第 3 章 递归算法

本章主要内容

- 递归算法简介；
- 线性递归与非线性递归；
- 问题与子问题；
- 递归与迭代；
- 多重递归；
- 经典递归；
- 优化递归。

递归算法是非常重要的算法，是很多算法的基础。递归算法不仅能使代码优美简练，容易理解解决问题的思路或发现数据的内部逻辑规律，而且具有很好的可读性。递归算法是分治算法思想的重要体现，或者说分治算法思想来源于递归：将规模大的问题逐步分解成规模小的问题，最终解决规模大的问题。和排序算法不同，许多经典的排序算法已经日臻完善，在许多应用中只要选择一种排序算法直接使用即可（见第 4 章和第 12 章），而对于递归算法，真正理解递归算法内部运作机制的细节才能针对实际问题写出正确的递归算法。所以，本书单独列出一章讲解递归算法。

3.1 递归算法简介

一个方法在执行过程中又调用了自身，形成了递归调用，这样的方法被称为递归方法或递归算法。递归方法是一个递归过程，方法调用自身一次就是一次递归。每一次递归又导致方法调用自身一次，形成下一次递归。结束递归需要条件，当这个条件满足时递归过程会立刻结束，即在某次递归中方法不再调用自身，结束递归。如果在某次递归中方法不再调用自身，那么此次递归就是方法最后一次调用自身，从递归开始到方法最后一次调用自身，方法被调用的总次数记作 $R(n)$，那么 $R(n)$是依赖于一个正整数 n 的函数。

假设方法的名字是 f，下面进一步说明递归过程中压栈、弹栈的细节。

递归方法 f 的递归过程是，第 k 次调用 f 需要等待第 $k+1$ 次调用 f 结束执行后才能结束本次调用的执行，那么第 $R(n)$次（最后一次）调用 f 结束执行后，就会依次使得第 k 次调用 f 结束执行（$k=R(n)-1, R(n)-2, \cdots, 1$），如图 3.1 所示。

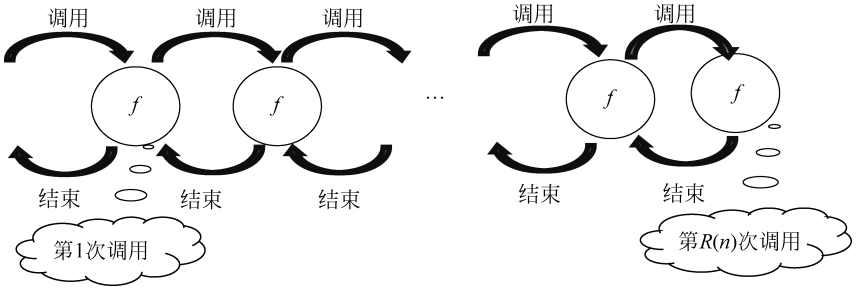

图 3.1 递归执行过程

方法被调用时，方法的（入口）地址会被压入栈（栈是一种先进后出的结构）中，称为压栈操作，同时方法的局部变量被分配内存空间。方法调用结束，会进行弹栈，称为弹栈操作，同时释放方法的局部变量所占用的内存空间。递归过程中的压栈操作会让栈的长度不断变大，而弹栈操作会让栈的长度不断变小，最终使栈的长度为 0，如图 3.2 所示，其中用方法的名字，例如 f，表示方法的地址。

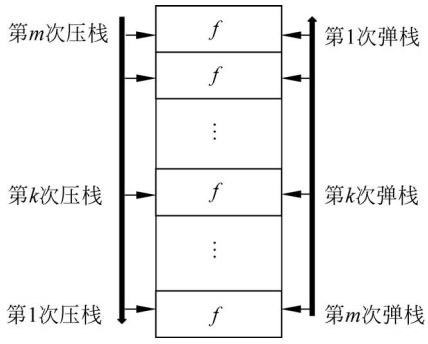

图 3.2　递归过程中的压栈、弹栈操作

1. 时间复杂度

递归方法是一个递归过程，从递归开始到递归结束，方法被调用的总次数 $R(n)$ 是依赖于一个正整数 n 的函数，那么递归方法中基本操作被执行的总次数 $T(n)$ 就依赖于递归的总次数 $R(n)$ 和每次递归时基本操作被执行的总次数，因此要针对具体的递归方法计算其时间复杂度。

2. 空间复杂度

递归过程的压栈操作增加栈的长度，弹栈操作减小栈的长度。需要注意的是，在递归过程中压栈操作和弹栈操作可能交替地进行，直到栈的长度为 0（见例 3-2），所以需要计算出递归过程中某一时刻（某一次递归）栈出现的最大长度和每次递归中方法的局部变量所占用的内存空间，即计算出栈的最大长度以及局部变量所占用的全部内存空间和所依赖的正整数的关系，这样才可以知道空间复杂度。大部分递归的空间复杂度通常是 $O(n)$，但也有的是 $O(\log n)$（见例 3-7）。

注意：递归会让栈的长度不断发生变化，如果栈的长度较大可能导致栈溢出，使得进程（运行的程序）被操作系统终止。

3.2　线性递归与非线性递归

▶ 3.2.1　线性递归

线性递归是指每次递归时方法调用自身一次。

例 3-1　判断一年的第 n 天是星期几。

为了知道一年的第 n 天是星期几，需要知道第 $n-1$ 天是星期几。本例中 Week 类的方法 f(int n)是一个递归方法，即 $f(n)$需要等待 $f(n-1)$返回的值，才能计算出自己的返回值，即才会知道第 n 天是星期几，这就形成了递归调用。f(int n)方法可以返回一年的第 n 天是星期几，其中返回值是 0 表示星期日，返回值是 1 表示星期一，……，返回值是 6 表示星期六。

Week. java

```
public class Week {
    public static int f(int n, int startWeekDay) {
        if(n == 1){                              //n 的值是 1,代表元旦
            return startWeekDay;                 //元旦的星期数
        }
        else {
            return (f(n - 1,startWeekDay) + 1) % 7;
        }
    }
}
```

递归方法 f(int n)的递归过程中的示意图如图 3.3 所示，向下方向的弧箭头表示方法被调用，向上方向的直箭头表示方法调用结束。图 3.4 示意了递归过程中压栈、弹栈操作产生的最长的栈。

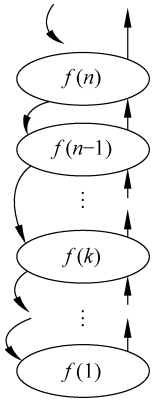

图 3.3 递归过程中方法的调用和结束

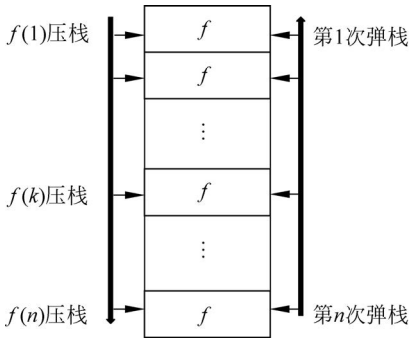

图 3.4 递归过程中最长栈的长度是 n

递归方法 f(int n)的递归的总次数：

$$R(n)=n$$

每次递归只有两个基本操作，一个是关系表达式 $n==1$，另一个是 return 语句，所以递归方法 f(int n)执行的基本操作的总次数是 $2\times n$，即递归结束后，执行的基本操作的总次数是 $2\times n$，因此 f(int n)的时间复杂度是 $O(n)$。

在递归的压栈操作过程中得到的栈的最大长度是 $R(n)=n$，每次递归只有两个局部变量——参数 n 和 startWeekDay，所占用的总内存是一个常量，例如 C，因此在递归过程中占用的最大内存是 $C\times n$，所以空间复杂度是 $O(n)$。

可以把例 3-1 的递归算法类比为，如果大家忘记了今天是星期几，就需要知道昨天是星期几，如此这般地向前翻日历，使得手中的日历越来越厚(相当于递归中的压栈，导致栈的长度在增加)，直到翻到了某个日历页上显示了星期几，就结束翻日历(相当于结束压栈)，然后一页一页地撕掉日历(相当于弹栈)，在撕掉日历的过程中，日历上出现了星期数，例如星期日、星期五等，即方法依次计算出自己的返回值。不断地撕掉日历退回到今天，就知道了今天是星期几。

```
如果元旦是星期日,
这一年的第126天是：星期一.
```

