CHAPTER

22

Developing Efficient Algorithms

Objectives

- To estimate algorithm efficiency using the Big *O* notation (§22.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§22.2).
- To determine the complexity of various types of algorithms (§22.3).
- To analyze the binary search algorithm (§22.4.1).
- To analyze the selection sort algorithm (§22.4.2).
- To analyze the Tower of Hanoi algorithm (§22.4.3).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, and exponential) (§22.4.4).
- To design efficient algorithms for finding Fibonacci numbers using dynamic programming (§22.5).
- To find the GCD using Euclid's algorithm (§22.6).
- To find prime numbers using the sieve of Eratosthenes (§22.7).
- To design efficient algorithms for finding the closest pair of points using the divide-and-conquer approach (§22.8).
- To solve the Eight Queens problem using the backtracking approach (§22.9).
- To design efficient algorithms for finding a convex hull for a set of points (§22.10).
- To design efficient algorithms for string matching using Boyer-Moore and KMP algorithms (§22.11).

22.1 Introduction

Algorithm design is to develop a mathematical process for solving a problem.
Algorithm analysis is to predict the performance of an algorithm.

The preceding two chapters introduced classic data structures (lists, stacks, queues, priority queues, sets, and maps) and applied them to solve problems. This chapter will use a variety of examples to introduce common algorithmic techniques (dynamic programming, divide-and-conquer, and backtracking) for developing efficient algorithms. Later in the book, we will introduce efficient algorithms in Chapters 23–29. Before introducing developing efficient algorithms, we need to address the question on how to measure algorithm efficiency.

22.2 Measuring Algorithm Efficiency Using Big *O* Notation

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.

what is algorithm efficiency?

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 times. However, there are two problems with this approach:

- First, many tasks run concurrently on a computer. The actual execution time of a particular program depends on the system load.
- Second, the actual 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 times. To overcome these problems, a theoretical approach was developed to analyze algorithm's running time 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*.

growth rates

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*." We call an algorithm with a time complexity of $O(n)$ linear algorithm, and it exhibits a linear growth rate.

Big *O* notation

Note

The time complexity (a.k.a. running time) of an algorithm is the amount of the time taken by the algorithm measured using the Big O notation.

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-and worst-case analyses are to analyze the algorithms for their best- and worst-case inputs. Best- and worst-case analyses 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

best-case input
worst-case input

average-case analysis

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 the 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 22.1. Therefore, $O(n) = O(n/2) = O(100n)$.

ignoring multiplicative constants

TABLE 22.1 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 and if n is 3, it takes 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 $O(n)$.

large input size

ignoring nondominating terms

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 method 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.

constant time

The following mathematical summations are often useful in algorithm analysis:

useful summations

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

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

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

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

Note

Time complexity is a measure of execution time using the Big O notation. Similarly, you can also measure *space complexity* using the Big O notation. *Space complexity* measures the amount of memory space used by an algorithm. The space complexity for most algorithms presented in this book is $O(n)$, that is, they exhibit linear growth rate to the input size. For example, the space complexity for linear search is $O(n)$.

time complexity

space complexity

Note

We presented the Big O notation in laymen's terms. Appendix J gives a precise mathematical definition for the Big O notation as well as the Big Omega and Big Theta notations.

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

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

$$\frac{(n^2+1)^2}{n}, \frac{(n^2+\log^2 n^2)}{n}, n^3+100n^2+n, 2^n+100n^2+45n, n2^n+n^22^n$$

22.3 Examples: Determining Big *O*

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 (int 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 22.1 to obtain the execution time for n = 1,000,000, 10,000,000, 100,000,000, and 1,000,000,000.

LISTING 22.1 PerformanceTest.java

```
1  public class PerformanceTest {
2    public static void main(String[] args) {
3      getTime(1000000);
4      getTime(10000000);
5      getTime(100000000);
6      getTime(1000000000);
7    }
8
9    public static void getTime(long n) {
10     long startTime = System.currentTimeMillis();
11     long k = 0;
12     for (long i = 1; i <= n; i++) {
13       k = k + 5;
14     }
15     long endTime = System.currentTimeMillis();
16     System.out.println("Execution time for n = " + n
17       + " is " + (endTime - startTime) + " milliseconds");
18   }
19 }
```

input size 1,000,000
input size 10,000,000
input size 100,000,000
input size 1,000,000,000
time before execution
time after execution

```
Execution time for n = 1,000,000 is 6 milliseconds
Execution time for n = 10,000,000 is 61 milliseconds
Execution time for n = 100,000,000 is 610 milliseconds
Execution time for n = 1,000,000,000 is 6048 milliseconds
```

Our analysis predicts a linear time complexity for this loop. As shown in the sample output, when the input size increases 10 times, the runtime increases roughly 10 times. The execution confirms to the prediction.

Example 2

What is the time complexity for the following loop?

```
for (int i = 1; i <= n; i++) {
  for (int 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* and it exhibits a quadratic growth rate. 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.

quadratic time

Example 3

Consider the following loop:

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

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

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

Example 4

Consider the following loop:

```
for (int i = 1; i <= n; i++) {
  for (int 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 (int j = 1; j <= 10; j++) {
  k = k + 4;
}
```

```
for (int i = 1; i <= n; i++) {
  for (int 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 following selection statement:

```
if (list.contains(e)) {
  System.out.println(e);
}
else
  for (Object t: list) {
    System.out.println(t);
  }
```

Suppose the list contains n elements. The execution time for `list.contains(e)` is $O(n)$. The loop in the `else` clause takes $O(n)$ time. Hence, 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:

```
result = 1;
for (int i = 1; i <= n; i++)
  result *= a;
```

The algorithm takes $O(n)$ time. Without loss of generality, assume that $n = 2^k$. 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 22.3.5.)

Note

An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm* and it exhibits a logarithmic growth rate. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. In algorithm analysis, the base is usually 2.

omitting base

22.3.1 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)

22.3.2 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++) {
  System.out.print('*');
}
```

(a)

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    System.out.print('*');
  }
}
```

(b)

```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```

(c)

```
for (int k = 0; k < 10; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```

(d)

22.3.3 Use the Big *O* notation to estimate the time complexity of the following methods:

```
public static void mA(int n) {
  for (int i = 0; i < n; i++) {
    System.out.print(Math.random());
  }
}
```

(a)

```
public static void mB(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(Math.random());
  }
}
```

(b)

```
public static void mC(int[ ] m) {
  for (int i = 0; i < m.length; i++) {
    System.out.print(m[i]);
  }

  for (int i = m.length - 1; i >= 0; )
  {
    System.out.print(m[i]);
    i--;
  }
}
```

(c)

```
public static void mD(int[] m) {
  for (int i = 0; i < m.length; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(m[i] * m[j]);
  }
}
```

(d)

22.3.4 Design an $O(n)$ time algorithm for computing the sum of numbers from $n1$ to $n2$ for ($n1 < n2$). Can you design an $O(1)$ for performing the same task?

22.3.5 Example 7 in Section 22.3 assumes $n = 2^k$. Revise the algorithm for an arbitrary n and prove that the complexity is still $O(\log n)$.

22.4 Analyzing Algorithm Time Complexity

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

binary search animation on the Companion Website

22.4.1 Analyzing Binary Search

The binary search algorithm presented in Listing 7.7, BinarySearch.java, 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,

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

logarithmic time

Ignoring constants and nondominating terms, the complexity of the binary search algorithm is $O(\log n)$. This is a logarithmic algorithm. 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. Therefore, a logarithmic-time algorithm is very efficient.

selection sort animation on the Companion Website

22.4.2 Analyzing Selection Sort

The selection sort algorithm presented in Listing 7.8, SelectionSort.java, 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,

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

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

22.4.3 Analyzing the Tower of Hanoi Problem

The Tower of Hanoi problem presented in Listing 18.8, TowerOfHanoi.java, 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 with $T(1) = 1$. Thus,

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

An algorithm with $O(2^n)$ time complexity is called an *exponential algorithm* and it exhibits an exponential growth rate. 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.

$O(2^n)$ exponential time

22.4.4 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 Tower of Hanoi is $T(n) = T\left(\frac{n}{2}\right) + c$, $T(n) = T(n-1) + O(n)$, and $T(n) = 2T(n-1) + O(1)$, respectively.

Table 22.2 summarizes the common recurrence relations.

TABLE 22.2 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 22.8.3
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 23)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

22.4.5 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. Table 22.3 lists some common growth functions and shows how growth rates change as the input size doubles from $n = 25$ to $n = 50$.

TABLE 22.3 Change of Growth Rates

Function	*Name*	$n = 25$	$n = 50$	$f(50)$ and $f(25)$
$O(1)$	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	3.36×10^7	1.27×10^{15}	3.35×10^7

These functions are ordered as follows, as illustrated in Figure 22.1.

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

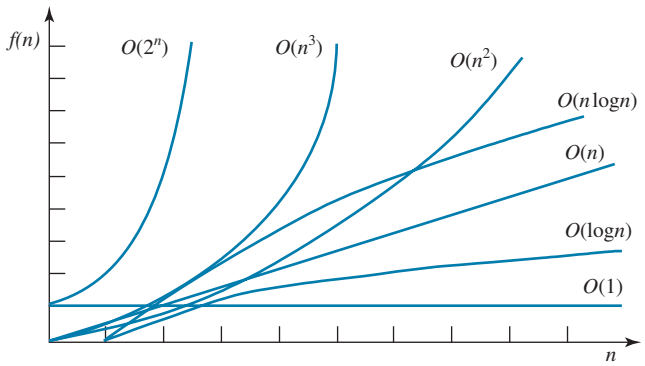


FIGURE 22.1 As the size n increases, the function grows.

Check Point

22.4.1 Put the following growth functions in order:

$$\frac{5n^3}{4{,}032}, 44 \log n, 10n \log n, 500, 2n^2, \frac{2^n}{45}, 3n$$

22.4.2 Estimate the time complexity for adding two $n \times m$ matrices and multiplying an $n \times m$ matrix by an $m \times k$ matrix.

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

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

22.4.5 Analyze the following sorting algorithm:

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

22.4.6 Analyze the complexity for computing a polynomial $f(x)$ of degree n for a given x value using a brute-force approach and the Horner's approach, respectively. A brute-force approach is to compute each term in the polynomial and add them together. The Horner's approach was introduced in Section 6.7.

