在页面上绘画了多个矩形，每个矩形有编号从0开始，矩形边都与x，y轴平行。其中某些矩形受到污染称之为dirty矩形；；覆盖所有dirty矩形的最小矩形区域称为dirty区域。请找出与dirty区域有重叠（含部分重叠）的矩形以便重新绘制，并按升序返回这些矩形编号的序列；若无重叠的矩形，则返回空序列[]

输入案例

Rectangles={  
 {1, 6, 2, 2},  
 {4, 6, 1, 1},  
 {6, 4, 2, 1},  
 {3, 2, 2, 3},  
 {8, 2, 1, 1},  
 {9, 5, 1, 1},  
 {6, 7, 2, 1},  
 {9, 1, 1, 1},  
 {2, 5, 1, 1},  
}

dirty= {1, 4, 6}

输出[1, 2, 3, 4, 6]

解释



* + 有3个dirty矩形（编号 1、4、6），覆盖所有dirty矩形的最小区域，即dirty区域，为虚线区域
  + 编号为 0 的矩形，其右下角 x坐标小于dirty区域左上角 x坐标，所以与dirty区域不重叠。 同理，编号为5、7、8 的矩形与 dirty 区域也不重叠。
  + 与dirty区域重叠的矩形有5个：三个 dirty 矩形（编号 1、4、6），两个非 dirty矩形（编号 2、3，其中3为部分重叠）

有重叠的矩形按编号升序排列输出为 [1 2 3 4 6]

输入案例2

Rectangles={  
 {1, 6, 2, 2},  
 {4, 6, 1, 1},  
}

dirtyRectangles= {}

输出[]

系统下的文件夹以下图所示的「树状格式」来表达：

home

GAMES

usr

DRIVERS

lib64

log

var

log

说明：  
- 所有文件夹名称由字母和数字组成并区分大小写  
- 只有一个根文件夹，且为首行，无缩进；其余每个文件夹都比其父文件夹缩进 2 个空格  
- 同一父文件夹下的文件夹名称不重复，非同一父文件夹下的名称允许重复  
- 同一父文件夹下的文件夹按名称的字典序升序排列

给定两个树状格式的文件夹：targetDir 和 srcDir，请将 srcDir 文件夹**下的内容**复制到 targetDir 中的指定文件夹 dstDir **下**（类似windows的文件夹复制）

dstDirLine表示复制daotargetDir的第几行

**输入**

**TargetDirStr=**

home

GAMES

usr

DRIVERS

local

lib64

log

var

log

**srcDirStr=**

bin

LICENSES

conf

local

lib32

log

**dstDirLine=3**

输出

home

GAMES

usr

DRIVERS

LICENSES

conf

local

lib32

lib64

log

var

log

解释

dstDirLine=3 表示复制到usr

将源文件夹 bin 下的内容（不含bin本身），复制到目的文件夹 usr 下：

local 文件夹同名，需要将其下的内容逐层合并

LICENSES 文件夹在 home/usr 下不存在，需要复制过去（含子文件夹）

各层文件夹名称按字典序升序排列

输入样例2

**TargetDirStr=**

home

GAMES

usr

GAMES

games

**srcDirStr=空**

**dstDirLine=3**

输出

home

GAMES

usr

GAMES

Games

public class T2 {  
 public static void main(String[] args) {  
 String[] targetDir = targetDirStr.split("\n");

String[] srcDir = srcDirStr.split("\n");  
 }  
  
 private static String[] copyDir(String[] targetDir, int dstDirLine, String[] srcDir) {  
 // 开发您的代码  
 }  
  
 static class Node {  
 int level = -1;  
 String name = null;  
 List<Node> children = null;  
  
 Node(String name, int level) {  
 this.name = name;  
 this.level = level;  
 this.children = new ArrayList<>();  
 }  
  
 void addChild(Node child) {  
 if (!this.children.contains(child)) {  
 this.children.add(child);  
 }  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 if (obj instanceof Node) {  
 Node temp = (Node) obj;  
 return this.name.equals(temp.name) && (this.level == temp.level);  
 }  
 return false;  
 }  
  
 @Override  
 public int hashCode() {  
 return this.name.hashCode() + this.level;  
 }  
 }  
  
