云南大学数学与统计学院

上机实践报告

|  |  |  |
| --- | --- | --- |
| **课程名称**：数据结构与算法实验 | **年级**：2015级 | **上机实践成绩**： |
| **指导教师**：陆正福 | **姓名**：刘鹏 |  |
| **上机实践名称**：递归实验 | **学号**：20151910042 | **上机实践日期**：2017-05-14 |
| **上机实践编号**：No.04 | **组号**： | **上机实践时间**：上午3、4节 |

# 一、实验目的

1. 熟悉递归算法设计模式

2. 熟悉Python递归程序设计

3. 熟悉主讲教材Chapter 4的代码片段

# 二、实验内容

1. 递归有关的数据结构设计与算法设计；

2. 调试主讲教材Chapter 4的Python 程序；

3. （选做） 任选一个算法,写出其递归版本与非递归版本,总结不同设计与实现的优缺点。

# 三、实验平台

Windows 10 1703 Enterprise 中文版；

Python 3.6.0；

Wing IDE Professional 6.0.5-1集成开发环境。

# 四、实验记录与实验结果分析

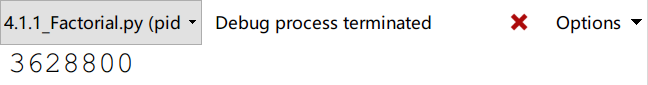
1题

4.1.1 The Factorial Function（递归函数）

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # 4.1.1 The Factorial Function  **def** factorial**(**n**):**  **if** n **==** 0**:**  **return** 1  **else:**  **return** n **\*** factorial**(**n **-** 1**)**  #----------------------------- my main function -----------------------------  **print(**factorial**(**10**))** |

程序代码 1



运行结果 1

2题

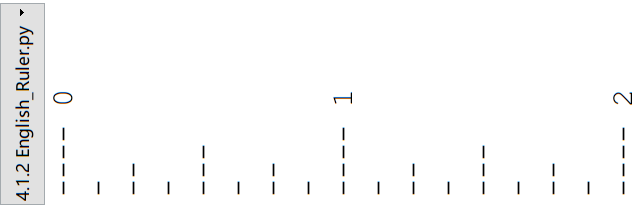
Drawing an English Ruler

＊画一个英式直尺

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | # 4.1.2 Drawing an English Ruler  **def** draw\_line**(**tick\_length**,**tick\_label**=**''**):**  """Draw one line with given tick length (followed by optional label)."""  line **=** '-' **\*** tick\_length  **if** tick\_label**:**  line **+=** ' ' **+** tick\_label  **print(**line**)**  **def** draw\_interval**(**center\_length**):**  """Draw tick interval based upon a central tick length."""  **if** center\_length **>** 0**:** # stop when length drops to 0  draw\_interval**(**center\_length **-** 1**)** # recursively draw top ticks  draw\_line**(**center\_length**)** # draw center tick  draw\_interval**(**center\_length **-** 1**)** # recursively draw bottom ticks  **def** draw\_ruler**(**num\_inchs**,**major\_length**):**  """Fraw English ruler with given number of inches, major tick length."""  draw\_line**(**major\_length**,**'0'**)** # draw inch 0 line  **for** j **in** range**(**1**,**1 **+** num\_inchs**):**  draw\_interval**(**major\_length **-** 1**)** # draw interior ticks for inch  draw\_line**(**major\_length**,**str**(**j**))** # draw inch j line and label  #------------------------------ my main function ------------------------------  draw\_ruler**(**2**,**4**)** |

程序代码 2



运行结果 2

算法分析：

可以很明显看出，递归就是不断地铺张内存，而循环就不会。一次递归就展开一段内存，当到了最后不满足判断条件，就开始收敛内存区域。正因如此，流程图也很好画。这个递归具有对称性和深度优先性，因为递归的过程中包含两个完全一样的子递归，子递归之间有一个画线函数，所以这个函数必然是对称的。

由于数据量不大，所以不需要分析是否是原址递归。其实这个肯定不是原址递归。



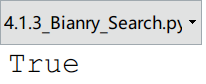
3题

二元搜索算法。

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | # 4.1.3 Binary Search  **def** binary\_search**(**data**,**target**,**low**,**high**):**  """Return True if target is found in indicated portion of a Python list.  The search only considers the protion from data[low] to data[high] inclusive  """  **if** low **>** high**:** # interval is empty; no match  **return** **False**  **else:**  mid **=** **(**low **+** high**)** **//** 2  **if** target **==** data**[**mid**]:** # found a match  **return** **True**  **elif** target **<** data**[**mid**]:**  # recur on the portion left of the middle  **return** binary\_search**(**data**,**target**,**low**,**mid**-**1**)**  **else:**  # recur on the portion right of the middle  **return** binary\_search**(**data**,**target**,**mid **+** 1**,** high**)**  #------------------------------ my main function ------------------------------  a **=** **[**1**,**2**,**3**,**5**,**8**,**15**,**45**,**666**,**3333**,**6222**,**9111**]**  **print(**binary\_search**(**a**,**9111**,**0**,**11**))** |

程序代码 3



运行结果 3

算法分析：

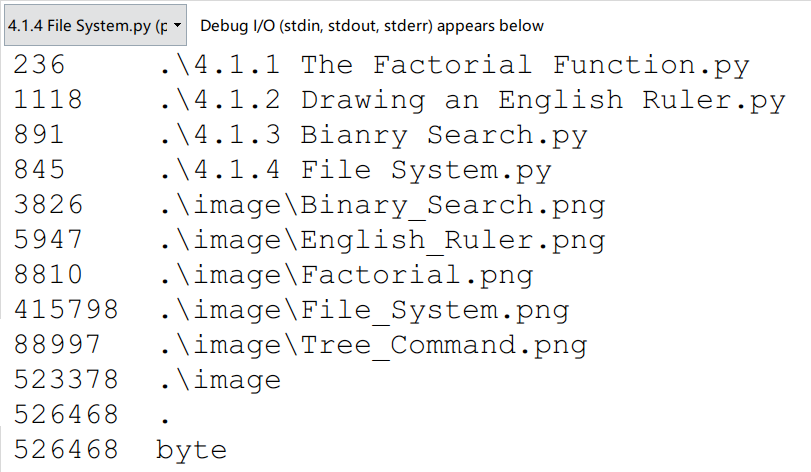
可以看到，二元搜索算法仅仅针对有序的序列。

4题

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # 4.1.4 File Systems  **import** os  **def** disk\_usage**(**path**):**  """Return the number of bytes used by a file/folder and any descendents"""  total **=** os**.**path**.**getsize**(**path**)** # account for direct usage  **if** os**.**path**.**isdir**(**path**):** # if this is a directory  **for** filename **in** os**.**listdir**(**path**):** # then for each child  childpath **=** os**.**path**.**join**(**path**,**filename**)** # compose full path to child  total **+=** disk\_usage**(**childpath**)** # add child's usage to total  **print(**'{0:<7}'**.**format**(**total**),**path**)** # descriptive output (optional)  **return** total # return the grand total  #------------------------------ my main function ------------------------------  path **=** '.' # current dir  **print(**disk\_usage**(**path**),**' byte'**)** |

程序代码 4



运行结果 4

算法分析：

课本的分析很清楚了。计算，然后打印，这就构成了递归函数。也是先深入到最底层，然后逐渐退出，cd ..，到了最外层的目录。

# 五、教材翻译

**Translation**

**Chapter 4 Recursion**

＊第四章 递归

One way to describe repetition within a computer program is the use of loops, such as Python’s while-loop and for-loop constructs described in Section 1.4.2. An entirely different way to achieve repetition is through a process known as ***recursion***.

＊循环是一种在计算机程序中描述重复行为的方法，正如在1.4.2节描述的，Python中就有while循环与for循环。另一种完全不同的但却同样可以实现重复性操作的方式叫做递归。

Recursion is a technique by which a function makes one or more calls to itself during execution, or by which a data structure relies upon smaller instances of the very same type of structure in its representation. There are many examples of recursion in art and nature. For example, fractal patterns are naturally recursive. A physical example of recursive used in art is in the Russian Matryoshka dolls. Each doll is either make of solid wood, or is hollow and contains another Matryoshka doll inside it.

＊通过递归这种方法，一个函数可以进行一次或者多次的自我调用，一个数据结构也可以依靠与其同类型的、但是规模较小的实例来实现自我构建。在艺术与自然的领域里，有很多递归的例子。举个例子，分形艺术就是自然递归。另一个在艺术中的递归的实例是俄罗斯套娃。套娃的一个，要么是实心的，要么是空心的，而且空心的这种里面还有一个套娃。

In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks. In fact, a few programming languages (e.g., Scheme, Smalltalk) do not explicitly support looping constructs and instead rely directly on recursion to express repetition. Most modern programming languages support functional recursion using the identical mechanism that is used to support traditional forms of function calls. When one invocation of the function make a recursive call, that invocation is suspended until the recursive call completes.

＊在计算中，递归为执行重复性任务提供了一种优雅而强有力的选择。事实上，有一小部分编程语言并不直接支持循环结构，比如说Scheme或Smalltalk，反而是直接靠递归来表达重复性操作。大多数的现代编程语言都采用了与传统的函数调用上的相同的机制，实现了函数性递归。当函数的一个调用进行递归式调用时，这个调用就被暂时停下，知道递归调用完成。

Recursion is an important technique in the study of data structures and algorithms. We will use it prominently in several later chapters of this book (most notably, Chapter 8 and 12). In this chapter, we begin with the following four illustrative examples of the use of recursion, providing a Python implementation for each.

＊对于数据结构与算法的学习而言，递归这种方法十分重要。我们将会在接下来的几个章节中（特别是第八章与第十二章）突出使用它。着这一章中，我们从这四个说明性的例子开始学习递归的使用，而且我们给出了它们在Python下的实现。

* The ***factorial function*** (commonly denoted as ) is a classic mathematical function that has a natural recursive definition.  
  ＊阶乘函数是一个拥有自然递归定义的经典数学函数。
* An ***English ruler*** has a recursive pattern that is a simple example of a fractal structure.  
  ＊英式直尺的刻度线有着递归模式，这种模式是分形结构的一个很简单的例子。
* ***Binary search*** is among the most important computer algorithms. It allows us to efficiently locate a desired value in a data set with upwards of billions of entries.  
  ＊二元搜索是计算机算法领域里最重要的算法之一。这个算法使得我们可以在无数数据中高效定位目标值。
* The ***file*** ***system*** for a computer has a recursive structure in which directories can be nested arbitrarily deeply within other directories. Recursive algorithms are widely used to explore and manage these system.  
  ＊计算机中的文件系统也具有递归结构，可以在其他目录中任意嵌套目录。递归算法被广泛用于探索以及管理这些系统。

We then describe how to perform a formal analysis of the running time of a recursive algorithm and we discuss some potential pitfalls when defining recursions. In the balance of the chapter, we provide many more examples of recursive algorithm, organized to highlight some common forms of design.

＊在这之后，我们将叙述一下如何对一个递归算法的时间复杂度进行正式的分析，而且我们也将会讨论许多在定义递归的时候所面临的陷阱。在本章的剩余部分，我们将会提供递归算法的更多实例，借此突出许多常见的设计形式。

4.1 Illustrative Examples

＊4.1节 实例

4.1.1 The factorial Function

＊阶乘函数

To demonstrate the mechanics of recursion, we begin with a simple mathematical example of computing the value of the ***factorial function***. The factorial of a positive integer n, denoted n!, is deﬁned as the product of the integers from 1 to n. If n = 0, then n! is deﬁned as 1 by convention. More formally, for any integer n ≥ 0,

1 if n = 0

n · (n − 1) · (n − 2)··· 3 · 2 · 1 if n ≥ 1.

For example, 5! = 5 · 4 · 3 · 2 · 1 = 120. The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items. For example, the

three characters a, b, and c can be arranged in 3! = 3 · 2 · 1 = 6 ways: abc, acb,

bac, bca, cab, and cba.

There is a natural recursive deﬁnition for the factorial function. To see this, observe that 5! = 5 · (4 · 3 · 2 · 1) = 5 · 4!. More generally, for a positive integer n, we can deﬁne n! to be n · (n − 1)!. This recursive deﬁnition can be formalized as

n! = 1 if n = 0

n · (n − 1)! if n ≥ 1.

This deﬁnition is typical of many recursive deﬁnitions. First, it contains one or more base cases, which are deﬁned nonrecursively in terms of ﬁxed quantities. In this case, n = 0 is the base case. It also contains one or more recursive cases, which are deﬁned by appealing to the deﬁnition of the function being deﬁned.

A Recursive Implementation of the Factorial Function

Recursion is not just a mathematical notation; we can use recursion to design a Python implementation of a factorial function, as shown in Code Fragment 4.1.

1 def factorial(n):

2 if n == 0:

3 return 1

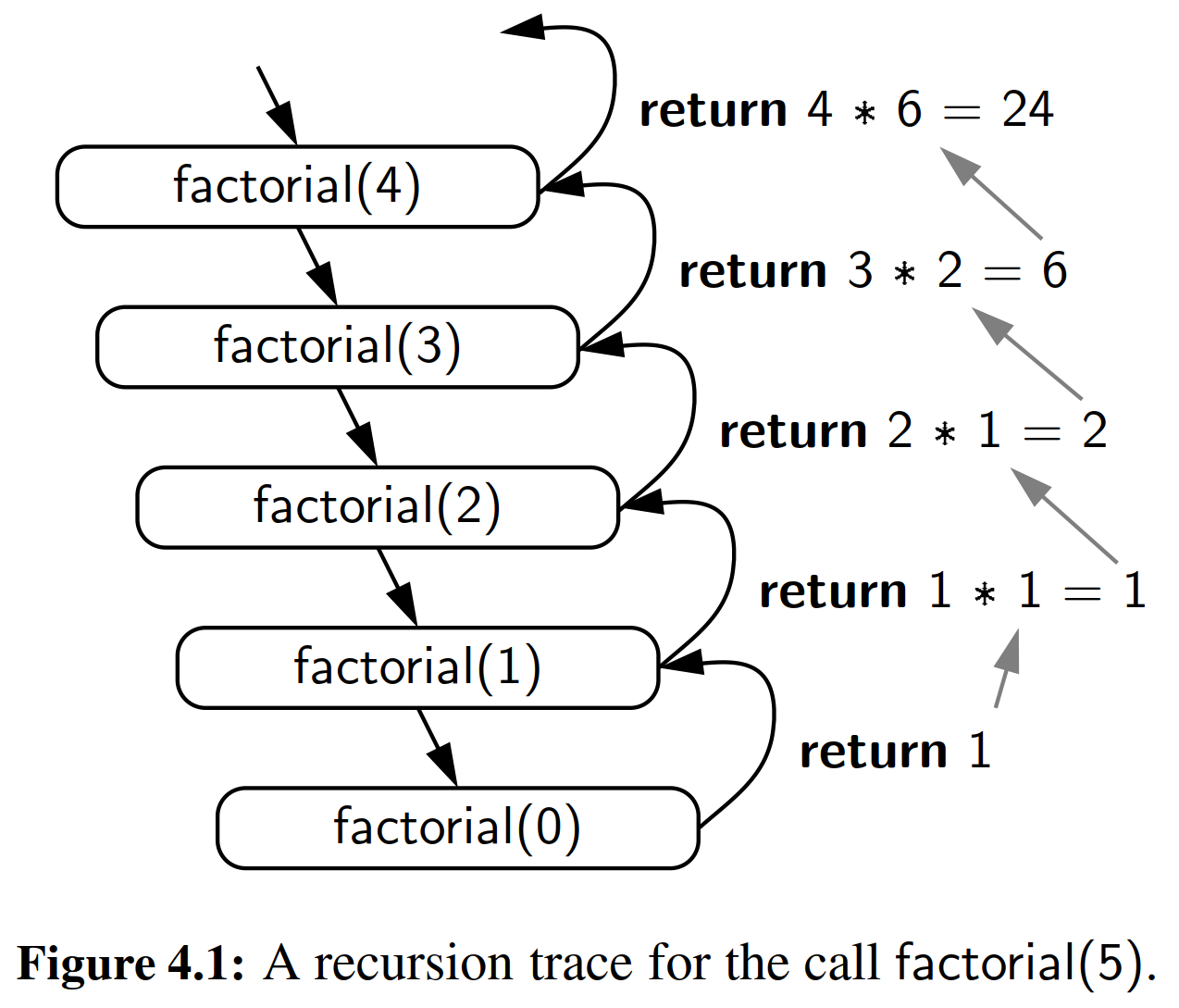
4 else:

5 return n factorial(n−1)

Code Fragment 4.1: A recursive implementation of the factorial function.

