递归算法讲解

大师 L. Peter Deutsch 说过：To Iterate is Human, to Recurse, Divine.中文译为：人理解迭代，神理解递归。毋庸置疑地，递归确实是一个奇妙的思维方式。对一些简单的递归问题，我们总是惊叹于递归描述问题的能力和编写代码的简洁，但要想真正领悟递归的精髓、灵活地运用递归思想来解决问题却并不是一件容易的事情。本文剖析了递归的思想内涵，分析了递归与循环的联系与区别，给出了递归的应用场景和一些典型应用，并利用递归和非递归的方式解决了包括阶乘、斐波那契数列、汉诺塔、杨辉三角的存取、字符串回文判断、字符串全排列、二分查找、树的深度求解在内的八个经典问题。

友情提示：

若读者需要本博文相关完整代码，请移步我的Github自行获取，项目名为 SwordtoOffer，链接地址为：https://github.com/githubofrico/SwordtoOffer。

一. 引子

大师 L. Peter Deutsch 说过：To Iterate is Human, to Recurse, Divine.中文译为：人理解迭代，神理解递归。毋庸置疑地，递归确实是一个奇妙的思维方式。对一些简单的递归问题，我们总是惊叹于递归描述问题的能力和编写代码的简洁，但要想真正领悟递归的精髓、灵活地运用递归思想来解决问题却并不是一件容易的事情。在正式介绍递归之前，我们首先引用知乎用户李继刚(https://www.zhihu.com/question/20507130/answer/15551917)对递归和循环的生动解释：

递归：你打开面前这扇门，看到屋里面还有一扇门。你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门，你继续打开它。若干次之后，你打开面前的门后，发现只有一间屋子，没有门了。然后，你开始原路返回，每走回一间屋子，你数一次，走到入口的时候，你可以回答出你到底用这你把钥匙打开了几扇门。

循环：你打开面前这扇门，看到屋里面还有一扇门。你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门（若前面两扇门都一样，那么这扇门和前两扇门也一样；如果第二扇门比第一扇门小，那么这扇门也比第二扇门小，你继续打开这扇门，一直这样继续下去直到打开所有的门。但是，入口处的人始终等不到你回去告诉他答案。

上面的比喻形象地阐述了递归与循环的内涵，那么我们来思考以下几个问题：

什么是递归呢?

递归的精髓(思想)是什么?

递归和循环的区别是什么?

什么时候该用递归?

使用递归需要注意哪些问题?

递归思想解决了哪些经典的问题?

