**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**：算法分析实验 | **学号**：20151910042 | **上机实践日期**：2017-04-10 |
| **上机实践编号**：No.03 | **组号**： | **上机实践时间**：上午3、4节 |

# 实验目的

1. 熟悉算法分析的基本概念与方法

# 二、实验内容

1. 实验研究法

任取一些典型算法, 进行Python实现，模仿3.1研究其运行时间，分析算法效率。

2. 渐近分析法

查阅Python文献，绘制图3.4，体会渐近分析法中经常遇到的几种函数的增长率。

# 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；

MATLAB R2017b。

# 四、实验记录与实验结果分析

1题

对典型算法的运行时间研究，要求用Python进行实现。

**Solution**：

由于目前接触的算法还不算很多，所以这里仅仅考虑不同的排序算法。典型的排序算法有冒泡排序（Bubble Sort）、选择排序（Selection Sort）、快速排序（Quick Sort）。我们这里选择这三种排序方法进行Python实现，然后运行，分析各自的运行时间，进而分析算法效率。这里采取的是实验分析法。

冒泡排序（Bubble Sort）算法解释：

冒泡排序采用的是循环方法。通过比较、交换，不断对原数组进行变化，最终得到有序的数组。

C:\Users\newton\AppData\Local\Microsoft\Windows\INetCache\Content.Word\绘图1.emf

Figure 1

在Figure 1中，我们假设有8个无序的数字，现在我们需要用冒泡法进行由小到大排序。先定义一个指向，如带尾缀的箭头，它指向假定的最小值，然后建立另一个指向，它从该假想值之后的位置开始，遍历剩下的所有元素，如果遇到了比假想值小的元素，就互换假想值与该元素，然后认定该元素为新的假想值。如此一来，经过第一次循环，最左侧就是最小值。然后把带尾缀的箭头指向该表的第二个元素，将该算法运行于除最小值之外的所有元素上。最终尾缀箭头指向该数组的倒数第二个元素，这个时候再与最后一个元素相比，算法完成。

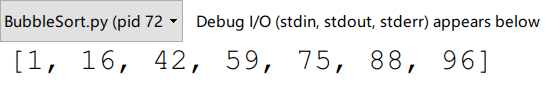
从上面的叙述中我们可以看到，当小箭头遍历完了剩余元素之后，才移动尾缀箭头，而且尾缀箭头的终点在倒数第二个元素位置上。而小箭头的起点都在对应的尾缀箭头的下一个位置上。由此写出Python代码。

程序代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # filename: BubbleSort  **def** BubbleSort**(**L**):**  **for** i **in** range**(**0**,**len**(**L**)-**2**):**  **for** j **in** range**(**i**,**len**(**L**)-**1**):**  **if** L**[**i**]** **>** L**[**j**]:**  L**[**i**],**L**[**j**]** **=** L**[**j**],**L**[**i**]**  **return** L  a **=** **[**1**,**96**,**88**,**75**,**42**,**16**,**59**]**  **print(**BubbleSort**(**a**))** |

程序代码 1

运行结果



运行结果 1

代码分析：

**对冒泡排序的复杂度的分析**。如果我们将数组按照递增顺序排列，那么显然，最坏的情况就是，输入的数组是按照递减顺序输入的，即，每次选中一个假定数值，在对该假定数值之后的所有数据进行遍历尝试的时候，由于剩余的数组部分都是去除上一次循环所剔除的那个数值之后，剩余数组元素的相对原顺序，所以剩余数组保留了递减顺序。这就导致每一次的遍历尝试都是完整遍历。而完整遍历的结果就是每一个判断语句都是为True，每一次都需要互换。这就导致了复杂度很高乃至是最高。一次判断、三次交换（虽然在Python中是一个语句，不过实际上还是如此进行，尤其是当元素不是数字，而是其他对象的时候），然后缩小规模（减1）继续操作。故假设输入规模为，那么第一次需要常量级操作次，之后第二次需要常量级操作次，……，一直到最终4次，那么总的常量级操作就是

，

总得来看，这是一个平方量级的最坏复杂度。

那么最好的情形，就是输入一个递增顺序的数组。这个时候，每一个判断都是假False，很显然，这种清醒只是避免了三次交换，所以就是上式去掉乘数4而已，也就是，也是一个量级的操作。所以，冒泡排序的复杂度，总是平方量级。即



值得指出的是，就算是所有的数组元素都一样大，那么这些判断也是不可以省略的，仍然是最小量级

快速排序（Quick Sort）：

对于快速排序而言，它的实现基础是分治策略，首先判断一次，然后进行数组分段，对分得的两段进行递归。当然，只要遇到了递归，就需要多加注意，很有可能因为递归使得内存占用很大，最终导致系统崩溃。在Python中，系统把递归深度限制在了900层左右，防止因为内存铺张影响总体性能。



Figure 2

在Figure 2中我们可以看到快速排序的行为方式。白圈就是我们选定的一个基，在它的基础上，我们进行左右分类，分为比它小的，比它大的，与它相等的（包括它自己），然后把这个算法递归性地应用的两类不等的数组中。当数组元素为1，结束划分。最后把这些合并起来，就是最后的有序数组。这个过程有很明显的递归性。

**对冒泡排序的复杂度的分析**。首先抛开复杂度分析，单纯对递归的过程进行内存占用分析。

假设我们输入了十万个递减排列的整数，要将它排列成递增顺序。首先调用sys模块的getsizeof()方法，可以看到一个不算很大的整数（100000以内）的内存占用是28个字节，即28Bytes，那么十万个整数就是2734KB，即2.67MB。这在一般的计算机上还是很容易得到的，毕竟是大内存机器时代。然而，在我们的分组中，只有首元素被分配到Middle数组中，剩下的元素都比首元素小，所以都在Left数组中（注意，这是一个新的数组，用append操作一个个扩充进去的，几乎是复制了一遍内存中的原数组），而Right数组为空。接下来对Left数组进行快速排序的递归调用。整个第一步，仅仅是容量减1，变成了99999个数值。就这样一直进行下去，直到数组容量变为1。很遗憾，这个减少量在这种情形下实在太过于微不足道，所以，即使在进行了1000次递归之后，数组还有99000个需要排列的，而这个时候，内存已经用了接近2.6GB，简直恐怖。难以想象以后会怎么样。

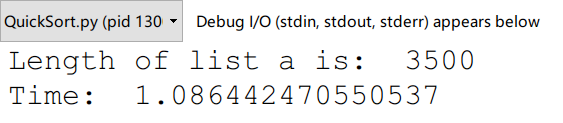
这里引出了我之前的操作与之后操作的不同。之前我随机输了10个数字进去，然后在编辑器里面进行全选、拷贝、粘贴，最后扩充到了十万，这样一来数据就比较均匀，Left与Right是相似的大小。再次分割，也几乎是对半分，这样的结果直接导致数组大小不停除以二，很快就变到1，从下面的这个程序中我们可以看到，对于一种最坏情形，这个快速排序几乎是非常愚蠢的行为，仅仅3500个整数，就耗时一秒钟。

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  26  27  28  29  30 | # Quick Sort for a list  **from** time **import** time  **import** sys  sys**.**setrecursionlimit**(**1000000**)**  **def** QuickSort**(**L**):**  L\_Left **=** **[]**  L\_Right **=** **[]**  L\_Middle **=** **[]**  **if** len**(**L**)** **<=** 1**:**  **return** L  **else:**  **for** i **in** L**:**  **if** i **>** L**[**0**]:**  L\_Left**.**append**(**i**)**  **elif** i **<** L**[**0**]:**  L\_Right**.**append**(**i**)**  **else:**  L\_Middle**.**append**(**i**)**  L\_Left **=** QuickSort**(**L\_Left**)**  L\_Right **=** QuickSort**(**L\_Right**)**  **return** L\_Left **+** L\_Middle **+** L\_Right  A **=** list**(**range**(**3500**))**  **print(**"Length of list a is: "**,**len**(**A**))**  begin **=** time**()**  QuickSort**(**A**)**  end **=** time**()**  **print(**"Time: "**,**end **-** begin**)** |

程序代码 2

运行结果



运行结果 2

在实验体会里，我对代码进行了修改，针对枢纽元的选择进行了重新的审视，因为要考虑的情形很多都不是随机均匀的。之前的十万个数字，实在是一个错误的例子。

**总结**

基于上面的两种算法，我们用十万个数字对他们进行排序速度测试。由于把数据写入文件再调用，会造成硬盘读写的干扰，为了最大化降低干扰，我们直接把数据写在代码里，运行的时候相当于直接调用内存中的文件，速度会快很多，干扰也会小很多。

在导入了time模块之后，Python也可以像C语言编译一样，查看运行时间。然而，遗憾的是，Python在进行排序的时候，会返回不了。当输入量是10, 000左右的时候，快速排序会很好地执行

2题

查阅Python文献，绘制图3.4，体会渐近分析法中经常遇到的几种函数的增长率。

**Solution**:

To sum up, Table 3.1 shows, in order, each of the seven commo functions used in algorithm analysis.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **constant** | **logarithm** | **Linear** | **n-log-n** | **quadratic** | **cubic** | **exponential** |
| 1 |  |  |  |  |  |  |

**Table 3.1: Classes of functions. Here we assume that  is a constant.**

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or *n*-log-*n* time. Algorithm with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 3.4.



|  |
| --- |
| **Figure 3.4：**Growth rates for the seven fundamental functions used in algorithm used in algorithm analysis. We use base a = 2 for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart. |

上面的图标是用MATLAB做出的。MATLAB代码如下：

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 | %% 输入自变量n的数量级  n **=** logspace**(**0**,**15**,**16**);**  %% 定义函数  f\_constant **=** n **-** n **+** 10e2**;**  f\_logarithm **=** log2**(**n**);**  f\_linear **=** n**;**  f\_nlogn **=** n **.\*** log2**(**n**);**  f\_quadratic **=** n**.^**2**;**  f\_cubic **=** n**.^**3**;**  f\_exponentional **=** 2.**^**n**;**  %% 作图  loglog**(**n**,**f\_constant**,**'-d'**);**  hold on**;**  loglog**(**n**,**f\_logarithm**,**'-o'**);**  loglog**(**n**,**f\_linear**,**'-s'**);**  loglog**(**n**,**f\_nlogn**,**'-p'**);**  loglog**(**n**,**f\_quadratic**,**'-h'**);**  loglog**(**n**,**f\_cubic**,**'-v'**);**  loglog**(**n**,**f\_exponentional**,**'-<'**);**  axis**([**1 10e15 1 10e44**]);**  xlabel**(**'n'**);**  ylabel**(**'f(n)'**);**  hold off |

程序代码 3

# 五、教材翻译

**Translation**

**Chapter 3 Algorithm Analysis**

＊第三章 算法分析

In a classic story, the famous mathematician Archimedes was asked to deter- mine if a golden crown commissioned by the king was indeed pure gold, and not part silver, as an informant had claimed. Archimedes discovered a way to perform this analysis while stepping into a bath. He noted that water spilled out of the bath in proportion to the amount of him that went in. Realizing the implications of this fact, he immediately got out of the bath and ran naked through the city shouting, “Eureka, eureka!” for he had discovered an analysis tool (displacement), which, when combined with a simple scale, could determine if the king’s new crown was good or not. That is, Archimedes could dip the crown and an equal-weight amount of gold into a bowl of water to see if they both displaced the same amount. This discovery was unfortunate for the goldsmith, however, for when Archimedes did his analysis, the crown displaced more water than an equal-weight lump of pure gold, indicating that the crown was not, in fact, pure gold.

＊在一个经典故事中，著名的数学家阿基米德被要求确定国王金冠是否是由纯金打造，而不是掺杂了部分白银。阿基米德在泡澡时发现了一种分析的方法。他指出，他的身体浸入澡盆水面越多，溢出的水量越大，而且两者严格相等。意识到这个事实，他马上脱身洗澡，赤身裸体穿过这个城市大喊：“尤里卡，尤里卡！”因为他发现了一个分析方法（排水量），当与排水容量进行简单结合时，就可以确定王冠是否是纯金的。也就是说，阿基米德可以将王冠和等量的纯金分别浸入水中，看看排水量是否相等。这个发现对于金匠来说是不幸的，当阿基米德做了他的分析时，皇冠比同等块纯金溢出了更多的水，这表明皇冠实际上并不是纯金。

In this book, we are interested in the design of “good” data structures and algorithms. Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some task in a ﬁnite amount of time. These concepts are central to computing, but to be able to classify some data structures and algorithms as “good,” we must have precise ways of analyzing them.

