

[CHAPTER]




11

SEARCHING, SORTING, AND Complexity Analysis

After completing this chapter, you will be able to:

- Measure the performance of an algorithm by obtaining running times and instruction counts with different data sets
- Analyze an algorithm's performance by determining its order of complexity, using big-O notation
- Distinguish the common orders of complexity and the algorithmic patterns that exhibit them
- Distinguish between the improvements obtained by tweaking an algorithm and reducing its order of complexity
- Write a simple linear search algorithm and a simple sort algorithm

Earlier in this book, you learned about several criteria for assessing the quality of an algorithm. The most essential criterion is correctness, but readability and ease of maintenance are also important. This chapter examines another important criterion of the quality of algorithms—run-time performance.



Algorithms describe processes that run on real computers with finite resources. Processes consume two resources: processing time and space or memory. When run with the same problems or data sets, processes that consume less of these two resources are of higher quality than processes that consume more, and so are the corresponding algorithms. In this chapter, we introduce tools for complexity analysis—for assessing the run-time performance or efficiency of algorithms. We also apply these tools to search algorithms and sort algorithms.

11.1 Measuring the Efficiency of Algorithms

Some algorithms consume an amount of time or memory that is below a threshold of tolerance. For example, most users are happy with any algorithm that loads a file in less than one second. For such users, any algorithm that meets this requirement is as good as any other. Other algorithms take an amount of time that is totally impractical (say, thousands of years) with large data sets. We can't use these algorithms, and instead need to find others, if they exist, that perform better.

When choosing algorithms, we often have to settle for a space/time tradeoff. An algorithm can be designed to gain faster run times at the cost of using extra space (memory), or the other way around. Some users might be willing to pay for more memory to get a faster algorithm, whereas others would rather settle for a slower algorithm that economizes on memory. Memory is now quite inexpensive for desktop and laptop computers, but not yet for miniature devices.

In any case, because efficiency is a desirable feature of algorithms, it is important to pay attention to the potential of some algorithms for poor performance. In this section, we consider several ways to measure the efficiency of algorithms.

11.1.1 Measuring the Run Time of an Algorithm

One way to measure the time cost of an algorithm is to use the computer's clock to obtain an actual run time. This process, called **benchmarking** or **profiling**, starts by determining the time for several different data sets of the same size and then calculates the average time. Next, similar data are gathered for larger and larger data sets. After several such tests, enough data are available to predict how the algorithm will behave for a data set of any size.

Consider a simple, if unrealistic, example. The following program implements an algorithm that counts from 1 to a given number. Thus, the problem size is the number. We start with the number 10,000,000, time the algorithm, and output the running time to the terminal window. We then double the size of this

number and repeat this process. After five such increases, there is a set of results from which you can generalize. Here is the code for the tester program:

```
"""
File: timing1.py
Prints the running times for problem sizes that double,
using a single loop.
"""

import time

problemSize = 10000000
print ("%12s16s" % ("Problem Size", "Seconds"))
for count in range(5):

    start = time.time()
    # The start of the algorithm
    work = 1
    for x in range(problemSize):
        work += 1
        work -= 1
    # The end of the algorithm
    elapsed = time.time() - start

    print ("%12d%16.3f" % (problemSize, elapsed))
    problemSize *= 2
```

The tester program uses the `time()` function in the `time` module to track the running time. This function returns the number of seconds that have elapsed between the current time on the computer's clock and January 1, 1970 (also called "The Epoch"). Thus, the difference between the results of two calls of `time.time()` represents the elapsed time in seconds. Note also that the program does a constant amount of work, in the form of two extended assignment statements, on each pass through the loop. This work consumes enough time on each iteration so that the total running time is significant, but has no other impact on the results. Figure 11.1 shows the output of the program.

Problem Size	Seconds
10000000	3.8
20000000	7.591
40000000	15.352
80000000	30.697
160000000	61.631

[FIGURE 11.1] The output of the tester program

A quick glance at the results reveals that the running time more or less doubles when the size of the problem doubles. Thus, one might predict that the running time for a problem of size 32,000,000 would be approximately 124 seconds.

As another example, consider the following change in the tester program's algorithm:

```
for j in range(problemSize):
    for k in range(problemSize):
        work += 1
        work -= 1
```

In this version, the extended assignments have been moved into a nested loop. This loop iterates through the size of the problem within another loop that also iterates through the size of the problem. This program was left running overnight. By morning it had processed only the first data set, 1,000,000. The program was then terminated and run again with a smaller problem size of 1000. Figure 11.2 shows the results.

Problem Size	Seconds
1000	0.387
2000	1.581
4000	6.463
8000	25.702
16000	102.666

[FIGURE 11.2] The output of the second tester program with a nested loop and initial problem size of 1000

Note that when the problem size doubles, the number of seconds of running time more or less quadruples. At this rate, it would take 175 days to process the largest number in the previous data set!

This method permits accurate predictions of the running times of many algorithms. However, there are two major problems with this technique:

- 1 Different hardware platforms have different processing speeds, so the running times of an algorithm differ from machine to machine. Also, the running time of a program varies with the type of operating system that lies between it and the hardware. Finally, different programming languages and compilers produce code whose performance varies. For example, an algorithm coded in C usually runs slightly faster than the same algorithm in Python byte code. Thus, predictions of performance generated from the results of timing on one hardware or software platform generally cannot be used to predict potential performance on other platforms.
- 2 It is impractical to determine the running time for some algorithms with very large data sets. For some algorithms, it doesn't matter how fast the compiled code or the hardware processor is. They are impractical to run with very large data sets on any computer.

Although timing algorithms may in some cases be a helpful form of testing, we also want an estimate of the efficiency of an algorithm that is independent of a particular hardware or software platform. As you will learn in the next section, such an estimate tells us how well or how poorly the algorithm would perform on any platform.

11.1.2 Counting Instructions

Another technique used to estimate the efficiency of an algorithm is to count the instructions executed with different problem sizes. These counts provide a good predictor of the amount of abstract work performed by an algorithm, no matter what platform the algorithm runs on. Keep in mind, however, that when you count instructions, you are counting the instructions in the high-level code in which the algorithm is written, not instructions in the executable machine language program.

When analyzing an algorithm in this way, you distinguish between two classes of instructions:

- 1 Instructions that execute the same number of times regardless of the problem size
- 2 Instructions whose execution count varies with the problem size

For now, you ignore instructions in the first class, because they do not figure significantly in this kind of analysis. The instructions in the second class normally are found in loops or recursive functions. In the case of loops, you also zero in on instructions performed in any nested loops or, more simply, just the number of iterations that a nested loop performs. For example, let us wire the algorithm of the previous program to track and display the number of iterations the inner loop executes with the different data sets:

```
"""
File: counting.py
Prints the number of iterations for problem sizes
that double, using a nested loop.
"""

problemSize = 1000
print ("%12s%15s" % ("Problem Size", "Iterations"))
for count in range(5):
    number = 0

    # The start of the algorithm
    work = 1
    for j in range(problemSize):
        for k in range(problemSize):
            number += 1
            work += 1
            work -= 1
    # The end of the algorithm

    print ("%12d%15d" % (problemSize, number))
    problemSize *= 2
```

As you can see from the results, the number of iterations is the square of the problem size (Figure 11.3).

