CHAPTER 16

Developing Efficiency Algorithms

Objectives

- To estimate algorithm efficiency using the Big O notation (§16.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§16.2).
- To determine the complexity of various types of algorithms (§16.3).
- To analyze the binary search algorithm (§16.4.1).
- To analyze the selection sort algorithm (§16.4.2).
- To analyze the insertion sort algorithm (§16.4.3).
- To analyze the Towers of Hanoi algorithm (§16.4.4).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, exponential) (§16.4.5).
- To design efficient algorithms for finding Fibonacci numbers (§16.5).
- To design efficient algorithms for finding gcd (§16.6).
- To design efficient algorithms for finding prime numbers (§16.7).
- To design efficient algorithms for finding a closest pair of points (§16.8).
- To solve the Eight Queens problem using the backtracking approach (§16.9).
- To design efficient algorithms for finding a convex hull for a set of points (§16.10).
16.1 Introduction

Key Point: Algorithm analysis is to predict the performance of the algorithm.

Suppose two algorithms perform the same task, such as search (linear search vs. binary search) or sort (selection sort vs. insertion sort). Which one is better? To answer this question, we might implement these algorithms in and run the programs to get execution time. But there are two problems with this approach:

First, many tasks run concurrently on a computer. The execution time of a particular program depends on the system load.

Second, the execution time depends on specific input. Consider, for example, linear search and binary search. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm’s execution time increases as the input size increases, so you can compare two algorithms by examining their growth rates.

16.2 Big O Notation

Key Point: The big O notation obtains a function for measuring algorithm time complexity based on the input size. You can ignore multiplicative constants and non-dominating terms in the function.

Consider linear search. The linear search algorithm compares the key with the elements in the list sequentially until the key is found or the list is exhausted. If the key is not in the list, it requires $n$ comparisons for a list of size $n$. If the key is in the list, it requires $n/2$ comparisons on average. The algorithm’s execution time is proportional to
the size of the list. If you double the size of the list, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of \( n \). Computer scientists use the Big \( O \) notation to represent “order of magnitude.” Using this notation, the complexity of the linear search algorithm is \( O(n) \), pronounced as “order of \( n \).”

For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input, and an input that results in the longest execution time is the worst-case input. Best case and worst case are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case. An average-case analysis attempts to determine the average amount of time among all possible inputs of the same size. Average-case analysis is ideal, but difficult to perform, because for many problems it is hard to determine the relative probabilities and distributions of various input instances. Worst-case analysis is easier to perform, so the analysis is generally conducted for the worst case.

The linear search algorithm requires \( n \) comparisons in the worst case and \( n/2 \) comparisons in the average case if you are nearly always looking for something known to be in the list. Using the Big \( O \) notation, both cases require \( O(n) \) time. The multiplicative constant \( (1/2) \) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for \( n/2 \) or \( 100n \) is the same as for \( n \), as illustrated in Table 16.1. Therefore, \( O(n) = O(n/2) = O(100n) \).

**Table 16.1**

*Growth Rates*

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>( n )</th>
<th>( n/2 )</th>
<th>( 100n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 100 )</td>
<td>100</td>
<td>50</td>
<td>10000</td>
</tr>
<tr>
<td>( 200 )</td>
<td>200</td>
<td>100</td>
<td>20000</td>
</tr>
</tbody>
</table>

\[ \frac{f(200)}{f(100)} = 2 \]
Consider the algorithm for finding the maximum number in a list of $n$ elements. To find the maximum number if $n$ is 2, it takes one comparison; if $n$ is 3, two comparisons. In general, it takes $n - 1$ comparisons to find the maximum number in a list of $n$ elements. Algorithm analysis is for large input size. If the input size is small, there is no significance in estimating an algorithm’s efficiency. As $n$ grows larger, the $n$ part in the expression $n - 1$ dominates the complexity. The Big $O$ notation allows you to ignore the nondominating part (e.g., -1 in the expression $n - 1$) and highlight the important part (e.g., $n$ in the expression $n - 1$). So, the complexity of this algorithm is $O(n)$.

The Big $O$ notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation $O(1)$. For example, a function that retrieves an element at a given index in a list takes constant time, because the time does not grow as the size of the list increases.

The following mathematical summations are often useful in algorithm analysis:

$$1 + 2 + 3 + \ldots + (n-1) + n = \frac{n(n+1)}{2}$$

$$a^0 + a^1 + a^2 + a^3 + \ldots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

### 16.3 Examples: Determining Big $O$

Key Point: This section gives several examples of determining Big $O$ for repetition, sequence, and selection statements.
Example 1

Consider the time complexity for the following loop:

```python
for i in range(n):
    k = k + 5
```

It is a constant time, $c$, for executing

$$k = k + 5$$

Since the loop is executed $n$ times, the time complexity for the loop is

$$T(n) = (a \text{ constant } c) \times n = O(n).$$

Example 2

What is the time complexity for the following loop?

```python
for i in range(n):
    for j in range(n):
        k = k + i + j
```

It is a constant time, $c$, for executing

$$k = k + i + j$$

The outer loop executes $n$ times. For each iteration in the outer loop, the inner loop is executed $n$ times. So, the time complexity for the loop is

$$T(n) = (a \text{ constant } c) \times n \times n = O(n^2)$$

An algorithm with the $O(n^2)$ time complexity is called a quadratic algorithm. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

Example 3
Consider the following loop:

```python
for i in range(n):
    for i in range(i):
        k = k + i + j
```

The outer loop executes \( n \) times. For \( i = 1, 2, ..., \) the inner loop is executed one time, two times, and \( n \) times. So, the time complexity for the loop is:

\[
T(n) = c + 2c + 3c + 4c + ... + nc
\]

\[
= cn(n + 1)/2
\]

\[
= (c/2) n^2 + (c/2)n
\]

\[
= O(n^2)
\]

Example 4

Consider the following loop:

```python
for i in range(n):
    for i in range(20):
        k = k + i + j
```

The inner loop executes 20 times, and the outer loop \( n \) times. So, the time complexity for the loop is:

\[
T(n) = 20 * c * n = O(n)
\]

Example 5

Consider the following sequences:

```python
for i in range(9):
    k = k + 4
```

```python
for i in range(n):
    for i in range(20):
        k = k + i + j
```

The first loop executes 10 times, and the second loop \( 20 * n \) times. So, the time complexity for the loop is
\[ T(n) = 10 \cdot c + 20 \cdot c \cdot n = O(n) \]

Example 6

Consider the following selection statement:

```python
if e in list:
    print(e)
else:
    for e in list:
        print(e)
```

Suppose the list contains \( n \) elements. The execution time for `e in list` is \( O(n) \). The loop in the `else` clause takes \( O(n) \) time. So, the time complexity for the entire statement is

\[ T(n) = \text{if test time} + \text{worst-case time(if clause, else clause)} = O(n) + O(n) = O(n) \]

Example 7

Consider the computation of \( a^n \) for an integer \( n \). A simple algorithm would multiply \( a \) \( n \) times, as follows:

```python
result = 1
for i in range(n):
    result *= a
```

The algorithm takes \( O(n) \) time. Without loss of generality, assume \( n = 2^k \). You can improve the algorithm using the following scheme:

```python
result = a
for i in range(k):
    result = result * result
```

The algorithm takes \( O(\log n) \) time. For an arbitrary \( n \), you can revise the algorithm and prove that the complexity is still \( O(\log n) \). (See Checkpoint Question 16.7.)
NOTE: For simplicity, since $O(\log n) = O(\log_2 n) = O(\log_a n)$, the constant base is omitted.

