2nd LGE Code Jam 2019 - Problem E (English)

Created by 박선현 sunhyun.park, last modified by 배성호 baver.bae on 2019/07/12

Subtask 1)

This can be considered as a maximum matching problem in bipartite graphs.

Since N, M <= 100, we have V <= 200, E <= 10000, So well-known O(VE) algorithms like Edmonds-Karp, Ford-Fulkerson is enough.

Subtask 2)

Bipartite matching doesn't work here, since N, M <= 100000 gives O(VE) = 10^15 and O(V√E) >O(10^12).

Then how about considering greedy solution? Consider the following approach:

1. Store information of passengers with balanced binary search tree with **(distance, number of people)** pairs:
   You can use C++ std::map, Java TreeMap
2. Sort cabs first in **ascending order by its endpoint**. For tiebreakers, sort in **descending order by its starting point** (only when endpoints equals)
3. Make decisions from the leftmost (which having the **minimum endpoint**) cab.
   To do so, first search a passenger **nearest from its start point** who can **actually take this cab**.
   You **should use binary search** to reduce time complexity. (C++ - std::map::lower_bound, Java - TreeMap.ceilingEntry)
   When the passenger-cab pair is matched, each values of entries is decremented. Don't forget erase the entry when the number of people with a typical distance becomes zero.
4. If we iterate over all cabs and apply 3., the number of maximum matching is the answer!!

Were we just lucky? Or is this really right?

Actually this is the optimal solution.

(Of course you can get the similar correct solution by sorting everything in the reversed order. The important thing is you should use sorting + greedy idea)

If we sorted **passengers distances' in ascending order**, we have to make decisions from **the cab which has the least capability for taking passengers** to get the optimal solution.

For this, at first, sort it in ascending order by its endpoint, and for tiebreakers, sort in descending order by its start point, which is the only way to "restrict" possible options of selecting passengers.

To check if this cab can actually take the passenger, pick the passenger nearest from the start point of cab = leftmost passenger this cab can take.

Therefore, if some people belongs to multiple ranges of cab, **we are always giving the priority to the cab which has the least capability,** so our solution is globally optimal.

Complexity : Building TreeMap O(N log N) + Sorting Cabs O(M log M) + Finding passengers O(M log N) = O(M log M + (M + N) log N)

Java C++

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.Arrays;
import java.util.Map;
import java.util.TreeMap;

public class Main {

    public static class Cab implements Comparable<Cab> {
        public Cab(int _start, int _end) {
            start = _start;
            end = _end;
        }
        int start;
        int end;

        @Override
        public int compareTo(Cab other) {
            if (this.end - other.end != 0) {
                return this.end - other.end;
            } else {
                return -this.start + other.start;
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        int TC = Integer.parseInt(br.readLine());

        for (int i = 0; i < TC; i++) {
            String str[] = br.readLine().trim().split(" ");
            int N = Integer.parseInt(str[0]);
            int M = Integer.parseInt(str[1]);

            TreeMap<Integer, Integer> peopleMap = new TreeMap<Integer, Integer>();

            String peopleString[] = br.readLine().trim().split(" ");
            for (int j = 0; j < N; j++) {
                int personDist = Integer.parseInt(peopleString[j]);
                peopleMap.put(personDist, peopleMap.getOrDefault(personDist, 0) + 1);
            }

            Cab cabs[] = new Cab[M];
            for (int j = 0; j < M; j++) {
                String cabString[] = br.readLine().trim().split(" ");
                int start = Integer.parseInt(cabString[0]);
                int end = Integer.parseInt(cabString[1]);
                cabs[j] = new Cab(start, end);
            }
            Arrays.sort(cabs);

            // find maximum matching
            int matching = 0;
            for (Cab cab : cabs) {
                Map.Entry<Integer, Integer> nearest = peopleMap.ceilingEntry(cab.start);
                if (nearest != null && nearest.getKey() <= cab.end) {
                    matching++;
                    int dist = nearest.getKey();
                    int count = nearest.getValue();
                    if (count == 1) {
                        peopleMap.keySet().remove(dist);
                    } else {
                        peopleMap.replace(dist, count - 1);
                    }
                }
            }
            bw.write(Integer.toString(matching) + "\n");
        }
        br.close();
        bw.close();
    }
}
```