CHAPTER 1

Basic Concepts of Recursive Programming

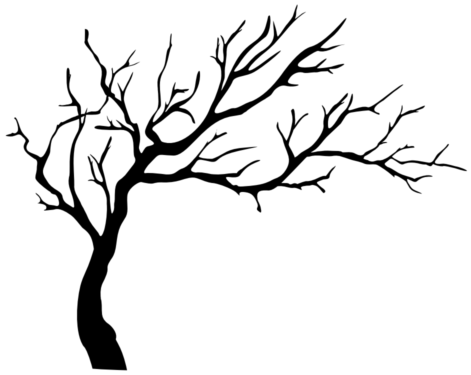
To iterate is human, to recurse divine.

— 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

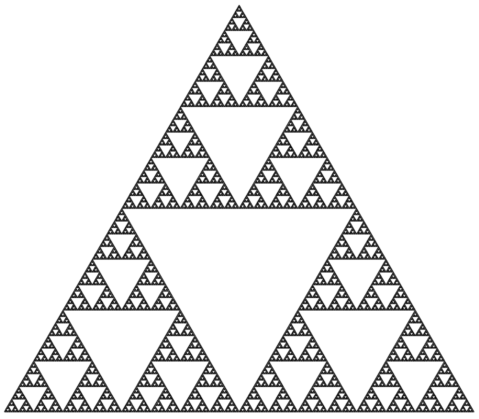
An entity or concept is said to be recursive when simpler or smaller self-similar instances form part of its constituents. Nature provides numerous examples where we can observe this property (see Figure 1.1). For instance, a branch of a tree can be understood as a stem, plus a set of smaller branches 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


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:

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

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,

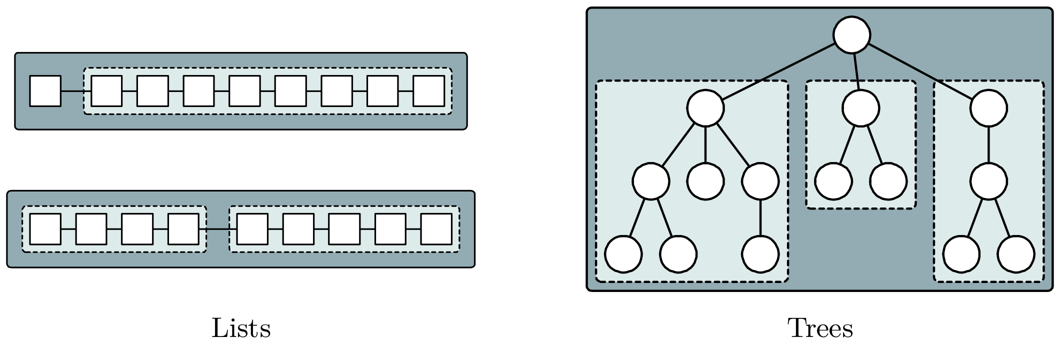


Figure 1.2 Recursive decomposition of lists and trees.

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

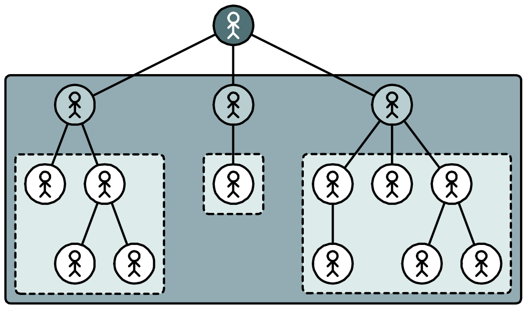


Figure 1.3 Family tree representing the descendants of a person, which are its children plus the descendants of its children.