＊在本书中，我们关心的是“好的”数据结构和算法。简单地说，数据结构是一种组织和访问数据的系统方式，算法是在有限的时间内执行一些任务的一步一步的过程。这些概念是计算的核心，但为了能够将一些数据结构和算法定义为“好”，我们必须有精确的分析方法。

The primary analysis tool we will use in this book involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of “goodness,” since time is a precious resource—computer solutions should run as fast as possible. In general, the running time of an algorithm or data structure operation increases with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by the hardware environment (e.g., the processor, clock rate, memory, disk) and software environment (e.g., the operating system, programming language) in which the algorithm is implemented and executed. All other factors being equal, the running time of the same algorithm on the same input data will be smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation. We begin this chapter by discussing tools for performing experimental studies, yet also limitations to the use of experiments as a primary means for evaluating algorithm efﬁciency.

＊在本书中使用的主要分析工具包括：算法运行时间、数据结构的操作方法、内存的占用。运行时间是判别是否“良好”的自然尺度，因为时间是宝贵的资源，计算机解决问题应尽可能快。通常，算法或数据结构操作的运行时间随着输入大小而增加，尽管对于相同大小的不同输入，运行时间也会不同。此外，运行时间受实施和执行算法的硬件环境（例如，处理器，时钟速率，存储器，磁盘）和软件环境（例如，操作系统，编程语言）的影响。所有其他因素相同，如果计算机具有比较快的处理器，或者如果是被编译为机器码、而不是在解释性程序中完成了实现，则相同输入数据上相同算法的运行时间将更短。我们从实验研究开始本章，但也将尽量避免使用实验方法作为评估算法效率的主要手段。

Focusing on running time as a primary measure of goodness requires that we be able to use a few mathematical tools. In spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input. We are interested in characterizing an algorithm’s running time as a function of the input size. But what is the proper way of measuring it? In this chapter, we “roll up our sleeves” and develop a mathematical way of analyzing algorithms.

＊为了将运行时间作为主要衡量标准的要求，我们需要使用几种数学工具。忽略来自不同环境因素的变化，我们重点关注算法的运行时间与其输入的大小之间的关系。我们把算法的运行时间视为输入量的函数。但是什么是衡量它的正确方法？在本章中，我们“撸起袖子”，开发一套分析算法的数学方法。

3.1 Experimental Studies

＊实验研究

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the time spent during each execution. A simple approach for doing this in Python is by using the time function of the time module. This function reports the number of seconds, or fractions thereof, that have elapsed since a benchmark time known as the epoch. The choice of the epoch is not signiﬁcant to our goal, as we can determine the *elapsed* time by recording the time just before the algorithm and the time just after the algorithm, and computing their difference, as follows:

＊如果一个算法已被实现，我们可以通过各种输入测试来记录其运行时间。在Python中执行此操作的简单方法是使用time模块的time()函数。此函数此时时刻的秒数或者分数，数据来源是系统时间。时刻的选择对我们的目标并不重要，因为我们可以通过记录算法运行之前的时间和算法结束之后的时间来确定经过的时间，并计算它们的差，如下所示：

from time import time

start time = time( ) # record the starting time

run algorithm

end time = time( ) # record the ending time

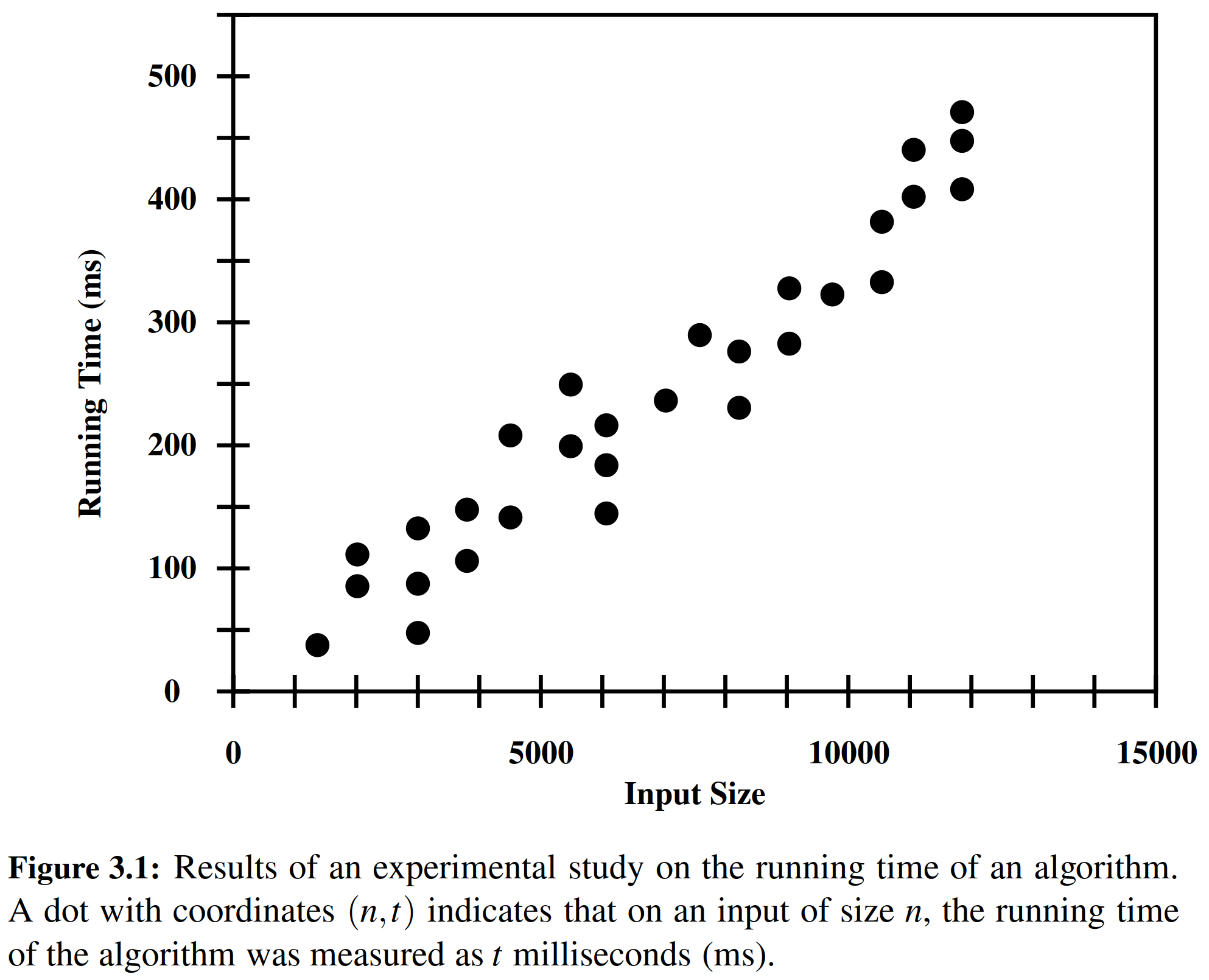
elapsed = end time − start time # compute the elapsed time

We will demonstrate use of this approach, in Chapter 5, to gather experimental data on the efﬁciency of Python’s list class. An elapsed time measured in this fashion is a decent reﬂection of the algorithm efﬁciency, but it is by no means perfect. The time function measures relative to what is known as the “wall clock.” Because many processes share use of a computer’s ***central processing unit*** (or ***CPU***), the elapsed time will depend on what other processes are running on the computer when the test is performed. A fairer metric is the number of CPU cycles that are used by the algorithm. This can be determined using the clock function of the time module, but even this measure might not be consistent if repeating the identical algorithm on the identical input, and its granularity will depend upon the computer system. Python includes a more advanced module, named timeit, to help automate such evaluations with repetition to account for such variance among trials.

＊我们将在第5章中演示使用这种方法来收集Python列表类的效率数据。以这种方式测量的经过时间是算法效率的一个有效的反映，但并不完美。时间功能依靠“系统时间”来测量。由于许多进程共享计算机的中央处理单元（CPU），所以经过的时间将取决于计算机上运行的其他进程。更公平的度量是算法使用的CPU周期数。这可以使用time模块的clock函数来确定，但即使在相同的输入上重复相同的算法，即使这种措施可能不一致，其差异将取决于计算机系统。Python涵盖一个更高级的模块，名为timeit，以自动化地重复这样的评估，以考虑试验之间的这种差异。

Because we are interested in the general dependence of running time on the size and structure of the input, we should perform independent experiments on many different test inputs of various sizes. We can then visualize the results by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size, n, and y-coordinate equal to the running time, *t*. Figure 3.1 displays such hypothetical data. This visualization may provide some intuition regarding the relationship between problem size and execution time for the algorithm. This may lead to a statistical analysis that seeks to ﬁt the best function of the input size to the experimental data. To be meaningful, this analysis requires that we choose good sample inputs and test enough of them to be able to make sound statistical claims about the algorithm’s running time.

＊我们致力于研究运行时间关于数据规模和数据结构的关系，所以我们应该对许多不同规模的不同输入进行独立的实验。然后，我们可以画一个二维坐标图，横轴x坐标表示输入大小，纵轴y坐标表示运行时间。图3.1显示了这样的关系。这种可视化操作可以提供关于算法的输入和执行时间之间的关系的一些直觉的形成。可以通过统计分析，确定与输入规模所匹配的最佳函数。为了有意义，这种分析要求我们选择良好的样本输入，并对此进行足够的测试，以便对算法的运行时间进行有效统计。



**Challenges of Experimental Analysis**

While experimental studies of running times are valuable, especially when ﬁne- tuning production-quality code, there are three major limitations to their use for algorithm analysis:

* Experimental running times of two algorithms are difﬁcult to directly com- pare unless the experiments are performed in the same hardware and software environments.
* Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
* An algorithm must be fully implemented in order to execute it to study its running time experimentally.

This last requirement is the most serious drawback to the use of experimental stud- ies. At early stages of design, when considering a choice of data structures or algorithms, it would be foolish to spend a signiﬁcant amount of time implementing an approach that could easily be deemed inferior by a higher-level analysis.

3.1.1 Moving Beyond Experimental Analysis

＊

Our goal is to develop an approach to analyzing the efﬁciency of algorithms that:

1. Allows us to evaluate the relative efﬁciency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without need for implementation.
3. Takes into account all possible inputs.

**Counting Primitive Operations**

To analyze the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm (either in the form of an actual code fragment, or language-independent pseudo-code). We deﬁne a set of primitive operations such as the following:

Assigning an identiﬁer to an object

* Determining the object associated with an identiﬁer
* Performing an arithmetic operation (for example, adding two numbers)
* Comparing two numbers
* Accessing a single element of a Python list by index
* Calling a function (excluding operations executed within the function)
* Returning from a function.

Formally, a primitive operation corresponds to a low-level instruction with an exe- cution time that is constant. Ideally, this might be the type of basic operation that is executed by the hardware, although many of our primitive operations may be trans- lated to a small number of instructions. Instead of trying to determine the speciﬁc execution time of each primitive operation, we will simply count how many prim- itive operations are executed, and use this number t as a measure of the running time of the algorithm.

