https://www.notion.so/Amazon-OA2-3afaad4e440f4de9af79b7f7a35fe6d8

https://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=717224&page=1#pid14990492

https://algo.monster/problems/amazon_online_assessment_questions

两道题莉蔻都可以找到，荔寇1481 1120

1. remove the distinct product id

1481. Least Number of Unique Integers after K Removals

```java
public int findLeastNumOfUniqueInts(int[] arr, int k) {
    if(k == arr.length){
        return 0;
    }
    Map<Integer, Integer> map = new HashMap<>();
    Integer[] nums = new Integer[arr.length];
    int index = 0;
    for(int n : arr){
        nums[index++] = n;
        map.put(n, map.getOrDefault(n, 0) + 1);
    }
    if(k == 0){
        return map.size();
    }
    Arrays.sort(nums, new Comparator<Integer>(){
       public int compare(Integer a, Integer b){
           int n1 = map.get(a), n2 = map.get(b);
           return (n1 != n2) ? (n1 - n2) : (a - b);
       }
    });
    Set<Integer> hs = new HashSet<>();
    for(int i = k; i < nums.length; i++){
        hs.add(nums[i]);
    }
    return hs.size();
}
```

2. find the root component

1120. Maximum Average Subtree

```java
public class Solution {

    private double res;

    public double maximumAverageSubtree(TreeNode root) {
        dfs(root);
        return res;
    }

    private int[] dfs(TreeNode cur) {
        // 如果是空树，则节点数和总和都是0
        if (cur == null) {
            return new int[]{0, 0};
        }

        int[] left = dfs(cur.left), right = dfs(cur.right);
        int count = left[0] + right[0] + 1, sum = left[1] + right[1] + cur.val;

        // 枚举cur为树根的情形，更新答案
        res = Math.max(res, (double) sum / count);
        return new int[]{count, sum};
    }
}

class TreeNode {
    int val;
    TreeNode left, right;

    public TreeNode(int val) {
        this.val = val;
    }
}
```

1041. Robot Bounded In Circle

```java
public boolean isRobotBounded(String instructions) {
        int x = 0;
        int y = 0;
           //Keep track of directions
        String dir = "North";
    //      N
    // W<----->E
    //      S
```

```
    //Caluculate position
       for(char c : instructions.toCharArray()){
           if(c == 'G'){
               if(dir.equals("North")){y++;}
               else if(dir.equals("East")){x++;}
               else if(dir.equals("South")){y--;}
               else {x--;}
           }
           else if(c == 'L'){
               if(dir.equals("North")){dir = "West";}
               else if(dir.equals("East")){dir = "North";}
               else if(dir.equals("South")){dir = "East";}
               else {dir = "South";}
           }
           else{
               if(dir.equals("North")){dir = "East";}
               else if(dir.equals("East")){dir = "South";}
               else if(dir.equals("South")){dir = "West";}
               else {dir = "North";}
           }
       }
       //Check if calutucated position is starting position
       if(x == 0 && y == 0){
           return true;
       }
          //check if the final faced direction is not North(Strarting
DIrection)
       if(dir.equals("North")){
           return false;
       }
       return true;
   }
```

Optimize Box Weight

Given a list of integers, partition it into two subsets S1 and S2 such that the sum of S1 is greater than that of S2. And also the number of elements in S1 should be minimal.
Return S1 in increasing order.
Notice if more than one subset A exists, return the one with the maximal sum.

Examples:
Input:

nums = [4, 5, 2, 3, 1, 2]
Output:
[4, 5]
Explanation:
We can divide the numbers into two subsets A = [4, 5] and B = [1, 2, 2, 3]. The sum of A is 9 which is greater than the sum of B which is 8. There are other ways to divide but A = [4, 5] is of minimal size of 2.

Input:
nums = [10, 5, 3, 1, 20]
Output:
[20]

Input:
nums = [1, 2, 3, 5, 8]
Output:
[5, 8]

```
// "static void main" must be defined in a public class.
public class OptimizeBox {
    public static void main(String[] args) {
        int[] ans1 = optimize(new int[]{1, 1, 2, 1});
        print("ans1", ans1);
        int[] ans2 = optimize(new int[]{3, 7, 6, 2});
        print("ans2", ans2);
    }
    private static int[] optimize(int[] arr) {
        long arraySum = 0;
        for (int num : arr) {
            arraySum += num;
        }
        Arrays.sort(arr);
        long max = 0;
        int idx = 0;

        while (idx < arr.length && max * 2 < arraySum) {
```

```
            max += arr[idx];
            idx++;
        }

        idx--; // idx now is the first element in box A

        int[] res = new int[arr.length - idx];
        for (int i = 0; i < arr.length - idx; i++) {
            res[i] = arr[idx + i];
        }
        return res;
    }
```

storage optimization

Create a function that takes in a matrix, and an array of rows and an array of columns to be removed from the matrix, and return the size of the biggest cubic space after removing the shelves and columns.

```
biggestStorage(matrix, rows, columns)

      return biggestCubicSize; // i.e. 3x4 = 12
```

solution leetcode: 切蛋糕 cut cake

```
public int maxArea(int h, int w, int[] horizontalCuts, int[] verticalCuts) {
    Arrays.sort(horizontalCuts);
    Arrays.sort(verticalCuts);
    int maxH = horizontalCuts[0];
    int maxW = verticalCuts[0];
    for (int i = 1; i < horizontalCuts.length; i++) {
        maxH = Math.max(maxH, horizontalCuts[i] - horizontalCuts[i - 1]);
    }
    for (int i = 1; i < verticalCuts.length; i++) {
        maxW = Math.max(maxW, verticalCuts[i] - verticalCuts[i - 1]);
    }
    maxH = Math.max(maxH, h - horizontalCuts[horizontalCuts.length - 1]);
    maxW = Math.max(maxW, w - verticalCuts[verticalCuts.length - 1]);
    return (int)((((long)maxH) * maxW) % (1000000007));
}
```

CloudFront Caching

A company owns N warehouses, identified as warehouse[0 to N-1]. The owner would like to measure the maintenance cost. A warehouse will be able to share the stock with other connected warehouses, making it less costly to restock.
The method to evaluate the maintenance cost is

I. If a warehouse is not connected to any others, its maintenance cost is 1.
II. If multiple warehouses are connected, the total maintenance cost of the group of connected warehouses will be the ceiling of the square root of K, where K is the number of warehouses in the group.
III. A warehouse connected to any of the warehouses in a group will be able to share stock with all in the group. For example, if warehouse 0 and warehouse 1 are connected, and warehouse 1 and warehouse 2 are connected, consider 0, 1 and 2 in the same group.
IV. The total maintenance cost is the sum of all costs.

Given the number of warehouses N and a 2d array, where every subarray is a pair of connected warehouses, build a function that return the total maintenance cost.

Function
int costEvaluation(int n, int[][] connections)

Examples
Input
n = 4
connections = [[0, 2], [1, 2]]

Output
3

Explanation
Warehouses 0, 1, 2 are in the same group, the cost is ceiling(sqrt(3)) = 2.
Warehouse 3 costs 1.
Total cost is 2 + 1 = 3.

Input
n = 10
connections = [[2, 6], [3, 5], [0, 1], [2, 9], [5, 6]]

Output
8

Explanation
Warehouses 0, 1 are in the same group, the cost is ceiling(sqrt(2)) = 2.
Warehouses 2, 3, 5, 6, 9 are in the same group, the cost is ceiling(sqrt(5)) = 3.
Warehouse 4 costs 1.
Warehouse 7 costs 1.

solution:

```java
public static int cost(int n, int[][] arr) {
    if (n < 2) return n;
    // build graph
    HashMap<Integer, List<Integer>> graph = new HashMap<>();
    for (int i = 0; i < n; i++) {
        graph.put(i, new ArrayList<>());
    }
    for (int[] a : arr) {
       int x = a[0];
       int y = a[1];
       graph.get(x).add(y);
       graph.get(y).add(x);
    }

    int island = 0;
    int res = 0;
    HashSet<Integer> visited = new HashSet<>();
    for (Map.Entry<Integer, List<Integer>> e : graph.entrySet()) {
        if (e.getValue().size() == 0) {
            island++;
        }else {
            int curK = e.getKey();
            if (!visited.contains(curK)) {
                HashSet<Integer> curGroup = new HashSet<>();
                Queue<Integer> q = new LinkedList<>();
                q.add(curK);
                curGroup.add(curK);
                while (!q.isEmpty()) {
                    int cur = q.poll();
                    List<Integer> curV = graph.get(cur);
                    for (int i = 0; i < curV.size(); i++) {
                        int iCurV = curV.get(i);
                        if (!curGroup.contains(iCurV) && !visited.contains(iCurV)) {
                            curGroup.add(iCurV);
                            visited.add(iCurV);
                            q.add(iCurV);
                        }
                    }
                }

                res += (int)Math.ceil(Math.sqrt(curGroup.size()));
            }
        }
    }
    return res + island;
}
```

union find solution:

```java
    public static int test(int n, int[][] arr) {
        UnionFind uf = new UnionFind(n);
        for (int i = 0; i < arr.length; i++) {
            uf.union(arr[i][0], arr[i][1]);
        }
        int res = 0;
        for (int i  = 0; i < n; i++) {
            if (uf.parent[i] == i) {
                res += Math.ceil(Math.sqrt(uf.rank[i]));
            }
        }
        return res;
    }
    static class UnionFind {
        int[] parent;
        int[] rank;
        int size;
        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            size = n;
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 1;
            }
        }

        int find(int x) {
            int root = x;
            if (parent[x] != x) {
                root = find(parent[x]);
            }
            return root;
        }

        void union(int a, int b) {
            int rootA = find(a);
            int rootB = find(b);
            if (rootA == rootB) return;
            if (rank[rootA] > rank[rootB]) {
                parent[rootB] = rootA;
                rank[rootA] += rank[rootB];
            } else {
                parent[rootA] = rootB;
                rank[rootB] += rank[rootA];
            }
        }
    }
}
```

