

## CHAPTER 18

### Developing Efficient Algorithms

#### Objectives

- To estimate algorithm efficiency using the Big  $O$  notation (§18.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§18.2).
- To determine the complexity of various types of algorithms (§18.3).
- To analyze the binary search algorithm (§18.4.1).
- To analyze the selection sort algorithm (§18.4.2).
- To analyze the Towers of Hanoi algorithm (§18.4.4).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, exponential) (§18.4.5).
- To design efficient algorithms for finding Fibonacci numbers using dynamic programming (§18.5).
- To find the GCD using Euclid's algorithm (§18.6).
- To find prime numbers using the sieve of Eratosthenes (§18.7).
- To design efficient algorithms for finding the closest pair of points using the divide-and-conquer approach (§18.8).
- To solve the Eight Queens problem using the backtracking approach (§18.9).
- To design efficient algorithms for finding a convex hull for a set of points (§18.10).

## 18.1 Introduction

*Key Point: Algorithm design is to develop a mathematical process for solving a program. Algorithm analysis is to predict the performance of an algorithm.*

This chapter will use a variety of examples to introduce common algorithmic techniques (dynamic programming, divide-and-conquer, and backtracking) for developing efficient algorithms. Suppose two algorithms perform the same task, such as search (linear search vs. binary search). Which one is better? To answer this question, you might implement these algorithms 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.

### **Pedagogical NOTE**

If you are not taking a data structures course, you may skip this chapter.

## 18.2 Measuring Algorithm Efficiency Using 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 nondominating terms in the function.*

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires  $n$  comparisons for an array of size  $n$ . If the key is in the array, it requires  $n/2$  comparisons on average. The algorithm's execution time is

proportional to the size of the array. If you double the size of the array, 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 the “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 analysis and worst-case analysis are to analyze the algorithms for their best-case input and worst-case input. Best-case and worst-case analysis 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 18.1**. Therefore,  $O(n) = O(n/2) = O(100n)$ .

Table 18.1

Growth Rates

$f(n)$ $n$	$n$	$n/2$	$100n$
100	100	50	10000
200	200	100	20000
	2	2	2

$f(200)/f(100)$

Consider the algorithm for finding the maximum number in an array 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  $n - 1$ ). Therefore, 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 an array takes constant time, because the time does not grow as the size of the array increases.

The following mathematical **summations** are often useful in algorithm analysis:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \dots + (n - 1) + n = \frac{n(n + 1)}{2} = O(n^2)$$

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

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

### Check point

18.1 Why is a constant factor ignored in the Big  $O$  notation? Why is a nondominating term ignored in the Big  $O$  notation?

18.2 What is the order of each of the following functions?

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

### 18.3 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:

```
for (i = 1; i <= n; i++)
{
    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

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

The theoretical analysis predicts the performance of the algorithm. To see how this algorithm performs, we run the code in Listing 18.1 to obtain the execution time for  $n = 250000000$ ,  $500000000$ ,  $1000000000$ , and  $2000000000$ .

#### Listing 18.1 PerformanceTest.cpp

```
1  #include <iostream>
2  #include <ctime> // for time function
3  using namespace std;
4
5  void getTime(int n)
6  {
7      int startTime = time(0);
8      double k = 0;
9      for (int i = 1; i <= n; i++)
10     {
11         k = k + 5;
12     }
13     int endTime = time(0);
14     cout << "Execution time for n = " << n
15          << " is " << (endTime - startTime) << " seconds" << endl;
16 }
17
18 int main()
19 {
20     getTime(250000000);
21     getTime(500000000);
```

```

22     getTime(1000000000);
23     getTime(2000000000);
24
25     return 0;
26 }

```

### *Sample output*

```

Execution time for n = 2500000000 is 2 seconds
Execution time for n = 5000000000 is 4 seconds
Execution time for n = 10000000000 is 8 seconds
Execution time for n = 20000000000 is 17 seconds

```

Our analysis predicts a linear time complexity for this loop. As shown in the sample output, when the input size increases two times, the runtime increases roughly two times. The execution confirms to the prediction.

### *Example 2*

What is the time complexity for the following loop?

```

for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        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. Thus, the time complexity for the loop is

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

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:

```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= i; j++)
    {
        k = k + i + j;
    }
}
```

The outer loop executes  $n$  times. For  $i = 1, 2, \dots$ , the inner loop is executed one time, two times, and  $n$  times. Thus, the time complexity for the loop is

$$\begin{aligned} T(n) &= c + 2c + 3c + 4c + \dots + nc \\ &= cn(n+1)/2 \\ &= (c/2)n^2 + (c/2)n \\ &= O(n^2) \end{aligned}$$

### Example 4

Consider the following loop:

```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= 20; j++)
```

```

    {
        k = k + i + j;
    }
}

```

The inner loop executes 20 times, and the outer loop  $n$  times. Therefore, the time complexity for the loop is

$$T(n) = 20 * c * n = O(n)$$

### Example 5

Consider the following sequences:

```

for (j = 1; j <= 10; j++)
{
    k = k + 4;
}

for (i = 1; i <= n; i++)
{
    for (j = 1; j <= 20; j++)
    {
        k = k + i + j;
    }
}

```

The first loop executes 10 times, and the second loop  $20 * n$  times. Thus, the time complexity for the loop is

$$T(n) = 10 * c + 20 * c * n = O(n)$$

### Example 6

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

```

result = 1;
for (int i = 1; i <= n; i++)
    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:

```
result = a;
for (int i = 1; i <= k; i++)
    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 18.7.)

#### NOTE

For simplicity, since  $O(\log n) = O(\log_2 n) = O(\log_a n)$ , the constant base is omitted.

#### Check point:

18.3 Count the number of iterations in the following loops.

```
int count = 1;
while (count < 30)
{
    count = count * 2;
}
```

(a)

```
int count = 15;
while (count < 30)
{
    count = count * 3;
}
```

(b)

```
int count = 1;
while (count < n)
{
    count = count * 2;
}
```

(c)

```
int count = 15;
while (count < n)
{
    count = count * 3;
}
```

(d)

18.4 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.

```
for (int i = 0; i < n; i++)
{
    cout << '*';
}
```

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        cout << '*';
    }
}
```

```
for (int k = 0; k < n; k++)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << '*';
        }
    }
}
```

```
for (int k = 0; k < 10; k++)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << '*';
        }
    }
}
```

18.5 Use the Big  $O$  notation to estimate the time complexity of the following functions:

```
void mA(int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << rand();
    }
}
```

```
void mB(int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
            cout << rand();
    }
}
```

```
void mC(int m, int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << m[i];
    }

    for (int i = size - 1; i >= 0; i--)
    {
        cout << m[i];
        i--;
    }
}
```