Problem Size	Iterations
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000

[FIGURE 11.3] The output of a tester program that counts iterations

Here is a similar program that tracks the number of calls of a recursive Fibonacci function, introduced in Chapter 6, for several problem sizes. Note that the function now expects a second argument, which is a **Counter** object. Each time the function is called at the top level, a new **Counter** object is created and passed to it. On that call and each recursive call, the function's counter object is incremented.

```
"""
File: countfib.py
Prints the number of calls of a recursive Fibonacci
function with problem sizes that double.
"""

class Counter(object):
    """Tracks a count."""

    def __init__(self):
        self._number = 0

    def increment(self):
        self._number += 1

    def __str__(self):
        return str(self._number)

def fib(n, counter):
    """Count the number of calls of the Fibonacci
    function."""
    counter.increment()
    if n < 3:
        return 1
    else:
        return fib(n - 1, counter) + fib(n - 2, counter)

problemSize = 2
print ("%12s%15s" % ("Problem Size", "Calls"))
for count in range(5):
    counter = Counter()

    # The start of the algorithm
    fib(problemSize, counter)
    # The end of the algorithm

    print ("%12d%15s" % (problemSize, counter))
    problemSize *= 2
```

The output of this program is shown in Figure 11.4.

Problem Size	Calls
2	1
4	5
8	41
16	1973
32	4356617

[FIGURE 11.4] The output of a tester program that runs the Fibonacci function

As the problem size doubles, the instruction count (number of recursive calls) grows slowly at first and then quite rapidly. At first, the instruction count is less than the square of the problem size, but the instruction count of 1973 is significantly larger than 256, the square of the problem size 16.

The problem with tracking counts in this way is that, with some algorithms, the computer still cannot run fast enough to show the counts for very large problem sizes. Counting instructions is the right idea, but we need to turn to logic and mathematical reasoning for a complete method of analysis. The only tools we need for this type of analysis are paper and pencil.

11.1.3 Measuring the Memory Used by an Algorithm

A complete analysis of the resources used by an algorithm includes the amount of memory required. Once again, we focus on rates of potential growth. Some algorithms require the same amount of memory to solve any problem. Other algorithms require more memory as the problem size gets larger.

11.1

Exercises

- 1 Write a tester program that counts and displays the number of iterations of the following loop:

```
while problemSize > 0:  
    problemSize = problemSize // 2
```

- 2 Run the program you created in Exercise 1 using problem sizes of 1000, 2000, 4000, 10,000, and 100,000. As the problem size doubles or increases by a factor of 10, what happens to the number of iterations?
- 3 The difference between the results of two calls of the `time` function `time()` is an elapsed time. Because the operating system might use the CPU for part of this time, the elapsed time might not reflect the actual time that a Python code segment uses the CPU. Browse the Python documentation for an alternative way of recording the processing time and describe how this would be done.

11.2

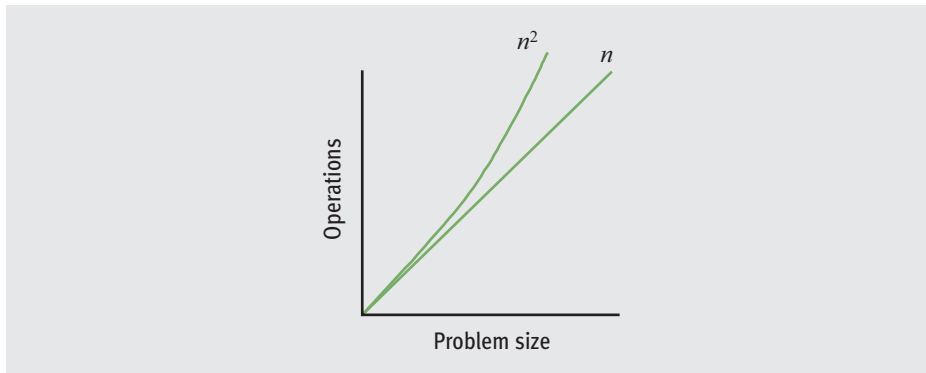
Complexity Analysis

In this section, we develop a method of determining the efficiency of algorithms that allows us to rate them independently of platform-dependent timings or impractical instruction counts. This method, called **complexity analysis**, entails reading the algorithm and using pencil and paper to work out some simple algebra.

11.2.1

Orders of Complexity

Consider the two counting loops discussed earlier. The first loop executes n times for a problem of size n . The second loop contains a nested loop that iterates n^2 times. The amount of work done by these two algorithms is similar for small values of n , but is very different for large values of n . Figure 11.5 and Table 11.1 illustrate this divergence. Note that when we say “work,” we usually mean the number of iterations of the most deeply nested loop.



[FIGURE 11.5] A graph of the amounts of work done in the tester programs

PROBLEM SIZE	WORK OF THE FIRST ALGORITHM	WORK OF THE SECOND ALGORITHM
2	2	4
10	10	100
1000	1000	1,000,000

[TABLE 11.1] The amounts of work in the tester programs

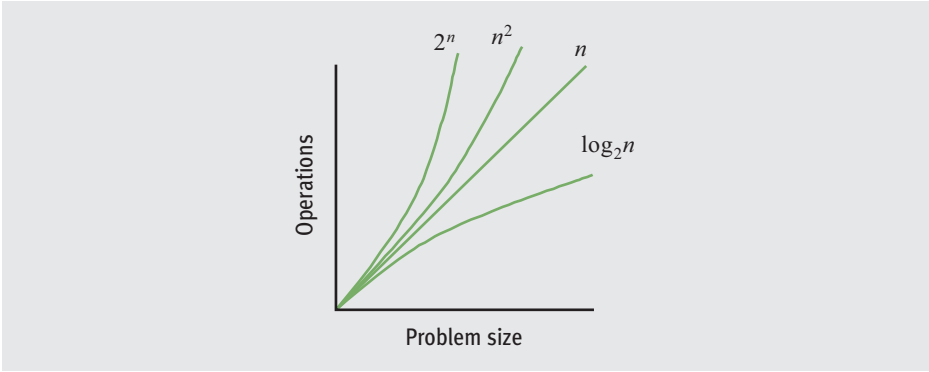
The performances of these algorithms differ by what we call an **order of complexity**. The performance of the first algorithm is **linear** in that its work grows in direct proportion to the size of the problem (problem size of 10, work of 10, 20 and 20, etc.). The behavior of the second algorithm is **quadratic** in that its work grows as a function of the square of the problem size (problem size of 10, work of 100). As you can see from the graph and the table, algorithms with linear behavior do less work than algorithms with quadratic behavior for most problem sizes n . In fact, as the problem size gets larger, the performance of an algorithm with the higher order of complexity becomes worse more quickly.

Several other orders of complexity are commonly used in the analysis of algorithms. An algorithm has **constant** performance if it requires the same number of operations for any problem size. List indexing is a good example of a constant-time algorithm. This is clearly the best kind of performance to have.

Another order of complexity that is better than linear but worse than constant is called **logarithmic**. The amount of work of a logarithmic algorithm is proportional to the \log_2 of the problem size. Thus, when the problem doubles in size, the amount of work only increases by 1 (that is, just add 1).