This operation count will correlate to an actual running time in a speciﬁc com- puter, for each primitive operation corresponds to a constant number of instructions, and there are only a ﬁxed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

**Measuring Operations as a Function of Input Size**

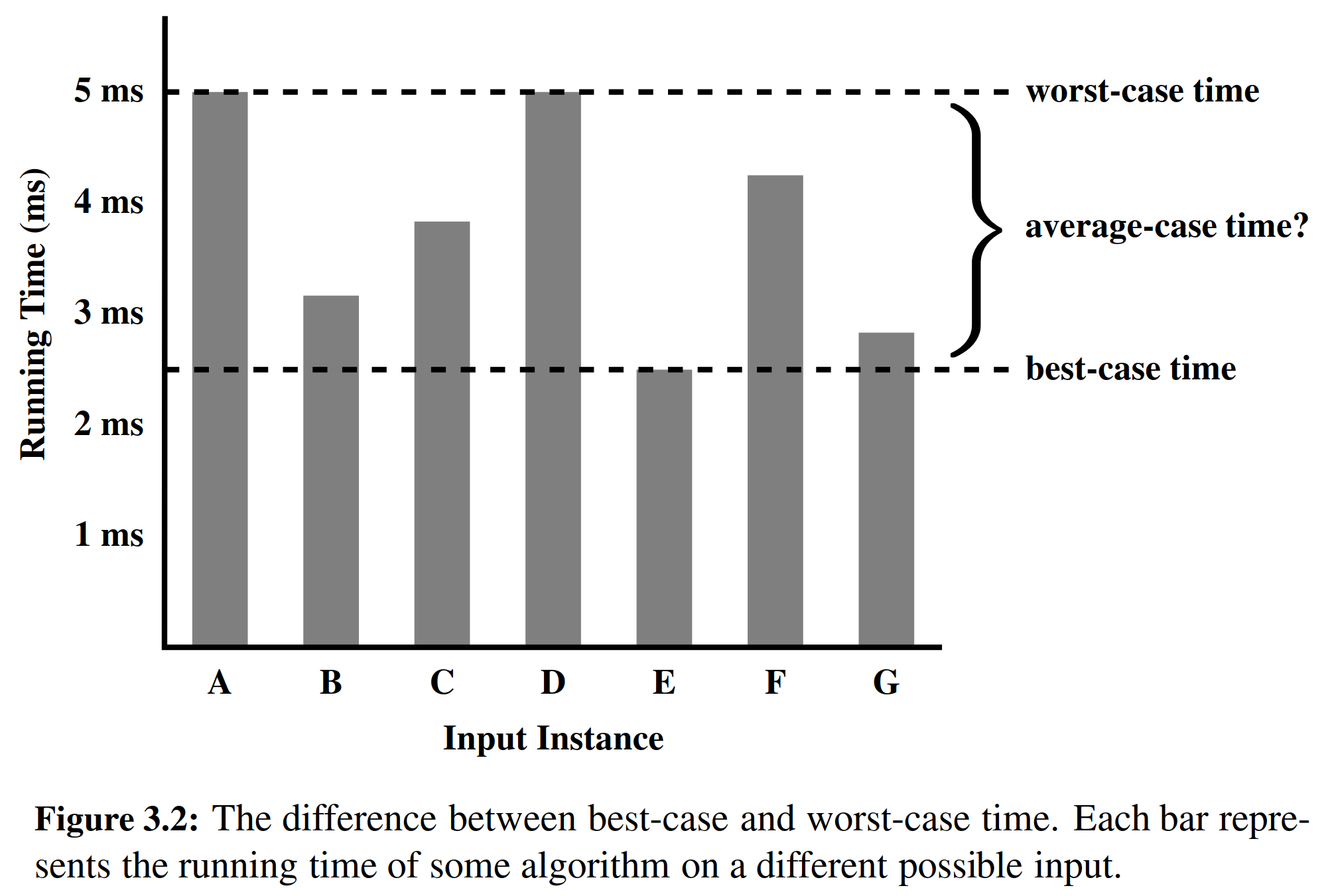
To capture the order of growth of an algorithm’s running time, we will associate, with each algorithm, a function *f* (*n*) that characterizes the number of primitive operations that are performed as a function of the input size n. Section 3.2 will in- troduce the seven most common functions that arise, and Section 3.3 will introduce a mathematical framework for comparing functions to each other.

**Focusing on the Worst-Case Input**

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Unfortunately, such an average-case analysis is typically quite challenging. It requires us to deﬁne a probability distribution on the set of inputs, which is often a difﬁcult task. Figure 3.2 schematically shows how, depending on the input distri- bution, the running time of an algorithm can be anywhere between the worst-case time and the best-case time. For example, what if inputs are really only of types “A” or “D”?

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. Therefore, for the remainder of this book, unless we specify otherwise, we will characterize running times in terms of the worst case, asa func- tion of the input size, n, of the algorithm.

Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple. Also, this approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input. That is, designing for the worst case leads to stronger algorithmic “muscles,” much like a track star who always practices by running up an incline.



3.2 The Seven Functions Used in This Book

In this section, we brieﬂy discuss the seven most important functions used in the analysis of algorithms. We will use only these seven simple functions for almost all the analysis we do in this book. In fact, a section that uses a function other than one of these seven will be marked with a star (\*) to indicate that it is optional. In addition to these seven fundamental functions, Appendix B contains a list of other useful mathematical facts that apply in the analysis of data structures and algorithms.

**The Constant Function**

The simplest function we can think of is the constant function. This is the function,

*f* (*n*)= *c*,

for some ﬁxed constant c, such as *c* = 5, *c* = 27, or *c* = 210. That is, for any argument n, the constant function *f* (*n*) assigns the value *c*. In other words, it does not matter what the value of n is; *f* (*n*) will always be equal to the constant value *c*.

Because we are most interested in integer functions, the most fundamental constant function is *g*(*n*)= 1, and this is the typical constant function we use in this book. Note that any other constant function, *f* (*n*)= *c*, can be written as a constant

*c* times *g*(*n*). That is, *f* (*n*)= *cg*(*n*) in this case.

As simple as it is, the constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

The Logarithm Function

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the ***logarithm function***, f (n)= logb n, for some constant b > 1. This function is deﬁned as follows:

*x* = log*b* *n* if and only if b*x* = *n*.

By deﬁnition, log*b* 1 = 0. The value *b* is known as the ***base*** of the logarithm.

The most common base for the logarithm function in computer science is 2, as computers store integers in binary, and because a common operation in many algorithms is to repeatedly divide an input in half. In fact, this base is so common that we will typically omit it from the notation when it is 2. That is, for us,

log *n* = log2 *n*.

We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base-10, not base-two.

Computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough for our pur- poses without calculus. In particular, we can easily compute the smallest integer greater than or equal to logb n (its so-called ***ceiling***, ). For positive integer, *n*, this value is equal to the number of times we can divide n by b before we get a number less than or equal to 1. For example, the evaluation of  is 3, because ((27/3)/3)/3 = 1. Likewise, is 3, because ((64/4)/4)/4 = 1, and  is 4, because (((12/2)/2)/2)/2 = 0.75 ≤ 1.

The following proposition describes several important identities that involve logarithms for any base greater than 1.

**Proposition 3.1 (Logarithm Rules)**: *Given real numbers* *a* > 0, *b* > 1, *c* > 0 *and* d > 1, *we have:*

1. 
2. 
3. 
4. 
5. 

By convention, the unparenthesized notation log nc denotes the value log(*nc*). We use a notational shorthand, logc n, to denote the quantity, (log *n*)*c*, in which the result of the logarithm is raised to a power.

The above identities can be derived from converse rules for exponentiation that we will present on page 121. We illustrate these identities with a few examples.

**Example 3.2**: We demonstrate below some interesting applications of the loga- rithm rules from Proposition 3.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).

* , *by rule 1*
* , *by rule 2*
* , *by rule 3*
* , *by rule 3*
* , *by rule 4*
* , *by rule 5*

*As a practical matter, we note that rule 4 gives us a way to compute the base-two logarithm on a calculator that has a base-10 logarithm button,* LOG*, for*

log2 n = LOG n / LOG 2.

**The Linear Function**

Another simple yet important function is the linear function,

f (n)= n.

That is, given an input value n, the linear function f assigns the value n itself.

This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of a sequence of size n will require n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer’s memory, because reading in the n objects already requires n operations.

**The *N-Log-N* Function**

The next function we discuss in this section is the *n-log-n* function,

f (n)= n log n,

that is, the function that assigns to an input n the value of n times the logarithm base-two of n. This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function; therefore, we would greatly prefer an algorithm with a running time that is proportional to n log n, than one with quadratic running time. We will see several important algorithms that exhibit a running time proportional to the n-log-n function. For example, the fastest possible algorithms for sorting n arbitrary values require time proportional to n log n.

**The Quadratic Function**

Another function that appears often in algorithm analysis is the quadratic function,

f (n)= n2.

That is, given an input value n, the function f assigns the product of n with itself (in other words, “n squared”).

The main reason why the quadratic function appears in the analysis of algo- rithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs n n = n2 operations.

Nested Loops and the Quadratic Function

The quadratic function can also arise in the context of nested loops where the ﬁrst iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is

1 + 2 + 3 + ···+ (n − 2)+ (n − 1)+ n.

In other words, this is the total number of operations that will be performed by the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop. This quantity also has an interesting history.

In 1787, a German schoolteacher decided to keep his 9- and 10-year-old pupils occupied by adding up the integers from 1 to 100. But almost immediately one of the children claimed to have the answer! The teacher was suspicious, for the student had only the answer on his slate. But the answer, 5050, was correct and the student, Carl Gauss, grew up to be one of the greatest mathematicians of his time. We presume that young Gauss used the following identity.

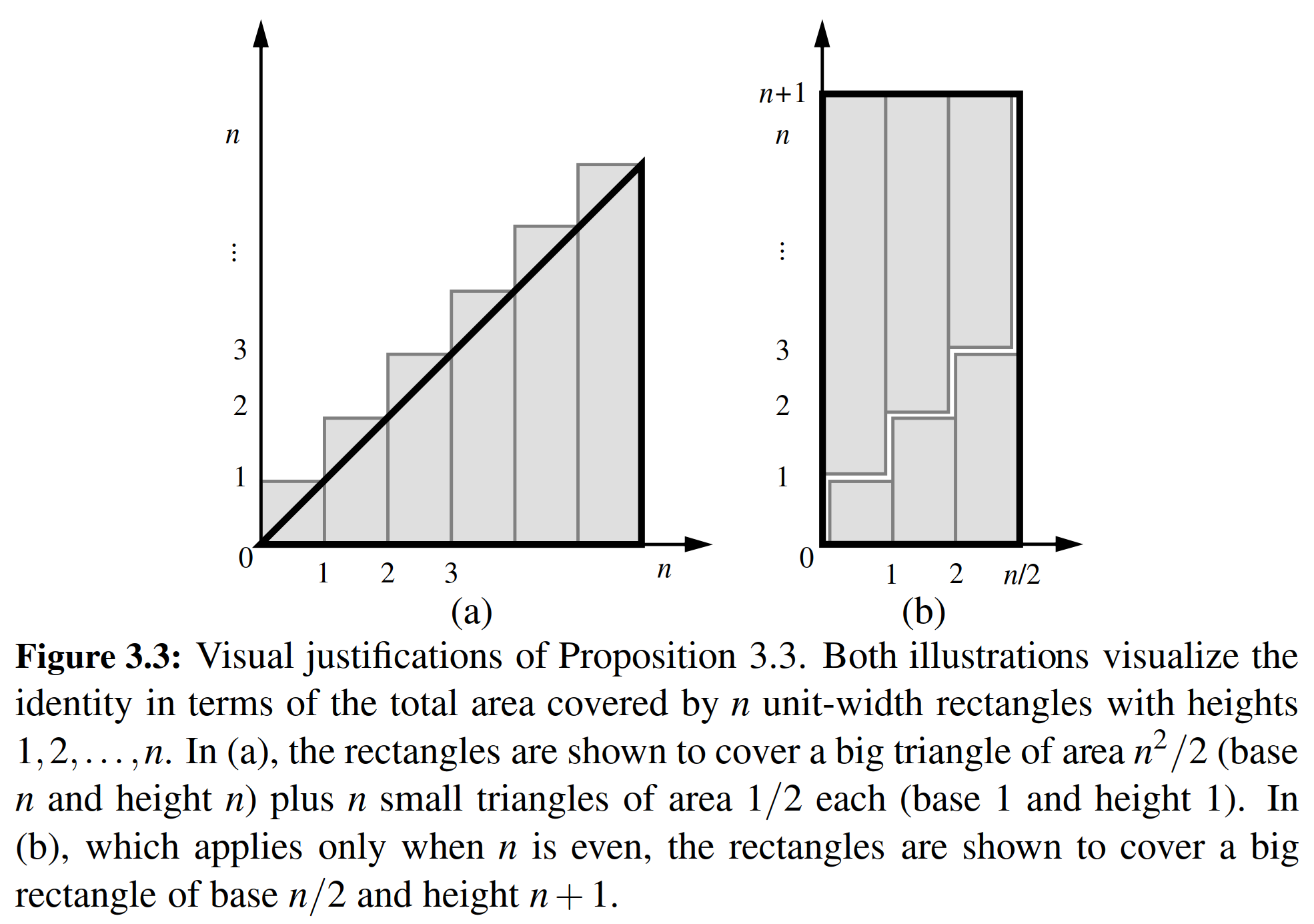
Proposition 3.3: For any integer n ≥ 1, we have:

1 + 2 + 3 + ···+ (n − 2)+ (n − 1)+ n =

n(n + 1)

2 .