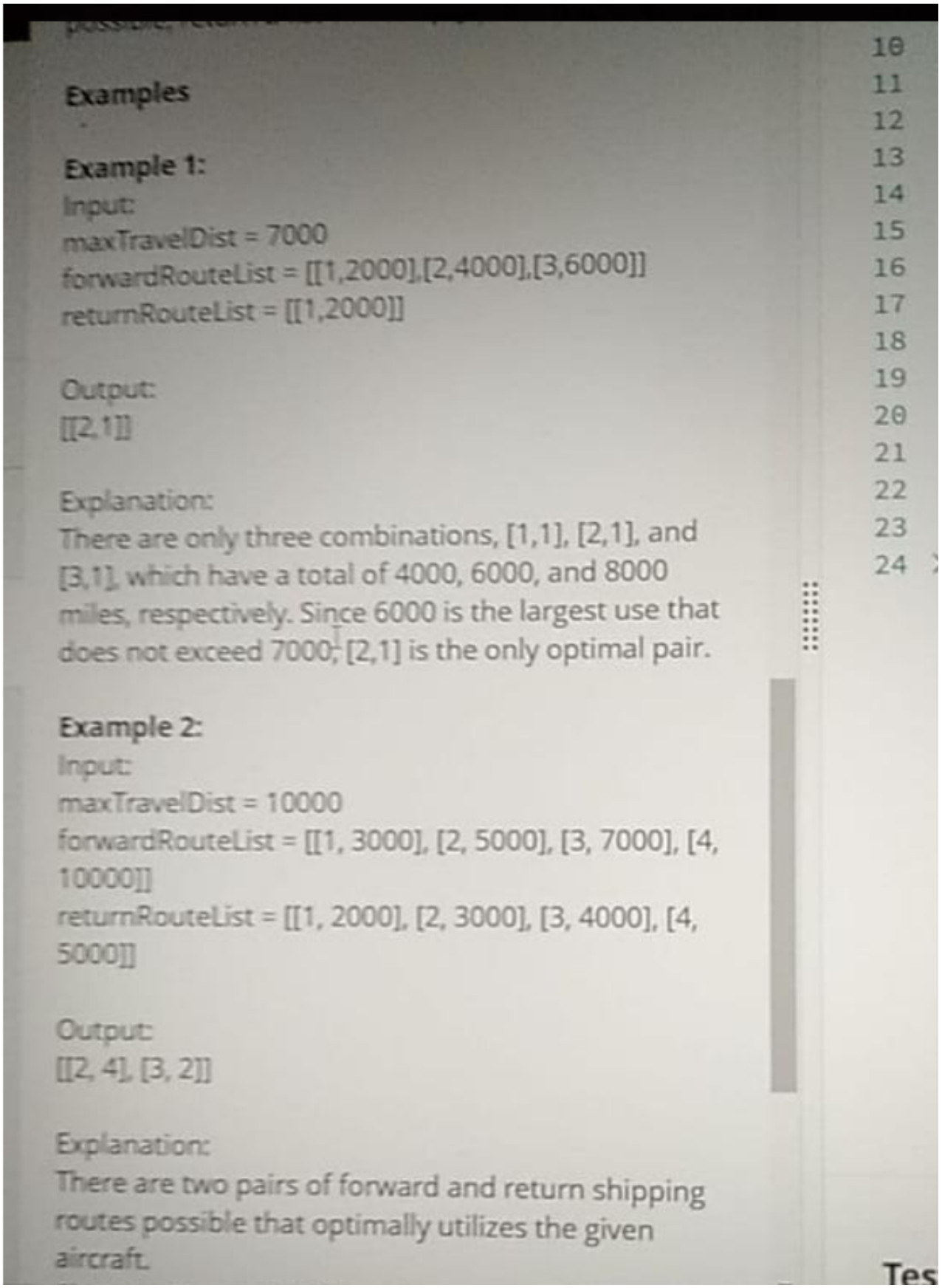
第一题 利口舅其散 套了个算餐馆距离壳子 restaurant

973. K Closest Points to Origin

```
public int[][] kClosest(int[][] points, int k) {
    PriorityQueue<Distance> pq = new PriorityQueue<>((a, b)->{
       if (a.distance < b.distance) {
           return -1;
       }  else if (a.distance > b.distance) {
           return 1;
       } else {
           return 0;
       }
    });

    for (int[] p: points) {
        double dis = (double)(p[0] * p[0] + p[1] * p[1]);
        pq.add(new Distance(p, dis));
    }

    int[][] res = new int[k][2];
    while (k > 0) {
        res[k-1] = pq.poll().p;
        k--;
    }
    return res;

}

private class Distance {
    int[] p;
    double distance;
    Distance(int[] _p, double _distance) {
        p = _p;
        distance = _distance;
    }
}
```

第二题Prime Air Route换了个壳子，sort后two pointer，处理好重复的地方。

prime air route



Examples

Example 1:
Input:
maxTravelDist = 7000
forwardRouteList = [[1,2000],[2,4000],[3,6000]]
returnRouteList = [[1,2000]]

Output:
[[2,1]]

Explanation:
There are only three combinations, [1,1], [2,1], and [3,1], which have a total of 4000, 6000, and 8000 miles, respectively. Since 6000 is the largest use that does not exceed 7000, [2,1] is the only optimal pair.

Example 2:
Input:
maxTravelDist = 10000
forwardRouteList = [[1, 3000], [2, 5000], [3, 7000], [4, 10000]]
returnRouteList = [[1, 2000], [2, 3000], [3, 4000], [4, 5000]]

Output:
[[2, 4], [3, 2]]

Explanation:
There are two pairs of forward and return shipping routes possible that optimally utilizes the given aircraft.

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

Tes

solution:

```java
public static List<int[]> getPair(int[][] a, int[][] b, int target) {
    Arrays.sort(a, Comparator.comparingDouble(o -> o[1]));
    Arrays.sort(b, Comparator.comparingDouble(o -> o[1]));
    List<int[]> res = new ArrayList<>();
    int max = -1;
    int aInd = 0;
    int bInd = b.length - 1;

    while (aInd < a.length && bInd >= 0) {
        //System.out.println("aid: " + aInd);
        //System.out.println("bid: " + bInd);
        int aVal = a[aInd][1];
        int bVal = b[bInd][1];
        int sum = aVal + bVal;
        //System.out.println("sum: " + sum);

        if (sum > target) {
            bInd--;
        } else {
            if (sum >= max) {
                if (sum > max) {
                    res.clear();
                    max = sum;
                }
                res.add(new int[]{a[aInd][0], b[bInd][0]});
                int index = bInd - 1;
                while (index >= 0 && b[index][1] == b[index + 1][1]) {
                     res.add(new int[]{a[aInd][0], b[index][0]});
                     index--;
                }
            }
            aInd++;
        }
    }
    return res;
}
```

solution2：

```java
public List<int[]> getPair(final int[][] A, final int[][] B, final int target) {
  int sum = -1;
  final List<int[]> result = new ArrayList<>();
  for (int[] a : A) {
    for (int[] b : B) {
      final int s = a[1] + b[1];
      if (s > target || s < sum) {
        continue;
      }
      if (s > sum) {
        result.clear();
        sum = s;
      }
      result.add(new int[]{a[0], b[0]});
    }
  }
  return result;
}
```

solution3：

```java
List<int[]> getOptimalRoute(int maxTravelDist,int[][] forwardRoute,int[][] returnRoute) {  // 按旅程从大到小排列
  Arrays.sort(forwardRoute, (a,b)->b[1]-a[1]);
  TreeMap<Integer,Stack<Integer>> map=new  TreeMap<>();
  for(var rr:returnRoute) {
    // 按旅程分组
    var stack=map.getOrDefault(rr[1],new LinkedList<>());
    stack.push(rr[0]);
    map.put(rr[1],stack);
  }
  var dest=new ArrayList<int[]>();
  for(var fr:forwardRoute) {
    // 贪心，永远取和最接近最大里程的组
    var re=map.floorEntry(maxTravelDist-fr[1]);
   // 如果没有合适的，说明最小值和这个值的和都大于里程数了
    if(re==null) continue;
    // 随便取一个
    var stack=re.getValue();
    dest.add(new int[] {fr[0],stack.pop()});
    // 如果空了删除主键
    if (stack.isEmpty()) map.remove(re.getKey());
  }
  return dest;
}
```

shopping options 类似lc454

Disappeared shopping options question, retrieved from google cache

A customer wants to buy a pair of jeans, a pair of shoes, a skirt, and a top but has a limited budget in dollars. Given different pricing options for each product, determine how many options our customer has to buy 1 of each product. You cannot spend more money than the budgeted amount.

