**算法分析与复杂性理论**

**硕8058 3118105270**

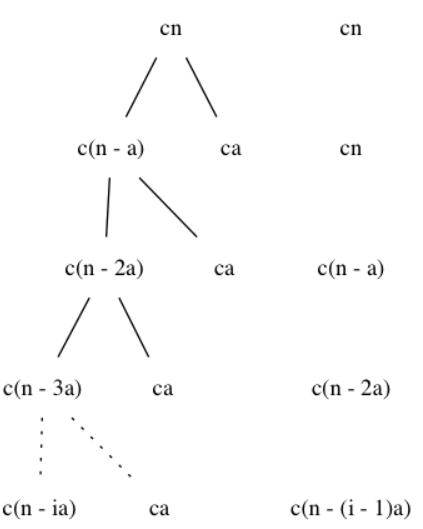
**雷鹏程 18292876073**

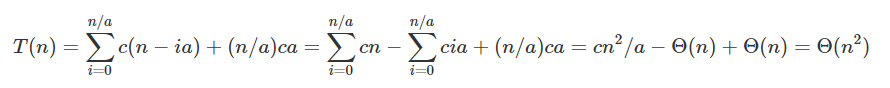
**问题1**

**问题描述**

1. 已知，(1<=i<=n-1) 其中*c*1和*c*2为正常数，请证明。

假设i=a时，为最大，得到递归树如下：





**问题2**

**问题描述**

【凸包问题】给定平面上n个点，从中找出一个最小点集，使该点集所组成的凸多边形包围所有的n个点。用分治法编写一个求解凸包问题的算法

**算法设计过程**

首先找到必在凸包集合中的点，任意画一条直线，把所有点往上投影，得到最大和最小两个点，（可能有两个点投影值一样，任取一个就行，因为都是必在凸包上的点）

Step1简单起见，直接取横坐标最大和最小的两个点，设p1，p2

Step2两个点连线，把整个点集分为两部分，子问题就是在两部分中继续找必在凸包上的点p3，可以有两种方式，一种通过斜率，一种通过计算集合中各个点到两点连线的距离，这两种方案都是可行的。