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x^1 + a_0$$

22.5 Finding Fibonacci Numbers Using Dynamic Programming

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

Section 18.3, Case Study: Computing Fibonacci Numbers, gave a recursive method for finding the Fibonacci number, as follows:

```
/** The method 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 fib(n), and c denote the constant time for comparing the index with `0` and `1`. 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 Tower 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 with the recursive `fib` method is that the method 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 `fib(2)` is redundantly invoked. We can improve it by avoiding repeatedly calling of the `fib` method with the same argument. Note 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 22.2.

```
                  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
```

FIGURE 22.2 Variables `f0`, `f1`, and `f2` store three consecutive Fibonacci numbers in the series.

The new method is implemented in Listing 22.2.

LISTING 22.2 ImprovedFibonacci.java

```
1  import java.util.Scanner;
2
3  public class ImprovedFibonacci {
4    /** Main method */
5    public static void main(String args[]) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter an index for the Fibonacci number: ");
9      int index = input.nextInt();
10
11     // Find and display the Fibonacci number
12     System.out.println(
13       "Fibonacci number at index " + index + " is " + fib(index));
14   }
15
16   /** The method for finding the Fibonacci number */
17   public static long fib(long n) {
18     long f0 = 0; // For fib(0)
19     long f1 = 1; // For fib(1)
20     long f2 = 1; // For fib(2)
21
22     if (n == 0)
23       return f0;
24     else if (n == 1)
25       return f1;
26     else if (n == 2)
27       return f2;
28
29     for (int i = 3; i <= n; i++) {
30       f0 = f1;
31       f1 = f2;
32       f2 = f0 + f1;
33     }
34
35     return f2;
36   }
37 }
```

input

invoke fib

f0
f1
f2

update f0, f1, f2

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

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

$O(n)$

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

Algorithm Design Note

dynamic programming

The algorithm for computing Fibonacci numbers presented here uses an approach known as *dynamic programming*. Dynamic programming is the process of solving subproblems, and 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.

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

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

22.6 Finding Greatest Common Divisors Using Euclid's Algorithm

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

GCD

The greatest common divisor (GCD) of two integers is the largest number that divides both integers. Listing 5.9, GreatestCommonDivisor.java, presented a brute-force algorithm for finding the GCD of two integers `m` and `n`.

brute force

Algorithm Design Note

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.

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 $O(n)$.

assume $m \geq n$

$O(n)$

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:

improved solutions

```
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 22.3.

LISTING 22.3 GCD.java

```
1  import java.util.Scanner;
2
3  public class GCD {
4    /** Find GCD for integers m and n */
5    public static int gcd(int m, int n) {
6      int gcd = 1;
7
8      if (m % n == 0)  return n;
9
10     for (int k = n /  2; k >= 1; k--) {
11       if (m % k == 0 && n % k == 0) {
12         gcd = k;
13         break;
14       }
15     }
16
17     return gcd;
18   }
19
20   /** Main method */
21   public static void main(String[] args) {
22     // Create a Scanner
23     Scanner input = new Scanner(System.in);
24
25     // Prompt the user to enter two integers
26     System.out.print("Enter first integer: ");
27     int m = input.nextInt();
28     System.out.print("Enter second integer: ");
29     int n = input.nextInt();
30
31     System.out.println("The greatest common divisor for " + m +
32       " and " + n + " is " + gcd(m, n));
33   }
34 }
```

check divisor

GCD found

input

input

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

```
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 5.9.

$O(n)$

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 Listing 5.9 and Listing 22.3 have the same complexity, but in practice, the one in Listing 22.3 is obviously better.

practical consideration

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:

Euclid's algorithm

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 divides `m` and `n` evenly must also divide `r` evenly. Therefore, `gcd(m, n)` is the same as `gcd(n, r)`, where `r = m % n`. The algorithm can be implemented as in Listing 22.4.

LISTING 22.4 GCDEuclid.java

```
1  import java.util.Scanner;
2
3  public class GCDEuclid {
4    /** Find GCD for integers m and n */
5    public static int gcd(int m, int n) {
6      if (m % n == 0)                                    base case
7        return n;
8      else
9        return gcd(n, m % n);                            reduction
10   }
11
12   /** Main method */
13   public static void main(String[] args) {
14     // Create a Scanner
15     Scanner input = new Scanner(System.in);
16
17     // Prompt the user to enter two integers
18     System.out.print("Enter first integer: ");
19     int m = input.nextInt();                           input
20     System.out.print("Enter second integer: ");
21     int n = input.nextInt();                           input
22
23     System.out.println("The greatest common divisor for " + m +
24       " and " + n + " is " + gcd(m, n));
25   }
26 }
```

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

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

best case
average case
worst case

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 <= 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` method. 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` method 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` method 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, $k \leq 2 \log n$. Thus, the time complexity of the `gcd` method 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:

$$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ 55\ 89 \ldots$$

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` method 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 22.4 summarizes the complexity of three algorithms for finding the GCD.

TABLE 22.4 Comparisons of GCD Algorithms

Algorithm	*Complexity*	*Description*
Listing 5.9	$O(n)$	Brute-force, checking all possible divisors
Listing 22.3	$O(n)$	Checking half of all possible divisors
Listing 22.4	$O(\log n)$	Euclid's algorithm

22.6.1 Prove the following algorithm for finding the GCD of the two integers `m` and `n` is incorrect:


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

22.7 Efficient Algorithms for Finding Prime Numbers

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 (https://www.eff.org/awards/coop).

Can you design a fast algorithm for finding prime numbers?

what is prime?

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 5.15 presented a brute-force algorithm for finding prime numbers. The algorithm checks whether `2`, `3`, `4`, `5`, . . . , or `n - 1` divides `n`. If not, `n` is prime. This algorithm takes $O(n)$ time to check whether `n` is prime. Note you need to check only whether `2`, `3`, `4`, `5`, . . . , or `n/2` divides `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 $\sqrt{n}$ divides `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`, . . . , `n`. The program is given in Listing 22.5.

LISTING 22.5 PrimeNumbers.java

```
1  import java.util.Scanner;
2
3  public class PrimeNumbers {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      System.out.print("Find all prime numbers <= n, enter n: ");
7      int n = input.nextInt();
8
9      final int NUMBER_PER_LINE = 10; // Display 10 per line
10     int count = 0; // Count the number of prime numbers
11     int number = 2; // A number to be tested for primeness
12
13     System.out.println("The prime numbers are:");
14
15     // Repeatedly find prime numbers
16     while (number <= n) {
17       // Assume the number is prime
18       boolean isPrime = true; // Is the current number prime?
19
20       // Test if number is prime
check prime  21       for (int divisor = 2; divisor <= (int)(Math.sqrt(number));
22           divisor++) {
23         if (number % divisor == 0) {  // If true, number is not prime
24           isPrime = false; // Set isPrime to false
25           break; // Exit the for loop
26         }
27       }
28
29       // Print the prime number and increase the count
increase count  30       if (isPrime) {
31         count++; // Increase the count
32
33         if (count % NUMBER_PER_LINE == 0) {
34           // Print the number and advance to the new line
35           System.out.printf("%7d\n", number);
36         }
37         else
38           System.out.printf("%7d", number);
39       }
40
41       // Check if the next number is prime
check next number  42       number++;
43     }
44
45     System.out.println("\n" + count +
46       " prime(s) less than or equal to " + n);
47   }
48 }
```

```
Find all prime numbers <= n, enter n: 1000 Enter
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
```

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

```
int squareRoot = (int)(Math.sqrt(number));
for (int divisor = 2; divisor <= squareRoot; divisor++) {
```

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

```
...
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 if 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 21–27) to check whether number i is prime, the algorithm takes $\sqrt{2} + \sqrt{3} + \sqrt{4} + \cdots + \sqrt{n}$ steps to find all the prime numbers less than or equal to n. Observe that

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

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

To determine whether i is prime, the algorithm checks whether `2`, `3`, `4`, `5`, . . . , or $\sqrt{i}$ divides 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 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 divides i. This leads to a more efficient algorithm for finding all prime numbers up to `n`, as given in Listing 22.6.

LISTING 22.6 EfficientPrimeNumbers.java

```
1  import java.util.Scanner;
2
3  public class EfficientPrimeNumbers {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      System.out.print("Find all prime numbers <= n, enter n: ");
```

```
 7      int n = input.nextInt();
 8
 9      // A list to hold prime numbers
10      java.util.List<Integer> list =
11        new java.util.ArrayList<>();
12
13      final int NUMBER_PER_LINE = 10; // Display 10 per line
14      int count = 0; // Count the number of prime numbers
15      int number = 2; // A number to be tested for primeness
16      int squareRoot = 1; // Check whether number <= squareRoot
17
18      System.out.println("The prime numbers are \n");
19
20      // Repeatedly find prime numbers
21      while (number <= n) {
22        // Assume the number is prime
23        boolean isPrime = true; // Is the current number prime?
24
25        if (squareRoot * squareRoot < number) squareRoot++;
26
27        // Test whether number is prime
28        for (int k = 0; k < list.size()
29                          && list.get(k) <= squareRoot; k++) {
30          if (number % list.get(k) == 0) {  // If true, not prime
31            isPrime = false; // Set isPrime to false
32            break; // Exit the for loop
33          }
34        }
35
36        // Print the prime number and increase the count
37        if (isPrime) {
38          count++; // Increase the count
39          list.add(number); // Add a new prime to the list
40          if (count % NUMBER_PER_LINE == 0) {
41            // Print the number and advance to the new line
42            System.out.println(number);
43          }
44          else
45            System.out.print(number + " ");
46        }
47
48        // Check whether the next number is prime
49        number++;
50      }
51
52      System.out.println("\n" + count +
53        " prime(s) less than or equal to " + n);
54    }
55  }
```

check prime

increase count

check next number

```
Find all prime numbers <= n, enter n: 1000 Enter
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
```

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**. Therefore, $\pi(2)$ 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).

