ESO207 Theoretical Assignment 2

Pragati Agrawal 220779

October 2023

Pragati Agrawal: 220779(Question 1)

Q1

(a)We have an array of n elements in which the first $(n-k)$ elements are arranged in a specific order and the last k elements, are all smaller than all elements in the first part. Our task is to find out if any element Val is present in the array or not. For this:

- We can first sort the last k elements in descending order. We can do this by first sorting the k elements in ascending order, and then reversing the sorted array. This complete step would take $O(k * log(k))$ time.
- Now, in the first $(n-k)$ elements, we know that some elements are arranged in ascending order, then remaining elements (upto $arr[n-1]$) in descending order. We can find out the position of the maximum element in $O(log(n))$ time. For this, we will perform a modified binary search: comparing the middle element to its predecessor and successor. Either the mid element is the maxima, or if not, then if they are in ascending order, then increment the left pointer, and if they are in descending order, then decrement the right pointer to find the local maxima. Let the position of the local maxima thus obtained be p.
- Now, we basically have two fragments of the array: first p elements in ascending order, next $(n-p)$ elements in descending order.

  Now to find Val in the array, we can perform binary search separately in these two portions of the array, as these are sorted. This would take $O(log(p) + log(n-p))$ time.

  The pseudo code for the algorithm is as: It consists of three functions, two to find the value using binary search, and one function "findVal" to find if Val is present or not. If it is present, we return 1, stating true. Otherwise, if not present, we return -1.

```
int binary_search_asc(arr, start, end, val){
    int l=start;
    int r=end;
    while(l<=r){
       int mid=(l+r)/2;
       if (val==arr[mid]){
           return mid;
```

```
        }
        else if (val>arr[mid]){
            l=mid+1;
        }
        else if (val<arr[mid]){
            r=mid-1;
        }
    }
    return -1;
}

int binary_search_dsc(arr, start, end, val){
    int l=start;
    int r=end;
    while(l<=r){
        int mid=(l+r)/2;
        if (val==arr[mid]){
            return mid;
        }
        else if (val>arr[mid]){
            r=mid-1;
        }
        else if (val<arr[mid]){
            l=mid+1;
        }
    }
    return -1;
}

int find_val(arr, n,k,val){
    sort_desc(arr+(n-k),arr+n);            //in descending order.
    int l=0, r=n-1;
    int max_idx;
    if (arr[0]>arr[1]){
        max-idx=0;
    }
    else if (arr[n-1]>arr[n-2]){
        max-idx=n-1;
    }
    while(l<r){
        int mid=(l+r)/2;
        if (arr[mid]>arr[mid-1] and arr[mid]>arr[mid+1]){
            max_idx=mid;
            break;
        }
        else if(arr[mid]>arr[mid-1] and arr[mid]<arr[mid+1]){
            l=mid+1;
        }
```

```
            else if(arr[mid]<arr[mid-1] and arr[mid]>arr[mid+1]){
                r=mid-1;
            }
        }
        int index1=binary_search_asc(arr,0, max_index, Val);
        int index2=binary_search_dsc(arr,max_index+1,n,Val);
        if (index1>0 or index2>0) return 1;
        else return -1;
    }
```

(*b*)

Correctness of Algorithm:

We make the given array sorted in a specific way for first $n-k$ elements. We sort last k elements in descending order. So by finding the position of maxima (p), we can say that the first p elements are in ascending order, and the remaining $(n-p)$ elements are made to be in descending order. Since all elements are distinct, we will either find the value Val in the ascending portion, or in the descending portion. We separately find Val in both these sub-arrays, and return 1 if found, and -1 if not found.

Time Complexity Analysis:

The overall time complexity of the algorithm is
$O((k * log(k)) + log(n) + q * (log(p) + log(n-p)))$ for q queries.

Now, since $p < n$ and $(n-p) < n$, therefore
$O(log(p) + log(n-p)) < O(log(n) + log(n)) = O(2 * log(n))$ i.e.
$O(log(p) + log(n-p)) = O(log(n))$.
Hence, the overall time complexity of the algorithm will be
$O(k * log(k) + q * log(n))$.

