Online Assessment

1. 卡车装 M 个箱子， N 个地点 List M<N 列出最近的 M 个位置。 要注意输入不正常的情况，比如只有一个输入

【Heap 问题】

1. Top K closest numbers / 找最近的 K 个餐馆 / 饭店最近的位置 / 给一堆的餐馆的位置，和 customer 的位置，找出离 customer 最近的餐馆。 / 找最近的 k 个餐馆，给了餐馆的坐标，和距离的定义 / 一个是求 K 个最近的餐馆，用 heap 做就好了。 Given a list of points, and a point, find the K closest points 函数：nearestKRestruants(List<List> Locations, int K)

```java
public int[][] kClosest(int[][] points, int k) {
    // max heap by distance
    PriorityQueue<int[]> pq = new PriorityQueue<>(k+1, new Comparator<int[]>() {
        public int compare(int[] a, int[] b) {
            return  (b[0]*b[0]+b[1]*b[1])-(a[0]*a[0] + a[1]*a[1]);
        }
    });
    for(int[] point: points) {
        pq.add(point);
        if(pq.size() > k) {
            pq.poll();
        }
    }
    int[][] list = new int[pq.size()][2];
    int i = 0;
    while(pq.size() > 0) {
        list[i++] = pq.poll();
    }
    return list;
}
```

参考解法：用 max heap 做，for 每个点，算 dist, heapify 所有点，最后 pop k top points。时间：O(n+klogn)。all 15 test cases passed。这里似乎不用考虑 integer overflow。

k 个地点距离，用 x^2 + y^2 就能通过所有 test case，非常简单。（但之后面试会不会问怎么 optimize 就不一定了，但我也没想出怎么 optimize）

2. Five hightest：选每个产品五个评分最高的求平均值。给一个 list，每个 element 是<productId, productRating>，求每个 product 最高的 5 个评分的 Average

参考解法：TreeMap<productId, Heap(5)> 然后遍历

3. 卡车送货， 求 k 个最近距离。一个二维 grid， 有些点标记成了送货地址， 输出从（0，0）出发最近的 k 个目标。 参考解法：Heap

【Follow-up】 第二题还是送货，有障碍，求到目标地址的最短距离 参考解法：bfs

4. 找最近的几个点

5. Reorder log files

```
public String[] reorderLogFiles(String[] logs) {
        Arrays.sort(logs, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                String[] strs1 = s1.split(" ");
                String[] strs2 = s2.split(" ");
                if (Character.isDigit(strs1[1].charAt(0)) &&
Character.isDigit(strs2[1].charAt(0))) return 0;
                if (Character.isLetter(strs1[1].charAt(0)) &&
Character.isDigit(strs2[1].charAt(0))) return -1;
                if (Character.isDigit(strs1[1].charAt(0)) &&
Character.isLetter(strs2[1].charAt(0))) return 1;

                String log1 = s1.substring(s1.indexOf(" ") + 1);
                String log2 = s2.substring(s2.indexOf(" ") + 1);
                if (log1.equals(log2)) {
                    return strs1[0].compareTo(strs2[0]);
                }
                return log1.compareTo(log2);
            }
        });
        return logs;
    }
```

5. prime, no prime 排序

Following is the question. Sort the given String Array. ["ykc 82 01", "eo first qpx", "o9z cat hamster", "06f 12 25 6", "azo first qpx", "236 cat dog rabbit snake"]

Each string in the array is represented in the follwoing format: <"identifier"> <"version"> <> <> <> ...

Sort based on version, if there is a conflict sort based on Identifier. Conditions:

If String's version is a number we should not sort that particular string. If there is a conflict while sorting strings based on "version" we should consider "identifier" "version" can be a number or a string. "Identifier" can be a number or alphanumeric. Output:

["236 cat dog rabbit snake", "o9z cat hamster", "azo first qpx", "eo first qpx", "ykc 82 01", "06f 12 25 6"]

```
        public List<String> method(String[] strArray) {
                List numStringList = new ArrayList<>();
                List stringList = new ArrayList<>();
                List outList = new ArrayList<>();

                for (String str : strArray) {
                        if (isNumString(str))
                                numStringList.add(str);
```

```
				else
					stringList.add(str);
			}
			Collections.sort(stringList, new Comparator() {

				@Override
				public int compare(String o1, String o2) {
					String[] str1Array = o1.split(" ");
					String[] str2Array = o2.split(" ");

					int sComp = str1Array[1].compareTo(str2Array[1]);

					if (sComp != 0)
						return sComp;

					return str1Array[0].compareTo(str2Array[0]);
				}

				@Override
				public int compare(Object o1, Object o2) {
					// TODO Auto-generated method stub
					return 0;
				}

			});
			outList.addAll(stringList);
			outList.addAll(numStringList);
			return outList;
		}

		private static boolean isNumString(String str) {

			String s = "0123456789";
			String[] strArray = str.split(" ");
			char[] chArray = strArray[1].toCharArray();
			for (int i = 0; i < chArray.length; i++)
				if (s.indexOf(strArray[1].charAt(0)) == -1)
					return false;
			return true;
	}

	private static String[] reorderLogFiles(String[] logs) {

		List<String> letterLogs = new LinkedList<>();
		List<String> digitLogs = new LinkedList<>();
		for (String input : logs) {
			String[] splittedString = input.split(" ");

			char c = splittedString[1].toCharArray()[0];

			//letter logs
			if (c >= 'a' && c <= 'z') {
				letterLogs.add(input);
			} else {
```

```
                digitLogs.add(input);
            }
        }

        // Sort the letter logs
        Comparator<String> letterLogsComparator = new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                String[] splitteds1 = o1.split(" ");
                String[] splitteds2 = o2.split(" ");

                int compare = splitteds1[1].compareTo(splitteds2[1]);

                if (compare != 0) {
                    return compare;
                } else {
                    return splitteds1[0].compareTo(splitteds2[0]);
                }
            }
        };

        letterLogs.sort(letterLogsComparator);

        int index = 0;
        String[] result = new String[logs.length];

        for (String letter : letterLogs) {
            result[index++] = letter;
        }

        for (String digitLog : digitLogs) {
            result[index++] = digitLog;
        }
        return result;
    }
```