For each number **i**, the algorithm checks whether a prime number less than or equal to $\sqrt{i}$ divides 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

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

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

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

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. dynamic programming

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 *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 22.3. Since we don't care about **primes[0]** and **primes[1]**, these values are marked $\times$ in the figure. Sieve of Eratosthenes

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	×	×	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
$k = 2$	×	×	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
$k = 3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k = 5$	×	×	Ⓣ	Ⓣ	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	F	F	F

FIGURE 22.3 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 $3 \leq i \leq n/3$. Because the multiples of **5** are not prime, set **primes[5 * i]** to **false** for all $5 \leq i \leq n/5$. Note 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**, and **9** need not be considered. You only need to consider the multiples of a prime number **k = 2**, **3**, **5**, **7**, **11**, . . . , 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 22.3, **2**, **3**, **5**, **7**, **11**, **13**, **17**, **19**, and **23** are prime numbers. Listing 22.7 gives the program for finding the prime numbers using the Sieve of Eratosthenes algorithm.

LISTING 22.7 SieveOfEratosthenes.java

```
1  import java.util.Scanner;
2
3  public class SieveOfEratosthenes {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      System.out.print("Find all prime numbers <= n, enter n: ");
7      int n = input.nextInt();
8
9      boolean[] primes = new boolean[n + 1]; // Prime number sieve
10
11     // Initialize primes[i] to true
12     for (int i = 0; i < primes.length; i++) {
13       primes[i] = true;
14     }
15
16     for (int k = 2; k <= n / k; k++) {
17       if (primes[k]) {
18         for (int i = k; i <= n / k; i++) {
19           primes[k * i] = false; // k * i is not prime
20         }
21       }
22     }
23
24     final int NUMBER_PER_LINE = 10; // Display 10 per line
25     int count = 0; // Count the number of prime numbers found so far
26     // Print prime numbers
27     for (int i = 2; i < primes.length; i++) {
28       if (primes[i]) {
29         count++;
30         if (count % NUMBER_PER_LINE == 0)
31           System.out.printf("%7d\n", i);
32         else
33           System.out.printf("%7d", i);
34       }
35     }
36
37     System.out.println("\n" + count +
38       " prime(s) less than or equal to " + n);
39   }
40 }
```

sieve

initialize sieve

nonprime

```
Find all prime numbers <= n, enter n: 1000 Enter
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 `k <= n / k` (line 16). 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

$$\frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \ldots$$

$$= O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \cdots\right) < O(n\pi(n))$$

$$= O\left(n\frac{\sqrt{n}}{\log n}\right)$$

The number of items in the series is $\pi(n)$.

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 22.5 summarizes the complexity of these three algorithms for finding all prime numbers up to n.

TABLE 22.5 Comparisons of Prime-Number Algorithms

Algorithm	*Complexity*	*Description*
Listing 5.15	$O(n^2)$	Brute-force, checking all possible divisors
Listing 22.5	$O\left(n\sqrt{n}\right)$	Checking divisors up to $\sqrt{n}$
Listing 22.6	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Checking prime divisors up to $\sqrt{n}$
Listing 22.7	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Sieve of Eratosthenes

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

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

Check Point

22.8 Finding the Closest Pair of Points Using Divide-and-Conquer

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

Key Point

Pedagogical Note

Starting from this section, we will present interesting and challenging problems. It is time that you begin to study advanced algorithms to become a proficient programmer. We recommend that you study the algorithms and implement them in the exercises.

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

closest-pair animation on Companion Website

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

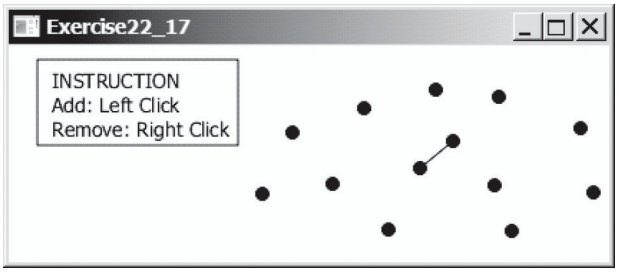


FIGURE 22.4 The closet-pair animation draws a line to connect the closest pair of points dynamically as points are added and removed interactively. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

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 can use an approach called *divide-and-conquer* to solve this problem efficiently in $O(n \log n)$ time.

divide-and-conquer

Algorithm Design Note

The *divide-and-conqure* approach divides the problem into subproblems, solves the subproblems, and 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 22.8 describes how to solve the closest pair problem using the divide-and-conquer approach.

LISTING 22.8 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 23, 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 22.5a.

For a point p in `stripL`, you need only consider a right point within the $d \times 2d$ rectangle, as shown in 22.5b. 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 given in Listing 22.9.

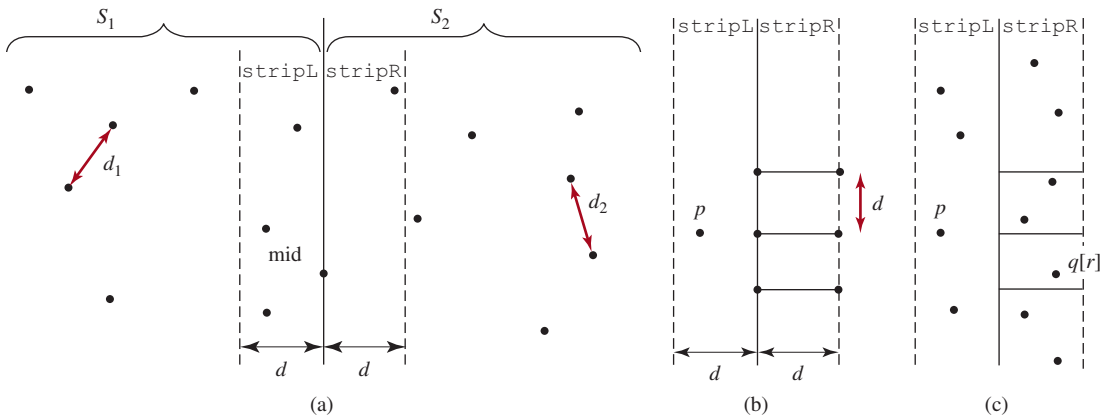


FIGURE 22.5 The midpoint divides the points into two sets of equal size.

LISTING 22.9 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;                                  stripL
4    else if (p is in S2 and p.x - mid.x <= d)
5      append p to stripR;                                  stripR
```

Let the points in **stripL** and **stripR** be $\{p_0, p_1, \ldots, p_k\}$ and $\{q_0, q_1, \ldots, q_t\}$, as shown in Figure 22.5c. The closest pair between a point in **stripL** and a point in **stripR** can be found using the algorithm described in Listing 22.10.

LISTING 22.10 Algorithm for Finding the Closest Pair in Step 3

```
1  d = min(d1, d2);
2  r = 0; // r is the index of a point in stripR
3  for (each point p in stripL) {
4    // Skip the points in stripR below p.y - d
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]);                            update closest pair
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 $p_0, p_1, \ldots, p_k$ 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, because the distance between two points in **stripR** cannot be less than **d**. Thus, the complexity for finding the closest pair in Step 3 is $O(n)$.

Note Step 1 in Listing 22.8 is performed only once to presort the points. Assume all the points are presorted. Let $T(n)$ denote the time complexity for this algorithm. Thus,

$$T(n) = \underbrace{2T(n/2)}_{\text{Step 2}} + \underbrace{O(n)}_{\text{Step 3}} = O(n \log n)$$

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 22.7).

22.8.1 What is divide-and-conquer approach? Give an example.

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

22.8.3 Can you design an algorithm for finding the minimum element in a list using divide-and-conquer? What is the complexity of this algorithm?

22.9 Solving the Eight Queens Problem Using Backtracking

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. The problem can be solved using recursion (see Programming Exercise 18.34). In this section, we will introduce a common algorithm design technique called *backtracking* for solving this problem.

backtracking

Algorithm Design Note

There are many possible candidates? How do you find a solution? 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.

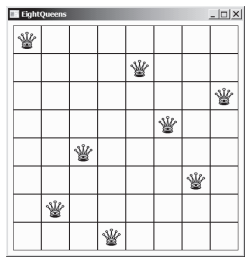
You can use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row. Thus, you can define the `queens` array as

```
int[] queens = new int[8];
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`. Figure 22.6 shows the contents of the `queens` array for the chessboard.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3

(a)



(b)

FIGURE 22.6 `queens[i]` denotes the position of the queen in row `i`. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

search algorithm

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 jth column in the row for $j = 0, 1, \ldots, 7$, 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.

Eight Queens animation on the Companion Website

To see how the algorithm works, go to http://liveexample.pearsoncmg.com/dsanimation/EightQueens.html.

Listing 22.11 gives the program that displays a solution for the Eight Queens problem.

LISTING 22.11 EightQueens.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.stage.Stage;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Label;
6  import javafx.scene.image.Image;
7  import javafx.scene.image.ImageView;
8  import javafx.scene.layout.GridPane;
9
10 public class EightQueens extends Application {
11   public static final int SIZE = 8; // The size of the chessboard
12   // queens are placed at (i, queens[i])
13   // -1 indicates that no queen is currently placed in the ith row
14   // Initially, place a queen at (0, 0) in the 0th row
15   private int[] queens = {-1, -1, -1, -1, -1, -1, -1, -1};                queen positions
16
17   @Override // Override the start method in the Application class
18   public void start(Stage primaryStage) {
19     search(); // Search for a solution                                    search for solution
20
21     // Display chessboard
22     GridPane chessBoard = new GridPane();
23     chessBoard.setAlignment(Pos.CENTER);
24     Label[][] labels = new Label[SIZE][SIZE];
25     for (int i = 0; i < SIZE; i++)
26       for (int j = 0; j < SIZE; j++) {
27         chessBoard.add(labels[i][j] = new Label(), j, i);                 create cells
28         labels[i][j].setStyle("-fx-border-color: black");
29         labels[i][j].setPrefSize(55, 55);
30       }
31
32     // Display queens
33     Image image = new Image("image/queen.jpg");
34     for (int i = 0; i < SIZE; i++)
35       labels[i][queens[i]].setGraphic(new ImageView(image));              set queen image
36
37     // Create a scene and place it in the stage
38     Scene scene = new Scene(chessBoard, 55 * SIZE, 55 * SIZE);
39     primaryStage.setTitle("EightQueens"); // Set the stage title
40     primaryStage.setScene(scene); // Place the scene in the stage
41     primaryStage.show(); // Display the stage
42   }
43
44   /** Search for a solution */
45   private boolean search() {
46     // k - 1 indicates the number of queens placed so far
47     // We are looking for a position in the kth row to place a queen
48     int k = 0;
49     while (k >= 0 && k < SIZE) {
50       // Find a position to place a queen in the kth row
51       int j = findPosition(k);                                            find a column
52       if (j < 0) {
53         queens[k] = -1;
54         k--; // Backtrack to the previous row                             backtrack
55       }  else {
```

place a queen
search the next row