**Check point**

16.1 What is the order of each of the following function?

$$\frac{(n^2 + 1)^2}{n}, \frac{(n^2 + \log^2 n)^2}{n}, n^3 + 100n^2 + n, 2^n + 100n^2 + 45n, n2^n + n^2 2^n$$

16.2 Put the following growth functions in order:

$$\frac{5n^3}{4032}, 44 \log n, 10n \log n, 500, 2n^2, \frac{2^n}{45}, 3n$$

16.3 Count the number of iterations in the following loops.

(a) 
```python
count = 1
while count < 30:
    count = count * 2
```

(b) 
```python
count = 15
while count < 30:
    count = count * 3
```

(c) 
```python
count = 1
while count < n:
    count = count * 2
```

(d) 
```python
count = 15
while count < n:
    count = count * 3
```

16.4 How many stars are displayed in the following code if $n$ is 10? How many if $n$ is 20? Use the Big $O$ notation to estimate the time complexity.

```python
for i in range(0, n):
    print('*')
for k in range(0, n):
    for j in range(0, n):
        print('*')
for k in range(0, 10):
    for i in range(0, n):
        for j in range(0, n):
            print('*')
```
16.5 Use the Big $O$ notation to estimate the time complexity of the following functions:

```python
def mA(n):
    for i in range(n):
        print(random.random())

def mB(n):
    for i in range(n):
        for j in range(n):
            print(random.random())

def mC(m):
    for i in range(len(m)):
        print(m[i])
    for i in range(len(m) - 1, -1, -1):
        print(m[i])

def mD(m):
    for i in range(len(m)):
        for j in range(i):
            print(m[i] * m[j])
```

### 16.4 Analyzing Algorithm Time Complexity

**Key Point:** This section analyzes the complexity of several well-known algorithms: binary search, selection sort, insertion sort, and Tower of Hanoi.

#### 16.4.1 Analyzing Binary Search

The binary search algorithm presented in Listing 10.10, BinarySearch.py, searches a key in a sorted list. Each iteration in the algorithm contains a fixed number of operations, denoted by $c$. Let $T(n)$ denote the time complexity for a binary search on a list of $n$ elements. Without loss of generality, assume $n$ is a power of 2 and $k = \log_2 n$. Since binary search eliminates half of the input after two comparisons,

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc$$

$$= T(1) + \log n = 1 + (\log n)c$$

$$= O(\log n)$$

Ignoring constants and nondominating terms, the complexity of the binary search algorithm is $O(\log n)$. An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm*. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

#### 16.4.2 Analyzing Selection Sort
The selection sort algorithm presented in Listing 10.11, SelectionSort.py, finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it after the first, and so on until the list contains only a single number. The number of comparisons is \( n - 1 \) for the first iteration, \( n - 2 \) for the second iteration, and so on. Let \( T(n) \) denote the complexity for selection sort and \( c \) denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

\[
T(n) = (n - 1) + c + (n - 2) + c + ... + 2 + c + 1 + c
= \frac{(n - 1)(n - 1 + 1)}{2} + c(n - 1) = \frac{n^2}{2} - \frac{n}{2} + cn - c
= O(n^2)
\]

Therefore, the complexity of the selection sort algorithm is \( O(n^2) \).

### 16.4.3 Analyzing Insertion Sort

The insertion sort algorithm presented in Listing 10.12, InsertionSort.py, sorts a list of values by repeatedly inserting a new element into a sorted partial list until the whole list is sorted. At the \( k \)th iteration, to insert an element into a list of size \( k \), it may take \( k \) comparisons to find the insertion position, and \( k \) moves to insert the element. Let \( T(n) \) denote the complexity for insertion sort and \( c \) denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

\[
T(n) = (2 + c) + (2 \times 2 + c) + ... + (2 \times (n - 1) + c)
= 2(1 + 2 + ... + n - 1) + c(n - 1)
= 2\left(\frac{(n - 1)n}{2}\right) + cn - c = n^2 - n + cn - c
= O(n^2)
\]

Therefore, the complexity of the insertion sort algorithm is \( O(n^2) \). So, the selection sort and insertion sort are of the same time complexity.
16.4.4 Analyzing the Towers of Hanoi Problem

The Towers of Hanoi problem presented in Listing 15.8, TowersOfHanoi.py, recursively moves \( n \) disks from tower A to tower B with the assistance of tower C as follows:

1. Move the first \( n - 1 \) disks from A to C with the assistance of tower B.
2. Move disk \( n \) from A to B.
3. Move \( n - 1 \) disks from C to B with the assistance of tower A.

The complexity of this algorithm is measured by the number of moves. Let \( T(n) \) denote the number of moves for the algorithm to move \( n \) disks from tower A to tower B. Thus \( T(1) \) is 1. So,

\[
T(n) = T(n-1) + 1 + T(n-1) \\
= 2T(n-1) + 1 \\
= 2(2T(n-2) + 1) + 1 \\
= 2(2(2T(n-3) + 1) + 1) + 1 \\
= 2^{n-1}T(1) + 2^{n-2} + ... + 2 + 1 \\
= 2^{n-1} + 2^{n-2} + ... + 2 + 1 = 2^n - 1 = O(2^n)
\]

An algorithm with \( O(2^n) \) time complexity is called an exponential algorithm. As the input size increases, the time for the exponential algorithm grows exponentially. Exponential algorithms are not practical for large input size. Suppose the disk is moved at a rate of 1 per second. It would take \( \frac{2^{32}/(365*24*60*60)}{365*24*60*60} = 136 \) years to move 32 disks and \( \frac{2^{64}/(365*24*60*60)}{365*24*60*60} = 585 \) billion years to move 64 disks.

16.4.5 Common Recurrence Relations

Recurrence relations are a useful tool for analyzing algorithm complexity. As shown in the preceding examples, the complexity for binary search, selection sort, and the towers of Hanoi is \( T(n) = T\left(\frac{n}{2}\right) + c \),

\[
T(n) = T(n-1) + O(n), \text{ and } T(n) = 2T(n-1) + O(1), \text{ respectively. Table 16.2 summarizes the common recurrence relations.}
\]
Table 16.2

Commons Recurrence Functions

<table>
<thead>
<tr>
<th>Recurrence Relation</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>$T(n) = O(\log n)$</td>
<td>Binary search, Euclid’s GCD</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>$T(n) = O(n)$</td>
<td></td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n\log n)$</td>
<td>Merge sort (Chapter 17)</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n\log n)$</td>
<td>$T(n) = O(n\log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
<td>Selection sort, insertion sort</td>
</tr>
<tr>
<td>$T(n) = 2T(n-1) + O(1)$</td>
<td>$T(n) = O(2^n)$</td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + T(n-2) + O(1)$</td>
<td>$T(n) = O(2^n)$</td>
<td>Recursive Fibonacci algorithm</td>
</tr>
</tbody>
</table>

16.4.6 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. Table 16.3 lists some common growth functions and shows how growth rates change as the input size doubles from $n = 25$ to $n = 50$.

Table 16.3

Change of Growth Rates

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
<th>$n = 25$</th>
<th>$n = 50$</th>
<th>$f(50)/f(25)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant time</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic time</td>
<td>4.64</td>
<td>5.64</td>
<td>1.21</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear time</td>
<td>25</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>$O(n\log n)$</td>
<td>Log-linear time</td>
<td>116</td>
<td>282</td>
<td>2.43</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic time</td>
<td>625</td>
<td>2500</td>
<td>4</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic time</td>
<td>15625</td>
<td>125000</td>
<td>8</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential time</td>
<td>$3.36 \times 10^7$</td>
<td>$1.27 \times 10^{15}$</td>
<td>$3.35 \times 10^7$</td>
</tr>
</tbody>
</table>

These functions are ordered as follows, as illustrated in Figure 16.1.
\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \]

**Figure 16.1**

*As the size \( n \) increases, the function grows.*

**Check point**

16.6 Estimate the time complexity for adding two \( n \times m \) matrices, and for multiplying an \( n \times m \) matrix by an \( m \times k \) matrix.

16.7 Describe an algorithm for finding the occurrence of the max element in a list. Analyze the complexity of the algorithm.

16.8 Describe an algorithm for removing duplicates from a list. Analyze the complexity of the algorithm.

16.9 Analyze the following sorting algorithm:

```python
for i in range(len(list) - 1):
    if list[i] > list[i + 1]:
        swap list[i] with list[i + 1]
    i = -1
```
16.10 Example 7 in §16.3 assumes \( n = 2^k \). Revise the algorithm for an arbitrary \( n \) and prove that the complexity is still \( O(\log n) \).

### 16.5 Case Studies: Finding Fibonacci Numbers

Key Point: This section analyzes and designs efficient algorithm for finding Fibonacci numbers.

