ADA Assignment 3 Theory

Ayushi Jain 2019031

Kesar Shrivastava 2019051

Problem 1.

We use binary search to find the maximum possible duration between two days of rain and then we use a function to check if it is possible to rain given k number of times using that particular duration as the maximised minimum duration.

Algorithm:

We apply binary search over the given dates and then if it is possible to rain on k days using that particular value as the maximum number of days then we check if we can further maximise the number of days. However, if it is not possible to rain on k days then we need to find a lower value of the number of days and then again check if it is still possible to rain on k days from the given n days.

Pseudocode:

```
public class Assign3 {
    static int check(int []arr, int mid, int n, int k, int[]
sol) {
        int festival = 1;
        int currFestival = arr[0];
        int i = 1;
        int index = 1;
        while (i < n) {
            if (arr[i] - currFestival >= mid) {
                sol[index] = arr[i];
                index+=1;
```

```
            currFestival = arr[i];
            festival+=1;
            if (festival == k)
                return 1;
        }
        i+=1;
    }
    return 0;
}

public static void main(String args[]) throws IOException {
    Reader.init(System.in);
    int n = Reader.nextInt();
    int k = Reader.nextInt();
    int []arr = new int[n];
    for(int i = 0; i<n; i++){
        arr[i] = Reader.nextInt();
    }
    int ans = 0;
    int lo = 0;
    int hi = arr[n-1];
    int mid;
    int[] sol = new int[k];
    sol[0] = arr[0];
    while (lo <= hi) {
        mid = (lo + hi) / 2;
        if (check(arr, mid, n, k, sol)==0)
            hi = mid - 1;
        else {
            ans = Math.max(ans, mid);
            lo = mid + 1;
    }
```

```
    }
        System.out.println(ans);
        for(int i: sol){
            System.out.print(i+" ");
        }
    }
}
```

Proof:

We binary search over the possible duration and try to rain over k days with searched duration as the maximised minimum duration. If it is possible then we check if we can further maximise the duration else we stop.

Proof using "Greedy stays ahead" by arguing that at every recursive call we never make a bad choice and hence the Greedy solution is as good as the optimal solution.

In the first call the binary search returns mid, now there are two possible cases: *check* returns true and in the other case it will return false.

In the *check* function we greedily check over every day if it is at a duration of at least d from the previous rainy day. If it returns false, then we will have to check for the lower values than mid because then the number of days on which we can rain over will be less than k. Since the duration is more than the actual answer, while checking if this is a possible answer we span over more number of days and hence cannot get the optimal solution. Because if the duration between two consecutive days is not at least d then it will not be added in the solution. This can be proved using contradiction by the following argument:

Let the minimum duration between any two consecutive days of rain that the greedy produces be d and suppose that this is not the maximum hence this can be maximized

further. This implies that with this duration as the minimum duration we can rain over k days in the Greedy solution but k+z days in some other sequence

(**Disclaimer:** This will be possible in most of the cases but not all cases. If it is not possible then it follows from the initial argument that it will not be possible to rain over exactly k days but less than that and hence we need to recurse over the left subarray. Because when d will be increased it will span more number of days in it than the Greedy solution but for the same d when it is not maximised we can rain over more than k days in the other sequence of days. Thus, using binary search is useful because we are always checking for the possibility if the greater value of d can satisfy our given condition.)

Suppose that X and X* are the greedy and the other solutions respectively. We need to prove that with the minimum duration d that is actually the maximised minimum, we can rain over only k days and not more than that.

$$X = u_1, u_2, ..., u_k$$

$$X^* = y_1, y_2, ..., y_k, y_{k+1}, ..., y_{k+z}$$

Since $X \neq X^*$, let this be true

$$u_i = y_i \quad \forall\ i <= m$$

And thus,

$$u_{j+1}-u_j = y_{j+1}-y_j = d \qquad \forall\ 1 <= j <= m-1$$

Now, we need to rain over k-m days in X and k+z-m days in X* but since the number of total days is the same for both the solution, the minimum duration between two consecutive rainy days in X* after the m-th day would be less than that of Greedy.

If the remaining days out of n given days is w then the number of rainy days in X is k-m while in X* is k-m+z. Now, by Pigeonhole Principle, there will be one such pair of rainy days in X* whose in between duration would be less than that of Greedy because we have to fill k-m in w in X and k-m+z in X*.

$$k-m+z > k-m$$

Hence the minimum duration for X* would get updated to d' such that d' < d, which is a contradiction as we assumed that the maximised minimum duration possible is d.

In other words, if we assume that we recurse over the right subarray when the function returns false then it will not be possible to get k out of n days because it has already returned false. It has returned false because with d as the minimum distance it could not rain over k days and hence now we need to look for d in the left subarray.

Also, the function *check* governs which subarray to recurse upon. If it returns true then the algorithm checks if it is possible to further maximize the duration hence we recurse over the right subarray but if we maximise d more than required then again *check* will return false because again it will not be possible to rain over k days but less than that, this follows from the above argument. But suppose if we recurse over the left subarray when it was actually possible to rain over k days using then we will essentially be minimising the value of and hence going away from the optimal which is a contradiction hence the *check* function is correct.
However, now if *check* returns true, then we again recurse over the rightmost half of the halved right subarray to further check if d can be maximized.
This implies that at every recursive call we are going towards the optimal solution and hence not making any bad choice.