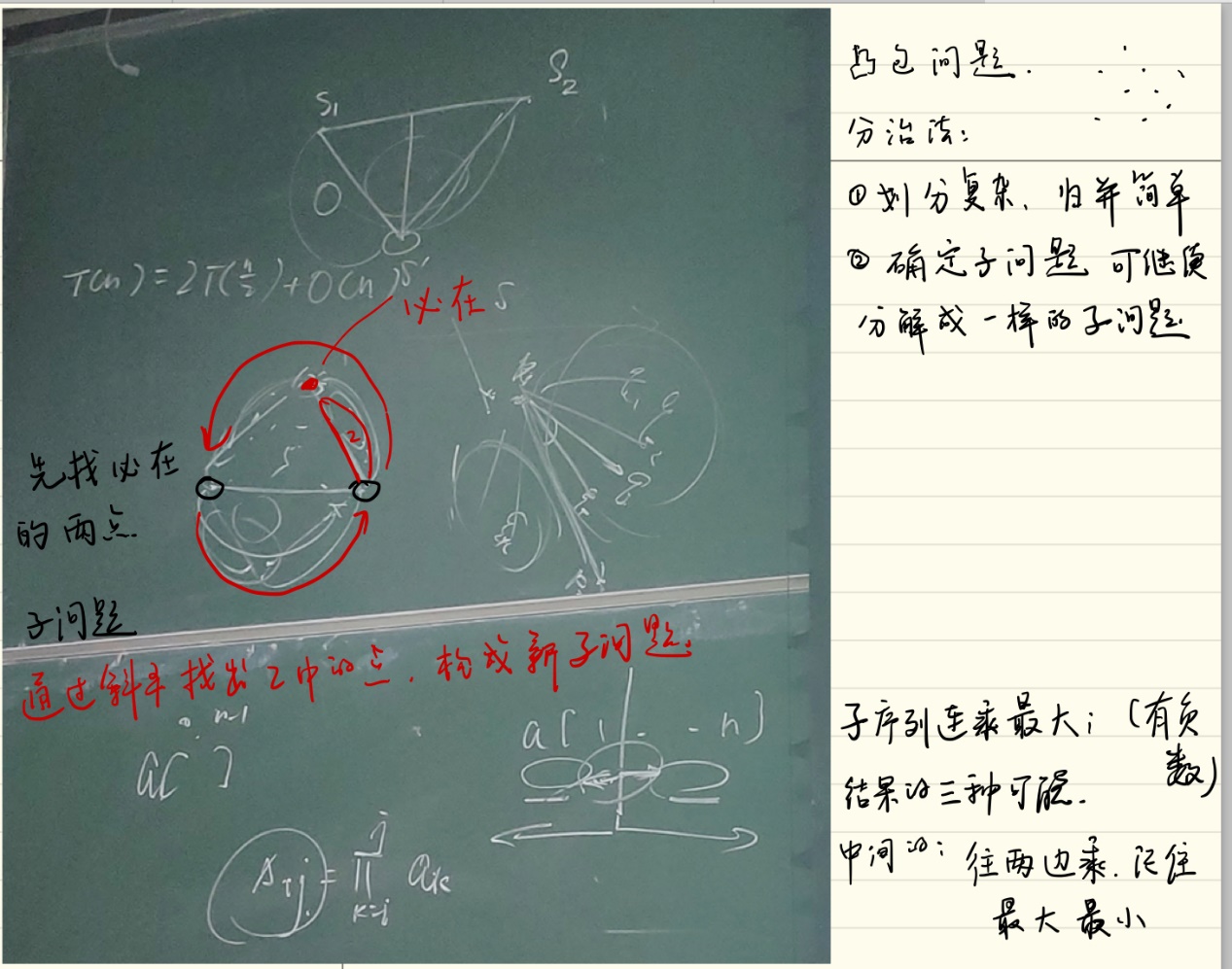
Step3 从上一步的点集中找出两个小凸包中的点集s1，s2

两个子问题为

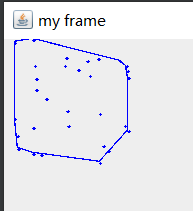
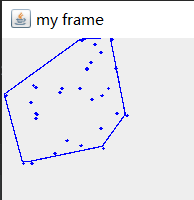
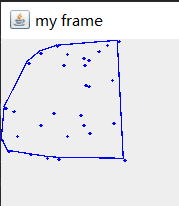
从p1，p3，点集s1中继续step2

从p3，p2，点集s2中继续step2

当划分的点集中没有点，返回这两个点，加入凸包集合中



**三组实验结果如下**



算法结果正确。

最后去查了一些凸包问题的其他常见解法，比如**Graham扫描法，其实就是采取了贪心策略**，时间复杂度为O(nlgn)，与上课时讲的贪心策略稍微不同的是，根据下一个点是不是往左转来（以逆时针为搜索方向）来判断点是不是符合凸包要求，上课时采用的是斜率

Melkman算法，能online，时间复杂度更低，能实时更新，我认为很有意思，对新点判断在不在凸包中，采用的方式很巧妙。

二维凸包的应用

围栏问题、城市规划问题、聚类分析

**源代码**

import javax.swing.\*;  
import java.awt.\*;  
import java.util.\*;  
import java.util.List;  
  
public class TuBao {  
 public static void main(String[] args) {  
 Random random = new Random();  
 double i2 = random.nextDouble();  
 double x, y;  
 Point p = null;  
 List<Point> points = new ArrayList<Point>();  
 for (int i = 0; i < 30; i++) {  
 p = new Point(random.nextInt(100), random.nextInt(100));  
 points.add(p);  
 }  
 JFrame frame = new JFrame("my frame"); //初始化一个窗口  
 frame.setSize(300, 300); // 设置窗口大小  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*); // 设置画图结束后的操作：退出画图程序  
 frame.setVisible(true); // 显示窗口  
 QuickTuBao tubao = new QuickTuBao(points);  
 List<Line> lines = new ArrayList<Line>();  
 List<Line> linest = tubao.eval();  
 for (int i=0;i<linest.size();i++){  
 lines.add(linest.get(i));  
 }  
 JPanel panel = new JPanel() { // 初始化一个新画布  
 private static final long *serialVersionUID* = 1L; // 不用管，可加可不加  
 @Override  
 public void paint(Graphics g) { //重写 pait 方法  
 super.paint(g); //这个要加上，但不加也能正常显示  
 g.setColor(Color.*BLUE*); //设置画笔颜色  
 for (int i = 0; i < points.size(); i++) {  
 g.fillOval((int) points.get(i).x, (int) points.get(i).y, 3, 3);  
 }  
 for (int i = 0; i < lines.size(); i++) {  
 g.drawLine((int) lines.get(i).p1.x, (int) lines.get(i).p1.y, (int) lines.get(i).p2.x, (int) lines.get(i).p2.y);  
 }  
 }  
 };  
 frame.setContentPane(panel); // 将画布添加到窗口中  
 }  
}  
class Line {  
 Point p1, p2;  
 Line(Point p1, Point p2) {  
 this.p1 = p1;  
 this.p2 = p2;  
 }  
 public double getLength() {  
 double dx = Math.*abs*(p1.x - p2.x);  
 double dy = Math.*abs*(p1.y - p2.y);  
 return Math.*sqrt*(dx \* dx + dy \* dy);  
 }  
}  
class Point{//点  
 double x;  
 double y;  
 public Point(double x,double y){  
 this.x=x;  
 this.y=y;  
 }  
}  
/\*  
 \* 分治法求凸包  
 \*/  
class QuickTuBao {  
 List<Point> pts = null;//给出的点集  
 List<Line> lines = new ArrayList<Line>();//点集pts的凸包  
  
 public void setPointList(List<Point> pts) {  
 this.pts = pts;  
 }  
  
 public QuickTuBao(List<Point> pts){  
 this.pts=pts;  
 }  
  
 //求凸包，结果存入lines中  
 public List<Line> eval() {  
 lines.clear();  
 if (pts == null || pts.isEmpty()) { return lines; }  
 List<Point> ptsLeft = new ArrayList<Point>();//左凸包中的点  
 List<Point> ptsRight = new ArrayList<Point>();//右凸包中的点  
  
 //按x坐标对pts排序  
 Collections.*sort*(pts, new Comparator<Point>() {  
 public int compare(Point p1, Point p2) {  
 if(p1.x-p2.x>0) return 1;  
 if(p1.x-p2.x<0) return -1;  
 return 0;  
 }  
 });  
  
