ESO207 THEORETICAL ASSIGNMENT 2

Submitted by: Aditi Khandelia, Roll No. 220061

8^{th} October 2023

1) You are given an array A[0, . . . , n−1] of n distinct integers. The array has following three properties:

1. **First (n−k) elements are such that their value increase to some maximum value and then decreases.**
2. **Last k elements are arranged randomly**
3. **Values of last k elements is smaller compared to the values of first n−k elements.**

(a)You are given q queries of the variable Val. For each query, you have to find out if Val is present in the array A or not. Write a pseudo-code for an O(klog(k)+qlog(n)) time complexity algorithm to do the task.

Ans.

First, the value of k can be determined using binary search. For each iteration, if the value of the middle index of the range is less than the value at the start of the array, we move our search to the part preceding the middle of the range. This is because the value of the last k elements are all smaller than the value of the first n−k elements, and thus smaller than the value at start of the array. Similarly, if the value at the middle index of the range is greater than the value at the start, we move our search to the part succeeding the middle.

Function to return the value of k:

```
findk(A,n) {
    left=0;
    right=n-1;
    while(left<right)
    {
        mid=(left+right)/2;
        if(A[mid]>A[0]) left=mid+1;
        else right=mid;
```

```
    }
    return left;
}
```

Next, we sort the values of the last k elements in decreasing order. Since the values of the last k elements are smaller than the values of the first n−k elements, the last k elements will follow the first n−k elements in descending order arrangement. Thus, sorting the last k elements sorts the entire array.

Function to sort the last k elements:

```
mergesort(A,left,right) //Initially, left=k and right=n-1
{
    if(left¡right)
    mid=(low+right)/2;
    mergesort(A,left,mid);
    mergesort(A,mid+1,right);
    A=merge(A,left,mid,right);
}

merge(A,left,mid,right)
{
    i=left;
    j=mid;
    B[right-left+1];
    k=0;
    while(i<=mid and j<=right)
    {

        if(A[i]>=A[j]) {B[k]=A[i];k++;i++;}
        else {B[k]=A[j];k++;j++}
    }
    while(i<=mid) {B[k]=A[i];k++;i++;}
    while(j<=right) {B[k]=A[j];k++;j++}
    k=0; for(int i=left;i<=right;i++)
    {

        A[i]=B[k];
        k++;
    }
    return A;
```

```
}

Function to answer an array of queries Q(of size q)
solvequery(A,Q,q)
{


    For(int i=0;i<q;i++)

    {


        tofind=Q[i];
        low=0;
        high=n−1;
        while(low<high)
        {


            mid=(low+high)/2;
            if(A[mid]==tofind) break;
            else if(A[mid]>tofind) low=mid+1;
            else high=mid-1;

        }
        if(low==high) cout<<Q[i] is present<<endl;
        else cout<<Q[i] is not present<<endl;

    }

}
```

(b) Explain the correctness of your algorithm and give the complete time complexity analysis for your approach in part (a).

Ans.
To prove the correctness of the algorithm, we need to prove that function findk finds the value of k and that sorting the last k elements sorts the entire array. Correctness of merge sort for sorting the array and binary search for finding an element in a sorted array has been given in lecture notes.

Proof of correctness of findk:
P(0) is true since initially left=0 and right=n−1 and k lies between left and right.
Let P(i) be true, i.e. after the i^{th} iteration, k is present between left and right.
P(i+1):

Suppose A[mid]>A[0]. Since the last k elements are all smaller than the first n−k elements, k will be strictly after mid. Since we set left=mid+1 and high remains the same, k is present in the new interval.
Suppose A[mid]<A[0]. Since the last k elements are all smaller than the first n−k elements, A[mid] belongs to the last k elements. Thus, k will be present before, or is equal to, mid. We set high=mid and left remains the same, and thus k is present in the final interval.
Since for every iteration left<right, mid>=left and mid<right, in each iteration we either increase left or decrease right, and exit the loop as soon as left=right, we finally get an interval of length 1. Since this interval must contain k, the final value of left gives k.

