**华东交通大学信息工程学院**

**上机实验报告册**

**专 业 计算机科学与技术**

**班 级 2020-4**

**姓 名 惠佳乐**

**学 号 2020151002000107**

**课程名称 算法设计与分析**

**教 师 李广丽**

**学 期 2022—2023学年第一学期**

**2021年10月30日**

|  |
| --- |
| 1. **实验目的**   1．加深学生对算法设计方法的基本思想、基本步骤、基本方法的理解与掌握；  2．提高学生利用课堂所学知识解决实际问题的能力；  3．提高学生综合应用所学知识解决实际问题的能力。  **二、实验任务**  **1、串匹配问题**  给定一段文本，在该文本中查找并定位任意给定字符串。  要求：  （1）实现BF算法；   1. 实现BF算法的改进算法：KMP算法   **2、采用分治法****求解最大连续子序列和问题**  给定一个有*n*（*n*≥1）个整数的序列，要求求出其中最大连续子序列的和。  例如： 序列（-2，11，-4，13，-5，-2）的最大子序列和为20序列（-6，2，4，-7，5，3，2，-1，6，-9，10，-2）的最大子序列和为16。  **3、用分治策略求众数问题**  **问题描述：**给定含有n个元素的多重集合S，每个元素在S中出现的次数称为该元素的重数。多重集S中重数最大的元素称为众数。要求对给定的又n个自然数的组成的多重集S，计算S的众数及其重数。  **4、最近点对问题**  设*p*1=(*x*1, *y*1), *p*2=(*x*2, *y*2), …, *pn*=(*xn*, *yn*)是平面上*n*个点构成的集合*S*，设计算法找出集合*S*中距离最近的点对。  （1）分别用蛮力法和分治法求解最近对问题；  （2）分析算法的时间性能，设计实验程序验证分析结论。 |
| 1. **实验设备及编程开发工具**   **实验设备：win10**  **开发工具：idea** |
| 1. **实验过程设计**（算法思路及描述，代码设计） 2. package com.hjl;  import java.util.Scanner;  */\*\*  \** ***@author*** *惠佳乐  \*/* public class bfsuanfa {  public static void main(String[] args) {  String s1,s2;  Scanner sc=new Scanner(System.*in*);  System.*out*.println("请给定一段文本");  s1=sc.nextLine();  System.*out*.println("输入在文本中要查找的一段");  s2=sc.next();  *bf*(s1,s2);  }  public static void bf(String ts, String ps) {  char[] t = ts.toCharArray();  char[] p = ps.toCharArray();  int i = 0; // 主串的位置  int j = 0; // 模式串的位置  while (i < t.length && j < p.length) {  if (t[i] == p[j]) {  i++;  j++;  } else {  i = i - j + 1;  j = 0;  }  }  if (j == p.length) {  System.*out*.println("匹配成功");;  } else {  System.*out*.println("匹配失败");;  }  } }   package com.hjl;  import java.util.Scanner;  */\*\*  \** ***@author*** *惠佳乐  \*/* public class kmpsuanfa {  public static void main(String[] args) {  String s1,s2;  Scanner sc=new Scanner(System.*in*);  System.*out*.println("请给定一段文本");  s1=sc.nextLine();  System.*out*.println("输入在文本中要查找的一段");  s2=sc.next();  *findIndexForKMP*(s1,s2);  }  public static int findIndexForKMP(String text, String pattern){  if(text == null || pattern == null){  return -1;  }  if(pattern.length() == 0){  return 0;  }  int[] next = *getNext*(pattern);  int i = 0;  int j = 0;  char[] charsT = text.toCharArray();  char[] charsP = pattern.toCharArray();  while(j < charsP.length && i < charsT.length){  if(charsT[i] != charsP[j]){  j = next[j];  if(j == -1){  i++;  j++;  }  }else{  i++;  j++;  }  }  if(j == charsP.length){  return i - j;  }  return -1;  }  public static int[] getNext(String pattern){  // 先求出字符串的前缀表  char[] charArr = pattern.toCharArray();  int[] next = new int[charArr.length];  // 因为字符串的第一个元素没有前后缀，所以共有最大字符串长度为0  next[0] = 0;  int len = 0;  int i = 1;  while (i < charArr.length){  if(charArr[i] == charArr[len]){  len++;  next[i] = len;  i++;  }else{  if(len > 0){  len = next[len-1];  }else{  next[i] = len;  i++;  }  }  }  for (int j = next.length -1; j > 0; j--) {  next[j] = next[j-1];  }  next[0] = -1;  return next;  }   1. package com.hjl;  import java.util.Scanner;  */\*\*  \** ***@author*** *惠佳乐  \*/* public class zixulieproblem {  public static void main(String[] args) {  Scanner sc=new Scanner(System.