图 3.5 使用递归算法输出星期

在例 3-1 的主类 Example3_1 中，假设元旦是星期日，输出这一年的第 126 天是星期一，运行效果如图 3.5 所示。

Example3_1.java

```
public class Example3_1 {
    public static void main(String args[]) {
        int day = 126;                  //第 126 天
        int m = Week.f(day,0);          //0 表示星期日,即假设元旦是星期日
        String week[] = {"星期日","星期一","星期二","星期三","星期四","星期五","星期六"};
        String dayWeek = switch(m){
            case 0  -> week[0] ;
            case 1  -> week[1] ;
            case 2  -> week[2] ;
            case 3  -> week[3] ;
            case 4  -> week[4] ;
            case 5  -> week[5] ;
            case 6  -> week[6] ;
            default  -> "";
        };
```

```
            System.out.printf("\n 如果元旦是星期日.");
            System.out.printf("\n 这一年的第 %d 天是: %s.",day,dayWeek);
        }
    }
```

▶ 3.2.2 非线性递归

非线性递归是指每次递归时方法调用自身两次或两次以上。

例 3-2 使用递归算法求 Fibonacci 数列的第 n 项。

Fibonacci 数列的特点是，前两项的值都是 1，从第 3 项开始，每项的值是前两项的值的和。Fibonacci 数列如下：

1,1,2,3,5,8,13,21,…

其中，Fibonacci 类的方法 f(long n)返回参数 n 指定的 Fibonacci 数列的第 n 项的值。$f(n)$ 需要知道第 $n-1$ 项的值和第 $n-2$ 项的值，即需要 $f(n-1)$和 $f(n-2)$的返回值，这就形成了递归调用，即 f(long n)是一个递归方法。

Fibonacci. java

```
public class Fibonacci {
    public static long f(long n){
        long result = -1;
        if(n==1||n==2) {
            result= 1;
        }
        else if(n>=3){
            result = f(n-1)+f(n-2);
        }
        return result;
    }
}
```

在递归过程中，每次递归方法调用自身两次，使得每次递归出现两个递归“分支”，然后选择一个分支，继续递归，直到该分支递归结束，再沿着下一分支继续递归；当两个分支都递归结束，递归过程才结束，而且递归过程交替地进行压栈和弹栈操作，直到栈的长度为 0。

递归过程可以用一棵二叉树来刻画，二叉树的结点数目恰好是递归的总次数，如图 3.6 所示。该二叉树至少有 2^k-1 个结点($k<n$)，k 随着 n 的增大而增大。例如，参数 n 的值是 6 时

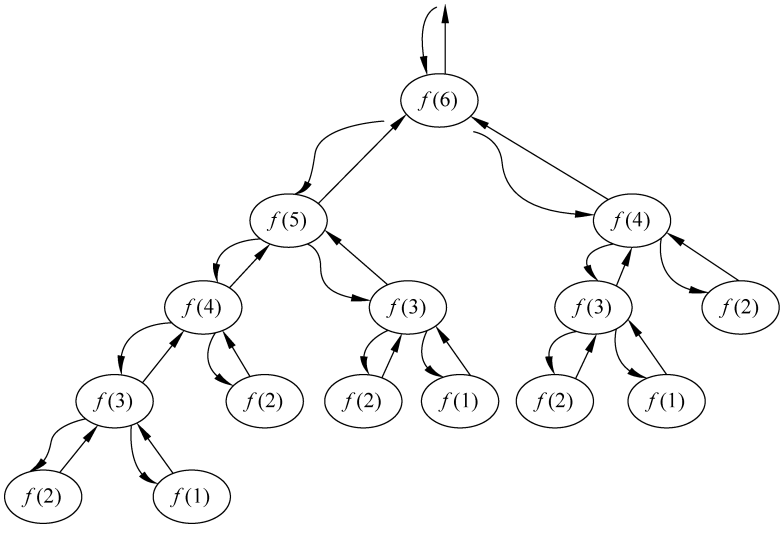

图 3.6 递归过程的二叉树示意图

(即求第 6 项的值时),调用方法的总次数(即递归的总次数)是 $2^4-1(n=6,k=4)$。在递归过程中 $f(\text{long } n)$被调用(压栈)和结束执行(弹栈),其中向下方向的弧箭头表示方法被调用(压栈),向上方向的直箭头表示方法结束(弹栈)。

k 随着 n 的增大而增大,在计算时间复杂度时可以设 $R(n)=2^n-1$,而每次递归的基本操作只有一个加法操作和一个 return 语句,即每次递归有两个基本操作,因此 $f(\text{long } n)$的执行过程(即递归过程)产生的基本操作的总次数 $T(n)$为:

$$T(n)=R(n)=2\times(2^n-1)=2^{n+1}-2$$

所以 $f(\text{long } n)$的时间复杂度是 $O(2^n)$。

递归过程是按照两个“分支”分别进行的,一个分支递归结束(压栈,弹栈结束),再进行另一个分支的递归。递归形成的二叉树至少有 2^k-1 个结点($k<n$),那么树的高度(或者叫深度)至少是 k(这是二叉树的简单规律)。递归过程交替地进行压栈和弹栈操作,因此在递归过程中栈的最大长度大于 k 小于 n,由于 k 随着 n 的增大而增大,所以 $f(\text{long } n)$的空间复杂度是 $O(n)$。图 3.6 中左侧的递归分支产生的压栈过程得到的最长栈如图 3.7 所示。

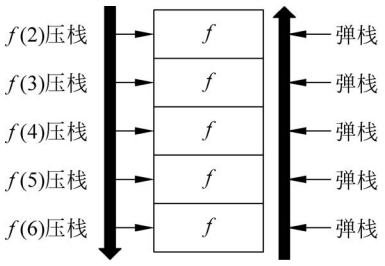

图 3.7 递归过程中的最长栈

注意:图 3.7 中 $f(2)$弹栈后,$f(3)$不马上弹栈,而是把 $f(1)$压栈,即 $f(1)$入栈。压栈、弹栈交替进行,直到递归结束。

```
Fibonacci数列的第22项是17711
黄金分割近似值:0.618(保留3位小数)
```

图 3.8 计算 Fibonacci 数列的某一项

例 3-2 中的主类 Example3_2 使用 Fibonacci 类的递归方法 $f(\text{long } n)$输出 Fibonacci 数列的第 22 项,并计算了黄金分割的近似值,运行效果如图 3.8 所示。

Example3_2.java

```
public class Example3_2 {
    public static void main(String args[]) {
        long item = 22;
        System.out.printf("Fibonacci 数列的第 %d 项是 %d\n",item,Fibonacci.f(item));
        System.out.printf("黄金分割近似值:%.3f(保留 3 位小数)\n",
                          (double)Fibonacci.f(19)/Fibonacci.f(20));
    }
}
```

Fibonacci 数列可以用来解释青蛙跳台阶问题。假设有 n 级台阶,青蛙每次只可以跳一个台阶或两个台阶,问青蛙完成跳 n 级台阶的任务(跳到最后一个台阶上算完成任务)一共有多少种跳法。

当 n 的值是 1 时,青蛙只有一种跳法,即跳一个台阶。当 n 的值是 2 时,青蛙有两种跳法:一种是每次跳一个台阶;另一种是每次跳两个台阶。当 n 的值是 3 时,青蛙可以选择先跳一个台阶,剩下的可能性跳法交给 $n-1$ 级台阶的情况;青蛙也可以先跳两个台阶,剩下的可能性跳法交给 $n-2$ 级台阶的情况。也就是说青蛙完成跳 n 级台阶的全部跳法的总数($n\geqslant 3$)满足 Fibonacci 数列,所以 Fibonacci 数列的第 $n+1$ 项的值就是青蛙跳 n 级台阶的全部跳法的总数。

注意:青蛙跳台阶使用递归是合理的,理由是跳台阶的各种可能性和起始方式有关,例如跳 4 级台阶,“1,1,2”和“2,2”是不同的跳法。

使用 Fibonacci 数列还可以计算黄金分割数的近似值。

$$f(n)/f(n+1) \quad (n=1,2,\cdots)$$

的极限是黄金分割数(0.618…,一个无理数)。

3.3 问题与子问题

一个问题的子问题就是数据规模比此问题的规模更小的问题。当一个问题可以分解成许多子问题时,就可以考虑用递归算法来解决这个问题。

例 3-3 计算 8+88+888+8888+…的前 n 项和。

计算前 n 项和与计算前 $n-1$ 项和就是问题和子问题的关系。如果解决了子问题,即计算出了前 $n-1$ 项和,那么前 n 项和的问题就解决了：前 n 项和等于前 $n-1$ 项和乘以 10 再加上 $8\times n$。