Pragati Agrawal: 220779(Question 2)

Q2

(a)
Part 1
Let's consider a perfect complete graph with n vertices. Now, we need to prove that

- between any pair of vertices, there is at most one edge
- for all $k \in \{0, 1, ..., n-1\}$, there exists a vertex v in the graph, such that $Outdegree(v) = k$.

According to the properties of a perfect complete graph, there is exactly one directed edge between every pair of distinct vertices. So, if the vertices are not distinct, there is no edge between them, otherwise exactly one edge. Hence, the first condition is satisfied.

We claim that *if there is a cycle in a perfect complete graph, it can not be transitive.*
Proof:
Consider a cycle in a perfect complete graph consisting of the vertices $\{a_1, a_2, a_3.....a_p\}$. Let there be directed edges from $a_1 \to a_2, a_2 \to a_3, a_3 \to a_4, ..., a_p \to a_1$ in the cycle. Now, since the graph is transitive, and there is an edge from $a_1 \to a_2, a_2 \to a_3$ there must be an edge from $a_1 \to a_3$. Similarly, an edge exits between $a_1 \to a_3$ and $a_3 \to a_4$ hence an edge must exist between $a_1 \to a_4$. Continuing this further, we must have an edge from $a_1 \to a_{p-1}$ and $a_{p-1} \to a_p$. This implies an edge from $a_1 \to a_p$ due to transitiveness. But we had assumed that there exists the edge $a_p \to a_1$. Since there can be atmost one edge between every pair of distinct vertices, both these edges simultaneously can not exist. Hence, our assumption is incorrect: a perfect complete graph cannot have a cycle.

Now, we claim that *a perfect complete graph will always have exactly one vertex of Outdegree* $= 0$.
Proof:
Consider a perfect complete graph that does not have such a vertex. This means there will be an outgoing edge from every vertex. Now, if we start from any vertex a_1 then go to its outgoing edge a_2, then to a_3 and so on, we must reach back a_1 at some point. This is because if we don't end up at a_1, we must have stopped at some end vertex, which has no further outgoing edge. But this is not possible because of our assumption. Hence there must be a cycle formation in the graph if $outdegree(v) > 0$ for all vertices $v \in V$. This means the graph can't be a perfect complete graph.

Using the above two claims, we can say that $\exists u \in V$,such that $Outdegree(u) = 0$. Now I remove this vertex u and all the edges incident on it.
Claim: Graph G' is a perfect complete graph.
This is because the edges from all pairs of distinct vertices $v \in V \backslash \{u\}$ are intact, even on removing u. Also, there was no outgoing edge from u, so removal of u doesn't affect transitiveness. Hence G' is a perfect complete graph, with vertices $|V| = n - 1$.
Again by the same arguments we can say that G' must have a vertex u', such that $Outdegree(u') = 0$ in G'. Since the $Outdegree(u) = 0$ in G, and $u' \in G'$, there must have been an edge from $u' \to u$ in G, and no other edge from u'. Hence the $Outdegree(u') = 1$ in G.
Now, remove the vertex u' in G' to obtain G'', with $|V| = n - 2$. Again by the same arguments, let there be the vertex u'' such that $Outdegree(u'') = 0$ in G''. This implies $Outdegree(u'') = 1$ in G', and $Outdegree(u'') = 2$ in G.
Continuing this, we will finally get a graph $(G')^{n-1}$ with $|V| = 1$, such that this vertex $(u')^{n-1}$ has

$Outdegree = 0$ in $(G')^{n-1}$. Going backwards, this vertex will have $Outdegree = n-1$ in G.
Therefore there exists n vertices in G each with distinct $Outdegree = \{0, 1, 2..., n-1\}$.
Hence proved that in any Perfect Complete Graph, there is atmost one edge between any two vertices, and for all $k \in \{0, 1, ..., n1\}$, there exists a vertex v in the graph G, such that $Outdegree(v) = k$.

Part 2: Proof of the Converse:
Given a directed graph G with atmost one edge between any pair of vertices, and and for all k {0, 1, ..., n 1}, there exists a vertex v in the graph, such that Outdegree(v) = k, then it is a Perfect Complete Graph.