 /\*\*  
 \* 根据输入信息完成树的构建，此部分代码可自行决定是否要改动和调用  
 \*  
 \* @param strs 输入的文件夹信息（字符串数组）  
 \* @return List<Node> 完成构建后的树节点列表，首元素为树的根节点  
 \*/  
 private static List<Node> buildTree(String[] strs) {  
 List<Node> nodes = new ArrayList<>();  
 for (int i = 0; i < strs.length; i++) {  
 int idx = strs[i].lastIndexOf(" ");  
 nodes.add(new Node(strs[i].substring(idx + 1), (idx + 1) / 2));  
 }  
 // 完成子节点挂接  
 for (int i = 0; i < nodes.size(); i++) {  
 Node cur = nodes.get(i);  
 for (int j = i + 1; j < nodes.size(); j++) {  
 Node temp = nodes.get(j);  
 if (cur.level >= temp.level) {  
 break;  
 }  
 if (temp.level == cur.level + 1) {  
 cur.addChild(temp);  
 }  
 }  
 }  
 return nodes;  
 }  
  
}

题目3

二维数组来表示密码板，二维数组中每一行的数字都按升序排列。

一个密码生成规则：按顺序从每一行中各选出一个数字，这组数字（其下标序列 {C1, C2, … Cn} ）之和做为一个密码。  
按规则选择多组数字，并要求其下标序列 {C1, C2, … Cn} 是唯一的（即至少一行的下标不同），则会形成多个密码。

请问，在所有可生成的密码组成的**降序**序列中，第 num 个密码（从1开始计数）是多少？

**输入**

* Nums=[ [1 4 7 10] [2 3 5 9] [1 1 12 13] ] num=5

输出

28

解释

从第一行选择10，第二行选择9，第三行选择13，组成的新数字32=10+9+13  
从第一行选择10，第二行选择9，第三行选择12，组成的新数字31=10+9+12  
以此类推，新数字组合可能为：[10,9,13]和为32, [10,9,12]和为31, [10,5,13]和为28, [10,5,12]和为27, [7,9,13]和为29, [7,9,12]和为28, ……  
降序序列的前 5 个为 32, 31, 29, 28, 28， 因此输出 28

以下是我的参考解法，不是很优化

public class T1 {

    public static void main(String[] args) {

        case03();

    }

    private static void case01() {

        int[][] rectangles = new int[][] {

            {1, 6, 2, 2},

            {4, 6, 1, 1},

            {6, 4, 2, 1},

            {3, 2, 2, 3},

            {8, 2, 1, 1},

            {9, 5, 1, 1},

            {6, 7, 2, 1},

            {9, 1, 1, 1},

            {2, 5, 1, 1},

        };

        int[] dirtyRectangles = new int[] {1, 4, 6};

        System.out.println(Arrays.toString(getDeleted(rectangles, dirtyRectangles)));

        System.out.println("[1, 2, 3, 4, 6] = 正确输出");

    }

    private static void case02() {

        int[][] rectangles = new int[][] {

            {1, 6, 2, 2},

            {4, 6, 1, 1},

        };

        int[] dirtyRectangles = new int[] {};

        System.out.println(Arrays.toString(getDeleted(rectangles, dirtyRectangles)));

        System.out.println("[] = 正确输出");

    }

    private static void case03() {

        int[][] rectangles = new int[][] {

            {1, 6, 2, 2},

            {3, 8, 2, 2},

            {6, 4, 2, 1},

            {3, 2, 2, 3},

            {8, 2, 1, 1},

            {8, 5, 2, 2},

            {6, 7, 2, 1},

            {8, 2, 2, 2},

            {2, 5, 3, 3},

        };

        int[] dirtyRectangles = new int[] {1, 4, 6};

        System.out.println(Arrays.toString(getDeleted(rectangles, dirtyRectangles)));

        System.out.println("[1, 2, 3, 4, 5, 6, 7, 8] = 正确输出");

    }

    private static int[] getDeleted(int[][] rectangles, int[] dirtyRectangles) {

        Set<Integer> dirtyIndexList = Arrays.stream(dirtyRectangles).boxed().collect(Collectors.toSet());

        ArrayList<Point> pointList = new ArrayList<>();

        ArrayList<Integer> result = new ArrayList<>();

        int x = Integer.MAX\_VALUE, y = Integer.MAX\_VALUE, x1 = 0, y1 = 0;

        // 1. 先转换为Point对象 x,y(左下角点) x1,y1(右上角点) 并 找到dirty的最大区域

        for (int i = 0; i < rectangles.length; i++) {

            int[] rectangle = rectangles[i];

            if (dirtyIndexList.contains(i)) {

                result.add(i);

                x = Math.min(x, rectangle[0]);

                y = Math.min(y, rectangle[1] - rectangle[3]);

                x1 = Math.max(x1, rectangle[0] + rectangle[2]);

                y1 = Math.max(y1, rectangle[1]);

