for col in range(0, ncols):
35             if M[row][col] == 'W':
36                 ax.add_patch(Rectangle((col, nrows - row), 1, -1,
37                                       linewidth=0, facecolor=gray))
38             elif M[row][col] == 'P':
39                 circ = plt.Circle((col + 0.5, nrows - row - 0.5),
40                                  radius=0.15, color=black, fill=True)
41                 ax.add_patch(circ)
42
43     ax.add_patch(Rectangle((0, 0), ncols, nrows, edgecolor=black,
44                           fill=False))
45     plt.axis('equal')
46     plt.axis('off')
47     plt.show()
48
49 M = read_maze_from_file('maze_01.txt') # some file
50 # Enter at top-left, exit at bottom-right
51 if find_path_maze_wrapper(M, 0, 0, len(M) - 1, len(M[0]) - 1):
52     draw_maze(M, 0, 0, len(M) - 1, len(M[0]) - 1)

```

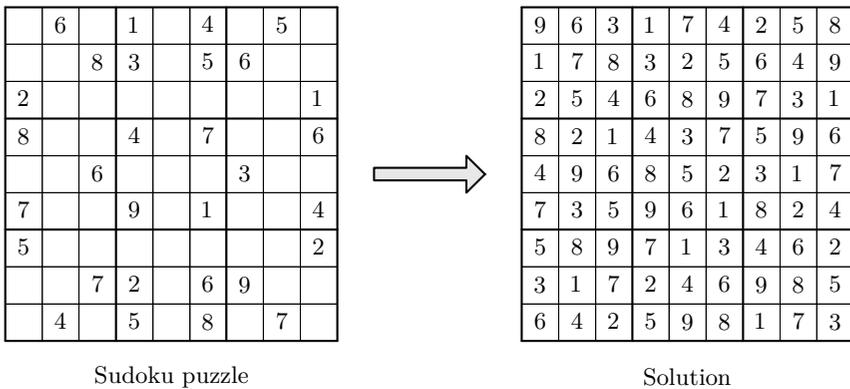


Figure 12.13 An instance of the sudoku puzzle and its solution.

12.6 THE SUDOKU PUZZLE

The sudoku puzzle is a constraint satisfaction problem where the goal is to fill a 9×9 grid with the digits from 1 to 9 (they can be any nine different symbols, since their numerical values are irrelevant). A particular digit can appear only once in each row, each column, and each of nine 3×3 non-overlapping subgrids, also called boxes, which cover the 9×9 grid. [Figure 12.13](#) shows an instance of the problem, which consists of a partially filled grid with fixed initial digits. The problem is well-posed when the puzzle has a unique solution (the initial grid must contain at least 17 digits).

We will use a list of lists of digits to implement the grid, where a zero represents an empty cell. The partial solutions will be partially filled grids, and the algorithm must enumerate the cells somehow in order to expand the partial solutions with new digits. The algorithm described below considers expanding them in row-major order, where it will begin inserting valid digits (i.e., candidates) in the top row from left to right, then in the second row, and so on. Thus, the algorithm will have found a solution if it is able to place a valid digit in the bottom-right cell.

In a recursive case the method generates nine possible candidates to include in an empty cell. Thus, a node of the recursion tree could have up to nine potential children, as shown in [Figure 12.14\(a\)](#). However, the problem requires handling a scenario that has not appeared in previous problems. When expanding the partial solution some of the digits in the grid will be the initial fixed ones. When the algorithm processes a cell

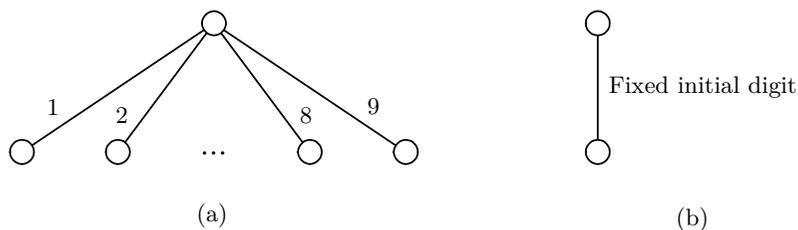


Figure 12.14 Recursive cases for the sudoku solver.

with one of these digits it must skip it (it cannot expand the partial solution for that cell), and carry out a recursive call that processes the following cell. This requires incorporating a second recursive case that is illustrated in (b).

Listings 12.10 and 12.11 show a backtracking procedure that solves the problem, together with several auxiliary methods. The recursive method `solve_sudoku_all_solutions` assumes that a sudoku might not be well-posed, and could potentially have several solutions (or even 0). Thus, it prints all of the valid solutions to a given sudoku grid. For example, if the top row of the sudoku in Figure 12.13 is replaced by an empty row, there are 10 different ways to fill the sudoku grid with digits.

The inputs to the recursive procedure are the coordinates (`row` and `col`) of a cell where it will try to include digits in order to expand the partial solution represented by the list of lists of digits `S`. Since the procedure expands the partial solution in row-major order, starting at cell $(0, 0)$, it will have obtained a valid solution when `row` is equal to 9. In that base case it can simply print the sudoku grid (line 4). Otherwise, the method checks whether the current cell is not empty (i.e., if it contains one of the initial fixed digits). If the result is `True` it skips the cell, and continues by invoking the recursive method (line 14) with the coordinates of the next cell in the row-major order (computed in line 11).

In the second recursive case it uses a loop to generate the nine possible candidates to include in the empty cell. Afterwards, in line 20 it analyzes whether it is feasible to incorporate candidate `k` in the partial solution `S`, at cell `(row,col)`. If it is, the method includes the candidate in the partial solution (line 23), and continues carrying out a recursive call with the next cell (line 29). When the loop finishes, the method needs to undo the changes made to the cell, leaving it empty (line 32). This is necessary for future explorations of solutions. Note that the cell has to

Listing 12.10 Code for solving a sudoku puzzle.