We are given that all n vertices have n distinct $Outdegrees$, from 0 to $n-1$. Let these vertices be $\{v_1, v_2, v_3, ...v_n\}$ respectively. Now, to prove that exactly one directed edge exists between any two distinct vertices, we need to show that in total (including the incoming and outgoing edges) there must be $n-1$ edges from each vertex, i.e. $deg(v) = n-1$ for all $v \in V$.
Consider the vertex v_n of $Outdegree(v_n) = n-1$. This vertex has an outgoing edge to every other vertex in the graph. So trivially $deg(v_n) = n-1$. Now, consider the vertex v_{n-1} of $Outdegree(v_{n-1}) = n-2$. We know that there are $n-2$ outgoing edges from it to other vertices, and also one incoming edge to it from v_n. So $deg(v_{n-1}) = n-2+1 = n-1$.
Similarly, for v_{n-2}, $Outdegree(v_{n-2}) = n-3$. It has $(n-3)$ outgoing edges, and 2 incoming edges, from v_n and v_{n-1}. So $degree(v_{n-2}) = n-3+2 = n-1$. Similarly we can see that a vertex v_i, has $outdegree = n-i-1$ and $degree(v_i) = n-i-1+i = n-1$. The vertex v_i will have incoming edges from vertices $\{v_{i+1}, v_{i+2}, ...v_n\}$, and outgoing edges to $\{v_1, v_2, v_3....., v_{i-1}\}$.
Therefore all vertices in the given graph G have $degree = n-1$, and since there is atmost one edge between them, so there exists exactly one directed edge between them.
Now, for the property of **Transitiveness**, we know that we have edges from $v_k \to v_i$, such that $0 < k < i$, and from $v_i \to v_j$, such that $i < j$. So for any three vertices a, b and c, without loss of generality, let $a = v_x$, $b = v_y$ and $c = v_z$ and if $x < y < z$, then edges exist: $a \to b$, $b \to c$ and also $a \to c$, since $x < z$. Hence the graph is Transitive.
Moreover, if it wasn't Transitive, there could never be a vertex with $Outdegree = 0$, as then there must have been one outgoing edge from every vertex: $a \to b, b \to c, c \to a$.
Hence proved that the given directed graph G is a perfect complete graph.

(*b*) Let the adjacency matrix be constructed such that if there is an edge from $a_1 \to a_2$ then we mark the cell $mat[0][1] = 1$, rest as 0.
For the first condition of atmost one edge, we can check in $O(n^2)$ time, for each cell $mat[i][j]$ and $j \geq i$, we can check:

- If $i = j$, then $mat[i][j]$ must be 0.
- else if $mat[i][j] = 1$, then $mat[j][i]$ must be 0.
- else if $mat[i][j] = 0$, then $mat[j][i]$ must be 1. If any of these conditions turn out false, we can return $false$. If the loop terminates, it means all these were held true, hence first condition is satisfied.

Then for the second condition of Outdegrees, if we sum up the number of times a "1" appeared in any row, we will get the $Outdegree$ of the vertex of that row. We can store the values of these $Outdegrees$ in an array. This will take $O(n^2)$ time. Now, on sorting that array, we can check if all the values from $\{0, 1, 2....., n-1\}$ are present or not. Sorting would take $O(n * log(n))$ time while the last check would take $O(n)$ time. If all of these are present, then the graph is a perfect complete graph. The pseudo code is as:

```
int arr[n]={0};
for(i=0;i<n;i++){
    arr[i]=0;
    for (j=0;j<n;j++){
        if (mat[i][j]==1){
            arr[i]+=1;
        }
    }
}

sort(arr,arr+n);
flag=1;
for (i=0;i<n;i++){
    if (arr[i]!=i){
        flag=-1;
        break;
    }
}
if (flag==1){
    return true;
}
else {
    return False;
}
```

The overall **Time Complexity** of the algorithm is:
For the first check, $O(n^2)$.
For the second check, time $= O(n^2) + O(n * log(n)) + O(n) = O(n^2)$.
Hence the overall time complexity of the algorithm is $O(n^2)$.

