

# Introduction to Recursive Programming

Manuel Rubio-Sánchez



CRC Press  
Taylor & Francis Group

A CHAPMAN & HALL BOOK

CPD  
CERTIFIED

The CPD Certification  
Service

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) (<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 DIAGRAMMS	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
<b>CHAPTER 3 ■ Runtime Analysis of Recursive Algorithms</b>		<b>57</b>
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
<b>CHAPTER 4 ■ Linear Recursion I: Basic Algorithms</b>		<b>105</b>
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 over- flow 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
<b>CHAPTER 11 ■ Tail Recursion Revisited and Nested Recursion</b>		<b>333</b>
<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 FUNC- TION GENERALIZATION	344
11.4.1	Factorial	345
11.4.2	Decimal to base $b$ conversion	348
11.5	EXERCISES	350
<b>CHAPTER 12 ■ Multiple Recursion III: Backtracking</b>		<b>353</b>
<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](http://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](mailto: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 <b>a</b> , 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 $2 \times 10$ rectangle with $1 \times 2$ or $2 \times 1$ 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 <sup>®</sup> 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>&lt;·,·&gt;</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 <b>L</b> 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 $3 \times 3$ 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 <b>a</b> , 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 <b>a</b> . 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 <i>s</i> .	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 <i>n</i> , which is greater than or equal to <i>m</i> .	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



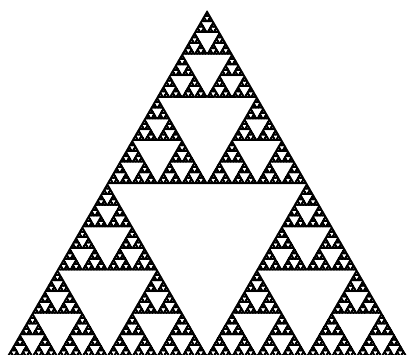
Branching rivers



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:

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

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,