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,

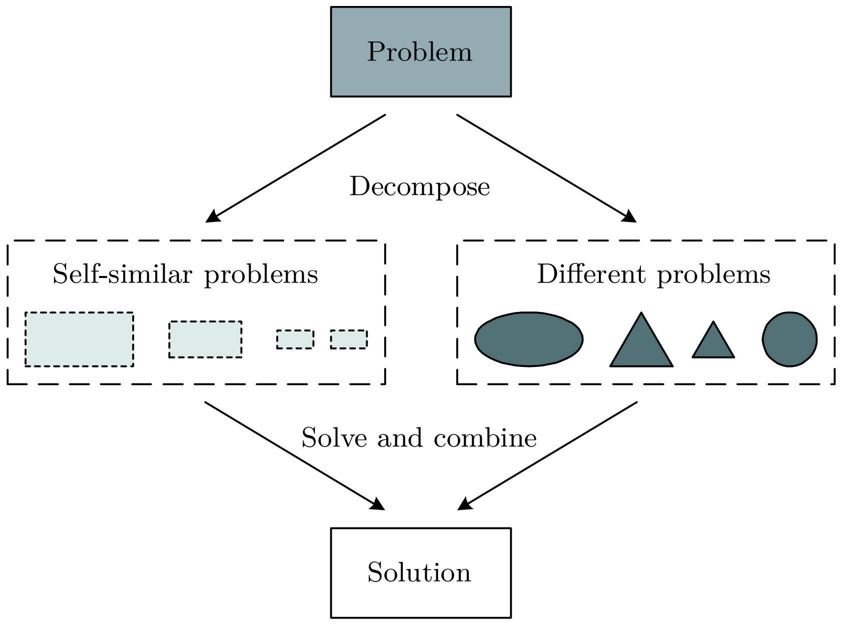


Figure 1.4 Recursive problem solving.

“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 $1+2+\cdots+(n-1)+n$. 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-

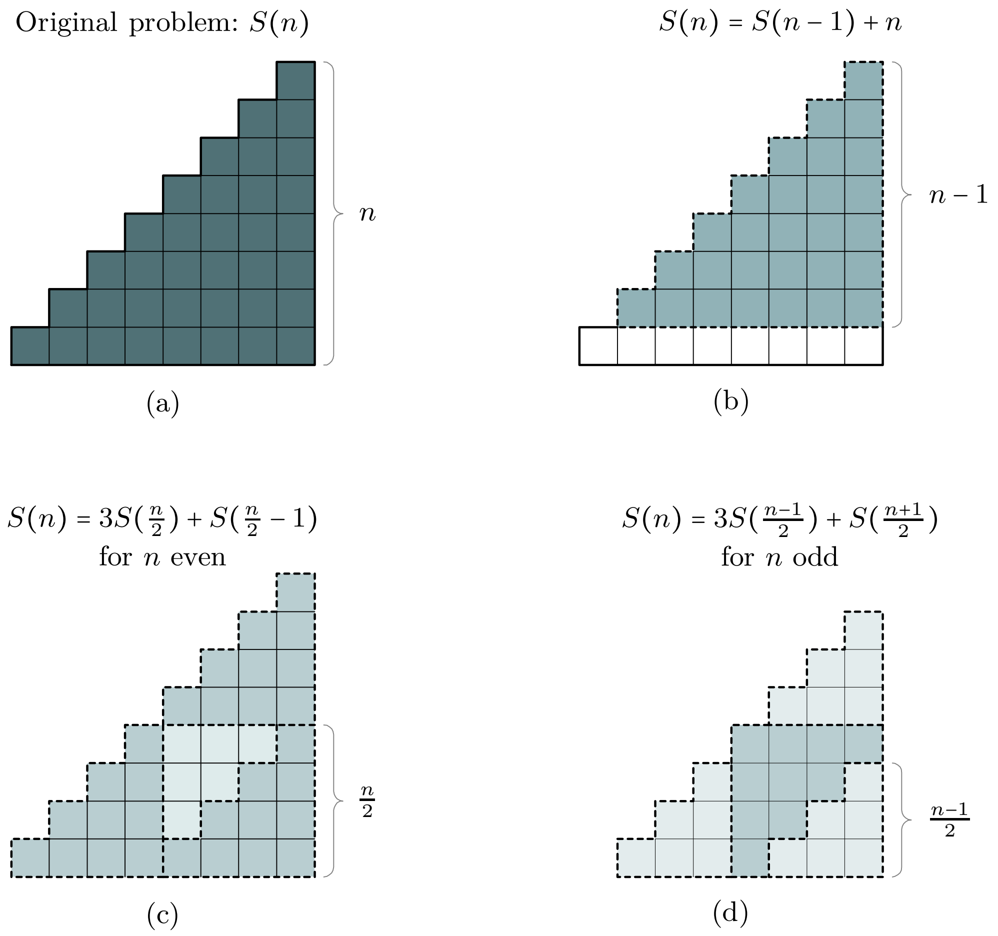


Figure 1.5 Decompositions of the sum of the first positive integers.

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 $2+\cdots+(n-1)+n$. 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: $m+(m+1)+\cdots+(n-1)+n$, 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:

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

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)):