```
56            queens[k] = j;
57            k++;
58          }
59        }
60
61        if (k == -1)
62          return false; // No solution
63        else
64          return true; // A solution is found
65      }
66
67      public int findPosition(int k) {
68        int start = queens[k] + 1; // Search for a new placement
69
70        for (int j = start; j < SIZE; j++) {
71          if (isValid(k, j))
72            return j; // (k, j) is the place to put the queen now
73        }
74
75        return -1;
76      }
77
78      /** Return true if a queen can be placed at (row, column) */
79      public boolean isValid(int row, int column) {
80        for (int i = 1; i <= row; i++)
81          if (queens[row - i] == column // Check column
82            || queens[row - i] == column - i // Check upleft diagonal
83            || queens[row - i] == column + i)  // Check upright diagonal
84            return false; // There is a conflict
85        return true; // No conflict
86      }
87    }
```

The program invokes `search()` (line 19) to search for a solution. Initially, no queens are placed in any rows (line 15). The search now starts from the first row with `k = 0` (line 48) and finds a place for the queen (line 51). If successful, place it in the row (line 56) and consider the next row (line 57). If not successful, backtrack to the previous row (lines 53–54).

The `findPosition(k)` method searches for a possible position to place a queen in row `k` starting from `queen[k] + 1` (line 68). It checks whether a queen can be placed at `start`, `start + 1`, . . . , and `7`, in this order (lines 70–73). If possible, return the column index (line 72); otherwise, return `-1` (line 75).

The `isValid(row, column)` method is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier (line 71). It ensures that no queen is placed in the same column (line 81), in the upper-left diagonal (line 82), or in the upper-right diagonal (line 83), as shown in Figure 22.7.

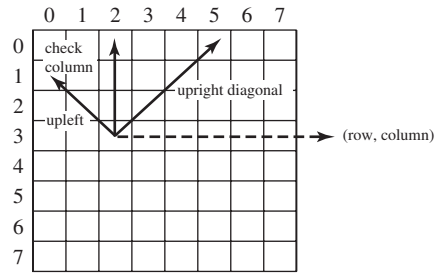


FIGURE 22.7 Invoking `isValid(row, column)` checks whether a queen can be placed at (row, column).

22.9.1 What is backtracking? Give an example.

22.9.2 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?

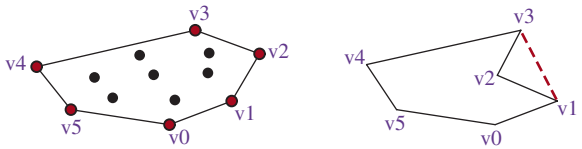
22.10 Computational Geometry: Finding a Convex Hull

This section presents efficient geometric algorithms for finding a convex hull for a set of points.

Computational geometry is to study the algorithms for geometrical problems. It has applications in computer graphics, games, pattern recognition, image processing, robotics, geographical information systems, and computer-aided design and manufacturing. Section 22.8 presented a geometrical algorithm for finding the closest pair of points. This section introduces geometrical algorithms for finding a convex hull.

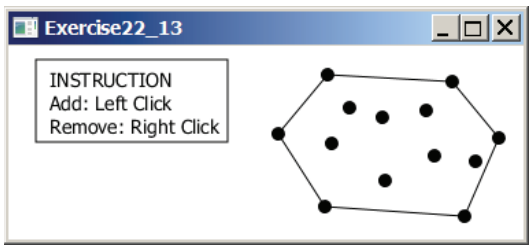
convex hull

Given a set of points, a *convex hull* is the smallest convex polygon that encloses all these points, as shown in Figure 22.8a. 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 22.8a form a convex polygon, but not in Figure 22.8b, because the line that connects v3 and v1 is not inside the polygon.



(a) A convex hull

(b) A nonconvex polygon



(c) Convex hull animation

FIGURE 22.8 A convex hull is the smallest convex polygon that contains a set of points. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

convex hull animation on the Companion Website

A 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 Listing.pearsoncmg.com/dsanimation/ConvexHull.html, as shown in Figure 22.8c. This tool allows you to add and remove points and displays the convex hull dynamically.

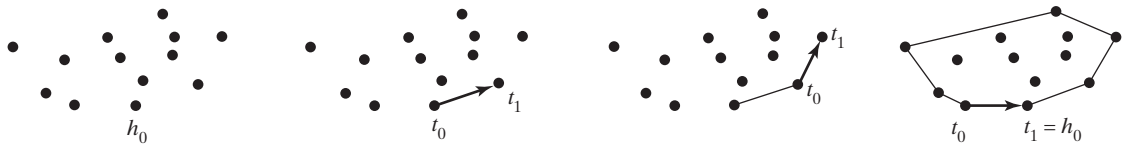
Many algorithms have been developed to find a convex hull. This section introduces two popular algorithms: the gift-wrapping algorithm and Graham's algorithm.

22.10.1 Gift-Wrapping Algorithm

An intuitive approach called the *gift-wrapping algorithm* works as described in Listing 22.12.

LISTING 22.12 Finding a Convex Hull Using Gift-Wrapping Algorithm

```
Step 1: Given a list of points S, let the points in S be labeled
s0, s1, . . . , sk. Select the rightmost lowest point S. As shown in
Figure 22.9a, h0 is such a point. Add h0 to list H. (H is initially emp-
ty. H will hold all points in the convex hull after the algorithm is
finished.) Let t0 be h0.
```



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

FIGURE 22.9 (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.

```
Step 2: Let t1 be s0.
        For every point p in S,
          if p is on the right side of the direct line from t0 to t1, then
            let t1 be p.
(After Step 2, no points lie on the right side of the direct line from t0
to t1, as shown in Figure 22.9b.)

Step 3: If t1 is h0 (see Figure 22.9d), the points in H form a convex
hull for S. Otherwise, add t1 to H, let t0 be t1, and go back to Step 2
(see Figure 22.9c).
```

correctness of the algorithm

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.

time complexity of the algorithm

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 determined in $O(1)$ time (see Programming Exercise 3.32). 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.

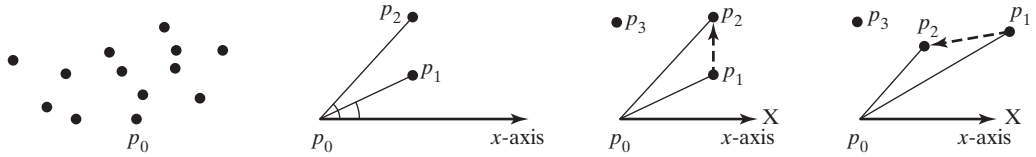
The implementation of this algorithm is left as an exercise (see Programming Exercise 22.9).

22.10.2 Graham's Algorithm

A more efficient algorithm was developed by Ronald Graham in 1972, as given in Listing 22.13.

LISTING 22.13 Finding a Convex Hull Using Graham's Algorithm

```
Step 1: Given a list of points S, select the rightmost lowest point and
name it p0. As shown in Figure 22.10a, p0 is such a point.
```



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

FIGURE 22.10 (a) p_0 is the rightmost lowest point in S. (b) Points are sorted by their angles. (c–d) A convex hull is discovered incrementally.

```
Step 2: Sort the points in S angularly along the x-axis with p0 as the
center, as shown in Figure 22.10b. If there is a tie and two points have
the same angle, discard the one that is closer to p0. The points in S are
now sorted as p0, p1, p2, . . . , pn-1.

Step 3: Push p0, p1, and p2 into stack H. (After the algorithm finishes,
H contains all the points in the convex hull.)
```

```
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 stack H.
     }
Step 5: The points in H form a convex hull.
```

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 22.10c), 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 22.10d), 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 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 list S.

correctness of the algorithm

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 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 determined in $O(1)$ time, as shown in Programming Exercise 3.32. Sorting in Step 2 can be done in $O(n \log n)$ time using the merge-sort or heap-sort algorithms that will be introduced in Chapter 23. Step 4 can be done in $O(n)$ time. Therefore, the algorithm takes $O(n \log n)$ time.

time complexity of the algorithm

The implementation of this algorithm is left as an exercise (see Programming Exercise 22.11).

22.10.1 What is a convex hull?

22.10.2 Describe the gift-wrapping algorithm for finding a convex hull. Should list `H` be implemented using an `ArrayList` or a `LinkedList`?

22.10.3 Describe Graham's algorithm for finding a convex hull. Why does the algorithm use a stack to store the points in a convex hull?

22.11 String Matching

This section presents the brute force, Boyer-Moore, and Knuth-Morris-Pratt algorithms for string matching.

String matching is to find a match for a substring in a string. The string is commonly known as the text and the substring is called a pattern. String matching is a common task in computer programming. The `String` class has the `text.contains(pattern)` method to test if a pattern is in a text and the `text.indexOf(pattern)` method to return the index of the first matching of the pattern in the text. A lot of research has been done to find efficient algorithms for string matching. This section presents three algorithms: the brute force algorithm, the Boyer-Moore algorithm, and the Knuth-Morris-Pratt algorithm.

The brute force algorithm is simply to compare the pattern with every possible substring in the text. Assume the length of text and pattern are `n` and `m`, respectively. The algorithm can be described as follows:

```
for i from 0 to n – m {
  test if pattern matches text[i .. i + m – 1]
}
```

Here text[i ... j] represents a substring in text from index i and index j. For an animation of this algorithm, see https://liveexample.pearsoncmg.com/dsanimation/StringMatch.html.

Listing 22.14 gives an implementation for the brute-force algorithm.

LISTING 22.14 StringMatch.java