这些问题正是笔者准备在本文中详细阐述的问题。

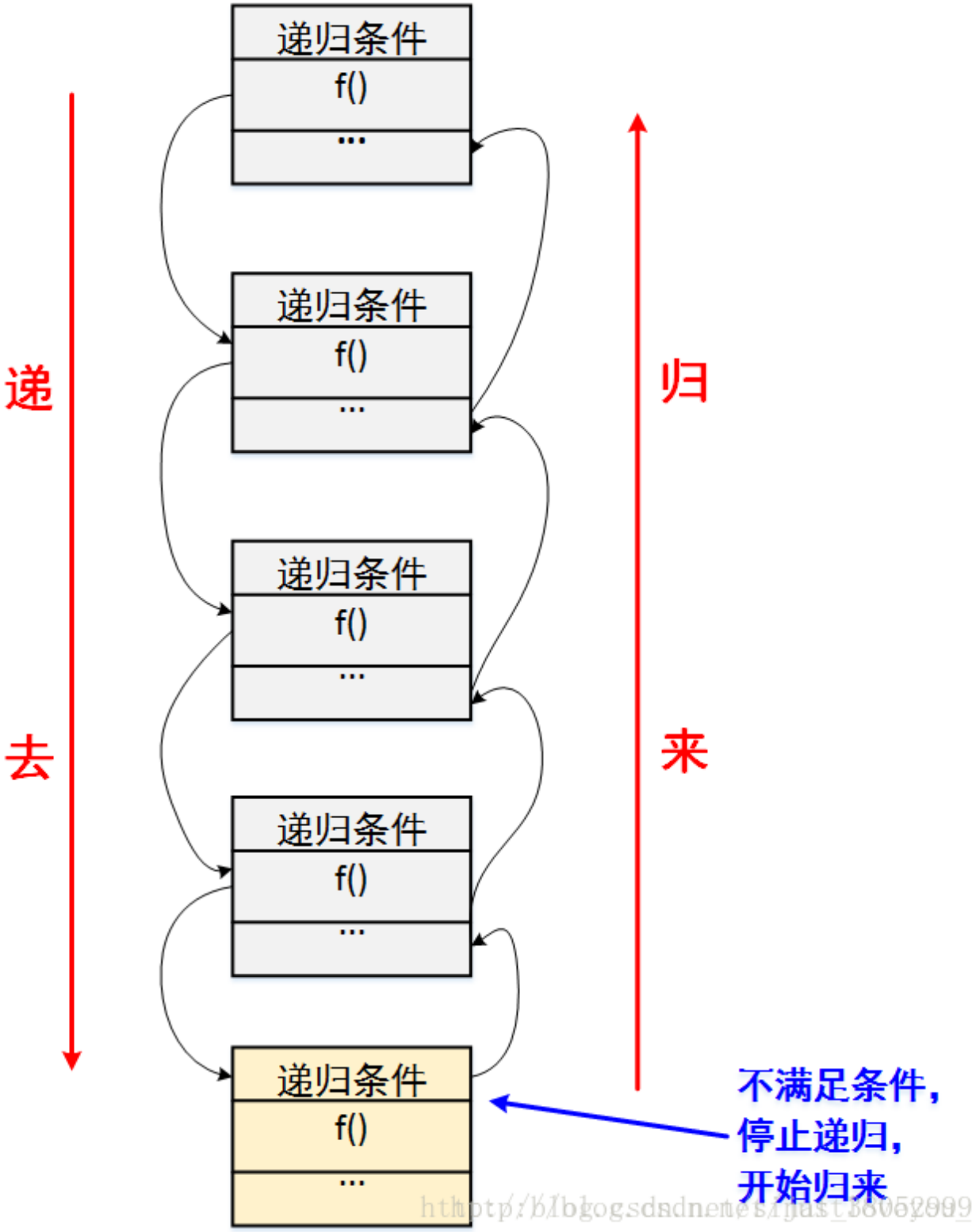
二. 递归的内涵

1、定义 (什么是递归？)

在数学与计算机科学中，递归(Recursion)是指在函数的定义中使用函数自身的方法。实际上，递归，顾名思义，其包含了两个意思：递 和 归，这正是递归思想的精华所在。

2、递归思想的内涵(递归的精髓是什么？)

正如上面所描述的场景，递归就是有去（递去）有回（归来），如下图所示。“有去”是指：递归问题必须可以分解为若干个规模较小，与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决，就像上面例子中的钥匙可以打开后面所有门上的锁一样；“有回”是指：这些问题的演化过程是一个从大到小，由近及远的过程，并且会有一个明确的终点(临界点)，一旦到达了这个临界点，就不用再往更小、更远的地方走下去。最后，从这个临界点开始，原路返回到原点，原问题解决。


递归条件
f()
...
递归条件
f()
...
递归条件
f()
...
递归条件
f()
...
递归条件
f()
...
递
去
归
来
不满足条件，
停止递归，
开始归来

更直接地说，递归的基本思想就是把规模大的问题转化为规模小的相似的子问题来解决。特别地，在函数实现时，因为解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数调用它自身的情况，这也正是递归的定义所在。格外重要的是，这个解决问题的函数必须有明确的结束条件，否则就会导致无限递归的情况。