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

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

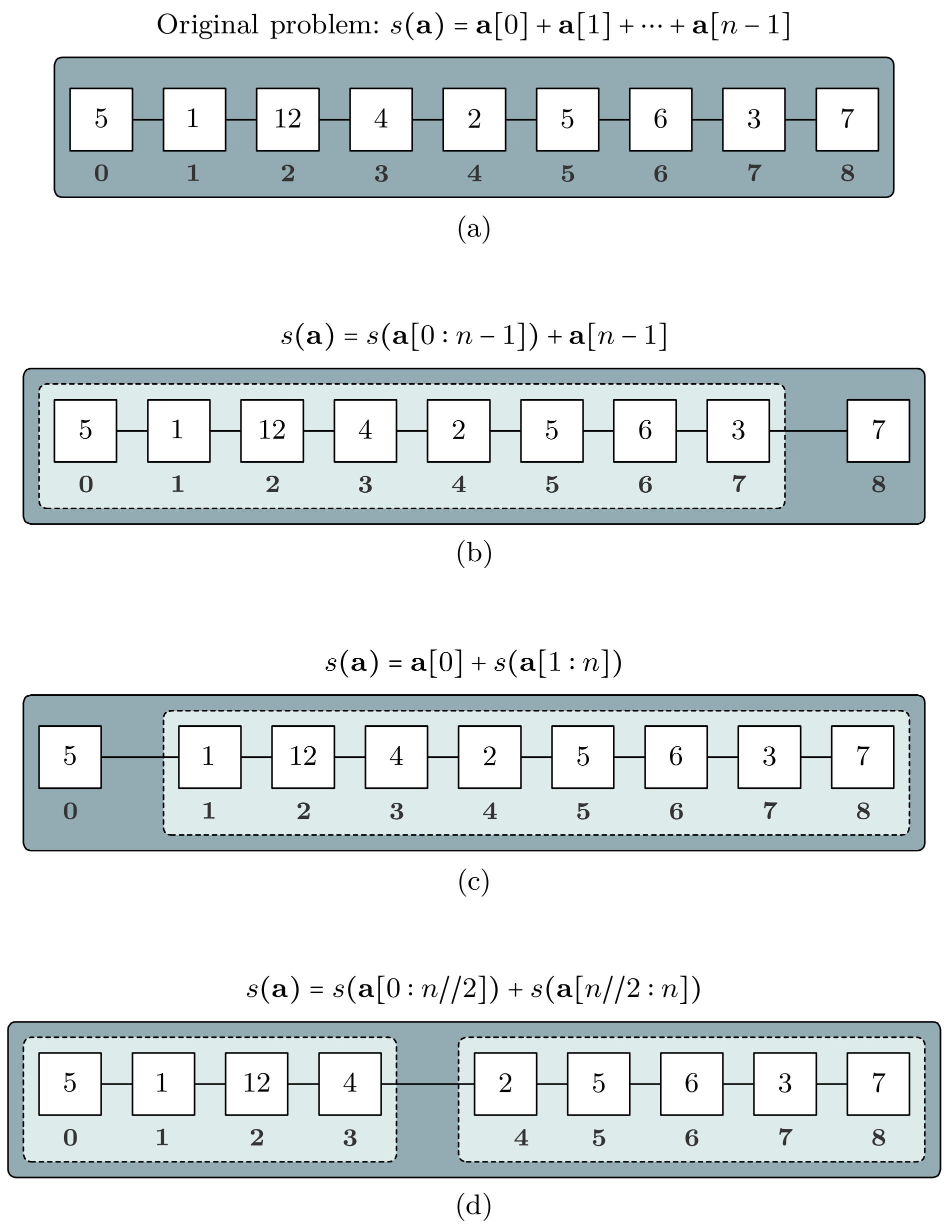

Original problem: $s(\mathbf{a}) = \mathbf{a}[0] + \mathbf{a}[1] + \cdots + \mathbf{a}[n-1]$
5 1 12 4 2 5 6 3 7
0 1 2 3 4 5 6 7 8
(a)
$s(\mathbf{a}) = s(\mathbf{a}[0:n-1]) + \mathbf{a}[n-1]$
5 1 12 4 2 5 6 3 7
0 1 2 3 4 5 6 7 8
(b)
$s(\mathbf{a}) = \mathbf{a}[0] + s(\mathbf{a}[1:n])$
5 1 12 4 2 5 6 3 7
0 1 2 3 4 5 6 7 8
(c)
$s(\mathbf{a}) = s(\mathbf{a}[0:n//2]) + s(\mathbf{a}[n//2:n])$
5 1 12 4 2 5 6 3 7
0 1 2 3 4 5 6 7 8
(d)

Figure 1.6 Decompositions of the sum of the elements in a list, denoted as **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 **a**, containing n numbers. Formally, the problem can be expressed as:

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

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 **a** is a collection of *contiguous* elements of **a**, unless explicitly stated otherwise. In contrast, in a **subsequence** of some initial sequence **s** its elements appear in the same order as in **s**, but they are not required to be contiguous in **s**. In other words, a subsequence can be obtained from an original sequence **s** by deleting some elements of **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:

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

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:

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

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:

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

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 $s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) = 0 + s(\mathbf{a}) \ldots$, 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:

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

Pascal:

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

MATLAB®:

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

Scala:

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

Haskell:

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

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.

```python
def sum_first_naturals(n):
    if n == 1:
        return 1  # Base case
    else:
        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 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.

