Member 1 : Kishan Sinha
2019428
Member 2 : Abhishek Chaturvedi
2019401

Problem 1:

Code:

```
#include<bits/stdc++.h>
using namespace std;



int ans = 0, n, k, a[100005];

int isPossible(int m) {
    int dates = 1, cur = a[0];
    for(int i = 1; i < n; i++) {
        if (a[i] - cur >= m) {

            dates++;

            cur = a[i];

            if (dates == k)
                return 1;
        }
    }
    return 0;
}

void binarySearch() {
    int l = 0, h = a[n - 1] - a[0];
    while (h >= l) {
        int m = l + (h - l) / 2;
        if (isPossible(m)) {
            ans = max(ans, m);
            l = m + 1;
        }
        else {
            h = m - 1;
        }
    }
}

int main() {
    cin >> n >> k;
    for(int i = 0 ; i < n ; i++ ) {
        cin >> a[i];
    }
    sort(a,a+n);


    binarySearch();
    cout << ans;
}
```

Proof of Correctness:

Description of the code:
We run a function called 'binarySearch' in which the following happens:

- First we initialize *m/2* with the max number of days of gap possible / 2 which is m/2 .
- Now we are going to check whether it is possible to have k rainy days with a gap of this m/2 days.

- We do this checking with a simple function called ‘isPossible’ which runs a simple loop to count the number of rainy days with the given number m/2.
- **If it is possible to have k rainy days it means this gap value = m/2 is a candidate answer.**
- **If it is not possible to have k rainy days it means then we binary search on the (m/2-1)/2 number of days and so on.**
  This is a valid argument because when it was not possible to have k rainy days with the given arrangement it meant the gap was too large to fit k rainy days.
- We finally take the max over all candidate answers which will be our final answer.

  Proof of our approach by cases:

Case 1: We have a gap period which has k rainy days possible. We already have a candidate answer. Now since we have an answer, we would like to an even better solution hence we raise our gap day limit. We do so by increasing the value of *l* to mid value +1. So now the new gap period will be: (m/2+1)/2 (if original was m)

Case 2: If it is not possible to have k rainy days. Since we are not able to fit k rainy days in the given gap day period. We reduce the gap period by lowering the value of h to m-1 . This way we may be able to fit k rainy days with the dates. The new gap period will be (m/2-1)/2.

Problem 2:

a) In this problem we need to find the Maximum Spanning Tree of a given Weighted Graph.
   Idea: We can modify Prim’s Algorithm to find the maximum Spanning Tree.
   Prims Algorithm originally uses a Greedy Approach to find the minimum Spanning Tree.
   We can modify the given graph by multiplying all the weights with -1, in this way the relative ordering of the weights will be reversed.
   As, if $x>y$, then $-x<-y$.
   Now if we find the Minimum spanning tree of this modified Graph we will get the

Maximum Spanning tree of the Original Weights.
In Prims Algo, at a particular point the minimum of all the active edges is taken into consideration for the MST edge. Now since the values are negated, the maximum value of the original active edges' weights will be taken into account. In this way, by this simple hack we can find the MST.
For the final answer we will just negate the answers to get the original values.

b)
We will try to prove this by the method of contradiction.

Contradictory Assumption: The Maximum Spanning Tree (T) of a Graph G, does not contain the widest path between some pair of vertices.

Let **A** and **B** be one of the pairs of such vertices in T, such that the minimum weighted edge is **c** in A and B 's widest path.

W(T) means the weight of the Maximum Spanning Tree

Let **d** be the minimum weighted edge in the path from **A** to **B**.

Now, |d| < |c| because we have assumed that T doesn't contain the widest path between vertices A and B ([|ed| = weight of edge ed])

We remove the edge d from T.
In doing so we are dividing T into two connected parts T-S and S, with vertex A belonging to S and vertex B belonging to T - S.
Note that S and T-S are connected wrt edge d.
Now, let's try and add edge c to this disconnected graph, and we see that two cases come forward:
Case i):
One vertex of edge y is in S and the other vertex is in T - S.
So we obtain a new Maximum Spanning Tree T'.
W(T') = W(T) - |d| + |c|
We know that |d| < |c| since W(T') > W(T)
We get a contradiction that T is a Maximum Spanning Tree of G.

Case ii):
In this situation the two (both) of the vertices of **c** belong to the same connected

component (either S, T - S)
Hence therefore, a cycle will be formed in this connected component (this is due to the reason that there is already one path between any two vertices and adding c would create another path between them which will result in a cycle).
Now our task is to remove an edge to get rid of the cycle.
Logically it makes sense to remove the minimum weighted edge to obtain the Maximum Spanning Tree we want in the end.
Let the removed edge be **e**.

|e| ≤ |c| is implied because **c** was introduced to this connected component, hence the removed edge weight has to be at most |c|.

Now picking an edge from G, where one vertex is in T-S and another vertex in S, which has maximum weight among all edges called **'f'**

|f| ≥ |d| is implied (x connects these two disconnected
components; because it was the one which was responsible for disconnecting it)

Connecting **"w"** we would get a Maximum Spanning Tree T'.

Weight of T': W(T') = W(T) - |d| + |c| - |e| + |f|
⇒ W(T') = W(T) + (|c| - |e|) + (|f| - |d|)
Note, W(T') > W(T) (as |c| ≥ |e| and |f| ≥ |d|)
This contradicts our assumption that T is a Maximum Spanning Tree for G.

From the above cases we can infer that a Maximum Spanning Tree without the widest path between vertices A and B does not exist.
Therefore, the Maximum Spanning Tree must have the widest path between all pairs of vertices.