ESO207 Theoretical Assignment 2

Dhruv Gupta (220361)

October 2023

Dhruv Gupta: 220361

Q1

(a) Let us denote the subarray of A from index i to j (both included) as $A(i, j)$. We will sort the subarray $A(n-k, n-1)$ such that all the k elements are now arrange in descending order. For doing this we can use first sort the subarray in ascending order in $O(klog(k))$ time and then reverse it in $O(k)$ time.
Pseudo code for sorting in descending order $A(i, j)$:

```
Sort_descending($A[i, ..., j]$){
        sort($A[i, ..., j]$);
    //Following for loop swaps $A[i]$ with $A[j]$, $A[i+1]$ with $A[j-1]$, and so on, thereby reversing the array.
        for ($p = 0$ to $(j - i)/2$) {
            $temp \leftarrow A[i + p]$;
            $A[p + i] \leftarrow A[j - p]$;
            $A[j - p] \leftarrow temp$;
        }
```

Now, since all the elements from index $n-k$ to $n-1$ are smaller than all the elements from index 0 to $n-k-1$, after calling **Sort_descending(**$A[n-k, ..., n-1]$**)**, the final array will have
$A[0] < A[1] < ... < A[m] > A[m+1] > ... > A[n-1]$, for some $0 \leq m \leq n-k-1$.
We will now find the index m of the maximum element $A[m]$ in the array.

```
Find_m($A[0, ..., n-1]$){
    // Find_m will return the index of $A[m]$. //We will keep two indices $L$ and $R$ and our search will be limited to elements between $L$ and $R$.
        $L \leftarrow 0$;
        $R \leftarrow n-1$;
    //$A[m]$ will be the only element in the array which will be larger than both its neighbours. If $0 < i < m$, $A[i-1] < A[i] < A[i+1]$, if $m < i < n-1$, $A[i-1] > A[i] > A[i+1]$. We can use this property and apply binary search type algorithm to find m.
    //Compare $A[mid]$ with its neighbours if it is greater than both, $mid$ is the required index. If they form an increasing sequence we need to search only in the right part of the array, and if they form a decreasing sequence we need to search only in the left part. We update $mid$ accordingly and restrict the search area to half its original size.
        if($A[0] > A[1]$){ return 0; }
        else if($A[n-1] > A[n-2]$){ return $n-1$; }
        while ($True$) {
            $mid \leftarrow (L + R)/2$;
            if($A[mid-1] < A[mid]$ and $A[mid] > A[mid+1]$){ return $mid$; }
            else if($A[mid-1] < A[mid] < A[mid+1]$){ $L \leftarrow mid+1$; }
            else if($A[mid-1] > A[mid] > A[mid+1]$){ $R \leftarrow mid-1$; }
        }
    }
```

Now once we have the value of m, we know that $A[0, ..., m]$ and $A[m+1, ...n-1]$ are sorted subarrays. Hence to find any given element in $A[0, ..., n-1]$ we can apply our standard binary search algorithm twice, once in $A[0, ..., m]$ then in $A[m+1, ...n-1]$.

```
    if(Binary_Search ($A[0, ..., m]$) = 1 or Binary_Search($A[m+1, ..., n-1]$) = 1){$ans \leftarrow 1$}
        else{$ans \leftarrow 0$}
```

Now, $ans = 1$ means the element is found. $ans = 0$ means the element is not found.

(b) Time Complexity Analysis:
Time for Find_m
Let the time taken for array of length n be $T(n)$. After every iteration, we are halving the length of the array. All the comparisons and assignment operations take some constant time ($O(1)$ time). We can therefore write the time taken $T(n)$ recursively as
$T(n) = T(n/2) + c$ (Total time taken = some constant + time taken to search half the array), and
$T(1) = c'$ (constant).
$T(n) = c+T(n/2) = c+c+T(n/4) = c+c+c+T(n/8) = ... = c+c+c+\dots(log_2 n\ times)+T(1) = c * log_2 n + c'$
Therefore, $T(n) = c * log2n + c' = O(log(n))$.
Total time taken = Time taken for one call of **Sort_descending** function + Time taken for one call of **Find_m** function + Time taken for q times binary search in $A[0,..,m]$ + Time taken for q times binary search in $A[m+1,...,n-1]$
$= O(klog(k)) + O(log(n)) + O(qlog(m+1)) + O(qlog(n-m-1))$
$= O(klog(k)) + O(qlog(n)) + O(qlog(n)) + O(qlog(n))$
(Because $1 \leq q$, $m+1 \leq n$, $n-m-1 \leq n$.)
$= O(klog(k)) + O(3qlog(n))$
$= O(klog(k) + qlog(n))$