Section 15.3, “Problem: Computing Fibonacci Numbers,” gave a recursive function for finding the Fibonacci number, as follows:

```
# The function for finding the Fibonacci number
def fib(index):
    if index == 0:  # Base case
        return 0
    elif index == 1:  # Base case
        return 1
    else:  # Reduction and recursive calls
        return fib(index - 1) + fib(index - 2)
```

We can now prove that the complexity of this algorithm is \( O(2^n) \). For convenience, let the index be \( n \). Let \( T(n) \) denote the complexity for the algorithm that finds \( \text{fib}(n) \) and \( c \) denote the constant time for comparing index with 0 and 1; i.e., \( T(1) \) is \( c \). So,

\[
T(n) = T(n - 1) + T(n - 2) + c
\leq 2T(n - 1) + c
\leq 2(2T(n - 2) + c) + c
= 2^2T(n - 2) + 2c + c
\]

Similar to the analysis of the Towers of Hanoi problem, we can show that \( T(n) \) is \( O(2^n) \).

This algorithm is not efficient. Is there an efficient algorithm for finding a Fibonacci number? The trouble in the recursive \texttt{fib} function is that the function is invoked redundantly with the same arguments. For example, to
compute \texttt{fib(4)}, \texttt{fib(3)} and \texttt{fib(2)} are invoked. To compute \texttt{fib(3)}, \texttt{fib(2)} and \texttt{fib(1)} are invoked. Note that \texttt{fib(2)} is redundantly invoked. We can improve it by avoiding repeated calling of the \texttt{fib} function with the same argument.

Note that a new Fibonacci number is obtained by adding the preceding two numbers in the sequence. If you use two variables \texttt{f0} and \texttt{f1} to store the two preceding numbers, the new number \texttt{f2} can be immediately obtained by adding \texttt{f0} with \texttt{f1}. Now you should update \texttt{f0} and \texttt{f1} by assigning \texttt{f1} to \texttt{f0} and assigning \texttt{f2} to \texttt{f1}, as shown in Figure 16.2.

\begin{verbatim}

f0  f1  f2
Fibonacci series: 0  1  1  2  3  5  8  13  21  34  55  89 ...
indices: 0  1  2  3  4  5  6  7   8   9   10  11

f0  f1  f2
Fibonacci series: 0  1  1  2  3  5  8  13  21  34  55  89 ...
indices: 0  1  2  3  4  5  6  7   8   9   10  11

f0  f1  f2
Fibonacci series: 0  1  1  2  3  5  8  13  21  34  55  89 ...
indices: 0  1  2  3  4  5  6  7   8   9   10  11

Figure 16.2

Variables \texttt{f0}, \texttt{f1}, and \texttt{f2} store three consecutive Fibonacci numbers in the series.

\end{verbatim}

The new function is implemented in Listing 16.1.

\begin{verbatim}

Listing 16.1 ImprovedFibonacci.py

1 def main():
2     index = eval(input("Enter an index for the Fibonacci number: "))
3     # Find and display the Fibonacci number
4     print("Fibonacci number at index", index,
5           "is", fib(index))
6     # The function for finding the Fibonacci number
7     def fib(n):
8         f0 = 0  # For fib(0)
9         f1 = 1  # For fib(1)
10        f2 = 1  # For fib(2)

\end{verbatim}
if n == 0:
    return f0
elif n == 1:
    return f1
elif n == 2:
    return f2
for i in range(3, n + 1):
    f0 = f1
    f1 = f2
    f2 = f0 + f1
return f2
main() # Call the main function

Sample output
Enter an index for the Fibonacci number: 6
Fibonacci number at index 6 is 8

Sample output
Enter an index for the Fibonacci number: 7
Fibonacci number at index 7 is 13

Obviously, the complexity of this new algorithm is \( O(n) \). This is a tremendous improvement over the recursive \( O(2^n) \) algorithm.

The algorithm for computing Fibonacci numbers presented here uses an approach known as dynamic programming. Dynamic programming is to solve subproblems, then combine the solutions of subproblems to obtain an overall solution. This naturally leads to a recursive solution. However, it would be inefficient to use recursion, because the subproblems overlap. The key idea behind dynamic programming is to solve each subprogram only once and storing the results for subproblems for later use to avoid redundant computing of the subproblems.

Check point
16.11 What is dynamic programming?
16.12 Why the recursive Fibonacci algorithm is inefficient, but the non-recursive Fibonacci algorithm is efficient?
16.6 Case Studies: Finding Greatest Common Divisors

Key Point: This section presents several algorithms in the search for an efficient algorithm for finding the greatest common divisor between two integers.

The greatest common divisor of two integers is the largest number that can evenly divide both integers. Listing 5.8, GreatestCommonDivisor.py, presented a brute-force algorithm for finding the greatest common divisor (GCD) of two integers m and n. Brute-force refers to an algorithmic approach that solves a problem in the most simple, direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a cleverer or sophisticated algorithm might do. On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

The brute-force algorithm checks whether k (for k = 2, 3, 4, and so on) is a common divisor for n1 and n2, until k is greater than n1 or n2. The algorithm can be described as follows:

```python
def gcd(m, n):
    gcd = 1
    k = 2
    while k <= m and k <= n:
        if m % k == 0 and n % k == 0:
            gcd = k
            k += 1
    return gcd
```

Assuming \( m \geq n \), the complexity of this algorithm is obviously \( O(n) \).

Is there any better algorithm for finding the gcd? Rather than searching a possible divisor from 1 up, it is more efficient to search from n down. Once a divisor is found, the divisor is the gcd. So you can improve the algorithm using the following loop:

```python
for k in range(n, 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break
```
This algorithm is better than the preceding one, but its worst-case time complexity is still $O(n)$.

A divisor for a number $n$ cannot be greater than $n / 2$. So you can further improve the algorithm using the following loop:

```python
for k in range(int(m / 2), 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break
```

However, this algorithm is incorrect, because $n$ can be a divisor for $m$. This case must be considered. The correct algorithm is shown in Listing 16.2.

Listing 16.2 GCD.py

```python
# Find gcd for integers m and n
def gcd(m, n):
    gcd = 1
    if m % n == 0:
        return n
    for k in range(int(n / 2), 0, -1):
        if m % k == 0 and n % k == 0:
            break
    return gcd

def main():
    # Prompt the user to enter two integers
    m = eval(input("Enter first integer: "))
    n = eval(input("Enter second integer: "))
    print("The greatest common divisor for", m,
          "and", n, "is", gcd(m, n))
main() # Call the main function
```

Sample output

Enter first integer: 2525
Enter second integer: 125
The greatest common divisor for 2525 and 125 is 25

Enter first integer: 3
Enter second integer: 3
The greatest common divisor for 3 and 3 is 3

Assuming $m \geq n$, the for loop is executed at most $n / 2$ times, which cuts the time by half from the previous algorithm. The time complexity of this algorithm is still $O(n)$, but practically, it is much faster than the algorithm in Listing 5.8.

NOTE:

The Big $O$ notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient. As shown in the preceding example, both algorithms in Listings 5.8 and 16.2 have the same complexity, but in practice the one in Listing 16.2 is obviously better.

A more efficient algorithm for finding gcd was discovered by Euclid around 300 B.C. This is one of the oldest known algorithms. It can be defined recursively as follows:

Let $gcd(m, n)$ denote the gcd for integers $m$ and $n$:

- If $m \% n$ is 0, $gcd (m, n)$ is $n$.
- Otherwise, $gcd(m, n)$ is $gcd(n, m \% n)$.

It is not difficult to prove the correctness of the algorithm. Suppose $m \% n = r$. So, $m = qn + r$, where $q$ is the quotient of $m / n$. Any number that is divisible by $m$ and $n$ must also be divisible by $r$. Therefore, $gcd(m, n)$ is same as $gcd(n, r)$, where $r = m \% n$. The algorithm can be implemented as in Listing 16.3.

