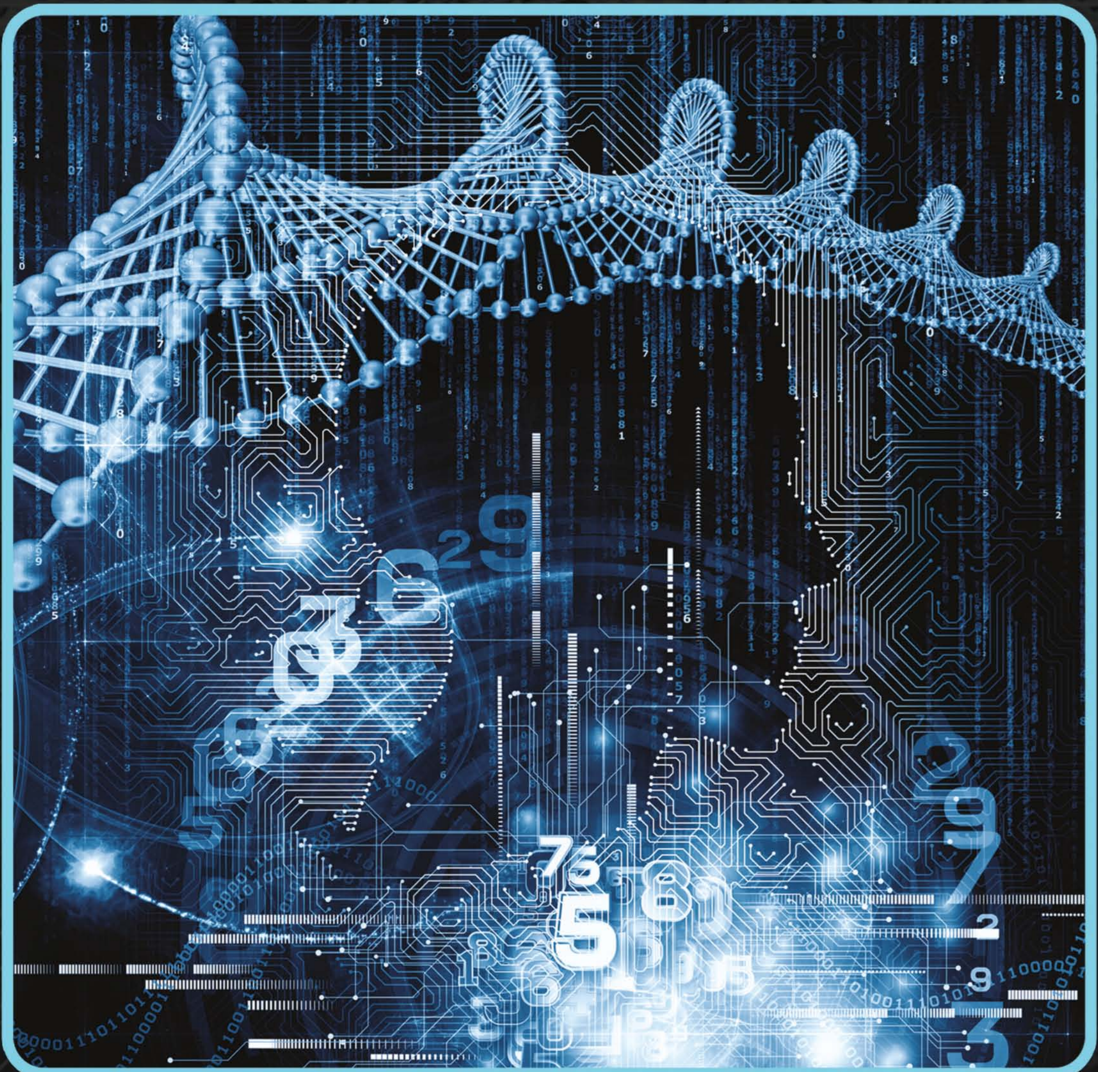
Introduction to Recursive Programming

Manuel Rubio-Sánchez



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

CPD CERTIFIED
The CPD Certification Service

Introduction to

Recursive Programming

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

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)