            } else {

                pointList.add(new Point(rectangle, i));

            }

        }

        // 2. dirty的最大区域构成的point

        Point pointDirty = new Point(x, y, x1, y1);

        // 4. 将非dirtyRectangles的point添加到 集合中并比较

        pointList.forEach(point -> {

            // 注意：这里不用pointDirty.isIn(point) 因为我们判断的是point是否在pointDirty里

            if (point.isIn(pointDirty)) {

                result.add(point.index);

            }

        });

        return result.stream().sorted().mapToInt(Integer::intValue).toArray();

    }

    private static class Point {

        private int index;

        private int x;

        private int y;

        private int x1;

        private int y1;

        /\*\*

         \* 构造,将 rectangles 中的点转换为Point对象

         \*/

        public Point(int[] rectangle, int index) {

            this.index = index;

            this.x = rectangle[0];

            this.y = rectangle[1] - rectangle[3];

            this.x1 = rectangle[0] + rectangle[2];

            this.y1 = rectangle[1];

        }

        public Point(int x, int y, int x1, int y1) {

            this.x = x;

            this.y = y;

            this.x1 = x1;

            this.y1 = y1;

        }

        /\*\*

         \* 3. 写个方法 给你个point 判断是否与这个point有覆盖

         \* 判断两个区域是否有交集

         \* ((this.y > p.y && this.y < p.y1) || (this.y1 > p.y && this.y1 < p.y1)) (只要y || y1在(p.y,p.y1) 区间就满足)

         \* && ((this.x > p.x && this.x < p.x1) || (this.x1 > p.x && this.x1 < p.x1)) (只要x || x1在(p.x,p.x1) 区间就满足)

         \*/

        public boolean isIn(Point p) {

            return ((this.y > p.y && this.y < p.y1) || (this.y1 > p.y && this.y1 < p.y1))

                && ((this.x > p.x && this.x < p.x1) || (this.x1 > p.x && this.x1 < p.x1));

        }

    }

}

public class T2 {

    public static void main(String[] args) {

        String targetDirStr = ""

            + "home\n"

            + "  GAMES\n"

            + "  usr\n"

            + "    DRIVERS\n"

            + "    local\n"

            + "      lib64\n"

            + "    log\n"

            + "  var\n"

            + "    log";

        String srcDirStr = ""

            + "bin\n"

            + "  LICENSES\n"

            + "    conf\n"

            + "  local\n"

            + "    lib32\n"

            + "  log";

        int dstDirLine = 3;

        // targetDirStr = ""

        //     + "home\n"

        //     + "  log\n"

        //     + "    log1\n"

        //     + "    log2\n"

        //     + "  temp\n"

        //     + "    temp1\n"

        //     + "    temp2\n"

        //     + "      temp2-1\n"

        //     + "        temp3-1\n"

        //     + "  usr\n"

        //     + "    usr\n"

        //     + "      temp\n"

        //     + "        temp1\n"

        //     + "      log";

        // srcDirStr = ""

        //     + "home\n"

        //     + "  log\n"

        //     + "    log1\n"

        //     + "    log2\n"

        //     + "    log3\n"

        //     + "  temp\n"

        //     + "    temp3\n"

        //     + "    temp1\n"

        //     + "      temp1-1\n"

        //     + "    temp2\n"

        //     + "      temp2-1\n"

        //     + "        temp3-2\n"

        //     + "  usr\n"

        //     + "    usr\n"

        //     + "      temp\n"

        //     + "        temp1\n"

        //     + "    log";

        // dstDirLine = 1;

        String[] targetDir = targetDirStr.split("\n");

        String[] srcDir = srcDirStr.split("\n");

        String[] results1 = copyDir20211206113747(targetDir.clone(), dstDirLine, srcDir.clone());

        String[] results2 = Test2.copyDir(targetDir.clone(), dstDirLine, srcDir.clone());

        System.out.println(check(results1, results2));

        // Arrays.stream(results).forEach(System.out::println);

    }

    private static String[] copyDir(String[] targetDir, int dstDirLine, String[] srcDir) {

        // code in here

        return null;