Listing 16.3 GCDEuclid.py

```
1  # Find gcd for integers m and n
2  def gcd(m, n):
3      if m % n == 0:
4          return n
5      else:
6          return gcd(n, m % n)
```
```python
8     def main():
9         # Prompt the user to enter two integers
10        m = eval(input("Enter first integer: "))
11        n = eval(input("Enter second integer: "))
12
13        print("The greatest common divisor for", m,
14            "and", n, "is", gcd(m, n))
15
16    main() # Call the main function

Sample output
Enter first integer: 2525
Enter second integer: 125
The greatest common divisor for 2525 and 125 is 25

Sample output
Enter first integer: 3
Enter second integer: 3
The greatest common divisor for 3 and 3 is 3

In the best case when \( m \% n \) is 0, the algorithm takes just one step to find the gcd. It is difficult to analyze the average case. However, we can prove that the worst-case time complexity is \( O(\log n) \).

Assuming \( m \geq n \), we can show that \( m \% n < m / 2 \), as follows:

If \( n \leq m / 2 \), \( m \% n < m / 2 \), since the remainder of \( m \) divided by \( n \) is always less than \( n \).

If \( n > m / 2 \), \( m \% n = m - n < m / 2 \). Therefore, \( m \% n < m / 2 \).

Euclid’s algorithm recursively invokes the \( \text{gcd} \) function. It first calls \( \text{gcd}(m, n) \), then calls \( \text{gcd}(n, m \% n) \), and \( \text{gcd}(m \% n, n \% (m \% n)) \), and so on, as follows:

\[
gcd(m, n) \\
= gcd(n, m \% n) \\
= gcd(m \% n, n \% (m \% n)) \\
= \ldots
\]

Since \( m \% n < m / 2 \) and \( n \% (m \% n) < n / 2 \), the argument passed to the gcd function is reduced by half after every two iterations. After invoking gcd two times, the second parameter is less than \( n/2 \). After invoking gcd four times, the second parameter is less than \( n/4 \). After invoking gcd six times, the second parameter is less
than $\frac{n}{2^3}$. Let $k$ be the number of times the gcd function is invoked. After invoking gcd $k$ times, the second parameter is less than $\frac{n}{2^{(k/2)}}$, which is greater than or equal to 1. That is,

$$\frac{n}{2^{(k/2)}} \geq 1 \Rightarrow n \geq 2^{(k/2)} \Rightarrow \log n \geq k/2 \Rightarrow k \leq 2 \log n$$

Therefore, $k \leq 2 \log n$. So, the time complexity of the gcd function is $O(\log n)$.

The worst case occurs when the two numbers result in most divisions. It turns out that two successive Fibonacci numbers will result in most divisions. Recall that the Fibonacci series begins with 0 and 1, and each subsequent number is the sum of the preceding two numbers in the series, such as:

1 1 2 3 5 8 13 21 34 55 89 ... 

The series can be recursively defined as

$$\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(\text{index}) &= \text{fib}(\text{index} - 2) + \text{fib}(\text{index} - 1); \text{ index } \geq 2 
\end{align*}$$

For two successive Fibonacci numbers fib(index) and fib(index -1),

$$\begin{align*}
gcd(\text{fib(index)}, \text{fib(index} - 1)) \\
= gcd(\text{fib(index} - 1), \text{fib(index} - 2)) \\
= gcd(\text{fib(index} - 2), \text{fib(index} - 3)) \\
= gcd(\text{fib(index} - 3), \text{fib(index} - 4)) \\
= ... \\
= gcd(\text{fib}(2), \text{fib}(1)) \\
= 1
\end{align*}$$
For example,

\[
\begin{align*}
gcd(21, 13) &= gcd(13, 8) \\
&= gcd(8, 5) \\
&= gcd(5, 3) \\
&= gcd(3, 2) \\
&= gcd(2, 1) \\
&= 1
\end{align*}
\]

So, the number of times the gcd function is invoked is the same as the index. We can prove that

\[index \leq 1.44 \log n,\] where \(n = \text{fib}(\text{index} - 1)\). This is a tighter bound than \(index \leq 2 \log n\).

Table 16.4 summarizes the complexity of three algorithms for finding the gcd.

Table 16.4

Comparisons of GCD Algorithms

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Listing 5.8</th>
<th>Listing 16.2</th>
<th>Listing 16.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(\log n))</td>
<td></td>
</tr>
</tbody>
</table>

16.7 Case Studies: Finding Prime Numbers

Key Point: This section presents several algorithms in the search for an efficient algorithm for finding the prime numbers.

A $100,000 award awaits the first individual or group who discovers a prime number with at least 10,000,000 decimal digits (www.eff.org/awards/coop.php). Can you design a fast algorithm for finding prime numbers?

An integer greater than \(1\) is prime if its only positive divisor is \(1\) or itself. For example, \(2, 3, 5,\) and \(7\) are prime numbers, but \(4, 6, 8,\) and \(9\) are not.

How do you determine whether a number \(n\) is prime? Listing 5.8 presented a brute-force algorithm for finding prime numbers. The algorithm checks whether \(2, 3, 5, ...,\) or \(n - 1\) is divisible by \(n\). If not, \(n\) is prime. This
algorithm takes $O(n)$ time to check whether $n$ is prime. Note that you need to check only whether $2, 3, 4, 5, \ldots$, and $n/2$ is divisible by $n$. If not, $n$ is prime. The algorithm is slightly improved, but it is still of $O(n)$.

In fact, we can prove that if $n$ is not a prime, $n$ must have a factor that is greater than 1 and less than or equal to $\sqrt{n}$. Here is the proof. Since $n$ is not a prime, there exist two numbers $p$ and $q$ such that $n = pq$ with $1 < p \leq q$. Note that $n = \sqrt{n} \sqrt{n}$. $p$ must be less than or equal to $\sqrt{n}$. Hence, you need to check only whether $2, 3, 4, 5, \ldots$ or $\sqrt{n}$ is divisible by $n$. If not, $n$ is prime. This significantly reduces the time complexity of the algorithm to $O(\sqrt{n})$.

Now consider the algorithm for finding all the prime numbers up to $n$. A straightforward implementation is to check whether $i$ is prime for $i = 2, 3, 4, \ldots, n$. The program is given in Listing 16.4.

**Listing 16.4 PrimeNumbers.py**

```python
from math import sqrt

def main():
    n = eval(input("Find all prime numbers <= n, enter n: "))
    NUMBER_PER_LINE = 10  # Display 10 per line
    count = 0  # Count the number of prime numbers
    number = 2  # A number to be tested for primeness

    print("The prime numbers are:")
    # Repeatedly find prime numbers
    while number <= n:
        # Assume the number is prime
        isPrime = True  # Is the current number prime?

        # Test if number is prime
        for divisor in range(2, int(sqrt(number)) + 1):
            # If true, number is not prime
            if number % divisor == 0:
                isPrime = False  # Set isPrime to false
                break  # Exit the for loop

        # Print the prime number and increase the count
        if isPrime:
            count += 1  # Increase the count
            if count % NUMBER_PER_LINE == 0:
                # Print the number and advance to the new line
                print(" "+str(number))
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
else:
    print("" + str(number), end = "")

    # Check if the next number is prime
    number += 1
    print("\n" + str(count) + " prime(s) less than or equal to "
          + str(n))

main()

Sample output
Find all prime numbers <= n, enter n: 1000
The prime numbers are:
2      3      5      7     11     13     17     19     9     29
       31     37     41     43     47     53     59     61     67     71
...  ...  ...  ...  
168 prime(s) less than or equal to 1000

The program is not efficient if you have to compute $\sqrt{number}$ for every iteration of the for loop (line 21).
A good compiler should evaluate $\sqrt{number}$ only once for the entire for loop. To ensure this happens, you may explicitly replace line 21 by the following two lines:

```
squareRoot = int(sqrt(number))
for divisor in range(2, squareRoot + 1):
```