Explanation of correctness :
Since last k elements are all smaller than all the starting $n-k$ elements, and elements in A increase upto some index m, and decrease after than till index $n-k-1$. Therefore after sorting the last k elements in descending order, array A will increase upto index m. and then decrease.
If we know the value of m, we know that $A[0,...,m]$ and $A[m+1,...,n-1]$ are sorted arrays. By the correctness of the **Binary_search** algorithm, we can argue that we can search for any given element in these two sorted subarrays in $O(log(n))$ time. (*Note :* For Binary search we are assuming that it firsts check that the array is ascending or descending (by comparing the first 2 elements, and then works accordingly). Now if the element is not found in any of these, it is not present in the array A, and if it is found in (atleast) one of these it is present in A.
The only thing we need to prove now is the correctness of the **Find_m** function.
If $m = 0$ or $n-1$, we check these cases separately. We need to prove for $0 < m < n-1$. For this we just neeed to argue that the loop terminates, which is only possible when we find m and return from the function. *Assertion* $P(i)$ *:* After i^{th} iterarion on the **while** loop, $L \leq m \leq R$.
Clearly it is true for $i = 0$.
Assume $P(i)$. If $A[mid-1] < A[mid] < A[mid+1]$, we know the mid lies in the increasing subarray, i.e., $mid < m \leq R$. So we change the value of L to $mid + 1$. Hence in this case after $(i+1)^{th}$ iteration, $L \leq m \leq R$.
If $A[mid-1] > A[mid] > A[mid+1]$, we know the mid lies in the decreasing subarray, i.e., $L \leq m < mid$. So we change the value of R to $mid - 1$. Hence in this case also after $(i+1)^{th}$ iteration, $L \leq m \leq R$.
If $A[mid-1] < A[mid] > A[mid+1]$, then $mid = m$, we return mid and loop terminates.
Hence $P(i+1)$ is true. Therefore $P(i)$ is true for all i. And since we keep on restricting the region between L, R, and we are sure that such an m exists, the loop is sure to terminate.
This completes the proof of correctness of our algorithm.

Dhruv Gupta: 220361

Q2

(a) A directed graph with n vertices is called Perfect Complete Graph if:
Property 1 $-$ There is exactly one directed edge between every pair of distinct vertices.
Property 2 $-$ For any three vertices a, b, c, if (a, b) and (b, c) are directed edges, then (a, c) is present in the graph.

We need to prove the following statements :
Statement 1 $-$ In a Perfect Complete Graph, there exists atmost one directed edge between any pair of vertices, and for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph, such that $Outdegree(v) = k$.
Statement 2 $-$ In a directed graph with n vertices if there exists atmost one directed edge between any pair of vertices, and for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph, such that $Outdegree(v) = k$, then it is a Perfect Complete Graph.

Proof of Statement 1
Consider a Perfect Complete Graph $G = (V, E)$. Because of Property 1, the first condition is already satisfied for a Perfect Complete Graph (PCG). The Outdegree condition needs to be proved.

Claim 1: In a Perfect Complete graph a cycle cannot exist.
Proof: Suppose there is a cycle $< a_0, a_1, a_2, ..., a_{p-1}, a_p, a_0 >$ exists, where $a_i \in V \ \forall\, i \le p$, $(a_i, a_{i+1}) \in E \ \forall\, i \le p-1$ and $(a_p, a_0) \in E$.
Now, we claim that $(a_0, a_i) \in E \ \forall\, i \le p$. We will prove this by induction on p.
Base case, $p = 1$ is clearly true. Suppose it is true for $p = k - 1$. Now since $(a_0, a_{k-1}) \in E$ and $(a_{k-1}, a_k) \in E$, then by Property 2, $(a_0, a_k) \in E$. Hence we have proven inductively that $(a_0, a_i) \in E \ \forall\, i \le p$.
This means that $(a_0, a_p) \in E$. But then we have two directed edges between a_0 and a_p which violates Property 1. Hence our assumption that there is a cycle cannot hold. Therefore, in a PCG, a cycle cannot exist.

Claim 2: In a PCG with n vertices there always exists a vertex v such that $Outdegree(v) = n-1$.
Proof: Note that every vertex is connected to every other vertex by exactly 1 edge, which implies that total number of edges coming to or going out of the vertex is $n - 1$.
Suppose there does not exist a vertex with $Outdegree(v) = n-1$. Then since total number of edges is $n - 1$, every vertex must have atleast one incoming edge. Now we will prove that under this assumption a cycle must exist in the graph.
Start from any vertex, let us call it v_0. v_0 must have an incoming edge from some other vertex, let us call that vertex v_1. We repeat this process for v_1, and name the corresponding vertex v_2, and so on. We keep repeating this until we either reach at a vertex which was already visited once, or until we have visited all n vertices atleast once. For the latter case, we look for the vertex from which v_{n-1} (the last visited vertex) has an incoming edge, now we are sure to get a vertex v_p, which was visited earlier. This case is therefore reduced to the former case.
Consider this vertex v_p which was visited twice, and make a path from all the vertices we visited in between. This path would be a cylce. But by Claim 1, a cycle cannot exist in a PCG, therefore our assumption is wrong. Hence, in every PCG with n vertices there always exists a vertex v such that its $Outdegree$ is $n - 1$.