3、用归纳法来理解递归

数学都不差的我们，第一反应就是递归在数学上的模型是什么，毕竟我们对于问题进行数学建模比起代码建模拿手多了。观察递归，我们会发现，递归的数学模型其实就是 数学归纳法，这个在高中的数列里面是最常用的了，下面回忆一下数学归纳法。

数学归纳法适用于将解决的原问题转化为解决它的子问题，而它的子问题又变成子问题的子问题，而且我们发现这些问题其实都是一个模型，也就是说存在相同的逻辑归纳处理项。当然有一个是例外的，也就是归纳结束的那一个处理方法不适用于我们的归纳处理项，当然也不能适用，否则我们就无穷归纳了。总的来说，归纳法主要包含以下三个关键要素：

步进表达式：问题蜕变成子问题的表达式

结束条件：什么时候可以不再使用步进表达式

直接求解表达式：在结束条件下能够直接计算返回值的表达式

事实上，这也正是某些数学中的数列问题在利用编程的方式去解决时可以使用递归的原因，比如著名的斐波那契数列问题。

4、递归的三要素

在我们了解了递归的基本思想及其数学模型之后，我们如何才能写出一个漂亮的递归程序呢？笔者认为主要是把握好如下三个方面：

1、明确递归终止条件；

2、给出递归终止时的处理办法；

3、提取重复的逻辑，缩小问题规模。123451). 明确递归终止条件

我们知道，递归就是有去有回，既然这样，那么必然应该有一个明确的临界点，程序一旦到达了这个临界点，就不用继续往下递去而是开始实实在在的归来。换句话说，该临界点就是一种简单情境，可以防止无限递归。

2). 给出递归终止时的处理办法

我们刚刚说到，在递归的临界点存在一种简单情境，在这种简单情境下，我们应该直接给出问题的解决方案。一般地，在这种情境下，问题的解决方案是直观的、容易的。

3). 提取重复的逻辑，缩小问题规模*

我们在阐述递归思想内涵时谈到，递归问题必须可以分解为若干个规模较小、与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决。从程序实现的角度而言，我们需要抽象出一个干净利落的重复的逻辑，以便使用相同的方式解决子问题。

5、递归算法的编程模型

在我们明确递归算法设计三要素后，接下来就需要着手开始编写具体的算法了。在编写算法时，不失一般性，我们给出两种典型的递归算法设计模型，如下所示。

模型一： 在递去的过程中解决问题

```
functionrecursion(大规模){if (end_condition){      // 明确的递归终止条件end;  // 简单情景
   }else{           // 在将问题转换为子问题的每一步，解决该步中剩余部分的问题
      solve;              // 递去
      recursion(小规模);    // 递到最深处后，不断地归来
   }
}12345678模型二： 在归来的过程中解决问题
functionrecursion(大规模){if (end_condition){      // 明确的递归终止条件end;  // 简单情景
   }else{           // 先将问题全部描述展开，再由尽头“返回”依次解决每步中剩余部分的问题
      recursion(小规模);    // 递去
      solve;              // 归来
   }
}123456786、递归的应用场景
```

在我们实际学习工作中，递归算法一般用于解决三类问题：

(1). 问题的定义是按递归定义的（Fibonacci函数，阶乘，…）；

(2). 问题的解法是递归的（有些问题只能使用递归方法来解决，例如，汉诺塔问题，…）；

(3). 数据结构是递归的（链表、树等的操作，包括树的遍历，树的深度，…）。

在下文我们将给出递归算法的一些经典应用案例，这些案例基本都属于第三种类型问题的范畴。

三. 递归与循环

递归与循环是两种不同的解决问题的典型思路。递归通常很直白地描述了一个问题的求解过程，因此也是最容易被想到解决方式。循环其实和递归具有相同的特性，即做重复任务，但有时使用循环的算法并不会那么清晰地描述解决问题步骤。单从算法设计上看，递归和循环并无优劣之别。然而，在实际开发中，因为函数调用的开销，递归常常会带来性能问题，特别是在求解规模不确定的情况下；而循环因为没有函数调用开销，所以效率会比递归高。递归求解方式和循环求解方式往往可以互换，也就是说，如果用到递归的地方可以很方便使用循环替换，而不影响程序的阅读，那么替换成循环往往是好的。问题的递归实现转换成非递归实现一般需要两步工作：