In fact, there is no need to actually compute $\sqrt{number}$ for every number. You need look only for the perfect squares such as 4, 9, 16, 25, 36, 49, and so on. Note that for all the numbers between 36 and 48, inclusively, their int($\sqrt{number}$) is 6. With this insight, you can replace the code in lines 16–26 with the following:

```
squareRoot = 1
# Repeatedly find prime numbers
while number <= n:
    # Assume the number is prime
    isPrime = True  # Is the current number prime?
    if squareRoot * squareRoot < number:
        squareRoot += 1

    # Test if number is prime
    for divisor in range(2, squareRoot + 1):
        if number % divisor == 0:  # If true, number is not prime
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
Now we turn our attention to analyzing the complexity of this program. Since it takes $\sqrt{i}$ steps in the for loop (lines 21–27) to check whether number $i$ is prime, the algorithm takes $\sqrt{2} + \sqrt{3} + \sqrt{4} + \ldots + \sqrt{n}$ steps to find all the prime numbers less than or equal to $n$. Observe that

$$\sqrt{2} + \sqrt{3} + \sqrt{4} + \ldots + \sqrt{n} \leq n\sqrt{n}$$

Therefore, the time complexity for this algorithm is $O(n\sqrt{n})$.

To determine whether $i$ is prime, the algorithm checks whether 2, 3, 4, 5, ..., and $\sqrt{i}$ are divisible by $i$. This algorithm can be further improved. In fact, you need to check only whether the prime numbers from 2 to $\sqrt{i}$ are possible divisors for $i$.

We can prove that if $i$ is not prime, there must exist a prime number $p$ such that $i = pq$ and $p \leq q$. Here is the proof. Assume that $i$ is not prime; let $p$ be the smallest factor of $i$. $p$ must be prime, otherwise, $p$ has a factor $k$ with $2 \leq k < p$. $k$ is also a factor of $i$, which contradicts that $p$ be the smallest factor of $i$. Therefore, if $i$ is not prime, you can find a prime number from 2 to $\sqrt{i}$ that is divisible by $i$. This leads to a more efficient algorithm for finding all prime numbers up to $n$, as shown in Listing 16.5.

```
Listing 16.5 EfficientPrimeNumbers.py

1 def main():
2     n = eval(input("Find all prime numbers <= n, enter n: "))
3
4     # A list to hold prime numbers
5     list = []
6
7     NUMBER_PER_LINE = 10  # Display 10 per line
8     count = 0  # Count the number of prime numbers
```
number = 2  # A number to be tested for primeness
squareRoot = 1  # Check whether number <= squareRoot

print("The prime numbers are \n")

# Repeatedly find prime numbers
while number <= n:
    # Assume the number is prime
    isPrime = True  # Is the current number prime?

    if squareRoot * squareRoot < number:
        squareRoot += 1

    # Test whether number is prime
    k = 0
    while k < len(list) and list[k] <= squareRoot:
        if number % list[k] == 0:  # If true, not prime
            isPrime = False  # Set isPrime to false
            break  # Exit the for loop
        k += 1

    # Print the prime number and increase the count
    if isPrime:
        count += 1  # Increase the count
        list.append(number)  # Add a new prime to the list
    if count % NUMBER_PER_LINE == 0:
        # Print the number and advance to the new line
        print(number);
    else:
        print(str(number) + " ", end = "")

    # Check whether the next number is prime
    number += 1

print("\n" + str(count) + ", prime(s) less than or equal to " + str(n))

main()
For each number \(i\), the algorithm checks whether a prime number less than or equal to \(\sqrt{i}\) is divisible by \(i\). The number of the prime numbers less than or equal to \(\sqrt{i}\) is

\[
\frac{\sqrt{i}}{\log\sqrt{i}} = \frac{2\sqrt{i}}{\log i}
\]

Thus, the complexity for finding all prime numbers up to \(n\) is

\[
\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \ldots + \frac{2\sqrt{n}}{\log n}
\]

Since \(\frac{\sqrt{i}}{\log i} < \frac{n}{\log n}\) for \(i < n\) and \(n \geq 16\),

\[
< \frac{2n\sqrt{n}}{\log n}
\]

Therefore, the complexity of this algorithm is \(O\left(\frac{n\sqrt{n}}{\log n}\right)\).

This algorithm is another example of dynamic programming. The algorithm stores the results of the subproblems in the array list and uses them later to check whether a new number is prime.

Is there any algorithm better than \(O\left(\frac{n\sqrt{n}}{\log n}\right)\)? Let us examine the well-known Eratosthenes algorithm for finding prime numbers. Eratosthenes (276–194 B.C.) was a Greek mathematician who devised a clever algorithm, known as the \textit{Sieve of Eratosthenes}, for finding all prime numbers \(\leq n\). His algorithm is to use a list named \texttt{primes} of \(n\) Boolean values. Initially, all elements in \texttt{primes} are set \texttt{True}. Since the multiples of 2 are not prime, set
primes[2 * i] to False for all $2 \leq i \leq n / 2$, as shown in Figure 16.3. Since we don’t care about primes[0] and primes[1], these values are marked × in the figure.

<table>
<thead>
<tr>
<th>primes array</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
</tr>
<tr>
<td>k=2</td>
</tr>
<tr>
<td>k=3</td>
</tr>
<tr>
<td>k=5</td>
</tr>
</tbody>
</table>

Figure 16.3
The values in primes are changed with each prime number $k$.

Since the multiples of 3 are not prime, set primes[3 * i] to False for all $3 \leq i \leq n / 3$. Since the multiples of 5 are not prime, set primes[5 * i] to False for all $5 \leq i \leq n / 5$. Note that you don’t need to consider the multiples of 4, because the multiples of 4 are also the multiples of 2, which have already been considered.

Similarly, multiples of 6, 8, 9 need not be considered. You only need to consider the multiples of a prime number $k = 2, 3, 5, 7, 11, \ldots$, and set the corresponding element in primes to False. Afterward, if primes[i] is still true, then i is a prime number. As shown in Figure 16.3, 2, 3, 5, 7, 11, 13, 17, 19, 9 are prime numbers.

Listing 16.6 gives the program for finding the prime numbers using the Sieve of Eratosthenes algorithm.

Listing 16.6 SieveOfEratosthenes.py