We give two “visual” justiﬁcations of Proposition 3.3 in Figure 3.3



The lesson to be learned from Proposition 3.3 is that if we perform an algorithm with nested loops such that the operations in the inner loop increase by one each time, then the total number of operations is quadratic in the number of times, n, we perform the outer loop. To be fair, the number of operations is n2/2 + n/2, and so this is just over half the number of operations than an algorithm that uses n operations each time the inner loop is performed. But the order of growth is still quadratic in n.

The Cubic Function and Other Polynomials

Continuing our discussion of functions that are powers of the input, we consider the cubic function,

f (n)= n3,

which assigns to an input value n the product of n with itself three times. This func- tion appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Polynomials

Most of the functions we have listed so far can each be viewed as being part of a larger class of functions, the polynomials. A polynomial function has the form,

f (n)= a0 + a1n + a2n2 + a3n3 + ···+ adnd,

where a0, a1,..., ad are constants, called the coefﬁcients of the polynomial, and

ad 0. Integer d, which indicates the highest power in the polynomial, is called

the degree of the polynomial.

For example, the following functions are all polynomials:

• f (n)= 2 + 5n + n

• f (n)= 1 + n

• f (n)= 1

• f (n)= n

• f (n)= n

Therefore, we could argue that this book presents just four important functions used in algorithm analysis, but we will stick to saying that there are seven, since the con- stant, linear, and quadratic functions are too important to be lumped in with other polynomials. Running times that are polynomials with small degree are generally better than polynomial running times with larger degree.

Summations

A notation that appears again and again in the analysis of data structures and algo- rithms is the summation, which is deﬁned as follows:

b

 f (i)= f (a)+ f (a + 1)+ f (a + 2)+ + f (b),

i=a

where a and b are integers and a ≤ b. Summations arise in data structure and algo- rithm analysis because the running times of loops naturally give rise to summations.

Using a summation, we can rewrite the formula of Proposition 3.3 as

n

 i =

i=1

n(n + 1)

2 .

Likewise, we can write a polynomial f (n) of degree d with coefﬁcients a0,..., ad as

d

f (n)=  aini.

i=0

Thus, the summation notation gives us a shorthand way of expressing sums of in- creasing terms that have a regular structure.

The Exponential Function

Another function used in the analysis of algorithms is the exponential function,

f (n)= bn,

where b is a positive constant, called the base, and the argument n is the exponent. That is, function f (n) assigns to the input argument n the value obtained by mul- tiplying the base b by itself n times. As was the case with the logarithm function, the most common base for the exponential function in algorithm analysis is b = 2. For example, an integer word containing n bits can represent all the nonnegative integers less than 2n. If we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2n.

We sometimes have other exponents besides n, however; hence, it is useful for us to know a few handy rules for working with exponents. In particular, the following exponent rules are quite helpful.

Proposition 3.4 (Exponent Rules): Given positive integers a, b, and c, we have

1. (ba)c = bac

2. babc = ba+c

3. ba/bc = ba−c

For example, we have the following:

• 256 = 16 = (2 ) = 2 = 2 = 256 (Exponent Rule 1)

• 243 = 3 = 3 = 3 3 = 9 · 27 = 243 (Exponent Rule 2)

• 16 = 1024/64 = 2 /2 = 2 = 2 = 16 (Exponent Rule 3)

We can extend the exponential function to exponents that are fractions or real numbers and to negative exponents, as follows. Given a positive integer k, we de- ﬁne b1/k to be kth root of b, that is, the number r such that rk = b. For example, 251/2 = 5, since 52 = 25. Likewise, 271/3 = 3 and 161/4 = 2. This approach al- lows us to deﬁne any power whose exponent can be expressed as a fraction, for ba/c = (ba)1/c, by Exponent Rule 1. For example, 93/2 = (93)1/2 = 7291/2 = 27. Thus, ba/c is really just the cth root of the integral exponent ba.

We can further extend the exponential function to deﬁne bx for any real number x, by computing a series of numbers of the form ba/c for fractions a/c that get pro- gressively closer and closer to x. Any real number x can be approximated arbitrarily closely by a fraction a/c; hence, we can use the fraction a/c as the exponent of b to get arbitrarily close to bx. For example, the number 2 is well deﬁned. Finally,

given a negative exponent d, we deﬁne bd = 1/b−d , which corresponds to applying Exponent Rule 3 with a = 0 and c = −d. For example, 2−3 = 1/23 = 1/8.

Geometric Sums

Suppose we have a loop for which each iteration takes a multiplicative factor longer than the previous one. This loop can be analyzed using the following proposition. Proposition 3.5: For any integer n ≥ 0 and any real number a such that a > 0 and

a /= 1, consider the summation

n

 ai = 1 + a + a2 + + an

i=0

(remembering that a0 = 1 if a > 0). This summation is equal to

an+1 − 1

.

a − 1

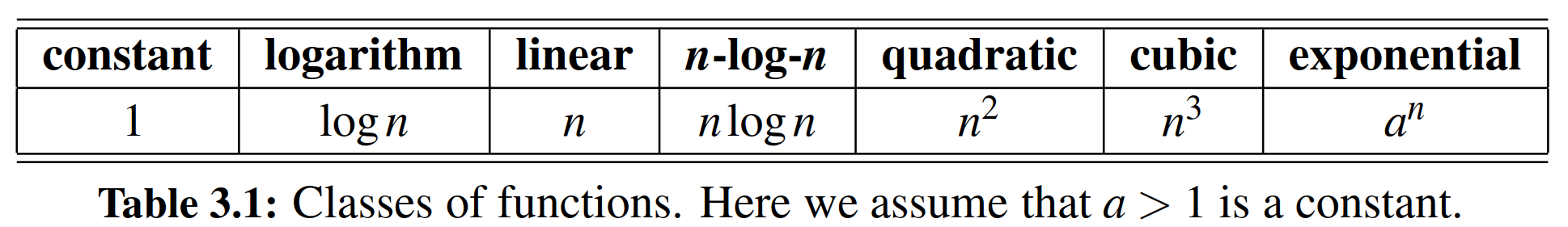
Summations as shown in Proposition 3.5 are called geometric summations, be- cause each term is geometrically larger than the previous one if a > 1. For example, everyone working in computing should know that

1 + 2 + 4 + 8 + ···+ 2n−1 = 2n − 1,

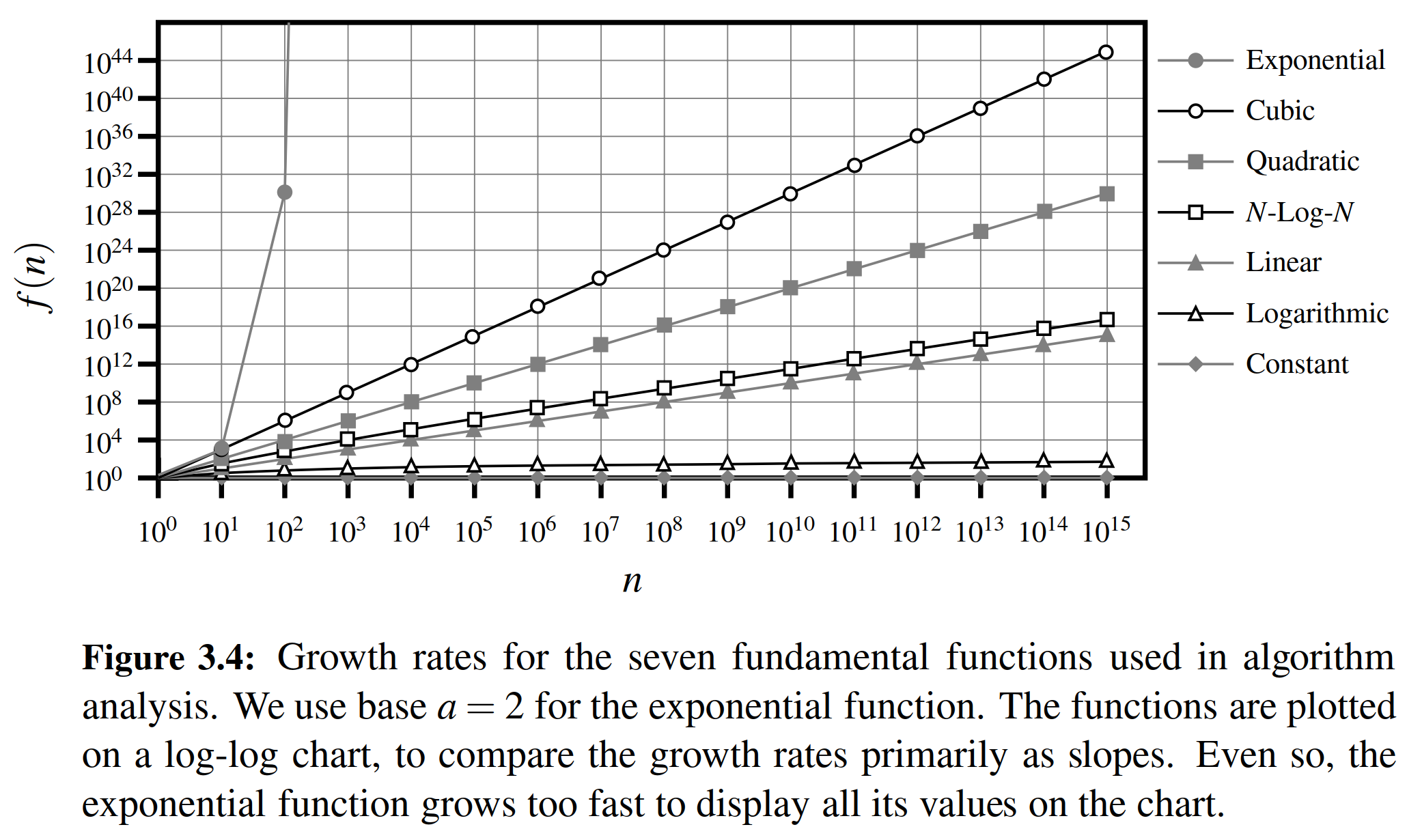
for this is the largest integer that can be represented in binary notation using n bits.

3.2.1 Comparing Growth Rates

To sum up, Table 3.1 shows, in order, each of the seven common functions used in algorithm analysis.



Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 3.4.



The Ceiling and Floor Functions

One additional comment concerning the functions above is in order. When dis- cussing logarithms, we noted that the value is generally not an integer, yet the running time of an algorithm is usually expressed by means of an integer quantity, such as the number of operations performed. Thus, the analysis of an algorithm may sometimes involve the use of the ﬂoor function and ceiling function, which are deﬁned respectively as follows:

• x囚= the largest integer less than or equal to x.

• 「x1 = the smallest integer greater than or equal to x.

3.3 Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a “big-picture” approach. For example, it is often enough just to know that the running time of an algorithm grows proportionally to n.

We analyze algorithms using a mathematical notation for functions that disre- gards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach re- ﬂects that each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations. Thus, we can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting bogged down in language-speciﬁc or hardware-speciﬁc analysis of the exact number of operations that execute on the computer.

As a tangible example, we revisit the goal of ﬁnding the largest element of a Python list; we ﬁrst used this example when introducing for loops on page 21 of Section 1.4.2. Code Fragment 3.1 presents a function named ﬁnd max for this task.

1 def ﬁnd max(data):

2 ”””Return the maximum element from a nonempty Python list.”””