```

1 def solve_sudoku_all_sols(row, col, S):
2     # Check if sudoku is complete
3     if row == 9:
4         print_sudoku(S) # print the completed sudoku
5         print()
6     else:
7         # Check if digit S[row][col] is an initial fixed symbol
8         if S[row][col] != 0:
9
10            # Advance to a new cell in row-major order
11            (new_row, new_col) = advance(row, col)
12
13            # Try to expand the partial solution
14            solve_sudoku_all_sols(new_row, new_col, S)
15        else:
16            # Generate candidate digits
17            for k in range(1, 10):
18
19                # Check if digit k is a valid candidate
20                if is_valid_candidate(row, col, k, S):
21
22                    # Include digit in cell (row,col)
23                    S[row][col] = k
24
25                    # Advance to a new cell in row-major order
26                    (new_row, new_col) = advance(row, col)
27
28                    # Try to expand the partial solution
29                    solve_sudoku_all_sols(new_row, new_col, S)
30
31                # Empty cell (i,j)
32                S[row][col] = 0
33
34
35    # Compute the next cell in row-major order
36    def advance(row, col):
37        if col == 8:
38            return (row + 1, 0)
39        else:
40            return (row, col + 1)

```

be free the next time the algorithm processes it after backtracking to an earlier cell.

Listing 12.11 Auxiliary code for solving a sudoku puzzle.

```

1 import math
2
3 # Check if the digit at cell (row,col) is valid
4 def is_valid_candidate(row, col, digit, S):
5     # Check conflict in column
6     for k in range(0, 9):
7         if k != col and digit == S[row][k]:
8             return False
9
10    # Check conflict in row
11    for k in range(0, 9):
12        if k != row and digit == S[k][col]:
13            return False
14
15    # Check conflict in box
16    box_row = math.floor(row / 3)
17    box_col = math.floor(col / 3)
18    for k in range(0, 3):
19        for m in range(0, 3):
20            if (row != 3 * box_row + k
21                and col != 3 * box_col + m):
22                if digit == S[3 * box_row + k][3 * box_col + m]:
23                    return False
24
25    return True
26
27 # Read a sudoku grid from a text file
28 def read_sudoku(filename):
29     file = open(filename, 'r')
30     S = [[None] * 9] * 9
31     i = 0
32     for line in file.readlines():
33         S[i] = [int(x) for x in line.split(' ')]
34         i = i + 1
35     file.close()
36     return S
37
38 # Print a sudoku grid on the console
39 def print_sudoku(S):
40     for s in S:
41         print(*s)
42
43 S = read_sudoku('sudoku01_input.txt') # Some file
44 solve_sudoku_all_sols(0, 0, S)

```

The code also includes: (a) a function that returns a tuple with the coordinates of the next cell in row-major order; (b) a function that determines whether the digit placed at some row and column violates the constraints of the problem, where the variables `box_row` and `box_col` are the top-left cells of the 3×3 boxes; (c) a function for reading a sudoku grid from a text file, where each row contains the nine initial digits of a sudoku row, separated by space bar characters; (d) a method for printing the grid; and (e) code for reading and solving a sudoku.

12.7 0-1 KNAPSACK PROBLEM

Backtracking algorithms are also used to solve optimization problems. In the 0-1 knapsack problem there are n objects with values v_i , and weights w_i , for $i = 0, \dots, n-1$. The goal of the problem is to introduce a subset of these objects in a knapsack with total weight capacity C , such that the sum of their values is maximized. Formally, the optimization problem can be written as:

$$\begin{aligned} & \text{maximize} && \sum_{i=0}^{n-1} x_i v_i \\ & \mathbf{x} && \\ & \text{subject to} && \sum_{i=0}^{n-1} x_i w_i \leq C, \\ & && x_i \in \{0, 1\}, \quad i = 0, \dots, n-1. \end{aligned}$$

The vector or list $\mathbf{x} = [x_1, \dots, x_n]$ is the variable of the problem, whose components can be either 0 or 1. In particular, the i -th object is introduced in the knapsack when $x_i = 1$. Therefore, \mathbf{x} plays the same role as the binary list in the subset generation or subset sum problem. The sum $\sum_{i=0}^{n-1} x_i v_i$ is called the objective function, and simply adds the values of the objects that are in the knapsack. The constraint $\sum_{i=0}^{n-1} x_i w_i \leq C$ indicates that the sum of their weights cannot exceed the capacity C .

The following subsections describe two approaches that perform an exhaustive search for the **optimal solution**. The first is a standard backtracking algorithm that prunes the recursion tree when the partial solution violates the constraints of the problem (i.e., when the sum of the weights exceeds the capacity of the knapsack). The second approach uses an algorithm design paradigm known as **branch and bound**, which is closely related to backtracking. The method enhances the search by also pruning the recursion tree as soon as the algorithm detects that it will

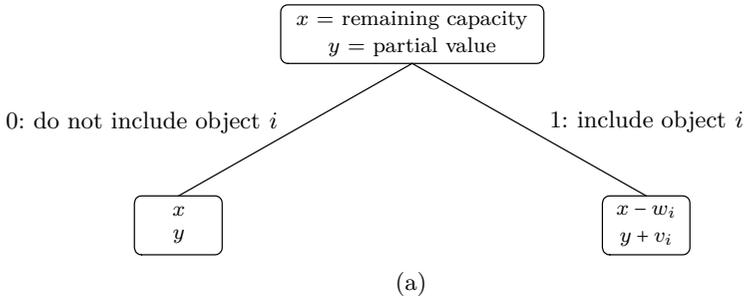
not be able to improve the best solution found in previous steps (by expanding a particular partial solution).

12.7.1 Standard backtracking algorithm

Since the solution is a subset of the n objects we can develop an algorithm whose recursion tree has the same binary structure as the one related to the algorithms for the subset generation or subset sum problem. [Figure 12.15](#) illustrates the recursion tree for weights $\mathbf{w} = [3, 6, 9, 5]$, values $\mathbf{v} = [7, 2, 10, 4]$, and knapsack capacity $C = 15$, which will be input parameters to the recursive method. The nodes represent method calls with specific partial solutions as parameters, where the numbers depicted inside the nodes specify the remaining capacity in the knapsack, and the sum of the values of the objects included in it, as shown in (a). For a given index i in the partial solution, the algorithm can discard introducing the associated object in the knapsack (left child), or it can include it (right child). If it discards the object both of the numbers in the child node do not vary. However, if an element is introduced in the knapsack its remaining capacity decreases by the weight of the object w_i , while the total value of the objects in the knapsack increases by v_i .