```
void mD(int m[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < i; j++)
            cout << m[i] * m[j];
    }
}
```

18.6 Design an  $O(n)$  time algorithm for computing the sum of numbers from  $n_1$  to  $n_2$  for ( $n_1 < n_2$ ). Can you design an  $O(1)$  for performing the same task?

18.7 Example 7 in Section 18.3 assumes  $n = 2^k$ . Revise the algorithm for an arbitrary  $n$  and prove that the complexity is still  $O(\log n)$ .

## 18.4 Analyzing Algorithm Time Complexity

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

### 18.4.1 Analyzing Binary Search

The binary search algorithm presented in [Listing 7.10](#), BinarySearch.cpp, searches for a key in a sorted array. 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  $k = \log n$ . Since a binary search eliminates half of the input after two comparisons,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + c \log n = 1 + (\log n)c \\ &= O(\log n) \end{aligned}$$

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. In the case of binary search, each time you double the array size, at most one more comparison will be required. If you square the input size of any logarithmic time algorithm, you only double the time of execution. So a logarithmic-time algorithm is very efficient.

#### 18.4.2 Analyzing Selection Sort

The selection sort algorithm presented in [Listing 7.11](#), SelectionSort.cpp, finds the smallest element in the list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list, and so on until the remaining list contains only one element left to be sorted. 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. Thus,

$$\begin{aligned} T(n) &= (n-1) + c + (n-2) + c + \dots + 2 + c + 1 + c \\ &= \frac{(n-1)(n-1+1)}{2} + c(n-1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\ &= O(n^2) \end{aligned}$$

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

### 18.4.3 Analyzing the Towers of Hanoi Problem

The Towers of Hanoi problem presented in Listing 17.7, TowersOfHanoi.cpp, 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. Thus,

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2(2(2T(n-3) + 1) + 1) + 1 \\ &= 2^{n-1}T(1) + 2^{n-2} + \dots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \dots + 2 + 1 = (2^n - 1) = O(2^n) \end{aligned}$$

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  $2^{32} / (365 * 24 * 60 * 60) = 136$  years to move 32 disks and  $2^{64} / (365 * 24 * 60 * 60) = 585$  billion years to move 64 disks.

### 18.4.4 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  $T(n) = T(\frac{n}{2}) + c$ ,

$T(n) = T(n-1) + O(n)$ , and  $T(n) = 2T(n-1) + O(1)$ , respectively. Table 18.2 summarizes the common recurrence relations.

Table 18.2

### Common Recurrence Functions

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	Checkpoint Question 18.20
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 19)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

### 18.4.5 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. Table 18.3 lists some common growth functions and shows how growth rates change as the input size doubles from  $n = 25$  to  $n = 50$ .

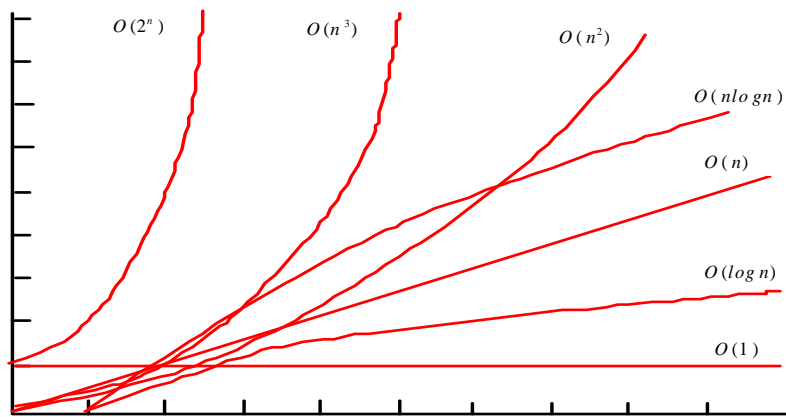
Table 18.3

#### Change of Growth Rates

Function	Name	$n = 25$	$n = 50$	$f(50)/f(25)$
$O(1)$	Constant time	1	1	1
$O(\log n)$	Logarithmic time	4.64	5.64	1.21
$O(n)$	Linear time	25	50	2
$O(n \log n)$	Log-linear time	116	282	2.43
$O(n^2)$	Quadratic time	625	2,500	4
$O(n^3)$	Cubic time	15,625	125,000	8
$O(2^n)$	Exponential time	$3.36 \times 10^7$	$1.27 \times 10^{15}$	$3.35 \times 10^7$

These functions are ordered as follows, as illustrated in Figure 18.1.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



**Figure 18.1**

*As the size  $n$  increases, the function grows.*

### Check point

18.8 Put the following growth functions in order:

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

18.9 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.

18.10 Describe an algorithm for finding the occurrence of the max element in an array. Analyze the complexity of the algorithm.

18.11 Describe an algorithm for removing duplicates from an array. Analyze the complexity of the algorithm.

18.12 Analyze the following sorting algorithm:

```
for (int i = 0; i < SIZE; i++)
{
    if (list[i] > list[i + 1])
    {
        swap list[i] with list[i + 1];
        i = 0;
    }
}
```

## 18.5 Finding Fibonacci Numbers Using Dynamic Programming

Key Point: *This section analyzes and designs an efficient algorithm for finding Fibonacci numbers using dynamic programming.*

**Section 17.3**, Case Study: Computing Fibonacci Numbers, gave a recursive function for finding the Fibonacci number, as follows:

```
// The function for finding the Fibonacci number
public static long fib(long index)
{
    if (index == 0) // Base case
        return 0;
    else if (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 the index with 0 and 1; that is,  $T(1)$  is  $c$ . Thus,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c \\ &\leq 2(2T(n-2) + c) + c \\ &= 2^2T(n-2) + 2c + c \end{aligned}$$

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

However, this algorithm is not efficient. Is there an efficient algorithm for finding a Fibonacci number?

The trouble with the recursive `fib` function is that the function is invoked redundantly with the same arguments.

For example, to compute `fib(4)`, `fib(3)` and `fib(2)` are invoked. To compute `fib(3)`, `fib(2)` and `fib(1)` are invoked. Note that `fib(2)` is redundantly invoked. We can improve it by avoiding repeated calling of the `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 the two variables `f0` and `f1` to store the two preceding numbers, the new number, `f2`, can be immediately obtained by adding `f0` with `f1`. Now you should update `f0` and `f1` by assigning

f1 to f0 and assigning f2 to f1, as shown in Figure 18.2.

		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	

T

**Figure 18.2**

Variables f0, f1, and f2 store three consecutive Fibonacci numbers in the series.

The new function is implemented in Listing 18.2.

