Algorithm Design and Analysis (CSE222) Winter 2021 Homework - 3

Abhimanyu Gupta - 2019226
Abhinav Gudipati- 2019227

<u>Question - 1</u>

Algorithm (Pseudocode) :

```
# n: number of festivals
# festivals: array of festivals days
# k: number of days to rain on
procedure rainDays(n, festivals, k)

        minimumGap = — ∞

        lowGap = 1
        highGap = ( festivals[n] - festivals[1] )

        while lowGap <= highGap do
                gap = ⌊ (highGap + lowGap) / 2 ⌋

                rainsLeft = k - 1
                festiveDay = 1
                while festiveDay ≤ n and rainsLeft > 0 do
                        nextRainDay = festivals[festiveDay] + gap
                        while festiveDay <= n and festivals[festiveDay] < nextRainDay do
                                festiveDay += 1
                End while
                If festiveDay ≤ n then
                    rainsLeft -= 1
                End if
            End while

            if rainsLeft == 0 then
                minimumGap = gap
                lowGap = gap + 1
            else
                highGap = gap - 1
            End if
        End while

        rainDaysList = [ ] # list of days on which it should rain
        day = 1
        rainDaysList.append(festivals[day])
        for rains = 1 to k - 1 do
            nextRainDay = festivals[day] + minimumGap
            while festivals[day] < nextRainDay do
                day += 1
            End while
            rainDaysList.append(festivals[day])
        End for

        return rainDaysList
End procedure
```

Proof of Correctness:

Through binary search, we are computing a possible value for the number of days apart which we would like to schedule rains on k festival days (gap between rainy days).

After computing such a gap for the rainy days, we are checking whether we can have k rainy days in our n festival days, with each consecutive rainy day being apart at least the gap we predicted.

There are two outcomes of the above checking:
Case 1: We can schedule k such rains.
In this case, the computed gap of the days would be the answer (in fact the best we have computed so far), therefore we would update the concerned variable to store this answer.

Now, as we have an answer, we would like to increase it further. Therefore we would raise our lower limit for search space(lowGap), which would directly raise the value of the gap in the next iteration. Therefore, creating a chance to find a more optimal value (increase in the number of days).
Case 2: We can't schedule k such rains.
As we cannot schedule k possible rainy days with the computed gap, we would try to reduce this gap value in order to accommodate the k number of rainy days. For this, we need to reduce the possible gap value in the following iteration, we do so by decreasing the high limit (highGap).

Proof of Cases:
Suppose, in any iteration, the value of lowGap = l and highGap = h
$\therefore$ gap $g = \lfloor (l + h) / 2 \rfloor$

- If this g satisfies the desired condition, we would be resetting the value of l to a higher value, as per the oracle.
  So, for the next iteration lowGap:
  $l` = g + 1$
  and highGap:
  $h` = h$
  $\therefore$ new gap:
  $g` = \lfloor (l` + h`) / 2 \rfloor$
  $\Rightarrow g` = \lfloor (g+1 + h) / 2 \rfloor$
  It is clear $g` \geq g$ (as $g \geq l$), so the oracle is guiding us towards a better optimal value.
- If this g doesn't satisfy the desired condition, we would be resetting the value of h to a lower value, as per the oracle.
  So, for the next iteration lowGap:
  $l` = l$
  and highGap:
  $h` = g - 1$
  $\therefore$ new gap:
  $g` = \lfloor (l` + h`) / 2 \rfloor$
  $g` = \lfloor (l + g - 1) / 2 \rfloor$
  It is clear $g` < g$ (as $g \leq h$), so the oracle is guiding us towards a solution (optimal).

From the above two points, it is clear that the oracle is guiding us towards the best solution.

Question - 2(a)

Algorithm:

Multiply all the edges' weights by -1, and run Kruskal's Algorithm to find the Minimum Spanning Tree on this modified graph.

Why it Works:

- Kruskal's Algorithm greedily checks the minimum weighted edges to be added in the Spanning Tree.
- If the algorithm was run without multiplying the weights by -1, then the edges would have been checked in the increasing order of their weights.
- By multiplying all the edges' weight by -1, we are effectively inverting this ordering of the edges. [$a < b \Rightarrow -a > -b$]
- Now, as a result, the heaviest edges would be checked first and so on.
- So, the obtained Minimum Spanning Tree (after multiplying weights by -1) would contain the heaviest of the edges.
- Hence, in actuality, a Maximum Spanning Tree would be obtained.

Question - 2(b)

Let's assume, on the contrary, that the Maximum Spanning Tree (T) of a Graph G, does not contain the widest path between some pair of vertexes.

Say, one of the pairs of such vertexes is U and V, with the minimum weighted edge being y in their widest path.
Let the weight of the T (Maximum Spanning Tree of G), be denoted by c(T) and the minimum weighted edge in the path from U to V in this T be x.
Now, $|x| < |y|$ [$|e|$ = weight of edge e] (since T doesn't contain the widest path from U to V)

Let's remove the edge x from T.
By Removing this edge, we are cutting T into two connected components S and T - S, with vertex U being in S and vertex V being in T - S.
We need to bear in mind that S and T-S are connected with respect to edge x.

Now, let's add edge y to this disconnected graph, there can be two possibilities:
Case 1: One vertex of y belongs to S and the other vertex belongs to T - S
In this case, a new Maximum Spanning Tree would be obtained.
Let's call this Maximum Spanning Tree T'.
Now weight of T', $c(T') = c(T) - |x| + |y|$
As, $|x| < |y|$ $\therefore$ $c(T') > c(T)$
This contradicts the fact that T is a Maximum Spanning Tree of G.
Case 2: Both the vertices of y belong to the same connected component (either S, T - S)
In this case, a cycle would be formed in this connected component (as there already exist one path between any two vertices and adding y would create another different path between them).

To remove this cycle, we must remove some edge.

It is best to remove the minimum-weighted edge (as in the end we aim for Maximum Spanning Tree, so removing the maximum-weighted edge, doesn't make much sense).
Let's call this removed edge z.
It can be seen that $|z| \leq |y|$ (y is added to this connected component, so the removed edge weight can be at most $|y|$).

Now, let's pick an edge from G, where one vertex is in S and another vertex in T - S, such that the weight of this edge is maximum among all such edges.
Let's call this edge w.
It can be seen that $|w| \geq |x|$ (we know that, x can connect these two disconnected components; after all, it was the one, which disconnected it)
Connecting "w" we would obtain a Maximum Spanning Tree T'.

Weight of T':

$$c(T') = c(T) - |x| + |y| - |z| + |w|$$

$$\Rightarrow c(T') = c(T) + (|y| - |z|) + (|w| - |x|)$$

It can be seen that, $c(T') > c(T)$ (as $|y| \geq |z|$ and $|w| \geq |x|$)
This contradicts the fact that T is a Maximum Spanning Tree for G.
Hence from the above cases, it is certain that a Maximum Spanning Tree without the widest path for U and V can't exist.

Hence, the widest path between U and V must lie in the Maximum Spanning Tree of G only.
Therefore, in general, the Maximum Spanning Tree of a Graph must contain the widest path for all pairs of vertices.