 Point p1 = pts.get(0);//最左边的点  
 Point p2 = pts.get(pts.size()-1);//最右边的点,用直线p1p2将原凸包分成两个小凸包  
 Point p3 = null;  
 double area = 0;  
 for (int i = 1; i < pts.size(); i++) {//穷举所有的点,  
 p3 = pts.get(i);  
 area = getArea(p1, p2, p3);//求此三点所成三角形的有向面积  
 if (area > 0) {  
 ptsLeft.add(p3);//p3属于左  
 } else if (area < 0) {  
 ptsRight.add(p3);//p3属于右  
 }  
 }  
 d(p1, p2, ptsLeft);//分别求解  
 d(p2, p1, ptsRight);  
 return lines;  
 }  
  
 private void d(Point p1, Point p2, List<Point> s) {  
 //s集合为空  
 if (s.isEmpty()) {  
 lines.add(new Line(p1, p2));  
 return;  
 }  
 //s集合不为空，寻找Pmax  
 double area = 0;  
 double maxArea = 0;  
 Point pMax = null;  
 for (int i = 0; i < s.size(); i++) {  
 area = getArea(p1, p2, s.get(i));//最大面积对应的点就是Pmax  
 if (area > maxArea) {  
 pMax = s.get(i);  
 maxArea = area;  
 }  
 }  
 //找出位于(p1, pMax)直线左边的点集s1  
 //找出位于(pMax, p2)直线左边的点集s2  
 List<Point> s1 = new ArrayList<Point>();  
 List<Point> s2 = new ArrayList<Point>();  
 Point p3 = null;  
 for (int i = 0; i < s.size(); i++) {  
 p3 = s.get(i);  
 if (getArea(p1, pMax, p3) > 0) {  
 s1.add(p3);  
 } else if (getArea(pMax, p2, p3) > 0) {  
 s2.add(p3);  
 }  
 }  
 //递归  
 d(p1, pMax, s1);  
 d(pMax, p2, s2);  
 }  
 // 三角形的面积等于返回值绝对值的二分之一  
 // 当且仅当点p3位于直线(p1, p2)左侧时，表达式的符号为正  
 //向量积  
 private double getArea(Point p1, Point p2, Point p3) {  
 return p1.x \* p2.y + p3.x \* p1.y + p2.x \* p3.y -  
 p3.x \* p2.y - p2.x \* p1.y - p1.x \* p3.y;  
 }  
}

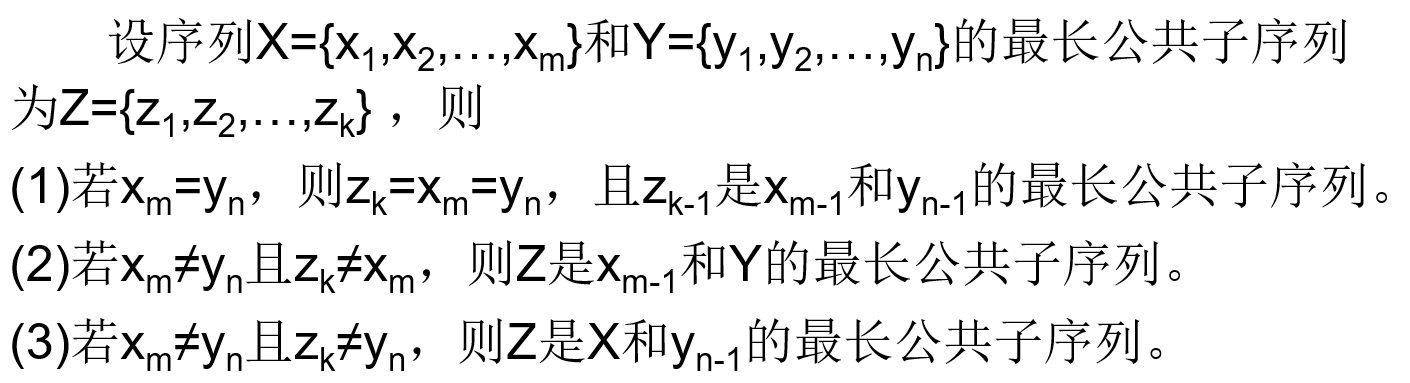
**问题3**

**问题描述**

设计一个动态规划算法，找到字符串T[1..n]中前向和后向相同的最长连续子串。前向子串和后向子串不能重叠。例如：

1. 输入“ALGORITHM”，算法返回空串；
2. 输入“RCURSION”，算法返回“R”；
3. 输入“REDIVIDE”，算法返回“EDI”。

**思路**



根据 **动态规划-最长公共子序列**的解法，把问题转化一下

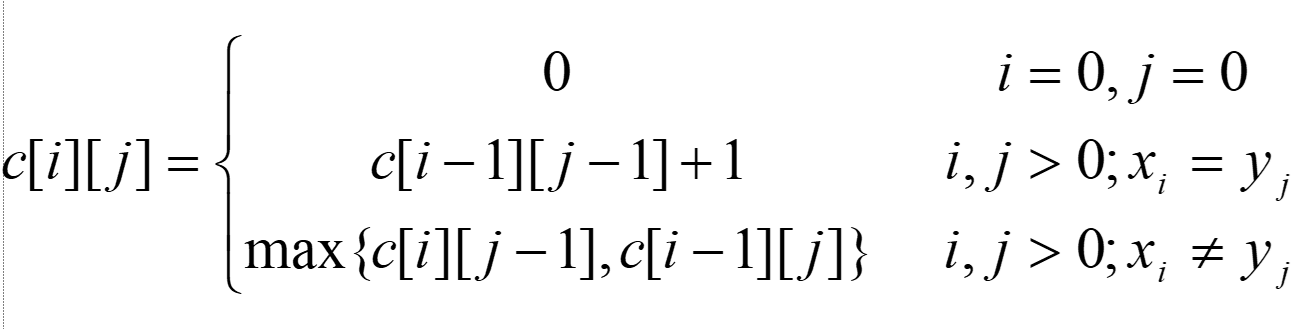
**将原字符串翻转形成新字符串，查找新字符串和原字符串中最长公共子序列，且公共子序列的索引之和小于等于原字符串序列长度，（因为不能重叠），其实问题就已经解决了。**

