Homework Assignment 3 (Theory)

Submitted by
Vidhi Sharma 2019286
Dolly Sidar 2019304

1. **a) Pseudocode:**

```
binarySearch( Array A, n, k)
{
    ans = 0
    l = 0;
    r = A[n-1] ;
    while( r >= l){
    mid = (l+r)/2 ;
    flag=0;
    first= A [0];     rains=1;

    // to tell if it's feasible
       for i = 1 to  n do
              if (mid <= A[i] - first)
              {
                 first = A[i];
                 rains++;
                 if (rains == k)
                 {
                    flag=1;
                    break;
                 }
              }
           end for
    // if flag==1 then it's feasible, otherwise not feasible

           if (flag==1)
           {
              l = mid + 1;
              if(ans<mid)
                ans=mid;
           }
           else
           {
              r = mid-1;
           }
       }
       return ans;
       }
```

Runtime: O (n log n)
As a binary search would take O(log n) time and O(n) for checking if it is possible to pick all k festivals out of the n festival.

Proof by Contradiction :
Suppose σ be the maximum-minimum duration produced by the binarySearch algorithm written by us.
Suppose σ* be the optimal maximum-minimum duration such that σ != σ*

Let the σ* be different from σ in such a manner that in one iteration we did
l=mid+1 in σ and did r=mid-1 in σ*
Or
r=mid-1 in σ and did l=mid+1 in σ*

Case 1: In one iteration: l=mid+1 in σ and did r=mid-1 in σ*
In this case since to get σ solution, we did l=mid+1,
mid is a feasible solution i.e. if k such festivals can be selected using mid as the maximum-minimum duration.
But to get σ* solution we did,
r=mid-1.
This will decrease the value of mid in the next iteration:
Since, mid is a feasible solution to select k festivals,
all m1<mid will also be feasible
This is because, when we check feasibility we check
If (mid <= A[i] - first) to select a festival
Now if m1< mid
Then the condition (m1 <= A[i] - first) will be checked.
for any case
if mid <= A[i] - first, then

m1< mid <= A[i] - first

This means that if the condition (mid < A[i] - first) was fulfilled k times for mid, then it will be fulfilled k times for m1<mid.

But we are decreasing optimal solution in this case (m1< mid) and therefore, mid will be the maximum-minimum feasible duration till now not m1<mid. (since mid is greater and feasible)

But, if we keep **mid = mid or increase mid in the next iteration** such that:

m2>=mid is feasible (m2 is maximum between m2 and mid)

In this case mid <= m2 <= A[i] - first (So, that (m2 <= A[i] - first) can be fulfilled k times, thereby selecting k festivals)

In this case, m1<mid<=m2

So, m1 < m2

So, optimal solution σ* < optimal solution σ

Since the optimal solution of some other algorithm is greater than the optimal solution σ*.

This contradicts the definition of the optimal "maximum" minimum duration σ*.

So, σ is the optimal solution.

Case 2: In one iteration: r=mid-1 in σ and did l=mid+1 in σ*

In this case since to get σ we did r=mid-1,

mid is not a feasible solution i.e. if k such festivals cannot be selected using mid as the maximum minimum duration.

But to get σ* we did,

l=mid+1.

This will increase the value of mid in the next iteration:

If mid is not a feasible solution to select k festivals then,

all m1>mid will also not be feasible

This is because, when we check feasibility we check

if (mid <= A[i] - first) to select a festival

Now if m1> mid

Then the condition if (m2 <= A[i] - first) will be checked.

for any case

if mid > A[i] - first, then

m1> mid > A[i] - first

This means that if the condition mid > A[i] - first was not fulfilled k times for mid, then it will not be fulfilled k times for m1>mid.

So, in σ* we will never find the maximum-minimum duration.

But, if we decrease mid in the next iteration
There will be some m2<mid which is feasible
Such that   m2 <= A[i] - first < mid (So, that (m2 <= A[i] - first) can be fulfilled k times, thereby selecting k festivals)
So, In this case we will find some optimal solution σ.
Since we won't find an optimal solution in σ *, it contradicts the definition of the optimal "maximum" minimum duration σ*.
So, σ is the optimal solution.

Q2 a)
A maximum spanning tree is a spanning tree of the graph which has the maximum weight.
We already know the algorithm for the **Minimum spanning tree using Prim's Algorithm**. Now in order to find the maximum spanning tree we can use the same algorithm with slight modifications in the original graph.

- For every edge multiply its corresponding weight with -1.
- Use Prim's algorithm to get the minimum spanning tree of this modified graph.
- Again Multiply the weights in the spanning tree with -1. This will assure that we get the maximum spanning tree with original weights.

This algorithm works because when we multiply the weights with -1 we make the sort order of weights opposite to the original order. If we find the minimum spanning tree with this opposite order of weights in the graph, then that minimum spanning tree will have the minimum weighted sum but if we multiply the weighted sum with -1 again, it will become the maximum weighted sum for the graph with the original order of weights.

This is based on the simple fact that:
If   a > b
We multiply both sides by -1
Then   -a < -b
Similarly if there is an increasing sequence:
        a1<a2<a3<a4<=a5
We multiply all ai with -1
Then   -a1> -a2> -a3> -a4>= -a5

Pseudocode:

```
initial= {0};
final = {1, 2, ..., N-1};
```

```
SpanningTree = {};
Cost [N+1][N+1];
Cost[0][0]= -inf;
for i=1 to N
        For j=0 to N
                Cost[i][j]= -1* Cost[i][j];
while ( final ≠ empty )
{
     Find edge e = (x, y)  which has the smallest Cost[x][y]
     SpanningTree = SpanningTree ∪ {e};
     initial   = initial ∪ {y};
     final= final - {y};
}
For all e = (x, y) edges in SpanningTree
{
        Cost[x][y]= -1*Cost[x][y];
}
```

Time Complexity of the algorithm is the same as the minimum spanning tree using Prim's algorithm which is O(V^2).

Q2 b)

To Prove: Prove that the maximum spanning tree of G contains the widest paths between every pair of vertices.
Proof by Contradiction :
Proof:
Suppose σ be the original spanning tree that contains the widest paths between every pair of vertices.
Suppose σ* be the maximum spanning tree(optimal) such that σ != σ*

Claim 1: There exist vertices X and Y with the widest path between them containing at least one edge, not in σ*.

Proof: Suppose this is not true. This means that any two vertices X and Y with the widest path between them have all its edges in σ*.

But this is only true for the spanning tree σ and hence contradicts the assumption that σ != σ*.

Construct another spanning tree σ'.

Claim 2: Let there be another spanning-tree σ' such that:

Proof:
In order to construct σ', remove the edge with the lowest weight edge R on the widest path between X and Y in the spanning tree σ*. Since, a spanning tree is a tree, it has no cycles. So, now nodes X and Y are not connected.

Then we add an edge R' from the widest path into the spanning tree σ' in such a way that there is no loop.
X and Y are not connected, so when we add an edge cycle is not formed. Also, each path between X and Y should have some edge between the two not connected nodes.

Weighted sum of σ' = Weighted sum of σ* - weight of R + weight of R'

weight of the edge R = the path's width of σ*,
and since σ' is such that it has the edge in the widest path between X and Y but not in σ*. We can say that,
Weight of each edge R' in σ' > Weight of each edge R in σ*.
So,
Weighted sum of σ' > Weighted sum of σ*
This contradicts the definition of the original "maximum" spanning tree σ*.
Therefore, σ* is not the optimal maximum spanning tree.