

## Questions:

- Longest palindrome, window sum(easy), overlapping rectangle(easy), k closest points
- MST, copy linkedlist with random pointer, order dependency, Average of 5 highest score(easy), subtree with Maximum average

## Overlapping rectangle (Easy) :

给定两个长方形的左上和右下的坐标 判断重合与否

```
public class OverLap {
    // rectangle A, B
    // time O(1), space O(1)
    public static boolean check(Node topLeftA, Node topLeftB, Node bottomRightA, Node
bottomRightB) {
        //if one is above another or x value range does not overlap
        //line overlap is considered as overlap
        if(topLeftA.x > bottomRightB.x || topLeftB.x > bottomRightA.x || bottomRightA.y >
topLeftB.y || bottomRightB.y > topLeftA.y){
            return false;
        }

        return true;
    }

    public static class Node {
        double x;
        double y;
        public Node(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }
}
```

不知道是左下右上还是左上右下 solution 按照左上右下处理

边重合视为重合

有可能左下右上的坐标是反的？反而左下是右上？

## WindowSum(Easy):

window sum 就是给一个包含整数的 `arraylist` 和一个 window size k, 返回所有长度为 k 的窗口的数的和。比如数组[1,2,3,4,5],window size 2, 那么长度为 2 的窗口就是 [1,2],[2,3],[3,4],[4,5],和就依次是 3,5,7,9.

```
import java.util.List;
import java.util.ArrayList;
```

```
public class WinSum {
    public List<Integer> GetSum(List<Integer> A, int k) {

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

        //corner case
        if(A == null || A.size() == 0){
            return res;
        }
        if(k > A.size() || k <= 0){
            return res;
        }

        int len = A.size();

        for (int i = 0; i < len - k + 1; ++i) {
            int sum = 0;
            for (int j = 0; j < k; ++j) {
                sum += A.get(i+j);
            }
            res.add(i,sum);
        }

        return res;
    }
}
```

记得 import 注意 edge case  
edge case 不确定

## Longest Palindrome Substring:

LeetCode 原题 <https://leetcode.com/problems/longest-palindromic-substring/>

```
public String longestPalindrome(String s) {
    //corner case
    if(s == null || s.length() == 0){
        return s;
    }

    String longestPal = "";
    for(int i = 0; i < s.length(); i++){
```

```

        int rightEvenIndex = i + 1;
        int rightOddIndex = i + 1;
        int leftEvenIndex = i;
        int leftOddIndex = i - 1;
        while(rightEvenIndex < s.length() && leftEvenIndex >=0 && s.charAt(rightEvenIndex) == s.charAt(leftEvenIndex)){
            rightEvenIndex++;
            leftEvenIndex--;
        }
        while(rightOddIndex < s.length() && leftOddIndex >=0 && s.charAt(rightOddIndex) == s.charAt(leftOddIndex)){
            rightOddIndex++;
            leftOddIndex--;
        }

        int oddLength = rightOddIndex - leftOddIndex - 1;
        int evenLength = rightEvenIndex - leftEvenIndex - 1;

        if(oddLength > evenLength){
            if(oddLength > longestPal.length()){
                longestPal = s.substring(leftOddIndex + 1, rightOddIndex);
            }
        }
        else{
            if(evenLength > longestPal.length()){
                longestPal = s.substring(leftEvenIndex + 1, rightEvenIndex);
            }
        }

    }

    return longestPal;

}

```

manacher's algorithm:

```

public class ManacherSolution {
    public String longestPalindrome(String s) {
        String T = preProcess(s);
        int n = T.length();
        int[] p = new int[n];
        int center = 0, right = 0;
        for (int i = 1; i < n - 1; i++) {
            int j = 2 * center - i; //j and i are symmetric around center
            p[i] = (right > i) ? Math.min(right - i, p[j]) : 0;

            // Expand palindrome centered at i