6. Amazon’s Sort Center

In Amazon’s sort center, a computer system decides what packages are to be loaded on what trucks. All rooms and spaces are abstracted into space units which is represented as an integer. For each type of truck, they have different space units. For each package, they will be occupying different space units. As a software development engineer in sort centers, you will need to write a method:

Given truck space units and a list of product space units, find out exactly TWO products that fit into the truck. You will also implement an internal rule that the truck has to reserve exactly 30 space units for safety purposes. Each package is assigned a unique ID, numbered from 0 to N-1.

Assumptions : You will pick up exactly 2 packages. You cannot pick up one package twice. If you have multiple pairs, select the pair with the largest package.

Input : The input to the function/method consists of two arguments : truckSpace , an integer representing the truck space. packagesSpace , a list of integers representing the space units occupying by packages.

Output : Return a list of integers representing the IDs of two packages whose combined space will leave exactly 30 space units on the truck.

Example Input : truckSpace = 90 packagesSpace = [1, 10, 25, 35, 60] Output : [2, 3] Explanation : Given a truck of 90 space units, a list of packages space units [1, 10, 25, 35, 60], Your method should select the third(ID-2) and fourth(ID-3) package since you have to reserve exactly 30 space units.

```
public int[] truckAndLoad(int space, int[] nums) {
    if (space <= 30) return new int[];
    space = space - 30;
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = space - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{i, map.get(complement)};
        }
        map.put(i, nums[i]);
    }
    throw new IllegalArgumentException("No solutions");
}
```

7. Roll Dice

A six-sided dice is a small cube with a different number of pips. Each face(side) ranging from 1 to 6. On any two opposite side of the cube, the number of pips adds up to 7; that is, there are three pairs of opposite sides: 1 and 6, 2 and 5, and 3 and 4. There are N dice lying on a table, each showing the pips on its top face. In one move, you can take one dice and rotate it to an adjacent face. For example, you can rotate a dice that shows 1 to show 2, 3, 4 or 5. However, it cannot show 6 in a single move, because the faces with one pip and six pips visible are opposite sides rather than adjacent. You want to show the same number of pips on the top face of all N dice. Given that each of the dice can be moved multiple times, count the minimum number of moves needed to get equal faces.

Write a function: that, given an array A consisting of N integers describing the number of pips (from 1 to 6) shown on each dice's top face, returns the minimum number of moves necessary for each dice show the same number of pips.

For example, given: • A = [1, 2, 3], the function should return 2, as you can pick the first two dice and rotate each of them in one move so that they all show three pips on the top face. Notice that you can also pick any other pair of dice in this case. • A = [1, 1, 6], the function should also return 2. The only optimal answer is to rotate the last dice so that it shows one pip. It is necessary to use two rotations to achieve this. • A = [1, 6, 2, 3], the function should return 3. For instance, you can make all dice show 2: just rotate each dice which is not showing 2 (and notice that for each dice you can do this in one move).

Assume that: • N is an integer within the range [1...100]; • each element of the array A is an integer within the range [1...6].

8. Movies on Flight

You are on a flight and wanna watch two movies during this flight. You are given int[] movie_duration which includes all the movie durations. You are also given the duration of the flight which is d in minutes. Now, you need to pick two movies and the total duration of the two movies is less than or equal to (d - 30min). Find the pair of movies with the longest total duration. If multiple found, return the pair with the longest movie.

e.g. Input movie_duration: [90, 85, 75, 60, 120, 150, 125] d: 250

Output [90, 125] 90min + 125min = 215 is the maximum number within 220 (250min - 30min)

leetcode. 1099

【求 Pair Sum】

1. 飞机来回路程

Amazon Prime Air is developing a system that divides shipping routes using flight optimization routing systems to a cluster of aircraft that can fulfill these routes. Each shipping route is identified by a unique integer identifier, requires a fixed non-zero amount of travel distance between airports, and is defined to be either a forward shipping route or a return shipping route. Identifiers are guaranteed to be unique within their own route type, but not across route types.

Each aircraft should be assigned two shipping routes at once: one forward route and one return route. Due to the complex scheduling of flight plans, all aircraft have a fixed maximum operating travel distance, and cannot be scheduled to fly a shipping route that requires more travel distance than the prescribed maximum operating travel distance. The goal of the system is to optimize the total operating travel distance of a given aircraft. A forward/return shipping route pair is considered to be “optimal” if there does not exist another pair that has a higher operating travel distance than this pair, and also has a total less than or equal to the maximum operating travel distance of the aircraft.

For example, if the aircraft has a maximum operating travel distance of 3000 miles, a forward/return shipping route pair using a total of 2900 miles would be optimal if there does not exist a pair that uses a total operating travel distance of 3000 miles, but would not be considered optimal if such a pair did exist.

Your task is to write an algorithm to optimize the sets of forward/return shipping route pairs that allow the aircraft to be optimally utilized, given a list of forward shipping routes and a list of return shipping routes.

- Input

The input to the function/method consists of three arguments: maxTravelDist: an integer representing the maximum operating travel distance of the given aircraft forwardRouteList: a list of pairs of integers where the first integer represents the unique identifier of a forward shipping route and the second integer represents the amount of travel distance required by this shipping route returnRouteList: a list of pairs of integers where the first integer represents the unique identifier of a return shipping route and the second integer represents the amount of travel distance required by this shipping route.

- Output

Return a list of pairs of integers representing the pairs of IDs of forward and return shipping routes that optimally utilize the given aircraft. If no route is possible, return an empty list.