The algorithm solves an optimization problem. Thus, it needs to keep track of the best solution found in previous steps, and must update it if it encounters a better one. The full recursion tree is shown in (b), where the shaded nodes indicate the method calls that update this best solution. Assuming that the initial optimal value is a negative number, the method would first update the optimal solution when reaching the first leaf. In that case the partial solution encodes the empty set, where the sum of values is obviously zero. Continuing the tree traversal, the method updates the best solution after introducing the last object in the knapsack ($w_3 = 5$, and $v_3 = 4$). Thus, the remaining capacity at the leaf node is 10, and the total value of the partial solution is 4. The process continues, and updates the best solution three more times (the corresponding best values are 10, 14, and 17). The last update naturally determines the optimal solution to the problem. In this case it consists of the subset formed by the first and third objects, where the optimal sum of values is 17. Lastly, it is also important to note how the method prunes the binary recursion tree. In particular, the remaining capacity associated with a valid partial solution (and node) cannot be negative.

[Listing 12.12](#) shows a possible implementation of the backtracking algorithm. Firstly, similarly to other methods, its input parameters in-



$w = [3, 6, 9, 5]$
 $v = [7, 2, 10, 4]$
 $C = 15$

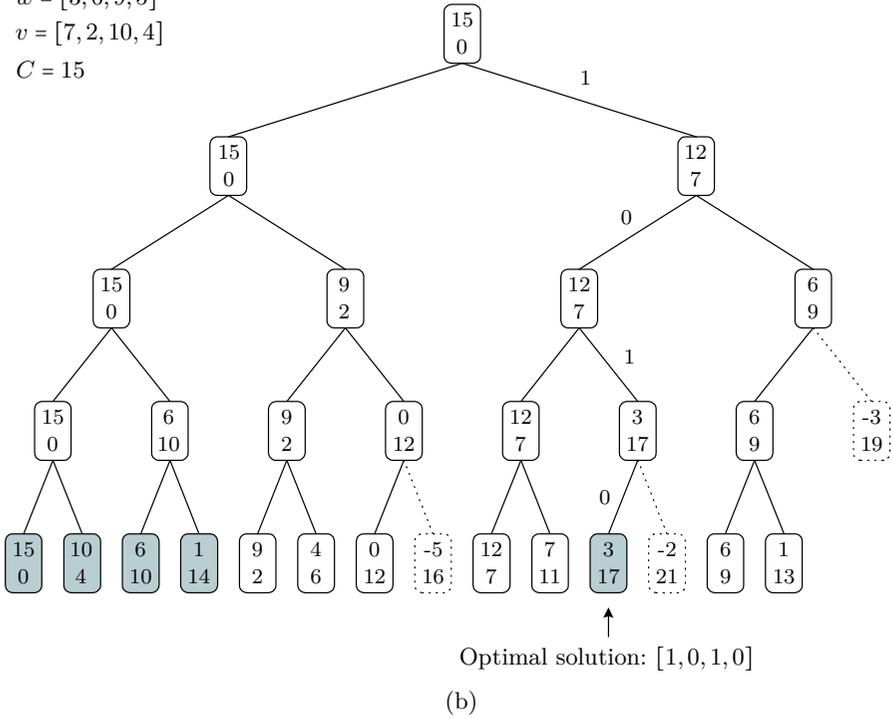


Figure 12.15 Recursion tree of a backtracking algorithm for the 0-1 knapsack problem.

Listing 12.12 Backtracking code for solving the 0-1 knapsack problem.

```

1 def knapsack_0_1(i, w_left, current_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             if k * w[i] <= w_left:
17
18                 # Expand partial solution
19                 sol[i] = k
20
21                 # Update remaining capacity and partial value
22                 new_w_left = w_left - k * w[i]
23                 new_current_v = current_v + k * v[i]
24
25                 # Try to expand partial solution
26                 opt_v = knapsack_0_1(i + 1, new_w_left,
27                                     new_current_v, sol,
28                                     opt_sol, opt_v, w, v, C)
29
30     # return value of optimal solution found so far
31     return opt_v
32
33
34 def knapsack_0_1_wrapper(w, v, C):
35     sol = [0] * (len(w))
36     opt_sol = [0] * (len(w))
37     total_v = knapsack_0_1(0, C, 0, sol, opt_sol, -1, w, v, C)
38     print_knapsack_solution(opt_sol, w, v, C, total_v)

```

clude the partial solution `sol` and the index `i` related to the object that may be introduced in the knapsack. The method also receives the lists of weights `w`, values `v`, and the capacity `C`. With these parameters it is possible to compute the remaining capacity and the accumulated sum of values of the objects in the knapsack, in `i+1` steps. However, it is more

efficient to include two extra parameters that contain these quantities. In particular, `w_left` stores the remaining capacity, while `current_v` holds the sum of the values of the objects. The advantage is that they can be updated in a single step.

Furthermore, the method needs to store the best solution found as it is executed. The parameter `opt_sol` stores this information, which will be a copy of the partial solution when reaching a leaf of the recursion tree. Lastly, `opt_v` contains the optimum sum of values associated with `opt_sol`. This parameter is also redundant in the sense that it is possible to compute its value given `opt_sol`. However, it is more efficient to pass this information as a parameter.

As usual, the recursive function begins by analyzing the base case. If the method has completed a partial solution (line 4) it checks whether the current value of the partial solution is greater than the best value found at any previous step (line 6). If the result is `True` it updates `opt_v`, and overwrites the values of `opt_sol` with the ones in `sol`. The method will later return `opt_v` in line 31. The values of the list `opt_sol` will be available after executing the method since it is a mutable object.

In the recursive case the method generates the candidates (zero or one) to include in the binary partial solution list, in line 13. It then uses the condition `k*w[i] <= w_left` to check if it can prune the recursion tree. On the one hand, if `k = 0` the object is not introduced in the knapsack and the condition is always `True` since `w_left` is nonnegative. However, if `k = 1` the algorithm can only include the object in the knapsack if it fits. In other words, w_i must be less than or equal to the remaining capacity of the knapsack. The method would continue by updating the partial solution, the remaining knapsack capacity, and the sum of the values of the objects in the knapsack. The function carries out a recursive call with these new parameters, storing the result in `opt_v`, which ends up returning in line 31.

Regarding the wrapper function, `i` is initially 0, `w_left` stores the total capacity C , while `current_v` is 0 since the knapsack is empty. The optimal value `opt_v` can be either 0 or some negative number. In [Figure 12.15](#) we have assumed that it is negative. This simply implies that the method will update the optimal solution even when it consists of an empty set (at the leftmost leaf).