```

```

        while (T.charAt(i + 1 + p[i]) == T.charAt(i - 1 - p[i]))
            p[i]++;

        // If palindrome centered at i expand past right,
        // then adjust center based on expand palindrome
        if (i + p[i] > right) {
            center = i;
            right = i + p[i];
        }
    }

    // Find the longest palindrome
    int maxLength = 0, centerIndex = 0;
    for (int i = 1; i < n - 1; i++) {
        if (p[i] > maxLength) {
            maxLength = p[i];
            centerIndex = i;
        }
    }

    centerIndex = (centerIndex - 1 - maxLength) / 2;
    return s.substring(centerIndex, centerIndex + maxLength);
}

// preprocess the original string.
// For example, s = "abcdefg", then the rvalue = "^#a#b#c#d#e#f#g#$"
private String preprocess(String s) {
    if (s == null || s.length() == 0) return "^$";
    StringBuilder rvalue = new StringBuilder("^");
    for (int i = 0; i < s.length(); i++)
        rvalue.append("#").append(s.substring(i, i+1));
    rvalue.append("#$");
    return rvalue.toString();
}
}

```

Pre-Process 的时候用 StringBuilder 会比较快

## K Closest Points:

Find the K closest points to the origin in a 2D plane, given an array containing N points.

就是 **sort array** 要掌握 **priority queue** 记得 **import**

```

import java.util.PriorityQueue;
import java.util.Comparator;
import java.lang.Math;

//heap sort
public class Practice {
    public static Point[] KNearestPoints(Point[] array, Point origin, int k) {
        //corner case
        if(array == null || array.length == 0){
            return array;
        }
        if(k <= 0){
            return new Point[0];
        }
    }
}

```

```

// if k > array's length return sorted array
if(k > array.length){
    k = array.length;
}
Point[] result = new Point[k];

//initialize pq
PriorityQueue<Point> pq = new PriorityQueue<Point>(array.length, new
Comparator<Point>(){
    public int compare(Point a, Point b){
        if(getDistance(origin, a) > getDistance(origin, b)){
            return 1;
        }
        if(getDistance(origin, a) < getDistance(origin, b)){
            return -1;
        }
        return 0;
    }
});

//put the points into pq
for(Point p : array){
    pq.offer(p);
}

//get the first k points from pq
for(int i = 0; i < k; i++){
    result[i] = pq.poll();
}

return result;
}

private static double getDistance(Point origin, Point p){
    return Math.sqrt((origin.x - p.x) * (origin.x - p.x) + (origin.y - p.y) *
(origin.y - p.y));
}

```

当 edge cases 不同的时候，返回的值也要不同。

如果是 list 为 null 或者 list 的 size 为空，返回的是原来的 list。

但是如果是 k 小于等于零的情况，要返回的是新建的 new arraylist 而不是 null。

K 大于数组的长度时 返回 sorted array

## Copy Linked List with Random Pointer:

用 hashtable 和不用 hashtable

非常要注意的是 deep copy 里面有一个 class ALNode，里面有一个 property 写错了，arbitrary 写成了 abritrary

不用 hash table 更优

```

public class Solution {

    private RandomListNode copyList(RandomListNode head){

```

```

        RandomListNode current = head;
        while(current != null){
            RandomListNode newNode = new RandomListNode(current.label);
            newNode.next = current.next;
            current.next = newNode;
            current = current.next.next;
        }
        return head;
    }

    private RandomListNode copyRandom(RandomListNode head){
        RandomListNode current = head;
        while(current != null){
            if(current.random == null){
                current.next.random = null;
            }
            else{
                current.next.random = current.random.next;
            }
            current = current.next.next;
        }
        return head;
    }

    private RandomListNode splitList(RandomListNode head){
        RandomListNode result = head.next;
        RandomListNode dummy = head.next;
        while(result.next != null){
            head.next = result.next;
            head = head.next;
            result.next = head.next;
            result = result.next;
        }

        head.next = null;

        return dummy;
    }

    public RandomListNode copyRandomList(RandomListNode head) {
        //corner case
        if(head == null){
            return head;
        }

        RandomListNode result = new RandomListNode(0);