Example

priceOfJeans = [2, 3]

priceOfShoes = [4]

priceOfSkirts = [2, 3]

priceOfTops = [1, 2]

budgeted = 10

The customer must buy shoes for 4 dollars since there is only one option. This leaves 6 dollars to spend on the other 3 items. Combinations of prices paid for jeans, skirts, and tops respectively that add up to 6 dollars or less are [2, 2, 2], [2, 2, 1], [3, 2, 1], [2, 3, 1]. There are 4 ways the customer can purchase all 4 items.

Function Description

Complete the getNumberOfOptions function in the editor below. The function must return an integer which represents the number of options present to buy the four items.

getNumberOfOptions has 5 parameters:

int[] priceOfJeans: An integer array, which contains the prices of the pairs of jeans available.

int[] priceOfShoes: An integer array, which contains the prices of the pairs of shoes available.

int[] priceOfSkirts: An integer array, which contains the prices of the skirts available.

int[] priceOfTops: An integer array, which contains the prices of the tops available.

int dollars: the total number of dollars available to shop with.

Constraints

1 ≤ a, b, c, d ≤ 103

1 ≤ dollars ≤ 109

1 ≤ price of each item ≤ 109

Note: a, b, c and d are the sizes of the four price arrays

solution:

```java
public static int getNumberOfOptions(
    List<Integer> priceOfJeans,
    List<Integer> priceOfShoes,
    List<Integer> priceOfSkirts,
    List<Integer> priceOfTops,
    int dollars
) {
    // combine 2 lists together and sort
    List<Integer> a = combine(priceOfJeans, priceOfShoes, dollars);
    List<Integer> b = combine(priceOfSkirts, priceOfTops, dollars);

    // search
    int left = 0;
    int right = b.size() - 1;
    int counts = 0;

    while (left < a.size() && right >= 0) {
        //System.out.println("a.get(left): " + a.get(left) + " b.get(right): " + b.get
        int sum = a.get(left) + b.get(right);
        if (sum <= dollars) {
            counts += right + 1;
            left++;
        } else {
            right--;
        }
    }
    return counts;
}
private static List<Integer> combine(List<Integer> a, List<Integer> b, int target) {
    List<Integer> sumList = new ArrayList<Integer>();
    for (int i : a) {
        for (int j : b) {
            int sum = i+ j;
            if (sum <= target) {
                sumList.add(sum);
            }
        }
    }
    sumList.sort((x, y) -> x - y);
    return sumList;
}
```

Music Runtime

Amazon music is working on a new feature to pair songs together to play while on bus.The goal of this feature is to select two songs from the list that will end exactly 30

secondsbefore the listener reaches their stop.You are tasked with writing the method that selects the songs from a list. Each song is assigned a unique id, numbered from 0 to N-1.

Assumptions:

1. You will pick exactly 2 songs.
2. you cannot pick the same song twice.
3. if you have multiple pais with the same total time, select the pair with the longest song.
4. there are atleast 2 songs available.

write an alorithm to return ID's of the 2 songs whose combined runtime will finish exactly 30 seconds before the bus arrives, keeping the original order. If no such pair is possible, return a pair with <-1,-1>.

input:

rideDuration = 90

songDuration = {1,10,25,35,60}

output:
[2,3]

solution:
You should add a condition that map.get(complement) != i (or it may return the same element twice.)

```java
private static int[] test(int rideDuration, int[] songDuration){
    int target = rideDuration - 30;
    int max = Integer.MIN_VALUE;
    int[] res = new int[]{-1, -1};
    if (rideDuration <= 30) return res;
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < songDuration.length; i++){
        if (map.containsKey(target - songDuration[i])){
            if (songDuration[i] > max || songDuration[map.get(target - songDuration[i])] > max) {
                res[0] = map.get(target - songDuration[i]);
                res[1] = i;
                max = Math.max(songDuration[i], songDuration[map.get(target - songDuration[i])]);
            }
        }
        map.put(songDuration[i], i);
    }
    return res;
}
```

Treasure Island

You have a map that marks the location of a treasure island. Some of the map area has jagged rocks and dangerous reefs. Other areas are safe to sail in. There are other explorers trying to find the treasure. So you must figure out a shortest route to the treasure island.

Assume the map area is a two dimensional grid, represented by a matrix of characters. You must start from the top-left corner of the map and can move one block up, down, left or right at a time. The treasure island is marked as `X` in a block of the matrix. `X` will not be at the top-left corner. Any block with dangerous rocks or reefs will be marked as `D`. You must not enter dangerous blocks. You cannot leave the map area. Other areas `O` are safe to sail in. The top-left corner is always safe. Output the minimum number of steps to get to the treasure.

Example:

```
Input:
[['O', 'O', 'O', 'O'],
 ['D', 'O', 'D', 'O'],
 ['O', 'O', 'O', 'O'],
 ['X', 'D', 'D', 'O']]

Output: 5
```

`Explanation: Route is (0, 0), (0, 1), (1, 1), (2, 1), (2, 0), (3, 0) The minimum route takes 5 steps.`

Solution:

BFS

```
private static final int[][] DIRS = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

public static int minSteps(char[][] grid) {
    Queue<Point> q = new ArrayDeque<>();
    q.add(new Point(0, 0));
    grid[0][0] = 'D'; // mark as visited
    for (int steps = 1; !q.isEmpty(); steps++) {
        for (int sz = q.size(); sz > 0; sz--) {
            Point p = q.poll();

            for (int[] dir : DIRS) {
                int r = p.r + dir[0];
                int c = p.c + dir[1];

                if (isSafe(grid, r, c)) {
                    if (grid[r][c] == 'X') return steps;
                    grid[r][c] = 'D';
                    q.add(new Point(r, c));
                }
            }
        }
    }
    return -1;
}

private static boolean isSafe(char[][] grid, int r, int c) {
    return r >= 0 && r < grid.length && c >= 0 && c < grid[0].length && grid[r][c] != 'D';
}

private static class Point {
    int r, c;
    Point(int r, int c) {
        this.r = r;
        this.c = c;
    }
}
```

treasure island 2

You have a map that marks the locations of treasure islands. Some of the map area has jagged rocks and dangerous reefs. Other areas are safe to sail in. There are other

explorers trying to find the treasure. So you must figure out a shortest route to one of the treasure islands.

Assume the map area is a two dimensional grid, represented by a matrix of characters. You must start from one of the starting point (marked as `S`) of the map and can move one block up, down, left or right at a time. The treasure island is marked as `X`. Any block with dangerous rocks or reefs will be marked as `D`. You must not enter dangerous blocks. You cannot leave the map area. Other areas `O` are safe to sail in. Output the minimum number of steps to get to any of the treasure islands.

Example:

```
Input:
[['S', 'O', 'O', 'S', 'S'],
 ['D', 'O', 'D', 'O', 'D'],
 ['O', 'O', 'O', 'O', 'X'],
 ['X', 'D', 'D', 'O', 'O'],
 ['X', 'D', 'D', 'D', 'O']]

Output: 3
Explanation:
You can start from (0,0), (0, 3) or (0, 4). The treasure locations are (2, 4) (3, 0) and (4, 0). Here the shortest route is (0, 3), (1, 3), (2, 3), (2, 4).
```

solution

```
final int[][] dirs = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
public int shortestPath(char[][] islands){
    if(islands.length == 0 || islands[0].length == 0) return -1;
    int R = islands.length, C = islands[0].length;
    Queue<int[]> queue = new LinkedList<>();
    int steps = 1;
    // add all sources to queue and set 'visited'.
    for(int i = 0; i < R; ++i){
        for(int j = 0; j < C; ++j){
            if(islands[i][j] == 'S'){
                queue.add(new int[]{i, j}); islands[i][j] = 'D';
            }
        }
    }
    while(!queue.isEmpty()){
        int size = queue.size();
        for(int i = 0; i < size; ++i){
            int[] pos = queue.poll();
            for(int[] dir: dirs){
                int x = pos[0] + dir[0], y = pos[1] + dir[1];
                if(x < 0 || x >= R || y < 0 || y >= C || islands[x][y] == 'D') continue;
                if(islands[x][y] == 'E') return steps;
                queue.add(new int[]{x, y});
                islands[x][y] = 'D';
            }
        }
        ++steps;
    }
    return -1;
}
```

items in container

Given a string *s* consisting of items as "*" and closed compartments as an open and close "|", an array of starting indices startIndices, and an array of ending indices endIndices, determine the number of items in closed compartments within the substring between the two indices, inclusive.