```
1  public class StringMatch {
2    public static void main(String[] args) {
3      java.util.Scanner input = new java.util.Scanner(System.in);
4      System.out.print("Enter a string text: ");
5      String text = input.nextLine();
6      System.out.print("Enter a string pattern: ");
7      String pattern = input.nextLine();
8
9      int index = match(text, pattern);
10     if (index >= 0)
11       System.out.println("matched at index "+ index);
12     else
13       System.out.println("unmatched");
14   }
15
16   // Return the index of the first match. -1 otherwise.
17   public static int match(String text, String pattern) {
18     for (int i = 0; i < text.length() - pattern.length() + 1; i++) {
19       if (isMatched(i, text, pattern))
20         return i;
21     }
22
23     return -1;
24   }
25
26   // Test if pattern matches text starting at index i
27   private static boolean isMatched(int i, String text,
28       String pattern) {
29     for (int k = 0; k < pattern.length(); k++) {
30       if (pattern.charAt(k) != text.charAt(i + k)) {
31         return false;
32       }
33     }
34
35     return true;
36   }
37 }
```

The `match(text, pattern)` method (lines 17–24) tests whether `pattern` matches a substring in `text`. The `isMatched(i, text, pattern)` method (lines 27–36) tests whether `pattern` matches `text[i ... i + m - 1]` starting at index `i`.

Clearly, the algorithm takes `O(nm)` time, since testing whether `pattern` matches `text[i ... i + m - 1]` takes `O(m)` time.

22.11.1 The Boyer-Moore Algorithm

The brute force algorithm searches for a match of the pattern in the text by examining all alignments. This is not necessary. The Boyer-Moore algorithm finds a matching by comparing the pattern with a substring in the text from right to left. If a character in the text does not match the one in the pattern and this character is not in the remaining part of the pattern, you can slide the pattern all the way passing this character. This can be best illustrated using the animation at https://liveexample.pearsoncmg.com/dsanimation/StringMatchBoyerMoore.html:

The Boyer-Moore algorithm can be described as follows:

```
i = m - 1;
while i <= n - 1
  Compare pattern with text[i - (m - 1) .. i]
  from right to left one by one, as shown in Figure 22.11.
  If they all match, done. Otherwise, let text[k] be the first one
  that does not match the corresponding character in pattern.

  Consider two cases:

  Case 1: If text[k] is not in the remaining pattern, slide the pat-
  tern passing text[k], as shown in Figure 22.12. Set i = k + m;

  Case 2: If text[k] is in pattern, find the last character, say
  pattern[j] in pattern that matches text[k] and slide the pattern
  right to align pattern[j] with text[k], as shown in Figure 22.13.
  Set i = k + m - j - 1.
```

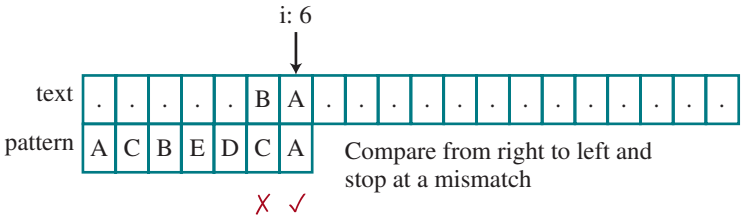


FIGURE 22.11 Test if the pattern matches a substring by comparing the characters from right to left and stop at a mismatch.

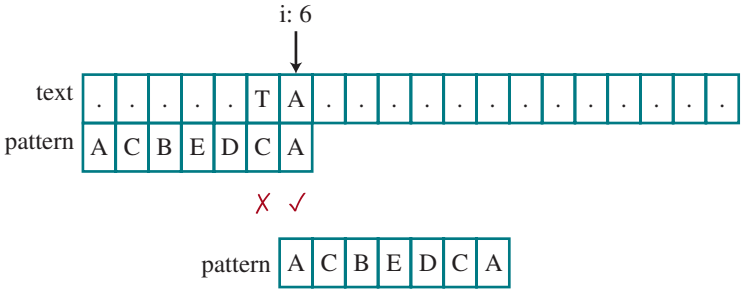


FIGURE 22.12 Since T is not in the remaining of the pattern, slide the pattern passing T and start the next test.

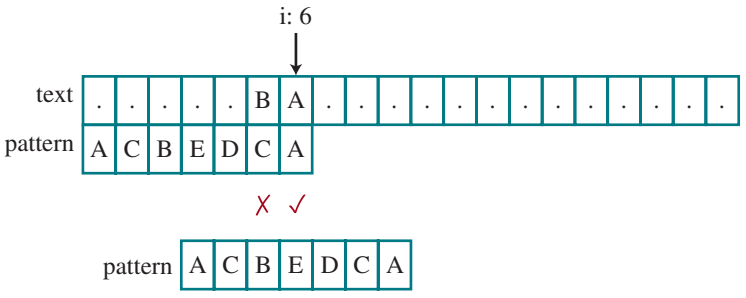


FIGURE 22.13 B does not match C. B first matches pattern[2] in the remaining part of the pattern from right to left, slide the pattern to align B in the text with pattern[2].

Listing 22.15 gives an implementation for the Boyer-Moore algorithm.

LISTING 22.15 StringMatchBoyerMoore.java

```
1  public class StringMatchBoyerMoore {
2    public static void main(String[] args) {
3      java.util.Scanner input = new java.util.Scanner(System.in);
4      System.out.print("Enter a string text: ");
5      String text = input.nextLine();
6      System.out.print("Enter a string pattern: ");
7      String pattern = input.nextLine();
8
9      int index = match(text, pattern);
10     if (index >= 0)
11       System.out.println("matched at index "+ index);
12     else
13       System.out.println("unmatched");
14   }
15
16   // Return the index of the first match. -1 otherwise.
17   public static int match(String text, String pattern) {
18     int i = pattern.length() - 1;
19     while (i < text.length()) {
20       int k = i;
21       int j = pattern.length() - 1;
22       while (j >= 0) {
23         if (text.charAt(k) == pattern.charAt(j)) {
24           k--; j--;
25         }
26         else {
27           break;
28         }
29       }
30
31       if (j < 0)
32         return i = pattern.length() + 1; // A match found
33
34       int u = findLastIndex(text.charAt(k), j - 1, pattern);
35       if (u >= 0) { // text[k] is in the remaining part of the pattern
36         i = k + pattern.length() - 1 - u;
37       }
38       else { // text[k] is in the remaining part of the pattern
39         i = k + pattern.length();
40       }
41     }
42
43     return -1;
44   }
45
46   // Return the index of the last element in pattern[0 .. j]
47   // that matches ch. -1 otherwise.
48   private static int findLastIndex(char ch, int j, String pattern) {
49     for (int  k = j; k >= 0; k--) {
50       if (ch == pattern.charAt(k)) {
51         return k;
52       }
53     }
54
55     return - 1;
56   }
57 }
```

The `match(text, pattern)` method (lines 17–44) tests whether `pattern` matches a substring in `text`. `i` indicates the last index of a substring. It starts with `i = pattern.length() - 1` (line 21) and compares `text[i]` with `pattern[j]`, `text[i-1]` with `pattern[j-1]`, and so on backward (lines 22–29). If `j < 0`, a match is found (lines 31–32). Otherwise, find the index of the last matching element for `text[k]` in `pattern[0 .. j-1]` using the `findLastIndex` method (line 34). If the `index` is >= 0, set `k + m - 1 - index` (line 36), where m is `pattern.length()`. Otherwise, set `k + m` to `i` (line 39).
In the worst case, the Boyer-Moore algorithm takes `O(nm)` time. An example that achieves the worst case is shown in Figure 22.14.

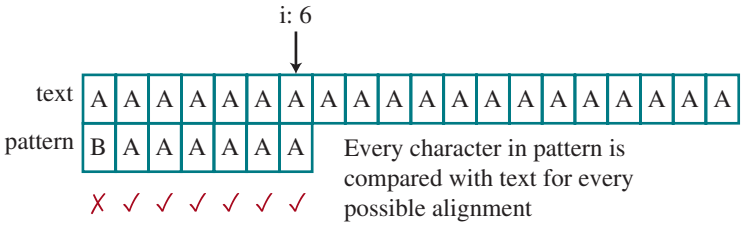


FIGURE 22.14 The text is all As and the pattern is BAAAAAA. Every character in the pattern is compared with a character in the text for each possible alignment.

However, on the average time, the Boyer-Moore algorithm is efficient because the algorithm is often able to skip a large portion of text. There are several variations of the Boyer-Moore algorithm. We presented a simplified version in this section.

22.11.2 The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is efficient. It achieves `O(m + n)` in the worst case. It is optimal because every character in the text and in the pattern must be checked at least once in the worst case. In the brute force or the Boyer-Moore algorithm, once a mismatch is found, the algorithm restarts to search for the next possible match by shifting the pattern one position to the right for the brute force algorithm and possibly multiple positions to the right for the Boyer-Moore algorithm. In doing so, the successful match of characters prior to the mismatch is ignored. The KMP algorithm takes consideration of the successful matches to find the maximum number of positions to shift in the pattern before continuing next search.

To find the maximum number of positions to shift in the pattern, we first define a failure function `fail(k)` as the length of the longest prefix of pattern that is a suffix of `pattern[0 .. k]`. The failure function can be precomputed for a given pattern. The failure function is actually an array with m elements. Suppose the pattern is `ABCABCDABC`. The failure functions for this pattern are shown in Figure 22.15:

k	0	1	2	3	4	5	6	7	8	9
pattern	A	B	C	A	B	C	D	A	B	C
fail	0	0	0	1	2	3	0	1	2	3

FIGURE 22.15 Failure function `fail[k]` is the length of the longest prefix that matches the suffix in `pattern[0..k]`.

For example, `fail[5]` is `3`, because `ABC` is the longest prefix that is suffix for `ABCABC`. `fail(7)` is 1, because `A` is the longest prefix that is suffix for `ABCABCDA`. When comparing

the text and the pattern, once a mismatch is found at index `k` in the pattern, you can shift the pattern to align the pattern at index `fail[k-1]` with the text, as shown in Figure 22.16.

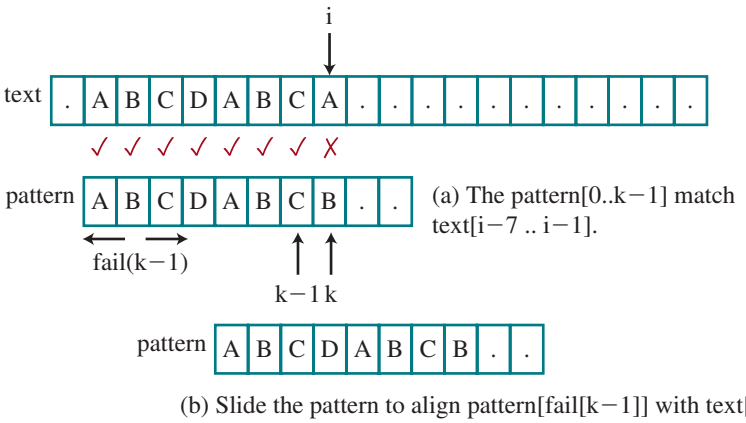


FIGURE 22.16 Upon a mismatch at `text[i]`, the pattern is shifted right to align the first `fail[k–1]` elements in the prefix with `text[i–1]`.

The KMP algorithm can be described as follows:

Step 1: First we precompute the failure functions. Now start with `i = 0` and `k = 0`.

Step 2: Compare `text[i]` with `pattern[k]`. Consider two cases:

- Case 1 `text[i]` is equal to `pattern[k]`: if `k` is `m-1`, a matching is found and return `i - m + 1`. Otherwise, increase `i` and `k` by `1`.
- Case 2 `text[i]` is not equal to `pattern[k]`: if `k > 0`, shift the longest prefix that matches the suffix in `pattern[k – 1]` so that the last character in the prefix is aligned with `text[i – 1]` by setting `k = fail[k – 1]`, else increase `i` by `1`.

Step 3: If `i < n`, repeat Step 2.

Now let us turn the attention to computing the failure functions. This can be done by comparing pattern with itself as follows:

Step 1: The failure function is an array with `m` elements. Initially, set all the elements to `0`. We start with `i = 1` and `k = 0`.

Step 2: Compare `pattern[i]` with `pattern[k]`. Consider two cases:

- Case 1 `pattern[i] == pattern[k]`: `fail[i] = k + 1`. Increase `i` and `k` by `1`.
- Case 2 `pattern[i] != pattern[k]`: if `k > 0`, set `k = fail[k – 1]`, else increase `i` by `1`.

Step 3: If `i < m`, repeat Step 2. Note that `k` indicates the length of the longest prefix in `pattern[0..i–1]` if `pattern[i] == pattern[k]`.

This is a sophisticated algorithm. The following animations are effective to help you understand the algorithms: https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMP.html and https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html

The following animation shows how to obtain the failure functions.

Listing 22.16 gives an implementation of the KMP algorithm.

LISTING 12.16 StringMatchKMP.java