Proof that sorting the last k elements in decreasing order sorts the entire array:
P(1): array is sorted for left=0 and right=n−k+1. This will always be true since for all i<n−k+1 elements, A[k]<A[i]. Since the first n−k elements are sorted in decreasing order and the $(n-k+1)^{th}$ element is smaller than the n−k elements, array is sorted.
P(i): assume that the array is sorted for left=0 and right=n−k+i.
P(i+1): Since the last k elements have been arranged in decreasing order, A[n−k+i+1]¡A[n−k+i].Now, array is sorted from the assumption till A[n−k+i] and we are appending a number smaller than all previous numbers at the end. Thus, array is sorted for left=0 and right=n−k+i+1.
Thus, we finally get a sorted array.

Time Complexity Analysis:
Let the execution time of findk for n be T(n)
Since in each ietration, left is either increased to mid+1 or right decreased to mid, the input size halves. Apart from thsi, each iteration contains some operations that require O(1) time. Thus, recurrence relation=

$$T(n)=T(n/2)+O(1),T(1)=0(1)$$
$$T(n)=T(n/4)+2*O(1)$$
$$T(n)=T(n/2^k)+k*O(1)$$
let k=floor(log(n)). Then,
$$T(n)=T(n/2k)+k*(O(1)) \rightarrow T(n)=log(n)*O(1)$$
$$T(n)=O(log(n))$$

We can assume the time complexity analysis of merge sort for input of size k to be O(klog(k))

Time complexity analysis for solvequery(A,Q,q):
The first for loop gets executed q times with the input array size being n.
The inner for loop is executed till the array size<=1. During each iteration of the inner loop, input size gets halved in the worst case(in the best case A[mid]=Val).
Let the time taken by the inner for loop be T(n).

$$T(n)=T(n/2)+O(1), T(1)=0(1)$$
$$T(n)=T(n/4)+2*O(1)$$
$$T(n)=T(n/2^k)+k*O(1)$$
let k=floor(log(n)). Then,
$$T(n)=T(n/2k)+k*(O(1)) \rightarrow T(n)=\log(n)*O(1)$$
$$T(n)=O(\log(n))$$

Time complexity of outer for loop= $q*O(\log(n))=O(q*\log(n))$
Thus, time complexity for whole algorithm= $O(k\log(k)+q*\log(n))$.

2.A directed graph with n vertices is called Perfect Complete Graph if:

1. **There is exactly one directed edge between every pair of distinct vertices.**
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.**

(a) Prove that a directed graph is a Perfect Complete Graph if and only if between any pair of vertices, there is at most one edge, and for all k $\in$ {0,1,...,n1}, there exist a vertex v in the graph, such that Outdegree(v) = k.

Ans.

1)If there is at most one edge between every 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 the directed graph is a Perfect Complete Graph:

The first condition is true since the characterisation of Perfect Complete Graphs states that each pair of vertices should have one directed edge. Thus, each pair of vertices can have atmost one edge between them.
For the second condition, we use Principle of Mathematical Induction on the number of vertices in the graph.
P(1):
Consider a graph with one vertex. Since to be characterised as a Perfect Complete Graph, it must not have any edge emanating from it, outdegree(v)=0.
Now, k $\in$ {0}. Thus, P(1) is true since we get a vertex for $\forall$ k $\in$ {0,1,...,n1}
P(n-1):
Assume that the condition is true for a graph with n$-$1 vertices, i.e. $\forall$ k $\in$ {0,1,2,...,n$-$2}, is there exists a vertex such that outdegree(v)=k and each pair of vertices has atmost one edge in between them, then the graph is a perfect complete graph.
P(n):
Let there be a graph with n vertices such that $\forall$ k $\in$ {0,1,2,...,n$-$1}, there exists a vertex such that outdegree(v)=k and each pair of vertices has atmost one edge in between them.
Consider the vertex v such that outdegree(v)=0.
Consider the vertex with outdegree=n-1. This vertex has an edge emanating from it to every other vertex in the graph.
Similarly, consider the vertex with outdegree=n-2. Thus, there is only vertex to which this vertex doesn't have an edge. If possible, let this vertex not be the vertex with outdegree=n-1. Then, there is an edge from v_{n-2} to v_{n-1}. It has already been proven that there is an edge from v_{n-1} to v_{n-2}.