- Example :
- 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.

Shipping Route ID#2 from the forwardShippingRouteList requires 5000 miles travelled, and Shipping Route ID#4 from returnShippingRouteList also requires 5000 miles travelled. Combined, they add up to 10000 miles travelled.

Similarly, Shipping Route ID#3 from forwardShippingRouteList requires 7000 miles travelled, and Shipping Route ID#2 from returnShippingRouteList requires 3000 miles travelled. These also add up to 10000 miles travelled.

Therefore, the pairs of forward and return shipping routes that optimally utilize the aircraft are [2, 4] and [3, 2].

油箱最多走 Max = 10000 去程：[1, 2000] [2, 5000] 回程：[1, 5000] [2, 2000] [3, 8000] 问飞机最多可以走哪几条路？不一定能走满 Max，可以是最靠近 Max 的数。

求两个给定list内的值相加， 最接近max的pair，返回在对应list中的index

复杂度分析：

Time: O(n*m) , assuming n is the size of forwardRouteList and m is the size of retumRouteList

Space: O(1), I only used constant extra space.

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

public class TwoSumClosestToTarget {
    public List<List<Integer>> twoSumClosest2Target(int maxTravelDist,
List<List<Integer>> forwardRouteList,
                                                     List<List<Integer>>
```

```
returnRouteList) {
        // CORNER CASE
        if(forwardRouteList == null || forwardRouteList.size() == 0 ||
returnRouteList == null || returnRouteList.size() == 0){
            return new ArrayList<>();
        }

        int sum ;
        int minSum = 0;
        List<List<Integer>> result = new ArrayList<>();

        /*
           PART1: find each possible pair
         */
        for (int i = 0; i < forwardRouteList.size(); i++) {
            for (int j = 0; j < returnRouteList.size(); j++) {
                sum = forwardRouteList.get(i).get(1) +
returnRouteList.get(j).get(1);
                if (Math.abs(sum - maxTravelDist) < Math.abs(minSum -
maxTravelDist)) {
                    minSum = sum;
                    List<Integer> list = new ArrayList<>();
                    list.add(forwardRouteList.get(i).get(0));
                    list.add(returnRouteList.get(j).get(0));
                    //why clean() instead of remove()?  Due to not ensuring result
size
                    result.clear();
                    result.add(list);
                }
                // if many matchable pairs, add them all to result
                else if (Math.abs(sum - maxTravelDist) == Math.abs(minSum -
maxTravelDist)) {
                    List<Integer> list = new ArrayList<>();
                    list.add(forwardRouteList.get(i).get(0));
                    list.add(returnRouteList.get(j).get(0));
                    result.add(list);
                }
            }
        }
        return result;
    }
    /*
       TEST CASE
     */
    public static void main(String[] args) {
        TwoSumClosestToTarget test = new TwoSumClosestToTarget();
        int maxTravelDist = 10000;
        // ----------- INPUT: forwardRouteList---------
        List<List<Integer>> forwardRouteList = new ArrayList<>();
        List<Integer> list1 = new ArrayList<>();
        list1.add(1);
        list1.add(3000);
        List<Integer> list2 = new ArrayList<>();
        list2.add(2);
        list2.add(5000);
        List<Integer> list3 = new ArrayList<>();
        list3.add(3);
        list3.add(7000);
        List<Integer> list4 = new ArrayList<>();
        list4.add(4);
        list4.add(10000);
        forwardRouteList.add(list1);
        forwardRouteList.add(list2);
        forwardRouteList.add(list3);
        forwardRouteList.add(list4);

        // ----------- INPUT: retumRouteList---------
        List<List<Integer>> retumRouteList = new ArrayList<>();
        List<Integer> lr1 = new ArrayList<>();
        lr1.add(1);
        lr1.add(2000);
        List<Integer> lr2 = new ArrayList<>();
        lr2.add(2);
        lr2.add(3000);
        List<Integer> lr3 = new ArrayList<>();
        lr3.add(3);
        lr3.add(4000);
        List<Integer> lr4 = new ArrayList<>();
        lr4.add(4);
        lr4.add(5000);
        retumRouteList.add(lr1);
        retumRouteList.add(lr2);
        retumRouteList.add(lr3);
        retumRouteList.add(lr4);

        List<List<Integer>> result = test.twoSumClosest2Target(maxTravelDist,
forwardRouteList, retumRouteList);
        for (int i = 0; i < result.size(); i++) {
            System.out.println("[" + result.get(i).get(0) + "," +
result.get(i).get(1) + "]");
        }
    }
}

private List<List<Integer>> optimalUtilization(
            int deviceCapacity,
            List<List<Integer>> foregroundAppList,
            List<List<Integer>> backgroundAppList)
    {
        // WRITE YOUR CODE HERE
        TreeMap<Integer, List<Integer>> tree = new TreeMap<>();
        for (List<Integer> pair : backgroundAppList) {
            List<Integer> list = tree.getOrDefault(pair.get(1), new ArrayList<>
());
            list.add(pair.get(0));
            tree.put(pair.get(1), list);
        }
        TreeMap<Integer, List<List<Integer>>> result = new TreeMap<>();
        for (List<Integer> pair : foregroundAppList) {
            Integer floorKey = tree.floorKey(deviceCapacity - pair.get(1));
            if (floorKey != null) {
                int diff = Math.abs(deviceCapacity - pair.get(1) - floorKey);
                List<List<Integer>> list = result.getOrDefault(diff, new
ArrayList<>());
                for (int id : tree.get(floorKey)) {
                    List<Integer> match = new ArrayList<>();
                    match.add(pair.get(0));
                    match.add(id);
                    list.add(match);
                }
                result.put(diff, list);
            }
        }
        return result.get(result.firstKey());
    }
```

参考解法：见题 2（思路一模一样）

2. 类似 Two Sum Closest

给两个数组。从两个数组中分别取一个数，和要小于等于 k。找到和最大的组合。 （其他描述：返回两个 list 中和最大但不超过一个值的下标－函数名 optimalUtilization） 给 2 个 sorted array，和一个整数 capacity，每个 array 各找出一个数，组成一个 pair。找出 pair 满足以下条件： 1）sum of pair <= capacity 2) sum is maximum 后来真的遇到这道题的时候还是先用 O(N*N) 的算法检查了所有可能的组合，oj 是可以过的。我觉得这题的考点不在降低时间复杂度，而在不能错过任何一个重复的组合。

参考思路： 两层 for 循环，穷举所有组合。这种方式简单能过。

其它思路：sort, 2pointer 或者 binary search。

输入输出比较复杂。不建议用 nlogn 的解法。直接双循环暴力解即可。 sortA, sortB: mlogm + nlogn, binary search num from A in B. mlogn, total: mlogm + nlogn + mlogn

容易出错点：数组里有重复元素，漏解

