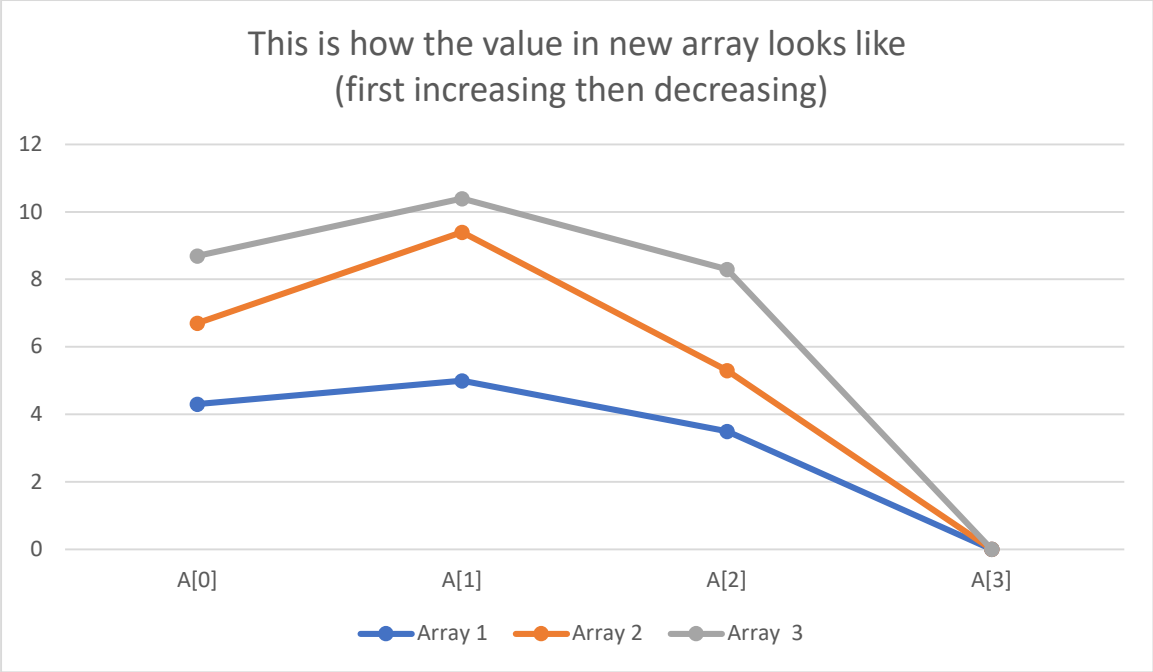
Question 1. Search Complicated

Divyansh Mathur, 210357

a.) Required time complexity= O(k log(k) + q log(n)):
To have this time complexity we need some precomputation. Idea is to sort the last k elements in decreasing order. Since the last k elements in the array were smaller than the first n-k elements, the array post the maximum element is now purely decreasing.
Now the task remains to find **val** in this *half increasing-half decreasing array*. To do this we will first find the maximum element location in precomputation. Then binary search in two Subarray i.e. from start to this max element and from max element to end.



Pseudocode:

```
Search Complicated(arr,n,k,q){
   sort(arr+n-k-1, arr+n-1);// sort all elements from n-k to n ,(0 based indexing)
   maxl=locatemax(arr,0,n-1);
   while(q--){
      input(val);
      if(findinc(val,0,maxl) || finddec(val,maxl,n-1) ==true) mark present;
      else mark not present;
   }
}

findinc(val, l, r){
```

```
    while(l<=r){
        mid=(l+r)/2;
        if(arr[mid]==val) return true;
        else if(arr[mid]>val) r=mid-1;
        else l=mid+1;
    }
}

finddec(val, l, r){
    while(l<=r){
        mid=(l+r)/2;
        if(arr[mid]==val) return true;
        else if(arr[mid]<val) r=mid-1;
        else l=mid+1;
    }
}

//logn function to find maximum element using binary search
locatemax(arr,l,r){
    while(l<=r){
        mid=(l+r)/2;
        if(arr[mid]>arr[mid-1] && arr[mid]>arr[mid+1]) return mid;
        else if(arr[mid]<arr[mid-1]) r= mid-1;
        else l=mid+1;
    }
}
```

b.) Proof of correctness:

Our new array is actually combination of two sorted arrays one increasing to a particular value and other decreasing from that value to some other value. We simply have to binary search these two subarrays. Either our val will be on right side of maximum element or on left side of the maximum element. If it is on neither side and not equal to the maximum element then it does not exist.