This is in contradiction to the assumption that there is atmost one edge between any pair of vertices.
Thus, there is no edge from v_{n-2} to v_{n-1}.
Similarly, it can be proven that v_i has an edge to every v_k such that k<i.
Thus, v_0 has an edge to it from every other vertex in the graph.
Remove v_0. Since all vertices in the graph had an edge to it, the outdegree of all is reduced by 1.
We thus now have a graph with n-1 vertices such that $\forall$ k $\in$ {0,1,...,n-2}, there exists a vertex v such that outdegree(v)=k and all pairs of vertices have atmost one edge between them.
By induction, this graph is a Perfect Complete Graph.
Thus, there is exactly one directed edge between every pair of distinct vertices in the graph.
Now, add v_0 to the graph as well as the edges from all the vertices to v_0 to restore the original graph.
Assume there are any two vertices in the graph a and b. If one of these if v_0, we have already proven an edge will exist between them.
If none of these vertices is v_0, the two vertices will belong to the previous Perfect Complete Graph and thus have an edge.

Take any three vertices in the graph a,b and c.
Case 1: One vertex is v_0. Let this be c.
Since a and b were present in the previous perfect complete graph, there exists an edge between them. Without loss of generality, assume the edge is from a to b.
There also exists an edge from b to v_0 and from a to v_0. Thus, when an edge exists from a to b and an edge exists from b to v_0, there exists an edge from a to v_0.
Case 2: Suppose none of the vertices are v_0. Then all of them belong to the previous Perfect Complete Graph and thus satisfy the second property of Perfect Complete Graphs.

Our graph thus satisfies both the properties of Perfect Complete Graphs.
Thus, if there is at most one edge between every 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 the directed graph is a Perfect Complete Graph.

2)For a perfect complete graph with number of vertices=n, there is at most one edge between every 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.

We prove this using Principle of Mathematical Induction.
P(1):
Consider a perfect complete graph with one vertex. It will not have have any edges emanating from it. Thus, bith the conditions are satisfied and P(1) is true.

P(n-1):
Suppose for every perfect complete graph with n−1 vertices, there is at most one edge between every pair of vertices, and for all k ∈ 0,1,...,n−2, there exist a vertex v in the graph, such that outdegree(v) = k.
P(n):
Let there be a perfect complete graph with n vertices. If possible, let there be no vertex with outdegree 0. Then all vertices have outdegree>=1. Suppose one such vertex is a_1. Since outdegree(a_1)>=1, it has atleast one edge emanating from it. Let this edge be to a_2. By assumption, outdegree(a_2)>=1. We continue this process for a_3,a_4,...,a_p.
Case 1:a_p=a_t ∃ t<p.
In a perfect complete graph, whenever a path exists from a_t to a_p, there is an edge from a_t to a_p. For a_{p-1}, there is an edge from a_{p-1} to a_t and an edge from a_t to a_{p-1}. This can't happen in a perfect complete graph.
Thus, $a_p \neq a_t$ ∀ t<p.
Case 2:p=n.
Since outdegree(a_n)>=1, a_n will contain an edge to atleast on a_t such that t<n. But an edge already exists from a_t to a_n. Thus, the first property of perfecxt complete graphs is violated.
Thus, there is atleast one vertex v in graph such that outdegree(v)=0.
Remove this vertex.
We now get a graph with n-1 vertices.
Consider any two vertices a and b in this graph. Since the original graph was a perfect complete graph, there exists an edge between a and b.
Consider any three vertices a,b and c.If (a, b) and (b, c) are directed edges, then (a, c) is present in the graph(therefore, the original graph is a perfect complete graph).
Thus,the graph obtained is a perfect complete graph.
By induction, there is at most one edge between every pair of vertices, and for all k ∈ 0,1,...,n−2, there exist a vertex v in the graph, such that outdegree(v) = k.
Now add the vertex we had removed(let this vertex be v). An edge is supposed to exist between any vertex and v. Since outdegree(v)=0, all the vertices in the graph have a directed edge to v.
If the outdegree of a vertex in the perfect complete graph with n-1 vertices was i, its outdegree in the original graph=i+1.Hence,for all k ∈ 0,1,...,n−1, there exist a vertex v in the graph, such that outdegree(v) = k.
Additionally, since the graph is a perfect complete graph, there is one directed edge between any pair of vertices.