```
public List<List<Integer>> PrimeMaxProfit(int maxTravelDist, List<List<Integer>>
forwardRouteList, List<List<Integer>> returnRouteList) {
     List<List<Integer>> res = new ArrayList<>();
     int forLen = forwardRouteList.size(), retLen = returnRouteList.size() ;
     if (maxTravelDist == 0 || forLen == 0 || retLen == 0) {
         return res;
     }

     Collections.sort(forwardRouteList, (a, b) -> (a.get(1) - b.get(1)));
     Collections.sort(returnRouteList, (a, b) -> (a.get(1) - b.get(1)));

     int l = 0, r = retLen - 1, diff = Integer.MAX_VALUE, sum;
     while (l < forLen && r >= 0) {
         sum = forwardRouteList.get(l).get(1) + returnRouteList.get(r).get(1);
```

```
            if (maxTravelDist - sum >= 0 && maxTravelDist - sum <= diff) {
                if (maxTravelDist - sum < diff) {
                    diff = maxTravelDist - sum;
                    res = new ArrayList<>();
                }
                res.add(Arrays.asList(forwardRouteList.get(l).get(0),
returnRouteList.get(r).get(0)));
            }

            if (sum >= maxTravelDist) {
                r--;
            } else {
                l++;
            }
        }

        return res;
    }
```

3. 有两个 List of apps， 里面存着每一个 app 的 index 和 它所需的 memory。 从 foregroundList（[1, 200], [2, 300]） 和一个 boregroundList ([1, 400], [2, 500]) 中各取一个组成一对，使得它们 memory 加起来的和最接近但不超过 capacity。 需要注意的是多个 app 的 memory 有可能是相同。

例如： foregroundList =[[1, 2000]], boregroundList = [[1, 8000], [2, 8000]], capacity = 10000. 需要返回 [[1,1], [1,2]]。

描述 2：给了两组 application 的 ID 和内存需要，以及一个目标内存，求一对 application 的 ID 使得它们的内存使用之和最接近目标内存。

参考思路： 可以用暴力解，也可以用 two pointers。我用了 treeset， all case passed 第二题：O(N*N) -> O(NlogN) 先 sort BACK list， 再遍历 Front list： for each element i in FRONT search for lower_bound(capacity-i) 输出的 candidates 用 max stack 存储，有新来的就一直 pop 直到 top 等于新来的或者 stack 为空。

顺序你是指相同的 capacity 的时候，不同 id number 组合的排序吗？好像是没有特别要求吧。

5. Most commom word

```
class Solution {
    public String mostCommonWord(String paragraph, String[] banned) {
        // split paragraph
        String[] words = paragraph.toLowerCase().split("\\W+"); // [^a-zA-Z]

        // add banned words to set
        Set<String> set = new HashSet<>();
        for(String word : banned){
            set.add(word);
        }
        // add paragraph words to hash map
        Map<String, Integer> map = new HashMap<>();
        for(String word : words){
            if(!set.contains(word)){
                map.put(word, map.getOrDefault(word, 0) + 1);
            }
        }

        // get the most frequent word
        int max = 0; // max frequency
        String res = "";
        for(String str : map.keySet()){
            if(map.get(str) > max){
                max = map.get(str);
                res = str;
            }
        }

        return res;
    }
}
```

6. Given n ropes of different lengths, you need to connect these ropes into one rope. You can connect only 2 ropes at a time. The cost required to connect 2 ropes is equal to sum of their lengths. The length of this connected rope is also equal to the sum of their lengths. This process is repeated until n ropes are connected into a single rope. Find the min possible cost required to connect all ropes.

Input ropes, an int arrary representing the rope length.

Output Return the min possible cost required to connect all ropes.

Examples 1 Input: ropes = [8, 4, 6, 12]

Output: 58

Explaination: Explanation: The optimal way to connect ropes is as follows

Connect the ropes of length 4 and 6 (cost is 10). Ropes after connecting: [8, 10, 12] Connect the ropes of length 8 and 10 (cost is 18). Ropes after connecting: [18, 12] Connect the ropes of length 18 and 12 (cost is 30). Total cost to connect the ropes is 10 + 18 + 30 = 58 Examples 2 Input: ropes = [20, 4, 8, 2]

Output: 54

Examples 3 Input: ropes = [1, 2, 5, 10, 35, 89]

Output: 224

Examples 4 Input: ropes = [2, 2, 3, 3]

Output: 20

```
/** Leetcode 1046 */
import java.util.*;

public class NumOfSubFiles {

        public int minimumTime(int numOfSubFiles, List<Integer> files) {
            int result = 0;
            while (files.size() > 1) {
                Integer[] sorted = files.toArray(new Integer[files.size()]);
                Arrays.sort(sorted);
                files = new ArrayList<>(Arrays.asList(sorted));