```python
def main():
    n = eval(input("Find all prime numbers <= n, enter n: "))
    primes = []  # Prime number sieve
    # Initialize primes[i] to true
    for i in range(n + 1):
        primes.append(True)
    k = 2
    while k <= n:
        if primes[k]:
            for i in range(k, int(n / k) + 1):
                primes[k * i] = False  # k * i is not prime
            k += 1
    count = 0  # Count the number of prime numbers found so far
    # Print prime numbers
    for i in range(2, len(primes)):
        if primes[i]:
            count += 1
```
if count % 10 == 0:
    print(i)
else:
    print(str(i) + " , end = ")
print("\n" + str(count) + 
    " prime(s) less than or equal to " + str(n))

Sample output
Find all prime numbers <= n, enter n: 1000
The prime numbers are:
    2  3  5  7  11  13  17  19  23  29
    31  37  41  43  47  53  59  61  67  71
    ...
    168 prime(s) less than or equal to 1000

Note that k <= n / k (line 11). Otherwise, k * i would be greater than n (line 19). What is the time complexity of this algorithm?

For each prime number k (line 17), the algorithm sets primes[k * i] to False (line 19). This is performed n / k - k + 1 times in the for loop (line 18). Thus, the complexity for finding all prime numbers up to n is

\[
\frac{n}{2} - 2 + \frac{n}{3} - 3 + \frac{n}{5} - 5 + \frac{n}{7} - 7 + \frac{n}{11} - 11 + 1\ldots \\
= O(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \ldots) < O(n \pi(n)) \\
= O(n \frac{\sqrt{n}}{\log n})
\]

This upper bound \(O(\frac{n\sqrt{n}}{\log n})\) is very loose. The actual time complexity is much better than \(O(\frac{n\sqrt{n}}{\log n})\). The Sieve of Eratosthenes algorithm is good for a small n such that the list primes can fit in the memory.

Table 16.5 summarizes the complexity of three algorithms for finding all prime numbers up to n.

Table 16.5
Comparisons of Prime-Number Algorithms

<table>
<thead>
<tr>
<th>Listing 5.14</th>
<th>Listing 16.4</th>
<th>Listing 16.5</th>
<th>Listing 16.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>$O(n^2)$</td>
<td>$O(n\sqrt{n})$</td>
<td>$O(\frac{n\sqrt{n}}{\log n})$</td>
</tr>
</tbody>
</table>

16.8 Case Studies: Closest Pair of Points

Key Point: This section presents efficient algorithms for finding a closest pair of points.

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. As shown in Figure 9.2, a line is drawn to connect two nearest points in the closest-pair animation. Section 7.3, “Problem: Finding a Closest Pair,” presented a brute-force algorithm for finding a closest pair of points. The algorithm computes the distances between all pairs of points and finds the one with the minimum distance. Clearly, the algorithm takes $O(n^2)$ time. Can we design a more efficient algorithm?

We will use an approach called divide-and-conquer to solve this problem. The approach divides the problem into subproblems, solves the subproblems, then combines the solutions of subproblems to obtain the solution for the entire problem. Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don’t overlap. A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem. In fact, all the recursive problems follow the divide-and-conquer approach.

Listing 16.7 describes how to solve the closest pair problem using the divide-and-conquer approach.

Listing 16.7 Algorithm for Finding a Closest Pair

Step 1: Sort the points in increasing order of $x$-coordinates. For the points with the same $x$-coordinates, sort on $y$-coordinates. This results in a sorted list $S$ of points.
Step 2: Divide $S$ into two subsets $S_1$ and $S_2$ of the equal size using the midpoint in the sorted list. Let the midpoint be in $S_1$. Recursively find the closest pair in $S_1$ and $S_2$. Let $d_1$ and $d_2$ denote the distance of the closest pairs in the two subsets, respectively.

Step 3: Find the closest pair between a point in $S_1$ and a point in $S_2$ and denote their distance to be $d_3$. The closest pair is the one with the distance $\min(d_1, d_2, d_3)$.

Selection sort and insertion sort take $O(n^2)$ time. In Chapter 14 we will introduce merge sort and heap sort. These sorting algorithms take $O(n \log n)$ time. So, Step 1 can be done in $O(n \log n)$ time.

Step 3 can be done in $O(n)$ time. Let $d = \min(d_1, d_2)$. We already know that the closest-pair distance cannot be larger than $d$. For a point in $S_1$ and a point in $S_2$ to form a closest pair in $S$, the left point must be in stripL and the right point in stripR, as pictured in Figure 16.4a.

![Figure 16.4](image)

The midpoint divides the points into two sets of equal size.

Figure 16.4
Further, for a point \( p \) in \( \text{stripL} \), you need only consider a right point within the \( d \times 2d \) rectangle, as shown in 16.2b. Any right point outside the rectangle cannot form a closest pair with \( p \). Since the closest-pair distance in \( S_2 \) is greater than or equal to \( d \), there can be at most six points in the rectangle. So, for each point in \( \text{stripL} \), at most six points in \( \text{stripR} \) need to be considered.

For each point \( p \) in \( \text{stripL} \), how do you locate the points in the corresponding \( d \times 2d \) rectangle area in \( \text{stripR} \)? This can be done efficiently if the points in \( \text{stripL} \) and \( \text{stripR} \) are sorted in increasing order of their \( y \)-coordinates. Let \( \text{pointsOrderedOnY} \) be the list of the points sorted in increasing order of \( y \)-coordinates. \( \text{pointsOrderedOnY} \) can be obtained beforehand in the algorithm. \( \text{stripL} \) and \( \text{stripR} \) can be obtained from \( \text{pointsOrderedOnY} \) in Step 3 as shown in Listing 16.8.

**Listing 16.8 Algorithm for obtaining \text{stripL} and \text{stripR}**

```python
for each point p in pointsOrderedOnY:
    if p is in S1 and mid.x - p.x <= d:
        append p to stripL
    elif p is in S2 and p.x - mid.x <= d:
        append p to stripR
```

Let the points in \( \text{stripL} \) and \( \text{stripR} \) be \([p_0, p_1, ..., p_k]\) and \([q_0, q_1, ..., q_t]\). A closest pair between a point in \( \text{stripL} \) and a point in \( \text{stripR} \) can be found using the algorithm described in Listing 16.16.

**Listing 16.9 Algorithm for Finding a Closest Pair in Step 3**

```python
d = min(d1, d2)
r = 0  # r is the index in stripR
for each point p in stripL:
    # Skip the points below the rectangle area
    while r < stripR.length and q[r].y <= p.y - d:
        r += 1

    let r1 = r
    while (r1 < stripR.length and |q[r1].y - p.y| <= d):
        # Check if (p, q[r1]) is a possible closest pair
        if distance(p, q[r1]) < d:
            d = distance(p, q[r1])
            (p, q[r1]) is now the current closest pair
        r1 = r1 + 1
r1 = r1 + 1
```
The points in \textit{stripL} are considered from \( p_0, p_1, \ldots, p_k \) in this order. For a point \( p \) in \textit{stripL}, skip the points in \textit{stripR} that are below \( p.y - d \) (lines 5–6). Once a point is skipped, it will no longer be considered. The \textbf{while} loop (lines 9–17) checks whether \((p, q[r1])\) is a possible closest pair. There are at most six such \( q[r1] \)'s. So, the complexity for finding a closest pair in Step 3 is \( O(n) \).

Let \( T(n) \) denote the time complexity for the algorithm. So,

\[
T(n) = 2T(n/2) + O(n) = O(n \log n)
\]

Therefore, a closest pair of points can be found in \( O(n \log n) \) time. The complete implementation of this algorithm is left as an exercise (see Programming Exercise 16.7).

\textbf{Check point}

16.13 What is divide-and-conquer? What is the difference between divide-and-conquer and dynamic programming?

16.14 Can you design an algorithm for finding the minimum element in a list using divide-and-conquer?

\textbf{16.9 Case Studies: The Eight Queen Problem}

Key Point: This section solves the Eight Queens problem using the backtracking approach.

The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. A recursive solution for solving the problem was introduced in §12.9. This section introduces a common algorithm design technique called \textit{backtracking} for solving this problem.
There are many possible candidates? How do you find a solution? The backtracking approach is to search for a candidate incrementally and abandons it as soon as it determines that the candidate cannot possibly be a valid solution, and explores a new candidate.

You may use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row. So, you may define array `queens` as follows:

```
queens = [-1, -1, -1, -1, -1, -1, -1, -1]
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`. Figure 16.5a shows the contents of array `queens` for the chessboard in Figure 16.5b. Initially, `queens[i] = -1` indicates that row `i` is not occupied.

![Figure 16.5](image)

**Figure 16.5**

`queens[i]` denotes the position of the queen in row `i`.

The search starts from the first row with `k = 0`, where `k` is the index of the current row being considered. The algorithm checks whether a queen can be possibly placed in `j`th column in the row for `j = 0, 1, ..., 7`, in this order. Now consider the following cases:

- If successful, continue to search for a placement for a queen in the next row. If the current row is the last row, a solution is found.
- If not successful, backtrack to the previous row and continue to search for a new placement in the previous row. If the algorithm backtracks to the first row and cannot find a new placement for a queen in this row, no solution can be found.
To see how the algorithm works, go to [www.cs.armstrong.edu/liang/animation/EightQueensAnimation.html](http://www.cs.armstrong.edu/liang/animation/EightQueensAnimation.html).

Listing 16.8 gives the program that displays a solution for the Eight Queens problem.

```python
Listing 16.8 EightQueensBackTracking.py
1  SIZE = 8 # The size of the chessboard
2  queens = [-1, -1, -1, -1, -1, -1, -1, -1] # Queen positions
3
4  # Check if a queen can be placed at row i and column j
5  def isValid(row, column):
6      for i in range(1, row + 1):
7          if (queens[row - i] == column # Check column
8              or queens[row - i] == column - i # Check upleft diagonal
9              or queens[row - i] == column + i): # Upright diagonal
10              return False # There is a conflict
11      return True # No conflict
12
13  def findPosition(k):
14      start = queens[k] + 1 # Search for a new placement
15
16      for j in range(start, 8):
17          if isValid(k, j):
18              return j # (k, j) is the place to put the queen now
19      return -1
20
21  # Search for a solution starting from a specified row
22  def search():
23      # k - 1 indicates the number of queens placed so far
24      # We are looking for a position in the kth row to place a queen
25      k = 0
26      while k >= 0 and k <= 7:
27          # Find a position to place a queen in the kth row
28          j = findPosition(k)
29          if j < 0:
30              queens[k] = -1
31              k -= 1 # back track to the previous row
32          else:
33              queens[k] = j
34              k += 1
35
36      if k == -1:
37          return False # No solution
38      else:
39          return True # A solution is found
40
41  search() # Search for a solution
42
43  # Display solution in queens
44  from tkinter import * # Import tkinter
45  window = Tk() # Create a window
46  window.title("Eight Queens") # Set a title
```
```python
image = PhotoImage(file = "image/queen.gif")
for i in range(8):
    for j in range(8):
        if queens[i] == j:
            Label(window, image = image).grid(row = i, column = j)
        else:
            Label(window, width = 5, height = 2, bg = "red")
            .grid(row = i, column = j)
window.mainloop()  # Create an event loop
```

The program invokes `search()` (line 42) to search for a solution. Initially, no queens are placed in any rows (line 2). The search now starts from the first row with \( k = 0 \) (line 26) and finds a place for the queen (line 29). If successful, place it in the row (line 34) and consider the next row (line 37). If not successful, backtrack to the previous row (lines 31-32).