It is thus proven that for a perfect complete graph with number of vertices=n, there is at most one edge between every pair of vertices, and for all k ∈ 0,1,...,n−1, there exist a vertex v in the graph, such that outdegree(v) = k.

b.Given the adjacency matrix of a directed graph, design an O(n^2)

algorithm to check if it is a perfect complete graph or not. Show the time complexity analysis. You may use the characterization given in part (a).

Ans.

The adjacency matrix will have a[i][j]=1 when a directed edge exists from i to j and 0 otherwise. Using the characterisation in part(a), we will calculate the out-degree of each row. If for all k $\in$ 0,1,...,n$-$1, there exist a vertex v in the graph, such that outdegree(v) = k and there exist no i, j such that a[i][j]=a[j][i](that is no pair of vertices with 0 or 2 edges between them), we get a perfect complete graph.

```
isperfectcomplete(a,n)
{
    int k[n]; // matrix to store the outdegrees
    for(int i=0;i<n;i++)

        outdegree=0;
        for(int j=0;j<n;j++)
            if(a[i][j]==1) outdegree++;
            if(a[i][j]=a[j][i]) return 0;
        k[outdegree]++;

    for(int i=0;i<n;i++) if(outdegree[i]!=1) return 0;
    return 1;
```

Time Complexity Analysis:
Consider the code block:
for(int i=0;i<n;i++) ...executed n times.

```
    outdegree=0; ...O(1)
    for(int j=0;j<n;j++) ...executed n times

        if(a[i][j]==1) outdegree++; ...O(1)
        if(a[i][j]=a[j][i]) return 0; ... O(1)

    k[outdegree]++; ...O(1)
```

Thus, total no. of instructions= n$*$(2+n$*$2) = 2n+2n^2

Next, consider the code block:
for(int i=0;i<n;i++) if(outdegree[i]!=1) return 0;
return 1;
Thus, total no of instructions= n+1(for loop executed n times with 1 instruction inside)
Hence, total no of instructions in the function= 3n+2n^2+1
Time Complexity=O(n^2)

3.You are given an array A = [a_1, a_2, a_3, . . . , a_n] consisting of n distinct, positive integers. In one operation, you are allowed to swap the elements at any two indices i and j in the present array for a cost of max(a_i,a_j). You are allowed to use this operation any number of times. Let π be a permutation of 1,2,...,n. For an array A of length n, let A(π) be the permuted array A(π) = [$a_{\pi_0(1)}$, $a_{\pi_0(2)}$, . . . , $a_{\pi_0(n)}$]. We define the score of an array A of length n as

$$\sum_{i=1}^{n-1} |a_{i+1} - a_i|$$

a.Explicitly characterise all the permutations $A(\Pi_0)$ = $a_{\pi_0(1)}$, $a_{\pi_0(2)}$, . . . , $a_{\pi_0(n)}$] of A such that a good permutation $A(\Pi_0)$ is one in which,

$$S(A(\Pi_0)) = \min_{\Pi} \sum_{i=1}^{n-1} |a_{i+1} - a_i|$$

Ans.