```

```

        result = copyList(head);
        result = copyRandom(result);
        return splitList(result);
    }
}

#####Hash Map version
public class Deep_Copy_List {
    public static RandomListNode deepCopy(RandomListNode head){
        //建一个对应的映射表
        Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode,
RandomListNode>();
        RandomListNode point = head;
        while (point != null){
            //来一个,建一个
            RandomListNode cur = new RandomListNode(point.val);
            map.put(point, cur);
            point = point.next;
        }
        point = head; //复位
        while (point != null){
            //开始了复制的旅程
            map.get(point).next = map.get(point.next);
            map.get(point).random = map.get(point.random);
            point = point.next;
        }
        return map.get(head);
    }
}

```

## Order Dependency:

与 coursera schedule 非常像—graph

<https://leetcode.com/problems/course-schedule-ii/>

```

import java.util.*;
//Order 和 OrderDependency 都是照着面经猜测，应该是给好的 class。
class Order{
    String order = "";
    public Order(String string){
        this.order = string;
    }
}

```

```

class OrderDependency{
    Order cur;
    Order pre;
    public OrderDependency(Order o1, Order o2){
        cur = o1;
        pre = o2;
    }
}

public class Order_Dependency {
    //这个参数可能是数组，这里先摆个容器，反正一个意思。
    public static List<Order> getOrderList(List<OrderDependency> orderDependencies){
        List<Order> result = new ArrayList<>();
        //建两个 map,第一个是当前的 order 指向多少个 order,就是先决条件
        Map<Order, ArrayList<Order>> map = new HashMap<>();
        //第二个是当前 order 被多少个 order 指,即为入度
        Map<Order, Integer> inMap = new HashMap<>();
        //把出现过的都记录下来
        Set<Order> set = new HashSet<>();
        for (OrderDependency od : orderDependencies){
            Order cur = od.cur;
            Order pre = od.pre;
            set.add(cur);
            set.add(pre);
            //有则加一,无则设 1
            if (inMap.containsKey(cur)){
                int indegree = inMap.get(cur);
                inMap.put(cur, indegree+1);
            }
            else {
                inMap.put(cur, 1);
            }
            //这里入度也要把 pre 加上,因为最后要找线头,就是入度为 0 的点。
            if (!inMap.containsKey(pre)){
                inMap.put(pre, 0);
            }

            if (map.containsKey(pre)){
                map.get(pre).add(cur);
            }
            else {
                map.put(pre, new ArrayList<>());
                map.get(pre).add(cur);
            }
            //注意这里存的时候,map 可以看成出度,出度为 0 也要存,或者在下面判断跳过 null
            if (!map.containsKey(cur)){

```



```

        map.put(cur, new ArrayList<Order>());
    }
}

Queue<Order> queue = new LinkedList<>();
int total = set.size();

for (Order od : inMap.keySet()){
    if (inMap.get(od) == 0){
        queue.offer(od);
    }
}
//这里使用了 BFS
while (!queue.isEmpty()){
    Order order = queue.poll();
    result.add(order);
    for (Order odr : map.get(order)){
        //这里查入度,类比剥洋葱,如果剥到了 0,说明是最外层。
        inMap.put(odr, inMap.get(odr) - 1);
        if (inMap.get(odr) == 0){
            queue.offer(odr);
        }
    }
}
//这里如果有环的话,肯定是剥不出来的,那么 set 里面的个数和 result 里面的个数不一致。
if (result.size() != total) return new ArrayList<Order>();
return result;
}
}

```

## Minimum Spanning Tree:

题目内容是这样的，给十几个城市供电，连接不同城市的花费不同，让花费最小同时连到所有的边。给出一系列 connection 类，里面是 edge 两端的城市名和它们之间的一个 cost，找出要你挑一些边，把所有城市连接起来并且总花费最小。不能有环，最后所以城市要连成一个连通块。