Note: If we remove a vertex and all the edges connected to that vertex from a PCG, the resulting graph is still a PCG.
Let the graph be $G = (V, E)$, and the removed vertex be v. Call the new graph $G' = (V', E')$, where $V' = V \backslash \{v\}$, and
$E' = E \backslash \{(a, b) | a = v \ or \ b = v, (a, b) \in E\}$. Clearly, we have not touched any other edges corresponding to the vertices in V', and thus E' contains all the edges connecting the vertices in V', i.e., Property 1 is satisfied. Now, $E' \subseteq E \ \cap (V' \times V')$. If we take any two elements from E', (a, b) and (b, c), it is guaranteed that $a, b, c \in V' \subseteq V$, and E satisfies Property 2, hence $(a, c) \in E'$. Hence Property 2 is also satisfied for E'. Therefore G' is also a PCG.

We know by Claim 2 that we have a vertex with Outdegree $n - 1$. Let us call this vertex v_{n-1}. Let us remove this vertex and edges connected to it from the graph G to get the graph G_1. Now G_1

has $n-1$ vertices, hence by Claim 2, we have a vertex in G_1 with Outdegree $n-2$, Let us call this v_{n-2}. Since in G, v_{n-1} had outgoing edges to all other vertices, v_{n-2} must have 1 incoming edge from v_{n-1} in G. Therefore, when we removed this edge while forming G_1, Outdegree of v_{n-2} must have remained same, hence, we have a vertex v_{n-2} in G with Outdegree $n-2$. Now we repeat this process, remove v_{n-2} to for G_2. Now, by similar logic we get a vertex v_{n-3}, with Outdegree $n-3$ in G_3. Now, v_{n-2} and v_{n-1} both had outgoing edges to v_{n-3} (We can say this by seeing v_{n-2} , v_{n-3} in G_1 and v_{n-1} , v_{n-3} in G). Hence v_{n-3} must have 2 incoming edges from v_{n-1} and v_{n-2} in G. Therefore, when we removed these edges while forming G_2 from G, Outdegree of v_{n-3} must have remained same, hence, we have a vertex v_{n-3} in G with Outdegree $n-3$. We can keep repeating this process, until we get v_0. In general for all $0 \leq i \leq n-2$, we can see that v_i has an incoming edge from v_j (We can say this by seeing v_j , v_i in G_{n-j-1} and v_{n-1} , v_i in G), where $i+1 \leq j \leq n-2$. And thus, when we removed these edges while forming G_{n-i} from G, Outdegree of v_i must have remained same, hence, we have a vertex v_i in G with Outdegree i.

Hence, in every Perfect Complete Graph with n vertices there exists atmost one directed edge between any pair of vertices, and for all $k \in \{0, 1, ..., n1\}$, there exist a vertex v in the graph, such that $Outdegree(v) = k$.

Proof of Statement 2

Consider a graph G with n vertices which satisfies the conditions of Statement 2. We are given that in a directed complete graph atmost one directed edge exist between any two points. For Property 1, we need to prove that exactly 1 edge exists between any two points, i.e., degree of every vertex must be $n-1$.
Proof for this is similar to that of the last part in Proof of Statement 1. Let v_i be the vertex with Outdegree i. Now, clearly v_{n-1} has $n-1$ outgoing edges and no incoming edge, so $deg(v_{n-1}) = n-1$, and v_{n-1} has an outgoing edge to every other vertex. Now, v_{n-2} has an incoming edge from v_{n-1} and $n-2$ outgoing edges, so it has an outgoing edge to every other vertex except v_{n-1}. Also, it is clear that $deg(v_{n-2}) = (n-2)+1 = n-1$. Similarly, v_{n-3} has 2 incoming edges one each from v_{n-1}, v_{n-2} and $n-3$ outgoing edges, so it has an outgoing edge to every other vertex except v_{n-1}, v_{n-2}. Also, it is clear that $deg(v_{n-2}) = (n-3)+2 = n-1$. We can generalize this to say that for all $i \in \{0, 1, ..., n-1\}$, v_i has $n-i-1$ incoming edges one each from $v_{i+1}, v_{i+2}, ..., v_{n-1}$ and i outgoing edges, one edge going to $v_0, v_1, ..., v_{i-1}$ each. In total $deg(v_i) = i + (n-i-1) = n-1$. Hence, we have proven that Property 1 is satisfied for G.

For Property 2, let us consider any 3 arbitrary distinct vertices v_a, v_b, v_c in G. Without loss of generality let us assume, $a > b > c$. We proven above in proof for Property 1, that any vertex in G, v_i has an incoming edge each from $v_{i+1}, v_{i+2}, ..., v_{n-1}$ and an outgoing edge each to $v_0, v_1, ..., v_{i-1}$. In other words, (v_j, v_1) exists if only if $i < j$. Hence, for v_a, v_b, v_c
$a > b \Rightarrow (v_a, v_b)$ exist, $b > c \Rightarrow (v_b, v_c)$ exist and, $a > b$ and $b > c \Rightarrow c > a \Rightarrow (v_c, v_a)$ exist. And since this is true for any 3 arbitrary distinct vertices in G, Property 2 holds for G.
Thus we have proven that G must be a Perfect Complete Graph.