Finally, [Listing 12.13](#) shows a simple iterative code that can be used to print a solution to the problem. Its optimal value can be computed from the partial solution and the list of values. However, the method includes such value as a parameter since the function `knapsack_0_1`

Listing 12.13 Auxiliary code related to the 0-1 knapsack problem.

```

1 def print_knapsack_solution(sol, w, v, C, opt_value):
2     n = len(sol)
3     k = 0
4     while k < n and sol[k] == 0:
5         k = k + 1
6
7     total_weight = 0
8     if k < n:
9         print('(' + w[k], ', ', v[k], ')', sep='', end='')
10        total_weight = total_weight + w[k]
11
12        for i in range(k + 1, n):
13            if sol[i] == 1:
14                total_weight = total_weight + w[i]
15                print(' + ', sep='', end='')
16                print('(' + w[i], ', ', v[i], ')', sep='', end='')
17
18        print(' => ', '(' + total_weight,
19              ', ', opt_value, ')', sep='')
20
21
22 w = [3, 6, 9, 5] # List of object weights
23 v = [7, 2, 10, 4] # List of object values
24 C = 15 # Weight capacity of the knapsack
25 knapsack_0_1_wrapper(w, v, C)

```

returns it, and is therefore readily available in `knapsack_0_1_wrapper`. The last lines simply define an instance of the problem, and compute its solution.

12.7.2 Branch and bound algorithm

Branch and bound can be viewed as a variant of backtracking that performs a more efficient exhaustive search of solutions to discrete and combinatorial optimization problems. The idea consists of using bounds in order to prune the recursion tree, not only when a partial solution does not satisfy the constraints of the problem, but also when it is certain that expanding the partial solution will not lead to a better solution than the best one found in previous steps.

The key to the branch and bound algorithm that we will examine for the 0-1 knapsack problem relies on using an extra parameter to store the maximum sum of values that could be obtained by expanding a partial

solution. If at some method call this value is smaller than the best value found by the algorithm in previous steps, it will not continue to expand the partial solution, since it will not be able to obtain a better solution. This allows us to prune the recursion tree at more nodes, which can lead to a considerably more efficient search.

Figure 12.16 illustrates the recursion tree for the same weights, values, and knapsack capacity as in Figure 12.15. In this case, the numbers inside the nodes indicate the partial value, and a bound on the maximum possible value that we can obtain by expanding the associated partial solution, as shown in (a). Initially, the partial value is 0, and the bound is the sum of all of the values of the objects (i.e., $23 = 7 + 2 + 10 + 4$), as illustrated in (b), since at first we have to contemplate the case where every object fits in the knapsack.

Each internal node of depth i can have two children. Descending through the left branch implies not introducing object i in the knapsack. Therefore, the partial value does not change, but the bound is decreased by v_i , since the object will not contribute its value to the total sum of a complete solution. Instead, the object is introduced in the knapsack when descending through the right branch. This implies adding v_i to the partial value, but the bound remains unaltered.

Similarly to the example in Figure 12.15, the method updates the best solution found so far in the shaded leaves. In addition, the nodes drawn with a dotted contour indicate method calls that are not carried out because the sum of the weights of the objects exceeds the knapsack's capacity. Furthermore, the figure shows a new type of node with a lighter dashed contour. The algorithm also discards the method calls associated with these nodes since the value of the bound is smaller than the best sum of values encountered previously. For instance, the best value after reaching the fourth leaf is 14. Afterwards, consider reaching the node with partial value 2 and bound 16. Discarding the third object implies reducing the bound by 10. This means that the maximum sum of values that it is possible to obtain by expanding the partial solution is $16 - 10 = 6$. Since $6 < 14$ (depicted underneath the node), the algorithm avoids calling the method, pruning the recursion tree. Observe that it has the same structure as the recursion tree in Figure 12.15, but it contains less nodes since it prunes the tree on more occasions. In practice, this enhancement can have a dramatic effect regarding efficiency.

Listing 12.14 shows a branch and bound code for solving the problem that is very similar to the one in Listing 12.12. The main difference between them is the new (fourth) parameter `max_v` that stores the bound.

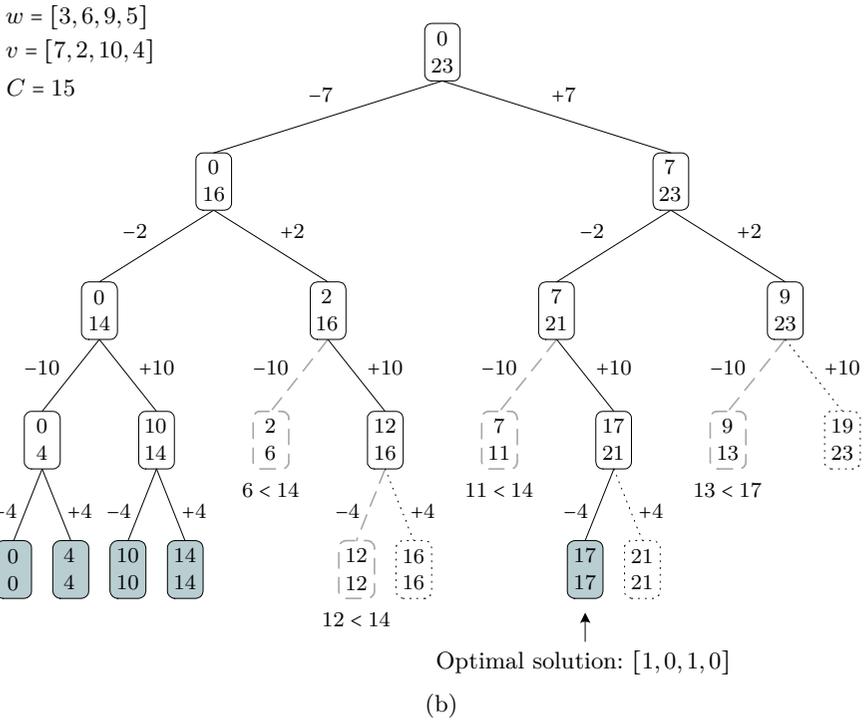
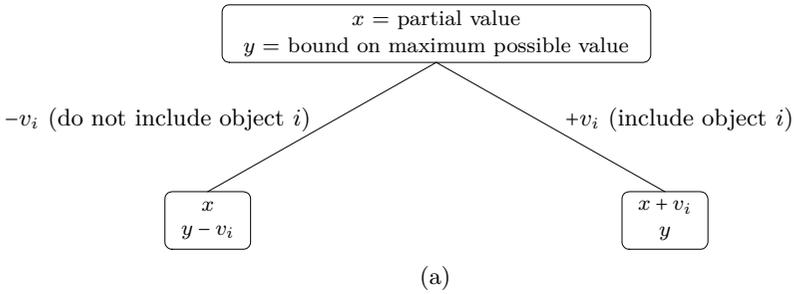


Figure 12.16 Recursion tree of a branch and bound algorithm for the 0-1 knapsack problem.