This function does not use any explicit loops. Repetition is provided by the repeated recursive invocations of the function. There is no circularity in this deﬁni- tion, because each time the function is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made.

We illustrate the execution of a recursive function using a recursion trace. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by a downward arrow to a new invocation. When the function returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow. An example of such a trace for the factorial function is shown in Figure 4.1.



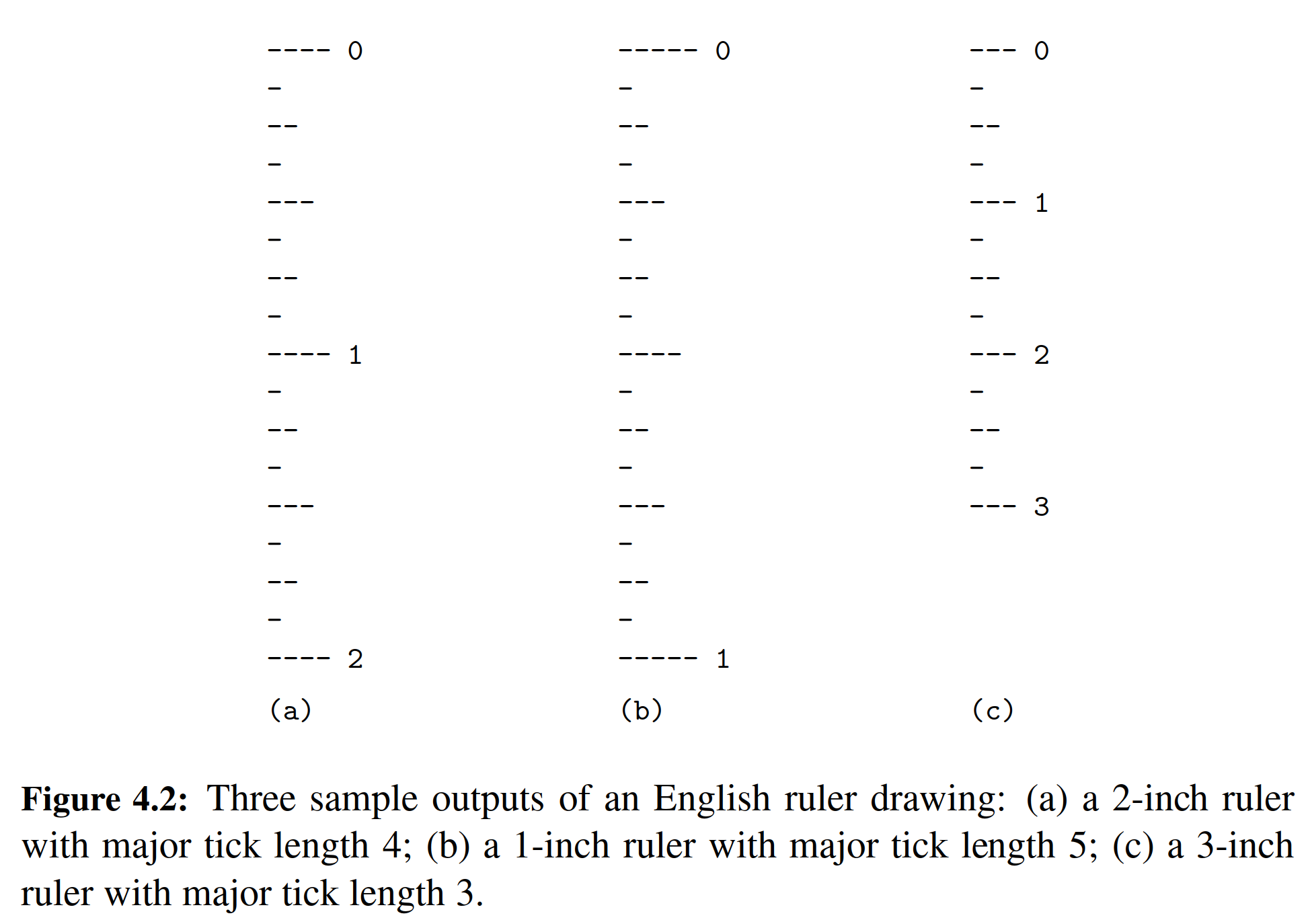
A recursion trace closely mirrors the programming language’s execution of the recursion. In Python, each time a function (recursive or otherwise) is called, a struc- ture known as an activation record or frame is created to store information about the progress of that invocation of the function. This activation record includes a namespace for storing the function call’s parameters and local variables (see Sec- tion 1.10 for a discussion of namespaces), and information about which command in the body of the function is currently executing.

When the execution of a function leads to a nested function call, the execu- tion of the former call is suspended and its activation record stores the place in the source code at which the ﬂow of control should continue upon return of the nested call. This process is used both in the standard case of one function calling a dif- ferent function, or in the recursive case in which a function invokes itself. The key point is that there is a different activation record for each active call.

4.1.2 Drawing an English Ruler

＊4.1.2节 生成刻度尺

In the case of computing a factorial, there is no compelling reason for preferring recursion over a direct iteration with a loop. As a more complex example of the use of recursion, consider how to draw the markings of a typical English ruler. For each inch, we place a tick with a numeric label. We denote the length of the tick designating a whole inch as the ***major tick length***. Between the marks for whole inches, the ruler contains a series of ***minor ticks***, placed at intervals of 1/2 inch, 1/4 inch, and so on. As the size of the interval decreases by half, the tick length decreases by one. Figure 4.2 demonstrates several such rulers with varying major tick lengths (although not drawn to scale).



＊在计算阶乘函数的例子中，我们选择递归而不选择循环是没有强制性理由的。现在给出一个使用递归的更加复杂的例子，那就是绘制典型的英文标尺的刻度标记。每增加一英尺，我们都要放置一个带有数字编号的刻度标签。我们将长度为一英尺的刻度设置为主刻度。在一个单位英尺的刻度线之间，还包含一些列小的刻度线，如二分之一英尺、四分之一英尺等等。刻度线的间隔大小减少一半，刻度线的高度的减少1。

**A Recursive Approach to Ruler Drawing**

The English ruler pattern is a simple example of a ***fractal***, that is, a shape that has a self-recursive structure at various levels of magniﬁcation. Consider the rule with major tick length 5 shown in Figure 4.2(b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3.

＊英式直尺上的刻度排布是一个分形的简单例子，而所谓的分形，指的就是一个在不同的放大倍率有着自我递归结构的图形。

In general, an interval with a central tick length L ≥ 1 is composed of:

* An interval with a central tick length L − 1
* A single tick of length L
* An interval with a central tick length L − 1

Although it is possible to draw such a ruler using an iterative process (see Exercise P-4.25), the task is considerably easier to accomplish with recursion. Our implementation consists of three functions, as shown in Code Fragment 4.2. The main function, draw\_ruler, manages the construction of the entire ruler. Its arguments specify the total number of inches in the ruler and the major tick length. The utility function, draw line, draws a single tick with a speciﬁed number of dashes (and an optional string label, that is printed after the tick).

＊

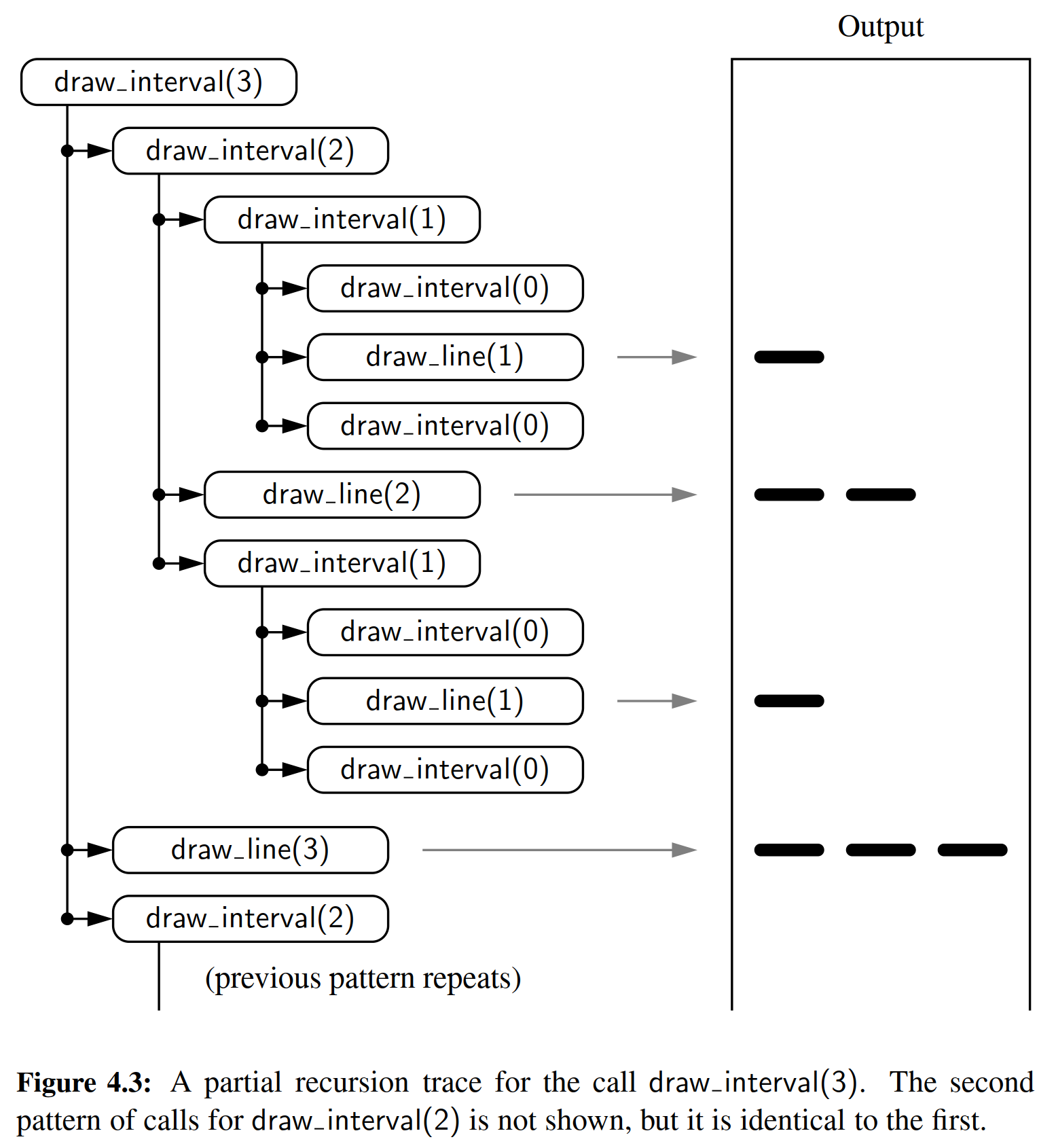
The interesting work is done by the recursive draw\_interval function. This function draws the sequence of minor ticks within some interval, based upon the length of the interval’s central tick. We rely on the intuition shown at the top of this page, and with a base case when L = 0 that draws nothing. For L ≥ 1, the ﬁrst and last steps are performed by recursively calling draw interval(L − 1). The middle step is performed by calling the function draw\_line(L).

＊

**Illustrating Ruler Drawing Using a Recursion Trace**

The execution of the recursive draw\_interval function can be visualized using a recursion trace. The trace for draw\_interval is more complicated than in the factorial example, however, because each instance makes two recursive calls. To illustrate this, we will show the recursion trace in a form that is reminiscent of an outline for a document. See Figure 4.3.

＊递归式函数draw\_interval的执行可以通过一个递归追踪而变得可视化。但是在这个函数里的追踪比阶乘函数里的要复杂，因为每递归一次都产生两个递归调用。为了表述清楚这件事，我们使用一种类似于文档大纲的形式来展示递归追踪。

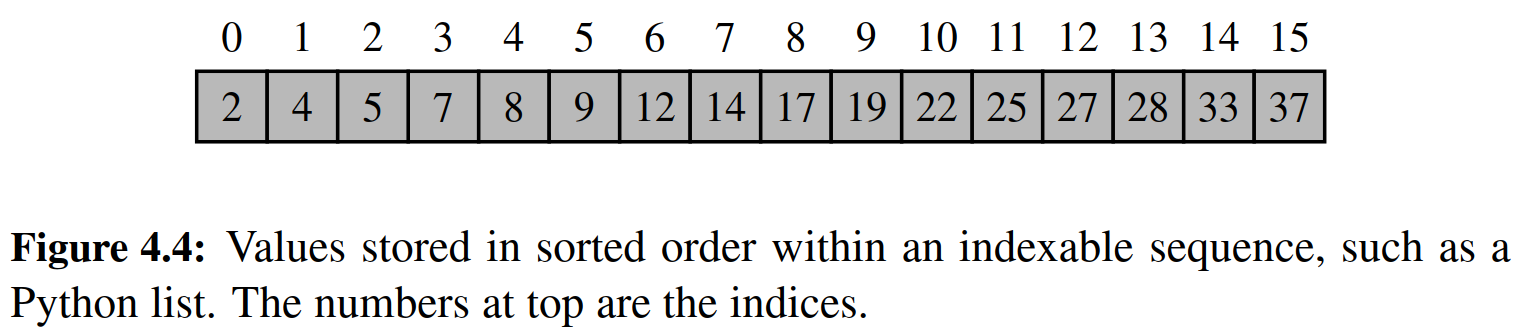


4.1.3 Binary Search

＊4.1.3节 二元搜索

In this section, we describe a classic recursive algorithm, binary search, that is used to efficiently locate a target value within a sorted sequence of elements. This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order (as in Figure 4.4).

＊在这一节中，我们将会描述一个经典的递归算法，那就是用于在有序序列中高效定位目标元素的二元搜索算法。该算法是计算机算法中最重要的一个之一，而这也是我们总是将数据有序存放的原因。



When the sequence is ***unsorted***, the standard approach to search for a target value is to use loop to examine every element, until either finding the target or exhausting the data set. This is known as the ***sequential search*** algorithm runs in  time (i.e., linear time) since every element is inspected in the worst case.

＊当序列处于无序状态时，标准的做法是通过循环检查每一个元素是否匹配，直到找到目标元素或者穷尽整个序列（译者按：当然不排除最后一个元素才找到）。这被称为序列式搜索算法，因为每一个元素都需要被检视，所以时间复杂度是，也就是线性时间复杂度。

When the sequence is ***sorted*** and ***indexable***, there is a much more efficient algorithm. (For intuition, think about how you would accomplish this task by hand!) For any index , we know that all the values stored at indices  are less than or equal to the value at index , and all the values stored at indices  are greater than or equal to that at index . This observation allows us to quickly “home in” on a search target using a variant of the children’s game “high-low.” We call an element of the sequence a ***candidate*** if, at the current stage of the search, we cannot rule out that this item matches the target. The algorithm maintains two parameters, low and high, such that all the candidate entries have index at least low and at the most high. Initially, low = 0 and high = . We then compare the target value to the median candidate, that is, the item data[mid] with index



＊当序列是有序的而且是有下标的那种时，我们就可以采取一个更加高效的算法。（从直观上想一想你可以采取什么措施手动完成这个任务吧！）对于随便一个下标，我们知道所有以为下标的元素数值都不大于下标为的元素，而所有以为下标的元素数值都不小于下标为的元素。这个发现使我们也已采取一种类似于小孩子那种“高还是低”的游戏进行快速定位。在搜索进程的当前状态下，我们把序列中的一个元素设置为候选值，当然我们肯定不能保证这就是我们需要的那个数值。这个算法包含两个参数，那就是low和high，所有候选值的下标最小为low，最大为high。我们把low的初始值设置为0，high的初始值设置为。在这之后，我们将目标值与下标处于low与high的中间值的那个数值进行比较。

We consider three cases:

mid = (low + high)/2囚.