(b) Pseudo code of Algorithm with comments :

```
// We consider vertices to be labelled as $0, 1, 2, ...n-1$. We assume in adjacency matrix $A$, $A[i][j] = 1$ means edge $(i, j)$ exists.
// We will traverse through each row of the adjacency matrix, and count the number of 1s to calculate the outdegree of each vertex.
//We will maintain an array $Outdeg[0, ..., n-1]$, such that $Outdeg[i]$ stores the Outdegree of vertex $i$.
for ($i = 0$ to $n-1$){
    $count \leftarrow 0$;
    for ($j = 0$ to $n-1$){
        if ($A[i][j] = 1$) $count++$;
    }
    $Outdeg[i] \leftarrow count$;
}
// Now we will sort the Outdeg array. If the given graph is a Perfect Complete Graph, Outdeg must contain all the values from 0 to $n-1$, and after sorting, $Outdeg[i] = i$ must be true for all $i$.
sort($Outdeg[0, ..., n-1]$);
$flag \leftarrow 1$;
for ($i = 0$ to $n-1$){
    if ($Outdeg[i] \neq i$){ $flag \leftarrow 0$; break; }
// At this point, if $flag = 0$ then it is not a Perfect Complete Graph, but if $flag = 1$ then we
also have to check that there is atmost one edge between every pair of vertices, i.e., if $(i, j)$ exists
then $(j, i)$ should not exist.
// For checking this, we traverse all the values in the upper triangle, i.e., $j \geq i$. and check if
$A[i][j]$ and $A[j][i]$ both exist, if yes, then we have 2 edges between $i$ and $j$. So it cannot be a
Perfect Complete Graph.
if ($flag = 1$){
    for ($i = 0$ to $n - 1$){
        if ($flag = 0$) break;
        for ($j = i$ to $n - 1$){
            if ($A[i][j] = 1$ and $A[j][i] = 1$) {$flag \leftarrow 0$; break; }
        }
    }
}
```

Time Complexity Analysis :
We have two sets of nested for loops (one in the starting and one at the end) each one with two loops one nested in the other. This will clearly take $O(n^2)$ time. Sorting will take $O(nlog(n))$ time. And the last for loop will take $O(n)$ time.
Hence total time taken = $O(n^2) + O(nlog(n)) + O(n) = O(n^2)$.

Dhruv Gupta: 220361

Q3

(a) Let G be the set of all *"good permutations"*, then G is given by,

$$G = \{\Pi \mid \Pi \; is \; a \; permutation \; of \; \{1, 2, ..., n\}, A(\Pi) \; is \; a \; sorted \; array\}$$

Clearly, there are only two such *"good permutations"* – one which arranges the array in increasing order, and other which arranges the array in descending order. Let us call them Π_1, Π_2 respectively. Therefore, $G = \{\Pi_1, \Pi_2\}$.
We can prove that these are the only two *"good permutations"*.
Let the given array be $A = [a_1, a_2, a_3, ..., a_n]$, and sorted array (ascending) be $B = [b_1, b_2, b_3, ..., b_n]$. Score of B, $S(B) = (b_2 - b_1) + (b_3 - b_2) + ... + (b_n - b_{n-1}) = b_n - b_1$. Similarly for a descending array $B' = [b_n, b_{n-1}, b_{n-2}, ..., b_1]$, Score of B', $S(B') = (b_n - b_{n-1}) + (b_{n-1} - b_{n-2}) + ... + (b_2 - b_1) = b_n - b_1$.
We can prove that for any other permutation of elements of A, Score will be higher using triangle inequality. Let $A' = A(\Pi) = [a'_1, a'_2, a'_3, ..., a'_n]$, $\Pi \neq \Pi_1, \Pi_2$. Let a'_M, a'_m be the maximum and minimum elements in A' respectively. WLOG assume, $m < M$.
$S(A') = (|a'_1 - a'_2| + |a'_2 - a'_3| + ... + |a'_{m-1} - a'_m|) + (|a'_m - a'_{m+1}| + ... + |a'_{M-1} - a'_M|) + (|a'_M - a'_{M+1}| + ... + |a'_{n-1} - a'_n|)$
$> |a'_1 - a'_m| + |a'_M - a'_m| + |a'_M - a'_n|$ (Using Triangle inequality separately for all three 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)
$> a'_M - a'_m = b_n - b_1$. Clearly, B and B' are permutations of A (as well as A'), $b_n = a'_M$, and $b_1 = a'_m$.
Hence, $S(A(\Pi)) = S(A') > S(B) = S(B') = S(A(\Pi_1) = S(A(\Pi_2)$ for all $\Pi \neq \Pi_1, \Pi_2$. Hence, Π_1, Π_2 are the only *"good permutations"*.