3 biggest = data[0] # The initial value to beat

4 for val in data: # For each value:

5 if val > biggest # if it is greater than the best so far,

6 biggest = val # we have found a new best (so far)

7 return biggest # When loop ends, biggest is the max

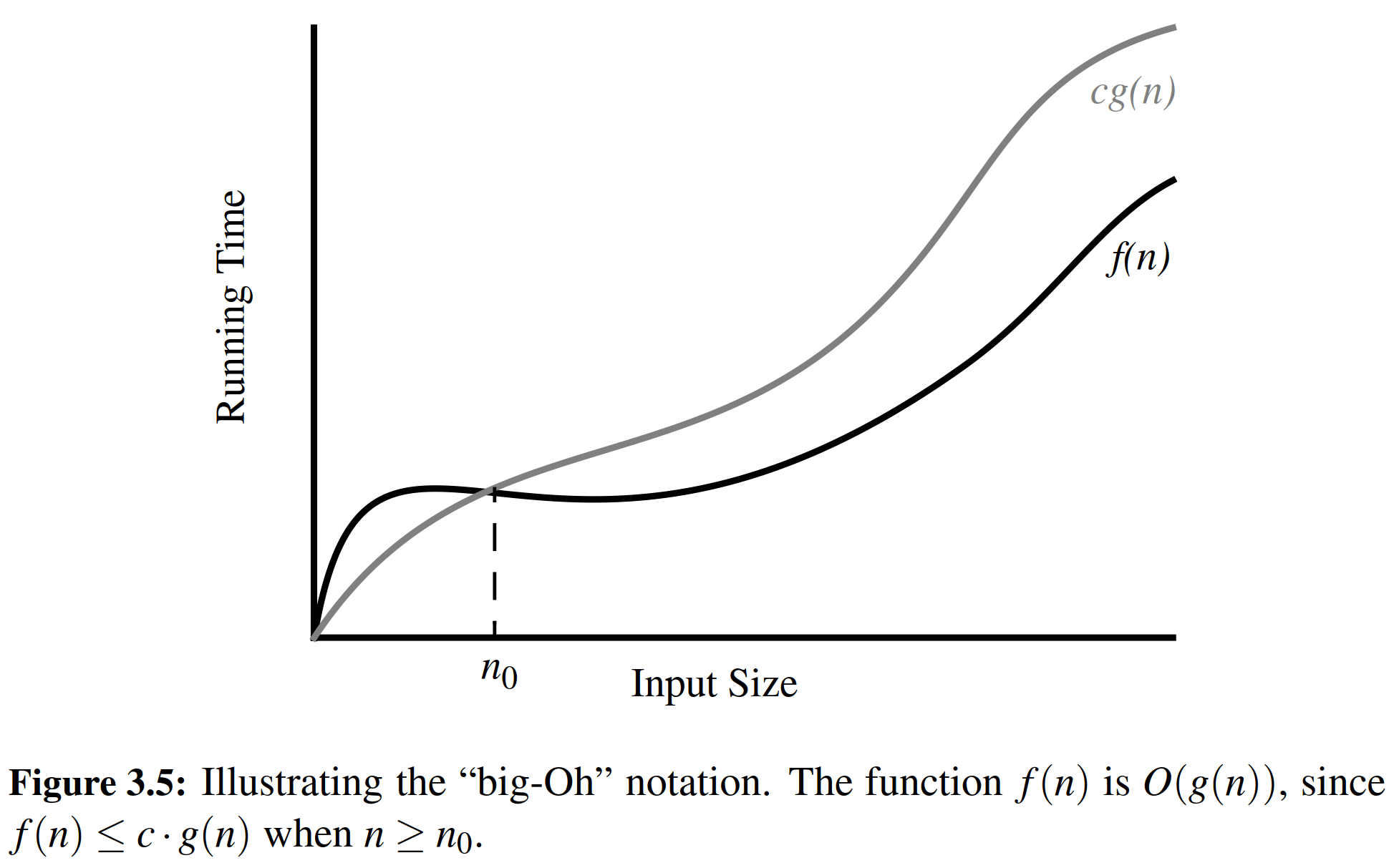
Code Fragment 3.1: A function that returns the maximum value of a Python list. This is a classic example of an algorithm with a running time that grows proportional to n, as the loop executes once for each data element, with some ﬁxed number of primitive operations executing for each pass. In the remainder of this section, we provide a framework to formalize this claim.

3.3.1 The “Big-Oh” Notation

Let f (n) and g(n) be functions mapping positive integers to positive real numbers. We say that f (n) is O(g(n)) if there is a real constant c > 0 and an integer constant n0 ≥ 1 such that

f (n) ≤ cg(n), for n ≥ n0.

This deﬁnition is often referred to as the “big-Oh” notation, for it is sometimes pro- nounced as “ f (n) is big-Oh of g(n).” Figure 3.5 illustrates the general deﬁnition.



Example 3.6: The function 8n + 5 is O(n).

Justiﬁcation: By the big-Oh deﬁnition, we need to ﬁnd a real constant c > 0 and an integer constant n0 ≥ 1 such that 8n + 5 ≤ cn for every integer n ≥ n0. It is easy to see that a possible choice is c = 9 and n0 = 5. Indeed, this is one of inﬁnitely many choices available because there is a trade-off between c and n0. For example, we could rely on constants c = 13 and n0 = 1.

The big-Oh notation allows us to say that a function f (n) is “less than or equal to” another function g(n) up to a constant factor and in the asymptotic sense as n grows toward inﬁnity. This ability comes from the fact that the deﬁnition uses “≤” to compare f (n) to a g(n) times a constant, c, for the asymptotic cases when n ≥ n0. However, it is considered poor taste to say “ f (n) ≤ O(g(n)),” since the big-Oh

already denotes the “less-than-or-equal-to” concept. Likewise, although common, it is not fully correct to say “ f (n)= O(g(n)),” with the usual understanding of the “=” relation, because there is no way to make sense of the symmetric statement, “O(g(n)) = f (n).” It is best to say,

“ f (n) is O(g(n)).”

Alternatively, we can say “ f (n) is order of g(n).” For the more mathematically inclined, it is also correct to say, “ f (n) ∈ O(g(n)),” for the big-Oh notation, techni- cally speaking, denotes a whole collection of functions. In this book, we will stick to presenting big-Oh statements as “ f (n) is O(g(n)).” Even with this interpretation,

there is considerable freedom in how we can use arithmetic operations with the big- Oh notation, and with this freedom comes a certain amount of responsibility.

Characterizing Running Times Using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always deﬁned as a chosen measure of the “size” of the problem. For example, if we are interested in ﬁnding the largest element in a sequence, as with the ﬁnd max algorithm, we should let n denote the number of elements in that collection. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of algorithm ﬁnd max (Code Fragment 3.1) for any computer.

Proposition 3.7: The algorithm, ﬁnd max, for computing the maximum element of a list of n numbers, runs in O(n) time.

Justiﬁcation: The initialization before the loop begins requires only a constant number of primitive operations. Each iteration of the loop also requires only a con- stant number of primitive operations, and the loop executes n times. Therefore, we account for the number of primitive operations being c/ + c// n for appropriate

constants c/ and c// that reﬂect, respectively, the work performed during initializa-

tion and the loop body. Because each primitive operation runs in constant time, we have that the running time of algorithm ﬁnd max on an input of size n is at most a constant times n; that is, we conclude that the running time of algorithm ﬁnd max is O(n).

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

Example 3.8: 5n4 + 3n3 + 2n2 + 4n + 1 is O(n4).

Justiﬁcation: Note that 5n4 + 3n3 + 2n2 + 4n + 1 ≤ (5 + 3 + 2 + 4 + 1)n4 = cn4, for c = 15, when n ≥ n0 = 1.

In fact, we can characterize the growth rate of any polynomial function.

Proposition 3.9: If f (n) is a polynomial of degree d, that is,

f (n)= a0 + a1n + ···+ adnd ,

and ad > 0, then f (n) is O(nd ).

Justiﬁcation: Note that, for n ≥ 1, we have 1 ≤ n ≤ n2 ≤ ··· ≤ nd; hence,

a0 + a1n + a2n2 + ···+ adnd ≤ (|a0|+ |a1|+ |a2|+ ···+ |ad|) nd.

We show that f (n) is O(nd ) by deﬁning c = |a0|+ |a1|+ ···+ |ad| and n0 = 1.

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial. We consider some additional properties of the big-Oh notation in the exercises. Let us consider some further examples here, focusing on combinations of the seven fundamental functions used in algorithm

design. We rely on the mathematical fact that log n ≤ n for n ≥ 1.

Example 3.10: 5n2 + 3n log n + 2n + 5 is O(n2).

Justiﬁcation: 5n2 + 3n log n + 2n + 5 ≤ (5 + 3 + 2 + 5)n2 = cn2, for c = 15, when

n ≥ n0 = 1.

Example 3.11: 20n3 + 10n log n + 5 is O(n3).

Justiﬁcation: 20n3 + 10n log n + 5 ≤ 35n3, for n ≥ 1.

Example 3.12: 3 log n + 2 is O(log n).

Justiﬁcation: 3 log n + 2 ≤ 5 log n, for n ≥ 2. Note that log n is zero for n = 1. That is why we use n ≥ n0 = 2 in this case.

Example 3.13: 2n+2 is O(2n).

Justiﬁcation: 2n+2 = 2n 22 = 4 2n; hence, we can take c = 4 and n0 = 1 in this case.

Example 3.14: 2n + 100 log n is O(n).

Justiﬁcation: 2n + 100 log n ≤ 102n, for n ≥ n0 = 1; hence, we can take c = 102 in this case.

Characterizing Functions in Simplest Terms

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function f (n)= 4n3 + 3n2 is O(n5) or even O(n4), it is more accurate to say that f (n) is O(n3). Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can ﬁnd some food, it may be truthful for the farmer to say, “certainly no longer than 12 hours,” but it is much more accurate (and helpful) for him to say, “you can ﬁnd a market just a few minutes drive up this road.” Thus, even with the big-Oh notation, we should strive as much as possible to tell the whole truth.

It is also considered poor taste to include constant factors and lower-order terms in the big-Oh notation. For example, it is not fashionable to say that the function 2n2 is O(4n2 + 6n log n), although this is completely correct. We should strive instead to describe the function in the big-Oh in simplest terms.

The seven functions listed in Section 3.2 are the most common functions used in conjunction with the big-Oh notation to characterize the running times and space usage of algorithms. Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time 4n2 + n log n is a quadratic-time algorithm, since it runs in O(n2) time. Likewise, an algorithm running in time at most 5n + 20 log n + 4 would be called a linear-time algorithm.

Big-Omega

Just as the big-Oh notation provides an asymptotic way of saying that a function is “less than or equal to” another function, the following notations provide an asymp- totic way of saying that a function grows at a rate that is “greater than or equal to” that of another.

Let f (n) and g(n) be functions mapping positive integers to positive real num- bers. We say that f (n) is (g(n)), pronounced “ f (n) is big-Omega of g(n),” if g(n) is O( f (n)), that is, there is a real constant c > 0 and an integer constant n0 ≥ 1 such that

f (n) ≥ cg(n), for n ≥ n0.

This deﬁnition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Example 3.15: 3n log n − 2n is (n log n).

Justiﬁcation: 3n log n − 2n = n log n + 2n(log n − 1) ≥ n log n for n ≥ 2; hence, we can take c = 1 and n0 = 2 in this case.

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that f (n) is (g(n)), pronounced “ f (n)

is big-Theta of g(n),” if f (n) is O(g(n)) and f (n) is (g(n)) , that is, there are real constants c/ > 0 and c// > 0, and an integer constant n0 ≥ 1 such that

c/g(n) ≤ f (n) ≤ c//g(n), for n ≥ n0.

Example 3.16: 3n log n + 4n + 5 log n is (n log n).