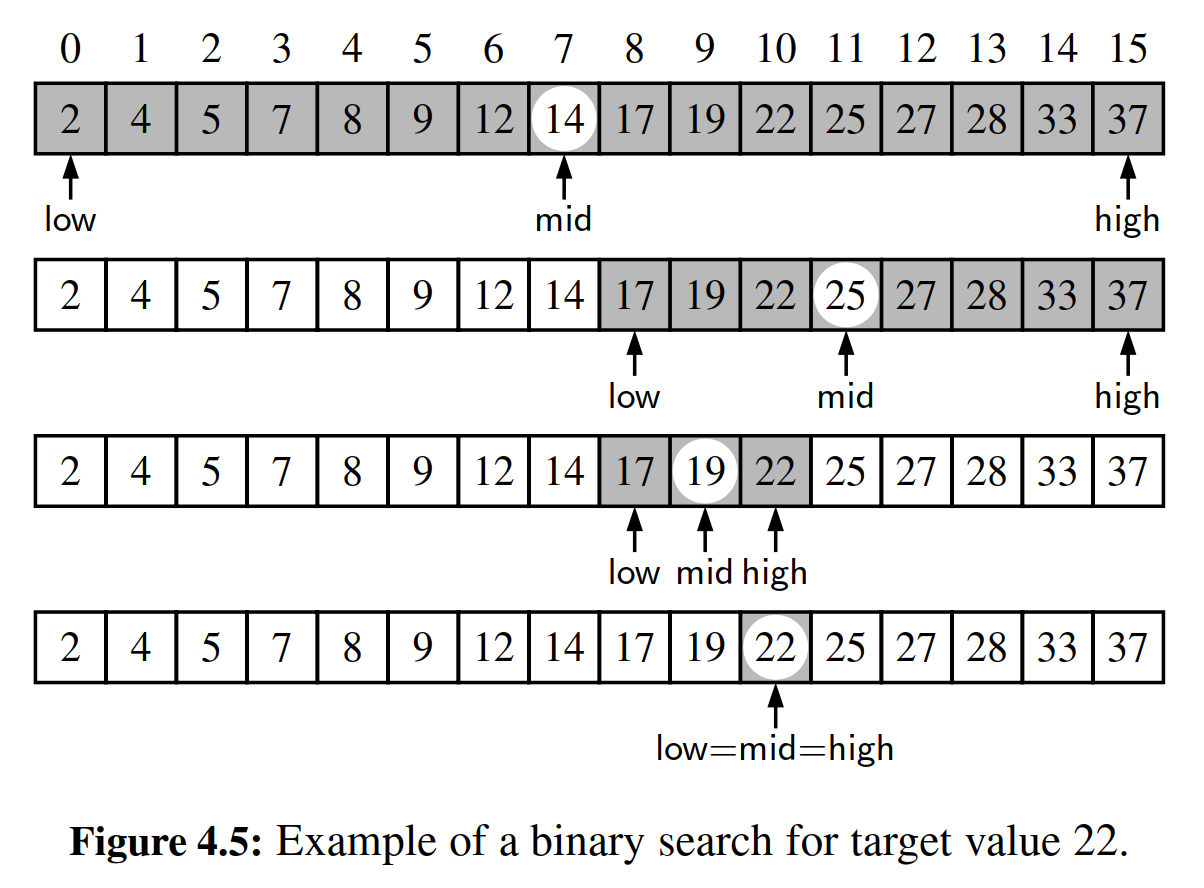
If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully.

• If target < data[mid], then we recur on the ﬁrst half of the sequence, that is, on the interval of indices from low to mid − 1.

• If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid + 1 to high.

An unsuccessful search occurs if low > high, as the interval [low, high] is empty.

This algorithm is known as binary search. We give a Python implementation in Code Fragment 4.3, and an illustration of the execution of the algorithm in Fig- ure 4.5. Whereas sequential search runs in O(n) time, the more efﬁcient binary search runs in O(log n) time. This is a signiﬁcant improvement, given that if n is one billion, log n is only 30. (We defer our formal analysis of binary search’s running time to Proposition 4.2 in Section 4.2.)

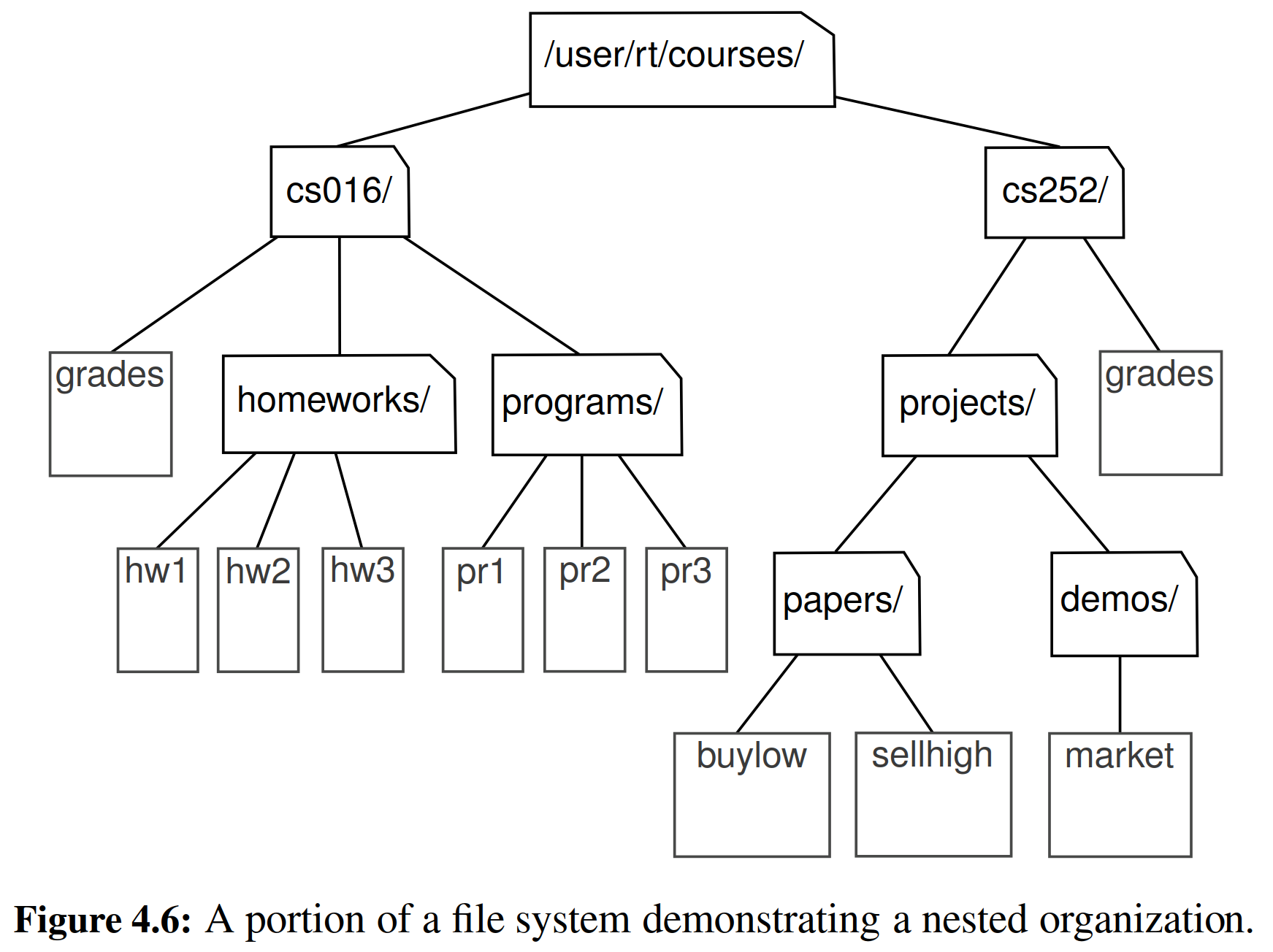


4.1.4 File Systems

＊文件系统

Modern operating systems deﬁne ﬁle-system directories (which are also sometimes called “folders”) in a recursive way. Namely, a ﬁle system consists of a top-level directory, and the contents of this directory consists of ﬁles and other directories, which in turn can contain ﬁles and other directories, and so on. The operating system allows directories to be nested arbitrarily deep (as long as there is enough space in memory), although there must necessarily be some base directories that contain only ﬁles, not further subdirectories. A representation of a portion of such a ﬁle system is given in Figure 4.6.

＊现代操作系统都用递归的方式定义了文件系统目录，或者称之为文件夹。换句话说，一个文件系统包含一个顶层目录，这个顶层目录的内容包括文件以及其他的目录，而这些被顶层目录包含的子目录也可以包含其他的文件与目录。文件系统允许这样的目录结构可以无限制地嵌套下去，当然前提是有足够的存储空间。但是总会有一些基本的底层目录，它们之下只有文件而没有子目录。（译者按：这也不一定吧，毕竟有空目录这种存在）

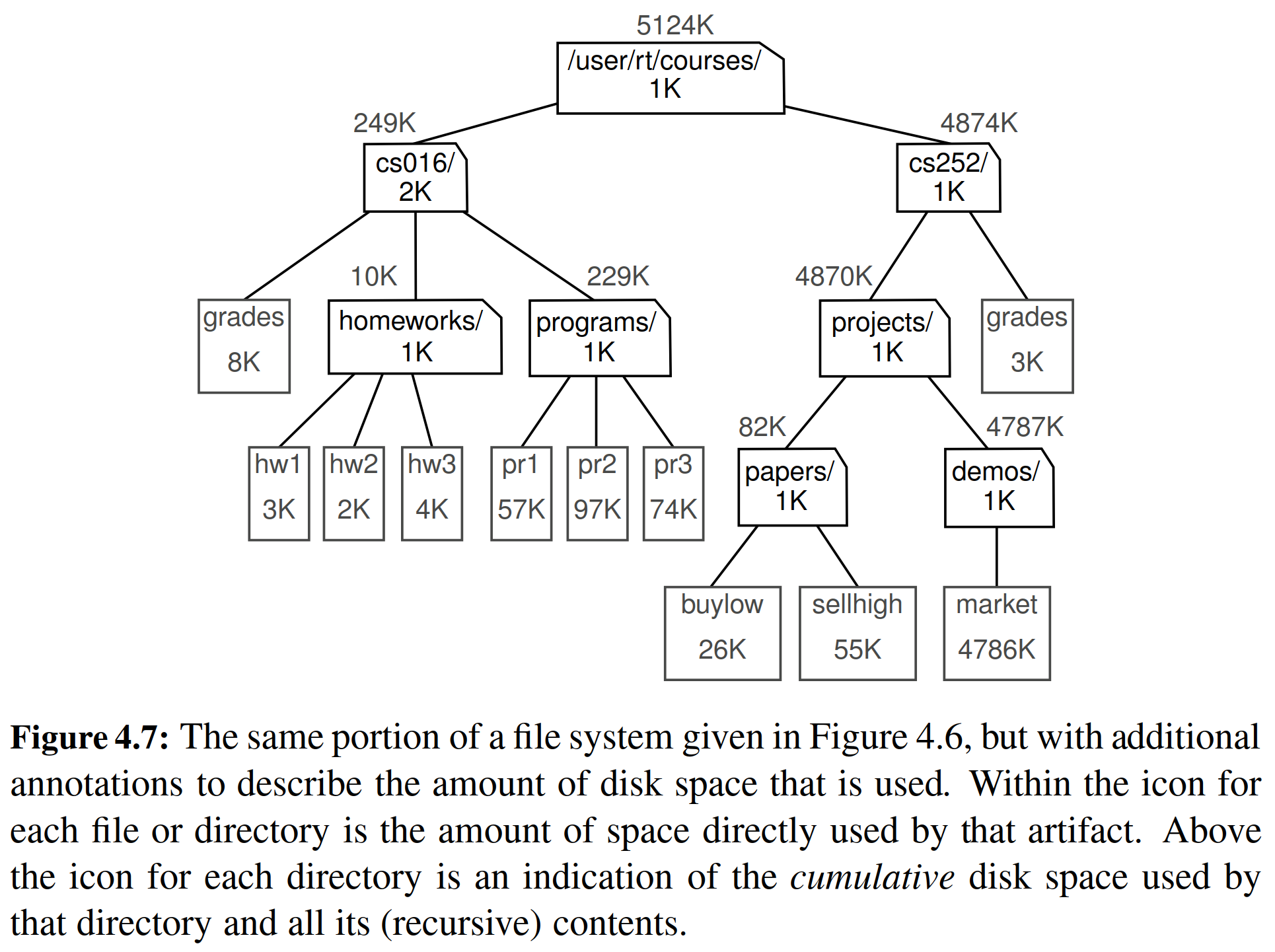


Given the recursive nature of the ﬁle-system representation, it should not come as a surprise that many common behaviors of an operating system, such as copying a directory or deleting a directory, are implemented with recursive algorithms. In this section, we consider one such algorithm: computing the total disk usage for all ﬁles and directories nested within a particular directory.

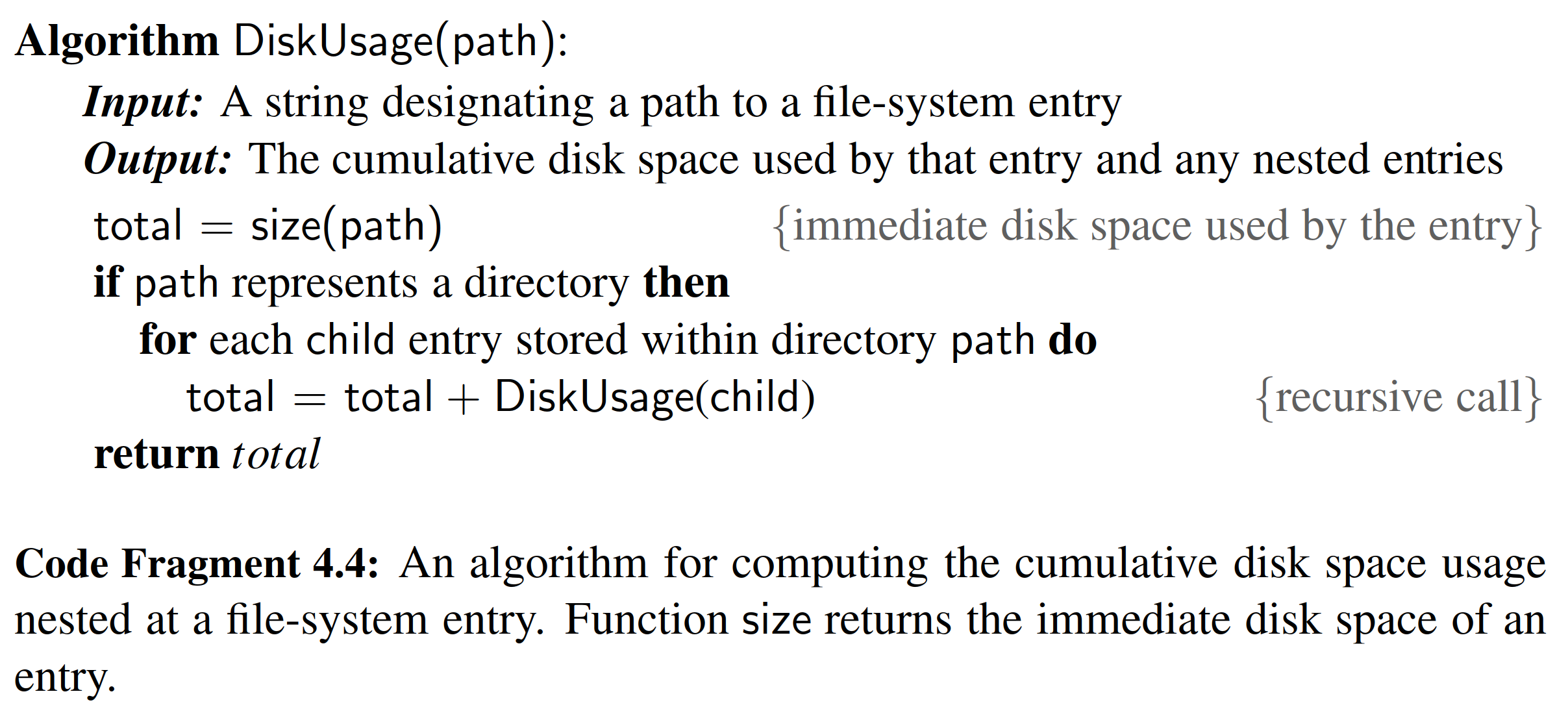
＊

For illustration, Figure 4.7 portrays the disk space being used by all entries in our sample ﬁle system. We differentiate between the immediate disk space used by each entry and the cumulative disk space used by that entry and all nested features. For example, the cs016 directory uses only 2K of immediate space, but a total of 249K of cumulative space.

＊



The cumulative disk space for an entry can be computed with a simple recursive algorithm. It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry. For example, the cumulative disk space for cs016 is 249K because it uses 2K itself, 8K cumulatively in grades, 10K cumulatively in homeworks, and 229K cumulatively in programs. Pseudo-code for this algorithm is given in Code Fragment 4.4.