(1). 自己建立“堆栈(一些局部变量)”来保存这些内容以便代替系统栈，比如树的三种非递归遍历方式；

(2). 把对递归的调用转变为对循环处理。

特别地，在下文中我们将给出递归算法的一些经典应用案例，对于这些案例的实现，我们一般会给出递归和非递归两种解决方案，以便读者体会。

四. 经典递归问题实战

第一类问题：问题的定义是按递归定义的(1). 阶乘

```
/**
 * Title: 阶乘的实现
```

```
 * Description:
 *     递归解法
 *     非递归解法
 * @author rico
 */publicclassFactorial {/**
     * @description 阶乘的递归实现
     * @author rico
     * @created 2017年5月10日 下午8:45:48
     * @param n
     * @return
     */publicstaticlongf(int n){
        if(n == 1)   // 递归终止条件 return1;    // 简单情景return n*f(n-1);  // 相同重复逻辑，缩小问题的规模
    }

--------------------------------我是分割线----------------------------------

    /**
     * @description 阶乘的非递归实现
     * @author rico
     * @created 2017年5月10日 下午8:46:43
     * @param n
     * @return
     */publicstaticlongf_loop(int n) {
        long result = n;
        while (n > 1) {
            n--;
            result = result * n;
        }
        return result;
    }
}12345678910111213141516171819202122232425262728293031323334353637383940(2). 斐波纳契数列
/**
* Title: 斐波纳契数列
*
```

```
* Description: 斐波纳契数列，又称黄金分割数列，指的是这样一个数列：1、1、2、3、5、8、13、21、……
* 在数学上，斐波纳契数列以如下被以递归的方法定义：F0=0，F1=1，Fn=F(n-1)+F(n-2)（n>=2，n∈N*）。
*
* 两种递归解法：经典解法和优化解法
* 两种非递归解法：递推法和数组法
*
 * @author rico
 */
publicclassFibonacciSequence {/**
    * @description 经典递归法求解
    *
    * 斐波那契数列如下:
    *
    *  1,1,2,3,5,8,13,21,34,...
    *
    * *那么，计算fib(5)时，需要计算1次fib(4),2次fib(3),3次fib(2)，调用了2次fib(1)*，即:
    *
    *  fib(5) = fib(4) + fib(3)
    *
    *  fib(4) = fib(3) + fib(2) ； fib(3) = fib(2) + fib(1)
    *
    *  fib(3) = fib(2) + fib(1)
    *
    * 这里面包含了许多重复计算，而实际上我们只需计算fib(4)、fib(3)、fib(2)和fib(1)各一次即可，
    * 后面的optimizeFibonacci函数进行了优化，使时间复杂度降到了O(n).
    *
    * @author rico
    * @created 2017年5月10日 下午12:00:42
    * @param n
    * @return
    */publicstaticintfibonacci(int n) {
      if (n == 1 || n == 2) {     // 递归终止条件return1;       // 简单情景
      }
      return fibonacci(n - 1) + fibonacci(n - 2); // 相同重复逻辑，缩小问题的规模
```