- An item is represented as an asterisk *
- A compartment is represented as a pair of pipes | that may or may not have items between them.

Example:

s = '|**|*|*'

startIndices = [1,1]

endIndices = [5,6]

The String has a total 2 closed compartments, one with 2 items and one with 1 item. For the first par of indices, (1,5), the substring is '|**|*'. There are 2 items in a compartment.

For the second pair of indices, (1,6), the substring is '|**|*|' and there 2+1=3 items in compartments.

Both of the answers are returned in an array. [2,3].

solution:

turnstile

9. Turnstile

A university has exactly one turnstile. It can be used either as an exit or an entrance. Unfortunately, sometimes many people want to pass through the turnstile and their directions can be different. The i^{th} person comes to the turnstile at *time[i]* and wants to either exit the university if *direction[i] = 1* or enter the university if *direction[i] = 0*. People form *2* queues, one to exit and one to enter. They are ordered by the time when they came to the turnstile and, if the times are equal, by their indices.

If some person wants to enter the university and another person wants to leave the university at the same moment, there are three cases:

- If in the previous second the turnstile was not used (maybe it was used before, but not at the previous second), then the person who wants to leave goes first.
- If in the previous second the turnstile was used as an exit, then the person who wants to leave goes first.
- If in the previous second the turnstile was used as an entrance, then the person who wants to enter goes first.

Passing through the turnstile takes *1* second.

For each person, find the time when they will pass through the turnstile.

Function Description

Complete the function *getTimes* in the editor below.

getTimes has the following parameters:

int time[n]: an array of *n* integers where the value at index *i* is the time in seconds when the i^{th} person will come

Function Description

Complete the function *getTimes* in the editor below.

getTimes has the following parameters:

int time[n]: an array of *n* integers where the value at index *i* is the time in seconds when the i^{th} person will come to the turnstile

int direction[n]: an array of *n* integers where the value at index *i* is the direction of the i^{th} person

Returns:

int[n]: an array of *n* integers where the value at index *i* is the time when the i^{th} person will pass the turnstile

Constraints

- $1 \le n \le 10^5$
- $0 \le time[i] \le 10^9$ for $0 \le i \le n - 1$
- $time[i] \le time[i + 1]$ for $0 \le i \le n - 2$
- $0 \le direction[i] \le 1$ for $0 \le i \le n - 1$

▶ **Input Format For Custom Testing**

▼ **Sample Case 0**

Sample Input 0

▼ **Sample Case 0**

Sample Input 0

```
STDIN    Function
-----    --------
4    →   time[] size n = 4
0    →   time = [0, 0, 1, 5]
0
1
5
4    →   direction[] size n = 4
0    →   direction = [0, 1, 1, 0]
1
1
0
```

Sample Output 0

```
2
0
1
5
```

solution:

```java
public class Turnstile {
    public static int[] getTimes(int[] time, int[] direction) {
        Queue<Integer> enters = new LinkedList<Integer>();
        Queue<Integer> exits = new LinkedList<Integer>();
        int n = time.length;
        for(int i = 0; i < n; i++) {
            Queue<Integer> q = direction[i] == 1 ? exits : enters;
            q.offer(i);
        }

        int[] result = new int[n];
        int lastTime = -2;
        Queue<Integer> lastQ = exits;
        while(enters.size() > 0 && exits.size() > 0) {
            int currentTime = lastTime + 1;
            int peekEnterTime = time[enters.peek()];
            int peekExitTime = time[exits.peek()];
            Queue<Integer> q;
            if (currentTime < peekEnterTime && currentTime < peekExitTime) {
                // The turnstile was not used
                // Take whoever has the earliest time or
                // if enter == exit, take exit
                q = (peekEnterTime < peekExitTime) ? enters : exits;
                int personIdx = q.poll();
                result[personIdx] = time[personIdx];
                lastTime = time[personIdx]; // time
                lastQ = q;
            } else {
                // Turnstile was used last second
                if (currentTime >= peekEnterTime && currentTime >= peekExitTime) {
                    // Have people waiting at both ends
                    // Prioritize last direction
                    q = lastQ;
                } else {
                    // current >= enters.peek() || current >= exits.peek()
                    q = currentTime >= peekEnterTime ? enters : exits; // take whatever that's queuing
                }
                int personIdx = q.poll();
                result[personIdx] = currentTime;
                lastTime = currentTime; // time
                lastQ = q;
            }
        }

        Queue<Integer> q = enters.size() > 0 ? enters : exits;
        while(q.size() > 0) {
            int currentTime = lastTime + 1;
            int personIdx = q.poll();
            if (currentTime < time[personIdx]) {
                // The turnstile was not used
                currentTime = time[personIdx];
            }

            result[personIdx] = currentTime;
            lastTime = currentTime; // time
        }

        return result;
    }
```

Optimal Utilization

Given 2 lists `a` and `b`. Each element is a pair of integers where the first integer represents the unique id and the second integer represents a value. Your task is to find an element from `a` and an element form `b` such that the sum of their values is less or equal to `target` and as close to `target` as possible. Return a list of ids of selected elements. If no pair is possible, return an empty list.

Example 1:

```
Input:
a = [[1, 2], [2, 4], [3, 6]]
b = [[1, 2]]
target = 7

Output: [[2, 1]]

Explanation:
There are only three combinations [1, 1], [2, 1], and [3, 1], which have a total sum of 4, 6 and 8, respectively.
Since 6 is the largest sum that does not exceed 7, [2, 1] is the optimal pair.
```

Example 2:

```
Input:
a = [[1, 3], [2, 5], [3, 7], [4, 10]]
b = [[1, 2], [2, 3], [3, 4], [4, 5]]
target = 10

Output: [[2, 4], [3, 2]]

Explanation:
There are two pairs possible. Element with id = 2 from the list `a` has a value 5, and element with id = 4 from the list `b` also has a value 5.
Combined, they add up to 10. Similarily, element with id = 3 from `a` has a value 7, and element with id = 2 from `b` has a value 3.
These also add up to 10. Therefore, the optimal pairs are [2, 4] and [3, 2].
```

solution:

```java
private List<int[]> getPairs(List<int[]> a, List<int[]> b, int target) {
    Collections.sort(a, (i,j) -> i[1] - j[1]);
    Collections.sort(b, (i,j) -> i[1] - j[1]);
    List<int[]> result = new ArrayList<>();
    int max = Integer.MIN_VALUE;
    int m = a.size();
    int n = b.size();
    int i =0;
    int j =n-1;
    while(i<m && j >= 0) {
        int sum = a.get(i)[1] + b.get(j)[1];
        if(sum > target) {
             --j;
        } else {
            if(max <= sum) {
                if(max < sum) {
                    max = sum;
                    result.clear();
                }
                result.add(new int[]{a.get(i)[0], b.get(j)[0]});
                int index = j-1;
                while(index >=0 && b.get(index)[1] == b.get(index+1)[1]) {
                     result.add(new int[]{a.get(i)[0], b.get(index--)[0]});
                }
            }
            ++i;
        }
    }
    return result;
}
```

描述是说把一个array 变为sorted 需要多少次swap 实质和315是一样的

用的就睡mergesort，testcase 全过

Given an array and a sorting algorithm, the sorting algorithm will do a selection swap. Find the number of swaps to sort the array.

Example 1:

Input: [5, 4, 1, 2]

Output: 5

Explanation:

Swap index 0 with 1 to form the sorted array [4, 5, 1, 2].

Swap index 0 with 2 to form the sorted array [1, 5, 4, 2].

Swap index 1 with 2 to form the sorted array [1, 4, 5, 2].

Swap index 1 with 3 to form the sorted array [1, 2, 5, 4].

Swap index 2 with 3 to form the sorted array [1, 2, 4, 5].

solution：

https://www.geeksforgeeks.org/number-swaps-sort-adjacent-swapping-allowed/

https://www.geeksforgeeks.org/counting-inversions/

```java
public static int cost(int[] arr) {
  HashMap<Integer, Integer> map = new HashMap<>();
  int[] sortedArr = new int[arr.length];
  for (int i = 0; i < arr.length; i++) {
      map.put(arr[i],i);
      sortedArr[i] = arr[i];
  }

  Arrays.sort(sortedArr);

  int res = 0;
  for (int i = 0; i < sortedArr.length; i++) {
      int targetVal = sortedArr[i];
      int targetIndex = map.get(targetVal);
      if (i != targetIndex) {
          res++;
          int valToSwap = arr[i];
          map.put(valToSwap, targetIndex);
          int temp = arr[i];
          arr[i] = arr[targetIndex];
          arr[targetIndex] = temp;
      }
  }
  return res;
}
```

prime order prioritization，就是给一个list of amazon orders，要根据是否是prime进行排序，sort by lexicographical, non prime order remain the same order.