```
1  public class StringMatchKMP {
2    public static void main(String[] args) {
3      java.util.Scanner input = new java.util.Scanner(System.in);
4      System.out.print("Enter a string text: ");
5      String text = input.nextLine();
6      System.out.print("Enter a string pattern: ");
7      String pattern = input.nextLine();
8
9      int index = match(text, pattern);
10     if (index >= 0)
11       System.out.println("matched at index "+ index);
12     else
13       System.out.println("unmatched");
14   }
15
16   // Return the index of the first match. -1 otherwise.
17   public static int match(String text, String pattern) {
18     int[] fail = getFailure(pattern);
19     int i = 0; // Index on text
20     int k = 0; // Index on pattern
21     while (i < text.length()) {
22       if (text.charAt(i) == pattern.charAt(k)) {
23         if (k == pattern.length() - 1) {
24           return i- pattern.length() + 1; // pattern matched
25         }
26         i++; // Compare the next pair of characters
27         k++;
28       }
29       else {
30         if (k > 0) {
31           k = fail[k -1]; // Matching prefix position
32         }
33         else {
34           i++; // No prefix
35         }
36       }
37     }
38
39     return -1;
40   }
41
42   // Compute failure function
43   private static int[] getFailure(String pattern) {
44     int[] fail = new int [pattern.length()];
45     int i = 1;
46     int k = 0;
47     while (i < pattern.length()) {
48       if (pattern.charAt(i) == pattern.charAt(k)) {
49         fail[i] = k + 1;
50         i++;
51         k++;
52       }
53       else if (k > 0) {
54         k = fail[k -1];
55       }
```

```
56          else {
57            i++;
58          }
59        }
60
61        return fail;
62      }
63    }
```

The `match(text, pattern)` method (lines 17–40) tests whether `pattern` matches a substring in `text`. `i` indicates the current position in the text, which always moves forward. `k` indicates the current position in the pattern. If `text[i] == pattern[k]` (line 22), both `i` and `k` is incremented by `1` (lines 26–27). Otherwise, if `k > 0`, set `fail(k – 1)` to `k` so to slide the pattern to align `pattern[k]` with `text[i]` (line 31), else increase `i` by `1` (line 34).

The `getFailure(pattern)` method (lines 43–62) compares `pattern` with `pattern` to obtain, `fail[k]`, the length of the maximum prefix that is the suffix in `pattern[0..k]`. It initializes the array `fail` to zeros (line 44) and set `i` and `k` to `1` and `0`, respectively (lines 45–46). `i` indicates the current position in the first pattern, which always moves forward. `k` indicates the current maximum length of a possible prefix that is also a suffix in `pattern[0..i]`. If `pattern[i] == pattern[k]`, set `fail[i]` to `k + 1` (line 49) and increase both `i` and `k` by `1` (lines 50–51). Otherwise, if `k > 0`, set `k` to `fail[k – 1]` to slide the second pattern to align `pattern[i]` in the first pattern with `pattern[k]` in the second pattern (line 54), else increase `i` by `1` (line 57).

To analyze the running time, consider three cases:

Case 1: `text[i]` is equal to `pattern[k]`. `i` is moved forward one position.
Case 2: `text[i]` is not equal to `pattern[k]` and `k` is `0`. `i` is moved forward one position.
Case 3: `text[i]` is not equal to `pattern[k]` and `k > 0`. The pattern is moved at least one position forward.

In any case, either `i` is moved forward one position on the text or the pattern is shifted at least one position to the right. Therefore, the number of iterations in the while loop for the match method is at most `2n`. Similarly, the number of the iterations in the `getFailure` method is at most `2m`. Therefore the running time of the KMP algorithm is `O(n + m)`.

Key Terms

average-case analysis 862
backtracking approach 886
best-case input 862
Big *O* notation 862
brute force 873
constant time 863
convex hull 889
divide-and-conquer approach 884
dynamic programming approach 872
exponential time 869
growth rate 862
logarithmic time 868
quadratic time 865
space complexity 863
time complexity 863
worst-case input 862

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, which enables you to 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 one that results in the longest execution time is called the *worst-case* input. Best- and worst-case analyses 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.

3. 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.

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

5. Linear search takes $O(n)$ time. An algorithm with the $O(n)$ time complexity is called a *linear algorithm* and it exhibits a linear growth rate. Binary search takes $O(\log n)$ time. An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm* and it exhibits a logarithmic growth rate.

6. The worst-time complexity for selection sort is $O(n^2)$. An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm* and it exhibits a quadratic growth rate.

7. The time complexity for the Tower of Hanoi problem is $O(2^n)$. An algorithm with the $O(2^n)$ time complexity is called an *exponential algorithm*, and it exhibits an exponential growth rate.

8. A Fibonacci number at a given index can be found in $O(n)$ time using dynamic programming approach.

9. Dynamic programming is the process of solving subproblems, then combining the solutions of the subproblems to obtain an overall solution. 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.

10. Euclid's GCD algorithm takes $O(\log n)$ time.

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

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

13. The divide-and-conquer approach divides the problem into subproblems, solves the subproblems, and 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.

14. The Eight Queens problem can be solved using backtracking.

15. The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines the candidate cannot possibly be a valid solution, then looks for a new candidate.

16. 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.

17. The brute force and Boyer-Moore string matching algorithms take O(nm) time and the KMP string matching algorithm takes O(n + m) time.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

***22.1** (*Natural numbers and their squares*) Knowing that $(x + 1)^2 = x^2 + 2x + 1$, and that multiplication is more time consuming than addition, write an efficient program that displays the first ten natural numbers and their squares. Knowing that $(x + 1)^3 = x^3 + 3x^2 + 3x + 1$, the same process can be followed to display the cubes of these numbers.

Here is a sample run:

```
0^2 = 0
1^2 = 1
2^2 = 4
...
8^2 = 64
9^2 = 81
10^2 = 100
```

****22.2** (*Efficient polynomial calculation*) For a polynomial $f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$, write an efficient program that prompts the user to enter the degree n, the coefficients a_0 to a_n, and the value for x, and computes and displays $f(x)$. Here is a sample run:

```
Enter n: 3 Enter
Enter x: 2 Enter
Enter a0: 1 Enter
Enter a1: 2 Enter
Enter a2: 3 Enter
Enter a3: 4 Enter
f (2.000000) = 49.000000
```

***22.3** (*Longest Palindrome*) Given a string, write an `O(n^2)` dynamic programming solution that returns the longest palindrome you can make from the string by deleting zero or more characters. For example, if the user enters "RACEF1CARNEW", the program returns 7 as RACECAR is a palindrome of length 7 after deleting "F1" and "NEW".

