

第20章

Introduction to Java Programming, 8E

递 归

学习目标

- 描述什么是递归方法以及使用递归方法的好处 (20.1节)。
- 开发递归数学函数的递归方法 (20.2~20.3节)。
- 理解在调用栈中如何处理递归方法的调用 (20.2~20.3节)。
- 使用一个重载的辅助方法派生一个递归方法 (20.5节)。
- 使用递归解决选择排序 (20.5.1节)。
- 使用递归解决二分查找 (20.5.2节)。
- 使用递归获取一个目录的大小 (20.6节)。
- 使用递归解决汉诺塔问题 (20.7节)。
- 使用递归绘制分形 (20.8节)。
- 使用递归解决八皇后问题 (20.9节)。
- 了解递归和迭代之间的联系与区别 (20.10节)。
- 了解尾递归方法以及为什么需要它 (20.11节)。

20.1 引言

假设希望找出某目录下所有包含某个特定单词的文件，该如何解决这个问题呢？有几种方式可以解决这个问题。一个直观且有效的解决方法是使用递归在子目录下递归地搜索所有的文件。

经典的八皇后难题就是将八个皇后放在棋盘上，而没有任何两个皇后可以互相攻击（即不会出现两个皇后在同一行、同一列或者同一条对角线上的情况），如图20-1所示。该如何编写程序解决这个问题呢？一个好的办法就是使用递归。

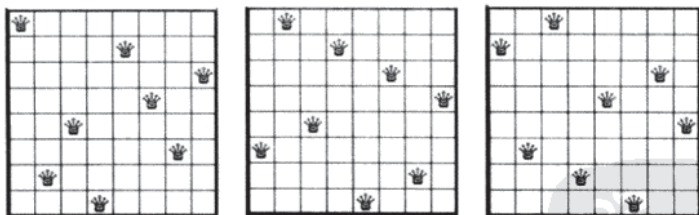


图20-1 可以使用递归解决八皇后问题

使用递归就是使用递归方法 (recursive method) 编程，递归方法就是直接或间接调用自身的方法。递归是一个很有用的程序设计技术。在某些情况下，对于用其他方法很难解决的问题，使用递归就能给出一个自然、直接的简单解法。本章介绍递归程序设计的概念和技术，并用例子来演示如何进行“递归思考”。

20.2 问题：计算阶乘

许多数学函数都是使用递归来定义的。我们从一个简单的例子开始。数字 n 的阶乘可以递归地定义如下：

```
0! = 1;
n! = n × (n - 1) × ... × 2 × 1 = n × (n - 1)!; n > 0
```

对给定的 n 如何求 $n!$ 呢？由于已经知道 $0!=1$ ，而 $1!=1 \times 0!$ ，因此很容易求得 $1!$ 。假设已知知道 $(n-1)!$ ，使用 $n!=n \times (n-1)!$ 就可以立即得到 $n!$ 。这样，计算 $n!$ 的问题就简化为计算 $(n-1)!$ 。当计算 $(n-1)!$ 时，可以递归地应用这个思路直到 n 递减为0。

假定计算 $n!$ 的方法是`factorial(n)`。如果用 $n=0$ 调用这个方法，立即就能返回它的结果。这个方法知道如何处理最简单的情况，这种最简单的情况称为基础情况（base case）或终止条件（stopping condition）。如果用 $n>0$ 调用这个方法，就应该把这个问题简化为计算 $n-1$ 的阶乘的子问题。子问题在本质上和原始问题是一样的，但是它比原始问题更简单也更小。因为子问题和原始问题具有相同的性质，所以可以用不同的参数调用这个方法，这称作递归调用（recursive call）。

计算`factorial(n)`的递归算法可以简单地描述如下：

```
if (n == 0)
    return 1;
else
    return n * factorial(n - 1);
```

一个递归调用可以导致更多的递归调用，因为这个方法继续把每个子问题分解成新的子问题。要终止一个递归方法，问题最后必须达到一个终止条件。当问题达到这个终止条件时，就将结果返回给调用者。然后调用者进行计算并将结果返回给它自己的调用者。这个过程持续进行，直到结果传回原始的调用者为止。现在，原始问题就可以由`factorial(n-1)`的结果乘以 n 得到。

程序清单20-1给出一个完整的程序，提示用户输入一个非负整数，然后显示这个数的阶乘。

程序清单20-1 `ComputeFactorial.java`

```
1 import java.util.Scanner;
2
3 public class ComputeFactorial {
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 a nonnegative integer: ");
9         int n = input.nextInt();
10
11         // Display factorial
12         System.out.println("Factorial of " + n + " is " + factorial(n));
13     }
14
15     /** Return the factorial for a specified number */
16     public static long factorial(int n) {
17         if (n == 0) // Base case
18             return 1;
19         else
20             return n * factorial(n - 1); // Recursive call
21     }
22 }
```

Enter a nonnegative integer: 4 Enter
Factorial of 4 is 24

Enter a nonnegative integer: 10 Enter
Factorial of 10 is 3628800

本质上讲，`factorial`方法（第16~21行）是把阶乘在数学上的递归定义直接转换为Java代码。因为对`factorial`的调用是调用它自己，所以这个调用是递归的。传递到`factorial`的参数一直递减，直

到达它的基础情况0。

图20-2描述了从n = 4开始执行的递归调用。递归调用对堆栈空间的使用如图20-3所示。

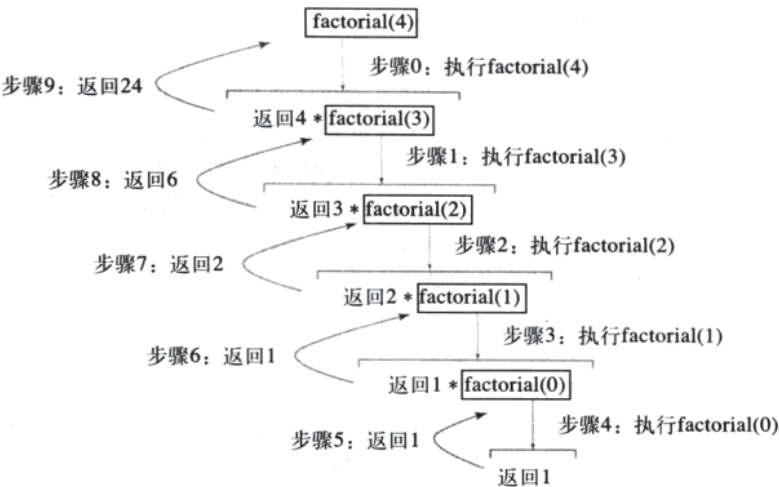


图20-2 调用factorial(4)会引起对factorial的递归调用

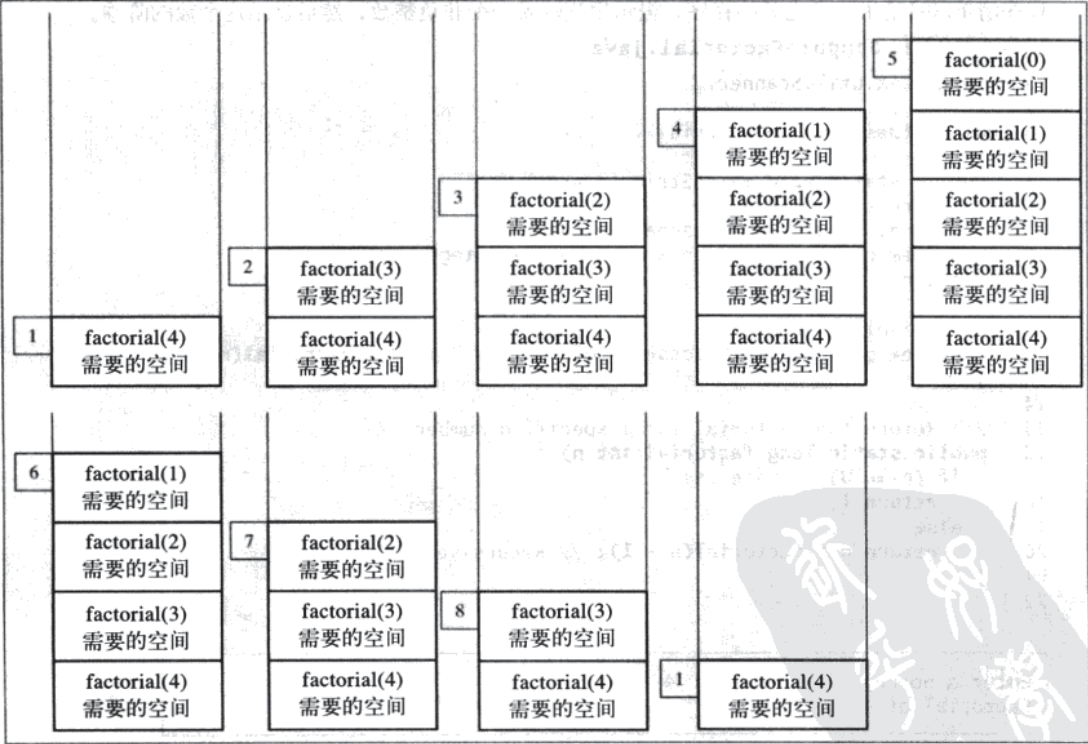


图20-3 执行factorial(4)时，factorial方法被递归调用，导致内存空间动态变化

警告 如果递归不能使问题简化并最终收敛到基础情况，就有可能出现无限递归。例如，假设将factorial方法错误地写成如下所示：

```
public static long factorial(int n) {
    return n * factorial(n - 1);
}
```