                Integer sum = files.get(0) + files.get(1);

                files.remove(0);
                files.remove(0);
                files.add(sum);
                result += sum;
            }

            return result;
        }
}
```

7. 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.

e.g. Input [['O', 'O', 'O', 'O'], ['D', 'O', 'D', 'O'], ['O', 'O', 'O', 'O'], ['X', 'D', 'D', 'O'],]

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

```
/**
* Time complexity: O(r * c).
* Space complexity: O(r * c).
*/
public class Main {
    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;
        }
    }

    public static void main(String[] args) {
        char[][] grid = {{'O', 'O', 'O', 'O'},
                         {'D', 'O', 'D', 'O'},
                         {'O', 'O', 'O', 'O'},
                         {'X', 'D', 'D', 'O'}};
        System.out.println(minSteps(grid));
    }
}
```

8. Given: 2D Matrix

Multiple Starting Points ‘S’, Multiple End Points ‘E’ and dead blocks ‘D’ and '1’s you can walk through. We can start from any ‘S’ point and can end at any ‘E’ point. Find the shortest distance from any starting point to any end point.

```
// Time complexity: O(r * c)
// Space complexity: O(r * c)
public class Main {
    private static final int[][] DIRS = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
    public static int minDist(char[][] grid) {
        Queue<Point> q = collectSources(grid);
        for (int dist = 0; !q.isEmpty(); dist++) {
            for (int sz = q.size(); sz > 0; sz--) {
                Point p = q.poll();

                if (grid[p.r][p.c] == 'E') return dist;
                grid[p.r][p.c] = 'D'; // mark as visited

                for (int[] dir : DIRS) {
                    int r = p.r + dir[0];
                    int c = p.c + dir[1];
                    if (isSafe(grid, r, c)) {
                        q.add(new Point(r, c));
                    }
                }

            }
        }
        return -1;
    }

    private static Queue<Point> collectSources(char[][] grid) {
        Queue<Point> sources = new ArrayDeque<>();
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == 'S') {
                    sources.add(new Point(r, c));
                }
            }
        }
        return sources;
    }

    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;
        }
    }

    public static void main(String[] args) {
        char[][] grid = {
            {'S', 'D', '1'},
            {'1', 'E', '1'},
            {'S', 'E', '1'}};
        test(minDist(grid), 1);
        char[][] grid2 = {
            {'S', 'S', '1'},
            {'1', 'D', 'E'},
            {'E', '1', '1'}};
        test(minDist(grid2), 2);

        char[][] grid3 = {
            {'S', 'D', '1', 'D', 'E'},
            {'1', 'D', '1', 'D', '1'},
            {'1', '1', '1', '1', '1'},
            {'1', 'D', '1', 'D', '1'},
            {'S', 'D', '1', 'D', 'E'}};
        test(minDist(grid3), 8);

        char[][] grid4 = {
            {'S', 'D', '1', 'D', 'E'},
            {'1', 'D', '1', 'D', '1'},
            {'1', 'D', '1', '1', '1'},
            {'1', 'D', '1', 'D', '1'},
            {'S', 'D', '1', 'D', 'E'}};
        test(minDist(grid4), -1);
    }

    private static void test(int actual, int expected) {
        if (actual == expected) {
            System.out.println("PASSED!");
        } else {
            System.out.println(String.format("FAILED! Expected: %d, but got: %d",
expected, actual));
        }
    }
}
```

9. 前后台进程

Give a computer with total K memory space, and an array of foreground tasks and background tasks the computer need to do. Write an algorithm to find a pair of tasks from each array to maximize the memory usage. Notice the tasks could be done without origin order.

- Input The input to the function/method consists of three arguments : foregroundTask, an array representing the memory usage of the foreground tasks, backgroundTask, an array representing the memory usage of the background tasks, K, the total memory space of the computer.

- Output Return a list of pairs of the task ids.

- Examples 1 Input: foregroundTasks = [1, 7, 2, 4, 5, 6] backgroundTasks = [3, 1, 2] K = 6

Output: [(3, 2), (4, 1)]

- Explaination: Here we have 5 foreground tasks: task 0 uses 1 memeory. task 1 uses 7 memeory. task 2 uses 2 memeory... And 5 background tasks: task 0 uses 3 memeory. task 1 uses 1 memeory. task 2 uses 2 memeory... We need to find two tasks with total memory usage sum <= K. So we find the foreground task 3 and background task 2. Total memory usage is 6. And the foreground task 4 and background task 1. Total memory usage is also 6.

- Examples 2 Input: foregroundTasks = [1, 7, 2, 4, 5, 6] backgroundTasks = [3, 1, 2] K = 10

Output: [(1, 2))]

- Explaination: Here we find the foreground task 1 and background task 2. Total memory usage is 7 + 2 = 9, which is < 10.

10. Merge Files by Pairs(背景：合并音乐文件/组装零件)

给定n个文件， 和一个list of file size， 求minimum time to merge file

思路： PriorityQueue

将files中所有元素放入minHeap，每次poll两个最小的，merge之后放回minHeap。直至minHeap中只剩一个元素。输出循环中的累计时间。

复杂度：

时间： O(nlogn) 空间： O(n)

```java
public class MergeFilesByPairs {
    public int mergeFiles(int numOfSubFiles, List<Integer> files) {
        int resSum = 0;
        PriorityQueue<Integer> minHeap = new PriorityQueue<>((o1, o2) -> o1 - o2);
        for (int i = 0; i < files.size(); i++) {
            minHeap.add(files.get(i));
        }
        while (minHeap.size() != 1) {
            int a = minHeap.poll();
            int b = minHeap.poll();
            int temSum = a + b;
            resSum += temSum;
            minHeap.add(temSum);
        }
        return resSum;
    }

    public static void main(String[] args) {
        MergeFilesByPairs test = new MergeFilesByPairs();
        // TEST CASE1: files =  {8， 4， 6， 12}
        int numOfSubFiles = 4;
        List<Integer> list = new ArrayList<>();
        list.add(8);
        list.add(4);
        list.add(6);
        list.add(12);
        System.out.println("TEST CASE1:");
        System.out.println(test.mergeFiles(numOfSubFiles, list));