本例使用 SumAndMulti 类中的递归方法 long sum(long a,int n)返回形如

$$a+aa+aaa+aaaa+\cdots$$

的前 n 项和(a 是 1～9 中的某个数字)。递归方法 long multi(long n)返回 $n!$(n 的阶乘)。

SumMulti.java

```
public class SumMulti {
    public static long sum(long a,int n){
        long sum = 0;
        if(n==1) {
            sum= a;
        }
        else if(n>=2) {
            sum = sum(a,n-1)*10+n*a;
        }
        return sum;
    }
    public static long multi(int n){
        long result= -1;
        if(n==1) {
            result= 1;
        }
        else if(n>=2) {
            result = multi(n-1)*n;
        }
        return result;
    }
}
```

不难计算出 sum(long a,int n)和 multi(int n)的时间复杂度都是 $O(n)$。

```
8+88+888+…前5项和是98760
10的阶乘是3628800
```

图 3.9 计算连续和与阶乘

例 3-3 中的主类 Example3_3 使用 SumMulti 类中的递归方法计算了 8+88+888+…的前 5 项的连续和以及 10!(10 的阶乘),运行效果如图 3.9 所示。

Example3_3.java

```
public class Example3_3 {
    public static void main(String args[]) {
       int a = 8,
           n = 5;
       int m = 10;
       System.out.printf("%d+ %d%d+ %d%d%d…前%d项和是%d\n",
                                  a,a,a,a,a,n,SumMulti.sum(a,n));
```

```
            System.out.printf("%d的阶乘是%d\n",m,SumMulti.multi(m));
        }
    }
```

例 3-4 使用 Reverse 类的 reverseString(String s)方法返回参数 s 中的反转(倒置)字符序列。

反转长度为 n 的字符序列 $s_1s_2\cdots s_n$,这一问题的子问题是反转长度为 $n-1$ 的字符序列 $s_1s_2\cdots s_{n-1}$,将字符 s_n 放在反转后的字符序列 $s_{n-1}s_{n-2}\cdots s_1$ 的前面,变成 $s_ns_{n-1}\cdots s_1$,就解决了反转长度为 n 的字符序列的问题。

例 3-4 中 MaxInArray 类的 getMaxInArray(int []a)方法返回数组 a 中元素的最大值。求长度为 n 的数组的最大值的子问题是求长度为 $n-1$ 的数组的最大值,因为求出索引从 1 开始的子数组的元素的最大值 m 后,那么 m 和 $a[0]$的最大值就是原数组的元素的最大值。

Reverse. java

```
public class Reverse {
    public static String reverseString(String s){
        String str = null;
        int n = s.length();
        if(n==1) {
            str = s.charAt(0)+"";
        }
        else if(n>=2) {
            str = s.charAt(n-1)+""+reverseString(s.substring(0,n-1));
        }
        return str;
    }
}
```

MaxInArray. java

```
import java.util.Arrays;
public class MaxInArray{
    public static int getMaxInArray(int []a){
        if(a.length==1) {
            return a[0];
        }
        else {
            int [] aCopy = Arrays.copyOfRange(a,1,a.length);
            return Math.max(a[0],getMaxInArray(aCopy));
        }
    }
}
```

不难计算出 reverseString(String s)方法和 getMaxInArray(int []a)方法的时间复杂度和空间复杂度都是 $O(n)$。

例 3-4 的主类 Example3_4 使用 reverseString(String s)方法得到一个字符序列的反转(倒置),并判断一个英文单词是否为回文单词(回文单词和它的反转相同);使用 getMaxInArray(int []a)方法计算了数组中元素的最大值,运行效果如图 3.10 所示。

```
abcdefg的反转是gfedcba
racecar的反转是racecar,二者是否相同:true
racecar是回文单词.
[120, 2025, 1000, 980, 10, 879, 89]元素值的最大值:2025
```

图 3.10 反转字符序列以及求最大值

Example3_4.java

```
import java.util.Arrays;
public class Example3_4 {
      public static void main(String args[]) {
         String s = "abcdefg";
         String reverse = Reverse.reverseString(s);
         System.out.printf("%s的反转是%s\n",s,reverse);
         s = "racecar";
         reverse = Reverse.reverseString(s);
         boolean isSame = s.equals(reverse);
         System.out.printf("%s的反转是%s,二者是否相同:%b\n",s,reverse,isSame);
         if(isSame)
               System.out.println(s+"是回文单词.");
         int []a = {120,2025,1000,980,10,879,89};
         int max = MaxInArray.getMaxInArray(a);
         System.out.println(Arrays.toString(a)+"元素值的最大值:"+max);
      }
}
```

3.4 递归与迭代

递归的思想是，根据上一次操作的结果确定当前操作的结果，当前结果和上一次的结果相同或者需要根据上一次的结果来确定本次操作的结果。迭代的思想是，根据当前操作的结果确定下一次操作的结果。对于解决相同的问题，递归的代码简练，容易理解解决问题的思路或发现数据的内部逻辑规律，具有很好的可读性。迭代的代码可能比较复杂，处理数据的过程也可能比较复杂，所以迭代的代码不如递归简练。用递归算法能解决的问题也可以用迭代算法解决。由于迭代不涉及方法的递归调用，所以通常情况下递归算法的空间复杂度会大于迭代算法的空间复杂度，当递归过程中的递归总次数比较大时会导致栈溢出。

例 3-5 计算圆周率的近似值。

下列无穷级数的和是圆周率的 1/4。

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$$

例 3-5 中 ComputePI 类的 recursionMethod(int n)方法和 iterationMethod(int n)方法都用来返回圆周率的近似值，其中 recursionMethod(int n)方法使用的是递归，iterationMethod(int n)方法使用的是迭代。

ComputePI.java

```
public class ComputePI {
      public static double recursionMethod(int n) {              //递归方法
         double sum = 0;
         if(n==1)
               sum = 1;
         else if(n%2==0) {
               sum = recursionMethod(n-1)-1.0/(2*n-1);
         }
         else {
               sum = recursionMethod(n-1)+1.0/(2*n-1);
         }
         return sum;
      }
```

```
    public static double iterationMethod(int n) {          //迭代方法
        double sum = 0;
        for(int i = 1;i <= n;i++){
            if(i % 2 == 0){
                sum = sum - 1.0/(2 * i - 1);
            }
            else {
                sum = sum + 1.0/(2 * i - 1);
            }
        }
        return sum;
    }
}
```

不难计算出基于递归的 recursionMethod(int n)方法的时间复杂度和空间复杂度都是 $O(n)$，基于迭代的 iterationMethod(int n)方法的时间复杂度是 $O(n)$，但空间复杂度是 $O(1)$。

```
递归计算圆周率(保留6位小数):3.141705
迭代计算圆周率(保留6位小数):3.141705
```

图 3.11 计算圆周率的近似值

例 3-5 中的主类 Example3_5 使用 ComputePI 类中的方法计算圆周率的近似值，运行效果如图 3.11 所示。

Example3_5.java

```
public class Example3_5 {
    public static void main(String args[]){
        int n = 8887;
        double pi = ComputePI.recursionMethod(n) * 4;
        System.out.printf("递归计算圆周率(保留 6 位小数):%.6f\n",pi);
        pi = ComputePI.iterationMethod(n) * 4;
        System.out.printf("迭代计算圆周率(保留 6 位小数):%.6f",pi);
    }
}
```

注意：递归算法可能导致栈溢出，在主类中，当 n 的值较大时递归算法会导致栈溢出。

例 3-6 判断某个数是否为数组的元素值。

例 2-9 中的 binarySearch(int []array,int number)方法使用迭代判断 number 是否为数组 array 的元素值。这里的例 3-6 中的 SearchNumber 类的 binarySearch(int []array,int start,int end,int number)方法使用递归判断 number 是否为数组 array 的元素值，递归的时间复杂度是 $O(\log n)$，空间复杂度都是 $O(n)$(时间复杂度和空间复杂度与例 2-9 中的迭代方法 binarySearch(int []array,int number)的相同)。

SearchNumber.java

```
public class SearchNumber {
    public static int binarySearch(int []array,int start,int end,int number) {
        int mid = -1;
        int index = -1;
        mid = (start + end)/2;
        int midValue = array[mid];
        if(start > end){
            index = -1;
        }
        else if(number == midValue) {
            index = mid;
        }
        else if(number < midValue){
            end = mid - 1;
            index = binarySearch(array,start,end,number);
        }
```

```
            else if(number > midValue){
                    start = mid + 1;
                    index = binarySearch(array,start,end,number);
            }
            return index;
        }
    }
```