The work of a **polynomial time algorithm** grows at a rate of n^k , where k is a constant greater than 1. Examples are n^2 , n^3 , and n^{10} .

Although n^3 is worse in some sense than n^2 , they are both of the polynomial order and are better than the next higher order of complexity. An order of complexity that is worse than polynomial is called **exponential**. An example rate of growth of this order is 2^n . Exponential algorithms are impractical to run with large problem sizes. The most common orders of complexity used in the analysis of algorithms are summarized in Figure 11.6 and Table 11.2.



[FIGURE 11.6] A graph of some sample orders of complexity

n	LOGARITHMIC ($\log_2 n$)	LINEAR (n)	QUADRATIC (n^2)	EXPONENTIAL (2^n)
100	7	100	10,000	Off the charts
1000	10	1000	1,000,000	Off the charts
1,000,000	20	1,000,000	1,000,000,000,000	Really off the charts

[TABLE 11.2] Some sample orders of complexity

11.2.2 Big-O Notation

An algorithm rarely performs a number of operations exactly equal to n , n^2 , or k^n . An algorithm usually performs other work in the body of a loop, above the loop, and below the loop. For example, we might more precisely say that an algorithm performs $2n + 3$ or $2n^2$ operations. In the case of a nested loop, the inner loop

might execute one less pass after each pass through the outer loop, so that the total number of iterations might be more like $\frac{1}{2}n^2 - \frac{1}{2}n$, rather than n^2 .

The amount of work in an algorithm typically is the sum of several terms in a polynomial. Whenever the amount of work is expressed as a polynomial, we focus on one term as **dominant**. As n becomes large, the dominant term becomes so large that the amount of work represented by the other terms can be ignored. Thus, for example, in the polynomial $\frac{1}{2}n^2 - \frac{1}{2}n$, we focus on the quadratic term, $\frac{1}{2}n^2$, in effect dropping the linear term, $\frac{1}{2}n$, from consideration. We can also drop the coefficient $\frac{1}{2}$ because the ratio between $\frac{1}{2}n^2$ and n^2 does not change as n grows. For example, if you double the problem size, the run times of algorithms that are $\frac{1}{2}n^2$ and n^2 both increase by a factor of 4. This type of analysis is sometimes called **asymptotic analysis** because the value of a polynomial asymptotically approaches or approximates the value of its largest term as n becomes very large.

One notation that computer scientists use to express the efficiency or computational complexity of an algorithm is called **big-O notation**. “O” stands for “on the order of,” a reference to the order of complexity of the work of the algorithm. Thus, for example, the order of complexity of a linear-time algorithm is $O(n)$. Big-O notation formalizes our discussion of orders of complexity.

11.2.3

The Role of the Constant of Proportionality

The **constant of proportionality** involves the terms and coefficients that are usually ignored during big-O analysis. However, when these items are large, they may have an impact on the algorithm, particularly for small and medium-sized data sets. For example, no one can ignore the difference between n and $n / 2$, when n is \$1,000,000. In the example algorithms discussed thus far, the instructions that execute within a loop are part of the constant of proportionality, as are the instructions that initialize the variables before the loops are entered. When analyzing an algorithm, one must be careful to determine that any instructions do not hide a loop that depends on a variable problem size. If that is the case, then the analysis must move down into the nested loop, as we saw in the last example.

Let’s determine the constant of proportionality for the first algorithm discussed in this chapter. Here is the code:

```
work = 1
for x in range(problemSize):
    work += 1
    work -= 1
```

Note that, aside from the loop itself, there are three lines of code, each of them assignment statements. Each of these three statements runs in constant time. Let's also assume that on each iteration, the overhead of managing the loop, which is hidden in the loop header, runs an instruction that requires constant time. Thus, the amount of abstract work performed by this algorithm is $3n + 1$. Although this number is greater than just n , the running times for the two amounts of work, n and $3n + 1$, increase at the same rate.

11.2

Exercises

- 1 Assume that each of the following expressions indicates the number of operations performed by an algorithm for a problem size of n . Point out the dominant term of each algorithm, and use big-O notation to classify it.
 - a $2^n - 4n^2 + 5n$
 - b $3n^2 + 6$
 - c $n^3 + n^2 - n$
- 2 For problem size n , algorithms A and B perform n^2 and $\frac{1}{2}n^2 + \frac{1}{2}n$ instructions, respectively. Which algorithm does more work? Are there particular problem sizes for which one algorithm performs significantly better than the other? Are there particular problem sizes for which both algorithms perform approximately the same amount of work?
- 3 At what point does an n^4 algorithm begin to perform better than a 2^n algorithm?

11.3

Search Algorithms

We now present several algorithms that can be used for searching and sorting lists. We first discuss the design of an algorithm, we then show its implementation as a Python function, and, finally, we provide an analysis of the algorithm's computational complexity. To keep things simple, each function processes a list of integers. Lists of different sizes can be passed as parameters to the functions. The functions are defined in a single module that is used in the case study later in this chapter.

11.3.1 Search for a Minimum

Python's **min** function returns the minimum or smallest item in a list. To study the complexity of this algorithm, let's develop an alternative version that returns the *position* of the minimum item. The algorithm assumes that the list is not empty and that the items are in arbitrary order. The algorithm begins by treating the first position as that of the minimum item. It then searches to the right for an item that is smaller and, if it is found, resets the position of the minimum item to the current position. When the algorithm reaches the end of the list, it returns the position of the minimum item. Here is the code for the algorithm, in function **ourMin**:

```
def ourMin(lyst):
    """Returns the position of the minimum item."""
    minpos = 0
    current = 1
    while current < len(lyst):
        if lyst[current] < lyst[minpos]:
            minpos = current
        current += 1
    return minpos
```

As you can see, there are three instructions outside the loop that execute the same number of times regardless of the size of the list. Thus, we can discount them. Within the loop, we find three more instructions. Of these, the comparison in the **if** statement and the increment of **current** execute on each pass through the loop. There are no nested or hidden loops in these instructions. This algorithm must visit every item in the list to guarantee that it has located the position of the minimum item. Thus, the algorithm must make $n - 1$ comparisons for a list of size n . Therefore, the algorithm's complexity is $O(n)$.

11.3.2 Linear Search of a List

Python's **in** operator is implemented as a method named **__contains__** in the **list** class. This method searches for a particular item (called the target item) within a list of arbitrarily arranged items. In such a list, the only way to search for a target item is to begin with the item at the first position and compare it to the target. If the items are equal, the method returns **True**. Otherwise, the method moves on to the next position and compares items again. If the method arrives at the last position and still cannot find the target, it returns **False**. This kind of

search is called a **sequential search** or a **linear search**. A more useful linear search function would return the index of a target if it's found, or -1 otherwise. Here is the Python code for a linear search function:

```
def linearSearch(target, lyst):  
    """Returns the position of the target item if found,  
    or -1 otherwise."""  
    position = 0  
    while position < len(lyst):  
        if target == lyst[position]:  
            return position  
        position += 1  
    return -1
```

The analysis of a linear search is a bit different from the analysis of a search for a minimum, as we shall see in the next subsection.