If an element is present in the array then it is compulsory for it to be either in 1st half(start to max element) or in other half (max element to end) or be equal to max element.
[because these 3 sets are mutually exclusive exhaustive set of array]
Now, since both these sets are sorted (increasing for 1st and decreasing for 2nd), It is easy to binary search on both these subarray

Proof of correctness of binary search:
Proving for increasing sorted array:
Assertion P[i]: $\boldsymbol{val} \notin$ {**A**[0],…,**A**[**L**-1] and $\boldsymbol{val} \notin$ { **A**[**R**+1],…,**A**[$n-1$]
Mid= (l+r)/2
If A[mid] >val then all the elements on right of mid are greater than val since increasing order hence new r=mid-1
Therefore P[i+1] still holds ($\boldsymbol{val} \notin$ { **A**[**R**+1],…,**A**[$n-1$])
If A[mid] <val then all the elements on left of mid are smaller than val since increasing order hence new l=mid+1
Therefore P[i+1] still holds ($\boldsymbol{val} \notin$ {**A**[0],…,**A**[**L**-1])
Similar for decreasing sorted array
Hence Prooved.

Time Complexity:

Precomputation Time:
Sorting k elements take klogk time
Finding maximum element using binary search take logn time
Total precomputation time = klogk+logn

Time for Query:
Finding an element in first subarray= log(M) { M location of maximum element}
Finding an element in second subarray= log(n-M)
Time for each query = log(M)+log(n-M) = O(logn)
For q query : O(qlogn)

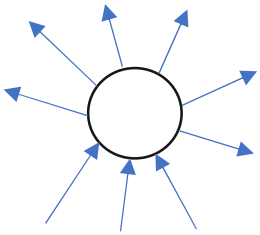
Overall Time complexity= O(klogk+qlogn)

Question 2: Perfect Complete Graph

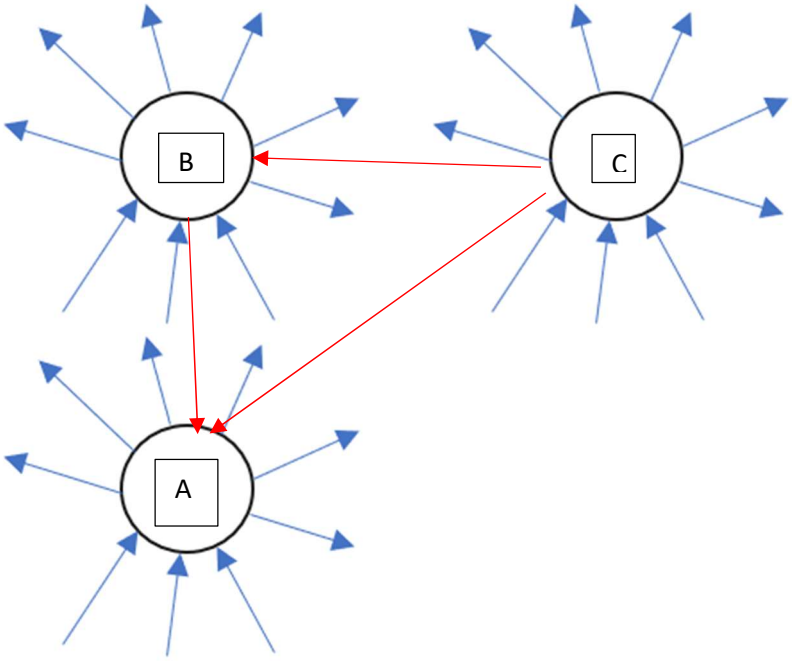
Divyansh Mathur, 210357

a.) CONDITION 1: For a directed graph to be perfect complete graph every two node should have exactly one edge between every pair of distinct vertices
CONDITION 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.

i.) **To prove:** If directed graph is a Perfect Complete Graph then between any pair of vertices, there is at most one edge, and for all k ∈ {0, 1, . . . , n − 1}, there exist a vertex v in the graph, such that Outdegree(v) = k



Consider this node,
Where k edges leaves this node (outdegree) and g edges enter this node (indegree). Total edges entering + leaving this node = n-1 since perfect complete graph condition 1. Therefore g=n-1-k
Assertion: Any two nodes can't have same outdegree.
Proof: Lets assume two nodes have same outdegree=k, name that A and B:



There is a edge between B and A. Assume B points to A.

Now take another node C which directs to B. There will be total n-1-k such nodes.

Now using condition 2 of perfect complete graph, this C node also points to A, So total indegree of A due to B and Cs=1+n-k-1 (1 from B and 1 from n-k-1 Cs)

Minimum possible indegree of A= n-k

➡ Maximum possible outdegree of A=k-1 (since total edges from a node=n-1 condition 1)

But this contradicts our assumption that A had outdegree =k.

Hence proved

ii.) To prove: If in a directed graph, between any pair of vertices, there is at most one edge, and for all k ∈ {0, 1, . . . , n − 1}, there exist a vertex v in the graph, such that Outdegree(v) = k then the graph is perfect complete graph.

According to the given statement there are n nodes everyone has different outdegree from 0 to n-1.

The node with outdegree n-1 has n-1 edges implies it points to all other (n-1) nodes,

Similarly the node with outdegree =n-2 points to all nodes except itself and except the above node {because the above node has already edge to this node and it is given that any two nodes can have at most 1 edge between them}

….