A good permutation will either be a the array sorted in the ascending or the descending orders.
In case we are able to minimize the each term in the sum, we will get the minimum possible score.
For any term a_i, the difference will be minimum for either the no. succeeding or preceding it ina sorted array.
Thus, arranging the array in ascending or descending order will ensure that this condition is satisfied at each point in the array, ensuring we get the minimum possible score.

b. Provide an algorithm which computes the minimum cost required to transform the given array A into a good permutation, A(π). The cost of a transformation is defined as the sum of costs of each individual operation used in the transformation.

Ans.

First, we create a 3D data structure that stores the value of the element in the permutation, its initial position and its final position.
This can be done using the following function. Let the array be A and its size be n.

```
createarray(A,n)
{
    B[n][n][n];
    for(int i=0;i¡n;i++)
    {
```

```
            B[i][0]=A[i];
            B[i][1]=i;

        mergesort(B,0,n-1); //this function will sort B on the basis of the
        first element
        for(int i=0;i¡n;i++)
        {

            B[i][2]=i;

        }

}
```

The following function will sort B on the basis of the first element:

```
mergesort(B,left,right)
{

        if(left¡right)
        mid=(low+right)/2;
        mergesort(B,left,mid);
        mergesort(B,mid+1,right);
        B=merge(B,left,mid,right);

}

merge(B,left,mid,right)
{

        i=left;
        j=mid;
        C[right-left+1][right-left+1];
        k=0;
        while(i<=mid and j<=right)
        {


            if(B[i][0]>=B[j][0]) {C[k][0]=B[i][0];C[k][1]=B[k][1];k++;i++;}
            else {C[k][0]=B[j][0];C[k][1]=B[j][1];k++;j++}

        }
        while(i<=mid) {C[k][0]=B[i][0];C[k][1]=B[k][1];k++;i++;}
        while(j<=right) {C[k][0]=B[j][0];C[k][1]=B[j][1];k++;j++}
        k=0; for(int i=left;i<=right;i++)
        {


            B[i][0]=C[k][0];
```

```
            B[i][1]=C[k][1];
            k++;
        }
        return B;
}
```

Thus, finally we get an array B such that it is sorted along the first dimension, and if B[i][1]=k, B[i][2]=π(k).
We can now detect cycles. Whenever we get a cycle, the cost will include all the members of the cycle except for the minimum element of the cycle.

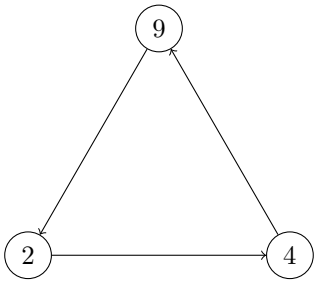
```
findmincost(B,n)
{
        cost=0;
        currcycle=n-1;
        visited[n];
        for(int i=0;i¡n;i++) visited[i]=0;
        while(currcycle>=0)
        {
            if(visited[currcycle]) currcycle–; continue;
            currreplace=B[currcycle][1];
            int min=B[currcycle][0];
            visited[currcycle]=true;
            while(currreplace≠currcycle)
            {
                cost+=B[currreplace][0];
                if(min>B[currreplace][0]) min=[currreplace][0];
                visited[currreplace]=true;
                currreplace=B[currreplace][1];
            }
            cost+=B[currreplace][0];
            cost-=min;
            currcycle–;
        }
        return cost;
}
```

This process needs to be repeated for when the array is arranged in descending oprder as well and the costs needs to be computed for this.
The minimum of the two costs will be the final cost.