(b) We will maintain an array of indices of A. Whenever we swap two elements of A, we will also swap the corresponding indices in the array of indices. We can modify our quick sort function for this purpose accordingly. Let $index_A$ be the array of indices of A. We will initialise it as : $index_A = [1, 2, ..., n]$.

```
//For sorting in ascending order. **QuickSort_modified** $(A, l, r, index_A)$\{
    **if** $(l < r)$ \{
        $i \leftarrow$ **Partition_modified**$(A, l, r, index_A)$;
        **QuickSort_modified** $(A, l, r, index_A)$;
        **QuickSort_modified** $(A, l, r, index_A)$;
    \}
\}

**Partition_modified**$(A, l, r, index_A)$\{
    $pivot \leftarrow A[l]$;
    $k \leftarrow r$;
    **for** $(i = r;\ i > l;\ i - -)$\{
        **if** $(A[i] > pivot)$\{ //To sort in descending order replace '>' by '<'
            **swap** $(A[i], A[k])$;
            **swap** $(index_A[i], index_A[k])$;
            $k - -$;
        \}
    \}
    **swap** $(A, k, l)$;
    **swap** $(index_A, k, l)$;
\}
```

We make a copy of the original array A and sort the copy. Let S_A be the sorted array obtained. For now let us assume it to be sorted in ascending order. Also we will get the $index_A$ array. Note that $index_A[i]$ gives the index of $S[i]$ in the original array A or in other words $S_A[i] = A[index_A[i]]$. So we can represent $index_A$ by a permutation Π such that $S_A(\Pi) = A$.
We will swap elements in A to convert it to S_A, and keep updating the total cost after each swap. Now consider the maximum element of A which is not at its right position (by right position we mean its position in S_A). To send it to its right position we will need atleast one swap. If we take

more than one swaps, it will increase the total cost, since cost$(i, j) = \max(A[i], A[j])$. To keep the total cost minimum, we will ensure that we do this in only one step.

We will first swap the the largest element in A to place it at its right position, then we do the same for the 2nd largest element, then 3rd largest and so on. For this we keep a counter i which is initially at n and we keep decrementing it after we place the element at its correct position as per the sorted array. This i actually stores the index of the element which we want to swap in the array S_A (we will refer it as the correct index of this element). If that element is already at its correct location, we decrement i and start again. Right now, we do not know the position of this element in A. Now, $index_A[i]$ gives us the index of the element to be swapped in A. Now we swap the $A[index_A[i]]$ (the element which we wanted to swap) with $y = A[i]$ (element at the correct position of $A[index_A[i]]$). But now to maintain the property that $index_A[i]$ gives the index of $S_A[i]$ in A, we need to update the $index_A$ array too. For this we search y in S_A. Let it be at some index x. Since y was earlier at index i in A, $index_A[x] = i$. Therefore, we swap $index_A[i]$ and $index_A[x]$.
Now we decrease i by 1 and repeat the above process, until i reaches 0.

We need to repeat the same process but this time we will take the array S'_A, sorted array in descending order. We will similarly get the array $index'_A$. And we start the counter i from 1 and keep incrementing i until we reach n.
Since both ascending order and descending order are *"good permutations"*, any of them will work, so we calculate cost for both of them, and take the minimum of the two.
Pseudo code with comments :

```
// Create a copy $S$ of $A$.
for ($i = 0$ to $n$) $S[i] \leftarrow A[i]$;
// Initialise $index$ array.
for ($i = 0$ to $n$) $index[i] \leftarrow i$;
//Sort in Ascending order.
QuickSort_modified ($S, 1, n$); $cost\_1 \leftarrow 0$;
$i \leftarrow n$;
for ($i = n$; $i > 0$; $i - -$){
    if ($A[i] = A[index[i]]$) continue;    $y \leftarrow A[i]$;
// Here we are assuming that swap $(A, i, j)$ swaps $A[i], A[j]$ in array $A$.
    swap ($A, index[i], i$);
    $cost\_1 \leftarrow cost\_1 + A[i]$; // We have ensured that $A[i] > A[index[i]]$, in our algorithm.
// Here we are assuming that Binary_search($M, p$) searches for $p$ in the array $M$, and returns
the index at which $p$ is found. Returns -1 if not found.
    $x \leftarrow$ Binary_search($index, y$)
    swap ($index, i, x$);
}
// Repeat the same for descending case.
// Create a copy $S$ of $A$.
for ($i = 0$ to $n$) $S[i] \leftarrow A[i]$;
// Initialise $index$ array.
for ($i = 0$ to $n$) $index[i] \leftarrow i$;
//Sort in Descending order.
QuickSort_modified ($S, 1, n$);
// Use the changed Partition function (after changing the sign in comparison step)
$cost\_2 \leftarrow 0$;
$i \leftarrow 1$;
for ($i = 0$; $i < n$; $i + +$){
    if ($A[i] = A[index[i]]$) continue;    $y \leftarrow A[i]$;
// Here we are assuming that swap $(A, i, j)$ swaps $A[i], A[j]$ in array $A$.
    swap ($A, index[i], i$);
    $cost\_2 \leftarrow cost\_2 + A[i]$; // We have ensured that $A[i] > A[index[i]]$, in our algorithm.
// Here we are assuming that Binary_search($M, p$) searches for $p$ in the array $M$, and returns
the index at which $p$ is found. Returns -1 if not found.
    $x \leftarrow$ Binary_search($index, y$)
    swap ($index, i, x$);
}
$cost \leftarrow$ min ($cost\_1, cost\_2$). // $cost$ stores our final answer.
```

Note : Time complexity analysis is given after the example.