11.3.3 Best-Case, Worst-Case, and Average-Case Performance

The performance of some algorithms depends on the placement of the data that are processed. The linear search algorithm does less work to find a target at the beginning of a list than at the end of the list. For such algorithms, one can determine the best-case performance, the worst-case performance, and the average performance. In general, we worry more about average and worst-case performances than about best-case performances.

Our analysis of a linear search considers three cases:

- 1 In the worst case, the target item is at the end of the list or not in the list at all. Then the algorithm must visit every item and perform n iterations for a list of size n . Thus, the worst-case complexity of a linear search is $O(n)$.
- 2 In the best case, the algorithm finds the target at the first position, after making one iteration, for an $O(1)$ complexity.
- 3 To determine the average case, you add the number of iterations required to find the target at each possible position and divide the sum by n . Thus, the algorithm performs $(n + n - 1 + n - 2 + \dots + 1) / n$, or $(n + 1) / 2$ iterations. For very large n , the constant factor of $/2$ is insignificant, so the average complexity is still $O(n)$.

Clearly, the best-case performance of a linear search is rare when compared with the average and worst-case performances, which are essentially the same.

11.3.4 Binary Search of a List

A linear search is necessary for data that are not arranged in any particular order. When searching sorted data, you can use a binary search.

To understand how a binary search works, think about what happens when you look up a person's number in a phone book. The data in a phone book are already sorted, so you don't do a linear search. Instead, you estimate the name's alphabetical position in the book, and open the book as close to that position as possible. After you open the book, you determine if the target name lies, alphabetically, on an earlier page or later page, and flip back or forward through the pages as necessary. You repeat this process until you find the name or conclude that it's not in the book.

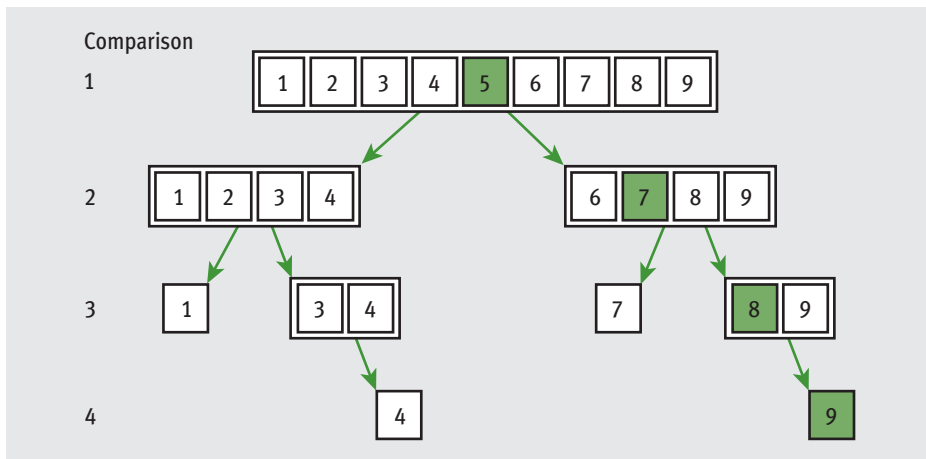
Now let's consider an example of a binary search in Python. To begin, let's assume that the items in the list are sorted in ascending order (as they are in a phone book). The search algorithm goes directly to the middle position in the list and compares the item at that position to the target. If there is a match, the algorithm returns the position. Otherwise, if the target is less than the current item, the algorithm searches the portion of the list before the middle position. If the target is greater than the current item, the algorithm searches the portion of the list after the middle position. The search process stops when the target is found or the current beginning position is greater than the current ending position.

Here is the code for the binary search function:

```
def binarySearch(target, lyst):
    left = 0
    right = len(lyst) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if target == lyst[midpoint]:
            return midpoint
        elif target < lyst[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```


There is just one loop with no nested or hidden loops. Once again, the worst case occurs when the target is not in the list. How many times does the loop run in the worst case? This is equal to the number of times the size of the list can be divided by 2 until the quotient is 1. For a list of size n , you essentially perform the reduction $n / 2 / 2 \dots / 2$ until the result is 1. Let k be the number of times we divide n by 2. To solve for k , you have $n / 2^k = 1$, and $n = 2^k$, and $k = \log_2 n$. Thus, the worst-case complexity of binary search is $O(\log_2 n)$.

Figure 11.7 shows the portions of the list being searched in a binary search with a list of 9 items and a target item, 10, that is not in the list. The items compared to the target are shaded. Note that none of the items in the left half of the original list are visited.



[FIGURE 11.7] The items of a list visited during a binary search for 10

The binary search for the target item 10 requires four comparisons, whereas a linear search would have required 10 comparisons. This algorithm actually appears to perform better as the problem size gets larger. Our list of 9 items requires at most 4 comparisons, whereas a list of 1,000,000 items requires at most only 20 comparisons!

Binary search is certainly more efficient than linear search. However, the kind of search algorithm we choose depends on the organization of the data in the list. There is some additional overall cost to a binary search which has to do with keeping the list in sorted order. In a moment, we examine several strategies for sorting a list and analyze their complexity. But first, we provide a few words about comparing data items.

11.3.5 Comparing Data Items

Both the binary search and the search for the minimum assume that the items in the list are comparable with each other. In Python, this means that the items are of the same type and that they recognize the comparison operators `==`, `<`, and `>`. Objects of several built-in Python types, such as numbers, strings, and lists, can be compared using these operators.

To allow algorithms to use the comparison operators `==`, `<`, and `>` with a new class of objects, the programmer should define the `__eq__`, `__lt__`, and `__gt__` methods in that class. The header of `__lt__` is the following:

```
def __lt__(self, other):
```

This method returns **True** if **self** is less than **other**, or **False** otherwise. The criteria for comparing the objects depend on their internal structure and on the manner in which they should be ordered.

For example, the **SavingsAccount** objects discussed in Chapter 8 include three data fields, for a name, a PIN, and a balance. If we assume that the accounts should be ordered alphabetically by name, then the following implementation of the `__lt__` method is called for:

```
class SavingsAccount(object):
    """This class represents a savings account
    with the owner's name, PIN, and balance."""