那么这个方法会无限地运行下去，并且会导致一个StackOverflowError。

教学注意 使用循环来实现factorial方法是比较简单且更加高效的。然而，这里使用的递归factorial方法是演示递归概念的一个很好的例子。在本章后续内容中，还将给出一些继承递归以及不使用递归很难解决的问题。

20.3 问题：计算斐波那契数

前一节中的factorial方法可以很容易地不使用递归改写。但是，在某些情况下，用其他方法不容易解决的问题可以利用递归给出一个自然、直接、简单的解法。考虑众所周知的斐波那契（Fibonacci）数列问题，如下所示：

数列：	0	1	1	2	3	5	8	13	21	34	55	89	...
下标：	0	1	2	3	4	5	6	7	8	9	10	11	

斐波那契数列从0和1开始，之后的每个数都是序列中前两个数的和。序列可以递归定义为：

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

斐波那契数列是以中世纪数学家Leonardo Fibonacci的名字命名的，他为建立兔子繁殖数量的增长模型而构造出这个数列。这个数列可用于数值优化和其他很多领域。

对给定的index，怎样求fib(index)呢？因为已知fib(0)和fib(1)，所以很容易求得fib(2)。假设已知fib(index-2)和fib(index-1)，就可以立即得到fib(index)。这样，计算fib(index)的问题就简化为计算fib(index-2)和fib(index-1)的问题。以这种方式求解，就可以递归地运用这个思路直到index递减为0或1。

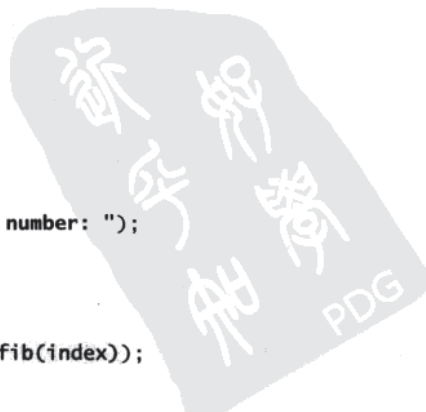
基础情况是index=0或index=1。若用index=0或index=1调用这个方法，它会立即返回结果。若用index>=2调用这个方法，则通过使用递归调用把问题分解成计算fib(index-2)和fib(index-1)的两个子问题。计算fib(index)的递归算法可以简单地描述如下：

```
if (index == 0)
    return 0;
else if (index == 1)
    return 1;
else
    return fib(index - 1) + fib(index - 2);
```

程序清单20-2给出一个完整的程序，提示用户输入一个下标，然后计算这个下标值相应的斐波那契数。

程序清单20-2 ComputeFibonacci.java

```
1 import java.util.Scanner;
2
3 public class ComputeFibonacci {
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 index) {
```




```

18     if (index == 0) // Base case
19         return 0;
20     else if (index == 1) // Base case
21         return 1;
22     else // Reduction and recursive calls
23         return fib(index - 1) + fib(index - 2);
24 }
25 }

```

Enter an index for the Fibonacci number: 1
Fibonacci number at index 1 is 1



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



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



程序并没有显示计算机在后台所做的大量工作。但是，图20-4显示出计算`fib(4)`所进行的连续递归调用。原始方法`fib(4)`产生两个递归调用`fib(3)`和`fib(2)`，然后返回`fib(3)+fib(2)`的值。但是，按怎样的顺序调用这些方法呢？在Java中，操作数是从左到右计算的，所以在完全计算完`fib(3)`之后才会调用`fib(2)`。图20-4中的标签表示方法调用的顺序。

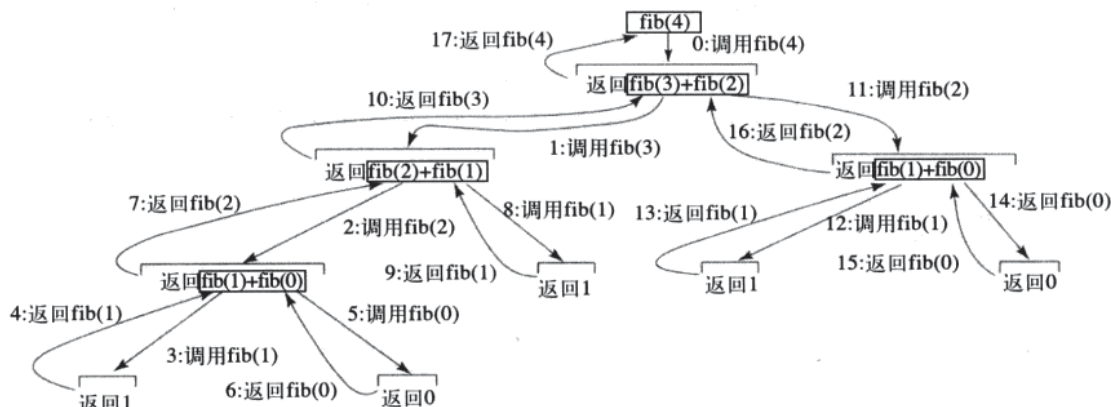


图20-4 调用`fib(4)`会引起对`fib`的递归调用

如图20-4所示，会出现很多重复的递归调用。例如，`fib(2)`调用了2次，`fib(1)`调用了3次，`fib(0)`也调用了2次。通常，计算`fib(index)`所需的递归调用次数大致是计算`fib(index-1)`所需次数的2倍。如果尝试更大的下标值，那么相应的调用次数会急剧增加。

除了大量的递归调用的次数，计算机还需要更多的时间和空间来运行递归的方法。

教学注意 `fib`方法的递归实现非常简单、直接，但是并不高效。参见练习题20.2中使用循环的高效方案。虽然递归的`fib`方法并不实用，但是它是一个演示如何编写递归方法的很好的例子。

20.4 使用递归解决问题

前几节给出了两个经典的递归例子。所有的递归方法都具有以下特点：

- 这些方法使用`if-else`或`switch`语句会导致不同的情况。

- 一个或多个基础情况（最简单的情况）用来停止递归。
- 每次递归调用都会简化原始问题，让它不断地接近基础情况，直到它变成这种基础情况为止。

通常，要使用递归解决问题，就要将这个问题分解为子问题。每个子问题几乎与原始问题是一样的，只是规模小一些。可以应用相同的方法来递归解决子问题。

考虑打印一条消息 n 次的简单问题。可以将这个问题分解为两个子问题：一个是打印消息一次，另一个是打印消息 $n-1$ 次。第二个问题与原始问题是一样的，只是规模小一些。这个问题的基础情况是 $n=0$ 。可以使用递归来解决这个问题，如下所示：

```
public static void nPrintln(String message, int times) {
    if (times >= 1) {
        System.out.println(message);
        nPrintln(message, times - 1);
    } // The base case is times == 0
}
```

需要注意的是，前面例子中的`fib`方法向其调用者返回一个数值，但是`nPrintln`方法的返回类型是`void`，并不向其调用者返回一个数值。

如果以递归的思路进行思考（think recursively），那么，本书前面章节中的许多问题都可以用递归来解决。考虑程序清单9-1中的回文问题。回想一下，如果一个字符串从左读和从右读是一样的，那么它就是一个回文串。例如，`mom`和`dad`都是回文串，但是`uncle`和`aunt`不是回文串。检查一个字符串是否是回文串的问题可以分解为两个子问题：

- 检查字符串中的第一个字符和最后一个字符是否相等。
- 忽略两端的字符之后检查子串的其余部分是否是回文。

第二个子问题与原始问题是一样的，但是规模小一些。基本状态有两个：1) 两端的字符不同；2) 字符串大小是0或1。在第一种情况下，字符串不是回文；而在第二种情况下，字符串是回文串。这个问题的递归方法可以在程序清单20-3中实现。

程序清单20-3 RecursivePalindromeUsingSubstring.java

```
1 public class RecursivePalindromeUsingSubstring {
2     public static boolean isPalindrome(String s) {
3         if (s.length() <= 1) // Base case
4             return true;
5         else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
6             return false;
7         else
8             return isPalindrome(s.substring(1, s.length() - 1));
9     }
10
11     public static void main(String[] args) {
12         System.out.println("Is moon a palindrome? "
13             + isPalindrome("moon"));
14         System.out.println("Is noon a palindrome? "
15             + isPalindrome("noon"));
16         System.out.println("Is a a palindrome? " + isPalindrome("a"));
17         System.out.println("Is aba a palindrome? " +
18             isPalindrome("aba"));
19         System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
20     }
21 }
```

```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```



第8行的substring方法创建了一个新字符串，它除了没有原始字符串中的第一个和最后一个字符，其余都是和原始字符串一样的。如果原始字符串中的两端字符相同，那么检查一个字符串是否是回文与检查子串是否是回文的方法是一样的。

20.5 递归的辅助方法

因为前面递归的isPalindrome方法要为每次递归调用创建一个新字符串，因此它不够高效。为避免创建新字符串，可以使用low和high下标来表明子串的范围。这两个下标必须传递给递归方法。由于原始方法是isPalindrome(String s)，因此，必须产生一个新方法isPalindrome(String s,int low,int high)来接受字符串的其余信息，如程序清单20-4所示。

程序清单20-4 RecursivePalindrome.java