Pragati Agrawal: 220779(Question 3)

Q3

(a) The way we are finding the scores of an array, we will obtain the minimum sum when the array is sorted: either in ascending order, or in descending order. So only those two permutations would give us minimum score. This is because for the sorted permutations, we would get the value of score to be $a_{high} - a_{low}$. But for any other permutation, we will get some term(s) in addition to this, which will be positive.
This can be proved using the **Triangle Inequality**. If x, y, z are the vertices of the sides of a triangle, then by triangle inequality, $|x-y| + |y-z| \geq |x-z|$, where mod represents the length of the side. Now, we can observe that the minimum score $|x-z|$ will only occur when the array is sorted. Extending this, we can say that if the individual terms $|a_{i+1} - a_i|$ are seen as the lengths of a polygon, the minimum score will be for a sorted array.
Consider any other permutation, where the smallest element a_m and the largest element a_M, occur somewhere in middle:
$A' = [a_1, a_2, a_3, ., a_m, .., a_M, ..., a_n]$
$Score(A') = (|a_2 - a_1| + |a_3 - a_2| + ... + |a_m - a_{m-1}|) + (|a_{m+1} - a_m| + ... + |a_M - a_{M-1}|) + (|a_{M+1} - a_M| + ... + |a_n - a_{n-1}|)$
$> |a_1 - a_m| + |a_{m+1} - a_m| + |a_M - a_{M-1}| + |a_M - a_n|$ (using triangle inequality for the individual brackets)
$= |a_1 - a_m| + |a_M - a_m| + |a_M - a_n|$ (since $a_m \leq a_i \leq a_M$ for all $i = 1$ to n)
$> a_M - a_m = a_{high} - a_{low}$.
(b) We need to sort the given array A, in such a way that the cost is minimum. For this, we need to compare the cost incurred while making it ascendingly sorted and also while making it descendingly sorted. The minimum of these two costs should be our answer. For the ascending order, we need to ensure that from the highest element to the lowest, if it's not on its correct position, we need to swap it from its current position, and bring it to its correct position. For e.g.
$A = \{27, 5, 60, 55, 89, 3\}$
$A = \{27, 5, 60, 55, 3, 89\}, swap = (89, 3)$
$A = \{27, 5, 3, 55, 60, 89\}, swap = (60, 3)$
$A = \{27, 5, 3, 55, 60, 89\}, noswap = X$
$A = \{3, 5, 27, 55, 60, 89\}, swap = (27, 3)$
$A = \{3, 5, 27, 55, 60, 89\}, noswap$
The total cost occurred is $= 89 + 60 + 27$. Similarly we can find the cost while sorting it in the descending order, keeping the highest value term on its correct position and moving forward. The algorithm goes as:

1. We will take the input in an array A and also simultaneously in a copy array S. We will also maintain an extra array A_idx, containing the indices of the elements of array A. Then we will create another array S_idx to store the original indices of the elements of array A. So, we will sort the array S and whenever a swap operation happens in the swap, we will also swap its corresponding indices in S_idx. So, after the sort completes, we will have 4 arrays:
   (a) A: the original input array.
   (b) A_idx: contains the indices of the elements of A from $\{1, 2, 3, ...n\}$.
   (c) S: elements of the array A, in the sorted order.

(d) S_idx: contains the original indices of each element of the array A, corresponding to the order in which these elements appear in S.

2. Now, if we have sorted S in the ascending order, we will first start from the end of S. Let the pointer in array A be i and in array S be j. Then we will check if the last elements of the arrays A and S are equal. If yes, we will decrement the pointers i, j and again check the same. Otherwise, we need to bring this highest element at the last position of array A. We know the highest element of A is the last element of S, which is currently at position S_idx in A, and there is some other element at the last position of A. We need to swap these two elements in the array A, find the positions of these elements in S and swap the elements at those positions in the array S_idx. This will give us the highest element on its correct position, and then we can decrement the pointers i, j.