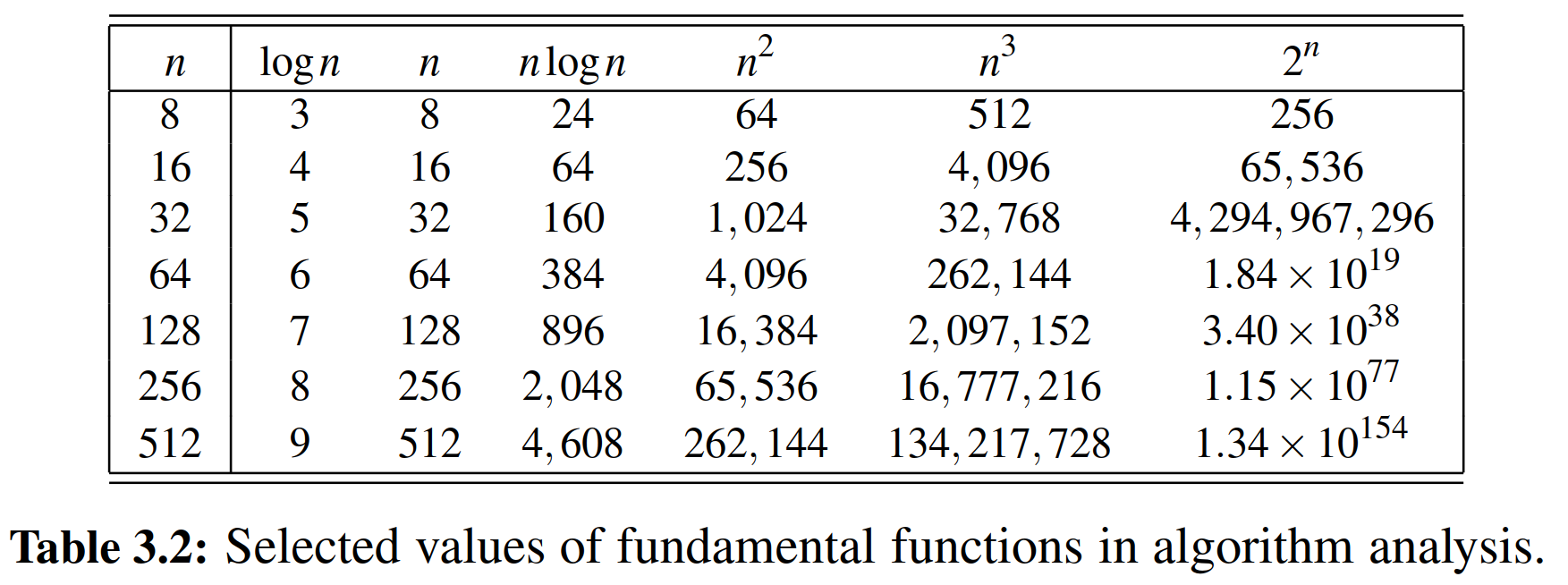
Justiﬁcation: 3n log n ≤ 3n log n + 4n + 5 log n ≤ (3 + 4 + 5)n log n for n ≥ 2.

3.3.2 Comparative Analysis

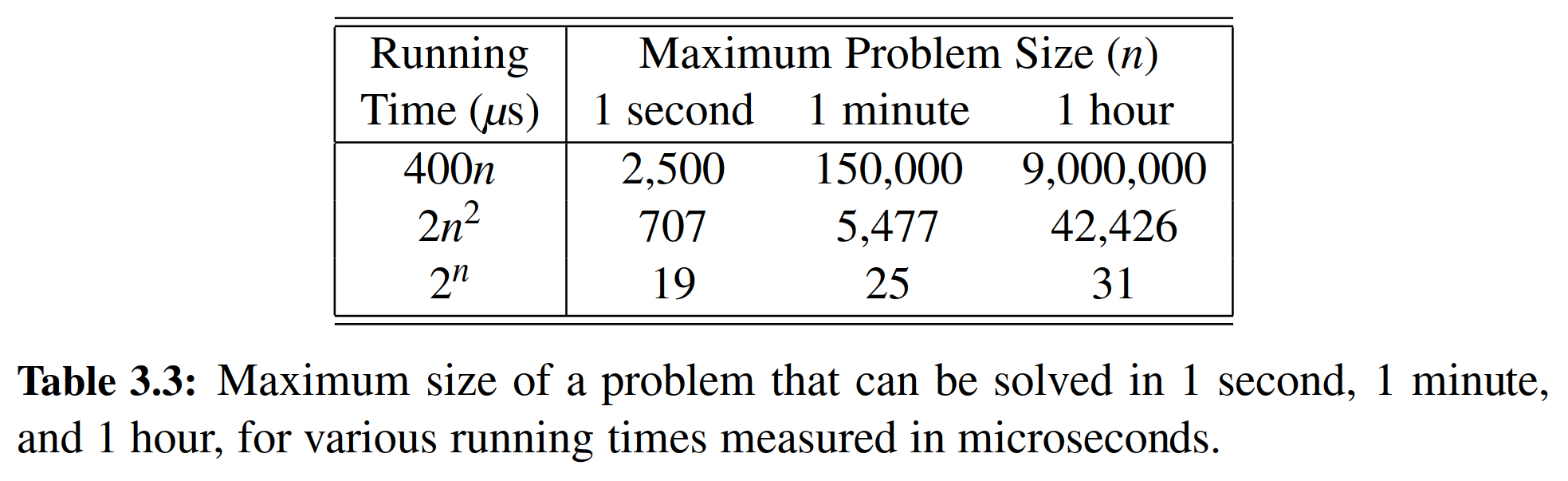
Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of O(n), and an algorithm B, which has a running time of O(n2). Which algorithm is better? We know that n is O(n2), which implies that algorithm A is asymptotically better than algorithm B, although for a small value of n, B may have a lower running time than A.

We can use the big-Oh notation to order classes of functions by asymptotic growth rate. Our seven functions are ordered by increasing growth rate in the fol- lowing sequence, that is, if a function f (n) precedes a function g(n) in the sequence, then f (n) is O(g(n)):

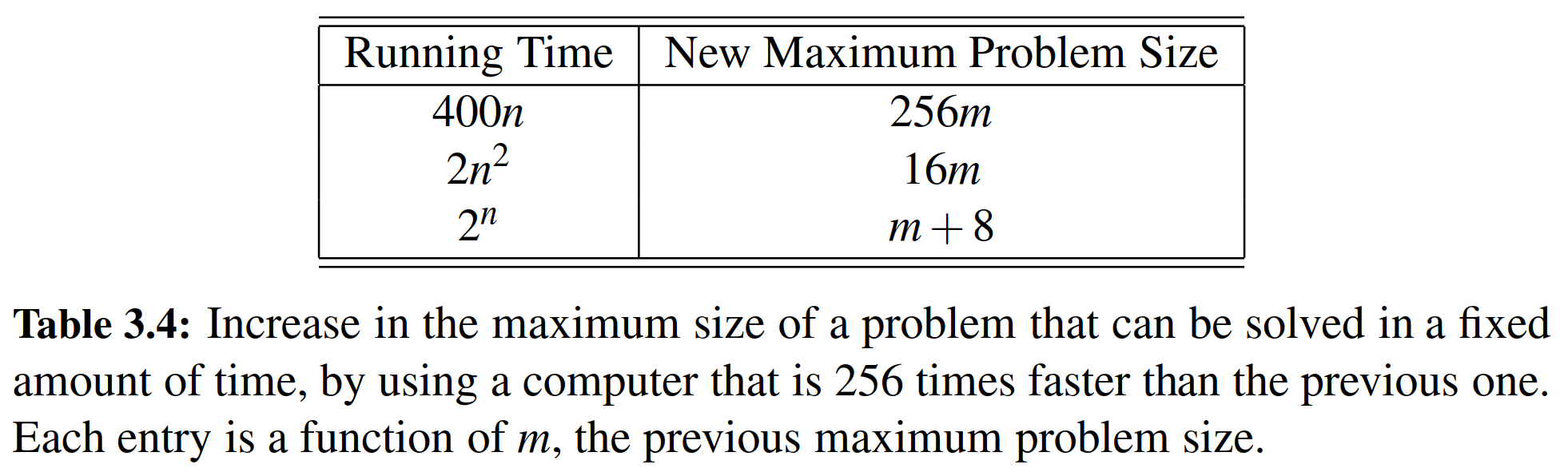
We illustrate the growth rates of the seven functions in Table 3.2. (See also Figure 3.4 from Section 3.2.1.)



We further illustrate the importance of the asymptotic viewpoint in Table 3.3. This table explores the maximum size allowed for an input instance that is pro- cessed by an algorithm in 1 second, 1 minute, and 1 hour. It shows the importance of good algorithm design, because an asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse.



The importance of good algorithm design goes beyond just what can be solved effectively on a given computer, however. As shown in Table 3.4, even if we achieve a dramatic speedup in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm. This table shows the new maximum problem size achievable for any ﬁxed amount of time, assuming algorithms with the given running times are now run on a computer 256 times faster than the previous one.



Some Words of Caution

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they “hide” be very large. For example, while it is true that the function 10100n is O(n), if this is the running time of an algorithm being compared to one whose running time is 10n log n, we should prefer the O(n log n)- time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10100, which is called “one googol,” is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower-order terms we are “hiding.”

The observation above raises the issue of what constitutes a “fast” algorithm. Generally speaking, any algorithm running in O(n log n) time (with a reasonable constant factor) should be considered efﬁcient. Even an O(n2)-time function may be fast enough in some contexts, that is, when n is small. But an algorithm running in O(2n) time should almost never be considered efﬁcient.

Exponential Running Times

There is a famous story about the inventor of the game of chess. He asked only that his king pay him 1 grain of rice for the ﬁrst square on the board, 2 grains for the second, 4 grains for the third, 8 for the fourth, and so on. It is an interesting test of programming skills to write a program to compute exactly the number of grains of rice the king would have to pay.

If we must draw a line between efﬁcient and inefﬁcient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinc- tion between algorithms with a running time that is O(nc), for some constant c > 1, and those with a running time that is O(bn), for some constant b > 1. Like so many notions we have discussed in this section, this too should be taken with a “grain of salt,” for an algorithm running in O(n100) time should probably not be considered “efﬁcient.” Even so, the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.

3.3.3 Examples of Algorithm Analysis

Now that we have the big-Oh notation for doing algorithm analysis, let us give some examples by characterizing the running time of some simple algorithms using this notation. Moreover, in keeping with our earlier promise, we illustrate below how each of the seven functions given earlier in this chapter can be used to characterize the running time of an example algorithm.

Rather than use pseudo-code in this section, we give complete Python imple- mentations for our examples. We use Python’s list class as the natural representa- tion for an “array” of values. In Chapter 5, we will fully explore the underpinnings of Python’s list class, and the efﬁciency of the various behaviors that it supports. In this section, we rely on just a few of its behaviors, discussing their efﬁciencies as introduced.

Constant-Time Operations

Given an instance, named data, of the Python list class, a call to the function, len(data), is evaluated in constant time. This is a very simple algorithm because the list class maintains, for each list, an instance variable that records the current length of the list. This allows it to immediately report that length, rather than take time to iteratively count each of the elements in the list. Using asymptotic notation, we say that this function runs in O(1) time; that is, the running time of this function is independent of the length, n, of the list.

Another central behavior of Python’s list class is that it allows access to an arbi- trary element of the list using syntax, data[j], for integer index j. Because Python’s lists are implemented as array-based sequences, references to a list’s elements are stored in a consecutive block of memory. The jth element of the list can be found, not by iterating through the list one element at a time, but by validating the index, and using it as an offset into the underlying array. In turn, computer hardware sup- ports constant-time access to an element based on its memory address. Therefore, we say that the expression data[j] is evaluated in O(1) time for a Python list.

Revisiting the Problem of Finding the Maximum of a Sequence

For our next example, we revisit the ﬁnd max algorithm, given in Code Frag- ment 3.1 on page 123, for ﬁnding the largest value in a sequence. Proposition 3.7 on page 125 claimed an O(n) run-time for the ﬁnd max algorithm. Consistent with our earlier analysis of syntax data[0], the initialization uses O(1) time. The loop executes n times, and within each iteration, it performs one comparison and possi- bly one assignment statement (as well as maintenance of the loop variable). Finally, we note that the mechanism for enacting a return statement in Python uses O(1) time. Combining these steps, we have that the ﬁnd max function runs in O(n) time.