```

1 public class RecursivePalindrome {
2     public static boolean isPalindrome(String s) {
3         return isPalindrome(s, 0, s.length() - 1);
4     }
5
6     public static boolean isPalindrome(String s, int low, int high) {
7         if (high <= low) // Base case
8             return true;
9         else if (s.charAt(low) != s.charAt(high)) // Base case
10            return false;
11        else
12            return isPalindrome(s, low + 1, high - 1);
13    }
14
15    public static void main(String[] args) {
16        System.out.println("Is moon a palindrome? "
17            + isPalindrome("moon"));
18        System.out.println("Is noon a palindrome? "
19            + isPalindrome("noon"));
20        System.out.println("Is a a palindrome? " + isPalindrome("a"));
21        System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
22        System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
23    }
24 }

```

程序中定义了两个重载的isPalindrome方法。第一个方法isPalindrome(String s)检查一个字符串是否是回文串，而第二个方法isPalindrome(String s,int low,int high)检查一个子串s(low..high)是否是回文串。第一个方法将low=0和high=s.length()-1的字符串s传递给第二个方法。第二个方法采用递归调用，检查不断缩减的子串是否是回文串。在递归程序设计中定义第二个方法来接收附加的参数是一个常用的设计技巧，这样的方法称为递归的辅助方法（recursive helper method）。

辅助方法在设计关于字符串和数组问题的递归方案上是非常有用的。下面几节将给出两个以上的例子。

20.5.1 选择排序

选择排序在6.10.1节中介绍过。回顾一下，选择排序法是先找到列表的最小数，首先将它放在列表中。然后，在剩余的数中找到最小数，再将它放到第一个数字的后面，这样的过程一直进行下去，直到列表中仅剩一个数为止。这个问题可以分解为两个子问题：

- 找出列表中的最小数，然后将它与第一个数进行交换。
- 忽略第一个数，对剩下的小一些的列表进行递归排序。

基础情况是该列表只包含一个数。程序清单20-5给出了递归的排序方法。

程序清单20-5 RecursiveSelectionSort.java

```

1 public class RecursiveSelectionSort {
2     public static void sort(double[] list) {
3         sort(list, 0, list.length - 1); // Sort the entire list

```

```

4  }
5
6  public static void sort(double[] list, int low, int high) {
7      if (low < high) {
8          // Find the smallest number and its index in list(low .. high)
9          int indexOfMin = low;
10         double min = list[low];
11         for (int i = low + 1; i <= high; i++) {
12             if (list[i] < min) {
13                 min = list[i];
14                 indexOfMin = i;
15             }
16         }
17
18         // Swap the smallest in list(low .. high) with list(low)
19         list[indexOfMin] = list[low];
20         list[low] = min;
21
22         // Sort the remaining list(low+1 .. high)
23         sort(list, low + 1, high);
24     }
25 }
26 }

```

程序中定义了两个重载的sort方法。第一个方法sort(double[] list)对数组list[0..list.length-1]进行排序，而第二个方法sort(double[] list, int low, int high)对数组list[low..high]进行排序。第二个方法采用递归调用，对不断缩小的子数组进行排序。

20.5.2 二分查找

二分查找在6.9.2节中介绍过。使用二分查找法的前提条件是数组元素必须已经排好序。二分查找法首先将关键字与数组的中间元素进行比较，考虑下面三种情况：

情况1：如果关键字比中间元素小，那么只需在前一半数组元素中进行递归查找。

情况2：如果关键字和中间元素相等，则匹配成功，查找结束。

情况3：如果关键字比中间元素大，那么只需在后一半数组元素中进行递归查找。

情况1和情况3都将查找范围降为一个更小的数列。当匹配成功时，情况2就是一个基本状态。另一个基本状态是查找完毕而没有一个成功的匹配。程序清单20-6使用递归给二分查找问题一个清晰、简单的解决方案。

程序清单20-6 递归二分查找法

```

1  public class RecursiveBinarySearch {
2      public static int recursiveBinarySearch(int[] list, int key) {
3          int low = 0;
4          int high = list.length - 1;
5          return recursiveBinarySearch(list, key, low, high);
6      }
7
8      public static int recursiveBinarySearch(int[] list, int key,
9          int low, int high) {
10         if (low > high) // The list has been exhausted without a match
11             return -low - 1;
12
13         int mid = (low + high) / 2;
14         if (key < list[mid])
15             return recursiveBinarySearch(list, key, low, mid - 1);
16         else if (key == list[mid])
17             return mid;
18         else
19             return recursiveBinarySearch(list, key, mid + 1, high);
20     }
21 }

```


第一个方法是在整个数列中查找关键字。第二个方法是在数列下标从low到high的数列中查找关键字。

第一个binarySearch方法是将low=0和high=list.length-1的初始数组传递给第二个binarySearch方法。第二个方法采用递归调用，在不断缩小的子数组中查找关键字。

20.6 问题：求出目录的大小

前面的例子可以不用递归很容易地解决。对于本节给出的这个问题，要是不使用递归是很难解决的。这里的问题是求出一个目录的大小。一个目录的大小是指该目录下所有文件大小之和。目录d可能会包含子目录。假设一个目录包含文件 f_1, f_2, \dots, f_m 以及子目录 d_1, d_2, \dots, d_n ，如图20-5所示。

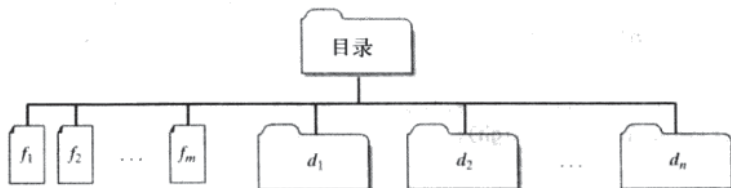


图20-5 一个目录包含多个文件和子目录

目录的大小可以如下递归地定义：

$$\text{size}(d) = \text{size}(f_1) + \text{size}(f_2) + \dots + \text{size}(f_m) + \text{size}(d_1) + \text{size}(d_2) + \dots + \text{size}(d_n)$$

9.6节介绍的File类可以用来表示一个文件或一个目录，并且获取文件和目录的属性。File类中的两个方法对这个问题是很有用的：

- length()方法返回一个文件的大小。
- listFiles()方法返回一个目录下的File对象构成的数组。

程序清单20-7给出一个程序，提示用户输入一个目录或一个文件，然后显示它的大小。


程序清单20-7 DirectorySize.java

```


1 import java.io.File;
2 import java.util.Scanner;
3
4 public class DirectorySize {
5     public static void main(String[] args) {
6         // Prompt the user to enter a directory or a file
7         System.out.print("Enter a directory or a file: ");
8         Scanner input = new Scanner(System.in);
9         String directory = input.nextLine();
10
11         // Display the size
12         System.out.println(getSize(new File(directory)) + " bytes");
13     }
14
15     public static long getSize(File file) {
16         long size = 0; // Store the total size of all files
17
18         if (file.isDirectory()) {
19             File[] files = file.listFiles(); // All files and subdirectories
20             for (int i = 0; i < files.length; i++) {
21                 size += getSize(files[i]); // Recursive call
22             }
23         }
24         else { // Base case
25             size += file.length();
26         }
27
28         return size;

```


```
29 }
30 }
```

```
Enter a directory or a file: c:\book 
48619631 bytes
```



```
Enter a directory or a file: c:\book\Welcome.java 
172 bytes
```



```
Enter a directory or a file: c:\book\NonExistentFile 
0 bytes
```



如果`file`对象表示一个目录（第18行），那么该目录下的每个子条目（文件或子目录）都被递归地调用来获取它的大小（第21行）。如果`file`对象表示一个文件（第24行），获取的就是该文件的大小（第25行）。

如果输入的是一个错误的目录或者不存在的目录，会发生什么情况呢？该程序将会发现它不是一个目录，并且调用`file.length()`（第25行），它会返回0。因此，在这种情况下，`getSize`方法将返回0。

提示 为了避免错误，测试基本状态是一个很好的尝试。例如，应该输入一个文件、一个空目录、一个不存在的目录以及一个不存在的文件来测试这个程序。

20.7 问题：汉诺塔

汉诺塔问题是一个经典的递归例子。这个问题用递归可以很容易地解决，但是，不使用递归则非常难解决。

这个问题是将指定个数而大小互不相同的盘子从一个塔上移到另一个塔上，移动要遵从下面的规则：

- 1) n 个盘子标记为1, 2, 3, ..., n ，而三个塔标记为A、B和C。
- 2) 任何时候盘子都不能放在比它小的盘子的上方。
- 3) 初始状态时，所有的盘子都放在塔A上。
- 4) 每次只能移动一个盘子，并且这个盘子必须在塔顶位置。

这个问题的目标是借助塔C把所有的盘子从塔A移到塔B。例如，如果有三个盘子，将所有的盘子从A移到B的步骤如图20-6所示。

注意 汉诺塔是一个经典的计算机科学问题。许多网站都有关于该问题的解法。其中一个很值得一看的网站是www.cut-the-knot.com/recurrence/hanoi.html。

在三个盘子的情况下，可以手动地找出解决方案。然而，当盘子数量较大时，即使是4个，这个问题还是非常复杂的。幸运的是，这个问题本身就具有递归性质，可以直接得到递归解法。

问题的基础情况是 $n=1$ 。若 $n==1$ ，就可以简单地把盘子从A移到B。当 $n>1$ 时，可以将原始问题拆成下面的三个子问题，然后依次解决。

- 1) 借助塔B将前 $n-1$ 个盘子从A移到C，如图20-7中的步骤1所示。
- 2) 将盘子 n 从A移到B，如图20-7中的步骤2所示。
- 3) 借助塔A将 $n-1$ 个盘子从C移到B，如图20-7中的步骤3所示。

下面的方法借助于辅助塔`auxTower`将 n 个盘子从原始塔`fromTower`移到目标塔`toTower`上：