```
Enter your word: MADAM Enter
Longest palindrome: 5
Enter your word: RACEF1CARNEW Enter
Longest palindrome: 7
```

***22.4** (*Computing exponentials*) Write a method `public static double` exp(`double` x, `int` n) that computes *exp(x, n)*, which is an approximation of the exponential of *x* to the order *n*.

$$exp(x, n) = x^0/0! + x^1/1! + x^2/2! + \text{c} + x^n/n!$$

Analyze the time complexity of your method. Here is a sample run:

```
exp(1.0, 0) = 1.0
exp(1.0, 2) = 2.5
exp(1.0, 4) = 2.708333333333333
exp(1.0, 6) = 2.7180555555555554
exp(1.0, 8) = 2.71827876984127
exp(1.0, 10) = 2.7182818011463845
```

***22.5** (*Fill and sort array*) Write a program that randomly fills an array of integers and then sorts it. The array size is entered by the user and the values are chosen at random in {−1, 0, 1}. Analyze the time complexity of your program. Here is a sample run:

```
Enter the size: 10 Enter
 1  0 1 -1 0 1 -1 0 0 0
-1 -1 0  0 0 0  0 1 1 1
```

***22.6** (*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 46 to index 50 using the algorithms in Listings 22.3 and 22.4. Your program should print a table like this:

	46	47	48	49	50
Listing 22.3 GCD					
Listing 22.4 GCDEuclid					

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

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

**22.7 (*Closest pair of points*) Section 22.8 introduced an algorithm for finding the closest pair of points using a divide-and-conquer approach. Implement the algorithm to meet the following requirements:

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

```
/** Return the distance of the closest pair of points */
public static Pair getClosestPair(double[][] points)

/** Return the distance of the closest pair of points */
public static Pair getClosestPair(Point2D[] points)

/** Return the distance of the closest pair of points
  * in pointsOrderedOnX[low..high]. This is a recursive
  * method. pointsOrderedOnX and pointsOrderedOnY are
  * not changed in the subsequent recursive calls.
  */
public static Pair distance(Point2D[] pointsOrderedOnX,
    int low, int high, Point2D[] pointsOrderedOnY)

/** Compute the distance between two points p1 and p2 */
public static double distance(Point2D p1, Point2D p2)

/** Compute the distance between points (x1, y1) and (x2, y2) */
public static double distance(double x1, double y1,
    double x2, double y2)
```

**22.8 (*All prime numbers up to* `10,000,000,000`) Write a program that finds all prime numbers up to `10,000,000,000`. There are approximately `455,052,511` such prime numbers. Your program should meet the following requirements:

- Your program should store the prime numbers in a binary data file, named **PrimeNumbers.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 `10000`. If no number in the array is a divisor for the new number, continue to read the next `10000` 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 start over from scratch.

**22.9 (*Geometry: gift-wrapping algorithm for finding a convex hull*) Section 22.10.1 introduced the gift-wrapping algorithm for finding a convex hull for a set of points. Implement the algorithm using the following method:

```
/** Return the points that form a convex hull */
public static ArrayList<Point2D> getConvexHull(double[][] s)
```

Point2D was introduced in Section 9.6.3.

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. Note that when you debug the code, you will discover that the algorithm overlooked two cases (1) when `t1 = t0` and (2) when there is a point that is on the same line from `t0` to `t1`. When either case happens, replace `t1` by point `p` if the distance from `t0` to `p` is greater than the distance from `t0` to `t1`. Here is a sample run:

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

22.10 (*Number of prime numbers*) Programming Exercise 22.8 stores the prime numbers in a file named **PrimeNumbers.dat**. Write a program that finds the number of prime numbers that are less than or equal to `12`, `120`, `1,200`, `12,000`, `100,000`, `1,200,000`, `12,000,000`, `120,000,000`, `1,200,000,000`, and `12,000,000,000`. Your program should read the data from **PrimeNumbers.dat**.

****22.11** (*Geometry: Graham's algorithm for finding a convex hull*) Section 22.10.2 introduced Graham's algorithm for finding a convex hull for a set of points. Assume Java's coordinate system is used for the points. Implement the algorithm using the following method:

```
/** Return the points that form a convex hull */
public static ArrayList<MyPoint> getConvexHull(double[][] s)
```

`MyPoint` is a static inner class defined as follows:

```
private static class MyPoint implements Comparable<MyPoint> {
  double x, y;

  MyPoint rightMostLowestPoint;

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

  public void setRightMostLowestPoint(MyPoint p) {
    rightMostLowestPoint = p;
  }

  @Override
  public int compareTo(MyPoint o) {
    // Implement it to compare this point with point o
    // angularly along the x-axis with rightMostLowestPoint
    // as the center, as shown in Figure 22.10b. By implementing
    // the Comparable interface, you can use the Array.sort
    // method to sort the points to simplify coding.
  }
}
```

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:

```
How many points are in the set? 6 Enter
Enter six 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
     (2.5, 2.0) (6.0, 2.4) (5.5, 9.0) (1.5, 34.5) (1.0, 2.4)
```

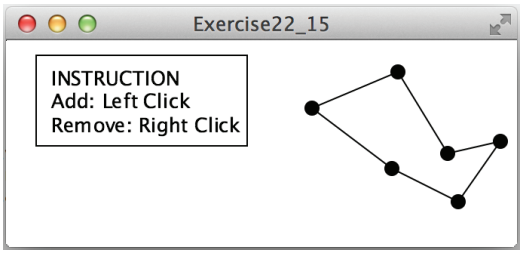
22.12 (*Last 100 prime numbers*) Programming Exercise 22.8 stores the prime numbers in a file named **PrimeNumbers.dat.** Write an efficient program that reads the last **100** numbers in the file. (*Hint*: Don't read all numbers from the file. Skip all numbers before the last 100 numbers in the file.)

22.13 (*Geometry: convex hull animation*) Programming Exercise 22.11 finds a convex hull for a set of points entered from the console. Write a program that enables the user to add or remove points by clicking the left or right mouse button and displays a convex hull, as shown in Figure 22.8c.

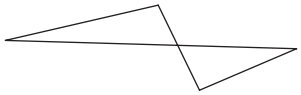
22.14 (*Execution time for prime numbers*) Write a program that obtains the execution time for finding all the prime numbers less than 9,000,000, 11,000,000, 13,000,000, 15,000,000, 17,000,000, and 19,000,000 using the algorithms in Listings 22.5–22.7. Your program should print a table like this:

	9000000	11000000	13000000	15000000	17000000	19000000
Listing 22.5						
Listing 22.6						
Listing 22.7						

22.15 (*Geometry: noncrossed polygon*) Write a program that enables the user to add or remove points by clicking the left or right mouse button and displays a noncrossed polygon that links all the points, as shown in Figure 22.17a. A polygon is crossed if two or more sides intersect, as shown in Figure 22.17b. Use the following algorithm to construct a polygon from a set of points:



(a)



(b) Crossed polygon

FIGURE 22.17 (a) Programming Exercise 22.15 displays a noncrossed polygon for a set of points. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) Two or more sides intersect in a crossed polygon.

Step 1: Given a set of points S, select the rightmost lowest point p_0 in the set 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 closer to p_0 is considered greater. The points in S are now sorted as $p_0, p_1, p_2, \ldots, p_{n-1}$.

Step 3: The sorted points form a noncrossed polygon.

**22.16 (*Linear search animation*) Write a program that animates the linear search algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 22.18. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm and repaints the histogram with a bar indicating the search position. This button also freezes the text field to prevent its value from being changed. When the algorithm is finished, display the status in the label at the top of the border pane to inform the user. Clicking the *Reset* button creates a new random array for a new start. This button also makes the text field editable.

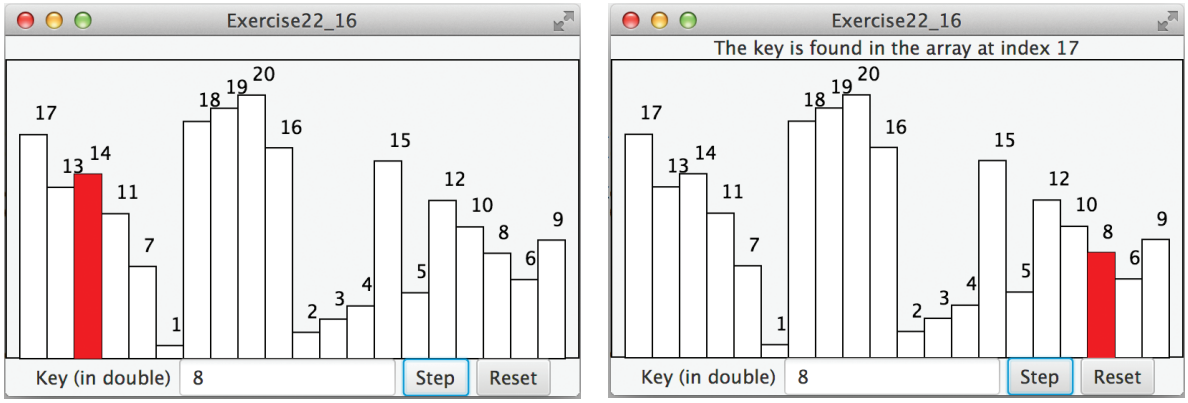


FIGURE 22.18 The program animates a linear search. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**22.17 (*Closest-pair animation*) Write a program that enables the user to add/remove points by clicking the left/right mouse button and displays a line that connects the pair of nearest points, as shown in Figure 22.4.

**22.18 (*Binary search animation*) Write a program that animates the binary search algorithm. Create an array with numbers from 1 to 20 in this order. The array elements are displayed in a histogram, as shown in Figure 22.19. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm. Use a light-gray color to paint the bars for the numbers in the current search range, and use a black color to paint the bar indicating the middle number in the search range. The *Step* button also freezes the text field to prevent its value from being changed. When the algorithm is finished, display the status in a label at the top of a border pane. Clicking the *Reset* button enables a new search to start. This button also makes the text field editable.

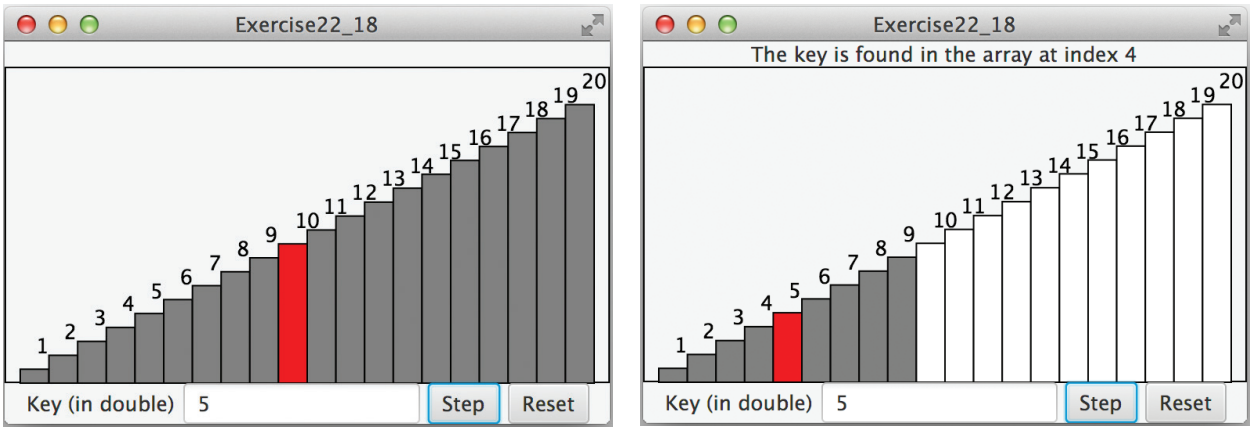
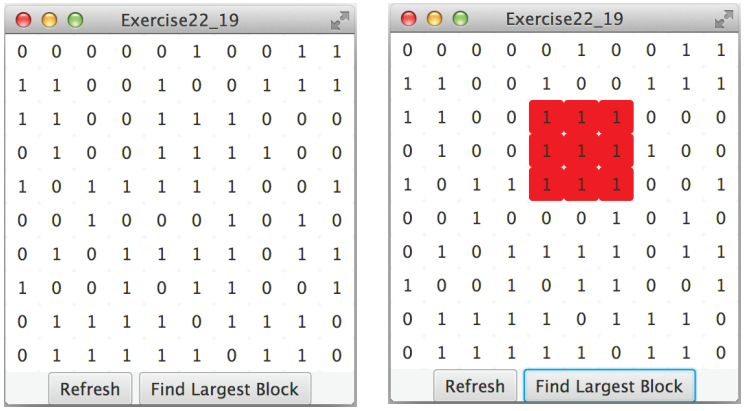


FIGURE 22.19 The program animates a binary search. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.19** (*Largest block*) The problem for finding a largest block is described in Programming Exercise 8.35. Design a dynamic programming algorithm for solving this problem in $O(n^2)$ time. Write a test program that displays a 10-by-10 square matrix, as shown in Figure 22.20a. Each element in the matrix is 0 or 1, randomly generated with a click of the *Refresh* button. Display each number centered in a text field. *Use a text field for each entry. Allow the user to change the entry value.* Click the *Find Largest Block* button to find a largest square submatrix that consists of 1s. Highlight the numbers in the block, as shown in Figure 22.20b.



(a) (b)

FIGURE 22.20 The program finds the largest block of 1s. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

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

*****22.21** (*Game: Sudoku*) The complete solution for the Sudoku problem is given in Supplement VI.C. Write a program that lets the user enter the input from the text fields, as shown in Figure 22.21a. Clicking the *Solve* button displays the result, as shown in Figures 22.21b and c.

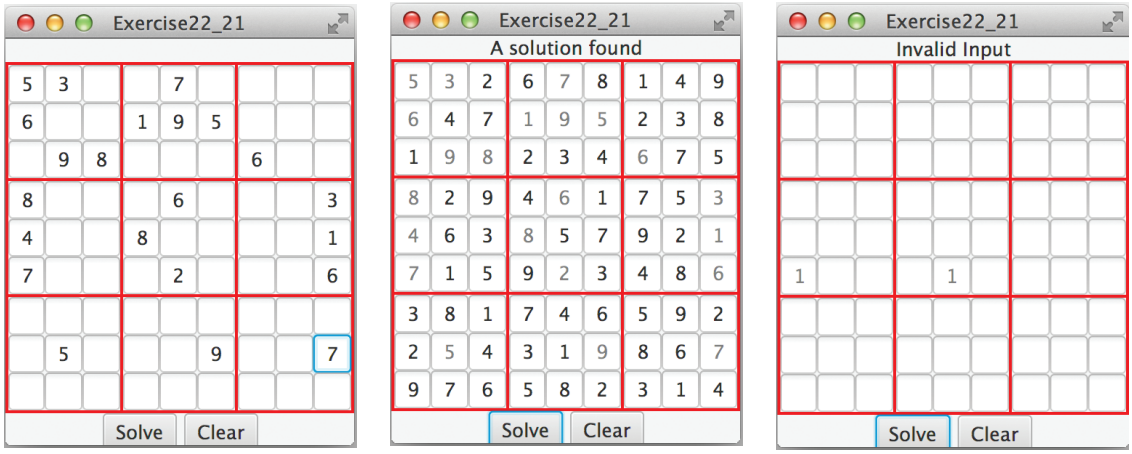


FIGURE 22.21 The program solves the Sudoku problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****22.22** (*Game: recursive Sudoku*) Write a recursive solution for the Sudoku problem.

*****22.23** (*Game: multiple Eight Queens solution*) Write a program to display all possible solutions for the Eight Queens puzzle in a scroll pane, as shown in Figure 22.22. For each solution, put a label to denote the solution number. (*Hint*: Place all solution panes into an `HBox` and place this one pane into a `ScrollPane`. If you run into a `StackOverflowError`, run the program using **java –Xss200m Exercise22_23** from the command window.)

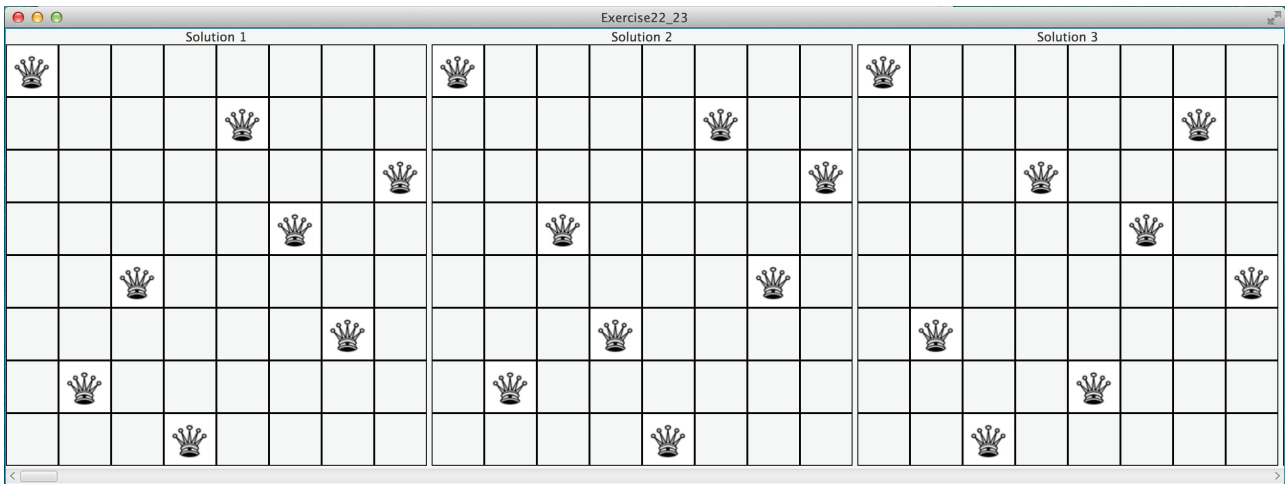
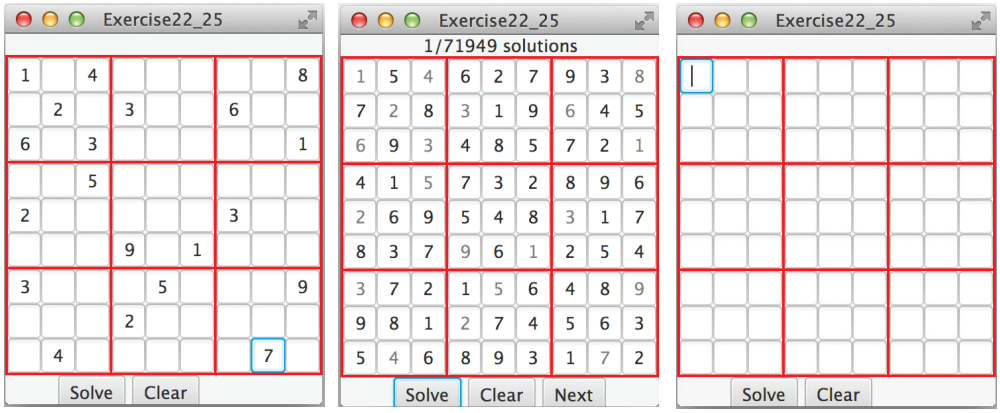


FIGURE 22.22 All solutions are placed in a scroll pane. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.24** (*Find the smallest number*) Write a method that uses the divide-and-conquer approach to find the smallest number in a list.

*****22.25** (*Game: Sudoku*) Revise Programming Exercise 22.21 to display all solutions for the Sudoku game, as shown in Figure 22.23a. When you click the *Solve* button, the program stores all solutions in an `ArrayList`. Each element in the list is a two-dimensional 9-by-9 grid. If the program has multiple solutions, the

Next button appears as shown in Figure 22.23b. You can click the *Next* button to display the next solution and also add a label to show the solution count. When the *Clear* button is clicked, the cells are cleared and the *Next* button is hidden as shown in Figure 22.23c.



(a) (b) (c)

FIGURE 22.23 The program can display multiple Sudoku solutions. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.26** (*Bin packing with smallest object first*) The bin packing problem is to pack the objects of various weights into containers. Assume each container can hold a maximum of 10 pounds. The program uses an algorithm that places an object with the *smallest weight* into the first bin in which it would fit. Your program should prompt the user to enter the total number of objects and the weight of each object. The program displays the total number of containers needed to pack the objects, and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8 Enter
Container 1 contains objects with weight 2 3 5
Container 2 contains objects with weight 5
Container 3 contains objects with weight 7
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

****22.27** (*Optimal bin packing*) Rewrite the preceding program so that it finds an optimal solution that packs all objects using the smallest number of containers. Here is a sample run of the program:

```
Enter the number of objects: 6 Enter
Enter the weights of the objects: 7 5 2 3 5 8 Enter
Container 1 contains objects with weight 7 3
Container 2 contains objects with weight 5 5
Container 3 contains objects with weight 2 8
The optimal number of bins is 3
```

What is the time complexity of your program?