跟lc937几乎一样

937. Reorder Data in Log Files

You are given an array of logs. Each log is a space-delimited string of words, where the first word is the identifier.

There are two types of logs:

- Letter-logs: All words (except the identifier) consist of lowercase English letters.
- Digit-logs: All words (except the identifier) consist of digits.

Reorder these logs so that:

1. The letter-logs come before all digit-logs.
2. The letter-logs are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.
3. The digit-logs maintain their relative ordering.

Return *the final order of the logs*.

Example 1:

```
Input: logs = ["dig1 8 1 5 1","let1 art can","dig2 3 6","let2 own kit dig","let3 art zero"]
Output: ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]
Explanation:
The letter-log contents are all different, so their ordering is "art can", "art zero", "own kit dig".
The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".
```

Example 2:

```
Input: logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]
Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]
```

solution：

```java
public String[] reorderLogFiles(String[] logs) {

    Comparator<String> myComp = new Comparator<String>() {
        @Override
        public int compare(String log1, String log2) {
            // split each log into two parts: <identifier, content>
            String[] split1 = log1.split(" ", 2);
            String[] split2 = log2.split(" ", 2);

            boolean isDigit1 = Character.isDigit(split1[1].charAt(0));
            boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

            // case 1). both logs are letter-logs
            if (!isDigit1 && !isDigit2) {
                // first compare the content
                int cmp = split1[1].compareTo(split2[1]);
                if (cmp != 0)
                    return cmp;
                // logs of same content, compare the identifiers
                return split1[0].compareTo(split2[0]);
            }

            // case 2). one of logs is digit-log
            if (!isDigit1 && isDigit2)
                // the letter-log comes before digit-logs
                return -1;
            else if (isDigit1 && !isDigit2)
                return 1;
            else
                // case 3). both logs are digit-log
                return 0;
        }
    };

    Arrays.sort(logs, myComp);
    return logs;
}
```

Transaction logs

A Company parses logs of online store user transactions/activity to flag fraudulent activity.

The log file is represented as an Array of arrays. The arrays consist of the following data:

[<# of transactions>]

For example:

[345366 89921 45]

Note: the data is space delimited

So, the log data would look like:

[

[345366 89921 45],

[029323 38239 23]

...

]

Write a function to parse the log data to find distinct users that meet or cross a certain threshold.

The function will take in 2 inputs:

logData: Log data in form an array of arrays

threshold: threshold as an integer

Output:

It should be an array of userids that are sorted.

If same userid appears in the transaction as userid1 and userid2, it should count as one occurrence, not two.

Example:

Input:

logData:

[

[345366 89921 45],

[029323 38239 23],

[38239 345366 15],

[029323 38239 77],

[345366 38239 23],

[029323 345366 13],

[38239 38239 23]

...

]

threshold: 3

Output: [345366 , 38239, 029323]

Explanation:

Given the following counts of userids, there are only 3 userids that meet or exceed the threshold of 3.

345366 -4 , 38239 -5, 029323-3, 89921-1

```java
static List<String> processLogs(List<String> logs, int threshold) {
    Map<String, Integer> map = new HashMap<>();
    for (String logLine : logs) {
        String[] log = logLine.split(" ");
        map.put(log[0], map.getOrDefault(log[0], 0) + 1);
        if (log[0] != log[1]) {
            map.put(log[1], map.getOrDefault(log[1], 0) + 1);
        }
    }

    List<String> userIds = new ArrayList<>();
    for (Map.Entry<String, Integer> entry : map.entrySet()) {
        if (entry.getValue() >= threshold) {
            userIds.add(entry.getKey());
        }
    }

    Collections.sort(userIds,new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
            return Integer.parseInt(s1) - Integer.parseInt(s2);
        }
    });

    return userIds;
}
```

AMAZON MUSIC PAIRS 60min

Amazon Music is working on a new “community radio station” feature which will allow users to iteratively populate

the playlist with their desired songs. Considering this radio station will also have other scheduled shows to be

aired, and to make sure the community soundtrack will not run over other scheduled shows, the user-submitted songs

will be organized in full-minute blocks. Users can choose any songs they want to add to the community list, but

only in pairs of songs with durations that add up to a multiple of 60 seconds (e.g. 60, 120, 180).

As an attempt to insist on the highest standards and avoid this additional burden on users, Amazon will let them

submit any number of songs they want, with any duration, and will handle this 60-second matching internally. Once

the user submits their list, given a list of song durations, calculate the total number of distinct song pairs that

can be chosen by Amazon Music.

A better way can reach O(N).

We just need to get the remainders of each element, for example, [60, 60, 60] will becomes [0, 0, 0]. Then we know there are 3 elements that can be divided by 60. So we just need to pick 2 of them to form a pair, so there will be n*(n-1)/2 pairs.

In other cases, if we have x with count of a and (60 – x) with count of b, then we can form a*b pairs.

So in summary:

1): get the remainders of each elements;

2): count the number of each remainders;

3): analyze the counts to find the valid pair

The time complexity is O(N), and space complexity is O(N).

See the code below:

```
int findPair(vector<int>& nums) {
int res = 0;
vector<int> cts(60, 0);
for(auto &a : nums) ++cts[a%60];
for(int i=1; i<30; ++i) res += cts[i]*cts[60-i];
res += cts[0]*(cts[0]-1)/2 + cts[30]*(cts[30]-1)/2;
return res;
}
```

same thing with java:

```java
public int numPairsDivisibleBy60(int[] time) {
    int res = 0;
    int[] rem = new int[60];
    for (int t : time) rem[t % 60]++;
    for (int i = 1; i < 30; i++) {
        res += rem[i] * rem[60 - i];
    }
    res += rem[0] * (rem[0] - 1) / 2 + rem[30] * (rem[30] - 1) /2;
    return res;
}
```

space is o1 not on

subfiles into a single file

Write an Algorithm to output the minimum possible time to merge the given N subfiles into a single file

Input: The input to the function/method consists of two arguments:

numOfSubFiles: an integer representing the number of subfiles;

files: a list of integers representing the size of the compressed subfiles

Output: Return an integer representating the minimum time required to merge all the subfiles

Constraints::

2 <= numOfSubFiles <= 10^6

1 <= files <= 10^6

Example:

input:

numOfSubFiles = 4

files = [4,8,6,12]

Output: 58

Explanation:

The optimal way to merge subfiles is as follows:

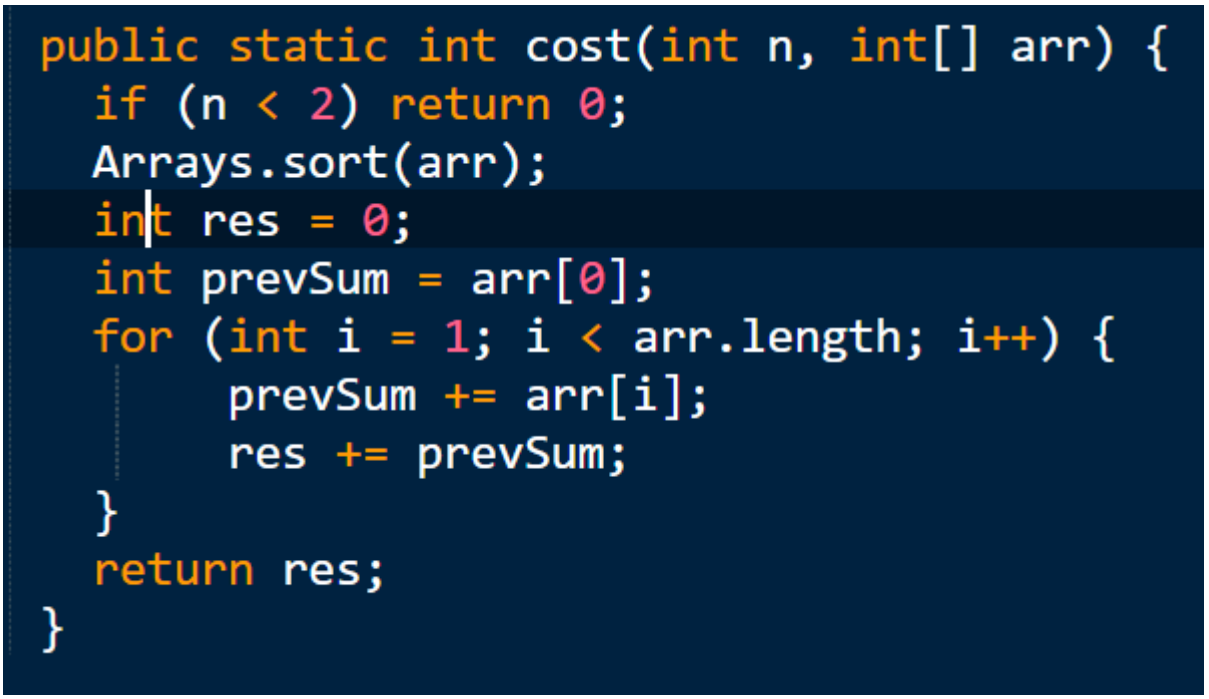
Step 1: Merge the files of size 4 and 6 (time required is 10). Size of subfiles after merging. [8,10,12]