        // TEST CASE2: files =  {3, 1, 2}
        int n2 = 3;
        List<Integer> l2 = new ArrayList<>();
        l2.add(3);
        l2.add(1);
        l2.add(2);
        System.out.println("TEST CASE2:");
        System.out.println(test.mergeFiles(n2, l2));

        // TEST CASE3: files =  {8, 3, 5, 2, 15}
        int n3 = 5;
        List<Integer> l3 = new ArrayList<>();
        l3.add(8);
        l3.add(3);
        l3.add(5);
        l3.add(2);
        l3.add(15);
        System.out.println("TEST CASE3:");
        System.out.println(test.mergeFiles(n3, l3));
    }
}
```

11. Closest K destinations(背景：卡车送货)

Amazon Fresh is a grocery delivery service that offers consumers the option of purchasing their groceries online and schedule future deliveries of purchased groceries. Amazon's backend system dynamically tracks each Amazon Fresh delivery truck and automatically assigns the next deliveries in a truck's plan. To accomplish this, the system generates an optimized delivery plan with X destinations. The most optimized plan would deliver to the closest X destinations from the start among all of the possible destinations in the plan. Given an array of N possible delivery destinations, implement an algorithm to create the delivery plan for the closest X destinations.

Input

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

numDestinations, an integer representing the total number of possible delivery destinations for the truck (N);

allLocations, a list where each element consists of a pair of integers representing the x and y coordinates of the delivery locations;

numDeliveries, an integer representing the number of deliveries that will be delivered in the plan (X).

Output

Return a list of elements where each element of the list represents the x and y integer coordinates of the delivery destinations.

Constraints

numDeliveries <= numDestinations

Note

The plan starts from the truck's location [0, 0]. The distance of the truck from a delivery destination (x, y) is the square root of x2 + y2. If there are ties then return any of the locations as long as you satisfy returning X deliveries.

Example

Input:

numDestinations = 3

allLocations = [[1, 2], [3, 4], [1, -1]

numDeliveries = 2

Output:

[[1, -1],[1,2]]

Explanation:

The distance of the truck from location [1, 2] is square root(5) = 2.236

The distance of the truck from location [3, 4] is square root(25) = 5

The distance of the truck from location [1, -1] is square root(2) = 1.414

numDeliveries is 2, hence the output is [1, -1] and [1, 2]

题意：

有n个目的地(用坐标表示)， 卡车从坐标(0,0)出发，求出送货距离最近的K个点

思路：

1. 将所有的truck (0,0) -> location 的距离一一算出来，存入PriorityQueue里去
2. 根据numDeliveries的个数决定从PriorityQueue里弹出值

复杂度分析：

Time: O(n*logn)

因为每add一个元素到PriorityQueue里，需要logn时间，而我需要将n个元素add进PriorityQueue里。

Space: O(n)

因为我将allLocations 的每个元素都放入了minHeap里

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

public class ClosestXDestinations {
    List<List<Integer>> ClosestXdestinations(int numDestinations,
List<List<Integer>> allLocations,
                                                    int numDeliveries) {
        // CORNER CASE
        if (allLocations == null || allLocations.size() == 0 ||
allLocations.size() < numDeliveries) {
            return new ArrayList<>();
        }

        List<List<Integer>> result = new ArrayList<>();
        PriorityQueue<Position> minHeap = new PriorityQueue<>((o1, o2) ->
o1.distance - o2.distance);
        /*
           PART1: add each position into minHeap
         */
        for (int i = 0; i < allLocations.size(); i++) {
            List<Integer> list = allLocations.get(i);
            // Pythagorean Theorem
            int distance = list.get(0) * list.get(0) + list.get(1) * list.get(1);
            Position p = new Position(list, distance);
            minHeap.add(p);
        }
        /*
           PART2: grab the number of numDeliveries from minHeap
         */
        for (int i = 0; i < numDestinations && i < numDeliveries; i++) {
            result.add(minHeap.poll().list);
        }
        return result;
    }

    class Position {
        List<Integer> list;
        int distance;

        public Position(List<Integer> list, int distance) {
            this.list = list;
            this.distance = distance;
        }
    }

    /*
       TEST CASE
     */
    public static void main(String[] args) {
        ClosestXDestinations test = new ClosestXDestinations();
        int numDestinations = 3;
        int numDeliveries = 2;
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();
        List<Integer> list3 = new ArrayList<>();
        List<List<Integer>> allLocations = new ArrayList<>();
        list1.add(1);
        list1.add(2);
        list2.add(3);
        list2.add(4);
        list3.add(1);
        list3.add(-1);
        allLocations.add(list1);
        allLocations.add(list2);
        allLocations.add(list3);
        List<List<Integer>> result = test.ClosestXdestinations(numDestinations,
allLocations, numDeliveries);
        for (int i = 0; i < result.size() ; i++) {
            System.out.println("[" + result.get(i).get(0) + "," +
result.get(i).get(1) + "]");
        }
    }
}
```

【BFS 的题】

1. list of list 中 BFS 最短路径题。 0，1，9 组成 grid。从（0，0）出发，4 个方向走。0 能走，1 不能走。到达 9 的最小步数。 给一个二维数组，里面有 1，0，9，找从左上到有 9 的最小距离