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def __lt__(self, other):
        return self._name < other._name

    # Other methods
```

Note that the `__lt__` method calls the `<` operator with the `_name` fields of the two account objects. The names are strings, and the string type includes the `__lt__` method as well. Python automatically runs the `__lt__` method when the `<` operator is applied in the same way as it runs the `__str__` method when the `str` function is called.

The next session shows a test of comparisons with several account objects:

```
>>> s1 = SavingsAccount("Ken", "1000", 0)
>>> s2 = SavingsAccount("Bill", "1001", 30)
>>> s1 < s2
False
>>> s2 < s1
True
>>> s1 > s2
True
>>> s2 > s1
False
>>> s2 == s1
False
>>> s3 = SavingsAccount("Ken", "1000", 0)
>>> s1 == s3
True
>>> s4 = s1
>>> s4 == s1
True
```

The accounts can now be placed in a list and sorted by name.

11.3 Exercises

- 1 Suppose that a list contains the values
20 44 48 55 62 66 74 88 93 99
at index positions 0 through 9. Trace the values of the variables **left**, **right**, and **midpoint** in a binary search of this list for the target value 90. Repeat for the target value 44.
- 2 The method we usually use to look up an entry in a phone book is not exactly the same as a binary search because, when using a phone book, we don't always go to the midpoint of the sublist being searched. Instead, we estimate the position of the target based on the alphabetical position of the first letter of the person's last name. For example, when we are looking up a number for "Smith," we look toward the middle of the second half of the phone book first, instead of in the middle of the entire book. Suggest a modification of the binary search algorithm that emulates this strategy for a list of names. Is its computational complexity any better than that of the standard binary search?

11.4 Sort Algorithms

Computer scientists have devised many ingenious strategies for sorting a list of items. We won't consider all of them here. In this chapter, we examine some algorithms that are easy to write but are inefficient. Each of the Python sort functions that we develop here operates on a list of integers and uses a **swap** function to exchange the positions of two items in the list. Here is the code for that function:

```
def swap(lyst, i, j):  
    """Exchanges the items at positions i and j."""  
    # You could say lyst[i], lyst[j] = lyst[j], lyst[i]  
    # but the following code shows what is really going on  
    temp = lyst[i]  
    lyst[i] = lyst[j]  
    lyst[j] = temp
```

11.4.1 Selection Sort

Perhaps the simplest strategy is to search the entire list for the position of the smallest item. If that position does not equal the first position, the algorithm swaps the items at those positions. It then returns to the second position and repeats this process, swapping the smallest item with the item at the second position, if necessary. When the algorithm reaches the last position in this overall process, the list is sorted. The algorithm is called **selection sort** because each pass through the main loop selects a single item to be moved. Table 11.3 shows the states of a list of five items after each search and swap pass of selection sort. The two items just swapped on each pass have asterisks next to them, and the sorted portion of the list is shaded.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

[TABLE 11.3] A trace of the data during a selection sort

Here is the Python function for a selection sort:

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:           # Do n - 1 searches
        minIndex = i                 # for the smallest
        j = i + 1
        while j < len(lyst):         # Start a search
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:            # Exchange if needed
            swap(lyst, minIndex, i)
        i += 1
```

This function includes a nested loop. For a list of size n , the outer loop executes $n - 1$ times. On the first pass through the outer loop, the inner loop executes $n - 1$ times. On the second pass through the outer loop, the inner loop executes $n - 2$ times. On the last pass through the outer loop, the inner loop executes once. Thus, the total number of comparisons for a list of size n is the following:

$$(n - 1) + (n - 2) + \dots + 1 =$$

$$n(n - 1) / 2 =$$

$$\frac{1}{2} n^2 - \frac{1}{2} n$$

For large n , you can pick the term with the largest degree and drop the coefficient, so selection sort is $O(n^2)$ in all cases. For large data sets, the cost of swapping items might also be significant. Because data items are swapped only in the outer loop, this additional cost for selection sort is linear in the worst and average cases.

11.4.2 Bubble Sort

Another sort algorithm that is relatively easy to conceive and code is called a **bubble sort**. Its strategy is to start at the beginning of the list and compare pairs of data items as it moves down to the end. Each time the items in the pair are out of order, the algorithm swaps them. This process has the effect of bubbling the largest items to the end of the list. The algorithm then repeats the process from the beginning of the list and goes to the next-to-last item, and so on, until it begins with the last item. At that point, the list is sorted.

Table 11.4 shows a trace of the bubbling process through a list of five items. This process makes four passes through a nested loop to bubble the largest item down to the end of the list. Once again, the items just swapped are marked with asterisks, and the sorted portion is shaded.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

[TABLE 11.4] A trace of the data during a bubble sort

Here is the Python function for a bubble sort:

```
def bubbleSort(lyst):
    n = len(lyst)
    while n > 1:
        i = 1
        while i < n:
            if lyst[i] < lyst[i - 1]: # Exchange if needed
                swap(lyst, i, i - 1)
            i += 1
        n -= 1
```

As with the selection sort, a bubble sort has a nested loop. The sorted portion of the list now grows from the end of the list up to the beginning, but the

performance of the bubble sort is quite similar to the behavior of selection sort: the inner loop executes $\frac{1}{2}n^2 - \frac{1}{2}n$ times for a list of size n . Thus, bubble sort is $O(n^2)$. Like selection sort, bubble sort won't perform any swaps if the list is already sorted. However, bubble sort's worst-case behavior for exchanges is greater than linear. The proof of this is left as an exercise for you.

You can make a minor adjustment to the bubble sort to improve its best-case performance to linear. If no swaps occur during a pass through the main loop, then the list is sorted. This can happen on any pass, and in the best case will happen on the first pass. You can track the presence of swapping with a Boolean flag and return from the function when the inner loop does not set this flag. Here is the modified bubble sort function:

```
def bubbleSort2(lyst):
    n = len(lyst)
    while n > 1:
        swapped = False
        i = 1
        while i < n:
            if lyst[i] < lyst[i - 1]: # Exchange if needed
                swap(lyst, i, i - 1)
                swapped = True
            i += 1
        if not swapped: return # Return if no swaps
        n -= 1
```

Note that this modification only improves best-case behavior. On the average, the behavior of bubble sort is still $O(n^2)$.

11.4.3 Insertion Sort

Our modified bubble sort performs better than a selection sort for lists that are already sorted. But our modified bubble sort can still perform poorly if many items are out of order in the list. Another algorithm, called an **insertion sort**, attempts to exploit the partial ordering of the list in a different way. The strategy is as follows:

- On the i th pass through the list, where i ranges from 1 to $n - 1$, the i th item should be inserted into its proper place among the first i items in the list.
- After the i th pass, the first i items should be in sorted order.

- This process is analogous to the way in which many people organize playing cards in their hands. That is, if you hold the first $i - 1$ cards in order, you pick the i th card and compare it to these cards until its proper spot is found.
- As with our other sort algorithms, insertion sort consists of two loops. The outer loop traverses the positions from 1 to $n - 1$. For each position i in this loop, you save the item and start the inner loop at position $i - 1$. For each position j in this loop, you move the item to position $j + 1$ until you find the insertion point for the saved (i th) item.