**Listing 18.2 ImprovedFibonacci.cpp**

```

1  #include <iostream>
2  using namespace std;
3
4  // The function for finding the Fibonacci number
5  int fib(int);
6
7  int main()
8  {
9      // Prompt the user to enter an integer
10     cout << "Enter an index for the Fibonacci number: ";
11     int index;
12     cin >> index;
13
14     // Display factorial
15     cout << "Fibonacci number at index " << index << " is "
16         << fib(index) << endl;
17
18     return 0;
19 }
20
21 // The function for finding the Fibonacci number
22 int fib(int n)
23 {

```



```

24     long f0 = 0; // For fib(0)
25     long f1 = 1; // For fib(1)
26     long f2 = 1; // For fib(2)
27
28     if (n == 0)
29         return f0;
30     else if (n == 1)
31         return f1;
32     else if (n == 2)
33         return f2;
34
35     for (int i = 3; i <= n; i++)
36     {
37         f0 = f1;
38         f1 = f2;
39         f2 = f0 + f1;
40     }
41
42     return f2;
43 }

```

#### 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 the process of solving subproblems, then combining the solutions of the 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 subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.**

#### Check point

18.13 What is dynamic programming? Give an example of dynamic programming.

18.14 Why is the recursive Fibonacci algorithm inefficient, but the nonrecursive Fibonacci algorithm efficient?

### 18.6 Finding Greatest Common Divisors Using Euclid's Algorithm

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

The greatest common divisor (GCD) of two integers is the largest number that can evenly divide both integers.

Listing 5.10, GreatestCommonDivisor.cpp, presented a brute-force algorithm for finding the greatest common divisor of two integers `m` and `n`. **Brute force** refers to an algorithmic approach that solves a problem in the simplest or most 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 more 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:

```
public static int gcd(int m, int n) {  
    int gcd = 1;  
  
    for (int k = 2; k <= m && k <= n; k++) {  
        if (m % k == 0 && n % k == 0)  
            gcd = k;  
    }  
  
    return gcd;  
}
```

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

Is there a 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. Therefore, you can improve the algorithm using the following loop:

```
for (int k = n; k >= 1; k--) {  
    if (m % k == 0 && 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:

```
for (int k = m / 2; k >= 1; k--) {
    if (m % k == 0 && 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 18.3.

#### Listing 18.3 GCD.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // Return the gcd of two integers
5  int gcd(int m, int n)
6  {
7      int gcd = 1;
8
9      if (m % n == 0) return n;
10
11     for (int k = n / 2; k >= 1; k--)
12     {
13         if (m % k == 0 && n % k == 0)
14         {
15             gcd = k;
16             break;
17         }
18     }
19
20     return gcd;
21 }
22
23 int main()
24 {
25     // Prompt the user to enter two integers
26     cout << "Enter first integer: ";
27     int n1;
28     cin >> n1;
29
30     cout << "Enter second integer: ";
31     int n2;
32     cin >> n2;
33
34     cout << "The greatest common divisor for " << n1 <<
```

```

35     " and " << n2 << " is " << gcd(n1, n2) << endl;
36
37     return 0;
38 }

```

### Sample output

```

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

```

### Sample output

```

Enter first integer: 3 ↵ Enter
Enter second integer: 3 ↵ Enter
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 5.10.

### 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 5.10 and 18.3 have the same complexity, but in practice the one in Listing 18.3 is obviously better.

A more efficient algorithm for finding the 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 this algorithm. Suppose `m % n = r`. Thus, `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 the same as `gcd(n, r)`, where `r = m % n`. The algorithm can be implemented as in Listing 18.4.

#### Listing 18.4 GCDEuclid.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // Return the gcd of two integers
5  int gcd(int m, int n)
6  {
7      if (m % n == 0)
8          return n;
9      else
10         return gcd(n, m % n);
11 }
12
13 int main()
14 {
15     // Prompt the user to enter two integers
16     cout << "Enter first integer: ";
17     int n1;
18     cin >> n1;
19
20     cout << "Enter second integer: ";
21     int n2;
22     cin >> n2;
23
24     cout << "The greatest common divisor for " << n1 <<
25         " and " << n2 << " is " << gcd(n1, n2) << endl;
26
27     return 0;
28 }
```

#### Sample output

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

### Sample output

```
Enter first integer: 3 ↵ Enter
Enter second integer: 3 ↵ Enter
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 **gcd** function. It first calls **gcd**( $m$ ,  $n$ ), then calls **gcd**( $n$ ,  $m \% n$ ), and **gcd**( $m \% n$ ,  $n \% (m \% n)$ ), and so on, as follows:

```
gcd(m, n)
= gcd(n, m % n)
= gcd(m % n, n % (m % n))
= ...
```

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,

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

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 the most divisions. It turns out that **two successive Fibonacci numbers will result in the 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:

0 1 1 2 3 5 8 13 21 34 55 89 ...

The series can be recursively defined as

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

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

```
gcd(fib(index), fib(index - 1))
= gcd(fib(index - 1), fib(index - 2))
= gcd(fib(index - 2), fib(index - 3))
= gcd(fib(index - 3), fib(index - 4))
= ...
= gcd(fib(2), fib(1))
= 1
```

For example,

```
gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
= gcd(2, 1)
= 1
```

Therefore, 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  $n = \text{fib}(index - 1)$ . This is a tighter bound than  $index \leq 2 \log n$ .

**Table 18.4** summarizes the complexity of three algorithms for finding the GCD.

*Table 18.4*

*Comparisons of GCD Algorithms*

Algorithm	Complexity	Description
Listing 5.9	$O(n)$	Brute-force, checking all possible divisors
Listing 18.2	$O(n)$	Checking half of all possible divisors
Listing 18.3	$O(\log n)$	Euclid's algorithm

### Check point

18.15 Prove that the following algorithm for finding the GCD of the two integers  $m$  and  $n$  is incorrect.

```

int gcd = 1;
for (int k = min(sqrt(n), sqrt(m)); k >= 1; k--)
{
    if (m % k == 0 && n % k == 0)
    {
        gcd = k;
        break;
    }
}

```

e.g.  $m=16, n=8$

## 18.7 Efficient Algorithms for Finding Prime Numbers

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

A \$150,000 award awaits the first individual or group who discovers a prime number with at least 100,000,000 decimal digits ([w2.eff.org/awards/coop-prime-rules.php](http://w2.eff.org/awards/coop-prime-rules.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 5.17 presented a brute-force algorithm for finding prime numbers. The algorithm checks whether 2, 3, 4, 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, ..., and  $n/2$  is divisible by  $n$ . If not,  $n$  is prime. This 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, ..., 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  $i = 2, 3, 4, \dots, n$ . The program is given in Listing 18.5.