*in*);  int n=sc.nextInt();  int arr[]=new int[n];  for(int i=0;i<n;i++) {  arr[i]=sc.nextInt();  }  System.*out*.println(*maxSubSum1*(arr));  System.*out*.println(*maxSubSum2*(arr));  }   public static int maxSubSum1(int[] a) {  int maxSum = 0;  for (int i = 0; i < a.length; i++) {  int tempSum = 0;  for (int j = i; j < a.length; j++) {  tempSum += a[j];  if (tempSum > maxSum) {  maxSum = tempSum;  }  }}  return maxSum;  }   public static int maxSubSum2(int[] a) {  return *subSum*(a, 0, a.length - 1);  }   public static int subSum(int[] a, int left, int right) {  //求a[left..high]序列中最大连续子序列和  long maxLeftSum,maxRightSum;  long maxLeftBorderSum,leftBorderSum;  long maxRightBorderSum,rightBorderSum;  if (left == right) //子序列只有一个元素时  if (a[left] > 0) //该元素大于0时返回它  return a[left];  else //该元素小于或等于0时返回0  return 0;   int mid = (left + right) / 2;  maxLeftSum = *subSum*(a, left, mid);  maxRightSum =*subSum*(a, mid+1, right);  maxLeftBorderSum = 0;leftBorderSum = 0;  for (int i= mid; i >= left; i--) { //求出以左边加上a[mid]元素  leftBorderSum += a[i]; //构成的序列的最大和  if (leftBorderSum > maxLeftBorderSum)  maxLeftBorderSum = leftBorderSum;  }   maxRightBorderSum = 0; rightBorderSum = 0;  for (int j = mid + 1; j <= right; j++) { //求出a[mid]右边元素  rightBorderSum += a[j]; //构成的序列的最大和  if (rightBorderSum > maxRightBorderSum)  maxRightBorderSum = rightBorderSum;  }   return (int) Math.*max*(Math.*max*(maxLeftSum, maxRightSum), maxLeftBorderSum + maxRightBorderSum);  } }   package com.hjl;  import java.util.Arrays;  */\*\*  \** ***@author*** *惠佳乐  \*/* public class zhongshu {  private static int *mode*;  private static int *count*;  public static int splid(int a[], int low, int high) {  if (low >= high) {  return 0;  }  int mid = (low + high) / 2;  int rlow = 0;  int lhigh = 0;  int sum = 0;  for (lhigh = mid; lhigh >= low; lhigh--) {  if (a[lhigh] != a[mid]) {  break;  }  sum++;  }   for (rlow = mid + 1; rlow <= high; rlow++) {  if (a[rlow] != a[mid]) {  break;  }  sum++;  }  if(sum>*count*){  *count*=sum;  *mode*=a[mid];  }  return *max*(sum, *splid*(a, low, lhigh), *splid*(a, rlow, high));  }   public static int max(int a, int b, int c) {  if (a >= b && a >= c) {  return a;  }  if (b >= a && b >= c) {  return b;  }  return c;  }   public static int[] arry(int length) {  int a[] = new int[length];  for (int i = 0; i < a.length; i++) {  a[i] = (int) (Math.*random*() \* 10);  }  return a;  }   public static void arryPrint(int[] a) {  System.*out*.print("数字序列为:");  for (int i = 0; i < a.length; i++) {  System.*out*.print(a[i] + " ");  }  System.*out*.println();  }   public static void main(String[] args) {  int a[] = *arry*(20);  Arrays.*sort*(a);  *arryPrint*(a);  System.