Step 2: Merge the files of size 8 and 10 (time required is 18). Size of subfiles after merging. [18,12]

Step 3: Merge the files of size 18 and 12 (time required is 30)

Total time required to merge the file is 10 + 18 + 30 = 58.

solution:



```
public static int cost(int n, int[] arr) {
  if (n < 2) return 0;
  Arrays.sort(arr);
  int res = 0;
  int prevSum = arr[0];
  for (int i = 1; i < arr.length; i++) {
      prevSum += arr[i];
      res += prevSum;
  }
  return res;
}
```

pq solution:

```java
public int connectSticks(int[] sticks) {
    if(sticks == null || sticks.length < 2){
        return 0;
    }

    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for(int num : sticks){
        minHeap.add(num);
    }

    int res = 0;
    while(minHeap.size() > 1){
        int merge = minHeap.poll() + minHeap.poll();
        res += merge;
        minHeap.add(merge);
    }

    return res;
}
```

time nlogn, space n

shopping patterns

```java
private Set<Integer>[] build(final int n, final List<Integer[]> edges) {
  final Set<Integer>[] graph = new HashSet[n + 1];
  for (var edge : edges) {
    // instantiate set if not done yet
    if (null == graph[edge[0]]) {
      graph[edge[0]] = new HashSet<>();
    }
    if (null == graph[edge[1]]) {
      graph[edge[1]] = new HashSet<>();
    }
    // undirected connection
    graph[edge[0]].add(edge[1]);
    graph[edge[1]].add(edge[0]);
  }
  return graph;
}
```

```java
public int getMinScore(final int n, final List<Integer[]> edges) {
  final Set<Integer>[] G = this.build(n, edges);
  int result = Integer.MAX_VALUE;
  for (int i = 1; i < G.length; ++i) {
    // if node that has trio potential
    if (G[i].size() < 2) {
      continue;
    }
    // it has trio potential
    // verify if it is indeed a trio
    final var neighbours = new ArrayList<>(G[i]);
    // get one neighbour
    for (int j = 0; j < neighbours.size(); ++j) {
      // see if it actually connects with other neighbour
      final var a = neighbours.get(j);
      for (int k = j + 1; j < neighbours.size(); ++j) {
        // if our neighbour j does not connect with neighbour k
        final var b = neighbours.get(k);
        if (!G[a].contains(b)) {
          // this is not a trio
          continue;
        }
        // now verified it is a true trio, time to calculate the cost
        // each node would connect to 2 other nodes, so we have 6 invalid edge needs to be removed
        final int cost = G[i].size() + G[a].size() + G[b].size() - 6;
        result = Math.min(result, cost);
      }
    }
  }
  return result == Integer.MAX_VALUE ? -1 : result;
}
```

utilization checks 25 60

```java
public int finalInstances(int instances, final int[] averageUtil) {
  for (int i = 0; i < averageUtil.length; ++i) {
    final int value = averageUtil[i];
    if (25 <= value && value <= 60) {
      continue;
    }
    if (25 > value) {
      if (1 == instances) {
        continue;
      }
      instances = Math.max((int) Math.ceil(1.0 * instances / 2), 1);
    } else {
      if (200000000 < instances * 2) {
        continue;
      }
      instances = Math.min(instances * 2, 200000000);
    }
    i += 10;
  }
  return instances;
}
```

min total container size move min

Your company is relocating its office, and employees have been asked to pack their belongings into boxes.

As movers, you can move one truckload a day. The truck size needed for day `i` is the maximum box size moved on that day.

There are `N` boxes and `k` days, where the number of boxes is greater than or equal to the number of days. You must move at least one box per day.

The list `P` represents the boxes, where each entry is the size of the box. You can move the box with index `i` if and only if all boxes with smaller indexes have already been moved.

Find the minimum total truck size required to move all the boxes.

Examples

Example 1:

Input: `P = [10, 2, 20, 5, 15, 10, 1], d = 3`

Output: `31`

Explanation:

- day 1 - `[10, 2, 20, 5, 15]`. `TruckSize = 20`
- day 2 - `[10]`. `TruckSize = 10`
- day 3 - `[1]`. `TruckSize = 1`

`Total = 20 + 10 + 1 = 31`

```java
class Solution {
    public static int minContainerSize(List<Integer> boxSizes, int days) {
        // WRITE YOUR BRILLIANT CODE HERE
        if(days > boxSizes.size())
            return -1;
        int boxs = boxSizes.size();
        int[][] dp = new int[boxs+1][days+1];

        for(int[] arr : dp){
            Arrays.fill(arr, Integer.MAX_VALUE/2);
        }
        dp[0][0] = 0;
        // dp[i][k] = min size Container that need to pack i items in k days.
        // dp[i][k] = min (dp[i][k], min(dp[j][k-1] + max(boxSizes from j ... i) ))
        // k - 1<= j < i
        for(int i = 1; i <= boxs; i++){

            for(int k = 1; k <= days; k++){
                int maxBox = 0;

                for(int j = i - 1; j >= k - 1; j--){

                    maxBox = Math.max(maxBox, boxSizes.get(j));
                    dp[i][k] = Math.min(dp[i][k], dp[j][k-1] + maxBox);
                    //System.out.println(i +" "+j +" "+k + " " +dp[i][j] );
                }
            }
        }


        return dp[boxs][days];
    }
```

box width，给的答案有三个test case不能过。但是可以加打印看数据，发现1）全重复素如何处理；2）数据值很大如何处理。

所以:重复元素，取第一个和最后一个如果排序后还一样，直接输出；

数据很大，running sum会溢出，用long。

994. Rotting Oranges

```java
class Solution {
    int[] dir = new int[] { 0, 1, 0, -1, 0 };
    public int orangesRotting(int[][] grid) {
        // Init: Find all rotten oranges
        int time = 0;
        Queue<int[]> queue = new LinkedList<>();
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j] == 2) queue.add(new int[]{ i, j });
            }
        }

        // Start: A rotten orange starts rotting its neighbors as time goes by
        while(!queue.isEmpty()) {
            int size = queue.size();
            for(int i = 0; i < size; i++) {
                int[] cur = queue.poll();
                // Find its unrotten neighbors
                for(int j = 0; j < 4; j++) {
                    int x = cur[0] + dir[j], y = cur[1] + dir[j+1];
                    if(0 <= x && 0 <= y && x < grid.length && y < grid[0].length && grid[x][y] == 1) {
                        grid[x][y] = 2;
                        queue.add(new int[]{ x, y });
                    }
                }
            }
            if(!queue.isEmpty()) time++;
        }

        // End: Check if all oranges are rotten. If yes, return time. Otherwise, return -1.
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j] == 1) return -1;
            }
        }
        return time;
    }
}
```

five start seller

Amazon Online Assessment (OA) - Five Star Seller

Stuck? Discuss

An arborist that operates a plant store in Brooklyn, NY would like to improve their online sales by improving their ratings.

In order to become a five-star store, they must maintain a certain threshold percentage of five-star ratings. Given their current situation, find the minimum number of additional five-star ratings they must receive to meet the threshold. The overall online store percentage is calculated by taking the sum of percentages of five-star ratings for each product.

Examples

Example 1:

Input:

`productCount` = 3

`productRatings` = `[[4,4],[1,2],[3,6]]` where each entry is `[five-star reviews, total reviews]`

`threshold` = 77

Output: 3

Explanation :

We need the sum of the percentages of five-star ratings for each product to add up to the threshold.

The current percentage of five-star ratings for this seller is `((4/4) + (1/2) + (3/6))/3 = 66.66%`

If we add a five star review to product #2, their threshold becomes `((4/4) + (2/3) + (3/6))/3 = 72.22%`

If we add another five star review to product #2, their threshold becomes `((4/4) + (3/4) + (3/6))/3 = 75%`

If we add a five star review to product #3, their threshold becomes `((4/4) + (3/4) + (4/7))/3 = 77.38%`

Examples

Example 1:

Input:

`productCount` = `3`

`productRatings` = `[[4,4],[1,2],[3,6]]` where each entry is `[five-star reviews, total reviews]`

`threshold` = `77`

Output: `3`

Explanation :

We need the sum of the percentages of five-star ratings for each product to add up to the threshold.