}12345678910111213141516171819202122324252627282930313233————————————我是分割线————————————-

```
/**
 * @description 对经典递归法的优化
 *
 * 斐波那契数列如下：
 *
 *  1,1,2,3,5,8,13,21,34,...
 *
 * 那么，我们可以这样看：fib(1,1,5) = fib(1,2,4) = fib(2,3,3) = 5
 *
 * 也就是说，以1,1开头的斐波那契数列的第五项正是以1,2开头的斐波那契数列的第四项，
 * 而以1,2开头的斐波那契数列的第四项也正是以2,3开头的斐波那契数列的第三项，
 * 更直接地，我们就可以一步到位：fib(2,3,3) = 2 + 3 = 5,计算结束。
 *
 * 注意，前两个参数是数列的开头两项，第三个参数是我们想求的以前两个参数开头的数列的第几项。
 *
12345678910111213141516    * 时间复杂度：O(n)
    *
    * @author rico
    * @param first 数列的第一项
    * @param second 数列的第二项
    * @param n 目标项
    * @return
    */
   publicstaticintoptimizeFibonacci(int first, int second, int n) {
       if (n > 0) {
           if(n == 1){    // 递归终止条件return first;       // 简单情景
           }elseif(n == 2){            // 递归终止条件return second;      // 简单情景
           }elseif (n == 3) {         // 递归终止条件return first + second;      // 简单情景
           }
           return optimizeFibonacci(second, first + second, n - 1);  // 相同重复逻辑，缩小问题规模
       }
       return -1;
   }
```

```
--------------------------------我是分割线----------------------------------

    /**
     * @description 非递归解法：有去无回
     * @author rico
     * @created 2017年5月10日 下午12:03:04
     * @param n
     * @return
     */publicstaticintfibonacci_loop(int n) {

        if (n == 1 || n == 2) {
            return1;
        }

        int result = -1;
        int first = 1;      // 自己维护的"栈",以便状态回溯int second = 1;     // 自己维护的"栈",以便状态回溯
for (int i = 3; i <= n; i++) { // 循环
            result = first + second;
            first = second;
            second = result;
        }
        return result;
    }

--------------------------------我是分割线----------------------------------

/**
     * @description 使用数组存储斐波那契数列
     * @author rico
     * @param n
     * @return
     */publicstaticintfibonacci_array(int n) {
        if (n > 0) {
            int[] arr = newint[n];   // 使用临时数组存储斐波纳契数列
            arr[0] = arr[1] = 1;

            for (int i = 2; i < n; i++) {   // 为临时数组赋值
```

```
            arr[i] = arr[i-1] + arr[i-2];
        }
        return arr[n - 1];
    }
    return -1;
  }
}123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869703. 杨辉三角的取值
/**
 * @description 递归获取杨辉三角指定行、列(从0开始)的值
 *              注意：与是否创建杨辉三角无关
123    * @author rico
    * @x  指定行
    * @y  指定列
    */
 /**
   * Title: 杨辉三角形又称Pascal三角形，它的第i+1行是(a+b)i的展开式的系数。
   * 它的一个重要性质是：三角形中的每个数字等于它两肩上的数字相加。
   *
   * 例如，下面给出了杨辉三角形的前4行：
   *    1
   *   1 1
   *  1 2 1
   * 1 3 3 1
   * @description 递归获取杨辉三角指定行、列(从0开始)的值
   *              注意：与是否创建杨辉三角无关
   * @author rico
   * @x  指定行
   * @y  指定列
   */publicstaticintgetValue(int x, int y) {
      if(y <= x && y >= 0){
         if(y == 0 || x == y){   // 递归终止条件return1;
         }else{
            // 递归调用，缩小问题的规模return getValue(x-1, y-1) + getValue(x-1, y);
         }
      }
      return -1;
```

```
    }
}12345678910111213141516171819202122324252627282930314. 回文字符串的判断
/**
* Title: 回文字符串的判断
* Description: 回文字符串就是正读倒读都一样的字符串。如”98789”, “abccba”都是回文字符串
*
* 两种解法:
* 递归判断;
* 循环判断;
*
 * @author rico
 */
publicclassPalindromeString {/**
     * @description 递归判断一个字符串是否是回文字符串
     * @author rico
     * @created 2017年5月10日 下午5:45:50
     * @param s
     * @return
     */publicstaticbooleanisPalindromeString_recursive(String s){
        int start = 0;
        int end = s.length()-1;
        if(end > start){   // 递归终止条件:两个指针相向移动，当start超过end时，完成判断
if(s.charAt(start) != s.charAt(end)){
                returnfalse;
            }else{
                // 递归调用，缩小问题的规模
return isPalindromeString_recursive(s.substring(start+1).substring(0, end-1));
            }
        }
        returntrue;
    }

--------------------------------我是分割线----------------------------------

    /**
     * @description 循环判断回文字符串
     * @author rico
```

```
     * @param s
     * @return
     */publicstaticbooleanisPalindromeString_loop(String s){
        char[] str = s.toCharArray();
        int start = 0;
        int end = str.length-1;
        while(end > start){  // 循环终止条件:两个指针相向移动，当start超过end时，完成判断
if(str[end] != str[start]){
                returnfalse;
            }else{
                end --;
                start ++;
            }
        }
        returntrue;
    }
}123456789101112131415161718192021222324252627282930313233343536373839404142434445464748(5). 字符串全排列
递归解法
/**
* @description 从字符串数组中每次选取一个元素，作为结果中的第一个元素;然后，对剩余的元素全排列
     * @author rico
     * @param s
     *            字符数组
     * @param from
     *            起始下标
     * @param to
     *            终止下标
     */
    publicstaticvoidgetStringPermutations3(char[] s, int from, int to) {
        if (s != null && to >= from && to < s.length && from >= 0) { // 边界条件检查if (from == to) { // 递归终止条件
                System.out.println(s); // 打印结果
            } else {
                for (int i = from; i <= to; i++) {