The `findPosition(k)` function searches for a possible position to place a queen in row \( k \) starting from `queens[k] + 1` (line 14). It checks whether a queen can be placed at `start`, `start + 1`, ..., and `start + SIZE - 1`, in this order (lines 16-18). If possible, return the column index (line 18); otherwise, return `-1` (line 20).

The `isValid(row, column)` function is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier (line 17). It ensures that no queen is placed in the same column (line 7), no queen is placed in the upper left diagonal (line 8), and no queen is placed in the upper right diagonal (line 9), as shown in Figure 16.6.

![Figure 16.6](image/queen.gif)

**Figure 16.6**

Invoking `isValid(row, column)` checks whether a queen can be placed at `(row, column)`.

**Check point**
16.15 What is backtracking?

16.16 If you generalize the Eight Queens problem to the n-Queens problem in a n by n chess board, what will be the complexity of the algorithm?

16.10 Case Studies: Finding a Convex Hull

Key Point: This section presents efficient algorithms for finding a convex hull for a set of points.

Given a set of points, a convex hull is a smallest convex polygon that encloses all these points, as shown in Figure 16.7a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices v0, v1, v2, v3, v4, and v5 in Figure 16.7a form a convex polygon, but not in Figure 16.7b, because the line that connects v3 and v1 is not inside the polygon.

![Image](image_url)

(a) A convex hull  (b) A non-convex polygon  (c) convex hull animation

**Figure 16.7**

A convex hull is a smallest convex polygon that contains a set of points.

Convex hull has many applications in game programming, pattern recognition, and image processing. Before we introduce the algorithms, it is helpful to get acquainted with the concept using an interactive tool from www.cs.armstrong.edu/liang/animation/ConvexHull.html, as shown in Figure 16.7c. The tool allows you to add/remove points and displays the convex hull dynamically.

Many algorithms have been developed to find a convex hull. This section introduces two popular algorithms: gift-wrapping algorithm and Graham’s algorithm.

16.10.1 Gift-Wrapping Algorithm

An intuitive approach, called the *gift-wrapping algorithm*, works as follows:
Step 1: Given a list of points $S$, let the points in $S$ be labeled $s_0, s_1, \ldots, s_k$. Select the rightmost lowest point $h_0$ in $S$.

As shown in Figure 16.8a, $h_0$ is such a point. Add $h_0$ to the convex hull $H$. $H$ is a list initially being empty. Let $t_0$ be $h_0$.

Step 2: Let $t_1$ be $s_0$.

For every point $s$ in $S$

if $s$ is on the right side of the direct line from $t_0$ to $t_1$:

let $t_1$ be $s$

(After Step 2, no points lie on the right side of the direct line from $t_0$ to $t_1$, as shown in Figure 16.8b.)

Step 3: If $t_1$ is $h_0$ (see Figure 16.6d), the points in $H$ form a convex hull for $S$. Otherwise, add $t_1$ to $H$, let $t_0$ be $t_1$, and go to Step 2 (see Figure 16.8c).

![Diagram of steps](image)

(a) Step 1         (b) Step 2      (c) repeat Step 2    (d) $H$ is found

**Figure 16.8**

(a) $h_0$ is the rightmost lowest point in $S$. (b) Step 2 finds point $t_1$. (c) A convex hull is expanded repeatedly. (d) A convex hull is found when $t_1$ becomes $h_0$.

The convex hull is expanded incrementally. The correctness is supported by the fact that no points lie on the right side of the direct line from $t_0$ to $t_1$ after Step 2. This ensures that every line segment with two points in $S$ falls inside the polygon.

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. Whether a point is on the left side of a line, right side, or on the line can decided in $O(1)$ time (see Programming Exercise 4.31). Thus, it takes $O(n)$
time to find a new point \( t_1 \) in Step 2. Step 2 is repeated \( h \) times, where \( h \) is the size of the convex hull. Therefore, the algorithm takes \( O(hn) \) time. In the worst case, \( h \) is \( n \).

The implementation of this algorithm is left as an exercise (see Programming Exercise 16.11).

16.10.2 Graham’s Algorithm

A more efficient algorithm was developed by Ronald Graham in 1972. It works as follows:

Step 1: Given a list of points \( S \), select the rightmost lowest point and name it \( p_0 \) in the set \( S \). As shown in Figure 16.9a, \( p_0 \) is such a point.

Step 2: Sort the points in \( S \) angularly along the x-axis with \( p_0 \) as the center, as shown in Figure 16.9b. If there is a tie and two points have the same angle, discard the one that is closest to \( p_0 \). The points in \( S \) are now sorted as \( p_0, p_1, p_2, \ldots, p_{n-1} \).

Step 3: Push \( p_0, p_1, \) and \( p_2 \) into a stack \( H \). A stack is a first-in, first out data structure. Elements are added/removed from the top of the stack. It can be implemented using a list in which the items are appended to the end of the list and retrieved or removed from the end of the list. So the end of the list is the top of the stack.

Step 4:

\[
i = 3
\]

while \( i < n \):

Let \( t_1 \) and \( t_2 \) be the top first and second element in stack \( H \);

if (\( p_i \) is on the left side of the direct line from \( t_2 \) to \( t_1 \)):

Push \( p_i \) to \( H \)

\( i += 1 \) # Consider the next point in \( S \)

else:
The convex hull is discovered incrementally. Initially, $p_0$, $p_1$, and $p_2$ form a convex hull. Consider $p_3$. $p_3$ is outside of the current convex hull since points are sorted in increasing order of their angles. If $p_3$ is strictly on the left side of the line from $p_1$ to $p_2$ (see Figure 16.9c), push $p_3$ into $H$. Now $p_0$, $p_1$, and $p_3$ form a convex hull. If $p_3$ is on the right side of the line from $p_1$ to $p_2$ (see Figure 16.9d), pop $p_2$ out of $H$ and push $p_3$ into $H$. Now $p_0$, $p_1$, and $p_3$ form a convex hull and $p_2$ is inside of this convex hull. You can prove by induction that all the points in $H$ in Step 5 form a convex hull for all the points in the input set $S$.

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. The angles can be computed using trigonometry functions. However, you can sort the points without actually computing their angles. Observe that $p_2$ would make a greater angle than $p_1$ if and only if $p_2$ lies on the left side of the line from $p_0$ to $p_1$. Whether a point is on the left side of a line can decided in $O(1)$ time as shown in Exercise 4.32. Sorting in Step 2 can be done in $O(n \log n)$ time using the merge-sort or heap-sort algorithm to be introduced in Chapter 14. Step 4 can be done in $O(n)$ time. Therefore, the algorithm takes $O(n \log n)$ time.

The implementation of this algorithm is left as an exercise (see Programming Exercise 16.12).
Key Terms

- average-case analysis
- backtracking approach
- best-case analysis
- Big O notation
- constant time
- convex hull
- divide-and-conquer approach
- dynamic programming approach
- exponential time
- exponential time
- exponential time
- growth rate
- logarithmic time
- quadratic time
- worst-case analysis

Chapter Summary

The Big O notation is a theoretical approach for analyzing the performance of an algorithm. It estimates how fast an algorithm’s execution time increases as the input size increases. So you can compare two algorithms by examining their growth rates.

An input that results in the shortest execution time is called the best-case input and one that results in the longest execution time is called the worst-case input. Best case and worst case are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case.
An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because for many problems it is hard to determine the relative probabilities and distributions of various input instances.