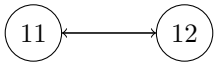
c.Prove that your algorithm computes the minimum cost of converting any array A into a good permutation.

Ans.

Suppose the array A is [4,3,9,2,12,11]. Then the resulting array B will be [[2,3,0],[3,1,1],[4,0,2],[9,2,3],[11,4,3],[12,3,4]].
Clearly, the numbers that need to be switched are 2, 4, 9,11 and 12.
Begin with 9. It needs to be moved to the position 3.
At position 3, we have 2.2 needs to be moved to position 0.
At position 0, we have 4. 4 needs to be moved to position 2, which was the position of 9.
This sequence of movements can be represented in the form of a cycle.



Now suppose we begin switching with 9 . It swaps position with 2.
The cost added is max(9,2)=9.
Now, we have 9 in its correct location.
Next, we swap the next biggest element 4 with the element in its position. Since 9 has already been put into its right position, the element will never be 9. In this case, it will be 2.
Cost added=max(4,2)=4.
Now after swapping 4 and 2, we see that all the elements have gotten their final positions.
Thus, we didn't have to add the cost of putting the smallest element.



Now suppose we begin switching with 12 . It swaps position with 11.
The cost added is max(11,12)=12.
Now, both 11 and 12 are in correct positions and only the cost of 12 gets added.

4.Batman gives you an undirected, unweighted, connected graph G = (V, E) with |V| = n, |E| = m, and two vertices s, t ∈ V . He wants to know dist(s,t) given that the edge (u,v) is destroyed, for each edge (u,v)∈E. In other words, for each (u, v)∈E, he wants to know the distance between s and t in the graph G' = (V', E'), where E' = E|{(u, v)}. Some constraints:

1. **The dist definition and notation used is the same as that in lectures.**
2. **It is guaranteed that t is always reachable from s using some sequence of edges in E, even after any edge is destroyed.**
3. **To help you, Batman gives you an NXN matrix M_{nxn}. You have to update M[u,v] to contain the value of dist(s, t) if the edge (u, v) is destroyed, for each (u, v) ∈ E.**
4. **You can assume that you are provided the edges in adjacency list representation.**
5. **The edge (u, v) is considered the same as the edge (v, u).**

a.Batman expects an algorithm that works in O(|V| · (|V| + |E|) = O(n · (n + m))

Ans.
The answer to this problem makes use of BFS traversal, which can give the shortest distance between two vertices in an undirected, unweighted, connected graph.
We also make use of the property that any path can have a maximum of n−1 edges in a graph with n vertices.
We maintain a 2D matrix arr[n][n] that stores dist(u,v) when edge (i,j) is removed from the graph, if present.
We first carry out BFS traversal, starting from index u to reach index v. We list out all the edges that exist in the path from u to v.
If any of these vertices are removed, we will have to carry out BFS traversal again to find the new shortest path to v. If an edge which doesn't belong to this path is removed, we needn't carry out BFS traversal again since the original path is retained.

```
findshortestdistancematrix(u,v)
{
    arr[n][n];
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++) arr[i][j]=∞
```

```
    queue q;
    distance(u)=0;
    visited(u)=true;
    parent(u)=null;
    q.enqueue(u);
    while(q.notempty)
    {
        a=q.dequeue();
        for neighbour b of a:
            if(visited(b)=false)
                distance(b)=distance(a)+1;
                visited(b)=true;
                parent(b)=a;
                q.enqueue(b);
    }
    node=v;
    while(parent(node)!=null)
    {
        parnode=parent(node);
        arr[parnode][node]=BFStraversalwithout(u,v,parnode,node);
        arr[node][parnode]=arr[parnode][node];
        node=parnode;
    }
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            if(arr[i][j]==∞) arr[i][j]=distance(v);
}
```

