# 实验一 递归与分治

## 实验目的：

理解递归与分治算法设计思想和方法。

## 实验课时：

4学时

## 实验原理：

一个规模为n的复杂问题的求解：可以划分成若干个规模较小<n的子问题进行求解，再将子问题的解合并成原问题的解，这便是分治的思想。

若划分成的每一个子问题都与原问题的性质相同，可用相同的求解方法；当子问题规模划分一定小时，子问题的解已知，则逆求原问题的解，这是递归的思想。

## 实验题目：

### 基本题：

1、二分查找问题

（1）设a[0:n-1]是一个已排好序的数组。请改写二分搜索算法，使得当搜索元素x不在数组中时，返回小于x的最大元素的位置i和大于x的最小元素位置j。当搜索元素在数组中时，i和j相同，均为x在数组中的位置。

import java.util.Scanner;  
  
public class FirstExercise {

private static int *N*=10;  
 private static int[] *a* = new int[*N*];

public static void main(String[] args){

for(int i = 0; i < *N*; i++){  
 *a*[i] = i\*2;  
 }

System.*out*.println("数组的存储的内容是：");

for (int j = 0; j < *N*; j++){  
 System.*out*.print(*a*[j]+"\t");  
 }  
  
 System.*out*.println();  
 System.*out*.println("请输入你要查找的数：");  
  
 Scanner input = new Scanner(System.*in*);

int check = input.nextInt();  
  
 int index = *BinarySearch*(*a*,check);  
  
 if(*a*[index] == check){  
 System.*out*.println(check + "的下标为：" + index);  
 }else {  
 System.*out*.println("小于" + check + "的最大元素下标为：" + index + "，值为：" + *a*[index]  
 + " ，大于 " + check + " 的最小元素下标为：" + (index + 1) + "，值为：" + *a*[index + 1]);  
 }  
 }  
  
 public static int BinarySearch(int[] list, int number){  
 int low = 0;  
 int high = list.length - 1;  
  
 while (low <= high){  
 int middle = (low + high) / 2;  
 if (*a*[middle] == number){  
 return middle;  
 }else if(*a*[middle] < number){  
 low = middle + 1;  
 }else {  
 high = middle - 1;  
 }  
 }

return high;  
 }  
}

实验结果如图1和图2所示，采用二分查找，若所查找的数字不在数组中，则返回下标high，否则，返回该整数的下标。将返回的下标与所查找的整数进行比较，若相等，则输出下标，否则，小于该整数的最大元素下标位置为high，大于该元素的最小下标为high+1。

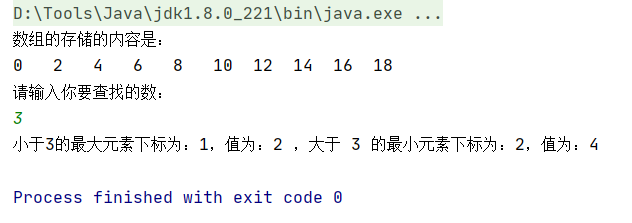


图1

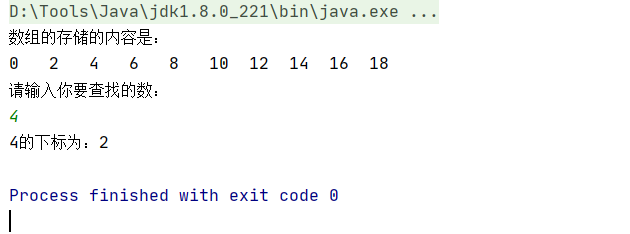


图2

（2）设有n个不同的整数排好序后存放于t[0:n-1]中，若存在一个下标i，0≤i＜n，使得t[i]=i，设计一个有效的算法找到这个下标。要求算法在最坏的情况下的计算时间为O（logn）。

public class ForthExercise {  
  
 private static int[] *a* = {-1, 1, 3, 4, 5, 6};  
  
 public static void main(String[] args){  
  
 System.*out*.println("数组的内容是：");  
  
 for(int i = 0; i < *a*.length; i++) {  
 System.*out*.print(*a*[i] + "\t");  
 }  
  
 System.*out*.println();  
  
 *findIndex*(*a*, 0, *a*.length - 1);  
 }  
  
 public static void findIndex(int[] list, int start, int end){  
 int middle = (start + end) / 2;  
  
 if (start <= end){  
 if(list[middle] == middle) {  
 System.*out*.println("下标为" + middle + "的值为 " + list[middle]);  
 return;  
 }else if (list[middle] > middle){  
 *findIndex*(list, start, middle - 1);  
 }else {  
 *findIndex*(list, middle + 1, end);  
 }  
 }  
 }  
}

实验结果如图3所示，我采用的方法是将数组元素进行二分查找，如果中间的元素的满足list[i]==i，则输出值，否则，进行比较，如果list[middle] > middle，则在数组左边进行查找，相反，则在数组右边进行查找。