**Python’s os Module**

To provide a Python implementation of a recursive algorithm for computing disk usage, we rely on Python’s os module, which provides robust tools for interacting with the operating system during the execution of a program. This is an extensive library, but we will only need the following four function:

＊为了能用Python实现这个计算磁盘用量的递归算法，我们需要Python下的一个名为os的模块。这个模块可以为我们的程序执行提供一种稳健的与操作系统交互的工具。这是一个扩展库，但是我们仅仅需要它下面的四个函数。

* **os.path.getsize(path)**Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path (e.g., /user/rt/courses)  
  ＊**os.path.getsize(path)**这个函数可以返回一个文件夹或者一个文件的磁盘使用量。而这个文件夹或者这个文件是通过路径的字符串形式给出的。（如果是文件，返回文件大小，如果是目录，就返回目录名字字符串所占用的大小）
* **os.path.isdir(path)**Return True if entry designated by string path is a directory; False otherwise.  
  ＊当**path**这个字符串所指定的目标是一个目录的时候，这个函数就会返回True，否则就会返回False。
* **os.listdir(path)**Return a list of strings that are the names of all entries within a directory designated by string path. In our sample file system, if the parameter is /user/rt/courses, this returns the list ['cs016','cs 252'].  
  ＊该函数返回的是**path**目录下的所有文件或者子目录。在我们的样本文件系统里，如果path是/user/rt/courses，那么就返回 ['cs016','cs 252'] 。
* **os.path.join(path,filename)**Compose the path string and filename string using an appropriate operating system separator between the two (e.g., the / character for a Unix/Linux system, and the \ character for Windows) Return the string that represents the full path to the file.  
  ＊这个函数使用与操作系统相匹配的分隔符（比如Unix/Linux系统下的“/”，Windows操作系统下的“\”），将路径与文件名组合成一个完成的文件路径，然后返回该完整路径所对应的字符串。

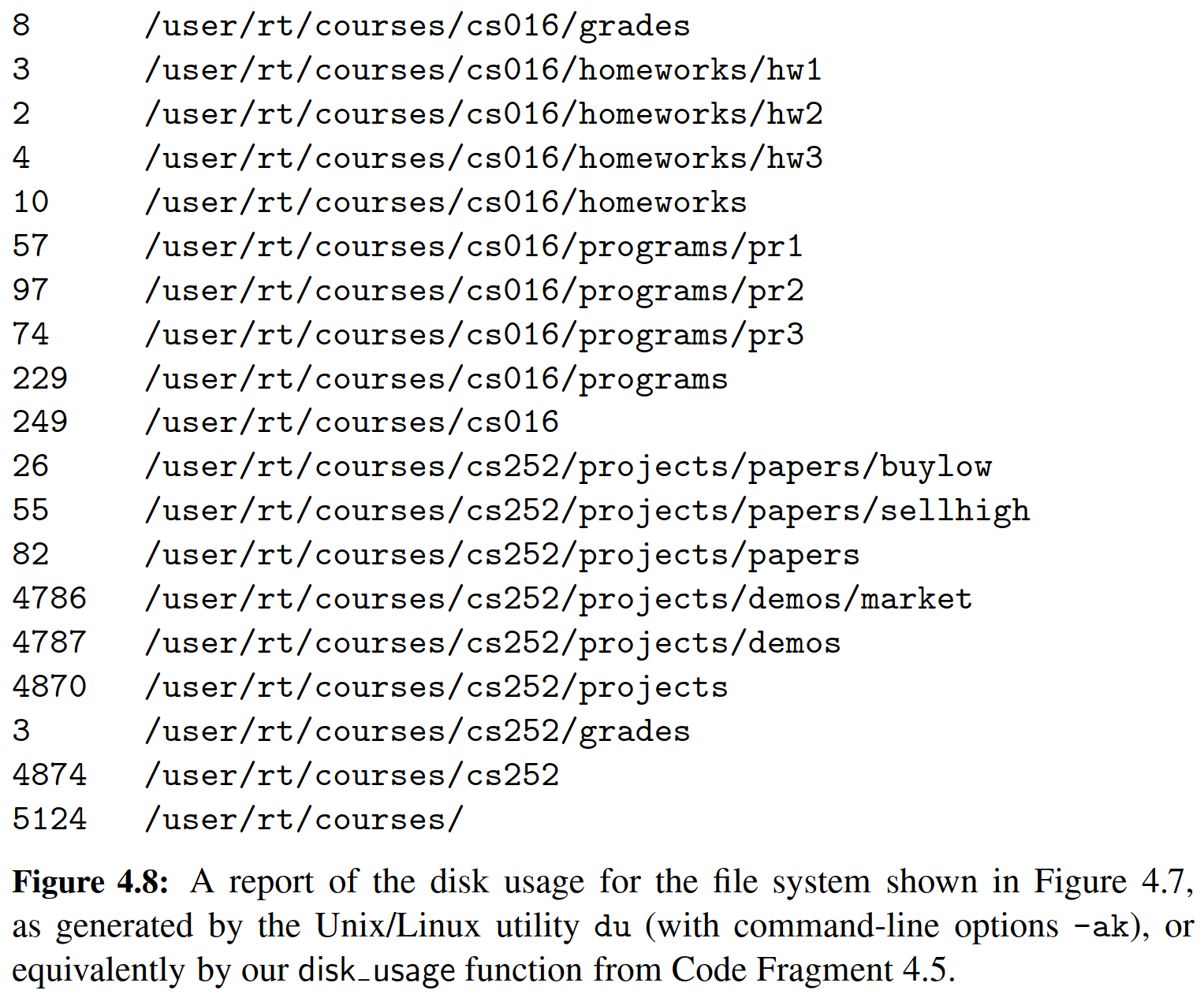
Python Implementation

With use of the os module, we now convert the algorithm from Code Fragment 4.4 into the Python implementation of Code Fragment 4.5.

Recursion Trace

To produce a different form of a recursion trace, we have included an extraneous print statement within our Python implementation (line 11 of Code Fragment 4.5). The precise format of that output intentionally mirrors output that is produced by a classic Unix/Linux utility named du (for “disk usage”). It reports the amount of disk space used by a directory and all contents nested within, and can produce a verbose report, as given in Figure 4.8.

Our implementation of the disk usage function produces an identical result, when executed on the sample ﬁle system portrayed in Figure 4.7. During the execution of the algorithm, exactly one recursive call is made for each entry in the portion of the ﬁle system that is considered. Because the print statement is made just before returning from a recursive call, the output shown in Figure 4.8 reﬂects the order in which the recursive calls are completed. In particular, we begin and end a recursive call for each entry that is nested below another entry, computing the nested cumulative disk space before we can compute and report the cumulative disk space for the containing entry. For example, we do not know the cumulative total for entry /user/rt/courses/cs016 until after the recursive calls regarding contained entries grades, homeworks, and programs complete.



4.2 Analyzing Recursive Algorithms

In Chapter 3, we introduced mathematical techniques for analyzing the efﬁciency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm. We use notations such as big-Oh to summarize the relationship between the number of operations and the input size for a problem. In this section, we demonstrate how to perform this type of running-time analysis to recursive algorithms.

With a recursive algorithm, we will account for each operation that is performed based upon the particular activation of the function that manages the ﬂow of control at the time it is executed. Stated another way, for each invocation of the function, we only account for the number of operations that are performed within the body of that activation. We can then account for the overall number of operations that are executed as part of the recursive algorithm by taking the sum, over all activations, of the number of operations that take place during each individual activation. (As an aside, this is also the way we analyze a nonrecursive function that calls other functions from within its body.)

To demonstrate this style of analysis, we revisit the four recursive algorithms presented in Sections 4.1.1 through 4.1.4: factorial computation, drawing an English ruler, binary search, and computation of the cumulative size of a ﬁle system. In general, we may rely on the intuition afforded by a recursion trace in recogniz- ing how many recursive activations occur, and how the parameterization of each activation can be used to estimate the number of primitive operations that occur within the body of that activation. However, each of these recursive algorithms has a unique structure and form.

Computing Factorials

It is relatively easy to analyze the efﬁciency of our function for computing factorials, as described in Section 4.1.1. A sample recursion trace for our factorial function was given in Figure 4.1. To compute factorial(n), we see that there are a total of n + 1 activations, as the parameter decreases from n in the ﬁrst call, to n 1

in the second call, and so on, until reaching the base case with parameter 0.

It is also clear, given an examination of the function body in Code Fragment 4.1, that each individual activation of factorial executes a constant number of operations. Therefore, we conclude that the overall number of operations for computing factorial(n) is O(n), as there are n + 1 activations, each of which accounts for O(1) operations.

Drawing an English Ruler

In analyzing the English ruler application from Section 4.1.2, we consider the fundamental question of how many total lines of output are generated by an initial call to draw interval(c), where c denotes the center length. This is a reasonable bench- mark for the overall efﬁciency of the algorithm as each line of output is based upon a call to the draw line utility, and each recursive call to draw interval with nonzero parameter makes exactly one direct call to draw line.

Some intuition may be gained by examining the source code and the recursion trace. We know that a call to draw interval(c) for c > 0 spawns two calls to draw interval(c 1) and a single call to draw line. We will rely on this intuition to prove the following claim.

Proposition 4.1: For c 0, a call to draw interval(c) results in precisely 2c 1

lines of output.

Justiﬁcation: We provide a formal proof of this claim by induction (see Section 3.4.3). In fact, induction is a natural mathematical technique for proving the correctness and efﬁciency of a recursive process. In the case of the ruler, we note that an application of draw interval(0) generates no output, and that 20 − 1 = 1 − 1 = 0. This serves as a base case for our claim.

More generally, the number of lines printed by draw interval(c) is one more than twice the number generated by a call to draw interval(c−1), as one center line is printed between two such recursive calls. By induction, we have that the number of lines is thus 1 + 2 · (2c−1 − 1)= 1 + 2c − 2 = 2c − 1.

This proof is indicative of a more mathematically rigorous tool, known as a recurrence equation that can be used to analyze the running time of a recursive algorithm. That technique is discussed in Section 12.2.4, in the context of recursive sorting algorithms.

**Performing a Binary Search**

Considering the running time of the binary search algorithm, as presented in Section 4.1.3, we observe that a constant number of primitive operations are executed at each recursive call of method of a binary search. Hence, the running time is proportional to the number of recursive calls performed. We will show that at most log n + 1 recursive calls are made during a binary search of a sequence having n elements, leading to the following claim.

Proposition 4.2: The binary search algorithm runs in O(log n) time for a sorted sequence with n elements.

Justiﬁcation: To prove this claim, a crucial fact is that with each recursive call the number of candidate entries still to be searched is given by the value

high − low + 1.

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Speciﬁcally, from the deﬁnition of mid, the number of remaining candidates is either

(mid − 1) − low + 1 = low + high − low ≤ high − low + 1

or

high − (mid + 1)+ 1 = high −

2

low + high ≤

2

high low + 1

2 .

Initially, the number of candidates is n; after the ﬁrst call in a binary search, it is at most n/2; after the second call, it is at most n/4; and so on. In general, after the jth call in a binary search, the number of candidate entries remaining is at most n/2 j. In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

n

2r < 1.

In other words (recalling that we omit a logarithm’s base when it is 2), r > log n. Thus, we have r = log n + 1, which implies that binary search runs in O(log n) time.

Computing Disk Space Usage

Our ﬁnal recursive algorithm from Section 4.1 was that for computing the overall disk space usage in a speciﬁed portion of a ﬁle system. To characterize the “prob- lem size” for our analysis, we let n denote the number of ﬁle-system entries in the portion of the ﬁle system that is considered. (For example, the ﬁle system portrayed in Figure 4.6 has n = 19 entries.)

To characterize the cumulative time spent for an initial call to the disk usage function, we must analyze the total number of recursive invocations that are made, as well as the number of operations that are executed within those invocations.

We begin by showing that there are precisely n recursive invocations of the function, in particular, one for each entry in the relevant portion of the ﬁle system. Intuitively, this is because a call to disk usage for a particular entry e of the ﬁle system is only made from within the for loop of Code Fragment 4.5 when processing the entry for the unique directory that contains e, and that entry will only be explored once.

To formalize this argument, we can deﬁne the nesting level of each entry such that the entry on which we begin has nesting level 0, entries stored directly within it have nesting level 1, entries stored within those entries have nesting level 2, and so on. We can prove by induction that there is exactly one recursive invocation of disk usage upon each entry at nesting level k. As a base case, when k = 0, the only recursive invocation made is the initial one. As the inductive step, once we know there is exactly one recursive invocation for each entry at nesting level k, we can claim that there is exactly one invocation for each entry e at nesting level k, made within the for loop for the entry at level k that contains e.

Having established that there is one recursive call for each entry of the ﬁle system, we return to the question of the overall computation time for the algorithm. It would be great if we could argue that we spend O(1) time in any single invocation of the function, but that is not the case. While there are a constant number of steps reﬂect in the call to os.path.getsize to compute the disk usage directly at that entry, when the entry is a directory, the body of the disk usage function includes a for loop that iterates over all entries that are contained within that directory. In the worst case, it is possible that one entry includes n − 1 others.

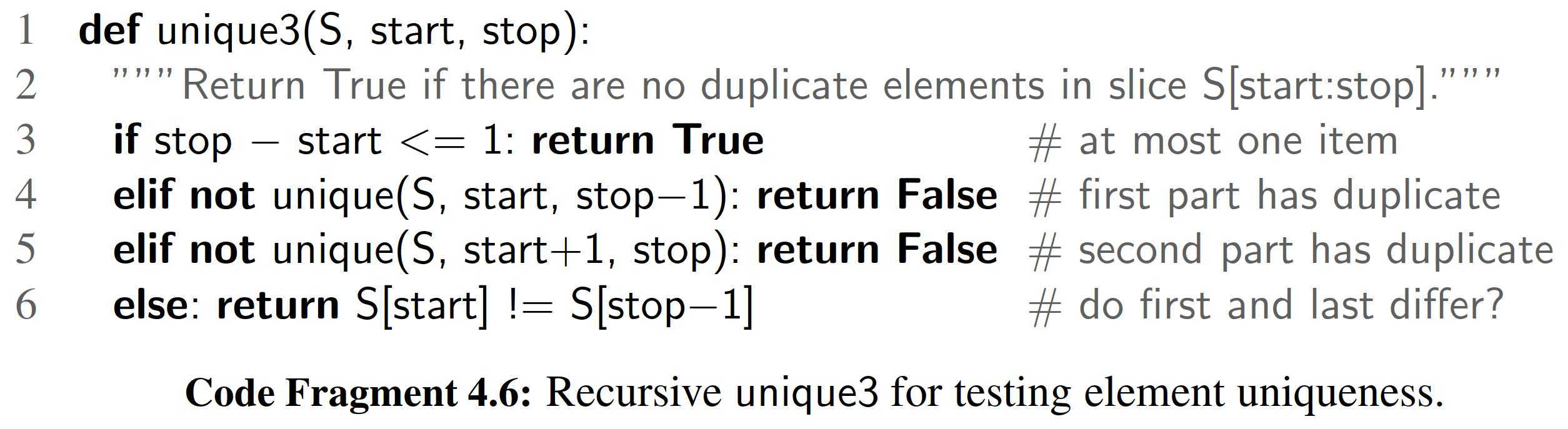
Based on this reasoning, we could conclude that there are O(n) recursive calls, each of which runs in O(n) time, leading to an overall running time that is O(n2). While this upper bound is technically true, it is not a tight upper bound. Remark- ably, we can prove the stronger bound that the recursive algorithm for disk usage completes in O(n) time! The weaker bound was pessimistic because it assumed a worst-case number of entries for each directory. While it is possible that some directories contain a number of entries proportional to n, they cannot all contain that many. To prove the stronger claim, we choose to consider the overall number of iterations of the for loop across all recursive calls. We claim there are precisely n − 1 such iteration of that loop overall. We base this claim on the fact that each iteration of that loop makes a recursive call to disk usage, and yet we have already concluded that there are a total of n calls to disk usage (including the original call). We therefore conclude that there are O(n) recursive calls, each of which uses O(1) time outside the loop, and that the overall number of operations due to the loop is O(n). Summing all of these bounds, the overall number of operations is O(n).

The argument we have made is more advanced than with the earlier examples of recursion. The idea that we can sometimes get a tighter bound on a series of operations by considering the cumulative effect, rather than assuming that each achieves a worst case is a technique called amortization; we will see a further example of such analysis in Section 5.3. Furthermore, a ﬁle system is an implicit example of a data structure known as a tree, and our disk usage algorithm is really a manifestation of a more general algorithm known as a tree traversal. Trees will be the focus of Chapter 8, and our argument about the O(n) running time of the disk usage algorithm will be generalized for tree traversals in Section 8.4.