二分法在处理数据时处理的数据量每次减少一半，递归的总次数 k 的最大可能取值是使得数组的长度为 1（见例 2-9），即：

$$\frac{n}{2^k}=1,\quad k=\log n$$

所以递归的总次数 $R(n)$ 的最大可能取值是 $\log n$。由于每次递归的基本操作的总次数是一个常量，例如 C，因此递归过程的基本操作的总次数

$$T(n)=C\times\log n$$

所以 binarySearch(int []array,int start,int end,int number)方法的时间复杂度是 $O(\log n)$。

算法中影响空间复杂度的是一维数组 array 的长度 n，因此空间复杂度是 $O(n)$。

例 3-6 中的主类 Example3_6 使用 binarySearch(int []array,int start,int end,int number)方法判断某个数是否为数组 array 的元素值，运行效果如图 3.12 所示（例 3-6 使用了例 2-7 中的 BubbleSort 类的 sort(int []a)方法）。

```
[-11, 1, 12, 56, 89, 100, 128, 128, 129, 199, 200, 289]
-11在数组中,数组索引位置是0
128在数组中,数组索引位置是6
11不在数组中
129在数组中,数组索引位置是8
289在数组中,数组索引位置是11
```

图 3.12　判断某个数是否在数组中

Example3_6.java

```
import java.util.Arrays;
public class Example3_6 {
    public static void main(String args[]) {
        int number [] = {-11,128,11,129,289};
        int []a = {128,129,199,200,289,-11,1,12,56,89,100,128};
        BubbleSort.sort(a);
        System.out.println(Arrays.toString(a));
        for(int i = 0;i < number.length;i++) {
            int index = SearchNumber.binarySearch(a,0,a.length,number[i]);
            if(index ==-1)
                System.out.printf("%d不在数组中\n",number[i]);
            else
                System.out.printf("%d在数组中,数组索引位置是%d\n",number[i],index);
        }
    }
}
```

例 3-7　求两个正整数的最大公约数。

例 2-10 中 Euclidean 类的 gcd(int n,int m)方法使用的是迭代，本例中 Euclidean 类的 gcd(int n,int m)方法使用的是递归，两者都是求两个正整数的最大公约数，但例 2-10 中 gcd(int n,int m)方法的空间复杂度是 $O(1)$，这里的 gcd (int n,int m)方法的空间复杂度是 $O(\log n)$，两者的时间复杂度都是 $O(\log n)$。

本例的 gcd(int n,int m)方法要比例 2-10 中的方法更能简练地体现辗转相除：如果 $n\%$

$m(n \geqslant m)$不等于0，那么n和m的最大公约数与m和$n\%m$的最大公约数相同；如果$n\%m$等于0，两者的最大公约数就是m。

Euclidean.java

```
public class Euclidean {
     public static int gcd(int n,int m){
        n = Math.abs(n);
        m = Math.abs(m);
        if(n % m == 0){
             return m;
        }
        else {
             return gcd(m,n % m);
        }
     }
}
```

由于$n\%m<n/2$，即辗转相除会使n的值至少减少一半，那么递归总次数k会满足：

$$\frac{n}{2^k}=1, \quad k=\log n$$

即栈的最大长度是$\log n$，因此空间复杂度和时间复杂度都是$O(\log n)$。

```
12,6的最大公约数是6.
63,42的最大公约数是21.
```

图3.13 求最大公约数

例3-7中的主类Example3_7使用递归方法gcd(int n,int m)输出两个正整数的最大公约数，运行效果如图3.13所示。

Example3_7.java

```
public class Example3_7 {
     public static void main(String args[]) {
        int a = 6,b = 12;
        System.out.printf(" % d, % d 的最大公约数是 % d.\n",a,b,Euclidean.gcd(a,b));
        a = 63;
        b = 42;
        System.out.printf(" % d, % d 的最大公约数是 % d.\n",a,b,Euclidean.gcd(a,b));
     }
}
```

3.5 多重递归

所谓多重递归，是指一个递归方法调用另一个或多个递归方法，称该递归方法是多重递归方法。

这里用一个数字问题来说明多重递归：求n位十进制数中含有偶数个6的数(数的某位上是数字6)的个数，但不要求输出含有偶数个6的数。两位十进制数中含有偶数个6的数的个数是1(66含有两个6)，含有奇数个6的数的个数是17：

16,26,36,46,56,60,61,62,63,64,65,67,68,69,76,86,96

用$a(n)$表示n位十进制数中含有偶数个6的数的个数，$b(n)$表示n位十进制数中含有奇数个6的数的个数，$c(n)$表示n位十进制数中不含有6的数的个数。

对于$n>2$，有下列递推关系成立，这里的“=”是数学意义上的等号：

$$a(n)=9\times a(n-1)+b(n-1)$$
$$b(n)=9\times b(n-1)+a(n-1)+c(n-1)$$

$$c(n)=9\times c(n-1)$$

非常容易证明上述等式，因为对于任意一个$(n-1)$位十进制数，例如$a_1a_2\cdots a_{n-1}$，如果$a_1a_2\cdots a_{n-1}$中出现了偶数个6，那么n位十进制数$a_1a_2\cdots a_{n-1}p(p=1,2,3,4,5,7,8,9,0)$，即$p$取6以外的其他个位数字，都出现了偶数个6；如果$a_1a_2\cdots a_{n-1}$中出现了奇数个6，那么$n$位十进制数$a_1a_2\cdots a_{n-1}6$出现了偶数多6，所以有：

$$a(n)=9\times a(n-1)+b(n-1)$$

另外两个等式的论证道理类似。如果将$a(n)$、$b(n)$对应到递归方法，那么所对应的递归方法就是多重递归方法。

例 3-8 求n位十进制数中含有偶数个6、奇数个6以及不含有6的数的个数。

例3-8中DoubleRecursion类的a(int n)方法和b(int n)方法是多重递归方法，不难验证两者的时间复杂度都是$O(2^n)$，空间复杂度都是$O(n)$（验证方法和例3-2类似）。

DoubleRecursion. java

```
public class DoubleRecursion {
     //返回 n 位十进制数中出现偶数个数字 6 的数的个数
     public static int a(int n) {                              //双重递归
          int result = 0;
          if(n == 1) {
               result = 0;
          }
          else if(n == 2){
               result = 1;
          }
          else if(n > 2){
              result = 9 * a(n - 1) + b(n - 1);
          }
          return result;
     }
     //返回 n 位十进制数中出现奇数个数字 6 的数的个数
     public static int b(int n) {                              //多重递归
          int result = 0;
          if(n == 1) {
               result = 1;
          }
          else if(n == 2) {
              result = 17;
          }
          else if(n > 2){
              result = 9 * b(n - 1) + a(n - 1) + c(n - 1);
          }
          return result;
   }
   //返回 n 位十进制数中未出现数字 6 的数的个数
     public  static int c(int n) {                             //单递归
          int result = 0;
          if(n == 1) {
               result = 9;
          }
          else if(n == 2){
               result = 72;
          }
          else {
               result = 9 * c(n - 1);
          }
```

```
            return result;
        }
    }
```

例 3-8 中的主类 Example3_8 使用 DoubleRecursion 类的多重递归方法输出了 8 位十进制数中含有偶数个 6、奇数个 6 以及不含有 6 的数的个数等信息，运行效果如图 3.14 所示。

```
8位十进制数中出现偶数个数字6的个数是14076280
8位十进制数中出现奇数个数字6的个数是37659968
8位十进制数中未出现数字6的数量是38263752
8位数一共有：90000000
```

图 3.14 输出数字的有关信息

Example3_8.java

```
public class Example3_8 {
    public static void main(String args[]) {
        int n = 8;                          //8 位十进制数
        int sum = 0;
        int count = DoubleRecursion.a(n);
        sum += count;
        System.out.printf
       ("%d 位十进制数中出现偶数个数字 6 的个数是 %d\n",n,count);
       count = DoubleRecursion.b(n);
        sum += count;
        System.out.printf
       ("%d 位十进制数中出现奇数个数字 6 的个数是 %d\n",n,count);
       count = DoubleRecursion.c(n);
       sum += count;
        System.out.printf
       ("%d 位十进制数中未出现数字 6 的个数是 %d\n",n,count);
       System.out.println(n + "位数一共有:" + sum + "");
    }
}
```