参考解法：BFS

```

private static boolean bfs(int[][] maze, int startx, int starty) {
    if (maze == null || (maze.length < 1 || maze[0].length < 1)) return false;
    LinkedList<Point> que = new LinkedList<Point>();

    Point p1 = new Point(startx, starty, maze[startx][starty]);
    que.offer(p1);

    int width = maze[0].length;
    int height = maze.length;
    p1 = que.poll();

    while (p1.val != 9) {
        int x = p1.x;
        int y = p1.y;
        maze[x][y] = -1;

        if (x + 1 < height && maze[x+1][y] > 0) {
            que.offer(new Point(x+1, y, maze[x+1][y]));
        }
        if (x - 1 >= 0 && maze[x-1][y] > 0) {
            que.offer(new Point(x-1, y, maze[x-1][y]));
        }
        if (y + 1 < width && maze[x][y+1] > 0) {
            que.offer(new Point(x, y+1, maze[x][y+1]));
        }
        if (y - 1 >=0 && maze[x][y-1] > 0) {
            que.offer(new Point(x, y-1, maze[x][y-1]));
        }
        if (que.isEmpty()) break;
        else {p1 = que.poll();}
    }
```

```
        if (p1.val == 9) return true;
        return false;
    }
```

2. Maze 返回最短距离。在 maze 里求到某个点的最短距离，OA 里面 input 是 List<List>

参考答案：BFS

需要注意的特殊情况是第一个就是需要找的9，eg. {{9}}

3. 停车场里找 obstable 最短路径的题目。记得没找到 obstacle 要返回 -1，不然只会过 12 个 test case。加了这行就 16 个 test case 全过了，不需要再 optimize。

You are in charge of preparing a recently purchased lot for one of Amazon's new buildings. The lot is covered with trenches and has a single obstacle that needs to be taken down before the foundation can be prepared for the building. The demolition robot must remove the obstacle before progress can be made on the building. Write an algorithm to determine the minimum distance required for the demolition robot to remove the obstacle.

Assumptions: • The lot is flat, except for trenches, and can be represented as a two-dimensional grid. • The demolition robot must start from the top-left corner of the lot, which is always flat, and can move one block up, down, left, or right at a time.

• The demolition robot cannot enter trenches and cannot leave the lot.

• The flat areas are represented as 1, areas with trenches are represented by 0 and the obstacle is represented by 9.

Input

The input to the function/method consists of three arguments: numRows, an integer representing the number of rows; numColumns, an integer representing the number of columns; lot, representing the two-dimensional grid of integers.

Output

Return an integer representing the minimum distance traversed to remove the obstacle else return -1.

Constraints

1 <= numRows, numColumns <=1000

Example

Input:

numRows= 3

numColumns = 3

lot= [[1, 0, 0],

```
      [1, 0, 0],

      [1, 9, 1]]
```

Output:

3

题意：

在一个matrix中， 1 代表有路，0代表没路，9代表目的地。从左上角出发，求达到目的地9的最短距离。如果到达不了，返回-1。

注意几个边界条件：这题隐藏了一个条件就是机器人从左上角的(0,0)坐标出发，意味着(0,0) 坐标对应的值一定为1，否则直接无解返回-1。但有一个坑，如果出发点为9，则需return 0

思路：BFS

复杂度分析：

Time: O(numRows * numColumns)

我需要走完matrix的每个position

Space: O(numRows * numColumns)

我需要将matrix中的每个position的信息存入 matrix[][]

```java
import java.util.LinkedList;
import java.util.*;
import java.util.Queue;

public class RemoveObstacle {
    int removeObstacle(int numRows, int numColumns, List<List<Integer>> lot){
        int[][] matrix = new int[numRows][numColumns];
        boolean[][]visited = new boolean[numRows][numColumns];
        int result = 0;
        int[][]dirs = new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}};
        Queue<int[]> queue = new LinkedList<>();
        // Robot starts from top-left {0,0}
        queue.offer(new int[]{0,0});

        /*
           PART1: convert List<List<Integer>> lot -> matrix 2D grid
        */
        for (int i = 0; i < lot.size() ; i++) {
            List<Integer> sub = lot.get(i);
            for (int j = 0; j < sub.size() ; j++) {
                matrix[i][j] = lot.get(i).get(j);
            }
        }
        /*
           PART2: bfs
         */
        while(!queue.isEmpty()){
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int[] curr = queue.poll();
                int x = curr[0];
                int y = curr[1];
                if( x < 0 || y < 0 || x >=numRows || y >=numColumns || matrix[x]
[y] == 0 || visited[x][y]){
                    continue;
                }
                visited[x][y] = true;
                if(matrix[x][y] == 9 ){
                    return result;
                }
                for(int[] dir : dirs){
                    int x_ = dir[0] + x;
                    int y_ = dir[1] + y;
                    queue.offer(new int[]{x_, y_});
                }
            }
            result++;
        }
        return -1;
    }

    /*
       TEST CASE
     */
    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>();
        list1.add(1);
        list1.add(0);
        list1.add(0);
        List<Integer> list2 = new ArrayList<>();
        list2.add(1);
        list2.add(0);
        list2.add(0);
        List<Integer> list3 = new ArrayList<>();
        list3.add(1);
        list3.add(9);
        list3.add(1);
        List<List<Integer>> lot = new ArrayList<>();
        lot.add(list1);
        lot.add(list2);
        lot.add(list3);
        for (int i = 0; i < lot.size(); i++) {
            System.out.println(lot.get(i).get(0) + "," + lot.get(i).get(1) + "," +
lot.get(i).get(2));
        }
        RemoveObstacle test = new RemoveObstacle();
        int numRows = 3;
        int numColumns = 3;
        System.out.println(test.removeObstacle(numRows, numColumns, lot));
    }
}
```

4. Black and White

【MST】

1. MST：给你一堆 connection，求能链接所有城市的最小的 cost。 具体谷歌搜索 mst algorithm 就有类似的题。 （感觉这是 oa2 的标配。 做得时候最后有一个 testcase 想了很久， 就是如果不能成功的链接所有程序，例如用 unionfind 有一个 城市跟别的城市不一样组，就返回空 list）