**最优子结构性质**：

已知最长公共子序列是符合最优子结构性质的，而本题可以转化成最长公共子序列问题的一部分来做，也是符合最优子结构性质的。

**算法设计过程**

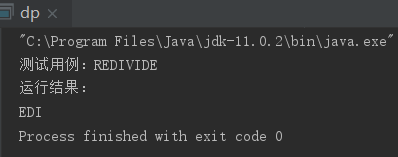
递归关系如下**：**

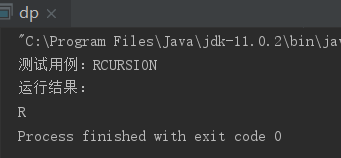


**只限制一下 i+j<=字符串长度**

**依然采用另一个二维数组来存公共子序列的索引。**

**测试用例与结果**





**源代码**

public class dp {  
  
 static void lcs(int i,int j,String x,int [][] b)  
 {  
 if (i ==0 || j==0) return;  
 if (b[i][j]== 1){  
 *lcs*(i-1,j-1,x,b);  
 System.*out*.print(x.charAt(i-1));  
 }  
 else if (b[i][j]== 2) *lcs*(i-1,j,x,b);  
 else *lcs*(i,j-1,x,b);  
 }  
  
 public static void main(String[] args) {  
  
 String x = "REDIVIDE";  
// String y = "EDIVIDER";  
// String x="RCURSION";  
 String y = new StringBuffer(x).reverse().toString();  
 int m = x.length() ;  
 int n = y.length() ;  
 int c[][] =new int[m+1][n+1];  
 int b[][]=new int[m+1][n+1];  
 for (int i =0;i<=m;i++){  
 c[i][0]=0;  
  
 }  
 for (int i =0;i<=n;i++){  
 c[0][i]=0;  
 }  
 for (int i = 1; i <=m; i++) {  
 for (int j = 1; j <= n; j++) {  
 if (j + i > x.length()) {  
 break;  
 }  
 if (x.charAt(i - 1) == y.charAt(j - 1)) {  
 c[i][j] = c[i - 1][j - 1] + 1;  
 b[i][j] = 1;  
 } else if (c[i - 1][j] >= c[i][j - 1]) {  
 c[i][j] = c[i - 1][j];  
 b[i][j] = 2;  
 } else {  
 c[i][j] = c[i][j - 1];  
 b[i][j] = 3;  
 }  
 }  
 }  
 System.*out*.println("测试用例："+x);  
 System.*out*.println("运行结果：");  
 for(int i=0;i<=m;i++){  
 for (int j = 0; j <= n; j++) {  
 if(b[i][j]==1)  
 System.*out*.print(x.charAt(i-1));  
 }  
 }  
 }  
}

**问题4**

**问题描述**

假定J={1,2,3,…,n}是n个等待在同一台机器上加工的作业集合，每个作业所需要的加工时间都为1个时间单位，且每个作业i都有一个最迟完成时间di。如果一个作业i能够在di之前完成，就可获得收益pi>0；反之，则不获得收益。请设计一个贪心的算法求该问题的一个最佳安排方案使得收益最大，并证明你所设计的贪心策略的最优性。