4.3 Recursion Run Amok

Although recursion is a very powerful tool, it can easily be misused in various ways. In this section, we examine several problems in which a poorly implemented recursion causes drastic inefﬁciency, and we discuss some strategies for recognizing and avoid such pitfalls.

We begin by revisiting the element uniqueness problem, deﬁned on page 135 of Section 3.3.3. We can use the following recursive formulation to determine if all n elements of a sequence are unique. As a base case, when n = 1, the elements are trivially unique. For n ≥ 2, the elements are unique if and only if the ﬁrst n – 1 elements are unique, the last n 1 items are unique, and the ﬁrst and last elements are different (as that is the only pair that was not already checked as a subcase). A recursive implementation based on this idea is given in Code Fragment 4.6, named unique3 (to differentiate it from unique1 and unique2 from Chapter 3).



Unfortunately, this is a terribly inefﬁcient use of recursion. The nonrecursive part of each call uses *O*(1) time, so the overall running time will be proportional to the total number of recursive invocations. To analyze the problem, we let n denote the number of entries under consideration, that is, let n= stop − start.

If n = 1, then the running time of unique3 is *O*(1), since there are no recursive calls for this case. In the general case, the important observation is that a single call to unique3 for a problem of size n may result in two recursive calls on problems of size n − 1. Those two calls with size n − 1 could in turn result in four calls (two each) with a range of size n 2, and thus eight calls with size n 3 and so on. Thus, in the worst case, the total number of function calls is given by the geometric summation

1 + 2 + 4 + ···+ 2n−1, which is equal to 2n 1 by Proposition 3.5. Thus, the running time of function unique3 is O(2n). This is an incredibly inefﬁcient function for solving the element uniqueness problem. Its inefﬁciency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly, which is something we address in Exercise C-4.11.

An Ineﬃcient Recursion for Computing Fibonacci Numbers

In Section 1.8, we introduced a process for generating the Fibonacci numbers, which can be deﬁned recursively as follows:

F0 = 0

F1 = 1

Fn = Fn−2 + Fn−1 for n > 1.

Ironically, a direct implementation based on this deﬁnition results in the function bad ﬁbonacci shown in Code Fragment 4.7, which computes the sequence of Fibonacci numbers by making two recursive calls in each non-base case.

1 def bad ﬁbonacci(n):

2 ”””Return the nth Fibonacci number.”””

3 if n <= 1:

4 return n

5 else:

6 return bad ﬁbonacci(n−2) + bad ﬁbonacci(n−1)

Code Fragment 4.7: Computing the nth Fibonacci number using binary recursion.

Unfortunately, such a direct implementation of the Fibonacci formula results in a terribly inefﬁcient function. Computing the nth Fibonacci number in this way requires an exponential number of calls to the function. Speciﬁcally, let cn denote the number of calls performed in the execution of bad ﬁbonacci(n). Then, we have the following values for the cn’s:

c0 = 1

c1 = 1

c2 = 1 + c0 + c1 = 1 + 1 + 1 = 3

c3 = 1 + c1 + c2 = 1 + 1 + 3 = 5

c4 = 1 + c2 + c3 = 1 + 3 + 5 = 9

c5 = 1 + c3 + c4 = 1 + 5 + 9 = 15

c6 = 1 + c4 + c5 = 1 + 9 + 15 = 25

c7 = 1 + c5 + c6 = 1 + 15 + 25 = 41

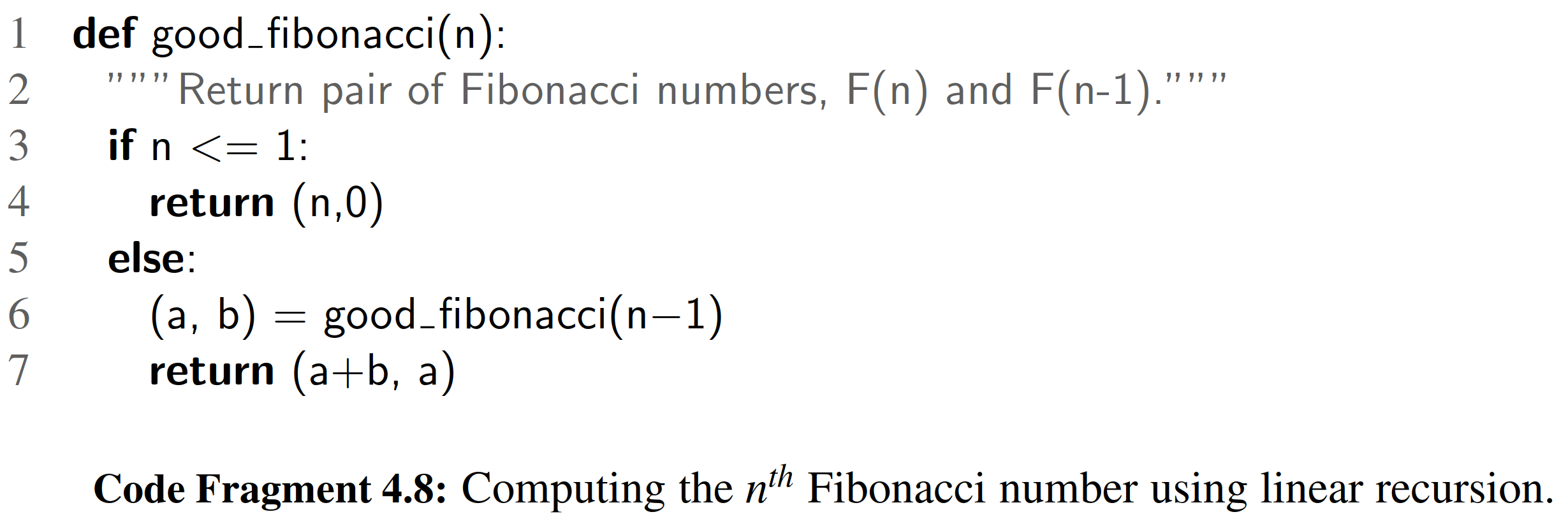
c8 = 1 + c6 + c7 = 1 + 25 + 41 = 67

If we follow the pattern forward, we see that the number of calls more than doubles for each two consecutive indices. That is, c4 is more than twice c2, c5 is more than twice c3, c6 is more than twice c4, and so on. Thus, cn > 2n/2, which means that bad ﬁbonacci(n) makes a number of calls that is exponential in n.

An Eﬃcient Recursion for Computing Fibonacci Numbers

We were tempted into using the bad recursion formulation because of the way the nth Fibonacci number, Fn, depends on the two previous values, Fn−2 and Fn−1. But notice that after computing Fn−2, the call to compute Fn−1 requires its own recursive call to compute Fn−2, as it does not have knowledge of the value of Fn−2 that was computed at the earlier level of recursion. That is duplicative work. Worse yet, both of those calls will need to (re)compute the value of Fn−3, as will the computation of Fn−1. This snowballing effect is what leads to the exponential running time of bad recursion.

We can compute Fn much more efﬁciently using a recursion in which each invocation makes only one recursive call. To do so, we need to redeﬁne the expectations of the function. Rather than having the function return a single value, which is the nth Fibonacci number, we deﬁne a recursive function that returns a pair of consecutive Fibonacci numbers (Fn, Fn−1), using the convention F−1 = 0. Although it seems to be a greater burden to report two consecutive Fibonacci numbers in- stead of one, passing this extra information from one level of the recursion to the next makes it much easier to continue the process. (It allows us to avoid having to recompute the second value that was already known within the recursion.) An implementation based on this strategy is given in Code Fragment 4.8.



In terms of efﬁciency, the difference between the bad recursion and the good recursion for this problem is like night and day. The bad ﬁbonacci function uses exponential time. We claim that the execution of function good ﬁbonacci(n) takes *O*(*n*) time. Each recursive call to good ﬁbonacci decreases the argument n by 1; therefore, a recursion trace includes a series of n function calls. Because the nonrecursive work for each call uses constant time, the overall computation executes in *O*(*n*) time.

4.3.1 Maximum Recursive Depth in Python

Another danger in the misuse of recursion is known as ***inﬁnite recursion***. If each recursive call makes another recursive call, without ever reaching a base case, then we have an inﬁnite series of such calls. This is a fatal error. An inﬁnite recursion can quickly swamp computing resources, not only due to rapid use of the CPU, but because each successive call creates an activation record requiring additional memory. A blatant example of an ill-formed recursion is the following:

**def** ﬁb(n):

return ﬁb(n) # ﬁb(n) equals ﬁb(n)

However, there are far more subtle errors that can lead to an inﬁnite recursion. Revisiting our implementation of binary search in Code Fragment 4.3, in the ﬁnal case (line 17) we make a recursive call on the right portion of the sequence, in particular going from index mid+1 to high. Had that line instead been written as

**return** binary\_search(data, target, mid, high)

# note the use of mid

this could result in an inﬁnite recursion. In particular, when searching a range of two elements, it becomes possible to make a recursive call on the identical range.

A programmer should ensure that each recursive call is in some way progressing toward a base case (for example, by having a parameter value that decreases with each call). However, to combat against inﬁnite recursions, the designers of Python made an intentional decision to limit the overall number of function activations that can be simultaneously active. The precise value of this limit depends upon the Python distribution, but a typical default value is 1000. If this limit is reached, the Python interpreter raises a RuntimeError with a message, maximum recursion depth exceeded.

For many legitimate applications of recursion, a limit of 1000 nested function calls sufﬁces. For example, our binary search function (Section 4.1.3) has O(log n) recursive depth, and so for the default recursive limit to be reached, there would need to be 21000 elements (far, far more than the estimated number of atoms in the universe). However, in the next section we discuss several algorithms that have recursive depth proportional to n. Python’s artiﬁcial limit on the recursive depth could disrupt such otherwise legitimate computations.

Fortunately, the Python interpreter can be dynamically reconﬁgured to change the default recursive limit. This is done through use of a module named sys, which supports a getrecursionlimit function and a setrecursionlimit. Sample usage of those functions is demonstrated as follows:

**import** sys

old = sys.getrecursionlimit()

# perhaps 1000 is typical

sys.setrecursionlimit(1000000)

# change to allow 1 million nested calls

4.4 Further Examples of Recursion

In the remainder of this chapter, we provide additional examples of the use of recursion. We organize our presentation by considering the maximum number of recursive calls that may be started from within the body of a single activation.

* If a recursive call starts at most one other, we call this a linear recursion.
* If a recursive call may start two others, we call this a binary recursion.
* If a recursive call may start three or more others, this is multiple recursion.

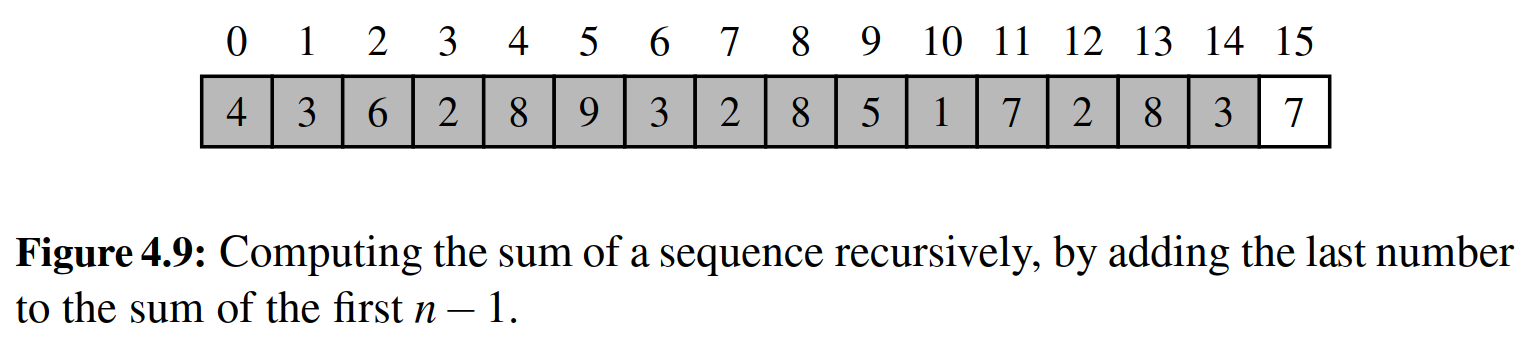
4.4.1 Linear Recursion

If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is know as linear recursion. Of the recursions we have seen so far, the implementation of the factorial function (Section 4.1.1) and the good ﬁbonacci function (Section 4.3) are clear examples of linear recursion. More interestingly, the binary search algorithm (Section 4.1.3) is also an example of linear recursion, despite the “binary” terminology in the name. The code for binary search (Code Fragment 4.3) includes a case analysis with two branches that lead to recursive calls, but only one of those calls can be reached during a particular execution of the body.

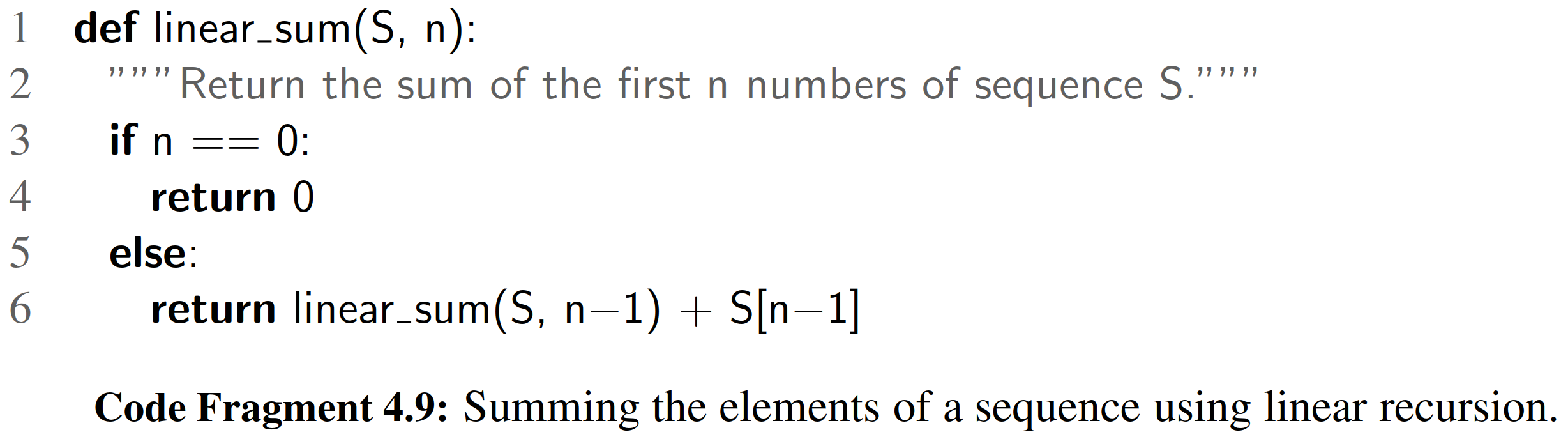
A consequence of the deﬁnition of linear recursion is that any recursion trace will appear as a single sequence of calls, as we originally portrayed for the factorial function in Figure 4.1 of Section 4.1.1. Note that the linear recursion terminology reﬂects the structure of the recursion trace, not the asymptotic analysis of the running time; for example, we have seen that binary search runs in *O*(log n) time.

**Summing the Elements of a Sequence Recursively**

Linear recursion can be a useful tool for processing a data sequence, such as a Python list. Suppose, for example, that we want to compute the sum of a sequence, S, of n integers. We can solve this summation problem using linear recursion by observing that the sum of all n integers in S is trivially 0, if n = 0, and otherwise that it is the sum of the ﬁrst n − 1 integers in S plus the last element in S. (See Figure 4.9.)