The current percentage of five-star ratings for this seller is `((4/4) + (1/2) + (3/6))/3 = 66.66%`

If we add a five star review to product #2, their threshold becomes `((4/4) + (2/3) + (3/6))/3 = 72.22%`

If we add another five star review to product #2, their threshold becomes `((4/4) + (3/4) + (3/6))/3 = 75%`

If we add a five star review to product #3, their threshold becomes `((4/4) + (3/4) + (4/7))/3 = 77.38%`

At this point, the `77%` threshold is met. The answer is `3`, because there is no other way to add less ratings and achieve `77%` or more.

Constraints:

There is always at least one product, and the threshold is between `1` and `99` inclusive. All values are positive.

solution:

```
class Solution {

    static class Product{

        public int pn;
        public int pd;
        public double rating;
        public double increasing;

        Product(){
        }
        Product(int pn, int pd){
            this.pn = pn;
            this.pd = pd;
            this.rating = (double) pn/pd;
            this.increasing = (double) (pn+1)/(pd+1) - this.rating;

        }
    }
```

```java
public static int fiveStarReviews(List<List<Integer>> ratings, int threshold) {
    // WRITE YOUR BRILLIANT CODE HERE
    int count = ratings.size();

    Queue<Product> q = new PriorityQueue<>((a,b) -> Double.compare(b.increasing,a.increasing));
    double currPercentage  = 0;
    for(List<Integer> list : ratings){

        int first = list.get(0);
        int second = list.get(1);

        q.offer(new Product(first, second));
        currPercentage += (double) first/second;
    }
    currPercentage/= count;

    int result = 0;
    while(currPercentage*100 < threshold){
        Product temp = q.poll();
        currPercentage += temp.increasing / count;
        temp = new Product(temp.pn+1, temp.pd+1);
        q.offer(temp);
        result++;
    }


    return result;
}
```

pn = product numerator
pd = product denominator

other solutions:
https://www.1point3acres.com/bbs/interview/amazon-software-engineer-717167.html
https://leetcode.com/playground/8seNPbYP

Substrings of size K with K distinct chars

Given a string `s` and an int `k`, return all unique substrings of `s` of size `k` with `k` distinct characters.

Example 1:

```
Input: s = "abcabc", k = 3
Output: ["abc", "bca", "cab"]
```

Example 2:

```
Input: s = "abacab", k = 3
Output: ["bac", "cab"]
```

Example 3:

```
Input: s = "awaglknagawunagwkwagl", k = 4
Output: ["wagl", "aglk", "glkn", "lkna", "knag", "gawu", "awun", "wuna", "unag",
"nagw", "agwk", "kwag"]
Explanation:
Substrings in order are: "wagl", "aglk", "glkn", "lkna", "knag", "gawu", "awun",
"wuna", "unag", "nagw", "agwk", "kwag", "wagl"
"wagl" is repeated twice, but is included in the output once.
```

solution:

```java
public static Set<String> uniqueSubstringSizeK(String s, int k) {
    Set<String> set = new HashSet<>();
    int[] ch = new int[26];
    int lo=0;
    int hi=0;
    while(lo<=hi && hi<s.length()) {
        ch[s.charAt(hi)-'a']++;
        while(ch[s.charAt(hi)-'a'] != 1) {
            ch[s.charAt(lo)-'a']--;
            lo++;
        }
        if(hi-lo+1 == k) {
            set.add(s.substring(lo, hi+1));
            ch[s.charAt(lo)-'a']--;
            lo++;
        }
        hi++;
    }
    System.out.println(set.size());
    Iterator<String> it = set.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
    return set;
}
```

postfix-notation，比如输入4 5 *，输出20（4*5=20），例如2 3/*输出1（2*3/6 = 1），也不难，直接写就行，同样Testcase比较难搞

solution:

```
public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();
    int op1 = 0, op2 = 0;
    for (String s : tokens) {
        if (s.equals("+") || s.equals("-") || s.equals("*") || s.equals("/")) {
            op2 = stack.pop(); op1 = stack.pop();
        }
        switch(s) {
            case "+":
                stack.push(op1 + op2);
                break;
            case "-":
                stack.push(op1 - op2);
                break;
            case "*":
                stack.push(op1 * op2);
                break;
            case "/":
                stack.push(op1 / op2);
                break;
            default:
                stack.push(Integer.valueOf(s));
        }
    }
    return stack.pop();
}
```

Throttling Gateway

```
int ans = 0 ;
for(int i = 0 ;  i < n; i++){

    if(i  >2 && requestTime[i] == requestTime[i-3]){
        ans++;
    } else if(i > 19 && (requestTime[i] - requestTime[i-20]) <10){
        ans++;
    } else if( i > 59 && (requestTime[i] - requestTime[i-60]) <60 ){
        ans++;
    }
}
```

number of provinces

There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

A province is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the `ith` city and the `jth` city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of provinces*.

```java
public int findCircleNum(int[][] isConnected) {
    int res = 0;
    HashSet<Integer> visited = new HashSet<>();
    for (int i = 0; i < isConnected.length; i++) {
        if (!visited.contains(i)) {
            res++;
            dfs(isConnected, visited, i);
        }
    }
    return res;
}

public void dfs(int[][] m, HashSet<Integer> visited, int cur) {
    visited.add(cur);
    for (int i = 0; i < m.length; i++) {
        if (m[i][cur] == 1 && !visited.contains(i)) {
            dfs(m, visited, i);
        }
    }

}
```

union find solution:

```java
public int findCircleNum(int[][] isConnected) {
    UnionFind uf = new UnionFind(isConnected.length);
    for (int i =0; i < isConnected.length; i++) {
        for (int j = i+1 ; j < isConnected.length; j++) {
            if (isConnected[i][j] == 1) uf.union(i,j);
        }
    }

    return uf.getCityProvince();
}



class UnionFind {
    int N;
    int count;
    int[] parents;

    public UnionFind(int N) {
        this.N = N;
        this.count = N;
        parents = new int[N];
        for (int i=0; i< N; i++) {
            parents[i] = i;
        }
    }

    public int find(int city) {
        while(city != parents[city]) city = parents[city];
        return city;
    }

    public void union(int city1, int city2) {
        int root1 = find(city1);
        int root2 = find(city2);

        if(root1 == root2) return;

        parents[root2] = root1;
        count--;
    }

    public int getCityProvince() {
        return count;
    }
}
```

move the obstacle (demolition robot)

You are in charge of preparing a recently purchased lot for the Company's building.

The lot is covered with trenches and has a single obstacle that needs to be taken down before the foundation is prepared for the building.

The demolition robot must remove the obstacle before the progress can be made on the building.

Assumptions

- The lot is flat, except the trenches, and can be represented by a 2D grid.
- The demolition robot must start at the top left corner of the lot, which is always flat, and can move on the block up, down, right, left
- The demolition robot cannot enter trenches and cannot leave the lot.
- The flat areas are indicated by `1`, areas with trenches are indicated by `0`, and the obstacle is indicated by `9`

Input

The input consists of one argument:

`lot`: a `2d grid of integers`

Output

Return an `integer` that indicated the minimum distance traversed to remove the obstacle else return `-1`

solution:

```
public static int moveObstacle(List<List<Integer>> lot) {
    if(lot == null)
        return -1;
    if(lot.size() <= 0 || lot.get(0).size()<=0)
        return -1;
    // WRITE YOUR BRILLIANT CODE HERE
    Queue<int[]> q = new LinkedList<>();
    boolean[][] visited = new boolean[lot.size()][lot.get(0).size()];
    int[][] directions = new int[][]{{0,1}, {0,-1}, {1,0}, {-1,0}};
    q.add(new int[]{0,0});
    int result = 0;
```

```
while(q.size() > 0){
    int size = q.size();

    for(int i = 0; i < size; i++){
        int[] node = q.poll();
        int x = node[0];
        int y = node[1];
        visited[x][y] = true;
        if(lot.get(x).get(y) == 9)
            return result;

        for(int[] dirt : directions){
            int nx = x + dirt[0];
            int ny = y + dirt[1];

            if(nx < 0 || nx >= lot.size() || ny <0 || ny >lot.get(nx).size() || lot.get(nx).get(ny) == 0 ||
              visited[nx][ny] == true)
                continue;
             q.offer(new int[]{nx,ny});

        }
    }
   result++;
}
return -1;
```

39. earliest time to complete delivers (schedule delivers)

You are the manager of logistics for a burger franchise, and you are tasked with delivering supplies as quickly as possible.

There are `n` restaurants with `4` receiving docks each. Each dock has a specified maximum receiving rate.

Input

`n` = number of restaurants

`openTimes` = a number that represents the time the `i`th restaurant opens

`deliveryTimeCost` = an array of numbers representing the time it takes to unload a delivery. There are exactly `n * 4` values in this list.