**算法设计过程**

**贪心算法1**

**思路：设想n个任务，对应长度为n的数组中的空位，把任务按收益排序，每次取最大收益，放置在刚好其截止时间前的一个位置，如果没有符合的空位，就跳过这个任务，计算累计收益，结束。**

**贪心算法2**

**1 将n个任务按收益从高到低排序，每次取最高收益的任务来安排**

**2 已安排任务按照截止时间排序，从小到大，贪心地来说，这样会使得能完成的任务数量最多，**

**3 如果安排某任务，会使得已安排的任务中出现超过截止时间的任务，则删除此任务**

**为什么会想到用两种贪心算法：**

**1 一开始采用暴力搜索来确定最优解任务序列，但这样当任务数量上升时间复杂度太大，所以选择另一种算法来验证贪心算法1的结果**

**2 希望比较两种贪心算法，看看采用不同贪心策略能否得到同样的结果**

**证明过程：反证法，证明贪心算法1**

**首先，产生的这个有收益任务序列，数量设为m，这m个任务是收益最大的m个任务，如果有其他有收益序列，是最优解，那么，数量一定超过了m。**

**但是从剩余任务中选任务加入m长度的序列中，就必然会导致序列中出现超时任务，因为序列中的最后一个任务必定是放在截止时间的前一个时间单位。也就是说不能再增加序列长度了。**

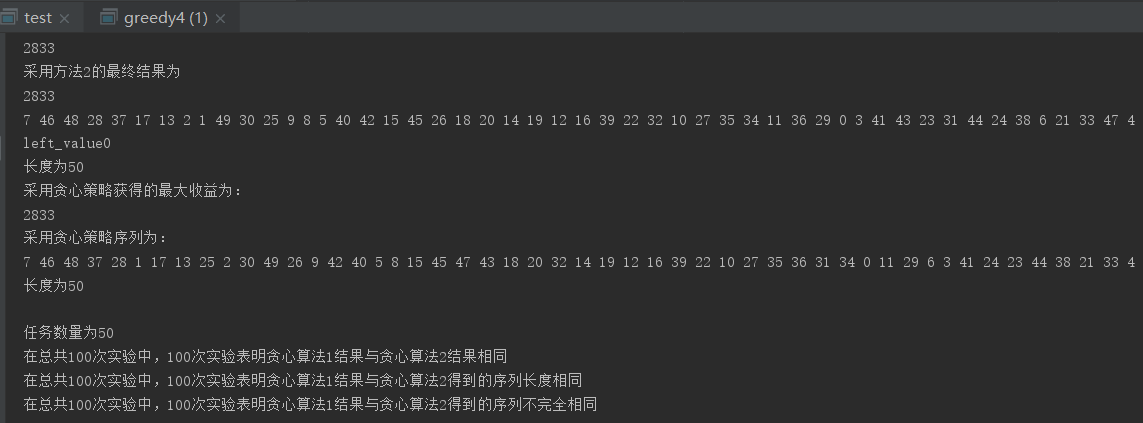
**实验结果**

**选择任务数量为50，随机生成截止时间序列和收益序列，**

**采用暴力枚举的方式，求出正确结果，然后来验证贪心法的结果，**

**重复以上，10次，得出结果，最终序列长度不定，且不为50，是因为没有打印无收益任务，无收益任务直接放最后就行。**

**两种贪心算法，产生的有收益任务安排序列长度相同，次序不同，总收益完全相同。**



**源代码**

package greedy3;  
  
import java.io.BufferedWriter;  
import java.io.File;  
import java.io.FileWriter;  
import java.util.\*;  
  
//思路：设想n个任务，对应数组中的空位，把任务按收益排序，每次取最大收益，放置在刚好ddl前的一个位置，如果没有符合的空位，就跳过这个任务，计算收益结束  
  
class Task{  
 int d;  
 int p;  
 int i;  
 public Task(int d,int p,int i){  
 this.d=d;  
 this.p=p;  
 this.i=i;  
 }  
  
}  
  
public class greedy4 {  
 public static void writeLog(String filepath,String s){  
 try{  
 FileWriter fw = new FileWriter(filepath, true);  
 BufferedWriter bw = new BufferedWriter(fw);  
 bw.append(s);  
 bw.close();  
 fw.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 }  
  
 public static void main(String[] args) {  
 int s=0;  
 int a=0;  
 int b=0;  
 int num =100;  
  
 for (int i=0;i<num;i++){  
 List<Integer> l=*f*();  
 s+=l.get(0);  
 a+=l.get(1);  
 b+=l.get(2);  
  
 }  
 System.*out*.println("任务数量为50");  
 System.*out*.println("在总共"+num+"次实验中，"+s+"次实验表明贪心算法1结果与贪心算法2结果相同");  
 System.*out*.println("在总共"+num+"次实验中，"+a+"次实验表明贪心算法1结果与贪心算法2得到的序列长度相同");  
 System.*out*.println("在总共"+num+"次实验中，"+b+"次实验表明贪心算法1结果与贪心算法2得到的序列不完全相同");  
  
 }  
  
 public static List<Integer> f(){  
 try {  
 File file = new File("a.txt");  
  
 if(file.delete()) {  
 System.*out*.println( file.getName() + " is deleted!");  
 }else {  
// System.out.println("Delete operation is failed.");  
 }  
 } catch(Exception e) {  
 e.printStackTrace();  
 }  
 Random random = new Random();  
 int a=random.nextInt(100);  
 //任务数量  
 int num =50;  
 int [] d=new int[num];  
 int [] p=new int[num];  
 Task[] tasks =new Task[num];  
  
 for(int i=0;i<num;i++){  
  
 d[i]=random.nextInt(num)+1;  
 p[i]=random.nextInt(100)+1;  
 tasks[i]=new Task(d[i],p[i],i);  
 }  
 int value\_max=0;  
 int value\_max\_index=0;  
 int seq[]=new int[num];  
 int selected []=new int [num];  
 int sum\_value =0;  
 List<Integer> order =new ArrayList<>();  
 Map<Integer,Integer> map = new HashMap<>();  
 for(int i=0;i<d.length;i++){  
 map.put(i,p[i]);  
 }  
 List<Map.Entry<Integer,Integer>> entry\_list = new ArrayList<>(map.entrySet());  
 Collections.*sort*(entry\_list, new Comparator<Map.Entry<Integer, Integer>>() {  
 @Override  
 public int compare(Map.Entry<Integer, Integer> o1, Map.Entry<Integer, Integer> o2) {  
 return o2.getValue().compareTo(o1.getValue());  
 }  
 });  
 for(Map.Entry<Integer,Integer> i : entry\_list) {  
 order.add(i.getKey());  
 }  
 int []v=new int[num];  
 for(int j=0;j<v.length;j++){  
 v[j]=-1;  
 }  
 for(Integer i :order) {  
 for (int k = num - 1; k >= 0; k--) {  
 if (v[k] == -1 && d[i] >= k + 1) {  
 v[k] = i;  
 sum\_value += p[i];  
 break;  
 }  
 }  
 }  
 System.*out*.println();  
 System.*out*.println();  
 //已安排列表  
 List<Task> arrange\_order =new ArrayList<>();  
  