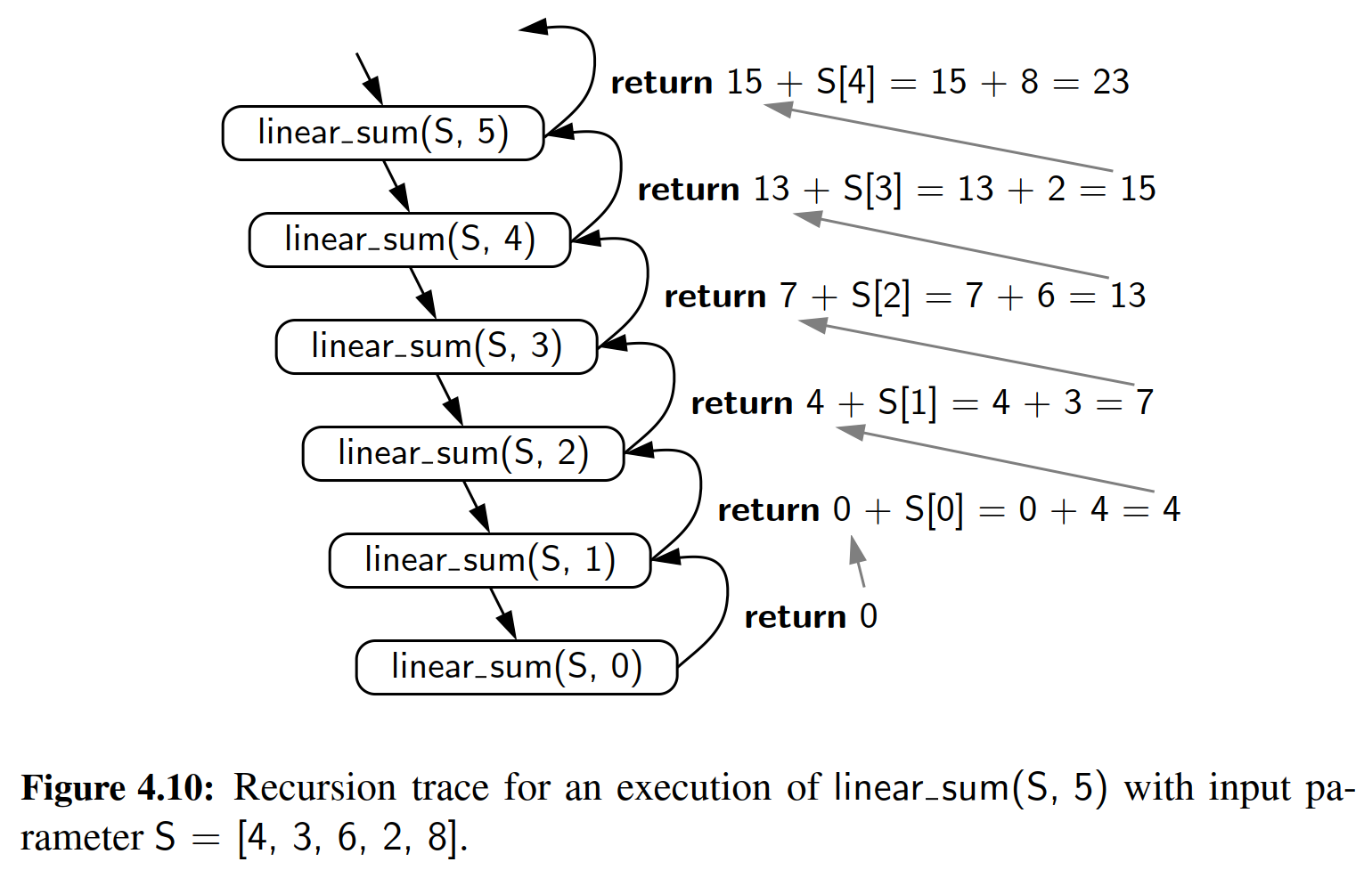
A recursive algorithm for computing the sum of a sequence of numbers based on this intuition is implemented in Code Fragment 4.9.



recursion trace of the linear sum function for a small example is given in

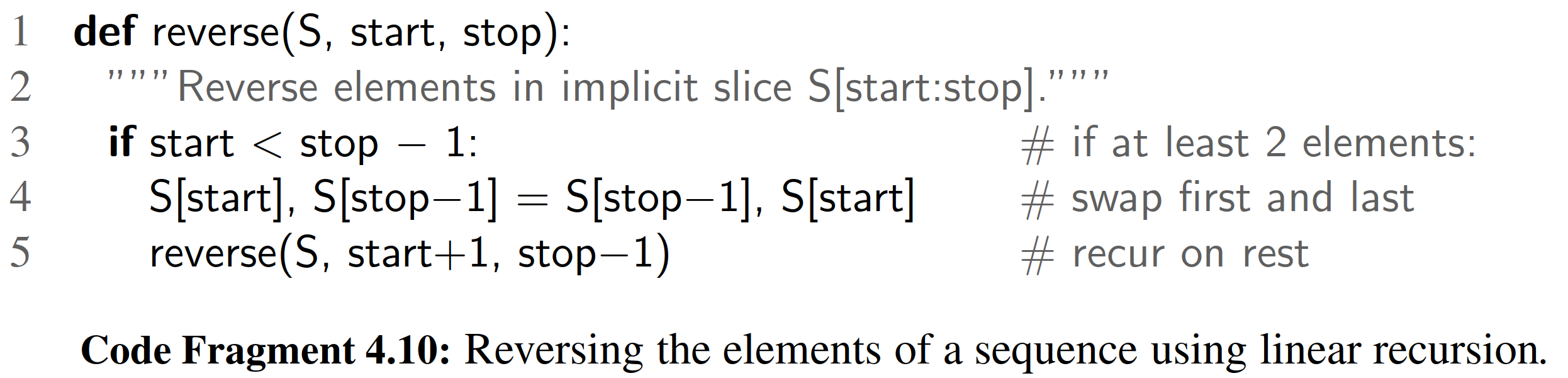
Figure 4.10. For an input of size n, the linear sum algorithm makes n + 1 function

calls. Hence, it will take O(n) time, because it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that the memory space used by the algorithm (in addition to the sequence S) is also O(n), as we use a constant amount of memory space for each of the n + 1 activation records in the trace at the time we make the ﬁnal recursive call (with n = 0).



Reversing a Sequence with Recursion

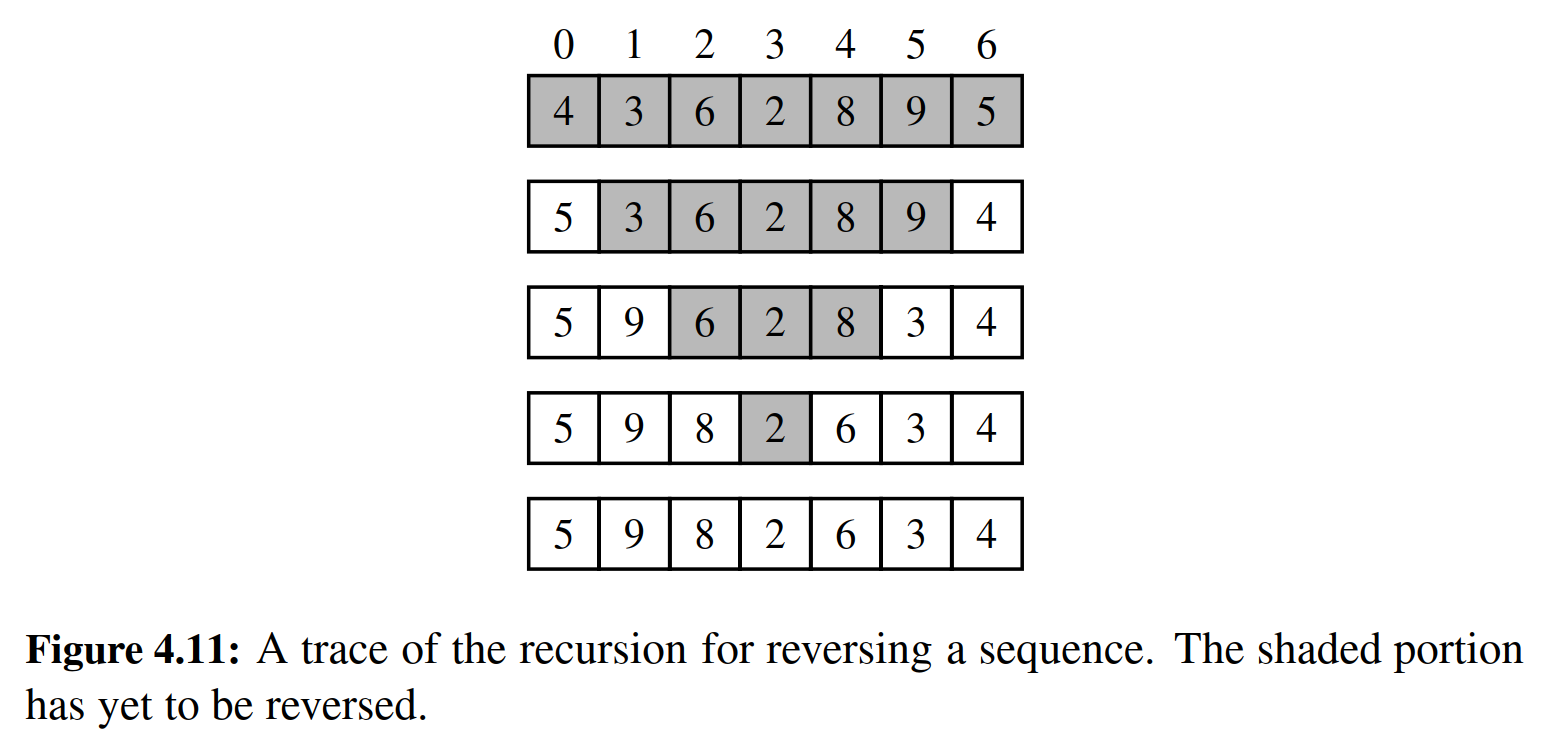
Next, let us consider the problem of reversing the n elements of a sequence, S, so that the ﬁrst element becomes the last, the second element becomes second to the last, and so on. We can solve this problem using linear recursion, by observing that the reversal of a sequence can be achieved by swapping the ﬁrst and last elements and then recursively reversing the remaining elements. We present an implementation of this algorithm in Code Fragment 4.10, using the convention that the ﬁrst time we call this algorithm we do so as reverse(S, 0, len(S)).



Note that there are two implicit base case scenarios: When start == stop, the implicit range is empty, and when start == stop−1, the implicit range has only one element. In either of these cases, there is no need for action, as a sequence with zero elements or one element is trivially equal to its reversal. When otherwise invoking recursion, we are guaranteed to make progress towards a base case, as the difference, stop−start, decreases by two with each call (see Figure 4.11). If n

is even, we will eventually reach the start == stop case, and if n is odd, we will eventually reach the start == stop − 1 case.

The above argument implies that the recursive algorithm of Code Fragment 4.10 is guaranteed to terminate after a total of 1 + n recursive calls. Since each call involves a constant amount of work, the entire process runs in O(n) time.



Recursive Algorithms for Computing Powers

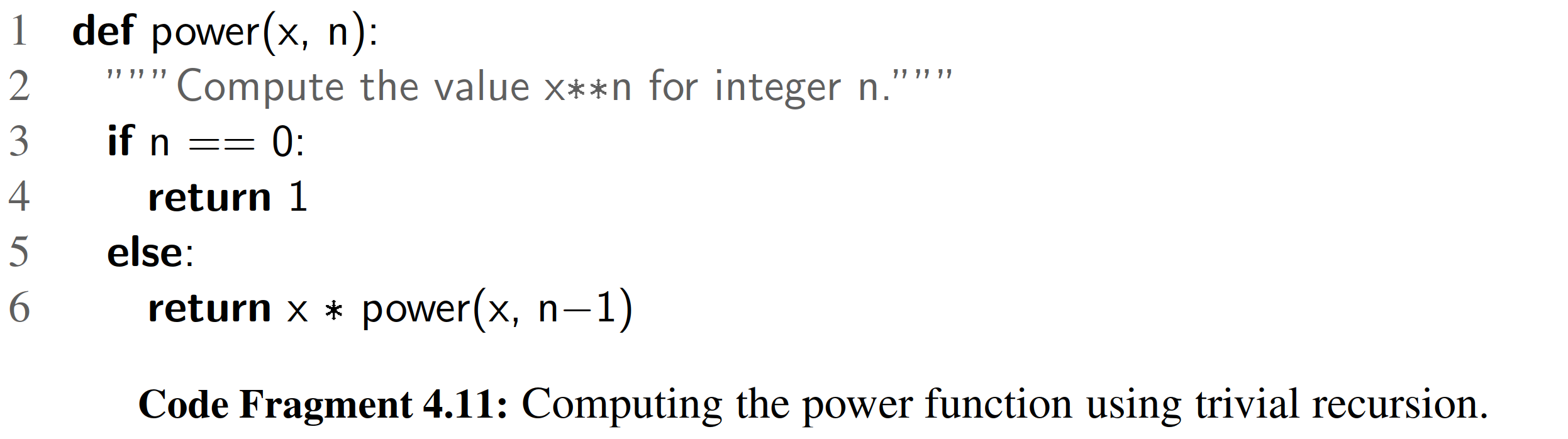
As another interesting example of the use of linear recursion, we consider the problem of raising a number x to an arbitrary nonnegative integer, n. That is, we wish to compute the power function, deﬁned as power(x, n) = x*n*. (We use the name “power” for this discussion, to differentiate from the built-in function pow that pro- vides such functionality.) We will consider two different recursive formulations for the problem that lead to algorithms with very different performance.

A trivial recursive deﬁnition follows from the fact that xn = x · xn−1 for n > 0.

power(x, n)= 1 if n = 0

x · power(x, n − 1) otherwise.

This deﬁnition leads to a recursive algorithm shown in Code Fragment 4.11.



A recursive call to this version of power(x, n) runs in O(n) time. Its recursion trace has structure very similar to that of the factorial function from Figure 4.1, with the parameter decreasing by one with each call, and constant work performed at each of n + 1 levels.

However, there is a much faster way to compute the power function using an alternative deﬁnition that employs a squaring technique. Let k = n denote the ﬂoor of the division (expressed as n // 2 in Python). We consider the expression

(xk)2. When n is even, I n = n and therefore (xk)2 = (xn \2 = xn. When n is odd,

2 2

I n = n−1 and (xk)2 = xn−1, and therefore xn = x · (xk)2, just as 213 = 2 · 26 · 26.

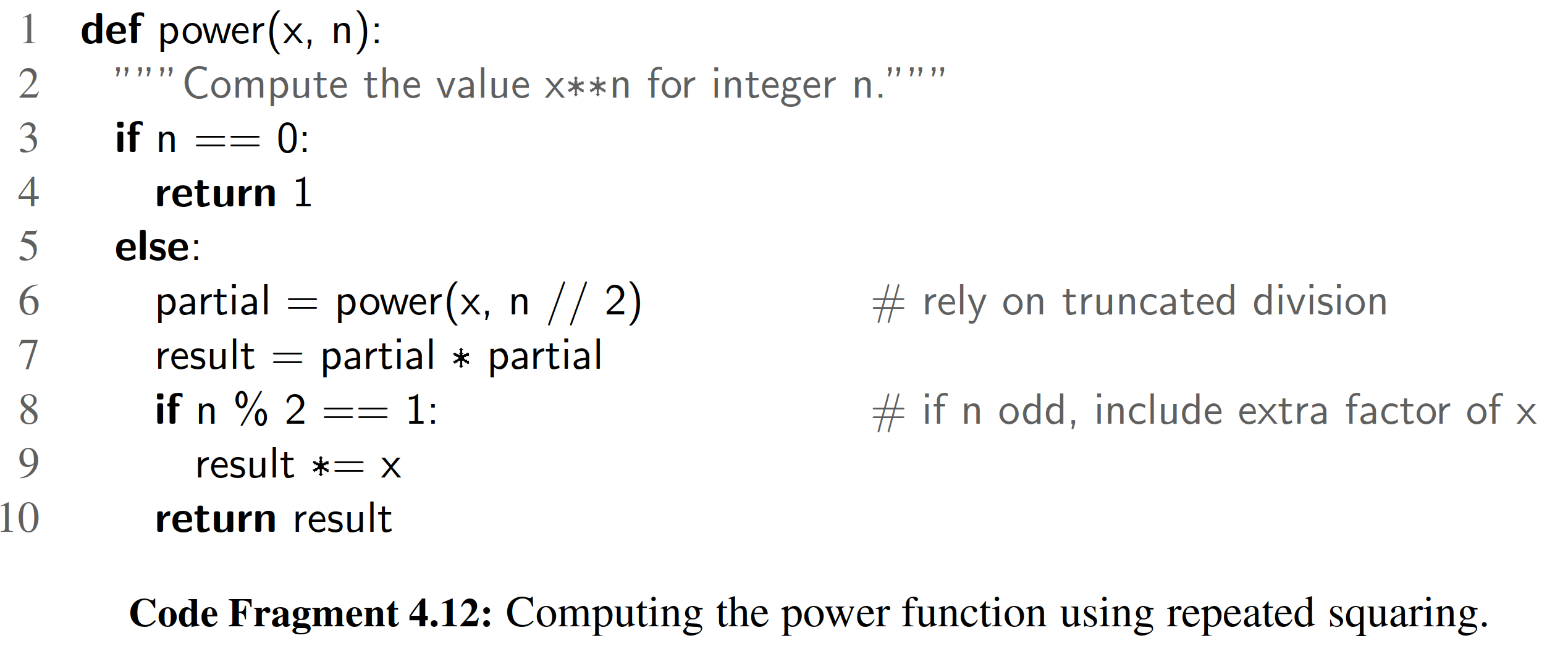
( I ))