Here is the code for the **insertionSort** function:

```
def insertionSort(lyst):
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j -= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```

Table 11.5 shows the states of a list of five items after each pass through the outer loop of an insertion sort. The item to be inserted on the next pass is marked with an arrow; after it is inserted, this item is marked with an asterisk.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

[TABLE 11.5] A trace of the data during an insertion sort

Once again, analysis focuses on the nested loop. The outer loop executes $n - 1$ times. In the worst case, when all of the data are out of order, the inner loop iterates once on the first pass through the outer loop, twice on the second pass, and so on, for a total of $\frac{1}{2} n^2 - \frac{1}{2} n$ times. Thus, the worst-case behavior of insertion sort is $O(n^2)$.

The more items in the list that are in order, the better insertion sort gets until, in the best case of a sorted list, the sort's behavior is linear. In the average case, however, insertion sort is still quadratic.

11.4.4

Best-Case, Worst-Case, and Average-Case Performance Revisited

As mentioned earlier, for many algorithms, a single measure of complexity cannot be applied to all cases. Sometimes an algorithm's behavior improves or gets worse when it encounters a particular arrangement of data. For example, the bubble sort algorithm can terminate as soon as the list becomes sorted. If the input list is already sorted, the bubble sort requires approximately n comparisons. In many other cases, however, bubble sort requires approximately n^2 comparisons. Clearly, a more detailed analysis may be needed to make programmers aware of these special cases.

As we discussed earlier, thorough analysis of an algorithm's complexity divides its behavior into three types of cases:

- 1 **Best case**—Under what circumstances does an algorithm do the least amount of work? What is the algorithm's complexity in this best case?
- 2 **Worst case**—Under what circumstances does an algorithm do the most amount of work? What is the algorithm's complexity in this worst case?
- 3 **Average case**—Under what circumstances does an algorithm do a typical amount of work? What is the algorithm's complexity in this typical case?

Let's review three examples of this kind of analysis for a search for a minimum, linear search, and bubble sort.

Because the search for a minimum algorithm must visit each number in the list, unless it is sorted, the algorithm is always linear. Therefore, its best-case, worst-case, and average-case performances are $O(n)$.

Linear search is a bit different. The algorithm stops and returns a result as soon as it finds the target item. Clearly, in the best case, the target element is in the first position. In the worst case, the target is in the last position. Therefore, the algorithm's best-case performance is $O(1)$, and its worst-case performance is $O(n)$. To compute the average-case performance, we add up all of the comparisons that must be made to locate a target in each position and divide by n . This is $(1 + 2 + \dots + n) / n$, or $n / 2$. Therefore, by approximation, the average-case performance of linear search is also $O(n)$.

The smarter version of bubble sort can terminate as soon as the list becomes sorted. In the best case, this happens when the input list is already sorted. Therefore, bubble sort's best-case performance is $O(n)$. However, this case is rare (1 out of $n!$). In the worst case, even this version of bubble sort will have to bubble each item down to its proper position in the list. The algorithm's worst-case performance is clearly $O(n^2)$. Bubble sort's average-case performance is closer to $O(n^2)$ than to $O(n)$, although the demonstration of this fact is a bit more involved than it is for linear search.

As we will see, there are algorithms whose best-case and average-case performances are similar, but whose performance can degrade to a worst case. Whether you are choosing an algorithm or developing a new one, it is important to be aware of these distinctions.

11.4 Exercises

- 1 Which configuration of data in a list causes the smallest number of exchanges in a selection sort? Which configuration of data causes the largest number of exchanges?
- 2 Explain the role that the number of data exchanges plays in the analysis of selection sort and bubble sort. What role, if any, does the size of the data objects play?
- 3 Explain why the modified bubble sort still exhibits $O(n^2)$ behavior on the average.
- 4 Explain why insertion sort works well on partially sorted lists.

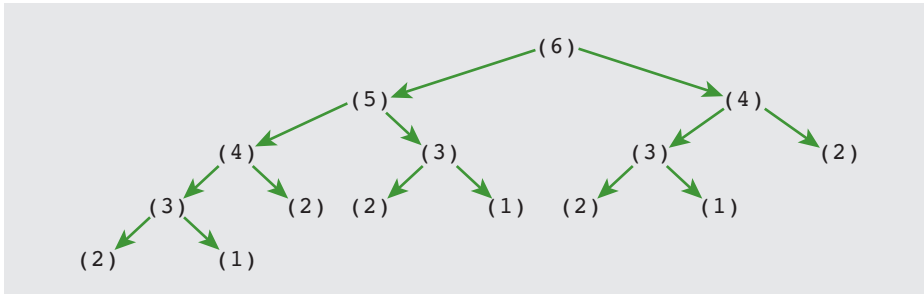
11.5

An Exponential Algorithm: Recursive Fibonacci

Earlier in this chapter, we ran the recursive Fibonacci function to obtain a count of the recursive calls with various problem sizes. You saw that the number of calls seemed to grow much faster than the square of the problem size. Here is the code for the function once again:

```
def fib(n):
    """The recursive Fibonacci function."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Another way to illustrate this rapid growth of work is to display a **call tree** for the function for a given problem size. Figure 11.8 shows the calls involved when we use the recursive function to compute the sixth Fibonacci number. To keep the diagram reasonably compact, we write **(6)** instead of **fib(6)**.



[FIGURE 11.8] A call tree for **fib(6)**

Note that **fib(4)** requires only 4 recursive calls, which seems linear, but **fib(6)** requires 2 calls of **fib(4)**, among a total of 14 recursive calls. Indeed, it gets much worse as the problem size grows, with possibly many repetitions of the same subtrees in the call tree.

Exactly how bad is this behavior, then? If the call tree were fully balanced, with the bottom two levels of calls completely filled in, a call with an argument of 6 would generate $2 + 4 + 8 + 16 = 30$ recursive calls. Note that the number of

calls at each filled level is twice that of the level above it. Thus, the number of recursive calls generally is $2^{n+1} - 2$ in fully balanced call trees, where n is the argument at the top or root of the call tree. This is clearly the behavior of an exponential, $O(k^n)$ algorithm. Although the bottom two levels of the call tree for recursive Fibonacci are not completely filled in, its call tree is close enough in shape to a fully balanced tree to rank recursive Fibonacci as an exponential algorithm. The constant k for recursive Fibonacci is approximately 1.63.

Exponential algorithms are generally impractical to run with any but very small problem sizes. Although recursive Fibonacci is elegant in its design, there is a less beautiful but much faster version that uses a loop to run in linear time (see the next section).

Alternatively, recursive functions that are called repeatedly with the same arguments, such as the Fibonacci function, can be made more efficient by a technique called **memoization**. According to this technique, the program maintains a table of the values for each argument used with the function. Before the function recursively computes a value for a given argument, it checks the table to see if that argument already has a value. If so, that value is simply returned. If not, the computation proceeds, and the argument and value are added to the table afterward.

Computer scientists devote much effort to the development of fast algorithms. As a rule, any reduction in the order of magnitude of complexity, say, from $O(n^2)$ to $O(n)$, is preferable to a “tweak” of code that reduces the constant of proportionality.

11.6 Converting Fibonacci to a Linear Algorithm

Although the recursive Fibonacci function reflects the simplicity and elegance of the recursive definition of the Fibonacci sequence, the run-time performance of this function is unacceptable. A different algorithm improves on this performance by several orders of magnitude and, in fact, reduces the complexity to linear time. In this section, we develop this alternative algorithm and assess its performance.

Recall that the first two numbers in the Fibonacci sequence are 1s, and each number after that is the sum of the previous two numbers. Thus, the new algorithm starts a loop if n is at least the third Fibonacci number. This number will be at least the sum of the first two ($1 + 1 = 2$). The loop computes this sum and then performs two replacements: the first number becomes the second one, and the second one becomes the sum just computed. The loop counts from 3 through n .