 int greedy\_sum\_value2=0;  
 for(Integer i :order){  
 if(arrange\_order.isEmpty()){  
 arrange\_order.add(new Task(d[i],p[i],i));  
 greedy\_sum\_value2+=p[i];  
 }else {  
 Task temp=new Task(d[i],p[i],i);  
 arrange\_order.add(temp);  
 int fl=0;  
 Collections.*sort*(arrange\_order, new Comparator<Task>() {  
 @Override  
 public int compare(Task o1, Task o2) {  
 return o1.d-o2.d;  
 }  
 });  
 for(Task task:arrange\_order){  
 if(task.d<arrange\_order.indexOf(task)+1){  
 fl=1;  
 break;  
 }  
 }  
 if(fl==1){  
 arrange\_order.remove(temp);  
 }else {  
 greedy\_sum\_value2+=p[i];  
 }  
 }  
 }  
 //把算法2中未排列的任务，加上，并计算收益  
 int pre =arrange\_order.size();  
 List<Task> unarrange\_task =new ArrayList<>();  
 List<Integer> arrange\_task =new ArrayList<>();  
 for(Task t:arrange\_order){  
 arrange\_task.add(t.i);  
 }  
 for(int i=0;i<num;i++){  
 if(!arrange\_task.contains(i)){  
 unarrange\_task.add(new Task(d[i],p[i],i));  
 }  
 }  
 Collections.*sort*(unarrange\_task, new Comparator<Task>() {  
 @Override  
 public int compare(Task o1, Task o2) {  
 return o1.d-o2.d;  
 }  
 });  
 int left\_value=0;  
 for(int i=0;i<unarrange\_task.size();i++){  
 if(unarrange\_task.get(i).d>=pre+i+1){  
 left\_value+=unarrange\_task.get(i).p;  
 }  
 }  
  
 System.*out*.println("采用方法2的结果为");  
 System.*out*.println(greedy\_sum\_value2);  
 System.*out*.println("采用方法2的最终结果为");  
 System.*out*.println(greedy\_sum\_value2+left\_value);  
 for(int i=0;i<arrange\_order.size();i++){  
 System.*out*.print(arrange\_order.get(i).i+" ");  
 }  
 System.*out*.println();  
 System.*out*.println("left\_value"+left\_value);  
 System.*out*.println("长度为"+arrange\_order.size());  
// System.out.println();  
 System.*out*.println("采用贪心策略获得的最大收益为：");  
 System.*out*.println(sum\_value);  
 System.*out*.println("采用贪心策略序列为：");  
 int v\_length=0;  
 for(int i=0;i<v.length;i++){  
  
 if(v[i]!=-1){  
 System.*out*.print(v[i]+" ");  
 v\_length+=1;  
 }  
 }  
 System.*out*.println();  
 System.*out*.println("长度为"+v\_length);  
  
 List<Integer> s=new ArrayList<Integer>();  
 List<Integer> rs=new ArrayList<Integer>();  
 for(int m=0;m<num;m++)  
 s.add(m);  
 List<List<Integer> > l=null;  
 int ii=0;  
 System.*out*.println();  
 int max\_value=0;  
// System.out.println(max\_value);  
 List<Integer> result=new ArrayList<>();  
  
 if ( greedy\_sum\_value2==sum\_value){  
 result.add(1);  
 }else if(greedy\_sum\_value2<sum\_value){  
 result.add(-1);  
 }else {  
 result.add(0);  
 }  
 if(v\_length==arrange\_order.size()){  
 result.add(1);  
 }else {  
 result.add(0);  
 }  
 int index\_task\_0=0;  
 int equal =1;  
  
 for(int i=0;i<v.length;i++){  
 if(v[i]!=-1){  
  
 if(v[i]!=arrange\_order.get(index\_task\_0).i){  
 equal=0;  
 break;  
 }  
 index\_task\_0+=1;  
 }  
 }  
 if(equal==1){  
 result.add(1);  
 }else {  
 result.add(0);  
 }  
 return result;  
 }  
  
}

**问题5**

**问题描述**

【集合覆盖问题】给定一个集合X={*x*1,*x*2,…*xn*}以及X的一个子集簇F={*f*1,*f*2,…*fn*}，其中，*fi* ⊆X。求F的一个最小子集C⊆F，使得C中的集合能够覆盖集合X，即。

1. 已知图的顶点覆盖问题是NPC，证明集合覆盖问题是NP难的；
2. 利用回溯法或分支界限法设计求解集合覆盖问题的算法。

1 证明

证明NP难问题的一般步骤

要证明集合覆盖问题是NP难的，我们只需要证明集合覆盖问题的判定问题是NP完全问题即可。而要证明一个判定问题属于NPC问题，一般需要两个步骤：第一步，证明集合覆盖的判定问题属于NP问题；第二步，证明集合覆盖的判定问题可以归约到一个已经被证明了的NPC问题上去。

**概念描述**

集合覆盖的判定问题：给定一个包含n个元素的集合U；一个由m个子集Si⊆U的构成的集合；以及一个整数k。判定是否存在集合C⊆{1,2,…,m},|C|≤k，使得相应子集Si能够覆盖集合U中的所有元素