if n = 0

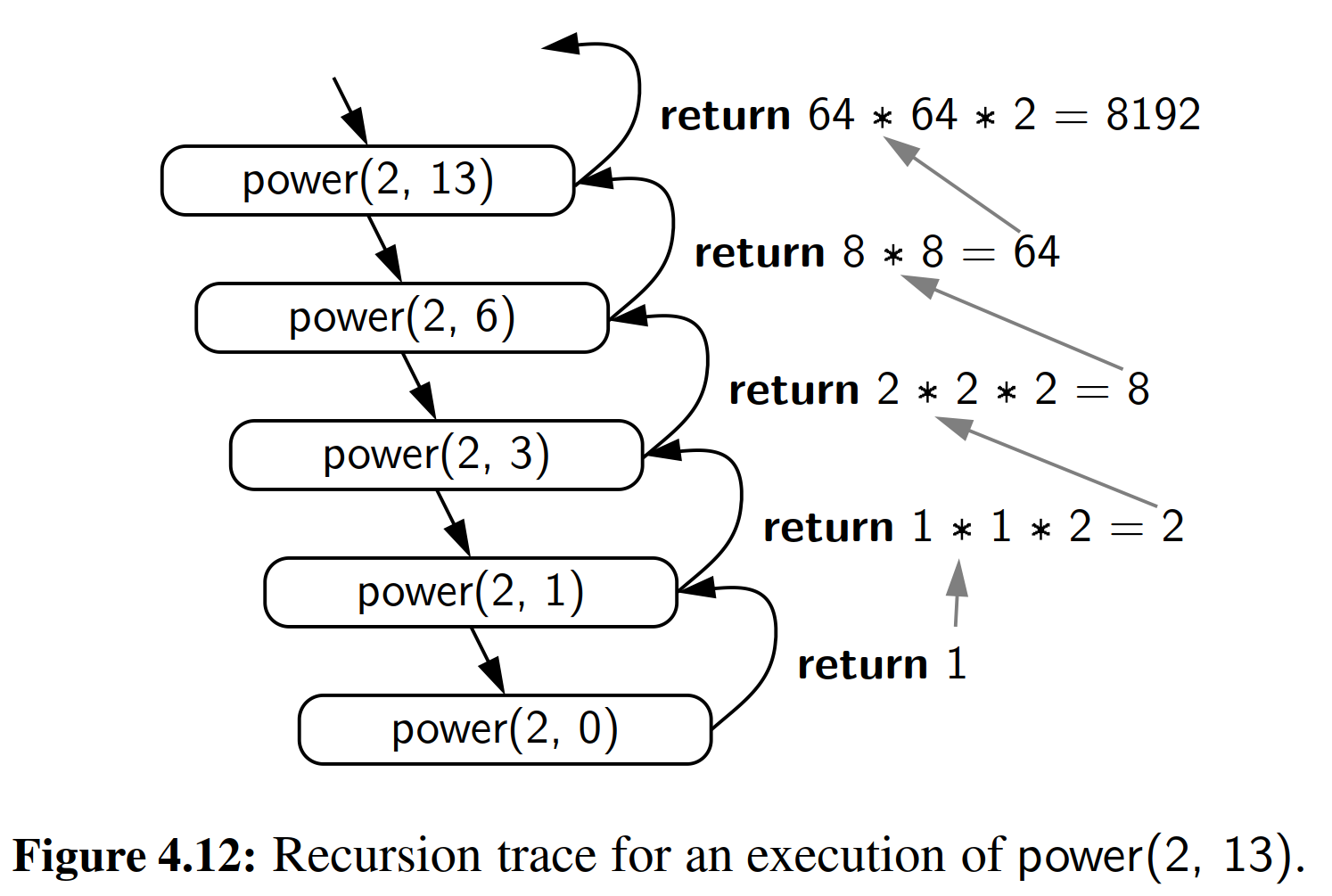
n2

2

If we were to implement this recursion making two recursive calls to compute power(x, I n ) · power(x, I n ), a trace of the recursion would demonstrate O(n) as a partial result, and then multiplying it by itself. An implementation based on this recursive deﬁnition is given in Code Fragment 4.12.



To illustrate the execution of our improved algorithm, Figure 4.12 provides a recursion trace of the computation power(2, 13).



To analyze the running time of the revised algorithm, we observe that the exponent in each recursive call of function power(x, n) is at most half of the preceding exponent. As we saw with the analysis of binary search, the number of times that we can divide n in half before getting to one or less is O(log n). Therefore, our new formulation of the power function results in O(log n) recursive calls. Each individual activation of the function uses O(1) operations (excluding the recursive calls), and so the total number of operations for computing power(x,n) is O(log n). This is a signiﬁcant improvement over the original O(n)-time algorithm.

The improved version also provides signiﬁcant saving in reducing the memory usage. The ﬁrst version has a recursive depth of O(n), and therefore O(n) activation records are simultaneous stored in memory. Because the recursive depth of the improved version is O(log n), its memory usages is O(log n) as well.

4.4.2 Binary Recursion

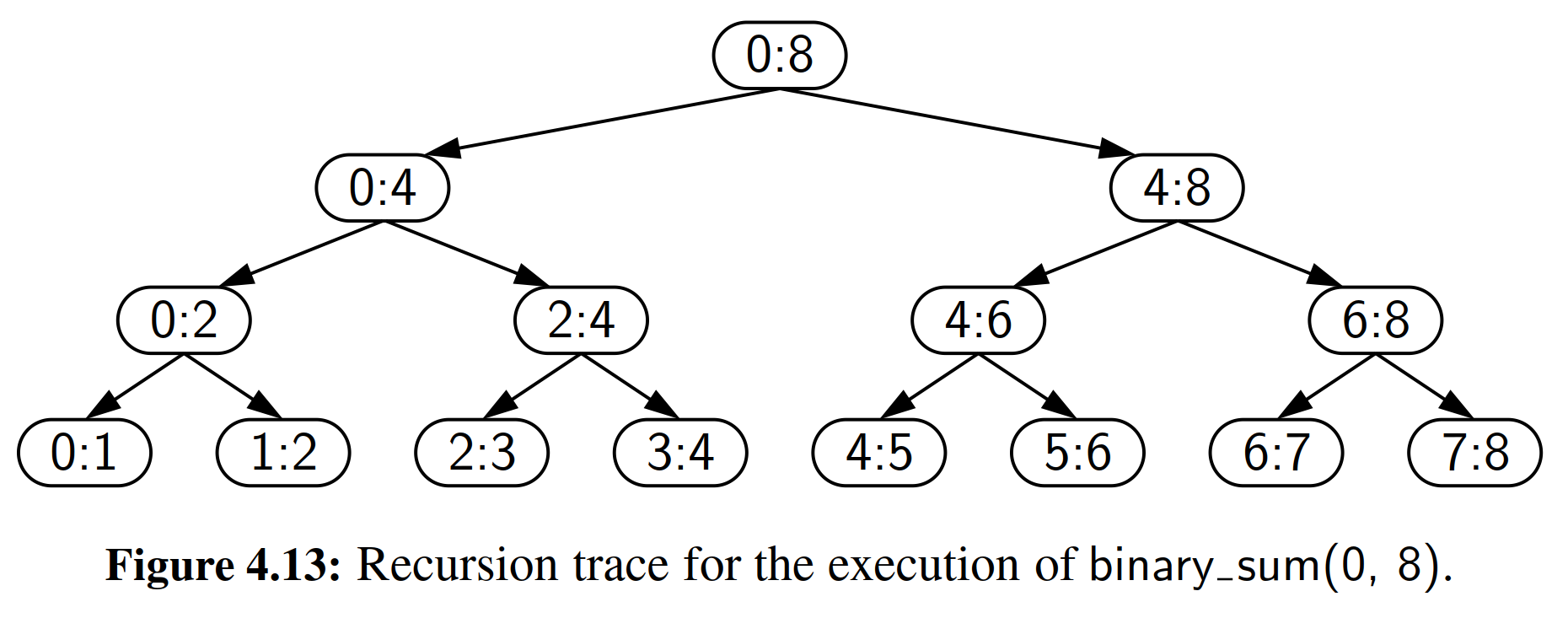
When a function makes two recursive calls, we say that it uses binary recursion. We have already seen several examples of binary recursion, most notably when drawing the English ruler (Section 4.1.2), or in the bad ﬁbonacci function of Section 4.3. As another application of binary recursion, let us revisit the problem of summing the n elements of a sequence, S, of numbers. Computing the sum of one or zero elements is trivial. With two or more elements, we can recursively compute the sum of the ﬁrst half, and the sum of the second half, and add these sums together. Our implementation of such an algorithm, in Code Fragment 4.13, is initially invoked as binary sum(A, 0, len(A)).



To analyze algorithm binary sum, we consider, for simplicity, the case where n is a power of two. Figure 4.13 shows the recursion trace of an execution of binary sum(0, 8). We label each box with the values of parameters start:stop for that call. The size of the range is divided in half at each recursive call, and so the depth of the recursion is 1 + log2 n. Therefore, binary sum uses O(log n) amount of additional space, which is a big improvement over the O(n) space used by the linear sum function of Code Fragment 4.9. However, the running time of binary sum is O(n), as there are 2n − 1 function calls, each requiring constant time.

To analyze algorithm binary sum, we consider, for simplicity, the case where n is a power of two. Figure 4.13 shows the recursion trace of an execution of binary sum(0, 8). We label each box with the values of parameters start:stop for that call. The size of the range is divided in half at each recursive call, and so the depth of the recursion is 1 + log2 n. Therefore, binary sum uses O(log n) amount of additional space, which is a big improvement over the O(n) space used by the linear sum function of Code Fragment 4.9. However, the running time of

binary sum is O(n), as there are 2n − 1 function calls, each requiring constant time.



4.4.3 Multiple Recursion

Generalizing from binary recursion, we deﬁne multiple recursion as a process in which a function may make more than two recursive calls. Our recursion for analyzing the disk space usage of a ﬁle system (see Section 4.1.4) is an example of multiple recursion, because the number of recursive calls made during one invocation was equal to the number of entries within a given directory of the ﬁle system. Another common application of multiple recursion is when we want to enumerate various conﬁgurations in order to solve a combinatorial puzzle. For example, the following are all instances of what are known as summation puzzles:

pot + pan = bib dog + cat = pig boy + girl = baby

To solve such a puzzle, we need to assign a unique digit (that is, 0, 1,... , 9) to each letter in the equation, in order to make the equation true. Typically, we solve such a puzzle by using our human observations of the particular puzzle we are trying to solve to eliminate conﬁgurations (that is, possible partial assignments of digits to letters) until we can work though the feasible conﬁgurations left, testing for the correctness of each one.

If the number of possible conﬁgurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one, without employing any human observations. In addition, such an algorithm can use multiple recursion to work through the conﬁgurations in a systematic way. We show pseudo- code for such an algorithm in Code Fragment 4.14. To keep the description general enough to be used with other puzzles, the algorithm enumerates and tests all k- length sequences without repetitions of the elements of a given universe U . We build the sequences of k elements by the following steps:

1. Recursively generating the sequences of k − 1 elements

2. Appending to each such sequence an element not already contained in it.

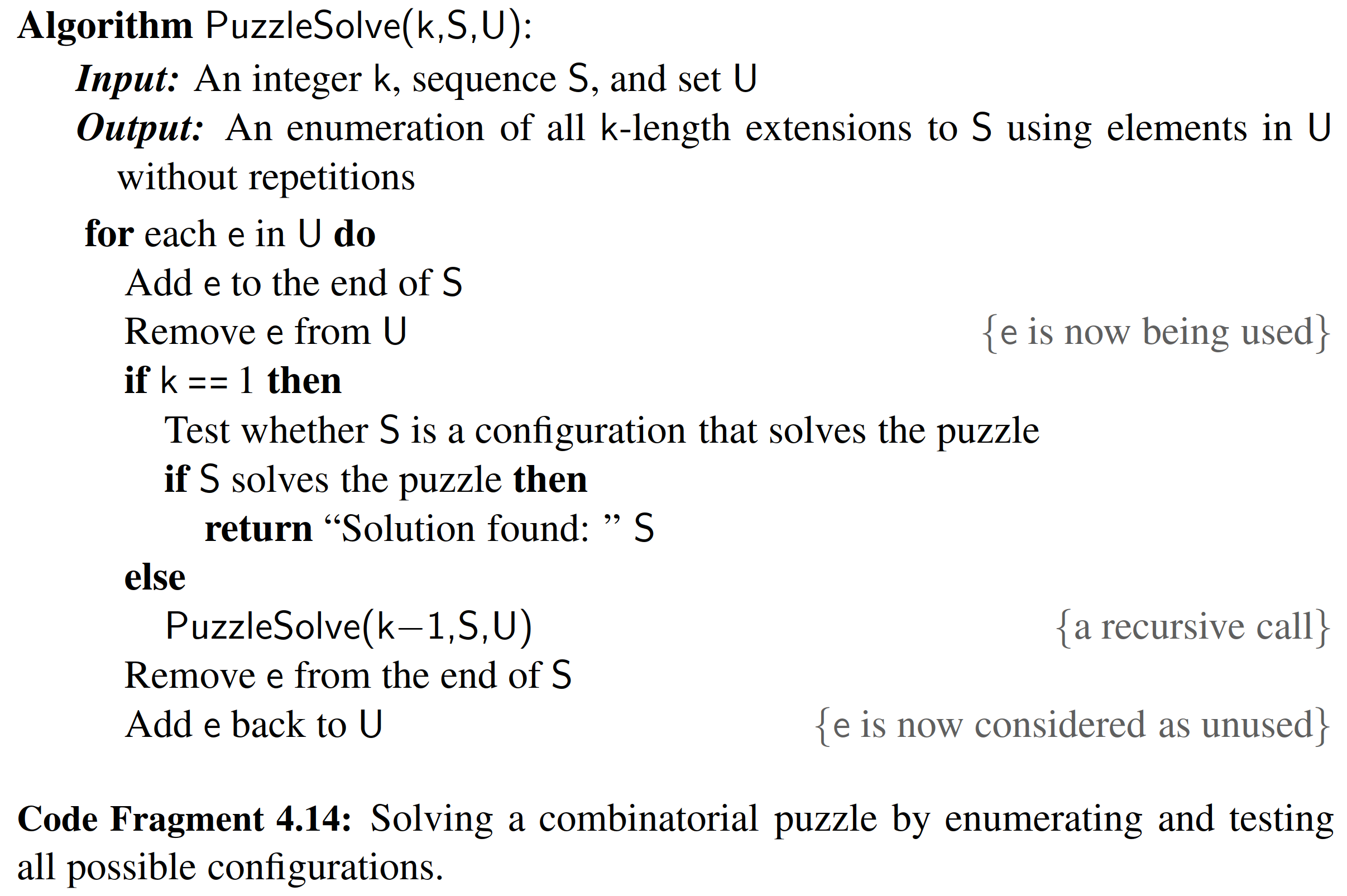
Throughout the execution of the algorithm, we use a set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U .

Another way to look at the algorithm of Code Fragment 4.14 is that it enumer- ates every possible size-k ordered subset of U , and tests each subset for being a possible solution to our puzzle.