```python
 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.

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

ing 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.

```python
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 $F(1)$, $F(2)$,..., $F(n-2)$. 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

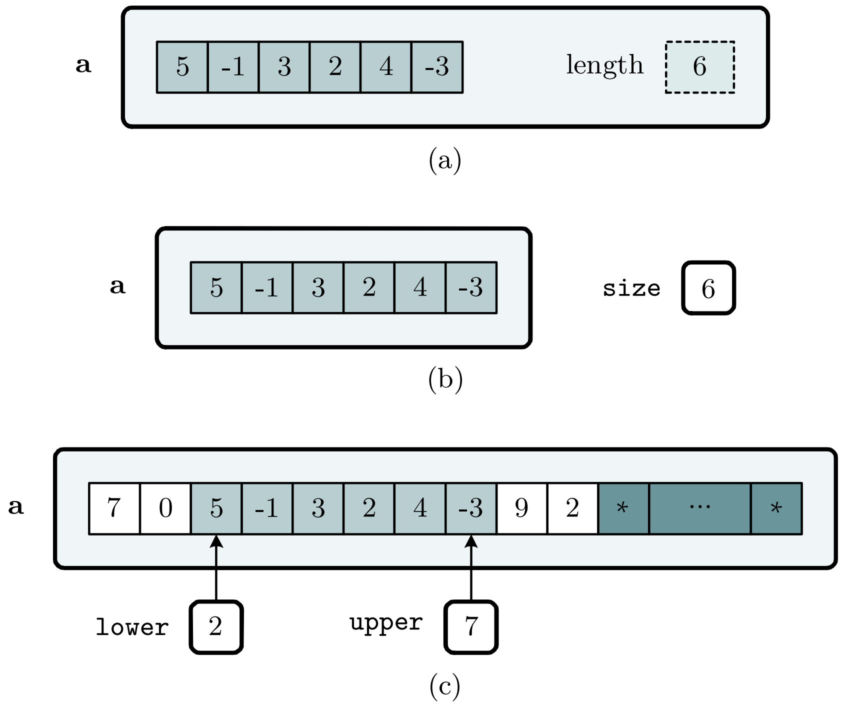


Figure 1.8 Data structures similar to lists, and parameters necessary for defining sublists.

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.

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


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


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


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

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

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.

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


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


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


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

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

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:

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

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

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

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:

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

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

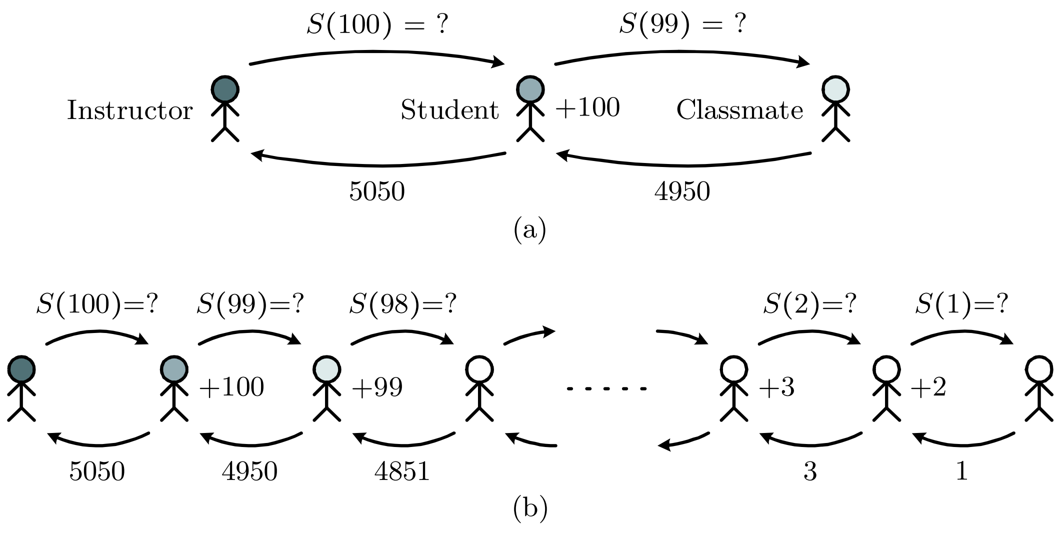


Figure 1.9 Thought experiment in a classroom, where an instructor asks a student to add the first 100 positive integers. $S(n)$ represents the sum of the first n positive integers.

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 assume 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 recursive 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:

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

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:

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

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 ($F(n) = f(n)$):

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

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:

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

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

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:

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

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 Figure1.8(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 $n = 1, \ldots, 10$.