顶点覆盖的判定问题：给定一个图G=⟨V,E⟩，一个整数q，判定是否存在小于q的顶点覆盖W，使得对任意一条边{i,j}∈E 都有 i∈E 或者j∈E。

**证明过程**

第一步，我们证明集合覆盖的判定问题是属于NP问题的。对于任意给定的一个集合C，可以在多项式时间内检测出C中元素的个数是否超过k，也可以检验出这|C|个子集的并集是否覆盖了U中的所有元素。

第二步，我们证明集合覆盖的判定问题可以归约到图的顶点覆盖问题上。任意给定一个顶点覆盖问题G=⟨V,E⟩以及整数q，我们将根据如下规则生成一个对应的集合覆盖问题。令全集U=E；集合中的n个元素定义如下：将图G中的顶点分别标记为1到n，子集Si表示所有指向顶点i边的集合，满足Si⊆U；整数k=q。经过这样的变换，我们可以在多项式的时间内将一个顶点覆盖问题转化为一个集合覆盖问题。假设我们有一个“魔法求解器”可以对集合覆盖进行快速判定。接下来证明顶点覆盖问题判定为真当且仅当对应的集合覆盖问题为真。

设图G存在大小为q的顶点覆盖，令W为这些顶点集合。根据变换规则，W对应集合覆盖问题中的一系列子集C，由于k=q，C最多包含k个元素。下面说明子集C覆盖U：对于U中的任意一个元素e={i,j} ，e是图G中的一条边；由于W是图G的顶点覆盖（{i,j}∈E都有i∈E 或者j∈E），所以元素 e={i,j} 至少跟W中点 i 或者 j 关联，所以元素e至少属于C=i或者C=j时对应的子集 Si 或Sj。所以有子集C覆盖U。

设在集合覆盖问题中存在k个子集C使得 ⋃i∈c Si=U。由于每一个集合Si 都跟图中的一个顶点相关联，我们令W是这些顶点的集合。所以W中最多包含k个元素。由于子集C覆盖U，所以对任意一条边 e={i,j}来说，至少存在一个子集 =C包含e，因此必然存在顶点j∈W覆盖e。

综上两个步骤的证明，集合覆盖问题的判定问题是NPC问题。从而求最小的集合覆盖问题是NP难的。

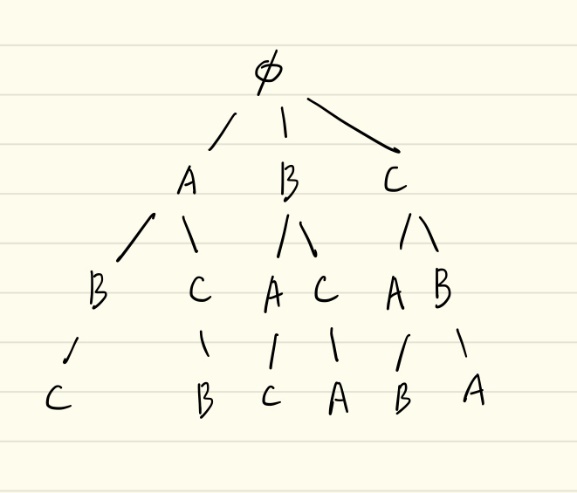
2

应该选用**分支界限法**，因为要求是求一个最小子集，回溯法能直接求所有子集，但是我们是要**找出最小的深度，所以要广度优先遍历**。

因为是**要求最小子集，应该采用排列树而不是子集树**，各节点优先级应该是越大的集合优先级越大，所以要采用优先队列式分支限界法，而不是队列式FIFO 分支界限法

分析问题解空间结构，假设子集数量为三，解空间结构如下，有一些重复的部分，但是不影响求解结果。图中每个节点存从根节点到此节点的路径序列（因为需要知道子节点的取值范围）和当前层数，BFS遍历到一个节点的时候就检查一下，是否覆盖了原集合。

一个节点对应一个子集，把子集按集合中元素数量排序



**算法流程**

1 层数t=0

2 找到所有可用子节点，加入队列，存好节点路径信息

3 把标志节点（-1）加入队列，之后如果遍历到这个节点，表示本层节点遍历结束，层数加一

4 取出队列中的节点， 判断该节点的路径序列对应的集合是否能覆盖原集合，能覆盖，程序结束，找到了一个最小覆盖，当前层数为集合个数，路径序列即为子集合。不能覆盖则跳转到5