Further Analysis of the Maximum-Finding Algorithm

A more interesting question about ﬁnd max is how many times we might update the current “biggest” value. In the worst case, if the data is given to us in increasing order, the biggest value is reassigned n 1 times. But what if the input is given to us in random order, with all orders equally likely; what would be the expected number of times we update the biggest value in this case? To answer this question, note that we update the current biggest in an iteration of the loop only if the current element is bigger than all the elements that precede it. If the sequence is given to us in random order, the probability that the jth element is the largest of the ﬁrst j elements is 1/ j (assuming uniqueness). Hence, the expected number of times we

update the biggest (including initialization) is Hn = n

1/ j, which is known as

the nth Harmonic number. It turns out (see Proposition B.16) that Hn is O(log n). Therefore, the expected number of times the biggest value is updated by ﬁnd max on a randomly ordered sequence is O(log n).

Preﬁx Averages

The next problem we consider is computing what are known as preﬁx averages of a sequence of numbers. Namely, given a sequence S consisting of n num- bers, we want to compute a sequence A such that A[ j] is the average of elements

S[0],... , S[ j], for j = 0,..., n − 1, that is,

A[ j]=

j i=0

S[i]

.

j + 1

Computing preﬁx averages has many applications in economics and statistics. For example, given the year-by-year returns of a mutual fund, ordered from recent to past, an investor will typically want to see the fund’s average annual returns for the most recent year, the most recent three years, the most recent ﬁve years, and so on. Likewise, given a stream of daily Web usage logs, a Web site manager may wish to track average usage trends over various time periods. We analyze three different implementations that solve this problem but with rather different running times.

A Quadratic-Time Algorithm

Our ﬁrst algorithm for computing preﬁx averages, named preﬁx average1, is shown in Code Fragment 3.2. It computes every element of A separately, using an inner loop to compute the partial sum.

In order to analyze the preﬁx average1 algorithm, we consider the various steps that are executed.

• The statement, n = len(S), executes in constant time, as described at the beginning of Section 3.3.3.

The statement, A = [0] n, causes the creation and initialization of a Python list with length n, and with all entries equal to zero. This uses a constant number of primitive operations per element, and thus runs in O(n) time.

• There are two nested for loops, which are controlled, respectively, by coun- ters j and i. The body of the outer loop, controlled by counter j, is ex- ecuted n times, for j = 0,... , n − 1. Therefore, statements total = 0 and

A[j] = total / (j+1) are executed n times each. This implies that these two statements, plus the management of counter j in the range, contribute a num- ber of primitive operations proportional to n, that is, O(n) time.

• The body of the inner loop, which is controlled by counter i, is executed j + 1

times, depending on the current value of the outer loop counter j. Thus, state- ment total += S[i], in the inner loop, is executed 1 + 2 + 3 + ··· + n times. By recalling Proposition 3.3, we know that 1 + 2 + 3 + ···+ n = n(n + 1)/2, which implies that the statement in the inner loop contributes O(n ) time.

A similar argument can be done for the primitive operations associated with maintaining counter i, which also take O(n2) time.

The running time of implementation preﬁx average1 is given by the sum of three terms. The ﬁrst and the second terms are O(n), and the third term is O(n2). By a simple application of Proposition 3.9, the running time of preﬁx average1 is O(n2).

Our second implementation for computing preﬁx averages, preﬁx average2, is presented in Code Fragment 3.3.

1 def preﬁx average2(S):

2 ”””Return list such that, for all j, A[j] equals average of S[0], ..., S[j].”””

3 n = len(S)

4 A = [0] n # create new list of n zeros

5 for j in range(n):

6 A[j] = sum(S[0:j+1]) / (j+1) # record the average

7 return A

Code Fragment 3.3: Algorithm preﬁx average2.

This approach is essentially the same high-level algorithm as in preﬁx average1, but we have replaced the inner loop by using the single expression sum(S[0:j+1]) to compute the partial sum, S[0]+ + S[ j]. While the use of that function greatly simpliﬁes the presentation of the algorithm, it is worth asking how it affects the efﬁciency. Asymptotically, this implementation is no better. Even though the ex- pression, sum(S[0:j+1]), seems like a single command, it is a function call and an evaluation of that function takes O( j + 1) time in this context. Technically, the computation of the slice, S[0:j+1], also uses O( j + 1) time, as it constructs a new

list instance for storage. So the running time of preﬁx average2 is still dominated by a series of steps that take time proportional to 1+ 2 + 3 +···+ n, and thus O(n ).

A Linear-Time Algorithm

Our ﬁnal algorithm, preﬁx averages3, is given in Code Fragment 3.4. Just as with our ﬁrst two algorithms, we are interested in computing, for each j, the preﬁx sum S[0]+ S[1]+ ··· + S[ j], denoted as total in our code, so that we can then compute

the preﬁx average A[j] =total / (j + 1). However, there is a key difference that results in much greater efﬁciency.

1 def preﬁx average3(S):

2 ”””Return list such that, for all j, A[j] equals average of S[0], ..., S[j].”””

3 n = len(S)

4 A = [0] n # create new list of n zeros

5 total = 0 # compute preﬁx sum as S[0] + S[1] + ...

6 for j in range(n):

7 total += S[j] # update preﬁx sum to include S[j]

8 A[j] = total / (j+1) # compute average based on current sum

9 return A

Code Fragment 3.4: Algorithm preﬁx average3.

In our ﬁrst two algorithms, the preﬁx sum is computed anew for each value of j. That contributed O( j) time for each j, leading to the quadratic behavior. In algo- rithm preﬁx average3, we maintain the current preﬁx sum dynamically, effectively

computing S[0]+ S[1]+ ··· + S[ j] as total + S[j], where value total is equal to the sum S[0]+ S[1]+ ··· + S[ j − 1] computed by the previous pass of the loop over j.

The analysis of the running time of algorithm preﬁx average3 follows:

• Initializing variables n and total uses O(1) time.

• Initializing the list A uses O(n) time.

• There isa single for loop, which is controlled by counter j. The maintenance of that counter by the range iterator contributes a total of O(n) time.

• The body of the loop is executed n times, for j = 0,... , n − 1. Thus, state- ments total += S[j] and A[j] = total / (j+1) are executed n times each. Since each of these statements uses O(1) time per iteration, their overall contribution is O(n) time.

The running time of algorithm preﬁx average3 is given by the sum of the four terms. The ﬁrst is O(1) and the remaining three are O(n). By a simple application of Proposition 3.9, the running time of preﬁx average3 is O(n), which is much better than the quadratic time of algorithms preﬁx average1 and preﬁx average2.

Three-Way Set Disjointness

Suppose we are given three sequences of numbers, A, B, and C. We will assume that no individual sequence contains duplicate values, but that there may be some numbers that are in two or three of the sequences. The three-way set disjointness problem is to determine if the intersection of the three sequences is empty, namely,

that there is no element x such that x ∈ A, x ∈ B, and x ∈ C. A simple Python

function to determine this property is given in Code Fragment 3.5.

1 def disjoint1(A, B, C):

2 ”””Return True if there is no element common to all three lists.”””

3 for a in A:

4 for b in B:

5 for c in C:

6 if a == b == c:

7 return False # we found a common value

8 return True # if we reach this, sets are disjoint

Code Fragment 3.5: Algorithm disjoint1 for testing three-way set disjointness.

This simple algorithm loops through each possible triple of values from the three sets to see if those values are equivalent. If each of the original sets has size n, then the worst-case running time of this function is O(n3).

We can improve upon the asymptotic performance with a simple observation. Once inside the body of the loop over B, if selected elements a and b do not match each other, it is a waste of time to iterate through all values of C looking for a matching triple. An improved solution to this problem, taking advantage of this observation, is presented in Code Fragment 3.6.

1 def disjoint2(A, B, C):

2 ”””Return True if there is no element common to all three lists.”””

3 for a in A:

4 for b in B:

5 if a == b: # only check C if we found match from A and B

6 for c in C:

7 if a == c # (and thus a == b == c)

8 return False # we found a common value

9 return True # if we reach this, sets are disjoint

Code Fragment 3.6: Algorithm disjoint2 for testing three-way set disjointness.

In the improved version, it is not simply that we save time if we get lucky. We claim that the worst-case running time for disjoint2 is O(n2). There are quadrat- ically many pairs (a, b) to consider. However, if A and B are each sets of distinct elements, there can be at most O(n) such pairs with a equal to b. Therefore, the innermost loop, over C, executes at most n times.

To account for the overall running time, we examine the time spent executing each line of code. The management of the for loop over A requires O(n) time. The management of the for loop over B accounts for a total of O(n2) time, since that loop is executed n different times. The test a == b is evaluated O(n2) times. The rest of the time spent depends upon how many matching (a, b) pairs exist. As we have noted, there are at most n such pairs, and so the management of the loop over C, and the commands within the body of that loop, use at most O(n2) time. By our standard application of Proposition 3.9, the total time spent is O(n2).

Element Uniqueness

A problem that is closely related to the three-way set disjointness problem is the element uniqueness problem. In the former, we are given three collections and we presumed that there were no duplicates within a single collection. In the element uniqueness problem, we are given a single sequence S with n elements and asked whether all elements of that collection are distinct from each other.

Our ﬁrst solution to this problem uses a straightforward iterative algorithm. The unique1 function, given in Code Fragment 3.7, solves the element uniqueness problem by looping through all distinct pairs of indices j < k, checking if any of

those pairs refer to elements that are equivalent to each other. It does this using two nested for loops, such that the ﬁrst iteration of the outer loop causes n − 1 iterations of the inner loop, the second iteration of the outer loop causes n − 2 iterations of

the inner loop, and so on. Thus, the worst-case running time of this function is proportional to

(n − 1)+ (n − 2)+ ···+ 2 + 1,

which we recognize as the familiar O(n2) summation from Proposition 3.3.

Using Sorting as a Problem-Solving Tool

An even better algorithm for the element uniqueness problem is based on using sorting as a problem-solving tool. In this case, by sorting the sequence of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, to determine if there are any duplicates, all we need to do is perform a sin- gle pass over the sorted sequence, looking for consecutive duplicates. A Python implementation of this algorithm is as follows:

1 def unique2(S):

2 ”””Return True if there are no duplicate elements in sequence S.”””

3 temp = sorted(S) # create a sorted copy of S

4 for j in range(1, len(temp)):

5 if S[j−1] == S[j]:

6 return False # found duplicate pair

7 return True # if we reach this, elements were unique

Code Fragment 3.8: Algorithm unique2 for testing element uniqueness.

The built-in function, sorted, as described in Section 1.5.2, produces a copy of the original list with elements in sorted order. It guarantees a worst-case running time of O(n log n); see Chapter 12 for a discussion of common sorting algorithms. Once the data is sorted, the subsequent loop runs in O(n) time, and so the entire unique2 algorithm runs in O(n log n) time.

1 def unique1(S):

2 ”””Return True if there are no duplicate elements in sequence S.”””

3 for j in range(len(S)):

4 for k in range(j+1, len(S)):

5 if S[j] == S[k]:

6 return False # found duplicate pair

7 return True # if we reach this, elements were unique

Code Fragment 3.7: Algorithm unique1 for testing element uniqueness.

those pairs refer to elements that are equivalent to each other. It does this using two nested for loops, such that the ﬁrst iteration of the outer loop causes n − 1 iterations of the inner loop, the second iteration of the outer loop causes n − 2 iterations of

the inner loop, and so on. Thus, the worst-case running time of this function is proportional to

(n − 1)+ (n − 2)+ ···+ 2 + 1,

which we recognize as the familiar O(n2) summation from Proposition 3.3.

Using Sorting as a Problem-Solving Tool

An even better algorithm for the element uniqueness problem is based on using sorting as a problem-solving tool. In this case, by sorting the sequence of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, to determine if there are any duplicates, all we need to do is perform a sin- gle pass over the sorted sequence, looking for consecutive duplicates. A Python implementation of this algorithm is as follows:

1 def unique2(S):

2 ”””Return True if there are no duplicate elements in sequence S.”””

3 temp = sorted(S) # create a sorted copy of S

4 for j in range(1, len(temp)):

5 if S[j−1] == S[j]:

6 return False # found duplicate pair

7 return True # if we reach this, elements were unique

Code Fragment 3.8: Algorithm unique2 for testing element uniqueness.

The built-in function, sorted, as described in Section 1.5.2, produces a copy of the original list with elements in sorted order. It guarantees a worst-case running time of O(n log n); see Chapter 12 for a discussion of common sorting algorithms. Once the data is sorted, the subsequent loop runs in O(n) time, and so the entire unique2 algorithm runs in O(n log n) time.