3.6 经典递归

本节通过杨辉三角形、老鼠走迷宫和汉诺塔 3 个经典的递归进一步体会递归算法，递归算法不仅能使代码简练，容易理解解决问题的思路或发现数据的内部逻辑规律，而且具有很好的可读性，特别是汉诺塔递归，通过其递归算法能够洞悉其数据规律，给出相应的迭代算法。

▶ 3.6.1 杨辉三角形

杨辉三角形：

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10   10   5   1
…
```

最早出现于中国南宋的数学家杨辉在 1261 年所著的《详解九章算法》中。法国数学家帕斯卡(Pascal)在 1654 年发现该三角形，因此又称帕斯卡三角形。

例 3-9 输出杨辉三角形。

按照编程语言的习惯，行、列的索引都是从 0 开始的。杨辉三角形的主要规律是：杨辉三

角形的第 0 行有一个数，第 1 行有两个数，…… ，第 n 行有 $n+1$ 个数，第 n 行的第 0 列和最后一列的数都是 1，即第 0 列和第 n 列的数都是 1。用 $C(n,j)$ 表示第 n 行、第 j 列的数（$j=0,\cdots,n$），那么递归如下：

$$C(n,0)=1, C(n,j)=C(n-1,j-1)+C(n-1,j)(0<j<n), C(n,n)=1 \quad (3\text{-}1)$$

例 3-9 中 PascalTriangle 类的 C(int n,int j)方法是依据式(3-1)的递归算法，可以计算杨辉三角形的第 n 行、第 j 列上的数，即该方法返回杨辉三角形的第 n 行、第 j 列上的数。

PascalTriangle.java

```
public class PascalTriangle {
     public static long C(int n,int j){
          long result = 0;
          if(j==0||j==n){                    //每行的第 0 列和第 n 列上的数都是 1
               result = 1;
          }
          else {
             result = C(n-1,j-1)+C(n-1,j);
          }
          return result;
     }
}
```

PascalTriangle 类的 C(int n,int j)方法属于非线性递归方法，时间复杂度是 $O(n^2)$，空间复杂度是 $O(n)$。因为杨辉三角形的前 n 行中共有 $n(n+1)/2$ 个数，那么递归过程中方法被调用的总次数是 $n(n+1)/2$，所以 C(int n,int j)的时间复杂度是 $O(n^2)$。在递归过程中，根据

$$C(n,j)=C(n-1,j-1)+C(n-1,j)$$

可知，一个递归分支当栈的长度达到 n 时就会依次弹栈，返回到上一个递归分支，因此空间复杂度是 $O(n)$。

在组合数学中，对于二项式有一个经典的公式，即对于杨辉三角形的第 n 行、第 j 列（$j=0,\cdots,n$）上的数 $Y(n,j)$ 有如下递归：

$$Y(n,0)=1, Y(n,j)=Y(n,j-1)\times(n-j+1)/j(j>0, j<n), Y(n,n)=1 \quad (3\text{-}2)$$

例 3-9 中 YanghuiTriangle 类的 Y(int n,int j)方法是依据式(3-2)的递归算法，可以计算杨辉三角形的第 n 行、第 j 列上的数，即该方法返回杨辉三角形的第 n 行、第 j 列上的数。

YanghuiTriangle.java

```
public class YanghuiTriangle {
     public static long Y(int n,int j){
          long result = 0;
          if(j==0||j==n){                    //每行的第 0 列和第 n 列上的数都是 1
               result = 1;
          }
          else if(j<n&&j>0){
             result = Y(n,j-1)*(n-j+1)/j;
          }
          return result;
     }
}
```

YanghuiTriangle 类的 Y(int n,int j)方法属于线性递归方法，时间复杂度是 $O(n)$，空间复杂度是 $O(n)$。因为方法被调用的总次数是 n，所以 Y(int n,int j)的时间复杂度是 $O(n)$。从递归过程可以看出压栈导致栈的长度最大是 n，因此空间复杂度是 $O(n)$。

例 3-9 中的主类 Example3_9 输出了杨辉三角形的前 9 行，并比较了 YanghuiTriangle 类的 Y(int n,int j)方法和 PascalTriangle 类的 C(int n,int j)方法计算第 n 行、第 j 列上的数的耗时，时间复杂度是 $O(n)$的耗时明显小于时间复杂度是 $O(n^2)$的耗时，运行效果如图 3.15 所示。

```
    1
    1    1
    1    2    1
    1    3    3    1
    1    4    6    4    1
    1    5   10   10    5    1
    1    6   15   20   15    6    1
    1    7   21   35   35   21    7    1
    1    8   28   56   70   56   28    8    1
线性递归求第35行,第17列4537567650的耗时是415800(纳秒)
非线性递归求第35行,第17列4537567650的耗时是9849510100(纳秒)
```

图 3.15 输出杨辉三角形的前 9 行，并比较两个递归的耗时

Example3_9.java

```
public class Example3_9 {
     public static void main(String args[]){
          long result = 0;
          int row = 8;
          for(int n = 0;n <= row;n++) {              //输出杨辉三角形的前 row + 1 行
              for(int j = 0;j <= n;j++){
                   result = PascalTriangle.C(n,j);
                   System.out.printf("%5d",result);
              }
              System.out.println();
          }
          int n = 35,j = n/2 ;
          long startTime = System.nanoTime();
          result = YanghuiTriangle.Y(n,j);
          long estimatedTime = System.nanoTime() - startTime;
          System.out.printf("线性递归求第%d行,第%d列%d的耗时是%d(纳秒)\n",
                                  n,j,result,estimatedTime);
          startTime = System.nanoTime();
          result = PascalTriangle.C(n,j);
          estimatedTime = System.nanoTime() - startTime;
          System.out.printf("非线性递归求第%d行,第%d列%d的耗时是%d(纳秒)\n",
                                 n,j,result,estimatedTime);
     }
}
```

▶ 3.6.2 老鼠走迷宫

本节用 m 行 n 列的二维数组模拟迷宫。

假设老鼠走迷宫的递归方法是 move(int[][] a,int i,int j)，老鼠在迷宫中某点 $p=(i,j)$ 的递归办法是：首先从 p 点向东调用 move(int[][] a,int i,int j)，如果找到出口，move(int[][] a,int i,int j)返回 true，结束递归；如果从 p 点向东无法找到出口，返回 false 结束递归，再从 p 点向南调用 move(int[][] a,int i,int j)，如果找到出口，move(int[][] a,int i,int j)返回 true，结束递归；如果从 p 点向南无法找到出口，返回 false 结束递归，再从 p 点向西调用 move(int[][] a,int i,int j)，如果找到出口，move(int[][] a,int i,int j)返回 true，结束递归；如果从 p 点向西无法找到出口，返回 false 结束递归，再从 p 点向北调用 move(int[][] a,int i,int j)，如果找到出口，move(int[][] a,int i,int j)返回 true，结束递归，否则返回 false

结束递归。

如果 move(int[][]a,int i,int j)最后返回的值是 true,说明老鼠找到出口,否则说明迷宫没有出口。

例 3-10 模拟老鼠走迷宫。

例 3-10 中的 move(int[][] a,int i,int j)是老鼠走迷宫的方法,该方法是一个递归方法。

Mouse.java

```
public class Mouse {
     public static boolean move(int[][] a,int i,int j){
           int m = a.length;
           int n = a[0].length;
           boolean isOut = false;
           if(a[i][j] == 2){                          //出口
                 isOut = true;
           }
           else if(a[i][j] == 0){
                 a[i][j] = -1;                        //标记此点已经递归过,即老鼠走过了该点
                 int t = j+1<n-1?j+1:n-1;             //东
                 boolean roadOrOut = a[i][t] == 0||a[i][t] == 2;  //是路或出口
                 if(roadOrOut&&move(a,i,t)){
                     isOut = true;
                     return isOut;
                 }
                 t = i+1<m-1?i+1:m-1;                 //南
                 roadOrOut = a[t][j] == 0||a[t][j] == 2;
                 if(roadOrOut&&move(a,t,j)){
                     isOut = true;
                     return isOut;
                 }
                 t = j-1<0?0:j-1;                     //西
                 roadOrOut = a[i][t] == 0||a[i][t] == 2;
                 if(roadOrOut&&move(a,i,t)){
                     isOut = true;
                     return isOut;
                 }
                 t = i-1<0?0:i-1;                     //北
                 roadOrOut = a[t][j] == 0||a[t][j] == 2;
                 if(roadOrOut&&move(a,t,j)){
                     isOut = true;
                     return isOut;
                 }
           }
           return isOut;
     }
}
```

不难验证,move(int[][] a,int i,int j)方法的时间复杂度是 $O(n^2)$,空间复杂度也是 $O(n^2)$。

例 3-10 中的主类 Example3_10 使用 move(int[][] a,int i,int j)方法走迷宫。其中,用 int 型二维数组模拟迷宫,二维数组的元素值是 1 表示墙,0 表示路,2 表示出口。在老鼠走过迷宫后,输出老鼠走过的路时,用☆表示老鼠走过的路,■表示墙,★表示出口,□表示老鼠未走过的路,运行效果如图 3.16 所示。

Example3_10.java