5 通过其路径序列，找到其子节点取值范围，将子节点加入队列

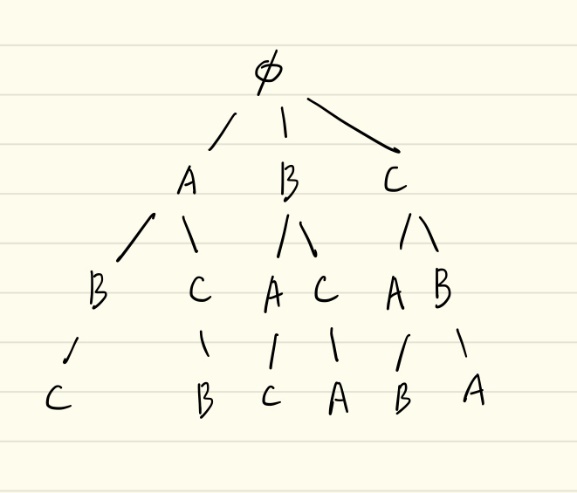
**编程实现**

首先，原集合中元素内容不重要，就选择 1~9 这些整数

随机生成不定数量，不定长度的各个子集，按照长度排序，长度越长，越优先遍历。

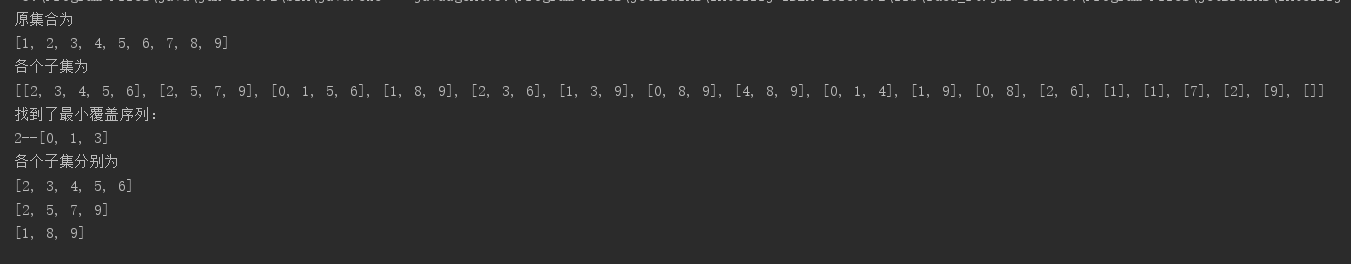
通过BFS来遍历各个排列，找到结果程序结束；

**难点**，在于记录可遍历子节点范围，这个和图的搜索不同，仅仅根据一个节点是不能知道子节点范围的。



**算法优化**：避免重复路径搜索，从这张图可以看到重复搜索了很多，这个题因为是求集合并集，各个节点的顺序是没用的，如果把搜索过了的集合组合加入到一个**已搜索列表**中，就可以减少计算量。如果子节点序列已经搜索过了，那么就把该节点剪枝掉，就是**不加入活节点队列**中。

实验结果



源代码

import java.util.\*;  
  
class Node{  
 int layer;  
 List<Integer> x=new ArrayList<>();  
 @Override  
 public String toString() {  
 return layer+"--"+x.toString();  
 }  
}  
public class BFS {  
 public static void main(String[] args) {  
 //随机生成数量不定的子集  
 List<Integer> X = new ArrayList<>();  
 Set<Integer> all\_X = new HashSet<>();  
 int size=10;  
 for (int i = 1; i < size; i++) {  
 X.add(i);  
 all\_X.add(i);  
 }  
 Random random =new Random();  
 List<Set<Integer>> setList =new ArrayList<>();  
 int subsetNum=random.nextInt(2\*size)+size;  
 for(int i=0;i<subsetNum;i++){//子集数量  
// System.out.println(i);  
 Set<Integer> temp=new HashSet<>();  
 for(int j=0;j<random.nextInt(size);j++){//子集长度  
 temp.add(random.nextInt(size));  
 }  
 setList.add(temp);  
 }  
  
 Collections.*sort*(setList, new Comparator<Set<Integer>>() {  
 @Override  
 public int compare(Set<Integer> o1, Set<Integer> o2) {  
 return o2.size()-o1.size();  
 }  
 });  
 System.*out*.println("原集合为");  
 System.*out*.println(all\_X);  
 System.*out*.println("各个子集为");  
 System.*out*.println(setList);  
 int set\_num = subsetNum-1;  
 Queue<Node> queue = new LinkedList<Node>();  
 Node sign = new Node();  
 sign.layer = -1;  
 ((LinkedList<Node>) queue).add(sign);  
 int t = 1;  
 Node cur = new Node();  
 cur.layer = 0;  
 cur.x.add(0);  
 int times=0;  
 while (times++<100000) {  
// System.out.println(queue);  
// System.out.println(cur.toString());  
 if(*check*(setList,cur,all\_X)){  
 System.*out*.println("找到了最小覆盖序列：");  
 System.*out*.println(cur);  
 System.*out*.println("各个子集分别为");  
 for(Integer i:cur.x){  
 System.*out*.println(setList.get(i));  
 }  
 break;  
 }  
 for (int k = 1; k <= set\_num; k++) {  
 Node q = new Node();  
 q.layer = t;  
 q.x.add(0);  
 for (int i = 1; i < t; i++) {  
 q.x.add(cur.x.get(i));  
 }  
 if (!q.x.contains(k)) {  
 q.x.add(k);  
 ((LinkedList<Node>) queue).add(q);  
 }  
 }  
 cur = ((LinkedList<Node>) queue).poll();  
 if (cur.layer == -1) {  
 t++;  
 ((LinkedList<Node>) queue).add(sign);  
 cur = queue.poll();  
 }  
 if (cur.layer == set\_num) {  
 System.*out*.println("finish");  
 System.*out*.println(cur.x);  
 }  
 }  
 }  
 static boolean check(List<Set<Integer>> subsets, Node node, Set<Integer> all\_x){  
 Set<Integer> set=new HashSet<>();  
 for( Integer i: node.x){  
 Set<Integer> item=subsets.get(i);  
 set.addAll(item);  
 }  
  
 if(set.equals(all\_x)){  
 return true;  
 }else return false;  
 }  
}

如果对此文档有疑问，请联系 2575502933@qq.com