The sum at the end of the loop is the n th Fibonacci number. Here is the pseudocode for this algorithm:

```
Set sum to 1
Set first to 1
Set second to 1
Set count to 3
While count <= N
    Set sum to first + second
    Set first to second
    Set second to sum
    Increment count
```

The Python function **fib** now uses a loop. The function can be tested within the script used for the earlier version. Here is the code for the function, followed by the output of the script:

```
def fib(n, counter):
    """Count the number of iterations in the Fibonacci
    function."""
    sum = 1
    first = 1
    second = 1
    count = 3
    while count <= n:
        counter.increment()
        sum = first + second
        first = second
        second = sum
        count += 1
    return sum
```

Problem Size	Iterations
2	0
4	2
8	6
16	14
32	30

As you can see, the performance of the new version of the function has improved to linear. Removing recursion by converting a recursive algorithm to one based on a loop can often, but not always, reduce its run-time complexity.

11.7

Case Study: An Algorithm Profiler

Profiling is the process of measuring an algorithm's performance, by counting instructions and/or timing execution. In this case study, we develop a program to profile sort algorithms.

11.7.1

Request

Write a program that allows a programmer to profile different sort algorithms.

11.7.2

Analysis

The profiler should allow a programmer to run a sort algorithm on a list of numbers. The profiler can track the algorithm's running time, the number of comparisons, and the number of exchanges. In addition, when the algorithm exchanges two values, the profiler can print a trace of the list. The programmer can provide her own list of numbers to the profiler or ask the profiler to generate a list of randomly ordered numbers of a given size. The programmer can also ask for a list of unique numbers or a list that contains duplicate values. For ease of use, the profiler allows the programmer to specify most of these features as options before the algorithm is run. The default behavior is to run the algorithm on a randomly ordered list of 10 unique numbers where the running time, comparisons, and exchanges are tracked.

The profiler is an instance of the class **Profiler**. The programmer profiles a sort function by running the profiler's **test** method with the function as the first argument and any of the options mentioned earlier. The next session shows several test runs of the profiler with the selection sort algorithm and different options:

```
>>> from profiler import Profiler
>>> from algorithms import selectionSort

>>> p = Profiler()

>>> p.test(selectionSort)      # Default behavior
Problem size: 10
Elapsed time: 0.0
Comparisons: 45
Exchanges: 7
```

continued

```
>>> p.test(selectionSort, size = 5, trace = True)
[4, 2, 3, 5, 1]
[1, 2, 3, 5, 4]
Problem size: 5
Elapsed time: 0.117
Comparisons: 10
Exchanges: 2
```

```
>>> p.test(selectionSort, size = 100)
Problem size: 100
Elapsed time: 0.044
Comparisons: 4950
Exchanges: 97
```

```
>>> p.test(selectionSort, size = 1000)
Problem size: 1000
Elapsed time: 1.628
Comparisons: 499500
Exchanges: 995
```

```
>>> p.test(selectionSort, size = 10000,
            exch = False, comp = False)
Problem size: 10000
Elapsed time: 111.077
```

The programmer configures a sort algorithm to be profiled as follows:

- 1 Define a sort function and include a second parameter, a **Profiler** object, in the sort function's header.
- 2 In the sort algorithm's code, run the methods **comparison()** and **exchange()** with the **Profiler** object where relevant, to count comparisons and exchanges.

The interface for the **Profiler** class is listed in Table 11.6.

Profiler METHOD	WHAT IT DOES
<code>p.test(function, lyst = None, size = 10, unique = True, comp = True, exch = True, trace = False)</code>	Runs function with the given settings and prints the results.
<code>p.comparison()</code>	Increments the number of comparisons if that option has been specified.
<code>p.exchange()</code>	Increments the number of exchanges if that option has been specified.
<code>p.__str__()</code>	Returns a string representation of the results, depending on the options.

[TABLE 11.6] The interface for the **Profiler** class

11.7.3 Design

The programmer uses two modules:

- 1 **profiler**—This module defines the **Profiler** class.
- 2 **algorithms**—This module defines the sort functions, as configured for profiling.

The sort functions have the same design as those discussed earlier in this chapter, except that they receive a **Profiler** object as an additional parameter. The **Profiler** methods **comparison** and **exchange** are run with this object whenever a sort function performs a comparison or an exchange of data values, respectively. In fact, any list-processing algorithm can be added to this module and profiled just by including a **Profiler** parameter and running its two methods when comparisons and/or exchanges are made.

As shown in the earlier session, one imports the **Profiler** class and the **algorithms** module into a Python shell and performs the testing at the shell prompt. The profiler's **test** method sets up the **Profiler** object, runs the function to be profiled, and prints the results.

11.7.4 Implementation (Coding)

Here is a partial implementation of the **algorithms** module. We omit most of the sort algorithms developed earlier in this chapter, but include one, **selectionSort**, to show how the statistics are updated.

```
"""
File: algorithms.py
Algorithms configured for profiling.
"""

def selectionSort(lyst, profiler):
    i = 0
    while i < len(lyst) - 1:
        minIndex = i
        j = i + 1
        while j < len(lyst):
            profiler.comparison()          # Count
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:
            swap(lyst, minIndex, i, profiler)
        i += 1

def swap(lyst, i, j, profiler):
    """Exchanges the elements at positions i and j."""
    profiler.exchange()                    # Count
    temp = lyst[i]
    lyst[i] = lyst[j]
    lyst[j] = temp

# Testing code can go here, optionally
```

The **Profiler** class includes the four methods listed in the interface as well as some helper methods for managing the clock.

```

"""
File: profiler.py

Defines a class for profiling sort algorithms.
A Profiler object tracks the list, the number of comparisons
and exchanges, and the running time. The Profiler can also
print a trace and can create a list of unique or duplicate
numbers.

Example use:

from profiler import Profiler
from algorithms import selectionSort

p = Profiler()
p.test(selectionSort, size = 15, comp = True,
        exch = True, trace = True)
"""

import time
import random

class Profiler(object):

    def test(self, function, lyst = None, size = 10,
            unique = True, comp = True, exch = True,
            trace = False):
        """
        function: the algorithm being profiled
        target: the search target if profiling a search
        lyst: allows the caller to use her list
        size: the size of the list, 10 by default
        unique: if True, list contains unique integers
        comp: if True, count comparisons
        exch: if True, count exchanges
        trace: if True, print the list after each exchange

        Run the function with the given attributes and print
        its profile results.
        """
        self._comp = comp
        self._exch = exch
        self._trace = trace
        if lyst != None:
            self._lyst = lyst
        elif unique:
            self._lyst = list(range(1, size + 1))
            random.shuffle(self._lyst)

```

continued

```

else:
    self._lyst = []
    for count in range(size):
        self._lyst.append(random.randint(1, size))
    self._exchCount = 0
    self._cmpCount = 0
    self._startClock()
    function(self._lyst, self)
    self._stopClock()
    print(self)

def exchange(self):
    """Counts exchanges if on."""
    if self._exch:
        self._exchCount += 1
    if self._trace:
        print(self._lyst)

def comparison(self):
    """Counts comparisons if on."""
    if self._comp:
        self._cmpCount += 1

def _startClock(self):
    """Record the starting time."""
    self._start = time.time()

def _stopClock(self):
    """Stops the clock and computes the elapsed time
    in seconds, to the nearest millisecond."""
    self._elapsedTime = round(time.time() - self._start, 3)

def __str__(self):
    """Returns the results as a string."""
    result = "Problem size: "
    result += str(len(self._lyst)) + "\n"
    result += "Elapsed time: "
    result += str(self._elapsedTime) + "\n"
    if self._comp:
        result += "Comparisons: "
        result += str(self._cmpCount) + "\n"
    if self._exch:
        result += "Exchanges: "
        result += str(self._exchCount) + "\n"
    return result