```
import java.util.Arrays;
public class Example3_10 {
```

```
迷宫数据:0表示路,1表示墙,2表示出口。
[0, 0, 0, 1, 1, 1, 1]
[1, 0, 0, 0, 0, 1, 1]
[1, 1, 0, 1, 0, 0, 1]
[1, 0, 0, 0, 1, 1, 1]
[1, 0, 0, 0, 0, 2, 1]
老鼠走迷宫过程:☆表示走过的路,■是墙,★是出口,□是未走过的路。
  ☆  ☆  ☆  ■  ■  ■  ■
  ■  □  ☆  ☆  ☆  ■  ■
  ■  ■  ☆  ■  ☆  ☆  ■
  ■  □  ☆  ☆  ■  ■  ■
  ■  □  □  ☆  ☆  ★  ■
true
```

图 3.16 老鼠走迷宫

```
    public static void main(String args[]){
        int [][] a ={{0,0,0,1,1,1,1},
                     {1,0,0,0,0,1,1},
                     {1,1,0,1,0,0,1},
                     {1,0,0,0,1,1,1},
                     {1,0,0,0,0,2,1}};
        System.out.println("迷宫数据:0 表示路,1 表示墙,2 表示出口。");
        for(int i=0;i<a.length;i++){
            System.out.println(Arrays.toString(a[i]));
        }
        boolean isOut = Mouse.move(a,0,0);
        System.out.println("老鼠走迷宫过程:☆表示走过的路," +
                           "■是墙,★是出口,□是未走过的路。");
        for(int i=0;i<a.length;i++){
            for(int j= 0;j<a[0].length;j++){
                if(a[i][j] == -1)          // -1 表示老鼠走过的路,见 Mouse 类中的算法
                    System.out.printf("%3c",'☆');
                else if(a[i][j] == 2)      //出口
                    System.out.printf("%3c",'★');
                else if(a[i][j] == 1)      //墙
                    System.out.printf("%3c",'■');
                else if(a[i][j] == 0)      //路
                    System.out.printf("%3c",'□');
            }
            System.out.println();
        }
        System.out.println(isOut);
    }
}
```

▶ 3.6.3 汉诺塔

汉诺塔(Hanoi Tower)问题是一个来源于印度的古老问题。有名字为 A、B、C 的 3 个塔，A 塔上有从小到大 64 个盘子，每次搬运一个盘子，最后要把 64 个盘子搬运到 C 塔。在搬运盘子的过程中，可以把盘子暂时放在 3 个塔中的任何一个上，但不允许大盘放在小盘的上面。3 个盘子的汉诺塔如图 3.17 所示。

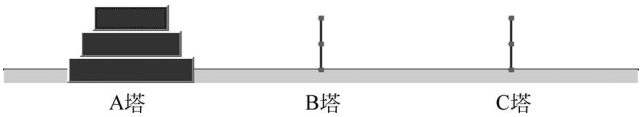

图 3.17 3 个盘子的汉诺塔

1. 汉诺塔的递归算法

汉诺塔的递归算法如下：

（1）如果 A 塔只有一个盘子，直接将盘子搬运到 C 塔。

（2）如果盘子的数目 n 大于 1，首先将 $n-1$ 个盘子从 A 塔搬运到 B 塔，然后将第 n 个盘子从 A 塔搬运到 C 塔，最后将 $n-1$ 盘子从 B 塔搬运到 C 塔。

3 个盘子的汉诺塔的搬运盘子的过程如图 3.18((a)～(g))所示。

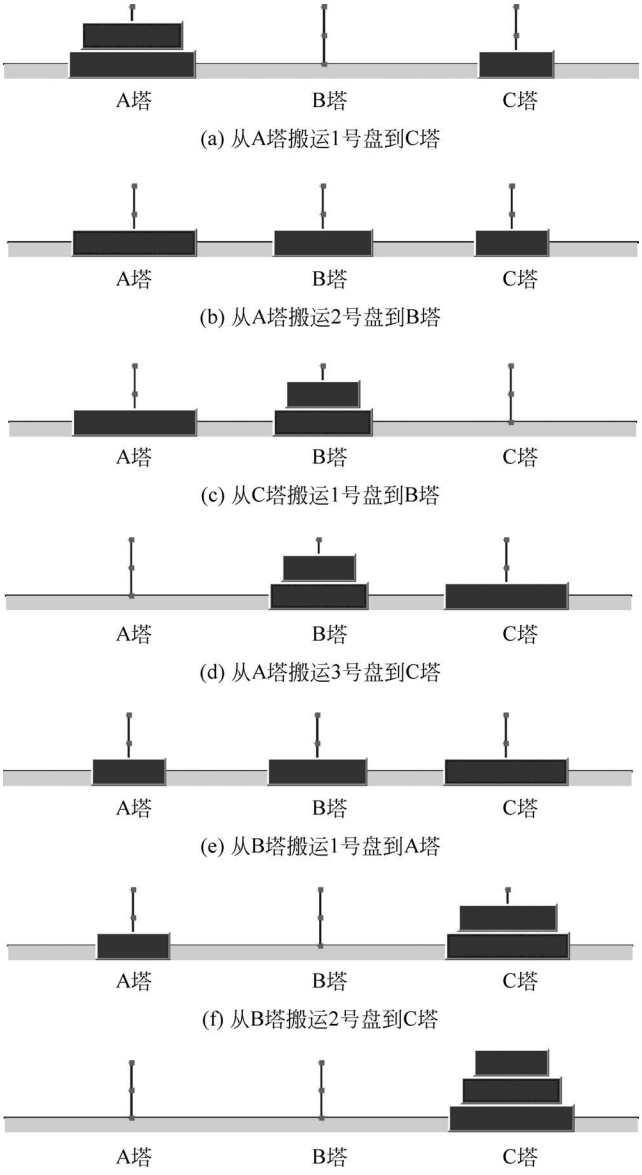

图 3.18　搬运 3 个盘子的汉诺塔

例 3-11　用递归算法实现汉诺塔。

例 3-11 中的 HanoiTower 类的 moveDish(int n char A,char B,char C)方法是搬运盘子的递归算法。

HanoiTower.java

```
public class HanoiTower {
    public static void moveDish(int n,char A,char B,char C){
        if(n == 1) {
            System.out.printf("从%c塔搬运%d号盘到%c塔\n",A,n,C);
        }
        else {
            moveDish(n-1,A,C,B);
            System.out.printf("从%c塔搬运%d号盘到%c塔\n",A,n,C);
            moveDish(n-1,B,A,C);
        }
    }
}
```

如果汉诺塔有 n 个盘子，那么需要搬动 2^n-1 次盘子，所以不难验证 moveDish(int n char A,char B,char C)方法的时间复杂度是 $O(2^n)$，空间复杂度是 $O(n)$(验证方法见例 3-2)。

例 3-11 中的主类 Example3_11 使用 HanoiTower 类的 moveDish(int n char A,char B,char C)方法搬运 3 个盘子的汉诺塔和 4 个盘子的汉诺塔，运行效果如图 3.19 所示。

```
汉诺塔有3个盘子
从A塔搬运1号盘到C塔
从A塔搬运2号盘到B塔
从C塔搬运1号盘到B塔
从A塔搬运3号盘到C塔
从B塔搬运1号盘到A塔
从B塔搬运2号盘到C塔
从A塔搬运1号盘到C塔
汉诺塔有4个盘子
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
从A塔搬运3号盘到B塔
从C塔搬运1号盘到A塔
从C塔搬运2号盘到B塔
从A塔搬运1号盘到B塔
从A塔搬运4号盘到C塔
从B塔搬运1号盘到C塔
从B塔搬运2号盘到A塔
从C塔搬运1号盘到A塔
从B塔搬运3号盘到C塔
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
```

图 3.19 用递归法搬运盘子

Example3_11.java