3. We will need to apply binary search technique to find the index of the last element of A in the sorted array S.

The entire algorithm can be continued in the similar way described above to arrange, the array A into S.
The algorithm demonstrated below is on the input array A:
$A = \{27, 5, 60, 5, 89, 3\}$
We simultaneously create the arrays S and A_idx while taking the input:
$A = \{27, 5, 60, 5, 89, 3\}$
$S = \{27, 5, 60, 5, 89, 3\}$
$A\_idx = \{1, 2, 3, 4, 5, 6\}$
We create an empty array S_idx of the same size.
Then we sort S and also maintain the indices of A in S_idx during each swap, to finally obtain :
$A = \{27, 5, 60, 55, 89, 3\}$
$A\_idx = \{0, 1, 2, 3, 4, 5\}$
$S = \{3, 5, 27, 55, 60, 89\}$
$S\_idx = \{5, 1, 0, 3, 2, 4\}$ Now, we will keep the arrays A_idx and S untouched, and will perform swaps in arrays A and S_idx, as follows:
For the beginning, let $i = n - 1 = 5$ be pointer in A The loop will perform decrement of the pointers by 1 after every iteration, and will go upto $i, j \geq 0$. We will also need a temporary integer container *temp*.

The algorithm is:

1. If $A[i] == S[i]$, then *continue*.

2. Else if $A[i] \neq S[i]$ Let $temp = A[i]$ and swap $(A[i], A[S\_idx[i]])$.

3. Binary Search *temp* in S. Let its position be p. The swap $(S\_idx[p], S\_idx[i])$.

4. Perform $i - -$.

The pseudo code is:

```
for (i=n-1;i>=0;i--){
    if (A[i]!=S[i]){
        temp=A[i];
        swap(A[i],A[S_idx[i]]);
```

```
            p=bin_search(S,temp);      //returns the position of temp in S
            swap(S_idx[p], S_idx[i]);
        }
    }
```

The algorithm is demonstrated on the array A below:

1. $A = \{27, 5, 60, 55, 89, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{5, 1, 0, 3, 2, 4\}$ //initial arrays

2. $A = \{27, 5, 60, 55, 3, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{4, 1, 0, 3, 2, 5\}$ // iteration i=n-1=5, swap (89,3) in A, and swap(4,5) in S_idx.

3. $A = \{27, 5, 3, 55, 60, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{2, 1, 0, 3, 4, 5\}$ // iteration n-2=4, swap (60,3) in A, and swap(4,2) in S_idx.

4. $A = \{27, 5, 3, 55, 60, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{2, 1, 0, 3, 4, 5\}$ // iteration n-3=3, 55 at its correct position, no swap.

5. $A = \{3, 5, 27, 55, 60, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ // iteration n-4=2, swap (27,3) in A, and swap(0,2) in S_idx.

6. $A = \{3, 5, 27, 55, 60, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ // iteration n-5=1, 5 at its correct position, no swap.

7. $A = \{3, 5, 27, 55, 60, 89\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{3, 5, 27, 55, 60, 89\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ // iteration n-6=0, 3 at its correct position, no swap.

$Cost\_asc = 89 + 60 + 27$.
In a similar way we can also find the cost to arrange in the descending order. The arrays would be as follows, and we would be starting the iterations from $i = 0$, going upto $i = n - 1$, incrementing it by 1 each time.
$A = \{27, 5, 60, 55, 89, 3\}$
$A\_idx = \{0, 1, 2, 3, 4, 5\}$
$S = \{89, 60, 55, 27, 5, 3\}$
$S\_idx = \{4, 2, 3, 0, 1, 5\}$

the algorithm remains the same, slight modifications in the pseudo-code need to be made:

```
for (i=0;i<n;i++){
    if (A[i]!=S[i]){
        temp=A[i];
        swap(A[i],A[S_idx[i]]);
        p=bin_search(S,temp);      //returns the position of temp in S
        swap(S_idx[p], S_idx[i]);
    }
}
```

The algorithm is demonstrated on A as:

1. $A = \{27, 5, 60, 55, 89, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{4, 2, 3, 0, 1, 5\}$ // initial arrays

2. $A = \{89, 5, 60, 55, 27, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 2, 3, 4, 1, 5\}$ //iteration i=0, swap(89,27) in A, swap(0,4) in S_idx.

3. $A = \{89, 60, 5, 55, 27, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 1, 3, 4, 2, 5\}$ //iteration i=1, swap(60,5) in A, swap(2,1) in S_idx.