3.4 Simple Justiﬁcation Techniques

Sometimes, we will want to make claims about an algorithm, such as showing that it is correct or that it runs fast. In order to rigorously make such claims, we must use mathematical language, and in order to back up such claims, we must justify or prove our statements. Fortunately, there are several simple ways to do this.

3.4.1 By Example

Some claims are of the generic form, “There is an element x in a set S that has property P.” To justify such a claim, we only need to produce a particular x in S that has property P. Likewise, some hard-to-believe claims are of the generic form, “Every element x in a set S has property P.” To justify that such a claim is false, we only need to produce a particular x from S that does not have property P. Such an instance is called a counterexample.

Example 3.17: Professor Amongus claims that every number of the form 2i 1

is a prime, when i is an integer greater than 1. Professor Amongus is wrong.

Justiﬁcation: To prove Professor Amongus is wrong, we ﬁnd a counterexample. Fortunately, we need not look too far, for 24 − 1 = 15 = 3 · 5.

Proposition 3.20: Consider the Fibonacci function F(n), which is deﬁned such that F(1)= 1, F(2)= 2, and F(n)= F(n 2)+ F(n 1) for n > 2. (See Sec- tion 1.8.) We claim that F (n) < 2n.

Justiﬁcation: We will show our claim is correct by induction.

Base cases: (n 2). F (1)= 1 < 2 = 21 and F(2)= 2 < 4 = 22.

Induction step: (n > 2). Suppose our claim is true for all n/ < n. Consider F(n). Since n > 2, F(n)= F(n − 2)+ F (n − 1). Moreover, since both n − 2 and n − 1 are less than n, we can apply the inductive assumption (sometimes called the “inductive hypothesis”) to imply that F(n) < 2n−2 + 2n−1, since

2n−2 + 2n−1 < 2n−1 + 2n−1 = 2 · 2n−1 = 2n.

Let us do another inductive argument, this time for a fact we have seen before.

Proposition 3.21: (which is the same as Proposition 3.3)

n

 i =

i=1

n(n + 1)

.

2

Justiﬁcation: We will justify this equality by induction.

Base case: n = 1. Trivial, for 1 = n(n + 1)/2, if n = 1.

Induction step: n ≥ 2. Assume the claim is true for n/ < n. Consider n.

n n−1

 i = n +  i.

i=1

By the induction hypothesis, then

n

 i = n +

i=1

i=1

(n 1)n 2 ,

which we can simplify as

2 2

(n − 1)n 2n + n − n n + n n(n + 1)

n + 2 = 2 = 2 = 2 .

We may sometimes feel overwhelmed by the task of justifying something true for all n ≥ 1. We should remember, however, the concreteness of the inductive tech- nique. It shows that, for any particular n, there is a ﬁnite step-by-step sequence of

implications that starts with something true and leads to the truth about n. In short, the inductive argument is a template for building a sequence of direct justiﬁcations.

Loop Invariants

The ﬁnal justiﬁcation technique we discuss in this section is the loop invariant. To prove some statement L about a loop is correct, deﬁne L in terms of a series of smaller statements L0, L1,..., Lk, where:

1. The initial claim, L0, is true before the loop begins.

2. If L j−1 is true before iteration j, then L j will be true after iteration j.

3. The ﬁnal statement, Lk, implies the desired statement L to be true.

Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, we use a loop invariant to justify that the function, ﬁnd (see Code Fragment 3.9), ﬁnds the smallest index at which ele- ment val occurs in sequence S.

1 def ﬁnd(S, val):

2 ”””Return index j such that S[j] == val, or -1 if no such element.”””

3 n = len(S)

4 j = 0

5 while j < n:

6 if S[j] == val:

7 return j # a match was found at index j

8 j += 1

9 return −1

Code Fragment 3.9: Algorithm for ﬁnding the ﬁrst index at which a given element occurs in a Python list.

To show that ﬁnd is correct, we inductively deﬁne a series of statements, L j, that lead to the correctness of our algorithm. Speciﬁcally, we claim the following is true at the beginning of iteration j of the while loop:

L j: val is not equal to any of the ﬁrst j elements of S.

This claim is true at the beginning of the ﬁrst iteration of the loop, because j is 0 and there are no elements among the ﬁrst 0 in S (this kind of a trivially true claim is said to hold vacuously). In iteration j, we compare element val to element S[ j] and return the index j if these two elements are equivalent, which is clearly correct and completes the algorithm in this case. If the two elements val and S[ j] are not equal, then we have found one more element not equal to val and we increment the index j. Thus, the claim j will be true for this new value of j; hence, it is true at the beginning of the next iteration. If the while loop terminates without ever returning an index in S, then we have j = n. That is, Ln is true—there are no elements of S equal to val. Therefore, the algorithm correctly returns −1 to indicate that val is not in S.

3.4.2 The “Contra” Attack

Another set of justiﬁcation techniques involves the use of the negative. The two primary such methods are the use of the contrapositive and the contradiction. The use of the contrapositive method is like looking through a negative mirror. To justify the statement “if p is true, then q is true,” we establish that “if q is not true, then p is not true” instead. Logically, these two statements are the same, but the latter, which is called the contrapositive of the ﬁrst, may be easier to think about.

Example 3.18: Let a and b be integers. If ab is even, then a is even or b is even.

Justiﬁcation: To justify this claim, consider the contrapositive, “If a is odd and b is odd, then ab is odd.” So, suppose a = 2 j + 1 and b = 2k + 1, for some integers j and k. Then ab = 4 jk + 2 j + 2k + 1 = 2(2 jk + j + k)+ 1; hence, ab is odd.

Besides showing a use of the contrapositive justiﬁcation technique, the previous example also contains an application of DeMorgan’s Law. This law helps us deal with negations, for it states that the negation of a statement of the form “p or q” is “not p and not q.” Likewise, it states that the negation of a statement of the form “p and q” is “not p or not q.”

Contradiction

Another negative justiﬁcation technique is justiﬁcation by contradiction, which also often involves using DeMorgan’s Law. In applying the justiﬁcation by con- tradiction technique, we establish that a statement q is true by ﬁrst supposing that q is false and then showing that this assumption leads to a contradiction (such as

2 /= 2 or 1 > 3). By reaching such a contradiction, we show that no consistent sit-

uation exists with q being false, so q must be true. Of course, in order to reach this conclusion, we must be sure our situation is consistent before we assume q is false.

Example 3.19: Let a and b be integers. If ab is odd, then a is odd and b is odd.

Justiﬁcation: Let ab be odd. We wish to show that a is odd and b is odd. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is even or b is even. In fact, without loss of generality, we can assume that a is even (since the case for b is symmetric). Then a = 2 j for some integer

j. Hence, ab = (2 j)b = 2( jb), that is, ab is even. But this is a contradiction: ab

cannot simultaneously be odd and even. Therefore, a is odd and b is odd.

3.4.3 Induction and Loop Invariants

Most of the claims we make about a running time or a space bound involve an inte- ger parameter n (usually denoting an intuitive notion of the “size” of the problem). Moreover, most of these claims are equivalent to saying some statement q(n) is true “for all n 1.” Since this is making a claim about an inﬁnite set of numbers, we cannot justify this exhaustively in a direct fashion.

Induction

We can often justify claims such as those above as true, however, by using the technique of induction. This technique amounts to showing that, for any particular n ≥ 1, there is a ﬁnite sequence of implications that starts with something known

to be true and ultimately leads to showing that q(n) is true. Speciﬁcally, we begin a justiﬁcation by induction by showing that q(n) is true for n = 1 (and possibly some other values n = 2, 3,... , k, for some constant k). Then we justify that the inductive “step” is true for n > k, namely, we show “if q( j) is true for all j < n, then q(n) is true.” The combination of these two pieces completes the justiﬁcation by induction.

**END**

# 六、实验体会

根据书中给出的分析方式，可以初步分析一些带有循环的算法的时间复杂度。但是有些还是很难自己证明。这有待进一步的学习。

在快速排序中，我曾经写过一段代码，直接导致了很错误的情况。数据结构与算法分析：C语言描述（原书第二版）的Page179，直接指出了这种愚蠢的做法。

**一种错误的方法**

通常的、没有经过充分考虑的选择是将第一个元素用作枢纽元。如果输入是随机的，那么这是可以接受的，但是如果输入是预排序的或是反序的，那么这样的枢纽元就产生一个劣质的分割，因为所有的元素不是都被划入就是都被划入。更有甚者，这种情况可能发生在所有的递归调用中。实际上，如果第一个元素用作枢纽元而且输入是预先排序的，那么快速排序花费的时间将是二次的，可是实际上却根本没干什么事，这是相当尴尬的。然而，预排序的输入（或具有一大段预排序的输入）是相当常见的，因此，使用第一个元素作为枢纽元是绝对糟糕的主意，应该立即放弃这种想法。另一种想法是选取前两个互异的关键字中的较大者作为枢纽元，不过这和只选取第一个元素作为枢纽元具有相同的害处。不要使用这两种选取枢纽元的策略。

**一种安全的做法**

一种安全的方针是随机选取枢纽元。一般来说这种策略非常安全，除非随机数生成器有问题（它不像你可能想象的那么罕见），因为随机的枢纽元不可能总在接连不断地产生劣质的分割。另一方面，随机数的生成一般是昂贵的，根本减少不了算法其余部分的平均运行时间。

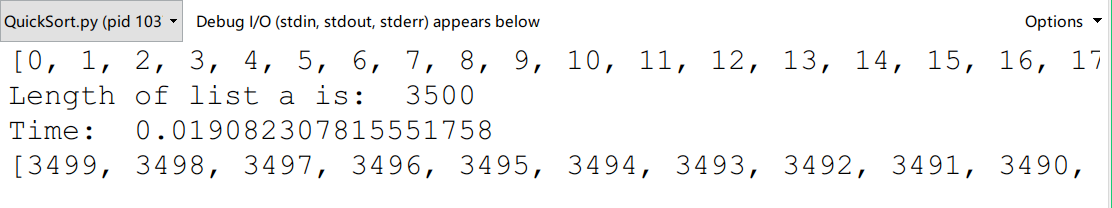
**三数中值分割法**

一组N个数的中值是第[N/2]个最大的数。枢纽元的最好的选择是数组的中值。不幸的是，这很难算出，且明显减慢快速排序的速度。这样的中值的估计量可以通过随机选取三个元素并使用他们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。。例如，输入为1，1，4，9，6，3，5，，7，0，它的左边元素是8，右边元素是0，中心位置（）上的元素是6.于是枢纽元则是。显然使用三数中值分割法消除了预排序输入的坏情况（在这种情况下，这些分割都是一样的），并且减少了快速排序大约5%的运行时间。

Python的强大便捷，使得最后的三数中值分割变得很简单。对代码进行稍微的改动就可以了。

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33 | # Quick Sort for a list  **from** time **import** time  **import** sys  sys**.**setrecursionlimit**(**1000000**)**  **def** QuickSort**(**L**):**  L\_Left **=** **[]**  L\_Right **=** **[]**  L\_Middle **=** **[]**  **if** len**(**L**)** **<=** 1**:**  **return** L  **else:**  **for** i **in** L**:**  pivot **=** **(**L**[**0**]** **+** L**[-**1**]** **+** L**[**len**(**L**)//**2**])/**3  **if** i **>** pivot**:**  L\_Left**.**append**(**i**)**  **elif** i **<** pivot**:**  L\_Right**.**append**(**i**)**  **else:**  L\_Middle**.**append**(**i**)**  L\_Left **=** QuickSort**(**L\_Left**)**  L\_Right **=** QuickSort**(**L\_Right**)**  **return** L\_Left **+** L\_Middle **+** L\_Right  A **=** list**(**range**(**3500**))**  **print(**A**)**  **print(**"Length of list a is: "**,**len**(**A**))**  begin **=** time**()**  A **=** QuickSort**(**A**)**  end **=** time**()**  **print(**"Time: "**,**end **-** begin**)**  **print(**A**)** |

程序代码 4



运行结果 3

可以看到，这个程序在改动pivot的意义之后，速度变得明显快了很多。之前需要一秒多的运行时间，这里只需要0.02秒，提升了50倍！

# 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[[1]](#endnote-1)

[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

1. [↑](#endnote-ref-1)