#### Listing 18.5 PrimeNumbers.cpp

```

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main()
6  {
7      cout << "Find all prime numbers <= n, enter n: ";
8      int n;
9      cin >> n;
10
11     const int NUMBER_PER_LINE = 10; // Display 10 per line
12     int count = 0; // Count the number of prime numbers
13     int number = 2; // A number to be tested for primeness
14
15     cout << "The prime numbers are:" << endl;
16
17     // Repeatedly find prime numbers
18     while (number <= n)
19     {
20         // Assume the number is prime
21         bool isPrime = true; // Is the current number prime?
22
23         // Test if number is prime
24         for (int divisor = 2; divisor <= sqrt(number * 1.0); divisor++)
25         {
26             if (number % divisor == 0)
27             { // If true, number is not prime
28                 isPrime = false; // Set isPrime to false
29                 break; // Exit the for loop
30             }
31         }
32
33         // Print the prime number and increase the count
34         if (isPrime)
35         {
36             count++; // Increase the count
37
38             if (count % NUMBER_PER_LINE == 0)
39             {

```

```

40         // Print the number and advance to the new line
41         cout << number << endl;
42     }
43     else
44         cout << number << " ";
45 }
46
47 // Check whether the next number is prime
48 number++;
49 }
50
51 cout << "\n" << count << " number of primes <= " << n << endl;
52
53 return 0;
54 }

```

### 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 number of primes <= 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 make sure this happens, you may explicitly replace line 21 by the following two lines:

```

int squareRoot = sqrt(number * 1.0);
for (int divisor = 2; divisor <= squareRoot; divisor++)

```

In fact, there is no need to actually compute `sqrt(number)` for every `number`. You need only look 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, their `static_cast<int>(sqrt(number))` is 6. With this insight, you can replace the code in lines 18–31 with the following code:

```

...
int squareRoot = 1;

// Repeatedly find prime numbers
while (number <= n)
{
    // Assume the number is prime
    boolean isPrime = true; // Is the current number prime?

    if (squareRoot * squareRoot < number) squareRoot++;
}

```

```

// Test whether number is prime
for (int divisor = 2; divisor <= squareRoot; divisor++)
{
    if (number % divisor == 0) // If true, number is not prime
    {
        isPrime = false; // Set isPrime to false
        break; // Exit the for loop
    }
}
...

```

Now we turn our attention to analyzing the complexity of this program. Since it takes  $\sqrt{i}$  steps in the for loop (lines 21–26) to check whether number  $i$  is prime, the algorithm takes  $\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{n}$  steps to find all the prime numbers less than or equal to  $n$ . Observe that

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

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  $\sqrt{i}$  is divisible by  $i$ . This algorithm can be further improved. In fact, you need only check 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 18.6.

#### Listing 18.6 EfficientPrimeNumbers.cpp

```

1  #include <iostream>
2  #include <cmath>

```

```

3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      cout << "Find all prime numbers <= n, enter n: ";
9      int n;
10     cin >> n;
11
12     const int NUMBER_PER_LINE = 10; // Display 10 per line
13     int count = 0; // Count the number of prime numbers
14     int number = 2; // A number to be tested for primeness
15     // A vector to hold prime numbers
16     vector<int> primeVector;
17     int squareRoot = 1; // Check whether number <= squareRoot
18
19     cout << "The prime numbers are:" << endl;
20
21     // Repeatedly find prime numbers
22     while (number <= n)
23     {
24         // Assume the number is prime
25         bool isPrime = true; // Is the current number prime?
26
27         if (squareRoot * squareRoot < number) squareRoot++;
28
29         // Test if number is prime
30         for (int k = 0; k < primeVector.size()
31              && primeVector.at(k) <= squareRoot; k++)
32         {
33             if (number % primeVector.at(k) == 0) // If true, not prime
34             {
35                 isPrime = false; // Set isPrime to false
36                 break; // Exit the for loop
37             }
38         }
39
40         // Print the prime number and increase the count
41         if (isPrime)
42         {
43             count++; // Increase the count
44             primeVector.push_back(number); // Add a new prime to the list
45             if (count % NUMBER_PER_LINE == 0)
46             {
47                 // Print the number and advance to the new line
48                 cout << number << endl;
49             }
50             else
51                 cout << number << " ";
52         }
53
54         // Check if the next number is prime
55         number++;
56     }
57
58     cout << "\n" << count << " number of primes <= " << n << endl;
59     return 0;
60 }

```

**Sample output**

Find all prime numbers  $\leq n$ , enter  $n$ :

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 number of primes  $\leq 1000$

Let  $\pi(i)$  denote the number of prime numbers less than or equal to  $i$ . The primes under 20 are 2, 3, 5, 7, 11, 13,

17, and 19. So,  $\pi(2)$  is 1,  $\pi(3)$  is 2,  $\pi(6)$  is 3, and  $\pi(20)$  is 8. It has been proved that  $\pi(i)$  is approximately  $\frac{i}{\log i}$

(see [primes.utm.edu/howmany.shtml](http://primes.utm.edu/howmany.shtml)).

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 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

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n}$$

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

$$\begin{aligned} & \frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n} \\ & < \frac{2n\sqrt{n}}{\log n} \end{aligned}$$

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

Is there an algorithm that is better than  $O(\frac{n\sqrt{n}}{\log n})$ ? 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 *Sieve of Eratosthenes*, for finding all prime numbers  $\leq n$ . His algorithm is to use an array named

`primes` of  $n$  Boolean values. Initially, all elements in `primes` are set `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 18.3. Since we don't care about

`primes[0]` and `primes[1]`, these values are marked  $\times$  in the figure.

	primes array																											
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	$\times$	$\times$	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k=2$	$\times$	$\times$	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k=3$	$\times$	$\times$	T	T	F	T	F	T	F	F	T	F	T	F	F	T	F	T	F	T	F	F	T	F	T	F	T	F
$k=5$	$\times$	$\times$	(T)	(T)	F	(T)	F	(T)	F	F	F	(T)	F	(T)	F	F	F	(T)	F	(T)	F	F	F	(T)	F	F	F	F

**Figure 18.3**

*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 they are also multiples of `2`, which have already been considered. Similarly, multiples of `6`,

`8`, `9` need not be considered. You need only consider the multiples of a prime number  $k = 2, 3, 5, 7, 11, \dots$ , 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 18.3, `2, 3, 5, 7, 11, 13, 17, 19, 23` are prime numbers. Listing 18.7 gives the

program for finding the prime numbers using the *Sieve of Eratosthenes* algorithm.