```
public class Example3_11 {
    public static void main(String args[]){
        int n= 3;
        HanoiTower.moveDish(n,'A','B','C');
    }
}
```

2. 汉诺塔的迭代算法

在 3.4 节讲过，递归的代码简练，容易理解解决问题的思路或发现数据的内部逻辑规律，具有很好的可读性。迭代的代码可能比较复杂，处理数据的过程也可能比较复杂，所以迭代的代码不如递归简练。

在给出汉诺塔的迭代算法之前，先总结一下汉诺塔问题中的一些规律。

（1）n 个盘子的汉诺塔需要搬运 2^n-1 次盘子。

（2）依次搬运的盘子的号码对应着 $1\sim2^{n+1}-1$ 中从小到大的偶数的二进制的尾部（低位）连续的 0 的个数。自然数的奇数的二进制的个位是 1，偶数是 0。例如盘子的数目是 3，依次搬运的盘子的号码与二进制的尾部连续的 0 的个数的对应关系如表 3.1 所示。

表 3.1　盘子的号码与二进制的尾部连续的 0 的个数的对应表

依次搬运的盘子的号码	偶数的十进制	偶数的二进制	尾部连续的 0 的个数
1	2	1**0**	1
2	4	1**00**	2
1	6	11**0**	1
3	8	1**000**	3
1	10	101**0**	1
2	12	11**00**	2
1	14	111**0**	1

根据表 3.1，在搬运盘子的过程中，搬运的盘子的号码依次是（如前面的图 3.18 所示）：

1,2,1,3,1,2,1

（3）二进制的尾部连续的 0 的个数，每隔一次这个数目就是 1。也就是说，在搬运盘子的过程中，每隔一次就要搬动一次 1 号盘（盘号最小的盘）。

（4）1 号盘的移动规律是：如果盘子的数目 n 是奇数，1 号盘找目标塔是以 CBA 的循环次序；如果 n 是偶数，1 号盘找目标塔是以 BCA 的循环次序。

（5）当搬运大号盘时（盘号大于或等于 2），上一次搬运的一定是 1 号盘（理由见（3）），所以搬运的大号盘的目标塔一定不是上一次搬运 1 号盘的目标塔（大盘不能放在小盘上面）。

注意：实际上，这些规律都是人们通过研究汉诺塔的递归算法发现的，也就是通过递归可以发现数据的内部逻辑规律。

一个偶数通过不断地右位移可计算出尾部连续的 0 的个数，例如 8 的二进制 1000 右位移 3 次得到奇数，因此知道 8 的二进制的尾部连续的 0 的个数是 3。

例 3-12　根据迭代算法的规律，给出汉诺塔的迭代算法。

HanoiTowerIterator 中的 moveDish(int n)方法是迭代算法，不难验证 moveDish(int n)方法的时间复杂度是 $O(2^n)$，空间复杂度是 $O(1)$。

尽管本例中的 moveDish(int n)的时间复杂度和例 3-11 中的递归算法相同，空间复杂度低于递归算法，但简练性和可读性远不如递归算法，在内存允许的范围内还是使用递归算法更好。

HanoiTowerIterator.java

```
import java.util.ArrayDeque;
import java.util.Stack;
public class HanoiTowerIterator {
     public static void moveDish(int n) {
          ArrayDeque<Character> deque = new ArrayDeque<Character>();
                                                        //队列,存放目标塔的名字
          Stack<Integer> A = new Stack<Integer>();      //栈,模拟 A 塔
          Stack<Integer> B = new Stack<Integer>();      //栈,模拟 B 塔
          Stack<Integer> C = new Stack<Integer>();      //栈,模拟 C 塔
          if(n%2 != 0 ){
```

```
            deque.add('C');
            deque.add('B');
            deque.add('A');
        }
        else{
            deque.add('B');
            deque.add('C');
            deque.add('A');
        }
        for(int i = n;i >= 1;i -- ){                          //初始状态,盘子都在 A 塔上
            A.push(i);
        }
        for(int i = 1;i <= (int)(Math.pow(2,n) - 1);i++){
            int dishNumber = ZeroCount.getZeroCount(2 * i);  //盘号
            if(dishNumber == 1){
                char target = deque.pop();                    //出列
                deque.add(target);                           //入列,以便循环使用队列中的数据
                if(A.contains(dishNumber)){
                    System.out.printf("从 A 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,target);
                    if(target == 'C')
                        C.push(A.pop());
                    else if(target == 'B')
                        B.push(A.pop());
                }
                else if(B.contains(dishNumber)){
                    System.out.printf("从 B 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,target);
                    if(target == 'A')
                        A.push(B.pop());
                    else if(target == 'C')
                        C.push(B.pop());
                }
                else if(C.contains(dishNumber)){
                    System.out.printf("从 C 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,target);
                    if(target == 'A')
                        A.push(C.pop());
                    else if(target == 'B')
                        B.push(C.pop());
                }
            }
            else if(dishNumber >= 2){
                char notTarget = deque.getLast();             //1 号盘刚去过的塔
                if(A.contains(dishNumber)){
                    if(notTarget == 'C'){     //目标塔只剩 B 一种可能(大盘不放在小盘之上)
                        B.push(A.pop());
                        System.out.printf("从 A 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,'B');
                    }
                    else if(notTarget == 'B'){          //目标塔只剩 C 一种可能
                        C.push(A.pop());
                        System.out.printf("从 A 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,'C');
                    }
                }
                else if(B.contains(dishNumber)){
                    if(notTarget == 'C'){
                        A.push(B.pop());
                        System.out.printf("从 B 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,'A');
                    }
                    else if(notTarget == 'A'){
                        C.push(B.pop());
                        System.out.printf("从 B 塔" + "搬运 %d 号盘到 %c 塔\n",dishNumber,'C');
                    }
```

```
                    }
                    else if(C.contains(dishNumber)){
                        if(notTarget == 'A'){
                            B.push(C.pop());
                            System.out.printf("从C塔" + "搬运%d号盘到%c塔\n",dishNumber,'B');
                        }
                        else if(notTarget == 'B'){
                            A.push(C.pop());
                            System.out.printf("从C塔" + "搬运%d号盘到%c塔\n",dishNumber,'A');
                        }
                    }
                }
            }
        }
    }
```

ZeroCount.java

```
public class ZeroCount {
    public static int getZeroCount(int n){          //返回n的二进制的尾部连续的0的个数
        int count = 0;
        if(n%2 == 0) {
            while(n%2 != 1) {
                n = n>>1;
                count++;
            }
        }
        return count;
    }
}
```

例3-12中的主类Example3_12使用HanoiTowerIterator的moveDish(int *n*)方法搬运3个盘子的汉诺塔和4个盘子的汉诺塔，运行效果如图3.20所示。

```
迭代法搬运盘子

汉诺塔有3个盘子
从A塔搬运1号盘到C塔
从A塔搬运2号盘到B塔
从C塔搬运1号盘到B塔
从A塔搬运3号盘到C塔
从B塔搬运1号盘到A塔
从B塔搬运2号盘到C塔
从A塔搬运1号盘到C塔
汉诺塔有4个盘子
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
从A塔搬运3号盘到B塔
从C塔搬运1号盘到A塔
从C塔搬运2号盘到B塔
从A塔搬运1号盘到B塔
从A塔搬运4号盘到C塔
从B塔搬运1号盘到C塔
从B塔搬运2号盘到A塔
从C塔搬运1号盘到A塔
从B塔搬运3号盘到C塔
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
```

图3.20　用迭代去搬运盘子

Example3_12.java