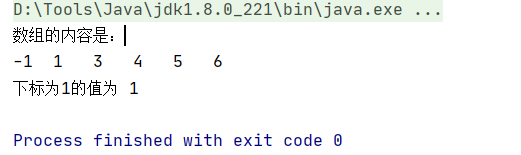


图3

2、快速排序问题

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

public class SecondExercise {  
 private static int *N* = 10;  
 private static int[] *a* = new int[*N*];

public static void main(String[] args){  
 System.*out*.println("随机生成整数:");  
 for (int i = 0; i < *a*.length; i++){  
 *a*[i] = (int) (1 + (Math.*random*() \* 10));  
 System.*out*.print(*a*[i] + "\t");  
 }  
 System.*out*.println();  
  
 *quickSort*(*a*, 0, *a*.length - 1);  
  
 System.*out*.println("排好序之后的数组为：");

for (int j = 0; j < *a*.length; j++) {  
 System.*out*.print(*a*[j] + "\t");  
 }  
 }  
  
 public static void quickSort(int[] a, int p, int r){  
 if (p < r) {  
 int q = *partition*(a, p, r);  
 *quickSort*(a, p, q - 1);  
 *quickSort*(a, q + 1, r);  
 }  
 }  
  
 public static int partition(int[] a, int p, int r){  
 int i = p;  
 int j = r + 1;  
 int x = a[p];  
  
 while (true){  
 while (a[++i] < x && i < r);  
 while (a[--j] > x);  
  
 if (i >= j){  
 break;  
 }  
  
 int temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
  
 a[p] = a[j];  
 a[j] = x;  
 return j;  
 }

}

实验结果如图4所示，随机生成10个整数，用快速排序算法进行排列。

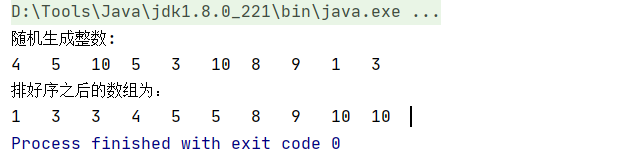


图4

3、设计一个递归算法生成n个元素的全排列

import java.util.Scanner;  
  
public class ThirdExercise {  
 static int *N*=3;  
 public static void main(String[] args){  
 int[] arr = new int[*N*];  
 System.*out*.println("请输入" + *N* + "个整数：");  
 Scanner input = new Scanner(System.*in*);  
  
 for(int i = 0; i < *N*; i++) {  
 arr[i] = input.nextInt();  
 }  
  
 *perm*(arr, 0, arr.length - 1);  
 }  
  
 public static void perm(int[] arr, int begin, int end){  
 if (begin == end){  
 for(int i = 0; i <= end; i++){  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println();  
 return;  
 }else {  
 for(int j = begin; j <= end; j++){  
 *swap*(arr, begin, j);  
 *perm*(arr, begin + 1, end);  
 *swap*(arr, begin, j);  
 }  
 }  
 }  
  
 public static void swap(int[] arr, int i, int j){  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
}

实验结果如图5所示，输入数字1,2,3并将其进行全排列。在程序中，begin记录起始位置，end记录结束位置，当begin = end时，表明，该次已经遍历完，可以输出结果，否则，将后一位数与前面的数进行交换并进行再次遍历。

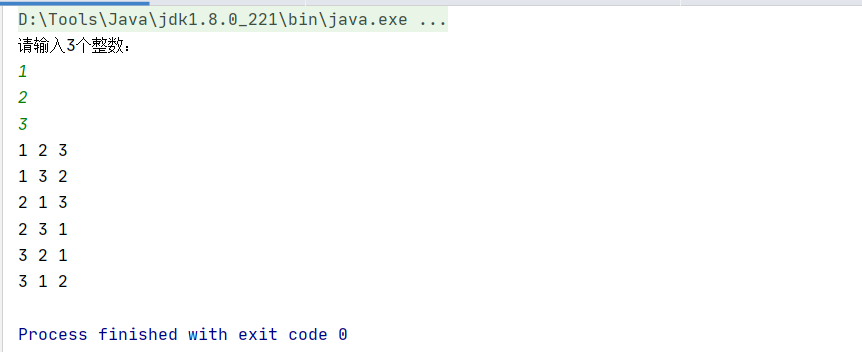
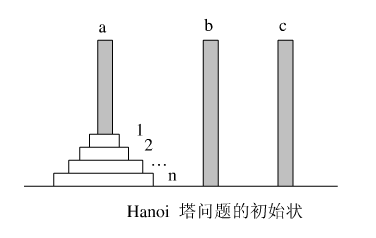