```
void moveDisks(int n, char fromTower, char toTower, char auxTower)
```

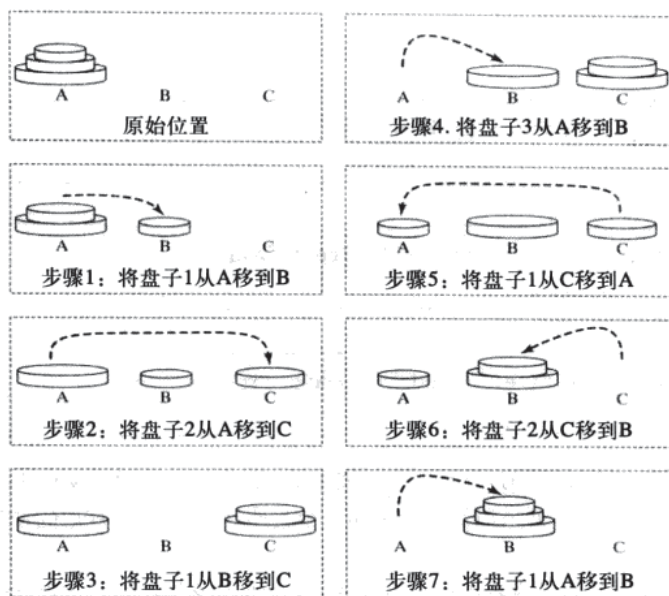


图20-6 汉诺塔问题的目的是在遵从规则的条件下把盘子从塔A移到塔B

这个方法的算法可以描述如下:

```

if (n == 1) // Stopping condition
    Move disk 1 from the fromTower to the toTower;
else {
    moveDisks(n - 1, fromTower, auxTower, toTower);
    Move disk n from the fromTower to the toTower;
    moveDisks(n - 1, auxTower, toTower, fromTower);
}

```

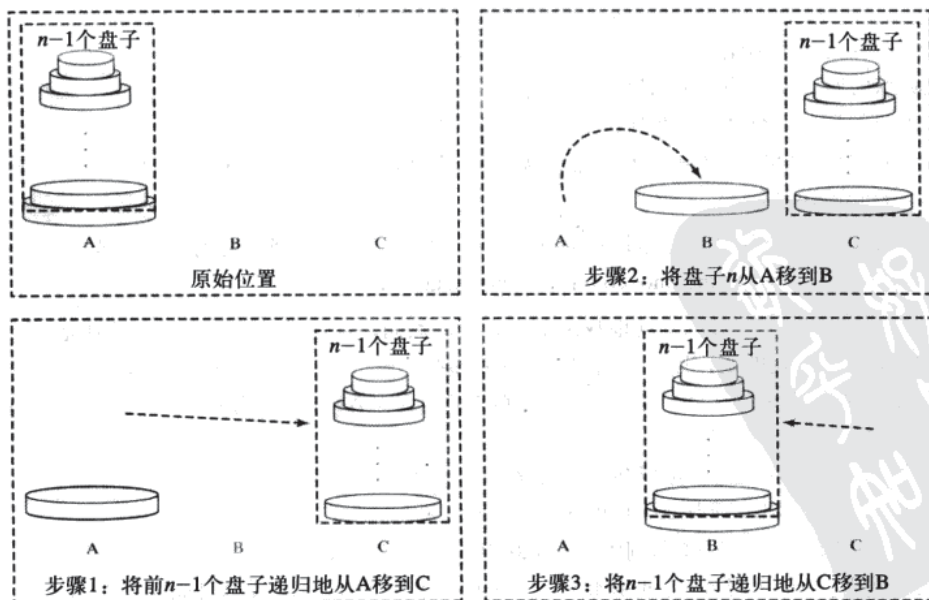


图20-7 汉诺塔问题可以分解成三个子问题

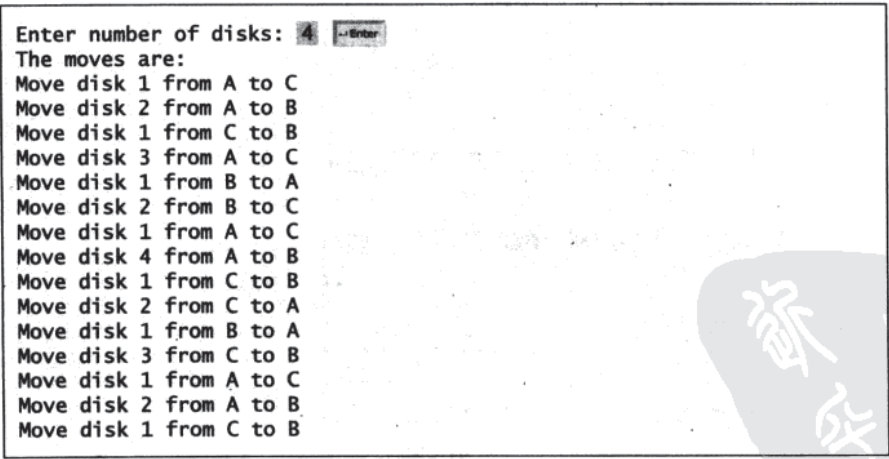
程序清单20-8给出一个程序，提示用户输入盘子个数，然后调用递归的方法moveDisks来显示移动盘子的解决方案。

程序清单20-8 TowerOfHanoi.java

```

1 import java.util.Scanner;
2
3 public class TowersOfHanoi {
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 number of disks: ");
9         int n = input.nextInt();
10
11         // Find the solution recursively
12         System.out.println("The moves are:");
13         moveDisks(n, 'A', 'B', 'C');
14     }
15
16     /** The method for finding the solution to move n disks
17         from fromTower to toTower with auxTower */
18     public static void moveDisks(int n, char fromTower,
19         char toTower, char auxTower) {
20         if (n == 1) // Stopping condition
21             System.out.println("Move disk " + n + " from " +
22                 fromTower + " to " + toTower);
23         else {
24             moveDisks(n - 1, fromTower, auxTower, toTower);
25             System.out.println("Move disk " + n + " from " +
26                 fromTower + " to " + toTower);
27             moveDisks(n - 1, auxTower, toTower, fromTower);
28         }
29     }
30 }

```



```

Enter number of disks: 4
The moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B

```



这个问题本质上是递归的。利用递归就能够找到一个自然、简单的解决方案。如果不使用递归，解决这个问题可能会很困难。

考虑跟踪 $n=3$ 的程序。连续的递归调用如图20-8所示。正如所见，编写这个程序比跟踪这个递归调用要容易些。系统使用栈来跟踪后台的调用。从某种程度上讲，递归提供了某种层次的抽象，这种抽象对用户隐藏迭代和其他细节。

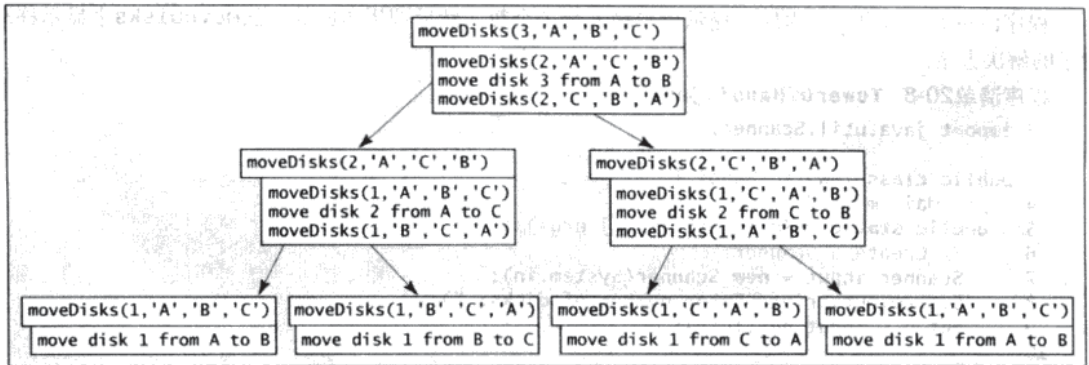


图20-8 调用moveDisks(3, 'A', 'B', 'C')会引起对moveDisks的递归调用

20.8 问题：分形

分形是一个几何图形，但是它不像三角形、圆形和矩形。分形可以分成几个部分，每部分都是整体的一个缩小的副本。分形有许多有趣的例子。本节介绍一个称为思瑞平斯基三角形 (Sierpinski triangle) 的简单分形，它是以一位著名的波兰数学家的名字来命名的。

思瑞平斯基三角形是如下创建的：

- 1) 从一个等边三角形开始，将它作为0阶（或0级）的思瑞平斯基分形，如图20-9a所示。
- 2) 将0阶三角形的各边中点连接起来产生1阶思瑞平斯基三角形（图20-9b）。
- 3) 保持中间的三角形不变，将另外三个三角形各边的中点连接起来产生2阶思瑞平斯基分形（图20-9c）。
- 4) 可以递归地重复同样的步骤产生3阶，4阶，…的思瑞平斯基三角形（图20-7d）。

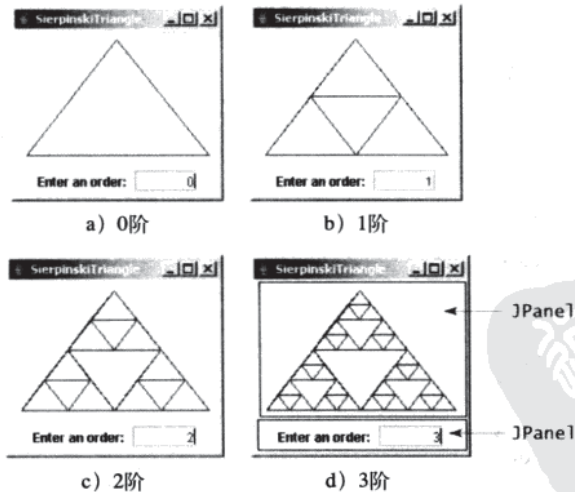


图20-9 思瑞平斯基三角形是一种递归三角形的图形

这个问题本质上是递归的。那么，该如何开发针对该问题的递归方案呢？考虑阶数为0的基础情况。这时，能够很容易地绘制出0阶思瑞平斯基三角形。如何绘制出1阶思瑞平斯基三角形呢？这个问题可以简化为绘制三个0阶思瑞平斯基三角形。如何绘制2阶思瑞平斯基三角形呢？这个问题可以简化为绘制三个1阶思瑞平斯基三角形。因此，绘制n阶思瑞平斯基三角形可以简化为绘制三个n-1阶思瑞平斯基三角形。

程序清单20-9给出显示任意阶的思瑞平斯基三角形的Java applet，如图20-9所示。用户可以在文本域

输入阶数，然后显示这个指定阶数的思瑞平斯基三角形。

程序清单20-9 SierpinskiTriangle.java