#### Listing 18.7 SieveOfEratosthenes.cpp

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "Find all prime numbers <= n, enter n: ";
7      int n;
8      cin >> n;
9  }
```


```

10  bool* primes = new bool[n + 1]; // Prime number sieve
11
12  // Initialize primes[i] to true
13  for (int i = 0; i < n + 1; i++)
14  {
15      primes[i] = true;
16  }
17
18  for (int k = 2; k <= n / k; k++)
19  {
20      if (primes[k])
21      {
22          for (int i = k; i <= n / k; i++)
23          {
24              primes[k * i] = false; // k * i is not prime
25          }
26      }
27  }
28
29  const int NUMBER_PER_LINE = 10; // Display 10 per line
30  int count = 0; // Count the number of prime numbers found so far
31  // Print prime numbers
32  for (int i = 2; i < n + 1; i++)
33  {
34      if (primes[i])
35      {
36          count++;
37          if (count % 10 == 0)
38              cout << i << endl;
39          else
40              cout << i << " ";
41      }
42  }
43
44  cout << "\n" << count << " number of primes <= " << n << endl;
45
46  delete [] primes;
47
48  return 0;
49 }

```

#### 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 number of primes <= 1000

```

Note that  $k \leq n / k$  (line 18). Otherwise,  $k * i$  would be greater than  $n$  (line 20). What is the time complexity of this algorithm?

For each prime number  $k$  (line 20), the algorithm sets `primes[k * i]` to `false` (line 24). This is performed  $n / k - k + 1$  times in the for loop (line 22). Thus, the complexity for finding all prime numbers up to  $n$  is

$$\begin{aligned} & \frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \dots \\ &= O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \dots\right) < O(n\pi(n)) \\ &= O\left(n \frac{\sqrt{n}}{\log n}\right) \end{aligned}$$

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

**Table 18.5** summarizes the complexity of these three algorithms for finding all prime numbers up to  $n$ .

Table 18.5

#### Comparisons of Prime-Number Algorithms

Algorithm	Complexity	Description
Listing 5.14	$O(n^2)$	Brute-force, checking all possible divisors
Listing 18.4	$O(n\sqrt{n})$	Checking divisors up to $\sqrt{n}$
Listing 8.5	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Checking prime divisors up to $\sqrt{n}$
Listing 8.6	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Sieve of Eratosthenes

#### Check point

18.16 Prove that if  $n$  is not prime, there must exist a prime number  $p$  such that  $p \leq \sqrt{n}$  and  $p$  is a factor of  $n$ .

18.17 Describe how the sieve of Eratosthenes is used to find the prime numbers.

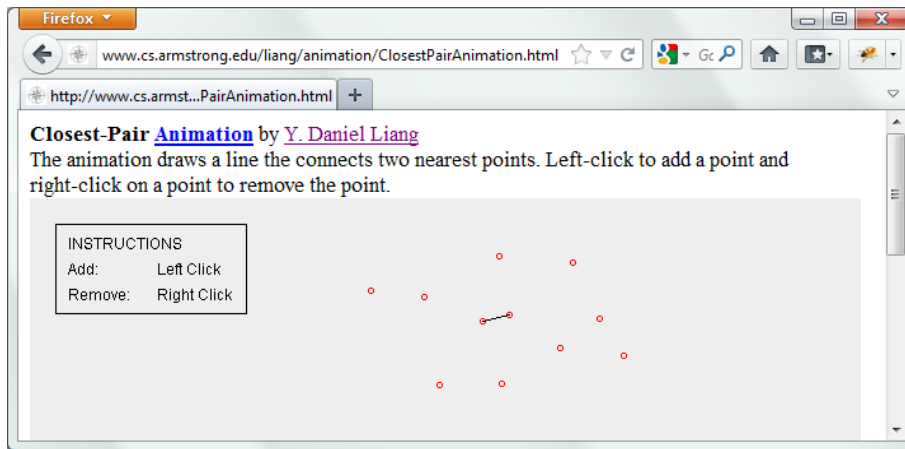


## 18.8 Finding the Closest Pair of Points Using Divide-and-Conquer

**Key Point:** *This section presents efficient algorithms for finding the closest pair of points using divide-and-conquer.*

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 18.4**, a line is drawn to connect the two nearest points in the closest-pair animation.



**Figure 18.4**

*The closet-pair animation draws a line to connect the closest pair of points dynamically as points are added and removed interactively.*

**Section 8.6, Case Study: Finding the Closest Pair**, presented a brute-force algorithm for finding the 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 the 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 solutions for recursive problems follow the divide-and-conquer approach.

**Listing 18.8** describes how to solve the closest pair problem using the divide-and-conquer approach.

### Listing 18.8 Algorithm for Finding the Closest Pair

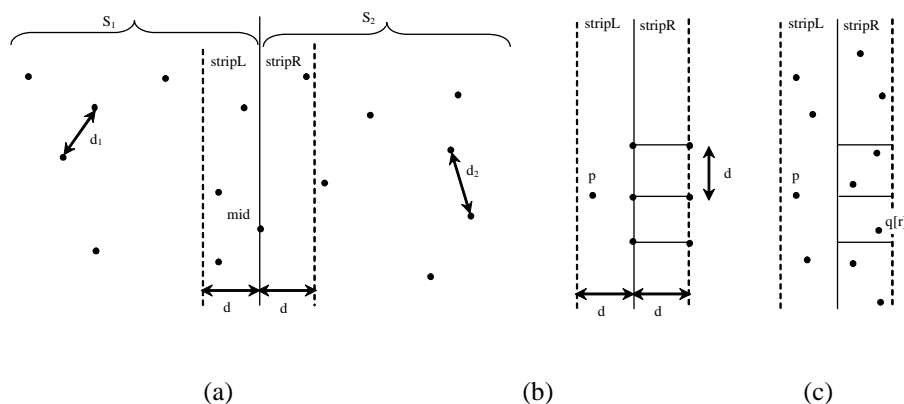
Step 1: Sort the points in increasing order of  $x$ -coordinates. For the points with the same  $x$ -coordinates, sort on  $y$ -coordinates. This results in a sorted list  $S$  of points.

Step 2: Divide  $S$  into two subsets,  $S_1$  and  $S_2$ , of 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 as  $d_3$ . The closest pair is the one with the distance  $\min(d_1, d_2, d_3)$ .

Selection sort takes  $O(n^2)$  time. In Chapter 19 we will introduce merge sort and heap sort. These sorting algorithms take  $O(n \log n)$  time. 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 the closest pair in  $S$ , the left point must be in `stripL` and the right point in `stripR`, as illustrated in Figure 18.5a.



**Figure 18.5**

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

For a point  $p$  in `stripL`, you need only consider a right point within the  $d \times 2d$  rectangle, as shown in 18.5b. Any right point outside the rectangle cannot form the 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. Thus, for each point in `stripL`, at most six points in `stripR` need to be considered.