                    swap(s, i, from); // 交换前缀,作为结果中的第一个元素，然后对剩余的元素全排列
```

```
                getStringPermutations3(s, from + 1, to); // 递归调用，缩小问题的规模
                swap(s, from, i); // 换回前缀，复原字符数组
            }
        }
    }
}

/**
 * @description 对字符数组中的制定字符进行交换
 * @author rico
 * @param s
 * @param from
 * @param to
 */publicstaticvoidswap(char[] s, int from, int to) {
    char temp = s[from];
    s[from] = s[to];
    s[to] = temp;
}12345678910111213141516171819202122324252627282930313233 34非递归解法(字典序全排列)
/**
* Title: 字符串全排列非递归算法(字典序全排列)
* Description: 字典序全排列，其基本思想是:
* 先对需要求排列的字符串进行字典排序，即得到全排列中最小的排列.
* 然后,找到一个比它大的最小的全排列，一直重复这一步直到找到最大值,即字典排序的逆序列.
*
* 不需要关心字符串长度
*
* @author rico
*/
public class StringPermutationsLoop {undefined
/**
 * @description 字典序全排列
 *
 * 设一个字符串(字符数组)的全排列有n个，分别是A1,A2,A3,...,An
 *
 * 1. 找到最小的排列 Ai
 * 2. 找到一个比Ai大的最小的后继排列Ai+1
```

```
 * 3. 重复上一步直到没有这样的后继
 *
 * 重点就是如何找到一个排列的直接后继:
 * 对于字符串(字符数组)a0a1a2……an,
 * 1. 从an到a0寻找第一次出现的升序排列的两个字符(即ai < ai+1),那么ai+1是一个极值，因为ai+1之后的字符为降序排列，记 top=i+1;
 * 2. 从top处(包括top)开始查找比ai大的最小的值aj，记 minMax = j;
 * 3. 交换minMax处和top-1处的字符;
 * 4. 翻转top之后的字符(包括top)，即得到一个排列的直接后继排列
 *
12345678910111213141516  * @author rico
     * @param s
     *            字符数组
     * @param from
     *            起始下标
     * @param to
     *            终止下标
     */
    publicstaticvoidgetStringPermutations4(char[] s, int from, int to) {

        Arrays.sort(s,from,to+1);  // 对字符数组的所有元素进行升序排列，即得到最小排列
        System.out.println(s);

        char[] descendArr = getMaxPermutation(s, from, to); // 得到最大排列,即最小排列的逆序列
while (!Arrays.equals(s, descendArr)) {  // 循环终止条件：迭代至最大排列
if (s != null && to >= from && to < s.length && from >= 0) { // 边界条件检查
int top = getExtremum(s, from, to); // 找到序列的极值int minMax = getMinMax(s, top, to);  // 从top处(包括top)查找比s[top-1]大的最小值所在的位置
                swap(s, top - 1, minMax);  // 交换minMax处和top-1处的字符
                s = reverse(s, top, to);   // 翻转top之后的字符
                System.out.println(s);
            }
        }
    }

    /**
     * @description 对字符数组中的制定字符进行交换
```

```
 * @author rico
 * @param s
 * @param from
 * @param to
 */publicstaticvoidswap(char[] s, int from, int to) {
    char temp = s[from];
    s[from] = s[to];
    s[to] = temp;
}

/**
 * @description 获取序列的极值
 * @author rico
 * @param s 序列
 * @param from 起始下标
 * @param to 终止下标
 * @return
 */publicstaticintgetExtremum(char[] s, int from, int to) {
    int index = 0;
    for (int i = to; i > from; i--) {
        if (s[i] > s[i - 1]) {
            index = i;
            break;
        }
    }
    return index;
}

/**
 * @description 从top处查找比s[top-1]大的最小值所在的位置
 * @author rico
 * @created 2017年5月10日 上午9:21:13
 * @param s
 * @param top 极大值所在位置
 * @param to
 * @return
 */publicstaticintgetMinMax(char[] s, int top, int to) {
```

```
    int index = top;
    char base = s[top-1];
    char temp = s[top];
    for (int i = top + 1; i <= to; i++) {
        if (s[i] > base && s[i] < temp) {
            temp = s[i];
            index = i;
        }
        continue;
    }
    return index;
}

/**
 * @description 翻转top(包括top)后的序列
 * @author rico
 * @param s
 * @param from
 * @param to
 * @return
 */publicstaticchar[] reverse(char[] s, int top, int to) {
    char temp;
    while(top < to){
        temp = s[top];
        s[top] = s[to];
        s[to] = temp;
        top ++;
        to --;
    }
    return s;
}

/**
 * @description 根据最小排列得到最大排列
 * @author rico
 * @param s 最小排列
 * @param from 起始下标
```

```
    * @param to 终止下标
    * @return
    */publicstaticchar[] getMaxPermutation(char[] s, int from, int to) {
        //将最小排列复制到一个新的数组中char[] dsc = Arrays.copyOfRange(s, 0, s.length);
        int first = from;
        int end = to;
        while(end > first){  // 循环终止条件char temp = dsc[first];
            dsc[first] = dsc[end];
            dsc[end] = temp;
            first ++;
            end --;
        }
        return dsc;
    }12345678910111213141516171819202122324252627282930313233343536373839404142434
44546474849505152535455565758596061626364656667686970717273747576777879808182838
48586878889909192939495969798991001011021031041051061071081091101111121131141151161171181191201211221236. 二分查找
/**
* @description 二分查找的递归实现
* @author rico
* @param array 目标数组
* @param low 左边界
* @param high 右边界
* @param target 目标值
* @return 目标值所在位置
*/
publicstaticintbinarySearch(int[] array, int low, int high, int target) {

        //递归终止条件if(low <= high){
            int mid = (low + high) >> 1;
            if(array[mid] == target){
                return mid + 1;  // 返回目标值的位置，从1开始
            }elseif(array[mid] > target){
                // 由于array[mid]不是目标值，因此再次递归搜索时，可以将其排除
return binarySearch(array, low, mid-1, target);
            }else{
```

```
            // 由于array[mid]不是目标值，因此再次递归搜索时，可以将其排除
return binarySearch(array, mid+1, high, target);
        }
    }
    return -1;  //表示没有搜索到
}
```