Let us see an example for clarity –
$S_A = [11, 13, 39, 42, 61, 99]$

$A = [42, 13, 61, 11, 99, 39]$
$index_A = [4, 2, 6, 1, 3, 5]$
$i = 6$
After 1st swap,
$A = [42, 13, 61, 11, 39, 99]$
$index_A = [4, 2, 5, 1, 3, 6]$
total cost = 99
$i = 5$
After 2nd swap,
$A = [42, 13, 39, 11, 61, 99]$
$index_A = [4, 2, 3, 1, 5, 6]$
total cost = 99 + 61 = 160
$i = 4$ After 3rd swap,
$A = [11, 13, 39, 42, 61, 99]$
$index_A = [1, 2, 3, 4, 5, 6]$
total cost = 99 + 61 + 42 = 202
$i = 3, 2, 1$: Nothing happens, array is sorted already.

Above example with descending order :
$S'_A = [99, 61, 42, 39, 13, 11]$

$A = [42, 13, 61, 11, 99, 39]$
$index'_A = [5, 3, 1, 6, 2, 4]$
$i = 1$
After 1st swap,
$A = [99, 13, 61, 11, 42, 39]$
$index'_A = [1, 3, 5, 6, 2, 4]$
total cost = 99
$i = 2$
After 2nd swap,
$A = [99, 61, 13, 11, 42, 39]$
$index'_A = [1, 2, 5, 6, 3, 4]$
total cost = 99 + 61 = 160
$i = 3$
After 3rd swap,
$A = [99, 61, 42, 11, 13, 39]$
$index'_A = [1, 2, 3, 6, 5, 4]$
total cost = 99 + 61 + 42 = 202
$i = 4$
After 4th swap,
$A = [99, 61, 42, 39, 13, 11]$
$index'_A = [1, 2, 3, 4, 5, 6]$
total cost = 99 + 61 + 42 +39 = 241
$i = 5, 6$: Nothing happens, array is already sorted.

Time Complexity Analysis :
We have used 4 **for** loops for initialisation which takes $O(n)$ time.
We use **QuickSort_modified** 2 times which takes $O(n * log(n))$ time.
Inside the two **for** loops where swaps occur, in each iteration we use **Binary_search** once, this takes $O(log(n))$ time. All other operations and the **swap** function calls take $O(1)$ time. So, each iteration takes $O(log(n))$ time. Each **for** loop runs for n iterations, thus total time taken is $O(n * log(n))$. All other operations including the **min** function call takes $O(1)$ time. Therefore, overall time complexity $= O(n) + O(n * log(n)) + O(n * log(n)) + O(1) = O(n * log(n))$.

Dhruv Gupta: 220361

Q4

(a) Pseudo code with comments :

```
// Apply BFS from vertex $s$. Let $Distance(t) = d$.
**BFS**$(s)$;
$d \leftarrow Distance(t)$;
//We will first find all the edges in the shortest path from $s$ to $t$. If there are multiple shortest paths, we take any one of them. To keep track of the shortest path we will maintain an array $Shortest\_path[0, 1, ..., d]$ of vertices.

$Shortest\_path[d] \leftarrow t$
$i \leftarrow d$;
while$(i > 0)${
    For each neighbour $w$ of $Shortest\_path[i]$
    {
        if$(Distance(w) = Distance(Shortest\_path[i]) - 1$ {
            $Shortest\_path[i-1] \leftarrow w$;
            $i \leftarrow i - 1$;
            break;
        }
    }
}
//Note : Working of the above block of code is explained in part (c).
//Now we will construct the matrix $M_{n\times n}$. For any $u, v$ if $(u, v)$ is not present in the $Shortest\_path$ array, then $M[u, v] = d$, otherwise we again apply BFS from $s$ to find $Distance(t)$
// Assume vertices to be labelled $1, 2, ..., n$.
For $(i = 1$ to $n)${
    For$(j = 1$ to $n)$ {
        $M[i, j] \leftarrow d$;
    }
}
// For each edge in Shortest path we will have to update $M$.
// Assuming adjacency list representation of graph. Let $adj[i]$ is the adjacency list of vertex $i$.
For $(i = 0$ to $d - 1)${
// Remove Edge $(Shortest\_path[i], Shortest\_path[i + 1])$ from $G$. Delete $i + 1$ from $adj[i]$ and Delete $i$ from $adj[i + 1]$.
    Delete$(adj[Shortest\_path[i]], Shortest\_path[i + 1])$;
    Delete$(adj[Shortest\_path[i + 1]], Shortest\_path[i])$;
// Recalculate the distances using BFS.
    BFS$(s)$;
    $M[i, i + 1] \leftarrow Distance(t)$;
    $M[i + 1, i] \leftarrow Distance(t)$;
// Insert the edges back.
    Insert$(adj[Shortest\_path[i]], Shortest\_path[i + 1])$;
    Insert$(adj[Shortest\_path[i + 1]], Shortest\_path[i])$;
}
```

(b) Time complexity Analysis :

The first **BFS** takes $O(|V| + |E|)$ time.