For each point  $p$  in `stripL`, how do you locate the points in the corresponding  $d \times 2d$  rectangle area in `stripR`? This can be done efficiently if the points in `stripL` and `stripR` are sorted in increasing order of their y-coordinates. Let `pointsOrderedOnY` be the list of the points sorted in increasing order of y-coordinates. `pointsOrderedOnY` can be obtained beforehand in the algorithm. `stripL` and `stripR` can be obtained from `pointsOrderedOnY` in Step 3 as shown in Listing 18.9.

**Listing 18.9 Algorithm for Obtaining `stripL` and `stripR`**

```

1  for each point p in pointsOrderedOnY
2      if (p is in S1 and mid.x - p.x <= d)
3          append p to stripL;
4      else if (p is in S2 and p.x - mid.x <= d)
5          append p to stripR;
```

Let the points in `stripL` and `stripR` be  $\{p_0, p_1, \dots, p_k\}$  and  $\{q_0, q_1, \dots, q_l\}$ , as shown in Figure 18.5c. The closest pair between a point in `stripL` and a point in `stripR` can be found using the algorithm described in Listing 18.10.

**Listing 18.10 Algorithm for Finding the Closest Pair in Step 3**

```

1  d = min(d1, d2);
2  r = 0; // r is the index in stripR
3  for (each point p in stripL) {
4      // Skip the points below the rectangle area
5      while (r < stripR.length && q[r].y <= p.y - d)
6          r++;
```

```

7
8   let r1 = r;
9   while (r1 < stripR.length && |q[r1].y - p.y| <= d) {
10      // Check if (p, q[r1]) is a possible closest pair
11      if (distance(p, q[r1]) < d) {
12          d = distance(p, q[r1]);
13          (p, q[r1]) is now the current closest pair;
14      }
15
16      r1 = r1 + 1;
17  }
18 }

```

The points in `stripL` are considered from  $p_0, p_1, \dots, p_k$  in this order. For a point  $p$  in `stripL`, skip the points in `stripR` that are below  $p.y - d$  (lines 5–6). Once a point is skipped, it will no longer be considered. The **while** loop (lines 9–17) checks whether  $(p, q[r1])$  is a possible closest pair. There are at most six such  $q[r1]$  pairs, so the complexity for finding the closest pair in Step 3 is  $O(n)$ .

Let  $T(n)$  denote the time complexity for this algorithm. Thus,

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Therefore, the 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 18.11).

### Check point

- 18.18 What is the divide-and-conquer approach? Give an example.
- 18.19 What is the difference between divide-and-conquer and dynamic programming?
- 18.20 Can you design an algorithm for finding the minimum element in a list using divide-and-conquer? What is the complexity of this algorithm?

## 18.9 Solving the Eight Queens Problem Using Backtracking

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

The Eight Queens problem, introduced in [Section 17.7](#), is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. The problem was solved using recursion. In this section, we will introduce a common algorithm design technique called [backtracking](#) for solving this problem. The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines that the candidate cannot possibly be a valid solution, and then looks for a new candidate.

Recall that we use a one-dimensional array to represent a chessboard, as shown in Figure 17.8a.

```
int queens[8];
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`.

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 the  $j$ th column in the row for  $j = 0, 1, \dots, 7$ , in this order. The search is implemented as follows:

- If successful, it continues 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, it backtracks to the previous row and continues to search for a new placement in the next column 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 18.11](#) gives the program that displays a solution for the Eight Queens problem.

#### Listing 18.11 EightQueenBacktracking.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  const int NUMBER_OF_QUEENS = 8; // Constant: eight queens
5  // queens are placed at (i, queens[i])
6  // -1 indicates that no queen is currently placed in the ith row
7  // Initially, place a queen at (0, 0) in the 0th row
8  int queens[NUMBER_OF_QUEENS] = {-1, -1, -1, -1, -1, -1, -1, -1};
9
10 // Check whether a queen can be placed at row i and column j
```

```

11 bool isValid(int row, int column)
12 {
13     for (int i = 1; i <= row; i++)
14         if (queens[row - i] == column           // Check column
15             || queens[row - i] == column - i    // Check upper left diagonal
16             || queens[row - i] == column + i)   // Check upper right diagonal
17             return false; // There is a conflict
18     return true; // No conflict
19 }
20
21 // Display the chessboard with eight queens
22 void printResult()
23 {
24     cout << "\n-----\n";
25     for (int row = 0; row < NUMBER_OF_QUEENS; row++)
26     {
27         for (int column = 0; column < NUMBER_OF_QUEENS; column++)
28             printf(column == queens[row] ? "| Q " : "|   ");
29         cout << "\n-----\n";
30     }
31 }
32
33 int findPosition(int k)
34 {
35     int start = queens[k] + 1; // Search for a new placement
36
37     for (int j = start; j < NUMBER_OF_QUEENS; j++)
38     {
39         if (isValid(k, j))
40             return j; // (k, j) is the place to put the queen now
41     }
42
43     return -1;
44 }
45
46 // Search for a solution
47 bool search()
48 {
49     // k - 1 indicates the number of queens placed so far
50     // We are looking for a position in the kth row to place a queen
51     int k = 0;
52     while (k >= 0 && k < NUMBER_OF_QUEENS)
53     {
54         // Find a position to place a queen in the kth row
55         int j = findPosition(k);
56         if (j < 0)
57         {
58             queens[k] = -1;
59             k--; // back track to the previous row
60         }
61         else
62         {
63             queens[k] = j;
64             k++;
65         }
66     }
67
68     if (k == -1)
69         return false; // No solution
70     else

```

```

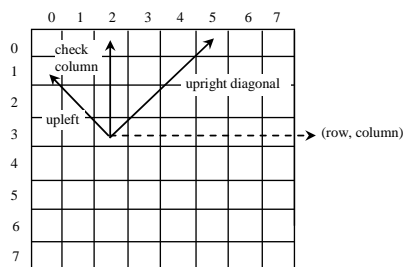
71     return true; // A solution is found
72 }
73
74 int main()
75 {
76     search(); // Start search from row 0. Note row indices are 0 to 7
77     printResult(); // Display result
78
79     return 0;
80 }

```

The program invokes `search()` (line 77) to search for a solution. Initially, no queens are placed in any rows (line 8). The search now starts from the first row with `k = 0` (line 52) and finds a place for the queen (line 56). If successful, place it in the row (line 64) and consider the next row (line 65). If not successful, backtrack to the previous row (lines 59–60).

The `findPosition(k)` function searches for a possible position to place a queen in row `k` starting from `queen[k] + 1` (line 36). It checks whether a queen can be placed at `start`, `start + 1`, ..., and `7`, in this order (lines 38–42). If possible, return the column index (line 41); otherwise, return `-1` (line 44).