```

1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class SierpinskiTriangle extends JApplet {
6     private JTextField jtfOrder = new JTextField("0", 5); // Order
7     private SierpinskiTrianglePanel trianglePanel =
8         new SierpinskiTrianglePanel(); // To display the pattern
9
10    public SierpinskiTriangle() {
11        // Panel to hold label, text field, and a button
12        JPanel panel = new JPanel();
13        panel.add(new JLabel("Enter an order: "));
14        panel.add(jtfOrder);
15        jtfOrder.setHorizontalAlignment(SwingConstants.RIGHT);
16
17        // Add a Sierpinski triangle panel to the applet
18        add(trianglePanel);
19        add(panel, BorderLayout.SOUTH);
20
21        // Register a listener
22        jtfOrder.addActionListener(new ActionListener() {
23            public void actionPerformed(ActionEvent e) {
24                trianglePanel.setOrder(Integer.parseInt(jtfOrder.getText()));
25            }
26        });
27    }
28
29    static class SierpinskiTrianglePanel extends JPanel {
30        private int order = 0;
31
32        /** Set a new order */
33        public void setOrder(int order) {
34            this.order = order;
35            repaint();
36        }
37
38        protected void paintComponent(Graphics g) {
39            super.paintComponent(g);
40
41            // Select three points in proportion to the panel size
42            Point p1 = new Point(getWidth() / 2, 10);
43            Point p2 = new Point(10, getHeight() - 10);
44            Point p3 = new Point(getWidth() - 10, getHeight() - 10);
45
46            displayTriangles(g, order, p1, p2, p3);
47        }
48
49        private static void displayTriangles(Graphics g, int order,
50            Point p1, Point p2, Point p3) {
51            if (order >= 0) {
52                // Draw a triangle to connect three points
53                g.drawLine(p1.x, p1.y, p2.x, p2.y);
54                g.drawLine(p1.x, p1.y, p3.x, p3.y);
55                g.drawLine(p2.x, p2.y, p3.x, p3.y);
56
57                // Get the midpoint on each edge in the triangle
58                Point p12 = midpoint(p1, p2);
59                Point p23 = midpoint(p2, p3);
60                Point p31 = midpoint(p3, p1);
61
62                // Recursively display three triangles
63                displayTriangles(g, order - 1, p1, p12, p31);

```

```

64         displayTriangles(g, order - 1, p12, p2, p23);
65         displayTriangles(g, order - 1, p31, p23, p3);
66     }
67 }
68
69 private static Point midpoint(Point p1, Point p2) {
70     return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
71 }
72 }
73 }

```

初始三角形有三个与面板大小成比例的点集（第42~44行）。如果`order ≥ 0`，那么，`displayTriangles(g, order, p1, p2, p3)`方法完成下面的任务：

- 1) 在第53~55行显示连接三个点`p1`、`p2`、`p3`的三角形，如图20-10a所示。
- 2) 获取`p1`和`p2`的中点（第58行），`p2`和`p3`的中点（第59行），以及`p3`和`p1`的中点（第60行），如图20-10b所示。
- 3) 使用递减的阶数来递归地调用`displayTriangles`，以显示三个更小的思瑞平斯基三角形（第63~66行）。注意，每个小的思瑞平斯基三角形除了阶数会少一个之外，其结构和原始的大思瑞平斯基三角形是一样的，如图20-10b所示。

在`SierpinskiTrianglePanel`中显示思瑞平斯基三角形。内部类`SierpinskiTrianglePanel`中的`order`属性表明思瑞平斯基三角形的阶数。16.4节中介绍的`Point`类表示组件上的一个点。`midpoint(Point p1, Point p2)`方法返回`p1`和`p2`的中点（第72~74行）。

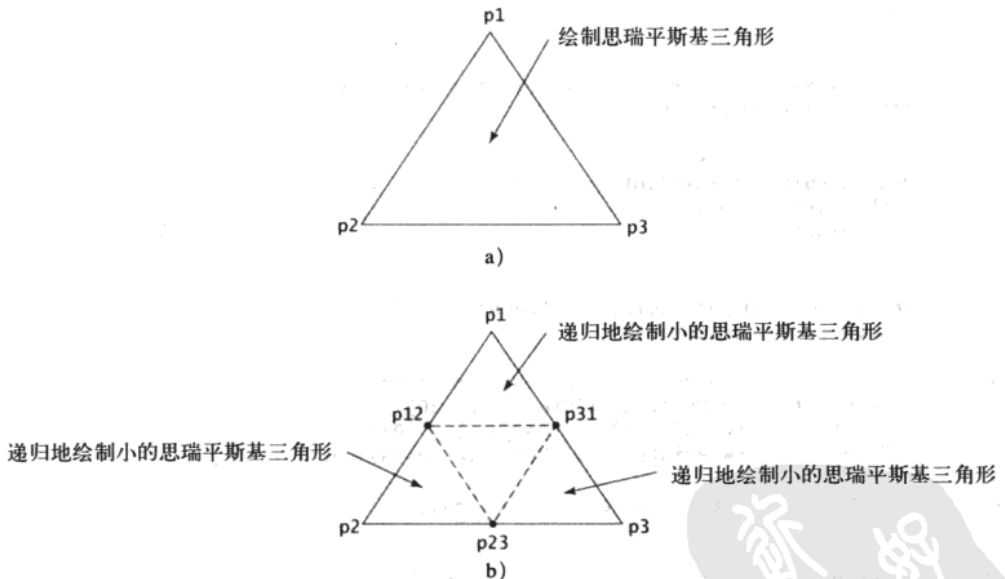


图20-10 绘制一个思瑞平斯基三角形会引起对绘制三个小的思瑞平斯基三角形的调用

20.9 问题：八皇后

本节给出在本章开始时提到的八皇后问题的递归解决方案。它的任务就是找出在棋盘上每行放一个皇后且不出现两个皇后互相攻击的解决方案。可以使用一个二维数组来表示一个棋盘。但是，因为每行都只能有一个皇后，所以，使用一个一维数组来表示皇后在行上的位置就足够了。因此，可以如下定义数组`queens`：

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

将j赋值给queens[i]表示皇后放置在i行j列。图20-11a给出如图20-11b所示的棋盘对应的数组queens的内容。

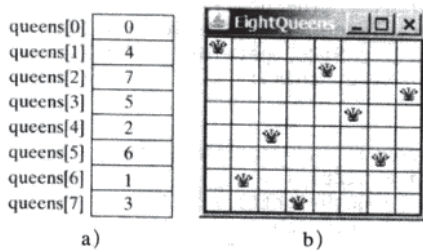


图20-11 queens[i]表示第i行的皇后的位置

程序清单20-10 EightQueens.java

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class EightQueens extends JApplet {
5     public static final int SIZE = 8; // The size of the chessboard
6     private int[] queens = new int[SIZE]; // Queen positions
7
8     public EightQueens() {
9         search(0); // Search for a solution from row 0
10        add(new ChessBoard(), BorderLayout.CENTER); // Display solution
11    }
12
13    /** Check if a queen can be placed at row i and column j */
14    private boolean isValid(int row, int column) {
15        for (int i = 1; i <= row; i++)
16            if (queens[row - i] == column // Check column
17                || queens[row - i] == column - i // Check upleft diagonal
18                || queens[row - i] == column + i // Check upright diagonal)
19                return false; // There is a conflict
20        return true; // No conflict
21    }
22
23    /** Search for a solution starting from a specified row */
24    private boolean search(int row) {
25        if (row == SIZE) // Stopping condition
26            return true; // A solution found to place 8 queens in 8 rows
27
28        for (int column = 0; column < SIZE; column++) {
29            queens[row] = column; // Place a queen at (row, column)
30            if (isValid(row, column) && search(row + 1))
31                return true; // Found, thus return true to exit for loop
32        }
33
34        // No solution for a queen placed at any column of this row
35        return false;
36    }
37
38    class ChessBoard extends JPanel {
39        private Image queenImage =
40            new ImageIcon("image/queen.jpg").getImage();
41
42        ChessBoard() {
43            this.setBorder(BorderFactory.createLineBorder(Color.BLACK, 2));
44        }
45
46        protected void paintComponent(Graphics g) {
47            super.paintComponent(g);