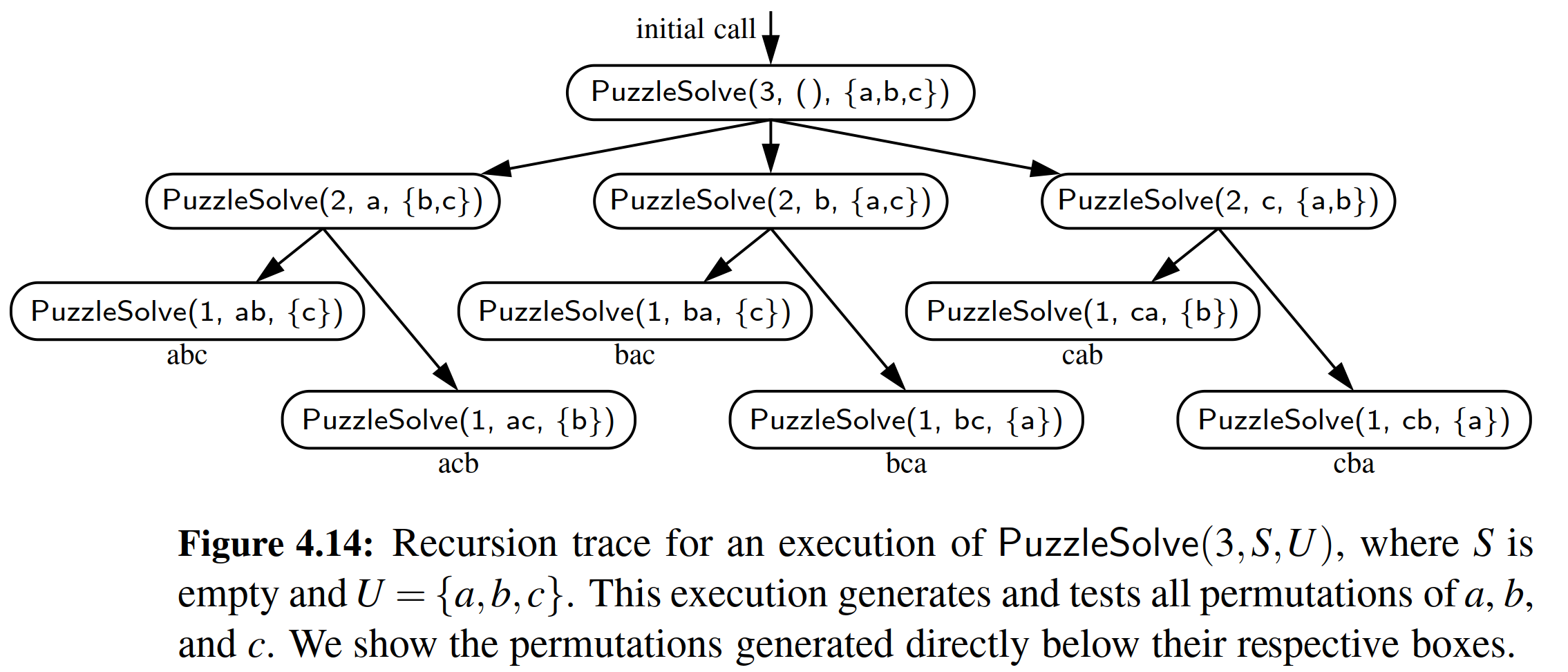
For summation puzzles, U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and each position in the

sequence corresponds to a given letter. For examp

le, the ﬁrst position could stand for b, the second for o, the third for y, and so on.



In Figure 4.14, we show a recursion trace of a call to PuzzleSolve(3, S,U ), where S is empty and U = a, b, c . During the execution, all the permutations of the three characters are generated and tested. Note that the initial call makes three recursive calls, each of which in turn makes two more. If we had executed PuzzleSolve(3, S,U ) on a set U consisting of four elements, the initial call would have made four recursive calls, each of which would have a trace looking like the one in Figure 4.14.



4.5 Designing Recursive Algorithms

In general, an algorithm that uses recursion typically has the following form:

* Test for base cases. We begin by testing for a set of base cases (there should be at least one). These base cases should be deﬁned so that every possible chain of recursive calls will eventually reach a base case, and the handling of each base case should not use recursion.
* Recur. If not a base case, we perform one or more recursive calls. This recursive step may involve a test that decides which of several possible recursive calls to make. We should deﬁne each possible recursive call so that it makes progress towards a base case.

**Parameterizing a Recursion**

To design a recursive algorithm for a given problem, it is useful to think of the different ways we might deﬁne subproblems that have the same general structure as the original problem. If one has difﬁculty ﬁnding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be deﬁned.

A successful recursive design sometimes requires that we redeﬁne the original problem to facilitate similar-looking subproblems. Often, this involved reparameterizing the signature of the function. For example, when performing a binary search in a sequence, a natural function signature for a caller would appear as binary search(data, target). However, in Section 4.1.3, we deﬁned our function with calling signature binary search(data, target, low, high), using the additional parameters to demarcate sublists as the recursion proceeds. This change in parameterization is critical for binary search. If we had insisted on the cleaner signature, binary search(data, target), the only way to invoke a search on half the list would have been to make a new list instance with only those elements to send as the ﬁrst parameter. However, making a copy of half the list would already take O(n) time, negating the whole beneﬁt of the binary search algorithm.

If we wished to provide a cleaner public interface to an algorithm like bi- nary search, without bothering a user with the extra parameters, a standard technique is to make one function for public use with the cleaner interface, such as binary search(data, target), and then having its body invoke a nonpublic utility function having the desired recursive parameters.

You will see that we similarly reparameterized the recursion in several other examples of this chapter (e.g., reverse, linear sum, binary sum). We saw a different approach to redeﬁning a recursion in our good ﬁbonacci implementation, by intentionally strengthening the expectation of what is returned (in that case, returning a pair of numbers rather than a single number).

4.6 Eliminating Tail Recursion

The main beneﬁt of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems. By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efﬁcient.

However, the usefulness of recursion comes at a modest cost. In particular, the Python interpreter must maintain activation records that keep track of the state of each nested call. When computer memory is at a premium, it is useful in some cases to be able to derive nonrecursive algorithms from recursive ones.

In general, we can use the stack data structure, which we will introduce in Section 6.1, to convert a recursive algorithm into a nonrecursive algorithm by man- aging the nesting of the recursive structure ourselves, rather than relying on the interpreter to do so. Although this only shifts the memory usage from the interpreter to our stack, we may be able to reduce the memory usage by storing only the minimal information necessary.

Even better, some forms of recursion can be eliminated without any use of axillary memory. A notable such form is known as tail recursion. A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion. By necessity, a tail recursion must be a linear recursion (since there is no way to make a second recursive call if you must immediately return the result of the ﬁrst).

Of the recursive functions demonstrated in this chapter, the binary search function of Code Fragment 4.3 and the reverse function of Code Fragment 4.10 are examples of tail recursion. Several others of our linear recursions are almost like tail recursion, but not technically so. For example, our factorial function of Code Fragment 4.1 is not a tail recursion. It concludes with the command:

return n factorial(n−1)

This is not a tail recursion because an additional multiplication is performed after the recursive call is completed. For similar reasons, the linear sum function of Code Fragment 4.9 and the good ﬁbonacci function of Code Fragment 4.7 fail to be tail recursions.

Any tail recursion can be reimplemented nonrecursively by enclosing the body in a loop for repetition, and replacing a recursive call with new parameters by a reassignment of the existing parameters to those values. As a tangible example, our binary search function can be reimplemented as shown in Code Fragment 4.15. We initialize variables low and high, just prior to our while loop, to represent the full extent of the sequence. Then, during each pass of the loop, we either ﬁnd 4.6 Eliminating Tail Recursion

The main beneﬁt of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems. By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efﬁcient.

However, the usefulness of recursion comes at a modest cost. In particular, the Python interpreter must maintain activation records that keep track of the state of each nested call. When computer memory is at a premium, it is useful in some cases to be able to derive nonrecursive algorithms from recursive ones.

In general, we can use the stack data structure, which we will introduce in Section 6.1, to convert a recursive algorithm into a nonrecursive algorithm by man- aging the nesting of the recursive structure ourselves, rather than relying on the interpreter to do so. Although this only shifts the memory usage from the interpreter to our stack, we may be able to reduce the memory usage by storing only the minimal information necessary.

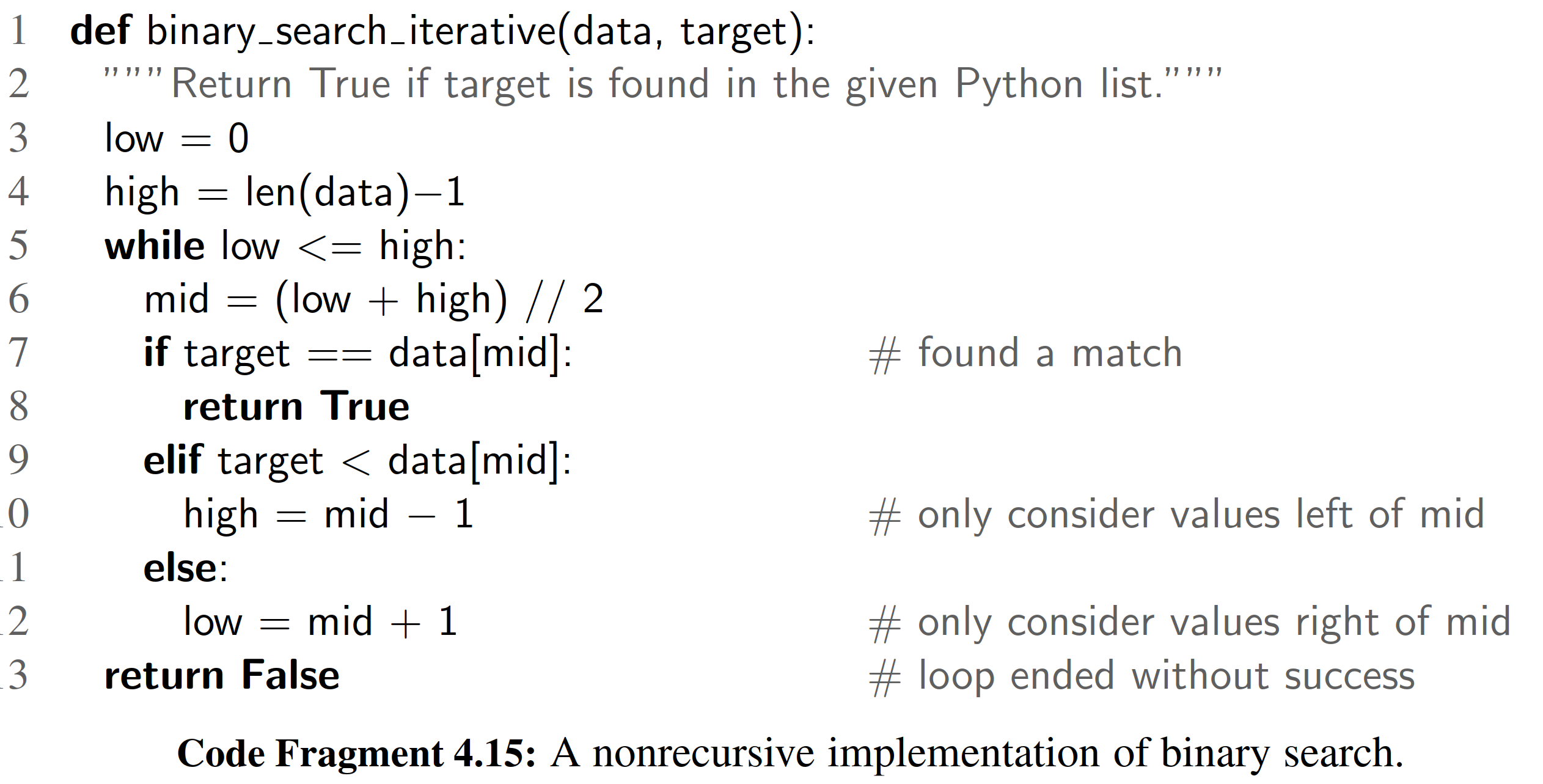
Even better, some forms of recursion can be eliminated without any use of axillary memory. A notable such form is known as tail recursion. A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion. By necessity, a tail recursion must be a linear recursion (since there is no way to make a second recursive call if you must immediately return the result of the ﬁrst).

Of the recursive functions demonstrated in this chapter, the binary search function of Code Fragment 4.3 and the reverse function of Code Fragment 4.10 are examples of tail recursion. Several others of our linear recursions are almost like tail recursion, but not technically so. For example, our factorial function of Code Fragment 4.1 is not a tail recursion. It concludes with the command:

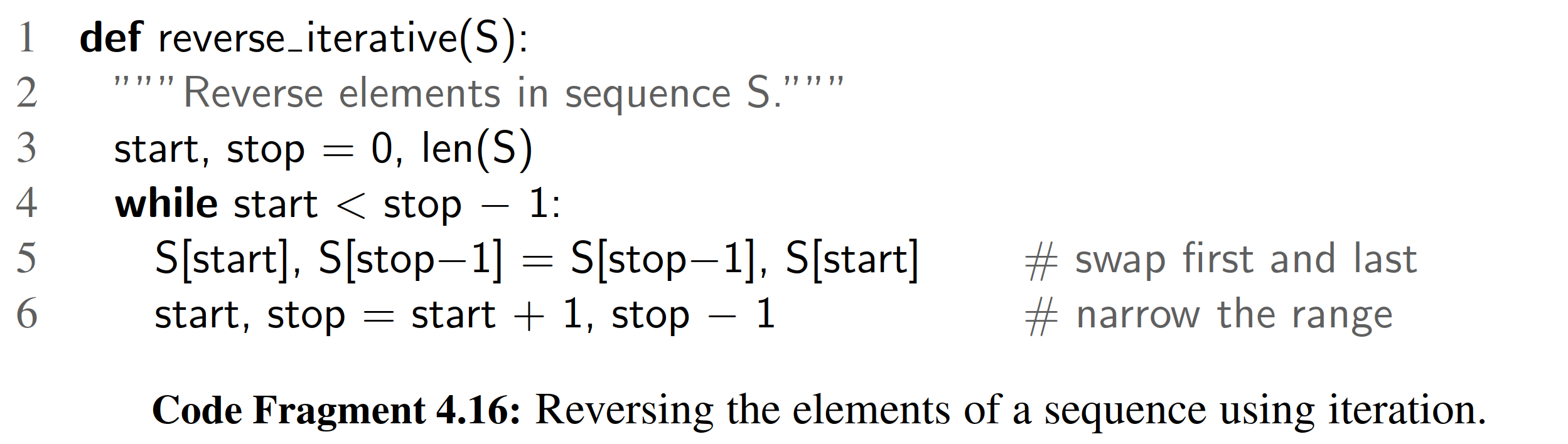
return n factorial(n−1)

This is not a tail recursion because an additional multiplication is performed after the recursive call is completed. For similar reasons, the linear sum function of Code Fragment 4.9 and the good ﬁbonacci function of Code Fragment 4.7 fail to be tail recursions.

Any tail recursion can be reimplemented nonrecursively by enclosing the body in a loop for repetition, and replacing a recursive call with new parameters by a reassignment of the existing parameters to those values. As a tangible example, our binary search function can be reimplemented as shown in Code Fragment 4.15. We initialize variables low and high, just prior to our while loop, to represent the full extent of the sequence. Then, during each pass of the loop, we either ﬁnd the target, or we narrow the range of the candidate subsequence. Where we made the recursive call binary search(data, target, low, mid −1) in the original version, we simply replace high = mid − 1 in our new version and then continue to the next iteration of the loop. Our original base case condition of low > high has simply been replaced by the opposite loop condition while low <= high. In our new implementation, we return False to designate a failed search if the while loop ends (that is, without having ever returned True from within).



We can similarly develop a nonrecursive implementation (Code Fragment 4.16) of the original recursive reverse method of Code Fragment 4.10.



In this new version, we update the values start and stop during each pass of the loop, exiting once we reach the case of having one or less elements in that range.

Many other linear recursions can be expressed quite efﬁciently with iteration, even if they were not formally tail recursions. For example, there are trivial non- recursive implementations for computing factorials, summing elements of a sequence, or computing Fibonacci numbers efﬁciently. In fact, our implementation of a Fibonacci generator, from Section 1.8, produces each subsequent value in O(1) time, and thus takes O(n) time to generate the nth entry in the series.

4.7 Exercises

**END**

# 六、实验体会

在本章的学习中，我们重点讨论了递归算法的实现，讲了几个重点例子。其中包括English Ruler的复杂递归实例。递归的核心在内存的铺展上。明白了这一点，就可以很快画出递归展开的踪迹图，然后就可以理解这个算法。

递归是分治策略的实现基础。当然，它包含着分割的思想，将大的问题简单化，当简单到一定程度的时候，就可以直接解决了。

# 七、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python

[2] 数据结构与算法分析：C语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社

[3] 算法导论（原书第三版），（美）科尔曼（Cormen，T.H.）等；殷建平等译. 北京：机械工业出版社