    }

    /\*\*

     \* 根据输入信息完成树的构建，此部分代码可自行决定是否要改动和调用

     \*

     \* @param strs 输入的文件夹信息（字符串数组）

     \* @return List<Node> 完成构建后的树节点列表，首元素为树的根节点

     \*/

    private static List<Node> buildTree(String[] strs) {

        List<Node> nodes = new ArrayList<>();

        for (int i = 0; i < strs.length; i++) {

            int idx = strs[i].lastIndexOf(" ");

            nodes.add(new Node(strs[i].substring(idx + 1), (idx + 1) / 2));

        }

        // 完成子节点挂接

        for (int i = 0; i < nodes.size(); i++) {

            Node cur = nodes.get(i);

            for (int j = i + 1; j < nodes.size(); j++) {

                Node temp = nodes.get(j);

                if (cur.level >= temp.level) {

                    break;

                }

                if (temp.level == cur.level + 1) {

                    cur.addChild(temp);

                }

            }

        }

        return nodes;

    }

    /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 以上都是已经提供的代码 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

    private static boolean check(String[] strings1, String[] strings2) {

        if (strings1.length != strings2.length) {

            return false;

        }

        for (int i = 0; i < strings1.length; i++) {

            if (strings1[i].equals(strings2[i])) {

                System.out.println(strings1[i]);

            } else {

                System.out.println("------------------------------------");

                System.out.println(strings1[i]);

                System.out.println(strings2[i]);

                return false;

            }

        }

        return true;

    }

    private static String[] copyDir20211206113747(String[] targetDir, int dstDirLine, String[] srcDir) {

        // 1. 利用api构成成树

        List<Node> targetTree = buildTree(targetDir);

        Node targetNode = targetTree.get(dstDirLine - 1);

        // 2. 更具level制作空格

        String space = " ";

        for (int i = 0; i < targetNode.level; i++) {

            space += "  ";

        }

        // 遍历srcDir 全部加入space

        for (int i = 0; i < srcDir.length; i++) {

            srcDir[i] = space + srcDir[i];

        }

        // 将srcDir 转换为tree

        List<Node> srcTree = buildTree(srcDir);

        // 获取srcTree的root

        Node srcRoot = srcTree.get(0);

        // 将srcRoot.child 加入到 targetNode.child中

        mergeNode(targetNode, srcRoot);

        // 将Node 转换为str

        Node root = targetTree.get(0);

        return buildStr(root).toArray(new String[] {});

    }

    private static List<String> buildStr(Node node) {

        ArrayList<String> result = new ArrayList<>();

        String temp = node.name;

        for (int i = 0; i < node.level; i++) {

            temp = "  " + temp;

        }

        result.add(temp);

        if (node.children != null && !node.children.isEmpty()) {

            node.children.forEach(child -> result.addAll(buildStr(child)));

        }

        return result;

    }

    private static void mergeNode(Node node1, Node node2) {

        List<Node> children = node1.children;

        HashMap<String, Node> map = new HashMap<>();

        children.forEach(node -> map.put(node.level + node.name, node));

        ArrayList<Node> childrenList = new ArrayList<>();

        for (Node child : node2.children) {

            Node node = map.get(child.level + child.name);

            if (node != null) {

                mergeNode(node, child);

            } else {

                childrenList.add(child);

            }

        }

        node1.children.addAll(childrenList);

        node1.children.sort(Comparator.comparing(o -> o.name));

    }

    static class Node {

        int level = -1;

        String name = null;

        List<Node> children = null;

        Node(String name, int level) {

            this.name = name;

            this.level = level;

            this.children = new ArrayList<>();

        }

        @Override

        public boolean equals(Object obj) {

            if (obj instanceof Node) {

                Node temp = (Node) obj;

                return this.name.equals(temp.name) && (this.level == temp.level);

            }

            return false;

        }

        @Override

        public int hashCode() {

            return this.name.hashCode() + this.level;

        }

        @Override

        public String toString() {

            return "Node{" +

                "level=" + level +

                ", name='" + name + '\'' +

                ", children=" + children +

                '}';

        }

        void addChild(Node child) {

            if (!this.children.contains(child)) {

                this.children.add(child);

            }

        }

    }

    /\*\*

     \* 功能描述

     \*

     \* @author h30009904

     \* @since 2021-12-03

     \*/

    private static class Test2 {

        /\*\*

         \* 待实现函数copyDir，在此函数中填入答题代码

         \*

         \* @param targetDir 输入的文件夹信息（字符串数组）

         \* @param dstDirLine 复制的目的文件夹在targetDir中的位置，即复制的目的文件夹为targetDir[dstDirLine - 1]

         \* @param srcDir 复制的源文件夹信息（字符串数组）

         \* @return String[] 「树状格式」输出信息

         \*/

        private static String[] copyDir(String[] targetDir, int dstDirLine, String[] srcDir) {