The **while** loop which computes the $Shortest\_path$ array continues for d iterations (d = number of edges in the shortest path = number of vertices in the shortest path -1). In each iteration we have a **For** loop, hich takes $O(deg(v))$ time. Therefore total time taken to compute the $Shortest\_path$ array $= O(\sum_{v\in Shortest\ path} deg(v)) \leq O(\sum_{v\in V} deg(v)) = O(2 * |E|) = O(|E|)$.

Creating the matrix M clearly takes $O(|V|^2)$ time because of the two nested **For** loops each having $n = |V|$ iterations.

The last **For** loop, which correctly modifies M runs for d iterations. For **Delete** and **Insert** operations, in worst case we have to search for the vertex in the adjacency list, so let us assume it takes $O(deg(i))$ time for $adj[i]$. Again, $\sum_{i=0}^{d-1} O(deg(i)) \leq \sum_{v\in V} O(deg(v)) = O(2 * |E|) = O(|E|)$.

Also in every iteration we apply **BFS** on the reduced graph with $|V|-1$ vertices and $|E|-1$ edges. This will take $O(|V|-1+|E|-1) = O(|V|+|E|)$ time for each iteration. Hence total time time taken by the this **For** loop is $d*O(|V|+|E|)+4*O(E) = O(|V|\cdot(|V|+|E|))+O(|E|)$ (since clearly, $d \leq |V|$). Hence total time taken $= O(|V|+|E|)+O(|E|)+O(|V|^2)+O(|V|\cdot(|V|+|E|))+O(|E|)$
$= O(|V|+|E|+|E|+|V|^2+|V|\cdot(|V|+|E|)$
$= O(2*|V|^2+|V|*|E|+|V|+2*|E|)$
$\leq O(2*|V|^2+|V|*|E|+|V|^2+2*|V|*|E|$
$= O(3*|V|\cdot(|V|+|E|)$
$= O(|V|\cdot(|V|+|E|)$.

(c) Proof of Correctness (along with detailed explaination of algorithm) :
We will use the correctness of the BFS algorithm that BFS(s) correctly calculates the shortest distance from s to every other vertex in graph (since the graph is connected all vertices are reachable). For now let us assume that we have all the edges of one of the Shortest path from s to t. Let us call this path P and its length be d.
Now, if we remove any edge from the graph which does not lie on P, we can always go from s to t via P, hence the distance of t remains same, i.e., d. So, when we initialise the Matrix M with all values $= d$, we have correctly initialised it for all indices i, j such that the edge (i, j) does not lie on P.
For those edges which lie on P, we remove one edge at a time, apply BFS(s) again to calculate the distance of t in the reduced graph, then we update that entry in M and then we restore the original graph by inserting back that edge. We do this for all the edges which lie on P. Since BFS(s) correctly computes the distances, we have correctly updated the entries in matrix M.
Our proof of correctness is complete, if we can prove the correctness of our algorithm to find the Shortest path from s to t, i.e., the part of code where we create the *Shortest_path* array.
If $P < s = v_0, v_1, v_2, ...v_{d-1}, v_d = t >$ is the shortest path from s to t, then we know that $Distance(v_i) = i$. We use this property to create *Shortest_path*.
Start from $t = v_d$, for each neighbour of v_d, check if its distance is $d-1$ or not, since v_d is reachable from s, there must exist atleast one such vertex. We call that vertex v_{d-1}. Now clearly, shortest path from s to v_d = shortest path from s to $v_{d-1} + (v_{d-1}, v_d)$. Now we can repeat the same process for v_{d-1} then for v_{d-2} and so on, until we reach s.

Assertion $A(i)$: $Shortest\_path[i]$ correctly stores a vertex w with $Distance(w) = i$, and the edge ($Shortest\_path[i]$, $Shortest\_path[i+1]$ exists if $i < d$ (otherwise it would not be a path).

Proof : Use induction on i. Base case, $i = d$ is true since BFS correctly computes $Distance(t)$ correctly. Now, assume $A[i]$. We need to prove $A[i-1]$. (Note that we are going backwards). We check for all the neighbours of the vertex $Shortest\_path[i]$ one by one. As soon as we find a neighbour w with $Distance(w) = Distance(Shortest\_path[i]) - 1$, we update $Shortest\_path[i-1]$ to store w and decrease i by 1, and continue next iteration of the **while** loop. Now since $A[i]$ is true, $Distance(Shortest\_path[i]) = i \Rightarrow Distance(w) = i-1 \Rightarrow Distance(Shortest\_path[i-1]) = i-1$. Also, since w was a neighbour of $Shortest\_path[i]$, the edge ($Shortest\_path[i-1]$, $Shortest\_path[i]$) exists. Hence, $A[i-1]$ is also true.
Thus our Proof of Correctness for above algorithm is complete.