Listing 12.14 Branch and bound code for solving the 0-1 knapsack problem.

```

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

```

Listing 12.15 Auxiliary code for the branch and bound algorithm related to the 0-1 knapsack problem.

```

1 def knapsack_0_1_branch_and_bound_wrapper(w, v, C):
2     sol = [0] * (len(w))
3     opt_sol = [0] * (len(w))
4     total_v = knapsack_0_1_bnb(0, C, 0, sum(v), sol,
5                               opt_sol, -1, w, v, C)
6     print_knapsack_solution(opt_sol, w, v, C, total_v)
7
8
9 w = [3, 6, 9, 5] # List of object weights
10 v = [7, 2, 10, 4] # List of object values
11 C = 15 # Weight capacity of the knapsack
12 knapsack_0_1_branch_and_bound_wrapper(w, v, C)

```

Note that it is initialized to the sum of all of the values of the objects in the wrapper method. In the recursive case, after making sure that a partial solution with a new candidate is valid (in line 17), the method computes the new value of the bound `new_max_v` in line 20. Observe that when $k = 0$ the new bound is decreased by v_i , while when $k = 1$ it remains unaltered. Subsequently, the method uses another `if` statement to check if it can prune the tree according to the new bound and the optimal value of a solution computed in earlier calls (line 24). The rest of the code is analogous to the backtracking algorithm. Finally, [Listing 12.15](#) contains an associated wrapper method, defines an instance of the problem (through `w`, `v`, and `C`), and solves it.

12.8 EXERCISES

Exercise 12.1 — There are numerous ways to build algorithms that generate subsets of elements. The methods described in [Section 12.2.1](#) were based on binary recursion trees. The goal of this exercise is to implement alternative procedures that also print all of the subsets of n items provided in a list. However, they must generate a subset at each node of the recursion tree illustrated in [Figure 12.17](#). Observe that there are exactly 2^n nodes, and the labels 0, 1, and 2 represent indices of the elements of an initial input list $[a, b, c]$. Furthermore, instead of using binary lists to indicate the presence of items in a subset, the partial solutions will contain precisely these indices. For example, the list $[0, 2]$

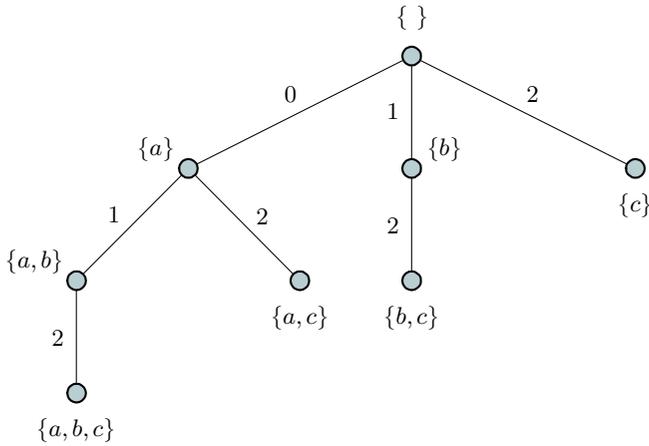


Figure 12.17 Alternative recursion tree of an algorithm that generates all of the subsets of three items.

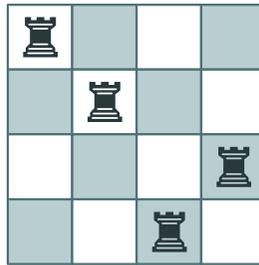


Figure 12.18 One solution to the four-rooks puzzle.

will represent $\{a, c\}$. Therefore, partial solutions will also correspond to complete solutions.

Exercise 12.2 — Implement a backtracking algorithm that prints all of the solutions to the n -rooks puzzle. It is analogous to the n -queens puzzle, but uses rooks instead of queens. Since rooks can only move vertically and horizontally, two (or more) rooks cannot appear in the same row or column. However, several can appear on a same diagonal. [Figure 12.18](#) shows a solution to the puzzle.

Exercise 12.3 — Implement a backtracking function that counts the number of valid solutions to a sudoku puzzle that may not be well-posed (i.e., it may have 0, 1, or more solutions).

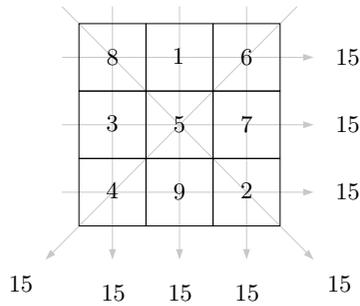
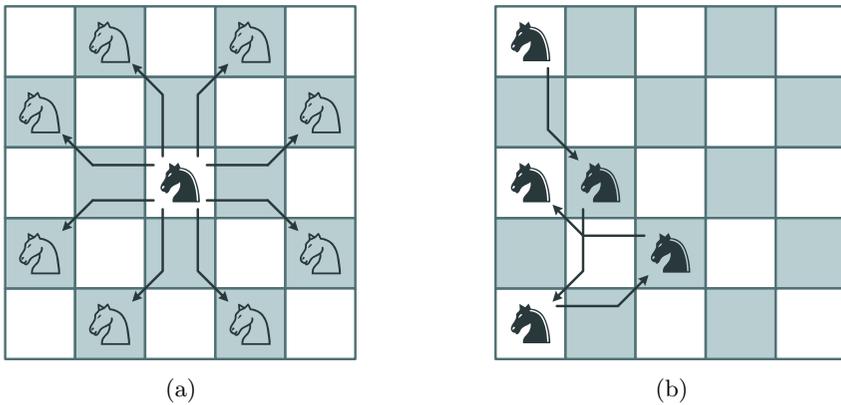
Figure 12.19 A 3×3 magic square.