```



```

48
49 // Paint the queens
50 for (int i = 0; i < SIZE; i++) {
51     int j = queens[i]; // The position of the queen in row i
52     g.drawImage(queenImage, j * getWidth() / SIZE,
53               i * getHeight() / SIZE, getWidth() / SIZE,
54               getHeight() / SIZE, this);
55 }
56
57 // Draw the horizontal and vertical lines
58 for (int i = 1; i < SIZE; i++) {
59     g.drawLine(0, i * getHeight() / SIZE,
60               getWidth(), i * getHeight() / SIZE);
61     g.drawLine(i * getWidth() / SIZE, 0,
62               i * getWidth() / SIZE, getHeight());
63 }
64 }
65 }
66 }

```

程序调用 `search(0)` (第9行) 开始搜索在第0行的解决方案, 它递归地调用 `search(1)`, `search(2)`, ..., `search(7)` (第30行)。

如果所有的行都被填满, 那么递归的 `search(row)` 方法返回 `true` (第25~26行)。该方法在一个 `for` 循环中检测一个皇后是否放置在第0列, 第1列, 第2列, ..., 第7列中 (第28行)。将一个皇后放置在某一列中 (第29行)。如果这种放法是合法的, 调用 `search(row+1)` 递归地查找下一行 (第30行)。如果查找是成功的, 返回 `true` 以退出这个 `for` 循环 (第31行)。在这种情况下, 无须查找某行的下一列。如果没有将一个皇后放置在这一行的任意一列的解决方案, 这个方法返回 `false` (第35行)。

假设调用行 `row` 为3的 `search(row)`, 如图20-12a所示。这个方法会以第0列, 第1列, 第2列, ... 这样的顺序填充一个皇后。对于每次尝试, 调用 `isValid(row, column)` 方法 (第30行) 来检测是否将一个皇后放在指定的位置会引起和以前放置的皇后的冲突。它确保没有皇后放在同一列 (第16行), 没有皇后放在左上对角线上 (第17行), 没有皇后放在右上对角线上 (第18行), 如图20-12a所示。如果 `isValid(row, column)` 返回 `false`, 那就检查下一列, 如图20-12b所示。如果 `isValid(row, column)` 返回 `true`, 那就递归地调用 `search(row+1)`, 如图20-12d所示。如果 `search(row+1)` 返回 `false`, 那就检查前一行的下一列, 如图20-12c所示。

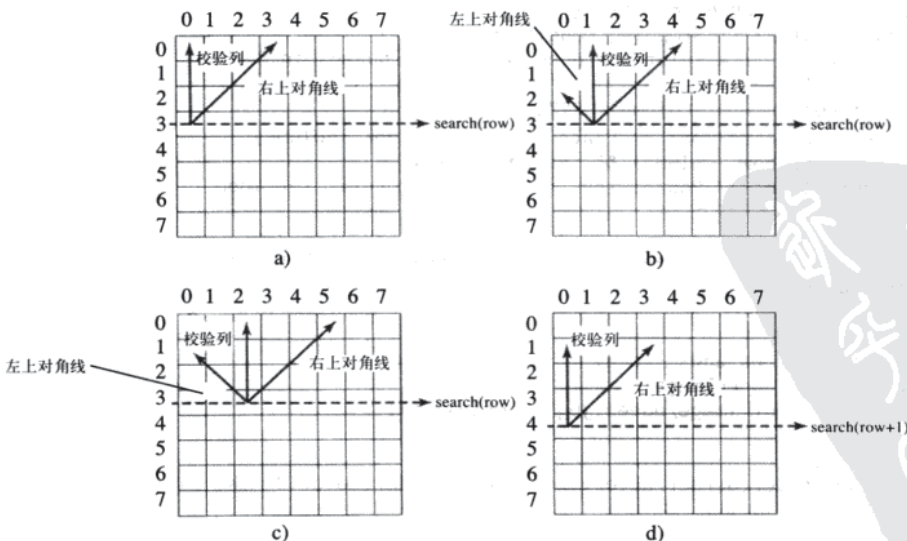


图20-12 调用 `search(row)` 在某行的一列上填充皇后

20.10 递归与迭代

递归是程序控制的一种替代形式，实质上就是不用循环控制的重复。使用循环时，可以指定一个循环体。循环控制结构控制循环体的重复。在递归中，方法重复地调用自己。必须使用一条选择语句来控制是否继续递归调用该方法。

递归会产生相当大的系统开销。程序每调用一个方法，系统就要给方法中所有的局部变量和参数分配空间。这就要占用大量的内存，还需要额外的时间来处理这些附加的空间。

任何用递归解决的问题都可以用迭代非递归地解决。递归有很多副作用：它耗费了太多的时间并占用太多的内存。那么，为什么还要用它呢？因为在某些情况下，本质上有递归特性的问题很难用其他方法解决，而递归可以给出一个清晰、简单的解决方案。像目录大小问题、汉诺塔问题和分形问题的例子都是不使用递归就很难解决的问题。

应该根据要解决的问题的本质和我们对这个问题的理解来决定是用递归还是用迭代。根据经验，选择使用递归还是迭代的原则，就是看它能否给出一个反映问题本质的直观解法。如果迭代的解决方案是显而易见的，那就使用迭代。迭代通常都比选择递归效率更高。

注意 递归程序可能会用完内存，引起一个StackOverflowError错误。

提示 如果关注程序的性能，就要避免使用递归，因为它会比迭代占用更多的时间且浪费更多的内存。

20.11 尾递归

如果在从递归调用返回时没有待定的操作要完成，那么这个递归方法就称为尾递归（tail recursive）。例如，因为在程序清单20-4中的第12行递归调用isPalindrome之后没有待定的操作，所以，递归的isPalindrome方法（第6~13行）就是尾递归的。但是，在程序清单20-1中，因为从每个递归调用返回时都有一个名为multiplication的待定操作要完成，所以，递归的factorial方法（第16~21行）就不是尾递归的。

因为当最后一个递归调用结束时，方法也该结束，所以尾递归是很必要的。因此，无须将中间调用存储在栈中。某些编译器可以优化尾递归以减小栈空间。

通常，可以使用辅助参数将非尾递归方法转换为递归方法。使用这些参数来控制结果，思路是将待定的操作和辅助参数以一种递归调用不再有待定操作的形式相结合。可能会定义一个带辅助参数的新的辅助递归方法，这个方法可以重载名字相同但签名不同的原始方法。例如，程序清单20-1中的factorial方法可以写成尾递归形式，如下所示：

```
1 /** Return the factorial for a specified number */
2 public static long factorial(int n) {
3     return factorial(n, 1); // Call auxiliary method
4 }
5
6 /** Auxiliary tail-recursive method for factorial */
7 private static long factorial(int n, int result) {
8     if (n == 1)
9         return result;
10    else
11        return factorial(n - 1, n * result); // Recursive call
12 }
```

第一个factorial方法只是调用了第二个辅助方法（第3行）。第二个方法包括了一个辅助参数result，它存储了n的阶乘的结果。这个方法在第11行被递归地调用。在返回调用之后，就没有了待定的操作。最终的结果在第9行返回，它也是在第3行调用factorial(n,1)的返回值。

关键术语

base case (基础情况)

infinite recursion (无限递归)

recursive method (递归方法)

recursive helper method (递归辅助方法)

stopping condition (终止条件)

tail recursion (尾递归)

本章小结

- 递归方法是一个直接或间接调用自己的方法。要终止一个递归方法，必须有一个或多个基础情况。
- 递归是程序控制的另外一种形式。本质上它是没有循环控制的重复。对于用其他方法很难解决而本质上是递归的问题，使用递归可以给出简单、清楚的解决方案。
- 为了进行递归调用，有时候需要修改原始方法使其接收附加的参数。为达到这个目的，可以定义递归的辅助方法。
- 递归需要相当大的系统开销。程序每调用一个方法一次，系统必须给方法中所有的局部变量和参数分配空间。这就要消耗大量的内存，并且需要额外的时间来管理这些附加的空间。
- 如果从递归调用返回时没有待定的操作要完成，这个递归的方法就称为尾递归 (tail recursive)。某些编译器会优化尾递归以减少栈空间。