Dhruv Gupta: 220361

Q5

(a) Let us first apply BFS on G from s. This will give us $dist(u, s)$, for all $u \in V$. Let D be the maximum possible distance from s, i.e., $D = \max(\{dist(u, s) \mid u \in V\})$.
Suppose, if $dist(u, s) \le k$ for all $u \in V$, then we can choose t to be s. Otherwise, if $dist(u, s) > k$ for some $u \in V$, we will choose t as follows –
Let $p = \lfloor \frac{k+D+1}{2} \rfloor$. Consider the set V_p. Choose any vertex from V_p, this will be our vertex t.
Pseudo Code for algorithm :

```
**BFS**($s$);
$max \leftarrow 0$; // This for loop is to calculate the maximum distance from $s$. **For** every $v$ in $V$ {
    **if** ($Distance(v) > max$){ $max \leftarrow Distance(v)$; }
}
$D \leftarrow max$;
**if** ($D \le k$) { $t \leftarrow s$;}
**else** {
    $p \leftarrow (k + D + 1)/2$; // Assuming integer division.
    **For** every $v$ in $V$ {
        **if** ($Distance(v) = p$){ $t \leftarrow v$; **break**; }
    }
};
```

After the execution of this code, t will be set to our required vertex.

(b) Proof of correctness :
If $dist(u, s) \le k$ for all $u \in V$, i.e., $D \le k$ then $\min(dist(u, s), dist(u, t)) \le dist(u, s) \le k$, for all $u \in V$, for any $t \in V$. So we can simply choose t to be s.
Now consider the case when $k < D$ or $D \ge k + 1$.
Let $S = \{u \in V \mid dist(u, s) \le k\}$ and $S' = V \backslash S = \{u \in V \mid dist(u, s) > k\}$. Take any $t \in V_p$, where $p = \lfloor \frac{k+D+1}{2} \rfloor$.
Claim : $dist(u, t) \le k$ for all $u \in S'$.
Proof : Since we are given that $\forall\ i \ge 0\ :\ u \in V_i, v \in V_{i+1} \Rightarrow (u, v) \in E$, for any $a < b$ if $x \in V_a$ and $y \in V_b$, there exists a path $P =< x, v_1, v_2, ..., v_{b-a-1}, y >$, where $v_i \in V_{a+i}$. Note that this path is the shortest possible path, since in a BFS graph, vertices at distance i have edges only with vertices at distance $i-1, i, i+1$. So we can only move atmost 1 unit in distance when going from one vertex to another. And since in our path we move towards y from x at each step, it must be the shortest path. Therefore, $dist(x, y) =$ length of the path $P = b - a$.
Note that $S = V_0 \cup V_1 \cup \cdots \cup V_k$, and $S' = V_{k+1} \cup V_{k+2} \cup \cdots \cup V_D$.
Now, since clearly, vertices belonging to V_{k+1} and V_D are farthest from t (because $t \in V_p$ and $k+1 \le p \le D$) . So for proving our claim by above argument, it suffices to prove that $p-(k+1) \le k$ and $D - p \le k$ because if the vertices in furthermost Layers are reachable within k steps from t, then all the vertices in the intermediate layers are also reachable within k steps from t . For this we will have to consider two cases.
Note: Note that D is the length of the path from a vertex s to some vertex in $V_D \subseteq V$. Thus, $D \le |V| \Rightarrow D \le 3k$.
<u>CASE 1:</u> $k + D + 1$ is even.
$p = (k + D + 1)/2$
$p - (k + 1) = D - p = (D - k - 1)/2 \le (3k - k - 1)/2 = (2k - 1)/2 \le 2k/2 = k$
$\Rightarrow p - (k + 1) \le k$ and $D - p \le k$.
<u>CASE 2:</u> $k + D + 1$ is odd.
$p = (k + D)/2$
$p - (k + 1) = (D - k)/2 - 1 \le (3k - k)/2 - 1 = (2k)/2 - 1 \le k - 1 \le k$
$\Rightarrow p - (k + 1) \le k$.
$D - p = (D - k)/2 \le (3k - k)/2 = 2k/2 = k$
$\Rightarrow D - p \le k$.
This proves our claim.

Then, $\min(dist(u, s), dist(u, t)) \le dist(u, s) \le k$, for all $u \in S$, and

$\min(dist(u,s), dist(u,t)) \leq dist(u,t) \leq k$, for all $u \in S'$.
Therefore, $\min(dist(u,s), dist(u,t)) \leq k$ for all $u \in S \cup S' = V$, for any $t \in V_p$, where $p = \lfloor \frac{k+D+1}{2} \rfloor$, where $D = \max(\{dist(u,s) \mid u \in V\})$.