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 40). It ensures that no queen is placed in the same column (line 14), in the upper-left diagonal (line 15), or in the upper-right diagonal (line 17), as shown in **Figure 18.6**.



**Figure 18.6**

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

### Check point

24.21 What is backtracking? Give an example.

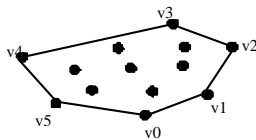
24.22 If you generalize the Eight Queens problem to the  $n$ -Queens problem in an  $n$ -by- $n$  chessboard, what will be the complexity of the algorithm?

### 18.10 Case Studies: Finding a Convex Hull

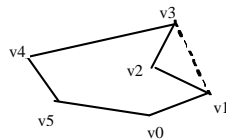
**Key Point:** *This section presents efficient geometric algorithms for finding a convex hull for a set of points.*

Given a set of points, a convex hull is a smallest convex polygon that encloses all these points, as shown in Figure

18.7a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices  $v_0$ ,  $v_1$ ,  $v_2$ ,  $v_3$ ,  $v_4$ , and  $v_5$  in Figure 18.7a form a convex polygon, but not in Figure 18.7b, because the line that connects  $v_3$  and  $v_1$  is not inside the polygon.



(a) A convex hull



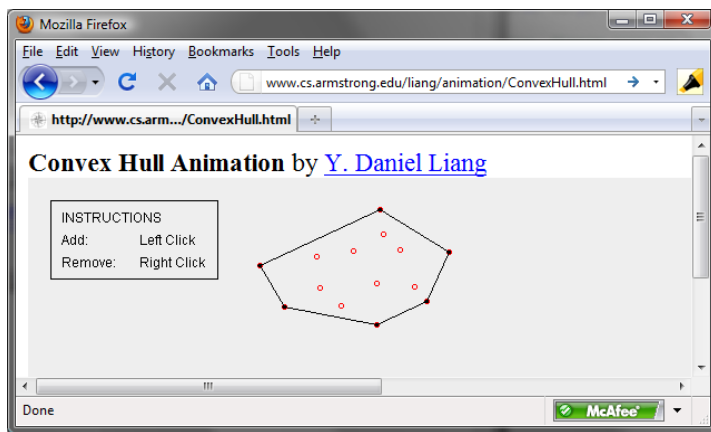
(b) A non-convex polygon

**Figure 18.7**

*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](http://www.cs.armstrong.edu/liang/animation/ConvexHull.html), as shown in Figure 18.8. The tool allows you to add/remove points and displays the convex hull dynamically.





**Figure 18.8**

*The animation tool enables you to add/remove points and display a 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.

### 18.9.1 Gift-Wrapping Algorithm

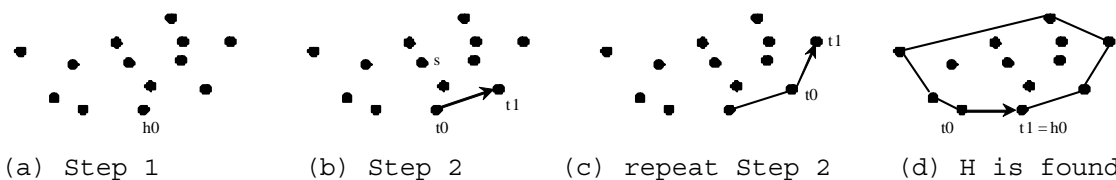
An intuitive approach, called the *gift-wrapping algorithm*, works as follows:

Step 1: Given a set of points  $S$ , let the points in  $S$  be labeled  $s_0, s_1, \dots, s_k$ . Select the rightmost lowest point  $h_0$  in the set  $S$ . As shown in Figure 18.9a,  $h_0$  is such a point. Add  $h_0$  to the convex hull  $H$ .  $H$  is a set 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  $t_1$  then  
         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 18.9b.)

Step 3: If  $t_1$  is  $h_0$  (see Figure 18.9d), 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 18.9c).



**Figure 18.9**

(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 be decided in  $O(1)$  time (see Programming Exercise 3.29). 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 18.11).

### 18.9.2 Graham's Algorithm

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

Step 1: Given a set of points  $S$ , select the rightmost lowest point  $p_0$  in the set  $S$ . As shown in Figure 18.10a,  $p_0$  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 18.10b. 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  $p_0, p_1, p_2, \dots, p_{n-1}$ .

Step 3: Push  $p_0, p_1$ , and  $p_2$  into a stack  $H$ .

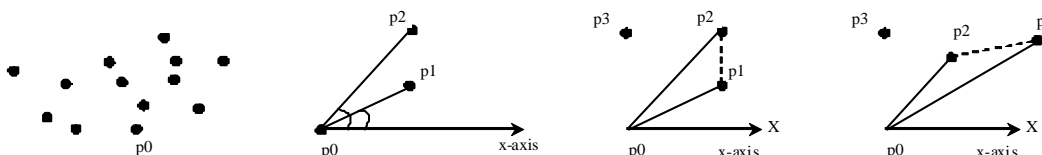
Step 4:

```

i = 3;
while (i < n)
{
    Let t1 and t2 be the top first and second element in stack H;
    if (pi is on the left side of the direct line from t2 to t1)
    {
        Push pi to H;
        i++; // Consider the next point in S.
    }
    else
        Pop the top element off the stack H.
}

```

Step 5: The points in  $H$  form a convex hull.



(a) Step 1                      (b) Step 2                      (c)  $p_3$  into H                      (d)  $p_2$  off H

**Figure 18.10**

(a)  $p_0$  is the rightmost lowest point in  $S$ . (b) points are sorted by the angles. (c-d) A convex hull is discovered incrementally.

*<Margin note: correctness of the algorithm>*

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 18.10c), push  $p_3$  into H. Now  $p_0$ ,  $p_1$ ,  $p_2$ , 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 18.10d), 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 be decided in  $O(1)$  time as shown in Programming Exercise 3.29. 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 19. 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 18.13).

*Check point*

- 18.23 What is a convex hull?
- 18.24 Describe the gift-wrapping algorithm for finding a convex hull. Should list  $H$  be implemented using a stack or a vector?
- 18.25 Describe Graham's algorithm for finding a convex hull. Why does the algorithm use a stack to store the points in a convex hull?

## Key Terms

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

## Chapter Summary

1. 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*.
2. 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 called the *worst-case* input.
3. Best case and worst case are not representative, but worst-case analysis is very useful. You can be sure that the algorithm will never be slower than the worst case.
4. An average-case analysis attempts to determine the average amount of time among all possible inputs of the same size.
5. 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.

6. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation  $O(1)$ .
7. Linear search takes  $O(n)$  time. An algorithm with the  $O(n)$  time complexity is called a *linear algorithm*.
8. Binary search takes  $O(\log n)$  time. An algorithm with the  $O(\log n)$  time complexity is called a *logarithmic algorithm*.
9. The worst-time complexity for selection sort is  $O(n^2)$ .
10. An algorithm with the  $O(n^2)$  time complexity is called a *quadratic algorithm*.
11. The time complexity for the Towers of Hanoi problem is  $O(2^n)$ .
12. An algorithm with the  $O(2^n)$  time complexity is called an *exponential algorithm*.
13. A Fibonacci number at a given index can be found in  $O(n)$  time.
14. Euclid's GCD algorithm takes  $O(\log n)$  time.
15. All prime numbers less than or equal to  $n$  can be found in  $O(\frac{n\sqrt{n}}{\log n})$  time.

## Quiz

Answer the quiz for this chapter online at [www.cs.armstrong.edu/liang/cpp3e/quiz.html](http://www.cs.armstrong.edu/liang/cpp3e/quiz.html).

## Programming Exercises

- \*18.1 (*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

Wel

**\*\*18.2 (Maximum increasingly ordered subsequence)** Write a program that prompts the user to enter a string and displays the maximum increasingly ordered substring. Analyze the time complexity of your program. Here is a sample run:

*Sample output*

```
Enter a string: Welcome ↵ Enter
Welo
```

**\*18.3 (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.* 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 C++ ↵ Enter
Enter a string s2: come ↵ Enter
matched at index 3
```

**\*18.4 (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. Analyze the time complexity of your algorithm. Here is a sample run of the program:

*Sample output*

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

\*18.5 (*Same number subsequence*) Write an  $O(n)$  program that prompts the user to enter a sequence of integers ending with 0 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 4 with 4 values
of 8
```

\*18.6 (*Execution time for GCD*) Write a program that obtains the execution time for finding the GCD of every two consecutive Fibonacci numbers from index 40 to index 45 using the algorithms in Listings 18.2 and 18.3. Your program should print a table like this:

	40	41	42	43	44	45
Listing 18.2 GCD1						
Listing 18.3 GCD2						

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

```
long startTime = time(0);
perform the task;
long endTime = time(0);
long executionTime = endTime - startTime;
```

\*\*18.7 (*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 18.4–18.6. Your program should print a table like this:

	8000000	10000000	12000000	14000000	16000000	18000000
Listing 18.4						
Listing 18.5						
Listing 18.6						



**\*\*18.8** (*All prime numbers up to 1,000,000,000*) Write a program that finds all prime numbers up to 1,000,000,000. There are approximately 50,847,534 such prime numbers. Your program should meet the following requirements:

- Your program should store the prime numbers in a binary data file, named Exercise18\_8.dat. When a new prime number is found, the number is appended to the file.
- To find whether a new number is prime, your program should load the prime numbers from the file to an array of the `long` type of size 100000. If no number in the array is a divisor for the new number, continue to read the next 100000 prime numbers from the data file, until a divisor is found or all numbers in the file are read. If no divisor is found, the new number is prime.
- Since this program takes a long time to finish, you should run it as a batch job from a UNIX machine. If the machine is shut down and rebooted, your program should resume by using the prime numbers stored in the binary data file rather than starting over from the scratch.

**\*\*\*18.9** (*Closest pair of points*) Section 18.8 introduced an algorithm for finding a closest pair of points using a divide-and-conquer approach. Implement the algorithm to meet the following requirements:

- Define a class named `Pair` with data fields `p1` and `p2` to represent two points, and a function named `getDistance()` that returns the distance of the two points.
- Implement the following functions:

```
// Return the closest pair of points
Pair* getClosestPair(vector<vector<double> > points)

// Return the closest pair of points
Pair* getClosestPair(vector<Point>& points)
```

```

// Return the distance of the closest pair of points
// in pointsOrderedOnX[low..high]. This is a recursive
// function. pointsOrderedOnX and pointsOrderedOnY are
// not changed in the subsequent recursive calls.
double distance(vector<Point>& pointsOrderedOnX,
    int low, int high, vector<Point>& pointsOrderedOnY)

// Return the distance between two points p1 and p2
double distance(Point& p1, Point& p2)

// Return the distance between points (x1, y1) and (x2, y2)
double distance(double x1, double y1, double x2, double y2)

```

18.10 (*Last 10 prime numbers*) Programming Exercise 18.8 stores the prime numbers in a file named Exercise18\_8.dat. Write an efficient program that reads the last 10 numbers in the file.

(Hint: Don't read every number. Skip all before the last 10 numbers in the file.)

\*\*18.11 (*Geometry: gift wrapping algorithm for finding a convex hull*) Section 18.9.1 introduced the gift-wrapping algorithm for finding a convex hull for a set of points. Implement the algorithm using the following function:

```

// Return the points that form a convex hull
vector<MyPoint> getConvexHull(vector<MyPoint>& s)

```

MyPoint is class defined as follows:

```

class MyPoint
{
public:
    double x, y;

    MyPoint(double x, double y)
    {
        this->x = x; this->y = y;
    }
};

```

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**

How many points are in the set? 6

↵ Enter

Enter 6 points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9

↵ Enter

The convex hull is

(1.5, 34.5) (5.5, 9.0) (6.0, 2.4) (2.5, 2.0) (1.0, 2.4)

18.12 (*Number of prime numbers*) Programming Exercise 18.8 stores the prime numbers in a file named Exercise18\_8.dat. Write a program that finds the number of prime numbers less than or equal to 10, 100, 1,000, 10,000, 100,000, 1,000,000, and 10,000,000. Your program should read the data from Exercise18\_8.dat. Note that the data file may continue to grow as more prime numbers are stored to it.

\*\*18.13 (*Geometry: convex hull*) Section 18.9.2 introduced Graham's algorithm for finding a convex hull for a set of points. Implement the algorithm using the following function:

```
// Return the points that form a convex hull
vector<MyPoint> getConvexHull(vector<MyPoint>& s)
```

**MyPoint** is defined as in Programming Exercise 9.4. 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

```
How many points are in the set? 6 
Enter 6 points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9 
The convex hull is
(1.5, 34.5) (5.5, 9.0) (6.0, 2.4) (2.5, 2.0) (1.0, 2.4)
```

\*\*\*8.14 (*Game: multiple Sudoku solutions*) The complete solution for the Sudoku problem is given in Supplement VII.A. A Sudoku problem may have multiple solutions. Modify Sudoku.cpp in Supplement VII.A to display the total number of the solutions. Display two solutions if multiple solutions exist.

\*\*8.15 (*Largest block*) The problem for finding a largest block is described in Programming Exercise

12.27. Design a dynamic programming algorithm for solving this problem in  $O(n^2)$  time.