4. $A = \{89, 60, 55, 5, 27, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 1, 2, 4, 3, 5\}$ //iteration i=2, swap(55,5) in A, swap(2,3) in S_idx.

5. $A = \{89, 60, 55, 27, 5, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ //iteration i=3, swap(5,27) in A, swap(3,4) in S_idx.

6. $A = \{89, 60, 55, 27, 5, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ //iteration i=4, no swap.

7. $A = \{89, 60, 55, 27, 5, 3\}$
   $A\_idx = \{0, 1, 2, 3, 4, 5\}$
   $S = \{89, 60, 55, 27, 5, 3\}$
   $S\_idx = \{0, 1, 2, 3, 4, 5\}$ //iteration i=5, no swap.

$Count\_des = 89 + 60 + 55 + 27$
The minimum of these two costs would be our answer.

Time Complexity Analysis:

1. Taking simultaneous input in A, S, A_idx, S_idx will take $O(n)$ time.
2. Sorting S and maintaining S_idx would take $O(n * log(n))$ time.
3. Then we traverse the array A, and make suitable swaps in A and S_idx. The checks and swaps will take constant time for each iteration. Finding position p of $temp$ in sorted array S would take $O(log(n))$ time. Traversing the array will take $O(n)$ time. In the worst case, we may need to find position p for every element of A. So this step would take $O(n * log(n))$ time.
4. Repeating the same for descending order S' would again take the same time.

So the overall time complexity of the algorithm will be $2 * (2 * O(n * log(n)) + O(n)) = O(n * log(n))$.

Pragati Agrawal: 220779(Question 4)

Q4

We are given an undirected, unweighted, connected graph $G = (V, E)$ with $|V| = n, |E| = m$, and two vertices $s, t \in V$. We are provided the graph in adjacency list representation, $adjL$. We need to find the $dist(s, t)$ after any edge (u, v) is destroyed. We need to update these distances in the matrix M, for each cell $M[u, v]$.
(a) In order to find the distance between vertices s and t, we will first apply BFS algorithm on the given graph G. This will update the *dist* field of each node to its *distance* from s. Now, we will get the distance of s from t, when no edge is removed. Let this distance be d_0. Now, whenever we remove an edge, the distance between s and t will only change when that edge belongs to the shortest path and there's no other alternative shortest path from s to t. So, we will:

- Find the shortest route between s and t. For this, we will backtrack our path from t to s, such that we always move to a vertex having distance parameter one less than the distance parameter of the previous vertex. Thus, we will create an array of vertices, starting from t to s, which is a shortest path. Let this array be $arrL$.

Also, we can see that the maximum length of the shortest path from s to t can be $|V| - 1$. This will occur when the graph is linear and s and t are at its two opposite ends.
If the edge being removed is present in the array, we will need to run the BFS algorithm again to find the new shortest distance. Otherwise, it will remain d_0.
The pseudo code for finding the shortest path from s to t in the graph is: (I have used "D" to represent d_0 in the pseudo code.)

```
find_shortest_path(arrL, graph G){
    arrL[D]=t;
    int temp=D;
    while (temp>0){
        for every neighbour w of arrL[temp]{
            if (distance(w)==distance(arrL[temp]-1)-1){
                temp--;
                arrL[temp]=w;
                break;
            }
        }
    }
}
```

- So, we will initially fill all cells of the matrix M with the value d_0. Now, for every edge (u, v) in array $arrL$, the value might change. So, we will remove that edge from the graph, by going to the linked list at u and deleting the vertex v from it, then going to the linked list at v, and deleting the vertex u from it. Then applying the BFS on the remaining graph to find the new distance d'. We will enter this value in the matrix M at cell (u, v) and at cell (v, u) and then insert back the vertex v at the end of the linked list of vertex u, and vertex u at the end of the linked list of vertex v.