```
public class Example3_12 {
    public static void main(String args[]){
        System.out.println("迭代法搬运盘子\n");
        int n = 3;
        System.out.printf("汉诺塔有%d个盘子\n",n);
        HanoiTowerIterator.moveDish(n);
        n = 4;
        System.out.printf("汉诺塔有%d个盘子\n",n);
        HanoiTowerIterator .moveDish(n);
    }
}
```

3.7 优化递归

在3.2节讲解了非线性递归，即每次递归时方法调用自身两次或两次以上。非线性递归可以形成多个递归分支，即形成多个子递归过程。例如例3-2中Fibonacci类的递归算法$f(\text{long } n)$(求Fibonacci数列的第n项)形成了两个递归分支$f(n-1)$和$f(n-2)$。

为了完成$f(n)$的调用，在递归过程中需要将$f(n-1)$分支进行完毕再进行$f(n-2)$分支。注意，在进行$f(n-1)$分支递归时会完成$f(n-2)$分支递归，那么再进行$f(n-2)$分支就是一个重复的递归过程。

优化递归就是在每次递归开始之前先到某个对象中(通常为散列表对象，也可以是数组)查找本次递归是否已经实施完毕，即是否已经有了递归结果，如果散列表对象中已经有了本次递归的结果，就直接使用这个结果，不再浪费时间进行本次递归，否则就进行本次递归，并将递归结果保存到散列表对象。简而言之，优化递归是为了避免重复子递归。

优化递归是典型的用空间换取时间的策略(需要额外地存储某些递归结果)。优化递归通常不会改变空间的复杂度，但一定可以降低时间复杂度，可能将指数复杂度降低为线性复杂度或多项式复杂度。许多非线性递归的时间复杂度都是指数复杂度，例如例3-2中计算Fibonacci数列的递归算法，其时间复杂度是$O(2^n)$。

例3-13 使用OptimizeFibonacci类优化递归，使得计算Fibonacci数列的递归算法的时间复杂度是$O(n)$。

例3-2中计算Fibonacci数列的递归算法的时间复杂度是$O(2^n)$，本例OptimizeFibonacci类中的递归算法避免了重复子递归，当n的值较大时，优化递归的耗时明显小于未优化递归的耗时。两者的空间复杂度都是$O(n)$。

注意：OptimizeFibonacci类的散列表是静态成员变量，会不断累积子递归的结果，尽管会浪费内存空间，但会使后面的递归速度越来越快。

OptimizeFibonacci.java

```
import java.util.Hashtable;
public class OptimizeFibonacci {
    public static Hashtable<Long,Long> table = new Hashtable<>();
    public static long f(long n){
        long result = -1;
        if(n==1||n==2) {
            result= 1;
        }
```

```
            else if(n>=3){
                  if(table.containsKey(n)) {        //containsKey()的时间复杂度是O(n)
                        result = table.get(n);      //散列表中已经有第n次递归结果,get()的时间
                                                    //复杂度是O(1)
                  }
                  else {
                        result = f(n-1)+f(n-2);
                        table.put(n,result);        //将第n次递归结果保存到散列表中,put()的时
                                                    //间复杂度是O(1)
                  }
            }
            return result;
      }
}
```

本例中的主类 Example3_13 比较了例 3-2 中 Fibonacci 类的 f(long n)方法和本例中 OptimizeFibonacci 类的 f(long n)方法的运行耗时，当 n 的值较大时，例如 n 的值大于 35，优化后的方法的运行耗时明显小于未优化的方法的运行耗时。在使用未优化的方法时需要耐心等待一段时间，例如 n 的值为 160 时，本机测试的优化方法的耗时仅是 151 200 纳秒(1 秒＝1 000 000 000 纳秒)，而未优化的方法的耗时可能需要 100 个小时以上(本机测试时没有耐心去等待了)，运行效果如图 3.21 所示。

```
优化求第50项12586269025的用时是377600(纳秒)
请耐心等待...
未优化求第50项12586269025的用时是29420449900(纳秒)
优化快了29420072300纳秒
优化求第160项8259707399215967867的用时是151200(纳秒)
```

图 3.21　Fibonacci 的优化和未优化递归的耗时

Example3_13.java

```
public class Example3_13 {
      public static void main(String args[]) {
            long item = 50;
            long result;
            long startTime = System.nanoTime();
            result = OptimizeFibonacci.f(item);
            long estimatedTime1 = System.nanoTime()-startTime;
            System.out.printf("优化求第%d项%d的用时是%d(纳秒)\n",
                              item,result,estimatedTime1);
            System.out.println("请耐心等待...");
            startTime = System.nanoTime();
            result = Fibonacci.f(item);
            long estimatedTime2 = System.nanoTime()-startTime;
            System.out.printf("未优化求第%d项%d的用时是%d(纳秒)\n",
                              item,result,estimatedTime2);
            System.out.println("优化快了"+(estimatedTime2-estimatedTime1)+"纳秒");
            startTime = System.nanoTime();
            estimatedTime1 = System.nanoTime()-startTime;
            item = 160;
            startTime = System.nanoTime();
            result = OptimizeFibonacci.f(item);
            estimatedTime1 = System.nanoTime()-startTime;
            System.out.printf("优化求第%d项%d的用时是%d(纳秒)\n",
                              item,result,estimatedTime1);
      }
}
```

例 3-14 使用 OptimizePascalTriangle 类优化递归，使得计算杨辉三角形的递归算法 $C(\text{int } n, \text{int } j)$ 的时间复杂度是 $O(n)$。

例 3-9 中 PascalTriangle 类的递归算法 $C(\text{int } n, \text{int } j)$ 的时间复杂度是 $O(n^2)$，本例 OptimizePascalTriangle 类中的递归算法避免了重复子递归，当 n 的值较大时，优化递归的耗时明显小于未优化递归的耗时。两者的空间复杂度都是 $O(n)$。

注意：OptimizePascalTriangle 类的散列表是静态成员变量，会不断累积子递归的结果，尽管会浪费内存空间，但会使后面的递归速度越来越快。

OptimizePascalTriangle.java

```
import java.util.Hashtable;
import java.awt.Point;
public class OptimizePascalTriangle {
     public static Hashtable<Point,Long> table = new Hashtable<>();
     public static long C(int n,int j){
          long result = 0;
          if(j==0||j==n){                    //每行的第 0 列和第 n 列上的数都是 1
               result = 1;
          }
          else {
             Point p = new Point(n,j);
             if(table.containsKey(p)){
                  result = table.get(p);
             }
             else {
                result = C(n-1,j-1)+C(n-1,j);
                table.put(p,result);
             }
          }
          return result;
     }
}
```

本例中的主类 Example3_14 比较了例 3-9 中 PascalTriangle 类的 $C(\text{int } n, \text{int } j)$ 方法和本例中 OptimizePascalTriangle 类的 $C(\text{int } n, \text{int } j)$ 方法的运行耗时，当 n 的值较大时，例如求杨辉三角形的第 n 行、第 $n/2$ 列的值，若 n 大于 35，优化后的方法的运行耗时明显小于未优化的方法的运行耗时。在使用未优化的方法时需要耐心等待一段时间，例如 n 的值是 1000 时，本机测试的优化方法的耗时仅是 75 毫秒，而未优化的方法的耗时可能需要 100 个小时以上(本机测试时没有耐心去等待了)，运行效果如图 3.22 所示。

```
优化求第35行,第17列4537567650的耗时是939500(纳秒)
请耐心等待...
未优化求第35行,第17列4537567650的耗时是9837757800(纳秒)
优化快了9836818300纳秒
优化求第1000行,第500列2548782591045708352的耗时是75 (毫秒)
```

图 3.22 杨辉三角形的优化和未优化递归的耗时

Example3_14.java

```
import java.awt.Point;
public class Example3_14 {
     public static void main(String args[]){
          int n = 35,j = n/2;
          long startTime = System.nanoTime();
          long result = OptimizePascalTriangle.C(n,j);
          long estimatedTime1 = System.nanoTime()-startTime;
```

```
            System.out.printf("优化求第%d行,第%d列%d的耗时是%d(纳秒)\n",
                              n,j,result,estimatedTime1);
            System.out.println("请耐心等待...");
            startTime = System.nanoTime();
            result = PascalTriangle.C(n,j);
            long estimatedTime2 = System.nanoTime() - startTime;
            System.out.printf("未优化求第%d行,第%d列%d的耗时是%d(纳秒)\n",
                              n,j,result,estimatedTime2);
              System.out.println("优化快了" + (estimatedTime2 - estimatedTime1) + "纳秒");
            n = 1000;
            j = n/2;
            startTime = System.nanoTime();
            estimatedTime1 = System.nanoTime() - startTime;
            startTime = System.nanoTime();
            result = OptimizePascalTriangle.C(n,j);
            estimatedTime1 = System.nanoTime() - startTime;
            System.out.printf("优化求第%d行,第%d列%d的耗时是%d(毫秒)\n",
                              n+1,j,result,estimatedTime1/1000000);
             result = OptimizePascalTriangle.C(n+1,j);
            estimatedTime1 = System.nanoTime() - startTime;
            System.out.printf("未优化求第%d行,第%d列%d的耗时是%d(毫秒)\n",
                              n,j,result,estimatedTime1/1000000);
        }
    }
```

习题 3

扫一扫

习题

扫一扫

自测题