Figure 12.20 Chess knight moves.

Exercise 12.4 — A magic square is an $n \times n$ grid of the first n^2 positive integers such that the sum of the elements in each row, column, and diagonal is equal. The particular sum is called the “magic constant,” and is $n(n^2 + 1)/2$ for a general n . Figure 12.19 shows an example of a 3×3 magic square, where the magic constant is 15. Design a backtracking algorithm that prints every possible 3×3 magic square. Note: it is not necessary to implement a method that finds magic squares for a general n . What problem could we run into when computing magic squares of larger values of n ?

Exercise 12.5 — A knight can jump from a square to another one in L -shaped patterns, as shown in Figure 12.20(a), where (b) shows a sequence of four moves. The “knight’s tour” problem consists of determining a sequence of moves of a knight on an $n \times n$ chessboard, when starting from

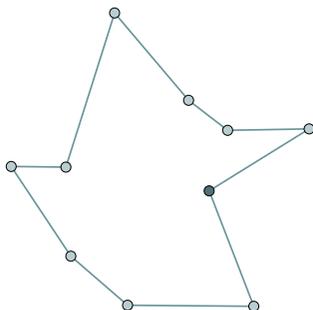


Figure 12.21 An instance and solution to the traveling salesman problem.

a specific square, such that it visits every square only once. Implement a backtracking algorithm that provides one knight’s tour, and test it for $n = 5$ and $n = 6$. Besides n , the method will receive the coordinates of an initial square on the chessboard. Finally, it is not necessary to search for a “closed” tour that ends at a square that is one move away from the initial square.

Exercise 12.6 — The “traveling salesman problem” is a classical discrete optimization problem. Given n cities, the goal is to find the shortest path that visits each one exactly once and returns to the starting city. Figure 12.21 shows the shortest path for 10 cities. Implement a backtracking algorithm that solves the problem. Assume that the locations of the cities are specified in a text file, where each line contains the x and y coordinates of the cities on a two-dimensional map. The file can be read with the `loadtxt` method from the NumPy package. The starting city will be the one specified in the first line of the file. In addition, consider Euclidean distances between cities (the salesman travels from one city to another by following a straight line). These can be computed, for example, with the method `pdist` in the SciPy package. In particular, the recursive backtracking method should receive a matrix of distances between the cities (their locations are not necessary when using this distance matrix). Finally, test the code for $n \leq 10$.

Exercise 12.7 — The following problem is known as “tug of war.” Given a nonempty set of n numbers, where n is even, the goal consists of dividing it into two subsets of $n/2$ elements in order to minimize the absolute difference between the sum of the numbers in each subset. For

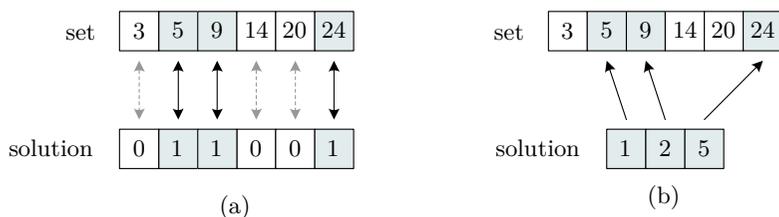


Figure 12.22 Two ways to represent a solution for the tug of war problem.

example, given the set $\{3, 5, 9, 14, 20, 24\}$, the optimal way to partition it leads to the two subsets: $\{5, 9, 24\}$, and $\{3, 14, 20\}$. The sums of their elements are 38 and 37, and the absolute difference between these sums is 1.

Implement a backtracking algorithm based on generating subsets that solves this problem. Assuming that the input set is coded as a list, the solution would be a binary list of length n with $n/2$ zeros, and $n/2$ ones. The positions of the zeros (or ones) would indicate the locations of the elements of a particular subset. In the example, one solution could be the list $[0, 1, 1, 0, 0, 1]$ (the list $[1, 0, 0, 1, 1, 0]$ would be equivalent), representing the subset $\{5, 9, 24\}$.

In addition, implement a more efficient strategy where the solution is a list \mathbf{s} of length $n/2$ whose elements appear in increasing order, and correspond to the locations of the elements of a particular subset. For example, the subset $\{5, 9, 24\}$ would be represented by the list $[1, 2, 5]$. Figure 12.22 shows these two ways to represent a solution.

Exercise 12.8 — In the problem described in Section 12.4 the goal consisted of printing every subset T of a set of n positive integers S , for which the sum of the elements of T was equal to some integer value x . Design an alternative recursive function that instead computes the (valid) subset T with the least cardinality (i.e., number of elements). In particular, it will store the optimal subset in a list, and will return the optimal cardinality. For example, for $S = \{1, 2, 3, 5, 6, 7, 9\}$ and $x = 13$, the method would store the subset $\{6, 7\}$, and return the value 2. In addition, note that in this optimization problem the optimal subset may not be unique, but the optimal cardinality is. For instance, for $S = \{2, 6, 3, 5\}$ and $x = 8$, both $\{2, 6\}$ and $\{3, 5\}$ would be optimal subsets. Code the partial (and optimal) solution as a list of binary digits, similarly to the method in Listing 12.7. Also, prune the tree considering the validity of a solution and the best cardinality found. Finally, code a wrapper method

that computes the optimal subset, and prints it if it exists (it may not be possible to find a subset T of S whose elements add up to x).

Further reading

MONOGRAPHS IN RECURSION

This book has provided a broad coverage of recursion, containing essential topics for designing recursive algorithms. However, the reader can find additional aspects, examples, and implementation details in other references. In particular, the following book:

- Jeffrey Soden Rohl. *Recursion Via Pascal*. Cambridge Computer Science Texts. Cambridge University Press, 1st edition, August 1984

contains numerous examples in Pascal of recursive algorithms on data structures (linked lists, trees, or graphs) implemented through pointers. In contrast, the current book avoids pointers since they are not used explicitly in Python. The suggested reference includes backtracking algorithms for generating additional combinatorial entities such as combinations, compositions, and partitions. Lastly, it contains a chapter on recursion elimination, which offers low-level explanations on how to transform recursive programs into equivalent iterative versions.