The pseudo code for the remaining part is:

```
BFS_traversal(s);
D=distance(t);
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        M[i][j]=D;
    }
}
int arrL[D+1];
find_shortest_path(arrL,G);
for(i=0;i<D-1;i++){
    u=arrL[i];
    v=arrL[i+1];
    Delete(u,v);         //to delete edge (u,v) in adjacency list of u
    Delete(v,u);        //to delete edge (v,u) in adjacency list of v
    BFS(s);
    New_dist=distance(t);
    M[u][v]=New_dist;
    M[v][u]=New_dist;
    Insert(u,v);        //to insert edge (u,v) in adjacency list of u
    Insert(v,u);        //to insert edge (u,v) in adjacency list of u
}
```

(*b*) **Time Complexity Analysis:**

- The first time when we apply BFS algorithm on the entire graph would take $O(|V| + |E|) = O(n + m)$ time.
- Then we will get the distance d_0 and updating the matrix M of size n^2 with this value, will take $O(n^2)$ time.
- Then we find a shortest path from s to t, by backtracking from t to s. This can contain atmost all vertices n. Hence the path can be of atmost $n-1$ length. In this we will check every edge from of the current vertex, and go to the one which has one less distance. So this would take $O(m)$ time. We store these vertices in order in an array.
- Now, for each edge $(u, v) \in arrL$, i.e. consecutive elements of $arrL$, in forward direction from s to t, we will remove the edge from graph and apply BFS. Since maximum number of edges in shortest path can be $n-1$, so for all edges, there can be maximum $n-1$ times BFS. We know that each BFS takes $O(m+n)$ time. So these checks would take maximum $O((n-1)*(m+n))$ time.
- Insertions and delete operations can take $O(n)$ time if adjacency list of u or v has (atmost) n vertices. The update operation in the matrix takes constant time.

We know that in a connected graph, $m = O(n^2)$. So all terms below $O(n * m)$ time complexity can be suitably merged into $O(n * m)$.
The overall time complexity of the algorithm will be
$O(n + m) + O(n^2) + O(m) + 2 * O(n) + O((n - 1) * (m + n))$
$= O(n * (m + n)) = O(|V| * (|V| + |E|))$.

(*c*) **Proof of Correctness:**
We know that the BFS algorithm correctly computes the distance of any vertex from any given vertex s. Now, we can clearly see that if the edge being removed doesn't lie on the shortest path, then this path will maintain the distance of t from s to be d_0.
Only when the edge lies on the shortest path, the distance might change. Now, in our algorithm, we only remove edges from the shortest path we found, and compute the distance using BFS from s to t every time each time. So again we will get the correct distance. Also, we reinsert the edge into the graph so that for any other edge (u', v'), we don't get incorrect result.
Now only part remaining to be established, is that the algorithm to find the shortest path from s to t indeed gives a shortest path from s to t.
In the algorithm, we move from t to its neighbour, which has distance one less. Now, since t is reachable from s, hence such a vertex must exist. Let this vertex be v_{d_0-1}. So the shortest path from s to t will be $s \to v_{d_0-1}, v_{d_0-1} \to t$.

Assertion: $arrL[i]$ *correctly stores a vertex* v *such that* $distance(v) = i$ *from* s*, and this path goes from* s *to* t*.*
Proof using induction on index i: Base case $i = d_0$, then $arrL[i] = t$. Assume it to be true for k. We will prove it for $k-1$, as we are creating the array from t to s. Now, at any iteration, we check all the neighbours of $arrL[k]$, and as we find a vertex whose distance is $k-1$, we add it to the array at position $arrL[k-1]$. As we assumed that $arrL[k]$ is true,
$distance(arrL[k]) = kdistance(arrL[k-1]) = distance(arrL[k]) - 1 = k - 1$. Now we decrement the pointer, so it now finds a vertex at a distance $k-2$. Since we have come from vertex $arrL[k]$ to vertex $arrL[k-1]$, therefore, that edge must exist. Continuing this, we will reach vertex s, which will be $arrL[0]$.
Hence the shortest path is computed correctly, and our proof is complete.