**mst** 需要注意的就是如果 connection 是空的 return 空 list，但是如果是无法 connect 所有的 city 的话 return 的是个 null

面经

<http://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=211005&ctid=371>

Kruskal+Union Find

Kruskal 详解：<http://www.cnblogs.com/yanlingyin/archive/2011/11/16/greedy.html>

Union Find 详解：[http://blog.csdn.net/dm\\_vincent/article/details/7655764](http://blog.csdn.net/dm_vincent/article/details/7655764)

//给好的 connection class，两个城市名，和一个 cost。

import java.util.\*; //这句话极度重要

class Connection{

String node1;

String node2;

int cost;

public Connection(String a, String b, int c){

node1 = a;

node2 = b;

cost = c;

}

}

//下面进入正题

public class City\_Connections {

private static int unionNum; //这里开个全局变量，不丢人。

//这个 static 是题目要求的，我自己一般不写，累。

public static ArrayList<Connection> getLowCost(ArrayList<Connection> connections){

//先把空的情形挡掉

if (connections == null || connections.size() == 0){

return new ArrayList<>();

}

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

Map<String, Integer> map = new HashMap<>();

//这里把 cost 小的往前排。

Collections.sort(connections, new Comparator<Connection>() {

@Override

public int compare(Connection o1, Connection o2) {

return o1.cost - o2.cost;

}

});

unionNum = 0;

for (Connection c : connections){

String a = c.node1;

```

String b = c.node2;
//看城市是不是连过了, 要是可以连, 那么就在 result 里面加上
if (union(map, a, b)){
    result.add(c);
}
}
//这里要检查一下,是不是所有的城市都属于同一个 union
String find = connections.get(0).node1;
int union = map.get(find);
for (String str : map.keySet()){
    // 如果我们中出了一个叛徒, 返回空表
    if (map.get(str) != union){
        return null;
    }
}
//这里最后要求按照城市的名字排序
Collections.sort(result, new Comparator<Connection>() {
    @Override
    public int compare(Connection o1, Connection o2) {
        if(o1.node1.equals(o2.node1)){
            return o1.node2.compareTo(o2.node2);
        }
        return o1.node1.compareTo(o2.node1);
    }
});
return result;
}
private static boolean union(Map<String, Integer> map, String a, String b){
    if (!map.containsKey(a) && !map.containsKey(b)){
        map.put(a, unionNum);
        map.put(b, unionNum);
        //这里用了一个新的没用过的数字
        unionNum++;
        return true;
    }
    //只有一方落单,那就加入有组织的一方
    if (map.containsKey(a) && !map.containsKey(b)){
        int aU = map.get(a);
        map.put(b, aU);
        return true;
    }
    if (!map.containsKey(a) && map.containsKey(b)){
        int bU = map.get(b);
        map.put(a, bU);
        return true;
    }
}

```

```

    }
    //两个人都有团伙的情况。
    int aU = map.get(a);
    int bU = map.get(b);
    //如果是自己人,那肯定要踢掉,否则成环了
    if(aU == bU) return false;
    //把所有对方的团伙都吃进来
    for (String s : map.keySet()) {
        if (map.get(s) == bU) map.put(s, aU);
    }
    return true;
}
public static void main(String[] args) {
    ArrayList<Connection> connections = new ArrayList<>();
    //下面的图是个苯环, 中间加上了几根, 如果想验证空表, 去掉几根线就行。
    connections.add(new Connection("A","B",6));
    connections.add(new Connection("B","C",4));
    connections.add(new Connection("C","D",5));
    connections.add(new Connection("D","E",8));
    connections.add(new Connection("E","F",2));
    connections.add(new Connection("B","F",10));
    connections.add(new Connection("E","C",9));
    connections.add(new Connection("F","C",7));
    connections.add(new Connection("B","E",3));
    connections.add(new Connection("A","F",16));
    //这里就输出一下看看结果。
    ArrayList<Connection> res = getLowCost(connections);
    for (Connection c : res){
        System.out.println(c.node1 + " -> " + c.node2 + " " + c.cost);
    }
}