Thus, after execution, the algorithm gives shortest distance matrix arr

The following function carries out BFS traversal when a particular edge is removed and returns dist(u,v):

```
BFStraversalwithout(u,v,parnode,node)
{
    queue q;
    distance(u)=0;
    visited(u)=true;
    q.enqueue(u);
    while(q.notempty)
    {
```

```
        a=q.dequeue();
        for neighbour b of a:
            if(visited(b)=false)
                if(b==node and a==parnode) ;
                else
                   distance(b)=distance(a)+1;
                   visited(b)=true;
                   parent(b)=a;
                   q.enqueue(b);

    }
    return distance(v);

}
```

After execution, the algorithm gives the matrix arr where arr[i][j] stores dist(u,v) when edge (i,j) has been removed.

b.He also wants you to provide him with proof of runtime of your algorithm, i.e., a Time- Complexity Analysis of the algorithm you provide.

Ans.
The first part of the algorithm involves initialisation of shortest distance matrix arr[n][n].
for(int i=0;i<n;i++) ...executed n times

    for(int j=0;j<n;j++) ...executed n times

        arr[i][j]=∞ ... 1 instruction

Total no. of instructions= n$*$n.

Second part of the algorithm involves BFS traversal to check the edges in the shortest path.
During BFS traversal we check all neighbours of a particular vertex by checking along all the edges.
Whenever an unvisited vertex is found, it is inserted in the queue after performing some operations that take constant time.
Thus, time complexity of BFS traversal=O(n+m)

Next part of the algorithm involves moving up along the path from v to u. Every time an edge is encountered, BFS traversal is executed.
node=v;
while(parent(node)!=null)
{

parnode=parent(node);
arr[parnode][node]=BFStraversalwithout(u,v,parnode,node);
arr[node][parnode]=arr[parnode][node];
node=parnode;

}
Thus, the loop iterates at most n-1 times.
Also, time complexity of each iteration=time complexity of BFStraversalwithout+ O(1) for some constant instructions.
Time complexity of BFStraversalwithout will be the same as the original BFS traversal since only some extra instructions are added while adding each node to the queue to check is the edge is the edge that is supposed to be removed.Thus, time complexity=O(n+m)
Final no. of instructions= (n-1)*(c*(n+m)+d)+e where d and e are fixed constants.

Lastly we traverse the entire matrix once to see which distances have not been assigned. These edges do not belong to the shortest path and hence, distance is not affected by their removal.
for(int i=0;i<n;i++) ...executed n times

for(int j=0;j<n;j++) ...executed n times

if(arr[i][j]==∞) arr[i][j]=distance(v); ...O(1)

Total number of instructions=n$*$n.
Thus, the final no of instructions is a*(n+m) + (n-1)*(c*(n+m)+d)+e + f*n*n = O(n*(n+m)).

c.Lastly, you also need to provide proof of correctness for your algorithm.
Ans.
The proof of correctness involves proving that a path can have atmost n-1 edges when a graph has n vertices and the correctness of BFS traversal.

Proof that a path can have atmost n$-$1 edges:
If possible let a graph with n vertices have a path with no. of edges >n-1.
Since a path can have a vertex appearing in it only once and each edge is bound by two vertices, no of vertices=no. of edges+1>n-1+1=n.
But our graph has n vertices. Thus, the maximum length of a path can be n-1.
Only the removal of these edges can thus potentially change the distance from u to v, and so BFS traversal needs to occur atmost n times.

Proof of Correctness of BFS traversal :
P(0):BFS traversal begins with u and thus visits all the vertices at distance 0 from it.
P(i):Assume BFS traversal visits all the vertices at distance i from it.

P(i+1):
Consider any vertex t at a distance i+1 from it. Consider all neighbours of t. Atleast one of these neighbours is at a distance i from u. Let this vertex be u'. When u' gets visited, it is added to the queue.
Since the loop runs till the queue has elements, u' gets dequeued.
When u' gets dequeued, all its unvisited neighbours get visited. Since distance of neighbour=distance of parent+1, distance(t)=distance(u')+1=i+1, visited(t)=true.
Thus, all the vertices at a distance i+1 get visited.