其他描述： 一道 MST 的题的变形，搜一下 MST 的算法，感觉挺难的，我反正写了好久 好像 8 个 test case 吧。 一个 MST 的问题，写之前可以 google 一下求 MST 的算法，练一练。

2. min cost to construct 思路

这是个最小生成树（MST）问题。但要注意整个图中已经有一些边了，不是从0开始的最小生成树。具体来说，可以先Union-Find所有已经有的路 in roadsAvailable list，然后把所有可以建的路 in costNewRoadsConstruct list 按照 cost 排序放入 min-heap。然后每次从 min-heap 中拿出最小 cost 的路来接着 Union-Find整个图。每次需要Union的时候，累积目前为止的 cost。当总的 edges 数目等于总的 vertices 数目减 1 时，整个图就被构建成了一颗树。这时输入累积的cost作为输出。

注意： 这个题不太容易过所有的 test case （目前有19个test cases），因为有些坑需要避免。

1. 城市的ID是从1开始，不是从0开始。所以UnionFind的时候要多注意。
2. 输入的roadsAvailable list 和 costNewRoadsConstruct list 互相之间可能有重复。所以不要在算Graph中的edges 数目的时候要格外注意。

参考解法：最小生成树两种算法及 Union Find

2. Maximum Minimum Path。

【Tree】

1. 求最大 subtree。

【LinkedList】

1. 合并两个 LinkedList / merge two sorted linkedlist 关键词：package， belt。

【String】

1. count substring with k distinct characters 参考解法：用一个 int[26] 存 a-z 字母出现的 index，一旦长度到了 k，存下来

2. Longest String Without 3 Consecutive Characters Given A, B, C, find any string of maximum length that can be created such that no 3 consecutive characters are same. There can be at max A ‘a’, B ‘b’ and C ‘c’.

Example 1:

Input: A = 1, B = 1, C = 6 Output: “ccbccacc” Example 2:

Input: A = 1, B = 2, C = 3 Output: “acbcbc”

[leetcode] 358 Rearrange String K Distance Apart

```
class Solution {
    public String longestString(int A, int B, int C) {
        //initialize the counter for each character
        final HashMap<Character, Integer> map = new HashMap<Character, Integer>();
        map.put('a', A);
        map.put('b', B);
        map.put('c', C);

        //sort the chars by frequency
        PriorityQueue<Character> queue = new PriorityQueue<Character>(new
Comparator<Character>(){
            @Override
            public int compare(Character c1, Character c2){
                if(map.get(c2).intValue()!=map.get(c1).intValue()){
                    return map.get(c2)-map.get(c1);
                }else{
                    return c1.compareTo(c2);
                }
            }
        });

        //Adding into Queue
        for (char c: map.keySet()) queue.offer(c);

        int len = A + B + C;

        StringBuilder sb = new StringBuilder();
        while (!queue.isEmpty()) {
            int cnt = Math.min()


        }
    }
}
```

3. Longest String Made Up Of Only Vowels

You are given with a string . Your task is to remove at most two substrings of any length from the given string such that the remaining string contains vowels(‘a’,‘e’,‘i’,‘o’,‘u’) only. Your aim is to maximise the length of the remaining string. Output the length of remaining string after removal of at most two substrings. NOTE: The answer may be 0, i.e. removing the entire string.

- Input: earthproblem letsgosomewhere
- Sample Output 3 2

```
private boolean isVowel(char c){
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}

public int longestString(String s) {
    int len = s.length();
    int start = 0, end = len - 1;
    while (start < len && isVowel(s.chartAt(start))) start++;
    while (end >= 0 && isVowel(s.chartAt(end))) end--;
    // checking area comes to [start,end]
    if (start >= len) return len;
    int res = start + len - 1 - end;
    int longest = 0, sum = 0;
    for (i = start + 1; i <= end; i++) {
        if (isVowel(s.chartAt(i))) {
            sum ++;
        } else {
            sum = 0;
        }
        longest = Math.max(sum, longest);
    }
    return longest + res;
}
```

4. Longest substring with at most K distinct characters

- Input: The input to the function/method consists of two arguments -
    - inputStr , representing the string;
    - num , an integer representing the number, K.
- Output: Return distinct substrings of input string of size K with K distinct characters.
- Constraints: 0 ≤ num ≤ 26
- Examples
    - Input:
        - inputStr = awaglknagawunagwkwagl
        - num = 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.

```
public class Main {

    public static List<String> kSubstring(String s, int k) {
        Set<Character> window = new HashSet<>();
        Set<String> result = new HashSet<>();
        for (int start = 0, end = 0; end < s.length(); end++) {
            for (; window.contains(s.charAt(end)); start++) {
                window.remove(s.charAt(start));
            }

            window.add(s.charAt(end));

            if (window.size() == k) {
                result.add(s.substring(start, end + 1));
                window.remove(s.charAt(start++));
            }
        }
        return new ArrayList<>(result);
    }

    public static void main(String[] args) {
        System.out.println(kSubstring("awaglknagawunagwkwagl", 4));
    }
}
```

5. longest palindromic substring

```
  public String longestPalindrome(String s) {
        int len = s.length();
        String result = "";
        // Defines state
        boolean[][] dp = new boolean[len][len];
        for (int i = len - 1; i >= 0; i--) {
            for (int j = i; j < len; j++) {
                dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i+1][j-
1]);
                if(dp[i][j] && (result.length() < j - i + 1)) {
                    result = s.substring(i, j+1);
                }
            }
        }
        return result;
    }
```

6. Find pair with maximum Appeal value

Input : Array Output : return index of two pairs {i, j} (i = j allowed) Appeal = A[i] +A[j] + abs(i-j)

Example input: {1, 3, -1} output: {1, 1} Appeal = A[1] + A[1] + abs(0) = 3 + 3+ 0 = 6