Pragati Agrawal: 220779(Question 5)

Q5

(a)We are given an undirected, unweighted and connected graph $G = (V, E)$, and a vertex $s \in V$, with $|V| = n, |E| = m$ and $n = 3k$ for some integer k. We define $dist(u, v)$ in the usual way as in lectures.
Now, at every layer, our graph has edges to every vertex in that layer from every vertex in the previous layer. So, we need to compute the maximum height of the graph, i.e., the maximum level distance of any node from s, on all the sides from s. Let it be d. If, this maximum level distance $d \leq k$ then we choose $t = s$ and we are done.
Otherwise, we find the maximum level distance remaining from s, after the first k nodes. This layer will be of size $(d - k)$. Now, if we find a node in between these layers, i.e. at the middle of layer $(k + 1)$ and layer d, (let this layer be p), then any node at this layer p, will satisfy the required conditions, and hence is a possible node t.
The algorithm is:

We know that the *Breadth First Search* traversal algorithm assigns distance to all nodes from any node s, in $O(|V| + |E|)$ time. Using BFS on the graph, we can find the distance of the farthest node. This is our d.
Required condition to be satisfied: $\boldsymbol{min(dist}(u, s)\boldsymbol{, dist}(u, t)\boldsymbol{)} \leq k$.
If $d \leq k$ then $t = s$ satisfies the above condition $\forall u \in V$.
If $d > k$, we find a node at a distance $\lfloor \frac{k+d+1}{2} \rfloor$ from s. This we can do using an algorithm similar to the BFS traversal from s, just that instead of assigning distance to each node, we will check all the neighbours of any vertex $w \in V$ if its distance is equal to $\lfloor \frac{k+d+1}{2} \rfloor$. This will take $O(|V| + |E|)$ time.
The pseudo code for the algorithm is:

```
BFS_traversal(s);
int d=0;
for (every v in V){
    if (distance(v)>d) d=distance(v);
}
if (d <=k) {
    t=s;
    return t;
}
else{
    temp=(d+k+1)/2;
    for(v in vertex){
        if (distance(v)==temp){
            t=v;
            break;
        }
    }
}
```

(b)**Proof of correctness:**

- If $d \leq k$, then $t = s$ and $min(dist(u, s), dist(u, t)) \leq dist(u, s) \leq k$. The proof is trivial.

- If $d > k$, the distance of the node t from s is $\lfloor \frac{k+d+1}{2} \rfloor$. Let this layer be p. Also, as we have edges from every vertex in a layer to every vertex in the next layer. Therefore, if $distance(v_1) = x$ and $distance(v_2) = y$, then the length of the shortest path from any vertex in layer x to any vertex in layer y will be $|(y - x)|$.
- Consider two sets of vertices $S_1 = \{V_0 \cup V_1 \cup V_2 \cup ....\cup V_k\}$ and $S_2 = \{V_{k+1} \cup V_{k+2} \cup V_{k+3} \cup ....\cup V_d\}$, where the subscripts denote the layer of vertices V_x at distance x from s.
- If $v \in S_1$, then we know $min(dist(u, s), dist(u, t)) \leq dist(u, s) \leq k$.
- If $v \in S_2$, we have $t \in V_p \in S_2$, since $p = \lfloor \frac{k+d+1}{2} \rfloor$. Also, the maximum value of d will be when the graph is entirely linear and in this case $d = n - 1$. Since the total number of nodes is $n = 3k$, $d \leq 3k - 1$. Now, the farthest vertices from t can be $v_1 \in V_{k+1}$ or $v_2 \in V_d$.
  1. For v_1, the $dist(v_1)$ from t will be $D = p - (k + 1)$. Since $p \leq (k + d + 1)/2$, therefore $D \leq \frac{k+1+d}{2} - (k + 1) = \frac{d-k-1}{2}$. Given that $d \leq 3k - 1$,
     $D \leq \frac{3k-1-k-1}{2} = \frac{2k-2}{2} = k - 1 < k$.
  2. For v_2, the $dist(v_1)$ from t will be $D = d - p$. Since $p > \frac{k+d+1}{2} - 1p > \frac{k+d-1}{2}$, therefore $D < d - \frac{k+d-1}{2} = \frac{d-k+1}{2}$. Given that $d \leq 3k - 1$,
     $D \leq \frac{3k-1-k+1}{2} = \frac{2k}{2} = k \leq k$.

Since $min(dist(u, s), dist(u, t)) \leq dist(u, t) \leq k$, for all vertices in S_1 and in S_2.
Hence Proved.