5.You are given an undirected, unweighted and connected graph G = (V,E), and a vertex s∈V, with |V| = n, |E| = m and n = 3k for some integer k. Let distance between u and v be denoted by dist(u,v) (same definition as that in lectures).
G has the following property:
Let $V_d \subseteq$ V be the set of vertices that are at a distance equal to d from s in G,then ∀ i >= 0: u∈V_i,v∈V_i+1 →(u,v)∈E

a.An O(|V| + |E|) time algorithm to find a vertex t∈V , such that the following property holds for every vertex u∈V:
min(dist(u, s), dist(u, t)) <= k
Note that your algorithm can report s as an answer if it satisfies the statement above.

Ans.

The algorithm will involve BFS traversal of the graph beginning from s.
The maximum distance of a vertex from s will be 3k-1 since the graph has 3k vertices.
Additionally, because of the property ∀ i >= 0: u∈V_i,v∈V_i+1 →(u,v)∈E, the distance between a vertex at a distance a from s and a vertex at a distance b from s will be |b-a|.
Suppose the maximum distance of a vertex from s is n. Then, t will be any vertex at a distance n−k if n>k, else t will be s.

```
givet(s,k)
{
    queue(q);
    distance(s)=0;
    currdistance=0;
    visited(s)=true;
    q=enqueue(q,s);
    while(notempty(q))
    {
        a=dequeue(q);
        for neighbour w of a:
        {
            if(!visited(w))
            {
                distance(w)=distance(a)+1;
                visited(w)=true;
                currdistance=distance(w);
                q=enqueue(q,w);
```

```
            }
        }
    }
    if(currdistance>k)
    {
        queue(q);
        if(distance(s)==(currdistance-k) return s;
        checked(s)=true;
        q=enqueue(q,s);
        while(notempty(q))
        {
            a=dequeue(q);
            for neighbour w of a:
            {
                if(!checked(w))
                {
                  if distance(w)==(currdistance-k) re-
                  turn w;
                  checked(w)=true;
                  currdistance=distance(w);
                  q=enqueue(q,w);
                }
            }
    }
    else return s;
}
```

b.Proof of correctness for your algorithm.
Ans.

Proof of correctness involves proving that for a vertex at a distance a from s and a vertex at a distance b from s, the distance between the two vertices will be |b-a| and that selecting a vertex t at a distance n-k, where n is the maximum distance from s, will ensure that min(dist(u, s), dist(u, t)) <= k $\forall$ u $\in$ V.

Proof that for a vertex at a distance a from s and a vertex at a distance b from s, the distance between the two vertices will be |b-a|:
Without loss of generality assume that b>a.
Consider any vertex v_b in V_b. By the property of V_d, we know that v_b is connected to all vertices in V_{b-1}. let one such vertex be v_{b-1}.Thus, no. of edges traversed becomes 1.
Now, do the same for V_{b-2},V_{b-3},...,V_{a+1}. Edges traversed=|b-a-1|.

For any vertex v_{a+1} in V_{a+1}. It will have an edge to v_a. Thus, we can reach v_a from v_b and the no. of edges traversed will be |b-a|.
Thus, the distance will be |b-a|.

Proof that selecting a vertex at a distance n-k, where n is the maximum distance from s, gives us the desired t:
The maximum distance of a vertex from s can be 3k-1 since the total no. of vertices in the graph is 3k. Thus, n<=3k-1. Hence, distance(t)<=2k-1.
If suppose a vertex v is at a distance <=k, dist(s,v)<=k and thus, min((dist(v,s),(v,t))<=k.
Consider any vertex at a distance >k. Maximum distance of such a vertex from t occurs when distance of vertex from s is k+1 and that of t is 2k-1. In this case, their distance is |k-2| which is less than k.
Thus, proven.