If the time is not related to the input size, the algorithm is said to take constant time with the notation $O(1)$.

Linear search takes $O(n)$ time. An algorithm with the $O(n)$ time complexity is called a linear algorithm. Binary search takes $O(\log n)$ time. An algorithm with the $O(\log n)$ time complexity is called a logarithmic algorithm.

The worst-time complexity for selection sort and insertion sort is $O(n^2)$. An algorithm with the $O(n^2)$ time complexity is called a quadratic algorithm.

The time complexity for the Towers of Hanoi problem is $O(2^n)$. An algorithm with the $O(2^n)$ time complexity is called an exponential algorithm.

A Fibonacci number at a given index can be found in $O(n)$ time.

Euclid’s gcd algorithm takes $O(\log n)$ time.

All prime numbers less than or equal to $n$ can be found in $O(\sqrt{n \log n})$ time.

A closest pair can be found in $O(n \log n)$ time using the divide and conquer approach.

A convex hull for a set of points can be found in $O(n^2)$ time using the gift-wrapping algorithm and in $O(n \log n)$ time using the Graham’s algorithm.

Multiple-Choice Questions
Programming Exercises

*16.1 (Maximum consecutive increasingly ordered substring) Write a program that prompts the user to enter a string and displays the maximum consecutive increasingly ordered substring. Analyze the time complexity of your program. Here is a sample run:

```
Sample output
    Enter a string: Welcome
    Maximum consecutive substring is Wel
```

**16.2 (Maximum increasingly ordered subsequence) Write a program that prompts the user to enter a string and displays the maximum increasingly ordered subsequence of characters. Analyze the time complexity of your program. Here is a sample run:

```
Sample output
    Enter a string: Welcome
    Maximum consecutive substring is ['W', 'e', 'l', 'o']
```

*16.3 (Pattern matching) Write a program that prompts the user to enter two strings and tests whether the second string is a substring in the first string. Suppose the neighboring characters in the string are distinct. (Don’t use the find method in the str class.) Analyze the time complexity of your algorithm. Your algorithm needs to be at least \( O(n) \) time. Here is a sample run of the program:

```
Sample output
    Enter a string s1: Welcome to Python
    Enter a string s2: come
    matched at index 3
```

*16.4 (Pattern matching) Write a program that prompts the user to enter two strings and tests whether the second string is a substring in the first string. (Don’t use the find method in the str class.) Analyze the time complexity of your algorithm. Here is a sample run of the program:

```
Sample output
    Enter a string s1: Mississippi
    Enter a string s2: sip
    matched at index 6
```

*16.5 (Same-number subsequence) Write an \( O(n) \) program that prompts the user to enter a sequence of integers and finds longest subsequence with the same number. Here is a sample run of the program:
Sample output
Enter a series of numbers ending with 0: 2 4 4 8 8 8 8 2 4 4 0
The longest same number sequence starts at index 3 with 4 values of 8

*16.6 (Execution time for GCD) Write a program that obtains the execution time for finding the GCD of every two consecutive Fibonacci numbers from the index 40 to index 45 using the algorithms in Listings 16.2 and 16.3. Your program should print a table like this:

<table>
<thead>
<tr>
<th></th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 16.2 GCD1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Listing 16.3 GCD2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Hint: You can use the code template below to obtain the execution time.)

```
import time

startTime = time.time()

# Whatever you want to time, put it here
time.sleep(3)  # Sleep for 3 seconds

dendTime = time.time()

elapsed = endTime - startTime

print("It took", elapsed, "seconds to run")
```

**16.7 (Closest pair of points) Section 16.8 introduced an algorithm for finding a closest pair of points using a divide-and-conquer approach. Implement the algorithm.

**16.8 (All prime numbers up to 10,000,000,000) Write a program that finds all prime numbers up to 10,000,000,000 and store them in a file. There are approximately 455,052,511 such prime numbers.

16.9 (Number of prime numbers) Exercise 16.8 stores the prime numbers in a file named Exercise16_8.dat. Write a program that finds the number of the prime numbers less than or equal to 10, 100, 1,000, 10,000, 100,000, 1,000,000, 10,000,000, 100,000,000, 1,000,000,000, and 10,000,000,000. Your program should read the data
from Exercise16_8.dat. Note that the data file may continue to grow as more prime numbers are stored to the file.

*16.10 (Execution time for prime numbers) Write a program that obtains the execution time for finding all the prime numbers less than 8,000,000, 10,000,000, 12,000,000, 14,000,000, 16,000,000, and 18,000,000 using the algorithms in Listings 16.4–16.6. Your program should print a table like this:

<table>
<thead>
<tr>
<th>Listing 16.4</th>
<th>Listing 16.5</th>
<th>Listing 16.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000000</td>
<td>10000000</td>
<td>12000000</td>
</tr>
<tr>
<td>14000000</td>
<td>16000000</td>
<td>18000000</td>
</tr>
</tbody>
</table>

**16.11 (Geometry: gift wrapping algorithm for finding a convex hull) Section 16.16.1 introduced the gift-wrapping algorithm for finding a convex hull for a set of points. Assume that Tkinter’s coordinate system is used for the points. Implement the algorithm using the following method:

```python
# Return the points that form a convex hull
def getConvexHull(points):
```

Write a test program that prompts the user to enter the set size and the points and displays the points that form a convex hull. Here is a sample run:

_Sample output_

Enter points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9
The convex hull is
[[2.5, 2], [6, 2.4], [5.5, 9], [1.5, 34.5], [1, 2.4]]

**16.12 (Geometry: Graham’s algorithm for finding a convex hull) Section 16.16.2 introduced Graham’s algorithm for finding a convex hull for a set of points. Assume that the Tkinter’s coordinate system is used for the points. Implement the algorithm using the following function:
# Return the points that form a convex hull
def getConvexHull(points):

Write a test program that prompts the user to enter the set size and the points and displays the points that form a
convex hull. Here is a sample run:

**Sample output**
Enter points: -100 25 -29 -15 14 15 26 56 24 55 -9 34 -19
The convex hull is
[[25, -29], [55, -9], [56, 24], [-15, 26], [-100, -24]]

**16.13 (Tkinter: convex hull using gift-wrapping algorithm)** Exercise 16.11 finds a convex hull for a set of points entered from the console. Write a Tkinter program that enables the user to add/remove points by clicking the
left/right mouse button, and displays a convex hull, as shown in Figure 16.5c.

**16.14 (Turtle: convex hull using gift-wrapping algorithm)** Extend Exercise 16.11 to display the points and their convex hull, as shown in Figure 16.8.

**16.15 (Tkinter: convex hull using Graham’s algorithm)** Extend Exercise 16.12 to display the points and their convex hull, as shown in Figure 16.5c.

**16.16 (Turtle: convex hull using Graham’s algorithm)** Extend Exercise 16.12 to display the points and their convex hull, as shown in Figure 16.10.

![Python Turtle Graphics](image)

**Figure 16.10**
The program displays the points and their convex hull.
**16.17 (Turtle: non-cross polygon) Write a program that prompts the user to enter points and displays a non-crossed polygon that links all the points, as shown in Figure 16.11a. A polygon is crossed if two or more sides intersect, as shown in Figure 16.11b. Use the following algorithm to construct a polygon from a set of points.

Step 1: Given a list of points S, select the rightmost lowest point and name it \( p_0 \) in S.

Step 2: Sort the points in S angularly along the x-axis with \( p_0 \) as the center. If there is a tie and two points have the same angle, the one that is closest to \( p_0 \) is consider greater. The points in S are now sorted as \( p_0, p_1, p_2, ..., p_{n-1} \).

Step 3: The sorted points form a non-cross polygon.

Here is a sample run:

**Sample output**

Enter points: -100 -24 25 -29 -15 14.5 -15 26 56 24 55 -9 34 34 78 -19

(a)              (b) crossed polygon

*Figure 16.11*

(a) The program displays a non-crossed polygon for a set of points. (b) Two or more sides intersect in a crossed polygon.

*16.18 (Largest block) The problem for finding a largest block is described in Exercise 11.47. Design an algorithm for solving this problem and analyze its complexity. Can you design an \( O(n^2) \) algorithm for this problem?