*out*.println("重数为："+*splid*(a, 0, a.length - 1)+ " "+"众数为："+*mode*+" ");  } }  4.package com.hjl;  import java.util.ArrayList; import java.util.Arrays; import java.util.Comparator; import java.util.List;  */\*\*  \** ***@author*** *惠佳乐  \*/* public class zuijindian {  public static void main(String[] args)  {  // 测试用例  Point[] points = new Point[7];   points[0] = new Point(1, 1);  points[1] = new Point(1, 9);  points[2] = new Point(2, 5);  points[3] = new Point(3, 1);  points[4] = new Point(4, 4);  points[5] = new Point(5, 8);  points[6] = new Point(6, 2);   // 预处理，基于x轴坐标排序，便于分治法实施  Arrays.*sort*(points, new Comparator<Point>()  {  @Override  public int compare(Point p1, Point p2)  {  return (p1.x > p2.x) ? 1 : (p1.x == p2.x) ? 0 : -1;  }   });  // 测试  System.*out*.println(*divide*(0, points.length-1, points));  }   */\*\*  \* 求平面上距离最近的两个点  \*  \*/* public static double divide(int left, int right, Point[] points)  {  // 当前最小两点距离，初始值设置为无穷大  double curMinDis = 1e20;  // 如果只有一个点，则不存在最近两点距离，返回无穷大  if (left == right)  {  return curMinDis;  }  // 这里是判断是否为只有两个点，如果只有两个点的话那么直接求解。  if (left + 1 == right)  {  return *distance*(points[left], points[right]);  }   // 分治法：第一步：分区，并求取左右分区最小两点距离  // 通过右移运算除2，对区域进行合理的划分，使得左右两边保持大致相等个数点  int middle = (left + right) >> 1;  double leftMinDis = *divide*(left, middle, points);  double rightMinDis = *divide*(middle, right, points);   curMinDis = (leftMinDis <= rightMinDis) ? leftMinDis : rightMinDis;   // 分治法：第二步：假设距离最近的两点分别在左右分区中  // 关键代码，距离最近的两个点，一个位于左边区域，一个位于右边区域，x轴搜索范围[middle-curMinDis, middle+curMinDis]  // 记录搜索区间内的点的索引，便于进一步计算最小距离  List<Integer> validPointIndex = new ArrayList<>();  for (int i = left; i <= right; i++)  {  if (Math.*abs*(points[middle].x - points[i].x) <= curMinDis)  {  validPointIndex.add(i);  }  }  // 基于索引，进一步计算区间内最小两点距离  for (int i = 0; i < validPointIndex.size() - 1; i++)  {  for (int j = i + 1; j < validPointIndex.size(); j++)  {  // 如果区间内的两点y轴距离大于curMinDis，则没必要计算了，因为，它们的距离肯定大于curMinDis，  if (Math.*abs*(points[validPointIndex.get(i)].y  - points[validPointIndex.get(j)].y) > curMinDis)  {  continue;  }  double tempDis = *distance*(points[validPointIndex.get(i)],  points[validPointIndex.get(j)]);   curMinDis = (tempDis < curMinDis) ? tempDis : curMinDis;  }  }   return curMinDis;  }   */\*\*  \* 计算两点间的距离  \*/* public static double distance(Point p1, Point p2)  {  return Math.*sqrt*((p2.y - p1.y) \* (p2.y - p1.y) + (p2.x - p1.x) \* (p2.x - p1.x));  } } */\*\*  \* 定义点  \*  \*/* class Point {  public int x;  public int y;   Point(int x, int y)  {  this.x = x;  this.y = y;  }  }    **五、实验结果及算法复杂度分析**  0)R%3()]HG01P%]KKUW})15]@EDU]YS6VEDR50YAD2DJQE  **]`%7L%GZF14TF@AQ}E`]M_Q** |
| 1. **实验小结**（包括问题和解决方法、心得体会等）   分治，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并    分治法：优点：结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来很大方便。  缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。 |