Another book that focuses entirely on recursion is:

- Eric S. Roberts. *Thinking Recursively*. Wiley, 1st edition, January 1986, which contains examples in Pascal, and a more recent edition:
 - Eric S. Roberts. *Thinking Recursively with Java*. Wiley, 1st edition, February 2006,

where the code is in Java. The book contains a chapter on recursive data types, and another on the implementation of recursion from a low-level point of view.

Java programmers can also benefit from:

- Irena Pevac. *Practicing Recursion in Java*. CreateSpace Independent Publishing Platform, 1st edition, April 2016,

which contains examples related to linked lists, linked trees, and graphical problems.

DESIGN AND ANALYSIS OF ALGORITHMS

The current book contains problems that can be solved by applying algorithm design techniques such as divide and conquer, or backtracking. There are numerous excellent texts that analyze more advanced problems, describe other design techniques like greedy algorithms or dynamic programming, or rely on advanced data structures in order to construct efficient algorithms. The following list of books is only a small subset of the broad literature on algorithm design (and analysis) techniques:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.
- Anany V. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2012.
- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley Publishing, 1st edition, 2013.

More ambitious readers can explore texts aimed at training for programming competitions (e.g., the ACM International Collegiate Programming Contest, or the International Olympiad in Informatics). Well-known references include:

- Steven S. Skiena and Miguel Revilla. *Programming Challenges: The Programming Contest Training Manual*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
- Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

Other textbooks focus exclusively on the analysis of algorithms, such as:

- Jeffrey J. McConnell. *Analysis of Algorithms: An Active Learning Approach*. Jones and Bartlett Publishers, Inc., USA, 1st edition, 2001

Among these,

- Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Professional, 2nd edition, 2013

is considerably more advanced, and is used by the author in a massive open online course offered by *coursera.org*. Similarly, this and other online learning platforms offer excellent courses on algorithms.

FUNCTIONAL PROGRAMMING

Recursion is omnipresent in functional programming. Thus, programmers should have mastered the contents of this book in order to be competent in this programming paradigm. Popular references include:

- Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.
- Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall Europe, April 1998.
- Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Incorporation, USA, 1st edition, 2008.

The book by Odersky et al. is used in a highly recommended course also offered at *coursera.org*.

PYTHON

It is assumed that the reader of this book has some programming experience. The following popular texts can be useful to readers interested in learning more Python features, looking up implementation details, or developing alternative and/or more efficient recursive variants of the examples covered throughout the book:

- Mark Pilgrim. *Dive Into Python 3*. Apress, Berkely, CA, USA, 2009.
- Mark Summerfield. *Programming in Python 3: A Complete Introduction to the Python Language*. Addison-Wesley Professional, 2nd edition, 2009.
- Mark Lutz. *Learning Python*. O'Reilly, 5th edition, 2013.

RESEARCH IN TEACHING AND LEARNING RECURSION

The current book incorporates several ideas that stem from research in teaching and learning recursion. In particular, it focuses on the declarative abstract level of problem decomposition and induction, instead of focusing on a computational model (e.g., a trace mental model) or iterative/imperative thinking. This is suggested, for example, in:

- D. Ginat and E. Shifroni. Teaching recursion in a procedural environment – how much should we emphasize the computing model? *SIGCSE Bull.*, 31(1):127–131, 1999.
- R. Sooriamurthi. Problems in comprehending recursion and suggested solutions. *SIGCSE Bull.*, 33(3):25–28, 2001.

The diagrams and methodology (the template is similar to the one in Sooriamurthi's paper) used in [Chapter 2](#) have been introduced precisely in order to focus on declarative thinking. In addition, the structure of the book also reflects this goal. Elements such as tracing, recursion trees, the program stack, or the relationship between iteration and tail recursion are not introduced until [Chapters 10](#) and [11](#).

The book also includes examples from articles where I have addressed teaching mutual recursion, and tail (and nested) recursion through function generalization:

- M. Rubio-Sánchez, J. Urquiza-Fuentes, and C. Pareja-Flores. A gentle introduction to mutual recursion. *SIGCSE Bull.*, 40(3):235–239, 2008.
- Manuel Rubio-Sánchez. 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.

Finally, there is rich literature on research in teaching and learning recursion. The following recent surveys contain hundreds of references related to the topic:

- Christian Rinderknecht. A survey on teaching and learning recursive programming. *Informatics in Education*, 13(1):87–119, 2014.
- Renée McCauley, Scott Grissom, Sue Fitzgerald, and Laurie Murphy. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, 25(1):37–66, 2015.

Index

- Accumulator variable, 337, 347, 374
- Ackermann function, 342, 351
- Activation
 - record, 306
 - tree, 300, 327, 334, 351
- Active
 - frame, 306
 - subroutine, 306
- Addition, *see* sum
- Algorithm, 8
 - bisection, 157, 336
 - breath-first search, 313
 - brute-force, 353
 - depth-first search, 213, 313, 316
 - Euclid's, 164, 300
 - Hoare's partitioning, 148, 150, 179
 - Horner's, 124
 - in-place, 179
 - Karatsuba's, 183, 203