```

Summary

- Different algorithms for solving the same problem can be ranked according to the time and memory resources that they require. Generally, algorithms that require less running time and less memory are considered better than those that require more of these resources. However, there is often a tradeoff between the two types of resources. Running time can occasionally be improved at the cost of using more memory, or memory usage can be improved at the cost of slower running times.
- The running time of an algorithm can be measured empirically using the computer's clock. However, these times will vary with the hardware and the types of programming language used.
- Counting instructions provides another empirical measurement of the amount of work that an algorithm does. Instruction counts can show increases or decreases in the rate of growth of an algorithm's work, independently of hardware and software platforms.
- The rate of growth of an algorithm's work can be expressed as a function of the size of its problem instances. Complexity analysis examines the algorithm's code to derive these expressions. Such an expression enables the programmer to predict how well or poorly an algorithm will perform on any computer.
- Big-O notation is a common way of expressing an algorithm's run-time behavior. This notation uses the form $O(f(n))$, where n is the size of the algorithm's problem and $f(n)$ is a function expressing the amount of work done to solve it.
- Common expressions of run-time behavior are $O(\log_2 n)$ (logarithmic), $O(n)$ (linear), $O(n^2)$ (quadratic), and $O(k^n)$ (exponential).
- An algorithm can have different best-case, worst-case, and average-case behaviors. For example, bubble sort and insertion sort are linear in the best case, but quadratic in the average and worst cases.
- In general, it is better to try to reduce the order of an algorithm's complexity than it is to try to enhance performance by tweaking the code.
- A binary search is substantially faster than a linear search. However, the data in the search space for a binary search must be in sorted order.
- Exponential algorithms are primarily of theoretical interest and are impractical to run with large problem sizes.

REVIEW QUESTIONS

- 1 Timing an algorithm with different problem sizes
 - a can give you a general idea of the algorithm's run-time behavior
 - b can give you an idea of the algorithm's run-time behavior on a particular hardware platform and a particular software platform
- 2 Counting instructions
 - a provides the same data on different hardware and software platforms
 - b can demonstrate the impracticality of exponential algorithms with large problem sizes
- 3 The expressions $O(n)$, $O(n^2)$, and $O(k^n)$ are, respectively,
 - a exponential, linear, and quadratic
 - b linear, quadratic, and exponential
 - c logarithmic, linear, and quadratic
- 4 A binary search
 - a assumes that the data are arranged in no particular order
 - b assumes that the data are sorted
- 5 A selection sort makes at most
 - a n^2 exchanges of data items
 - b n exchanges of data items
- 6 The best-case behavior of insertion sort and modified bubble sort is
 - a linear
 - b quadratic
 - c exponential
- 7 An example of an algorithm whose best-case, average-case, and worst-case behaviors are the same is
 - a linear search
 - b insertion sort
 - c selection sort
- 8 Generally speaking, it is better
 - a to tweak an algorithm to shave a few seconds of running time
 - b to choose an algorithm with the lowest order of computational complexity

- 9 The recursive Fibonacci function makes approximately
 - a n^2 recursive calls for problems of a large size n
 - b 2^n recursive calls for problems of a large size n
- 10 Each level in a completely filled binary call tree has
 - a twice as many calls as the level above it
 - b the same number of calls as the level above it

PROJECTS

- 1 A linear search of a sorted list can halt when the target is less than a given element in the list. Define a modified version of this algorithm, and state the computational complexity, using big-O notation, of its best-, worst-, and average-case performances.
- 2 The list method **reverse** reverses the elements in the list. Define a function named **reverse** that reverses the elements in its list argument (without using the method **reverse**!). Try to make this function as efficient as possible, and state its computational complexity using big-O notation.
- 3 Python's **pow** function returns the result of raising a number to a given power. Define a function **expo** that performs this task, and state its computational complexity using big-O notation. The first argument of this function is the number, and the second argument is the exponent (non-negative numbers only). You may use either a loop or a recursive function in your implementation.
- 4 An alternative strategy for the **expo** function uses the following recursive definition:

```
expo(number, exponent)
= 1, when exponent = 0
= number * expo(number, exponent - 1), when exponent is odd
= (expo(number, exponent // 2))2, when exponent is even
```

Define a recursive function **expo** that uses this strategy, and state its computational complexity using big-O notation.

- 5 Python's **list** method **sort** includes the keyword argument **reverse**, whose default value is **False**. The programmer can override this value to sort a list in descending order. Modify the **selectionSort** function discussed in this chapter so that it allows the programmer to supply this additional argument to redirect the sort.
- 6 Modify the recursive Fibonacci function to employ the memoization technique discussed in this chapter. The function should expect a dictionary as an additional argument. The top-level call of the function receives an empty dictionary. The function's keys and values should be the arguments and values of the recursive calls. Also use the **Counter** object discussed in this chapter to count the number of recursive calls.
- 7 Profile the performance of the memoized version of the Fibonacci function defined in Project 6. The function should count the number of recursive calls. State its computational complexity using big-O notation, and justify your answer.
- 8 The function **makeRandomList** creates and returns a list of numbers of a given size (its argument). The numbers in the list are unique and range from 1 through the size. They are placed in random order. Here is the code for the function:

```
def makeRandomList(size):  
    lyst = []  
    for count in range(size):  
        while True:  
            number = random.randint(1, size)  
            if not number in lyst:  
                lyst.append(number)  
                break  
    return lyst
```

You may assume that **range**, **randint**, and **append** are constant time functions. You may also assume that **random.randint** more rarely returns duplicate numbers as the range between its arguments increases. State the computational complexity of this function using big-O notation, and justify your answer.

- 9 As discussed in Chapter 6, a computer supports the calls of recursive functions using a structure called the call stack. Generally speaking, the computer reserves a constant amount of memory for each call of a function. Thus, the memory used by a recursive function can be subjected to complexity analysis. State the computational complexity of the memory used by the recursive factorial and Fibonacci functions, as defined in Chapter 6.

- 10 The function that draws c-curves, and which was discussed in Chapter 7, has two recursive calls. Here is the code:

```
def cCurve(t, x1, y1, x2, y2, level):  
  
    def drawLine(x1, y1, x2, y2):  
        """Draws a line segment between the endpoints."""  
        t.up()  
        t.goto(x1, y1)  
        t.down()  
        t.goto(x2, y2)  
  
    if level == 0:  
        drawLine(x1, y1, x2, y2)  
    else:  
        xm = (x1 + x2 + y1 - y2) // 2  
        ym = (x2 + y1 + y2 - x1) // 2  
        cCurve(t, x1, y1, xm, ym, level - 1)  
        cCurve(t, xm, ym, x2, y2, level - 1)
```

You can assume that the function **drawLine** runs in constant time. State the computational complexity of the **cCurve** function, in terms of the level, using big-O notation. Also, draw a call tree for a call of this function with a level of 3.