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

Contents

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 primarily 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 computational 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

List of Tables

List of Listings

CHAPTER 1

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

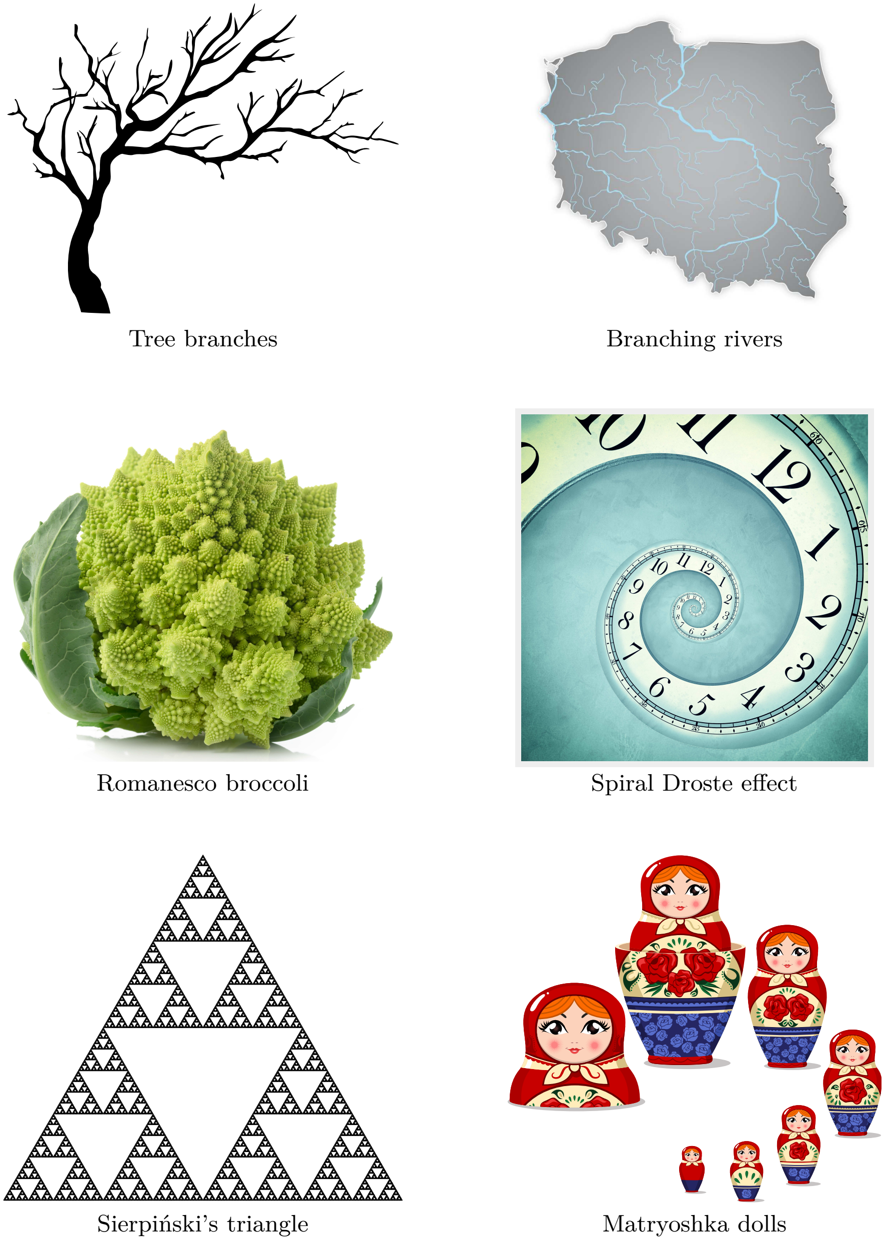


Figure 1.1 Examples of recursive entities.

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:

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

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:

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

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:

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

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:

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

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

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

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 $S(n-1) = 1 + 2 + \cdots + (n-1)$, which leads to the following recursive definition:

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

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:

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

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,