----------我是分割线----------

```
/**
 * @description 二分查找的非递归实现
 * @author rico
 * @param array 目标数组
 * @param low 左边界
 * @param high 右边界
 * @param target 目标值
 * @return 目标值所在位置
 */publicstaticintbinarySearchNoRecursive(int[] array, int low, int high, int target) {

    // 循环while (low <= high) {
        int mid = (low + high) >> 1;
        if (array[mid] == target) {
            return mid + 1; // 返回目标值的位置，从1开始
        } elseif (array[mid] > target) {
            // 由于array[mid]不是目标值，因此再次递归搜索时，可以将其排除
            high = mid -1;
        } else {
            // 由于array[mid]不是目标值，因此再次递归搜索时，可以将其排除
            low = mid + 1;
        }
    }
    return -1;  //表示没有搜索到
}12345678910111213141516171819202122232425262728293031323334353637383940414243444546第二类问题：问题解法按递归算法实现(1). 汉诺塔问题
/**
* Title: 汉诺塔问题
```

* Description:古代有一个梵塔，塔内有三个座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上。
* 有一个和尚想把这64个盘子从A座移到C座，但每次只能允许移动一个盘子，并且在移动过程中，3个座上的盘子始终保持大盘在下，
* 小盘在上。在移动过程中可以利用B座。要求输入层数，运算后输出每步是如何移动的。
*
 * @author rico
 */
publicclassHanoiTower {/**
     * @description 在程序中，我们把最上面的盘子称为第一个盘子，把最下面的盘子称为第N个盘子
     * @author rico
     * @param level：盘子的个数
     * @param from 盘子的初始地址
     * @param inter 转移盘子时用于中转
     * @param to 盘子的目的地址
     */publicstaticvoidmoveDish(int level, char from, char inter, char to) {