Output

The earliest time all deliveries can be completed.

Examples

Example 1:

Input:

`n = 2`

`openTimes = [8, 10]`

`deliveryTimeCost = [2,2,3,1,8,7,4,5]`

Output: `16`

Explanation:

For the restaurant that opens at `8`, assign deliveries with cost `[8, 7, 5, 4]`. These will complete at `(8+8)`, `(8+7)`, `(8+5)`, and `(8+4)`, which are `16`, `15`, `13`, and `12` respectively.

For the restaurant that opens at `10`, assign deliveries with cost `[3, 2, 2, 1]`. These will complete at `(10+3)`, `(10+2)`, `(10+2)`, and `(10+1)`, which are `13`, `12`, `12`, and `11` respectively.

The lastest of all of the delivery completion time is at `16`.

analysis: match min open time with the max 4 cost time

solution:

```
public static int earliestTime(int n, List<Integer> openTimes, List<Integer> deliveryTimeCost) {
    // WRITE YOUR BRILLIANT CODE HERE
    Collections.sort(openTimes);
    Collections.sort(deliveryTimeCost);
    Collections.reverse(deliveryTimeCost);

    int result = 0;
    int j = 0;
    for(int i = 0; i < openTimes.size(); i++){
        result = Math.max(result, openTimes.get(i) + deliveryTimeCost.get(j));
        j+=4;

    }

    return result;
}
```

40. break a palindrome

Given a palindromic string of lowercase English letters `palindrome`, replace **exactly one** character with any lowercase English letter so that the resulting string is **not** a palindrome and that it is the **lexicographically smallest** one possible.

Return *the resulting string. If there is no way to replace a character to make it not a palindrome, return an **empty string**.*

A string `a` is lexicographically smaller than a string `b` (of the same length) if in the first position where `a` and `b` differ, `a` has a character strictly smaller than the corresponding character in `b`. For example, `"abcc"` is lexicographically smaller than `"abcd"` because the first position they differ is at the fourth character, and `'c'` is smaller than `'d'`.

Example 1:

```
Input: palindrome = "abccba"
Output: "aaccba"
Explanation: There are many ways to make "abccba" not a palindrome, such as "zbccba", "aaccba",
and "abacba".
Of all the ways, "aaccba" is the lexicographically smallest.
```

solution:

```java
public String breakPalindrome(String palindrome) {

    if(palindrome.length() < 2)
            return "";

    char[] chs = palindrome.toCharArray();

    for(int i = 0; i < chs.length/2; i++){

        if(chs[i] != 'a'){
            chs[i] = 'a';
            return String.valueOf(chs);
        }

    }

    chs[chs.length - 1] = 'b';

    return String.valueOf(chs);
}
```

41. Top K frequently mentioned keywords

Find the keywords that are most frequently mentioned in a given list of text snippets. Return a list of the top `k` most frequently mentioned keywords, sorted in increasing order by their frequency, ingnoring case sensitivity.

A "mention" of a keyword happens when the keyword appears at least once in a text snippet. If a keyword appears more than once in a snippet, it only counts as one "mention". Sort alphabetically if multiple keywords are mentioned the same number of times.

Input

`k`: a number

`keywords`: a list of words

`snippets`: a list of text snippets, each containing a single contiguous paragraph

Output

A list of words sorted from most frequenly mentioned to least frequently mentioned.

solution:

```java
public static List<String> topMentioned(int k, List<String> keywords, List<String> reviews) {
    // WRITE YOUR BRILLIANT CODE HERE
    if(keywords == null ||reviews == null || keywords.size() == 0 || reviews.size() == 0)
        return new ArrayList<>();

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

    for(String word: keywords){
        for(String sentence : reviews){
            if(sentence.toLowerCase().contains(word.toLowerCase())){
                map.put(word, map.getOrDefault(word, 0) + 1);
            }
        }
    }
    if(map.size() == 0)
        return new ArrayList<>();

    Queue<String> q = new PriorityQueue<>((a,b) ->
                                          map.get(a) == map.get(b)? a.compareTo(b) :map.get(b)-map.get(a));

    List<String> result = new ArrayList<>();
    for(String key : map.keySet()){
        q.offer(key);
    }

    while(k > 0){
        result.add(q.poll());
        k--;
    }


    return result;
}
```

42. two sum unique pair

```java
public static int twoSumUniquePairs(List<Integer> nums, int target) {
    // WRITE YOUR BRILLIANT CODE HERE
    Set<Set<Integer>> seen = new HashSet<>();
    Map<Integer, Integer> map = new HashMap<>();

    for(int num : nums){
        if(map.containsKey(target - num)){

            seen.add(new HashSet<>(Arrays.asList(target - num , num)));
        }
        else{
            map.put(num, 1);
        }

    }

    return seen.size();
}
```

43. packaging automation

The Fulfillment Center consists of a packaging bay where orders are automatically packaged in groups(n). The first group can only have 1 item and all the subsequent groups can have one item more than the previous group. Given a list of items on groups, perform certain operations in order to satisfy the constraints required by packaging automation.

The conditions are as follows:

-The first group must contain 1 item only.

-For all other groups, the difference between the number of items in adjacent groups must not be greater than 1. In other words, for 1<=i<n, arr[i]-arr[i-1]<=1

To accomplish this, the following operations are available:

- Rearrange the groups in any way.
- Reduce any group to any number that is at least 1

Write an algorithm to find the maximum number of items that can be packaged in the last group with the conditions in place.

Input

The function/method consists of two arguments:

numGroups, an integer representing the number of groups(n);

arr, a list of integers representing the number of items in each group

Output

Return an integer representing the maximum items that can be packaged for the final group of the list given the conditions above.

Example1:

Input:

[3,1,3,4]

Output:

4

Explanation:

Subtract 1 from the first group making the list [2, 1, 3, 4]. Rearrange the list into [1, 2, 3, 4]. The final maximum of items that can be packaged in the last group is 4.

Example2:

Input:

[1,3,2,2]

Output:

3

solution：

```
public int packaging(int numGroups, int[] arr) {

    Arrays.sort(arr);
    arr[0] = 1;

    for(int i = 1; i < numGroups; i++){

        if(arr[i] > arr[i-1]){
            arr[i] = arr[i-1] + 1;
        }

    }

    return arr[arr.length - 1];
}
```

cannot pass using sort, which time is nlogn

```
public int packaging(int numGroups, int[] arr) {
    int result = 0;

    for(int num : arr){
        result += num;
    }

    while(result < numGroups*(numGroups+1)/2){

        result--;
        numGroups--;
    }
    return numGroups;
}
```

44. cutoff rank

A group of work friends at Amazon is playing a competitive video game together. During the game, each player receives a certain amount of points based on their performance. At the end of a round, players who achieve at least a cutoff rank get to "level up" their character, gaining increased abilities for them. Given the scores of the players at the end of the round, how many players will be able to level up their character?

Note that players with equal scores will have equal ranks, but the player with the next lowest score will be ranked based on the position within the list of all players' scores. For example, if there are four players, and three players tie for first place, their ranks would be 1,1,1, and 4. Also, no player with a score of O can level up, no matter what their rank.

Write an algorithm that returns the count of players able to level up their character.

Input

The input to the function/method consists of three arguments:

cutOffRank, an integer representing the cutoff rank for levelin up the player's character;

num, an integer representing the total number of scores;

scores, a list of integers representing the scores of the players.

Output

Return an integer representing the number of players who will be able to level up their characters at the end of the round.

Constraints

1 <= num <= 10^5

0 <= scores[i] <= 100

0 <= i < num

cutOffRank <= num

Examples

Example 1:

Input:

cutOffRank = 3

num= 4

scores=[100, 50, 50, 25]

Output:

3

Explanation:

There are num= 4 players, where the cutOffRank is 3 and scores = [100, 50,50, 25]. These players' ranks are [1, 2, 2, 4]. Because the players need to have a rank of at least 3 to level up their characters, only the first three players will be able to do so.

So, the output is 3.

solution:

```
public int cutOffRank(int cutOffRank, int num, int[] scores) {


    if(cutOffRank == 0 || num > 100000)
        return 0;

    Arrays.sort(scores);

    int currRank = 1;
    int globalRank = 1;
    int count = 1;
    for(int i = num - 2; i >= 0; i--){
        globalRank++;

        if(scores[i] < scores[i+1]){
            currRank = globalRank;
        }

        if(currRank <= cutOffRank)
            count++;
        else
            break;
    }

    return count;
}
```

max area of island

```java
class Solution {
    int result = 0, area = 0;
    public int maxAreaOfIsland(int[][] grid) {
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == 1) {
                    dfs(grid, i, j);
                    area = 0;
                }
            }
        }
        return result;
    }
    private void dfs(int[][] grid, int i, int j) {
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0) {
            return;
        }
        grid[i][j] = 0;
        area++;
        result = Math.max(result, area);
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }
}
```