            //

            if (srcDir == null || srcDir.length == 0) {

                return targetDir;

            }

            List<Node> targetNodeTree = buildTree(targetDir);

            // 将srcDirNodeTree下的值全部复制过去

            List<Node> srcDirNodeTree = buildTree(srcDir);

            // 挂载在nodeList下面

            Node node = targetNodeTree.get(dstDirLine - 1);

            //

            copyNode(node, srcDirNodeTree.get(0));

            List<String> list = new ArrayList<>();

            readTree(targetNodeTree.get(0), list);

            String[] res = new String[list.size()];

            for (int i = 0; i < list.size(); i++) {

                res[i] = list.get(i);

            }

            return res;

        }

        /\*\*

         \* 读取数结构的值

         \*

         \* @param node 读取树结构

         \* @param list 装载返回值

         \*/

        private static void readTree(Node node, List<String> list) {

            if (node == null) {

                return;

            }

            int level = node.level;

            StringBuilder preName = new StringBuilder();

            for (int i = 0; i < level; i++) {

                preName.append("  ");

            }

            list.add(preName + node.name);

            if (node.children != null) {

                for (Node child : node.children) {

                    readTree(child, list);

                }

            }

        }

        /\*\*

         \* 复制结构

         \*

         \* @param targetNode 目标文件夹结构

         \* @param srcDirNode 待复制文件结构

         \*/

        private static void copyNode(Node targetNode, Node srcDirNode) {

            if (srcDirNode == null) {

                return;

            }

            List<Node> children = targetNode.children;

            if (children == null) {

                children = srcDirNode.children;

            }

            Map<String, Node> nodeNameMap = children.stream()

                .collect(Collectors.toMap(node -> node.name, Function.identity()));

            List<Node> tempList = new ArrayList<>();

            List<Node> srcChildNodeList = srcDirNode.children;

            for (Node value : srcChildNodeList) {

                Node node = nodeNameMap.get(value.name);

                if (node == null) {

                    resolveLevel(value, targetNode.level + 1);

                    tempList.add(value);

                } else {

                    copyNode(node, value);

                }

            }

            children.addAll(tempList);

            children.sort(Comparator.comparing(node -> node.name));

        }

        /\*\*

         \* 调整层级结构

         \*

         \* @param node 待调整node

         \* @param level 层级

         \*/

        private static void resolveLevel(Node node, int level) {

            if (node == null) {

                return;

            }

            node.level = level;

            if (node.children == null) {

                return;

            }

            for (Node child : node.children) {

                resolveLevel(child, level + 1);

            }

        }

    }

}

public class T3 {

    public static void main(String[] args) {

        int[][] ints = new int[][] {

            {1, 4, 7, 10},

            {2, 3, 5, 9},

            {1, 1, 12, 13},

        };

        System.out.println(solution20211206195034(ints, 5));

        int row = RandomUtil.randomInt(1000);

        int col = RandomUtil.randomInt(1000);

        ArrayList<List<Integer>> lists = new ArrayList<>();

        int[][] ints1 = new int[row][col];

        for (int i = 0; i < row; i++) {

            ArrayList<Integer> integers = new ArrayList<>();

            for (int j = 0; j < col; j++) {

                integers.add(RandomUtil.randomInt(10000));

            }

            List<Integer> collect = integers.stream().sorted().collect(Collectors.toList());

            lists.add(collect);

            ints1[i] = collect.stream().mapToInt(Integer::valueOf).toArray();

        }

        // System.out.println(lists);

        // System.out.println(row \* col);

        int index = RandomUtil.randomInt(row \* col);

        // System.out.println(solution20211206195034(ints1, index));

    }

    /\*\*

     \* 暴力

     \* 一定超时！！！

     \*/

    public static int solution20211206195034(int[][] nums, int index) {

        List<Integer> extracted = extracted(nums, 0, 0);

        extracted.sort((o1, o2) -> o2 - o1);

        System.out.println(extracted.size());

        return extracted.get(index - 1);

    }

    private static List<Integer> extracted(int[][] nums, int sum, int rowIndex) {

        ArrayList<Integer> integers = new ArrayList<>();

        for (int i : nums[rowIndex]) {

            sum += i;

            if (rowIndex != nums.length - 1) {

                integers.addAll(extracted(nums, sum, rowIndex + 1));

            } else {

                integers.add(sum);

            }

            sum -= i;

        }

        return integers;

    }

}