 - Newton's method, 169
 - out-of-place, 179
 - searching
 - binary, 142, 160, 168
 - linear, 139
 - quickselect, 148, 155
 - sorting, 172, 173
 - counting sort, 167, 174
 - insertion sort, 132
 - merge sort, 174
 - quicksort, 148, 177
 - selection sort, 121
 - Strassen's, 190, 204
- Alphabet, 273
- Arithmetic series, 62
- Asymptotic notation, 73, 76
 - big-O, 73
 - big-Omega, 75
 - big-Theta, 75
- Backtracking, 353
 - template, 353
- Backward substitution, *see* expansion method
- Base, 115
- Base case, 4, 34, 295
 - misconception, 24, 35, 295
 - missing, 36, 52
 - redundant, 35, 132
- Base conversion, 117, 340, 348
- Basketball, 251
- Binomial coefficient, 58, 131, 226, 242, 245, 331
- Bits, 115, 131, 183, 251
- Branch and bound, 388, 393
- Calculator, 273
- Call stack, 306
- Candidate, 354, 361, 380, 384
- Catalan number, 248, 251
- Ceiling, 66, 80
- Characteristic polynomial, 90
- Combination, 240
- Combinatorics, 235
- Compiler, 273
- Complete solution, 354

- Computational complexity, 70
- Computational problem, 7
- Constraint satisfaction problem, 353
- Control
 - flow, 292
 - stack, 306
- Counting problem, 235
- Data structure, 6
 - FIFO, 305
 - LIFO, 305
 - list, 6, 17
 - queue, 305, 313
 - stack, 305, 313, 316
 - tree, 6
- Debug, 26, 52, 295, 327–330, 337, 362
- Declarative programming, 25, 26
- Dependency graph, 322, 331
- Dequeue, 306
- Descendant, 7, 29
- Difference equation, 89
- Digital root, 343
- Digits, 5, 47, 118, 131, 132, 134, 167, 203
- Discrete state space, 354
- Divide and conquer, 28, 51, 84, 133, 171
- Domino tiles, 251
- Dot product, 68
- Droste effect, 3
- Dynamic programming, 217, 322
- Efficiency, *see* computational complexity
- Enqueue, 306
- Execution time, 70
- Exhaustive search, 353, 355
- Expansion method, 80
 - misconceptions, 83
- Expression, 274, 279, 282
 - parenthesized, 279, 287
- Factor, 279, 286
- Factorial, 4, 35, 238, 337
- Fibonacci
 - function, 4, 11, 17, 27–29, 244, 260, 265, 268, 288, 300, 318, 351
 - number, 4, 250, 258, 288, 323
 - sequence, 4
- File system, 313
- Floor, 66, 80
- Formal
 - grammar, 273, 279
 - language, 273
- Fractal, 220, 224, 230, 233
 - Koch, 330
- Function generalization, 344
- Functional programming, 25
- General method for difference equations, 89
- Generality, 36
- Geometric series, 63
- Grammar, *see* formal grammar
- Greatest common divisor, 164
- Guido van Rossum, 337
- Heap, 312
- Hilbert curve, 233
- Imperative programming, 25, 341
- Induction, 20, 41
 - hypothesis, 21
 - proof, 20
- Inorder, 216
- Iteration, 25, 333, 337
 - method, *see* expansion method

- Knapsack, 388
- Knight's tour, 399
- Koch
 - curve, 220
 - snowflake, 220
- Limit, 59
- List, 132, 167, 168, 203, 350
 - head, 30
 - tail, 30
- Logarithm, 58, 76
- Magic square, 399
- Majority element, 180
- Manhattan paths, 244
- Master theorem, 84, 176, 304
- Matplotlib, 193, 223, 382
- Matrix, 67, 129
 - block, 40
 - product, 69, 186, 190, 204
 - rotation, 69, 223
 - symmetric, 69
 - transpose, 203
- Matryoshka dolls, 3
- Maze, 377
- McCarthy 91 function, 342
- Memoization, 219, 317, 331, 332
- Mental model, 41, 291
- Methodology, *see* recursive design
 - template
- Moor traversal, 205
- Multiplayer game, 255, 288
- Multiplication, *see* product
 - principle, 239
- N-queens puzzle, 354, 368
- N-rooks puzzle, 398
- NumPy, 129, 208, 223
- Optimal solution, 388, 392
- Optimization problem, 217, 353, 388, 393, 400, 401
- Order of growth, 71, 76
- Overlapping subproblems, 219, 317
- Palindrome, 120, 217, 230, 320, 332
- Parity, 52, 129, 254
- Partial solution, 354
- Partially filled array, 18
- Partitioning scheme, 148, 155
- Pascal's triangle, 125, 131
- Passive flow, 295
- Permutation, 236, 364, 369
 - with repetition, 245
- Pitfall, 36, 37, 52
- Polygon, 245
- Polynomial, 167, 203
- Pop, 305
- Postorder, 216, 299
- Power, 58, 106, 129, 240, 350
- Precondition, 16
- Preorder, 216, 299, 357
- Problem, 37
 - decomposition, 5, 7, 37, 171
 - instance, 5, 8
 - size, 5, 32, 110, 171
- Procedure, 47
- Product, 65, 130, 183, 203
- Production rule, 273, 279
- Program stack, 26, 306, 308–310
- Programming paradigm, 25
- Push, 305
- Pyramid, 43, 63, 248, 252
- Rabbit population, 256, 258
- Radix, *see* base
- Recurrence relation, 57, 76
 - homogeneous, 90

- nonhomogeneous, 94
- Recursion, *see* recursion
 - depth, 36, 312
 - infinite, 14, 36, 52, 53, 312
 - tree, 297, 326, 356, 359, 364
 - height, 303, 310
 - pruned, 358, 374, 393
- Recursive
 - call, 14
 - case, 4, 37, 41
 - descent parser, 274, 282
 - design
 - diagram, 37, 41
 - template, 31
 - leap of faith, 22
 - type, 27
 - binary, 28
 - final, 133
 - indirect, 28, 253
 - linear, 27, 105, 133, 300
 - multiple, 28, 171, 205
 - mutual, 28, 253
 - nested, 29, 342
 - tail, 27, 300, 333, 337
- Resistor, 127
- Return address, 306
- Romanesco broccoli, 1
- Root finding, 157, 169
- Ruler, 227
- Scalability, 336
- Short-circuit evaluation, 135, 137
- Sierpiński
 - carpet, 224
 - triangle, 3, 230
- Skyline, 196
- Spin-out[®] brainteaser, 289
- Stack
 - frame, 306, 308, 310, 335
 - overflow, 26, 312, 336
 - trace, 309
- Staircase climbing, 242
- String, 119, 120, 136, 276
- Sublist, 13, 230
- Subproblem, 5, 8, 37
 - graph, 322
- Subsequence, 13, 217, 230, 332
- Subset, 359, 372, 388, 397
- Substring, 217, 320
- Sudoku, 384, 398
- Sum, 4, 11, 12, 14, 19, 42, 60, 113, 130, 350
 - partial, 62
- Swamp traversal, 205, 226
- Term, 279, 281, 284, 285
- Test, 52
- Token, 273, 274, 279
- Towers of Hanoi, 209
 - cyclic, 268, 288
 - variant, 229, 268, 288
- Traceback module, 309
- Transpose, 68, 203
- Traveling salesman problem, 400
- Tree, 313
 - binary, 251, 304, 359, 389
 - binary search, 143, 215, 326
 - method, 304
 - traversal, 213
- Triangulation, 245
- Trigonometry, 66
- Tromino tiling, 192
- Tug of war, 400
- Variation with repetition, 239, 250
- Vector, 68, 221
- Vowel, 131
- Water treatment plant, 263
- Woodcutter problem, 158, 350
- Worst case, 75