复习题

20.1~20.3节

20.1 什么是递归的方法？描述递归的方法的特点。什么是无限递归？

20.2 编写一个递归的数学定义来计算 2^n ，其中 n 为正整数。

20.3 编写一个递归的数学定义来计算 x^n ，其中 n 为正整数， x 为实数。

20.4 编写一个递归的数学定义来计算 $1+2+3+\cdots+n$ ，其中 n 为正整数。

20.5 方法factorial(6)会调用程序清单20-1中的factorial方法多少次？

20.6 方法fib(6)会调用程序清单20-2中的fib方法多少次？

20.4~20.6节

20.7 分别使用程序清单20-3和20.4中定义的方法，给出isPalindrome("abcba")的调用栈。

20.8 使用程序清单20-5中定义的方法，给出selectionSort(new double[] {2,3,5,1})的调用栈。

20.9 什么是递归的辅助方法？

20.7节

20.10 为了调用moveDisks(5, 'A', 'B', 'C')，会调用程序清单20-8中的moveDisks方法多少次？

20.9节

20.11 下面的语句中哪些是正确的：

- 任何递归方法都可以转换为非递归方法。
- 执行递归方法比执行非递归方法要占用更多的时间和内存。
- 递归方法总是比非递归方法简单一些。
- 递归方法中总是有一个选择语句检查是否达到基础情况。

20.12 引起栈溢出异常的原因是什么？

综合题

20.13 给出下面程序的输出：

```
public class Test {
    public static void main(String[] args) {
        System.out.println(
            "Sum is " + xMethod(5));
    }

    public static int xMethod(int n) {
        if (n == 1)
            return 1;
        else
            return n + xMethod(n - 1);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        xMethod(1234567);
    }

    public static void xMethod(int n) {
        if (n > 0) {
            System.out.print(n % 10);
            xMethod(n / 10);
        }
    }
}
```

20.14 给出下面两个程序的输出：

```
public class Test {
    public static void main(String[] args) {
        xMethod(5);
    }

    public static void xMethod(int n) {
        if (n > 0) {
            System.out.print(n + " ");
            xMethod(n - 1);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        xMethod(5);
    }

    public static void xMethod(int n) {
        if (n > 0) {
            xMethod(n - 1);
            System.out.print(n + " ");
        }
    }
}
```

20.15 下面的方法有什么错误？

```
public class Test {
    public static void main(String[] args) {
        xMethod(1234567);
    }

    public static void xMethod(double n) {
        if (n != 0) {
            System.out.print(n);
            xMethod(n / 10);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        System.out.println(test.toString());
    }

    public Test() {
        Test test = new Test();
    }
}
```

20.16 说明本章中的尾递归方法。

20.17 使用尾递归改写程序清单20-2中的fib方法。

编程练习题

20.2~20.3节

*20.1（计算阶乘）使用14.12节介绍的BigInteger类，求得大数字的阶乘（例如，100!）。编写一个程序，提示用户输入一个整数，然后显示它的阶乘。使用递归实现这个方法。

*20.2（斐波那契数）使用迭代改写程序清单20-2中的fib方法。

提示 不使用递归来计算fib(n)，首先要得到fib(n-2)和fib(n-1)。设f0和f1表示前面的两个斐波那契数，那么当前的斐波那契数就是f0+f1。这个算法可以描述为如下所示：

```
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)

for (int i = 1; i <= n; i++) {
    currentFib = f0 + f1;
    f0 = f1;
    f1 = currentFib;
}
```

// After the loop, currentFib is fib(n)

*20.3 (使用递归求最大公约数) 求最大公约数的gcd(m,n)方法也可以如下递归地定义:

- 如果 $m \% n$ 为0, 那么gcd(m,n)的值为n。
- 否则, gcd(m,n)就是gcd(n, $m \% n$)。

编写一个递归的方法来求最大公约数。编写一个测试程序, 计算gcd(24, 16)和gcd(255, 25)。

20.4 (对数求和) 编写一个递归的方法来计算下面的级数:

$$m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

20.5 (对数求和) 编写一个递归的方法来计算下面的级数:

$$m(i) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \frac{4}{9} + \frac{5}{11} + \frac{6}{13} + \cdots + \frac{i}{2i+1}$$

*20.6 (对数求和) 编写一个递归的方法来计算下面的级数:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{i}{i+1}$$

*20.7 (斐波那契数列) 修改程序清单20-2, 使程序可以找出调用fib方法的次数。

提示 使用一个静态变量, 每当调用这个方法时, 该变量就加1。

20.4节

*20.8 (以逆序输出一个整数中的数字) 编写一个递归的方法, 使用下面的方法头在控制台上以逆序显示一个int型的值:

```
public static void reverseDisplay(int value)
```

例如, reverseDisplay(12345)显示的是54321。

*20.9 (以逆序输出一个字符串中的字符) 编写一个递归的方法, 使用下面的方法头在控制台上以逆序显示一个字符串:

```
public static void reverseDisplay(String value)
```

例如, reverseDisplay("abcd")显示dcba。

*20.10 (字符串中某个指定字符出现的次数) 编写一个递归的方法, 使用下面的方法头求一个指定字符在字符串中出现的次数。

```
public static int count(String str, char a)
```

例如, count("Welcome", 'e')会返回2。

*20.11 (使用递归求一个整数各位数之和) 编写一个递归的方法, 使用下面的方法头计算一个整数中各位数之和:

```
public static int sumDigits(long n)
```

例如, sumDigits(234)返回的是2+3+4=9。

20.5节

**20.12 (以逆序打印字符串中的字符) 使用辅助方法改写练习题20.9, 将子串的high下标传递给这个方法。辅助方法头为:

```
public static void reverseDisplay(String value, int high)
```

*20.13 (找出数组中的最大数) 编写一个递归的方法, 返回一个数组中的最大整数。

*20.14 (求字符串中大写字母的个数) 编写一个递归的方法, 返回一个字符串中大写字母的个数。

*20.15 (字符串中某个指定字符出现的次数) 使用辅助方法改写练习题20.10, 将子串的high下标传递给这个方法。辅助方法头为:

```
public static int count(String str, char a, int high)
```

*20.16 (求数组中大写字母的个数) 编写一个递归的方法, 返回一个字符串数组中大写字母的个数。需要定义下面两个方法。第二个方法是一个递归的辅助方法。

```
public static int count(char[] chars)
public static int count(char[] chars, int high)
```

*20.17 (数组中某个指定字符出现的次数) 编写一个递归的方法, 求出数组中一个指定字符出现的次数。需要定义下面两个方法, 第二个方法是一个递归的辅助方法。

```
public static int count(char[] chars, char ch)
public static int count(char[] chars, char ch, int high)
```

20.6节

*20.18 (汉诺塔) 修改程序清单20-8, 使程序可以求得将 n 个盘子从塔A移到塔B所需的移动次数。

提示 使用一个静态变量, 每当调用方法一次, 该变量就加1。

*20.19 (思瑞平斯基三角形) 修改程序清单20-9, 开发一个applet, 让用户使用Increase和Decrease按钮将当前阶数增1或减1, 如图20-13a所示。初始阶数为0。如果当前阶数为0, 就忽略Decrease按钮。

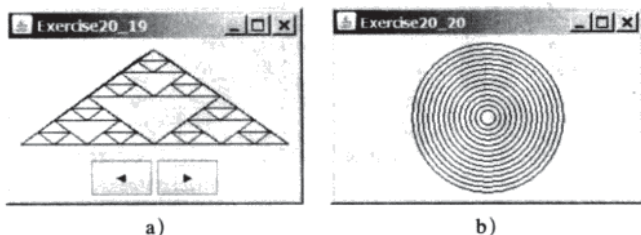


图20-13 a) 编程练习题20.19使用Increase和Decrease按钮将当前阶数增加1或减小1;

b) 练习题20.20使用递归方法绘制多个圆

20.20 (显示多个圆) 编写一个Java applet显示多个圆, 如图20-13b所示。这些圆都处于面板的中心位置。两个相邻圆之间相距10个像素, 面板和最大圆之间也相距10个像素。

综合题

*20.21 (将十进制数转换为二进制数) 编写一个递归的方法, 将一个十进制数转换为一个二进制数的字符串。方法头如下:

```
public static String decimalToBinary(int value)
```

*20.22 (将十进制数转换为十六进制数) 编写一个递归的方法, 将一个十进制数转换为一个十六进制数的字符串。方法头如下:

```
public static String decimalToHex(int value)
```

*20.23 (将二进制数转换为十进制数) 编写一个递归的方法, 将一个二进制数转换为一个十进制数的字符串。方法头如下:

```
public static int binaryToDecimal(String binaryString)
```

*20.24 (将十六进制数转换为十进制数) 编写一个递归的方法, 将一个十六进制数转换为一个十进制数的字符串。方法头如下:

```
public static int hexToDecimal(String hexString)
```

**20.25 (字符串排列) 编写一个递归的方法, 打印输出一个字符串的所有排列。例如, 对于字符串abc, 输出为:

```
abc
acb
bac
bca
cab
cba
```

提示 定义下面的两个方法, 第二个方法是一个辅助方法。

```
public static void displayPermuation(String s)
public static void displayPermuation(String s1, String s2)
```