图5

### 提高题：

1、汉诺塔（hanoi）问题。

设有 A、B、 C 共 3 根塔座， 在塔座 A 上堆叠 n个金盘， 每个盘大小不同， 只允许小盘在大盘之上，最底层的盘最大，如下图 所示。现在要求将 A 上的盘全都移到 C 上，在移的过程中要遵循以下原则：每次只能移动 一个盘；圆盘可以插在 A、B 和 C 任一个塔座上；在任何时刻，大盘不能放在小盘的上面。



*//汉诺塔问题*import java.util.Scanner;  
  
public class SixthExercise {  
 public static void main(String[] args){  
 Scanner in = new Scanner(System.*in*);  
 System.*out*.println("请输入圆盘的数量");  
 int num = in.nextInt();  
 *hanoi*(num, 'A', 'B', 'C');*//起始柱、辅助柱、目标柱默认为A、B、C* }  
  
 public static void hanoi(int num, char a, char b, char c){  
 if (num == 1) {  
 System.*out*.println("第" + num + "个圆盘从" + a + " -> " + c);  
 }else{  
 *hanoi*(num - 1, a, c, b);*//借助c把第 num 个以外的圆盘从a移动到b* System.*out*.println("第" + num + "个圆盘从" + a + " -> " + c);*//把第num个从a移动到c  
 hanoi*(num - 1, b, a, c);*//借助a把第 num 个以外的圆盘从b移动到c* }  
 }  
}

实验结果如图6所示，当A柱上的圆盘过多时，采用分治思想解决该问题，我们可以借助柱C将num-1个圆盘从A柱移动到B柱，然后将第num个圆盘从A柱移动到C柱，最后，借助A柱将num-1个圆盘从B柱移动到C柱。

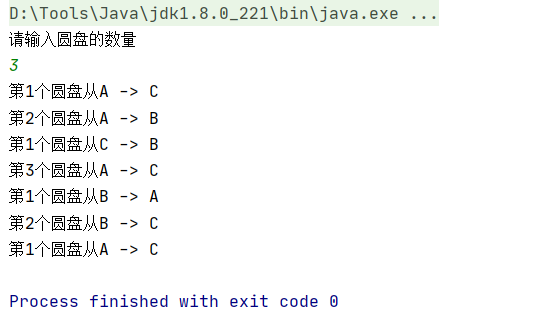


图6

2、求正整数n的不同划分个数。

import java.util.Scanner;  
  
*// 整数划分问题*public class FifthExercise {  
 public static void main(String[] args){  
 Scanner input = new Scanner(System.*in*);  
 System.*out*.println("请输入一个整数：");  
 int num = input.nextInt();  
 System.out.println("整数" + num + "的划分有" + counter(num , num) + "种");  
 }  
  
 public static int counter(int num , int max){  
 if(num == 1 || max == 1){ *//当n或者m其中任何一个为1的时候，都返回1* return 1;  
 }  
 if(num == max){ *//当n=m时，返回1+f（n，m-1）* return *counter*(num,num-1)+1;  
 }  
 if(num > max){ *//当n>m时，返回 f（n，m-1）+f（n-m，m）* return *counter*(num,max-1)+*counter*(num-max,max);  
 }  
 if(num < max){ *//当n<m时，返回f（n，n）* return *counter*(num,num);  
 }  
 else return -1;  
 }  
}

实验结果如图7所示，输入整数6时，程序进行判断，递归的终止条件是输入的数字为1或者最大加数为1的情况，这时，程序返回1；当最大加数小于输入的整数时，就有多种情况，使用*counter*(num,max-1)+*counter*(num-max,max)进行查找，得出最终结果。

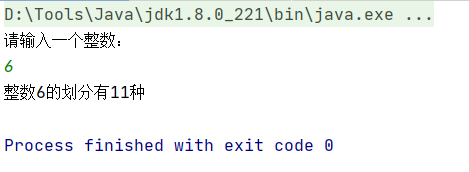


图7

3、棋盘覆盖问题

在一个2k×2k 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

*//棋盘问题*public class SeventhExercise {  
  
 int size;*//容量* int[][] board;*//棋盘* int specialRow;*//特殊点横坐标* static int *number* = 0;*//L形编号，这个一定要是静态的，可以在任何地方访问到* int specialCol;*//特殊点纵坐标* public SeventhExercise(int specialRow, int specialCol, int size) {  
 this.size = size;  
 this.specialCol = specialCol;  
 this.specialRow = specialRow;  
 board = new int[size][size];  
 }  
  
 public void setBoard(int specialRow, int specialCol, int leftRow, int leftCol, int size) {  
 *//大小为1时，结束递归* if (1 == size) {  
 return;  
 }  
  
 int subSize = size / 2;  
 *number*++;  
 int n = *number*;*//注意这里一定要吧number存在当前的递归层次里，否则进入下一层递归全局变量会发生改变  
  