This proves that our algorithm is correct.

Problem 2.(a)

We can use Prim's algorithm to find the maximum spanning tree of the given edge-weighted graph. Prim is a greedy algorithm that helps to find the spanning tree of a graph.

We need to build the tree from any arbitrary vertex and then add the maximum edge weight from the tree to the next vertex.

The algorithm is described below:

- Initialise an array ancestor[ ] to store the tree.
- Initialize an array visited[ ] to keep track of the visited vertex in the graph. Initially all the values are 0.
- Initialize an array cost[ ] to store the maximum cost to connect the vertex to the tree.
- Since the first vertex has no ancestor we initialize the ancestor of this vertex as -1.
- Now, we pick a vertex having maximum cost from the unvisited vertices and mark it as visited.
- Also, we will need to update the cost of all the unvisited adjacent vertices of the vertex in consideration. We iterate over all the unvisited vertices from the vertex and if the cost between the vertex and the neighbour vertex is greater than the previous value of the vertex we update the value of vertex with that cost.
- Here ancestor[i] with i is the edge having weight graph[i][ancestor[i]].

```
MaximumSpanningTree(graph[1...V][1...V], V){

      visited = new int[V];
      cost = new int[V];
      ancestor = new int[V];

      for i = 2 to V{
            cost[i] = min_value;
      }
```

```
        cost[1] = max_value;
        ancestor[1] = -1;

        for i = 1 to V-1{
                index = -1;
                maxCost = min_value;

                for i = 1 to V{
                        if(visisted[i]==0 && cost[i]>maxCost){
                                maxCost = cost[i];
                                index = i;

                        }

                }

                visited[index] = 1;

                for j = 1 to V{
                        if(graph[j][index]>cost[j] && visited[j]==0){
                                cost[j] = graph[j][index];
                                ancestor[j] = index;
                        }
                }

        }

}
```

Complexity: The time complexity is O(V^2) where V is the number of vertices.

Proof of correctness:

The algorithm we propose is same as the Prim's algorithm except for the following changes:

1. The initial distance of all nodes will be -INFINITY

2. The greedy algorithm will choose the unvisited vertex at the greatest distance from the already existing set of vertices,
3. The vertex distance will be updated when the present distance of the vertex is lesser than the distance to be updated

So, the algorithm will generate a spanning tree (as proved in lectures), as our algorithm has the same loop invariant with only the variation of maximum, minimum updation and initial values of the nodes.

So, we now prove that the spanning tree is indeed the maximum spanning tree using the exchange argument.

Say, we have two sets of vertices S and S′, suppose the edge e is the maximum cost edge connecting S and S′. We assume that e is not a part of T*.

But then since S and S′ form a cut, by empty cut lemma there must be an edge crossing this cut, let that edge be f and let e be a part of T+

cost(T+) = cost(T*) - cost(f) + cost(e)

Since cost(f) < cost(e)

cost(T+) > cost(T*)

In this case we now have another spanning tree T+ which has more weight than the original spanning tree

This is a contradiction and hence our assumption was wrong.

-> e is a part of the T*

But it may happen that after removing f and joining e the tree gets disconnected.

To remove this flaw:

We consider a subgraph T* ∪ {e}

This contains a cycle involving e.

By double cross lemma there exists another edge e′ in the cycle crossing that (S, S′)

Now, T+ = T* ∪ {e} - {e′}

cost(T+) = cost(T*) + cost(e) - cost(e′)

So T+ is a spanning tree and cost(T+) > cost(T*)

A contradiction, hence our assumption is wrong.

Thus, the algorithm produces a maximum spanning tree of any given graph.

Problem 2.(b)

Suppose we have a spanning tree T with a suboptimal path between two vertices s and t. Suboptimal path here refers to the path whose total cost is not the maximum. In other words, this particular path is not the widest path between s and t.

Now, we remove an edge e' from the path which is the width of the path. Since T is a tree, it now becomes disconnected in two sets of vertices, say S and S'. WLOG let s be in S and t in S' then (S, S') becomes a cut, by the empty cut lemma there is no edge crossing this cut. Effectively s and t are not connected anymore so we need to introduce an edge to make the tree connected. Suppose we add an edge from the widest path in the given graph. By lonely edge corollary, we can add an edge and this would not be part of any. There will be no cycle either because we add an edge after removing one edge. Also, when we add the edge from the optimal path then it is ensured that both the sets are connected since we added the edge from a path between s and t to the tree which we intently made disconnected. This results in a tree say T*.

In the optimal path the width is maximum and hence cost(e') is less than any other edge in the optimal path.

cost (T*) = cost(T) - cost(e') + cost(e)

Since, cost(e') < cost(e)

=> cost(T*) > cost(T)

This is a contradiction as the new spanning tree has cost strictly more than the original maximum spanning tree. Hence, our assumption is false and the maximum spanning tree of the given graph contains widest paths between every pair of vertices.