        if (level == 1) { // 递归终止条件
            System.out.println("从" + from + " 移动盘子" + level + " 号到" + to);
        } else {
            // 递归调用：将level-1个盘子从from移到inter(不是一次性移动，每次只能移动一个盘子,其中to用于周转)
            moveDish(level - 1, from, to, inter); // 递归调用，缩小问题的规模// 将第level个盘子从A座移到C座
            System.out.println("从" + from + " 移动盘子" + level + " 号到" + to);
            // 递归调用：将level-1个盘子从inter移到to,from 用于周转
            moveDish(level - 1, inter, from, to); // 递归调用，缩小问题的规模
        }
    }

    publicstaticvoidmain(String[] args) {
        int nDisks = 30;
        moveDish(nDisks, 'A', 'B', 'C');
    }12345678910111213141516171819202122324252627282930第三类问题：数据的结构是按递归定义的(1). 二叉树深度
/**
* Title: 递归求解二叉树的深度

```
* Description:
* @author rico
* @created 2017年5月8日 下午6:34:50
*/
publicclassBinaryTreeDepth {/**
    * @description 返回二叉数的深度
    * @author rico
    * @param t
    * @return
    */publicstaticintgetTreeDepth(Tree t) {

        // 树为空if (t == null) // 递归终止条件return0;

        int left = getTreeDepth(t.left); // 递归求左子树深度，缩小问题的规模
int right = getTreeDepth(t.left); // 递归求右子树深度，缩小问题的规模
return left > right ? left + 1 : right + 1;
    }
}12345678910111213141516171819 20(2). 二叉树深度
/**
 * @description 前序遍历(递归)
 * @author rico
 * @created 2017年5月22日 下午3:06:11
 * @param root
 * @return
 */
1234567public String preOrder(Node<E> root) {
        StringBuilder sb = new StringBuilder(); // 存到递归调用栈if (root == null) {   // 递归终止条件
return"";     // ji
        }else { // 递归终止条件
            sb.append(root.data + " "); // 前序遍历当前结点
            sb.append(preOrder(root.left)); // 前序遍历左子树
            sb.append(preOrder(root.right)); // 前序遍历右子树return sb.toString();
        }
    }
```