 //假设特殊点在左上角区域* if (specialRow < leftRow + subSize && specialCol < leftCol + subSize) {  
 setBoard(specialRow, specialCol, leftRow, leftCol, subSize);  
 }  
 else {  
 *//不在左上角，设左上角矩阵的右下角就是特殊点（和别的一起放置L形）* board[leftRow + subSize - 1][leftCol + subSize - 1] = n;  
 setBoard(leftRow + subSize - 1, leftCol + subSize - 1, leftRow, leftCol, subSize);  
 }  
  
 *//假设特殊点在右上方* if (specialRow < leftRow + subSize && specialCol >= leftCol + subSize) {  
 setBoard(specialRow, specialCol, leftRow, leftCol + subSize, subSize);  
 }  
 else {  
 *//不在右上方，设右上方矩阵的左下角就是特殊点（和别的一起放置L形）* board[leftRow + subSize -1][leftCol + subSize] = n;  
 setBoard(leftRow + subSize -1, leftCol + subSize, leftRow, leftCol + subSize, subSize);  
 }  
  
 *//特殊点在左下方* if (specialRow >= leftRow + subSize && specialCol < leftCol + subSize) {  
 setBoard(specialRow, specialCol, leftRow + subSize, leftCol, subSize);  
 }  
 else {  
 *//不在左下方，设左下方矩阵的右上角就是特殊点（和别的一起放置L形）* board[leftRow + subSize][leftCol + subSize - 1] = n;  
 setBoard(leftRow + subSize, leftCol + subSize - 1, leftRow + subSize, leftCol, subSize);  
 }  
  
 *//特殊点在右下角* if (specialRow >= leftRow + subSize && specialCol >= leftCol + subSize) {  
 setBoard(specialRow, specialCol, leftRow + subSize, leftCol + subSize, subSize);  
 }  
 else {  
 *//不在右下角，设右下角矩阵的左上就是特殊点（和别的一起放置L形）* board[leftRow + subSize][leftCol + subSize] = n;  
 setBoard(leftRow + subSize, leftCol + subSize, leftRow + subSize, leftCol + subSize, subSize);  
 }  
 }  
  
 public static void main(String[] args){  
*//棋盘的大小* int N = 8;  
 *//特殊点的坐标* int specialRow = 0;  
 int specialCol = 1;  
 SeventhExercise chessProblem = new SeventhExercise(specialRow , specialCol , N);  
 chessProblem.printBoard(specialRow, specialCol, N);  
 }  
  
 *//输出棋盘* public void printBoard(int specialRow,int specialCol,int size) {  
 setBoard(specialRow, specialCol, 0, 0, size);  
 for (int i = 0; i < board.length; i++) {  
 for (int j = 0; j < board.length; j++) {  
 System.*out*.print(board[i][j] + "\t");  
 }  
 System.*out*.println();  
 }  
 }  
}

实验结果如图8所示，采用分治法解决棋盘问题，首先将棋盘的长和宽缩小一倍，查看特殊方框的位置，然后在分割线的位置查看是否有特殊棋盘（第一次划分将棋盘分为大小相等的四个小棋盘），如果没有，则将该位置标记为特殊方框，否则，不标记。然后，这四个小棋盘又可以归结为最大棋盘的解决方式，在小棋盘中进行划分，得到最终结果。

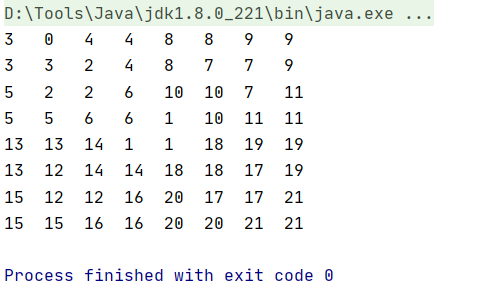


图8

## 思考问题：

1.递归的关键问题在哪里？

答：（1）递归是有截止条件的，不能无限制递归；

（2）在递归中，逻辑是相似的，我们要找到递归关系，递归即自己调用自己，问题的求解方式是一致的。

2.递归与非递归之间程序的转换？

答:在递归算法中，需判断递归的截止条件，以及问题的递归关系，但由于递归调用过程中会使用栈，容易导致栈溢出，所以，递归的规模不能太大。递归算法转换为非递归算法的方式有：（1）采用循环结构消除递归这种直接转化法没有通用的转换算法，对于具体问题要深入分析对应的递归结构，设计有效的[循环语句](https://so.csdn.net/so/search?q=%E5%BE%AA%E7%8E%AF%E8%AF%AD%E5%8F%A5&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/SYGODNICE/article/details/_blank)进行递归到非递归的转换。（2）用栈模拟系统的运行过程，通过分析只保存必须保存的信息，从而用非递归方法替代递归算法。