```

## Five Highest Scores:

输入是 List<节点>, 节点里面有两个属性学生 id 和分数, 同时保证保证每个学生至少会有 5 个节点(即至少有 5 个分数), 返回一个 map, key 是 id, value 是每个人最高 5 个分数的平均分。

```

import java.util.*;
class Result{
    int id;
    int value;
    public Result(int id, int value){
        this.id = id;
        this.value = value;
    }
}

```

```

public class High_Five {
    public static Map<Integer, Double> getHighFive(Result[] results){
        Map<Integer, Double> map = new HashMap<>();
        //这里 pValue 的命名,就是每个 person 都有哪些 value。
        Map<Integer, ArrayList<Integer>> pValue = new HashMap<>();
        //对照着 ID 把成绩塞给对应的人。
        for (Result res : results){
            int id = res.id;
            if (pValue.containsKey(id)){
                //这里 curL 表示 current List
                ArrayList<Integer> curL = pValue.get(id);
                curL.add(res.value);
                pValue.put(id, curL);
            }
            else {
                ArrayList<Integer> curL = new ArrayList<>();
                curL.add(res.value);
                pValue.put(id, curL);
            }
        }
        for (Integer id : pValue.keySet()){
            ArrayList<Integer> list = pValue.get(id);
            //这里写法有些风骚了,就是懒的重写 comparator
            Collections.sort(list);
            Collections.reverse(list);
            double value = 0;
            for (int k = 0; k < 5; k++){
                value += list.get(k);
            }
            value = value/5.0;
            map.put(id, value);
        }
        return map;
    }
}

```

## Maximum Average SubTree:

就是给一棵多叉树，表示公司内部的下级关系。每个节点表示一个员工，节点包含的成员是他工作了几个月(int)，以及一个下属数组(ArrayList<Node>)。目标就是找到一棵子树，满足：这棵子树所有节点的工作月数的平均数是所有子树中最大的。最后返回这棵子树的根节点。这题补充一点，返回的不能是叶子节点(因为叶子节点没有下属)，一定要是一个有子节点的节点。

不可以用全局变量：把全局变量放在一个长度为一的数组里输入

用全局变量的版本：

```
import java.util.*; //这次差点儿忘了这个
```

```
class Node { //这个是题目给好的
```

```
    int val;
```

```
    ArrayList<Node> children;
```

```
    public Node(int val){
```

```
        this.val = val;
```

```
        children = new ArrayList<Node>();
```

```
    }
```

```
}
```

//这个类是自己写的,要不不好找,两个成员变量分别是当前的总和和人数

```
class SumCount{
```

```
    int sum;
```

```
    int count;
```

```
    public SumCount(int sum, int count){
```

```
        this.sum = sum;
```

```
        this.count = count;
```

```
    }
```

```
}
```

```
public class Company_Tree {
```

```
    //两个全局变量用来找最小的平均值,和对应的节点
```

```
    private static double resAve = Double.MIN_VALUE;
```

```
    private static Node result;
```

```
    public static Node getHighAve(Node root){
```

```
        if (root == null) return null;
```

```
        dfs(root);
```

```
        return result;
```

```
    }
```

```
    //后序遍历递归。
```

```
    private static SumCount dfs(Node root){
```

```
        // 这里必须先把叶子节点刨掉, 注意看我的手法, 其实没什么。
```

```
        if (root.children == null || root.children.size() == 0){
```

```
            return new SumCount(root.val, 1);
```

```
        }
```

```
        //把当前 root 的材料都准备好
```

```
        int curSum = root.val;
```

```
        int curCnt = 1;
```

```
        //注意了这里开始遍历小朋友了
```

```
        for (Node child : root.children) {
```

```
            SumCount cSC = dfs(child);
```

```
            //每次遍历一个都把 sum,count 都加上, 更新
```

```
            curSum += cSC.sum;
```

```
            curCnt += cSC.count;
```

```
        }
```

```
        double curAve = (double) curSum/curCnt;
```

```
//这里看一下最大值要不要更新
if (resAve < curAve){
    resAve = curAve;
    result = root;
}

return new SumCount(curSum,curCnt);
}
}
```