第一个方法简单地调用displayPermuation("",s)。第二个方法使用循环,将一个字符从s2移到s1,并使用新的s1和s2递归地调用该方法。基础情况是s2为空,将s1打印到控制台。

****20.26** (创建一个迷宫) 编写一个applet,在迷宫中寻找一条路径,如图20-14a所示。该迷宫由一个 8×8 的棋盘表示。路径必须满足下列条件:

- 路径在迷宫的左上角单元和右下角单元之间。
- applet允许用户在一个单元格中放入或移走一个标志。路径由相邻的未放标志的单元格组成。如果两个单元格在水平方向或垂直方向相邻,但在对角线方向上不相邻,那么就称它们是相邻的。
- 路径不包含能形成一个正方形的单元格。例如,在图20-14b中的路径就不满足这个条件。(这个条件使得面板上的路径很容易识别。)

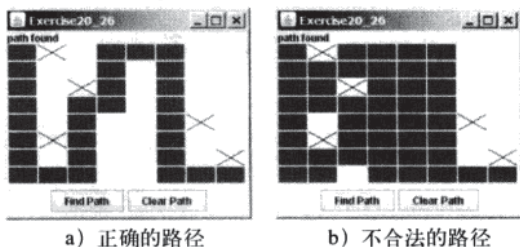


图20-14 程序求出从左上角到右下角的路径

****20.27** (科赫雪花分形) 本章给出了思瑞平斯基三角形分形。在本练习中,要编写一个applet,显示另一个称为科赫雪花(Koch snowflake)的分形,这是根据一位著名的瑞典数学家的名字命名的。科赫雪花按如下方式产生:

- (1) 从一个等边三角形开始,将其作为0阶(或0级)科赫分形,如图20-15a所示。
- (2) 将图形中的每条边分成三个相等的线段,以中间的线段作为底边向外画一个等边三角形,产生1阶科赫分形,如图20-15b所示。
- (3) 重复步骤(2)产生2阶科赫分形,3阶科赫分形, ..., 如图20-15c~图20-15d所示。

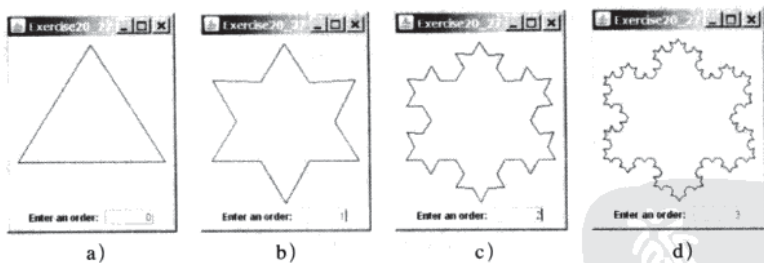


图20-15 科赫雪花是一个从三角形开始的分形

****20.28** (非递归目录大小) 不使用递归改写程序清单20-7。

***20.29** (某个目录下的文件数目) 编写一个程序,提示用户输入一个目录,然后显示该目录下的文件数。

****20.30** (找出单词) 编写一个程序,递归地找出某个目录下的所有文件中某个单词出现的次数。从命令行如下传递参数:

```
java Exercise20_30 dirName word
```

****20.31** (替换单词) 编写一个程序,递归地用一个新单词替换某个目录下的所有文件中出现的某个单词。从命令行如下传递参数:

```
java Exercise20_31 dirName oldWord newWord
```

***20.32 (游戏: 骑士的旅途) 骑士的旅途是一个古老的谜题。它的目的是使骑士从棋盘上的任意一个正方形开始移动, 经过其他的每个正方形一次, 如图20-16a所示。注意, 骑士只能做L形的移动 (两个空格在一个方向上而一个空格在垂直的方向上)。如图20-16b所示, 骑士可以移动到八个正方形的位置。编写一个程序, 在applet中显示骑士的移动, 如图20-16c所示。

提示 这个问题的穷举方法是将骑士从一个正方形随意地移动到另一个可用的正方形。使用这样的方法, 程序将需要很多时间来完成。比较好的方法是采用一些探索技巧。依据骑士目前的位置, 它可以有两个、三个、四个、六个或八个可能的移动线路。直觉上讲, 应该首先尝试将骑士移动到最小的可访问的正方形, 将那些更多的可访问的正方形保留为开放的, 这样, 在查找的结尾就会有更好的成功机会。

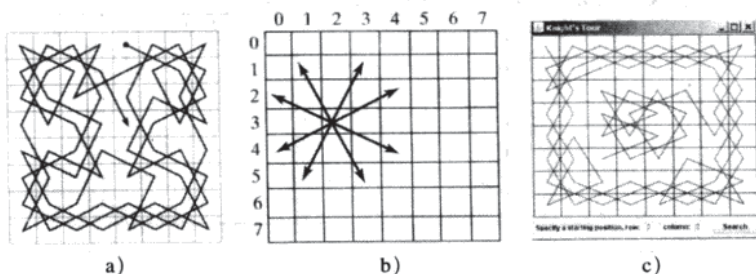


图20-16 a) 骑士遍历所有的正方形一次; b) 骑士作L形的移动; c) applet显示骑士的旅途路径

***20.33 (游戏: 骑士旅途的动画) 编写一个骑士旅途的问题的applet。applet应该允许用户将骑士放到任何一个起始正方形, 并点击Solve按钮, 用动画展示骑士沿着路径的移动, 如图20-17所示。



图20-17 骑士沿着路径遍历

**20.34 (游戏: 九宫格) 编写一个程序, 使用递归来解决九宫格问题。

**20.35 (H树分形) 一个H树分形如下定义:

- (1) 从字母H开始。H的三条线长度一样, 如图20-18a所示。
- (2) 字母H (以它的sans-serif形式, H) 有四个端点。以这四个端点为中心位置绘制一个1阶H树, 如图20-18b所示。这些H的大小是包括这四个端点的H的一半。
- (3) 重复步骤 (2) 来创建2阶, 3阶, ...的H树, 如图20-18c~图20-18d所示。

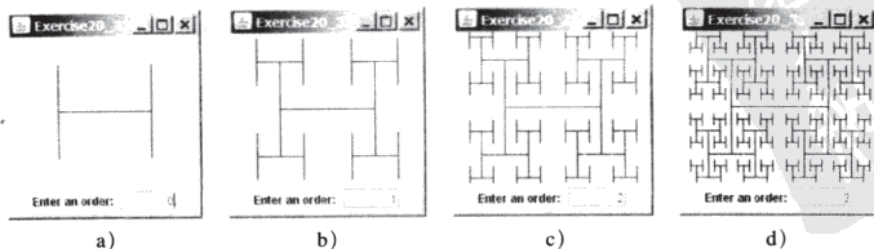


图20-18 H树是一个从H形状的三个等长的线开始的分形

在VLSI中，H树设计为一个时钟分布网络，以相同的传播延时将定时信号路由给芯片的所有部件。编写applet来绘制一个H树，如图20-18所示。

***20.36 (游戏：所有九宫格的解决方案) 改写练习题20.34，找出九宫格问题的所有可能的解决方案。

***20.37 (游戏：多种八皇后的解决方案) 编写一个applet在滚动窗格中显示八皇后谜题的所有可能的解决方案，如图20-19所示。对每个解决方案都放置一个标签来表示解决方案的个数。

提示 将所有的解决方案面板都放到一个面板中，然后将这个面板放入JScrollPane。解决方案面板类应该覆盖getPreferredSize()方法以确保正确地显示一个解决方案面板。参见程序清单15-3了解如何覆盖getPreferredSize()。

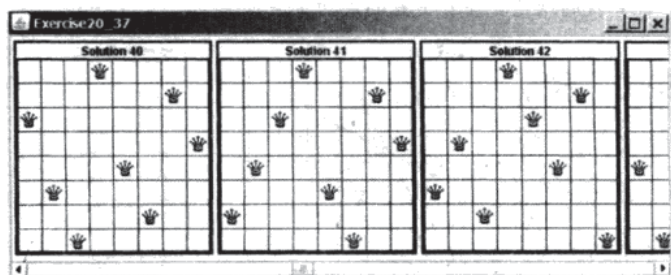


图20-19 所有的解决方案都放置在一个滚动窗格中

**20.38 (递归树) 编写一个applet来显示一个递归树，如图20-20所示。

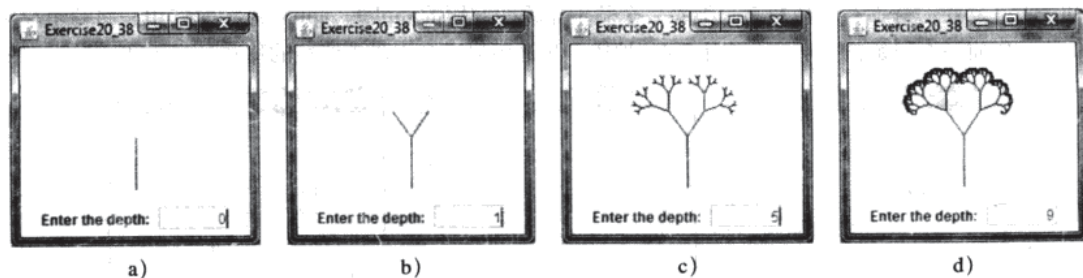


图20-20 一个带特定深度的递归树

**20.39 (拖动树) 修改练习题20.38，将树移动到鼠标指针所在的位置。