The node with 0 outdegree donot point to any node but it has indegree=n-1 because of above analysis.

From above analysis the following statement can be stated:

Statement X: If oudegree(b) > outdegree(a), then there is an edge between a and b where b points to a and
if b points to a then outdegree(b)> outdegree(a)

Therefore, a node is pointed by all nodes whose outdegree is greater than the node's outdegree. And it points to all nodes whose outdegree less than the node's outdegree. Hence total edges at each node =n-1

➡ any 2 nodes taken at a time are connected. (condition 1 of perfect complete graph)

Assume any three vertices a, b, c, such that (a, b) and (b, c) are directed edges i.e. a points to b and b points to c. This implies that Outdegree(a)> Outdegree(b) > Outdegree(c)

Outdegree(a)> Outdegree(c), using statement X, there is an edge between a and c where a points to c. (condition 2 of perfect complete graph).

Hence Proved.

b.) Adjacency matrix:

Sending nodes / Receiving node	0	1	.	.	n-1
0					
1					
.					
.					

n-1					

We can make use of an observation that each node is having an edge with each other node. Therefore if we see, adj[i][j] and adj[j][i] exactly one of them should be 1 and other should be 0. Now, we see the outdegree for each node and store them in another array say Out[n].

Then According to a.) part this array should have all values from 0 to n-1. Sort this array and compare with whole no. from 0 to n-1

Pseudocode:

```
Is_perfect_complete_graph{
        if(check(matrix)==false){
           return false;
        }else{
           for(i from 0 to n-1){
              out[i]=0;
              for(j from 0 to n-1){
                 out[i]=out[i]+matrix[i][j];
              }
           }
           sort(out.begin, out.end);
           for(i from 0 to n-1){
              if(i!=out[i]) return false;
           }
           return true;
        }
}
check(matrix){
          for(i from 0 t0 n-1){
             for(j from 0 to n-1){
                if(matrix[i][j]+matrix[j][i]!=1) return false;
             }
           }
           return true;
}
```

Time Complexity:

In check function we are visiting each node one time. This take $O(n^2)$ time. Further our algorithm sorts the n element array. This may take O(nlogn) time worst case also we are seeing each element of out array takes O(n) time. So our overall time complexity $=O(n^2)$.

Question 3. PnC

Divyansh Mathur, 210357

a.) Characterise all the permutations A(Π_0) = [aΠ_0(1), aΠ_0(2), aΠ_0(3) . . . , aΠ_0(n)] of A such that S(A(Π_0)) = min Π S(A(Π))

To minimize the score, we need to minimize the sum $\sum_{i=1}^{i=n-1} | a_{i+1} - a_i |$.

Point to note is that the maximum element will always come out of the modulus as +ve and the smallest element will come out of the modulus as negative so we will have

$$a_{max} - a_{min} + S$$

Our aim remains to minimize this sum S, this sum can be easily visualised as 0 when the array is sorted in **increasing and decreasing order.**

When in decreasing order

$$\text{Score} = | a_0 - a_1 | + | a_1 - a_2 | + \cdots . | a_{n-2} - a_{n-1} |$$
$$= a_0 - a_1 + a_1 - a_2 + \cdots . . a_{n-2} - a_{n-1}$$
(since sorted in decreasing order)
$$= a_0 - a_{n-1} \{ a_{max} - a_{min} \}$$

When in increasing order

$$\text{Score} = | a_1 - a_0 | + | a_2 - a_1 | + \cdots . | a_{n-1} - a_{n-2} |$$
$$= a_1 - a_0 + a_2 - a_1 + \cdots . . a_{n-1} - a_{n-2}$$
(since sorted in decreasing order)
$$= a_{n-1} - a_0 \{ a_{max} - a_{min} \}$$

Good permutations : Array sorted in 1. Increasing and 2. Decreasing order.

b.) Algorithm which computes the minimum cost required to transform the given array A into a good permutation:

In previous part we have seen that the good permutation is just the increasing or decreasing order of the array.

Idea is to sort the array (in both increasing and decreasing fashion) and find the minimum cost to sort the array.

Observations:

Consider the maximum element, if this element is not located on its required place then it is to be swapped and whichever element it is swapped with, the cost will be the maximum element. So we first place this maximum element on its required position i.e. at the last. Now our working space has been reduced, this element will not interfere with the cost anymore. Again we place the maximum element of the remaining array on its correct position and so on.

Algorithm:

Let the given Array be A[n]

First sort the array and store it in another array say B[n].

Create two arrays: temp[n] and pos[n] ,

temp array: stores where is the element in the sorted array such that A[i]= B[temp[i]]

pos array : stores where is the element in the original array such that B[i]=A[pos[i]]

temp array and pos array are related such that pos[temp[i]]=i

temp array can be easily filled using binary search on the sorted array for each element.

Ex. A[5]= 7,2,5,4,1

Sorting it: B[5]= 1,2,4,5,7

Filling temp array{

Find A[0] in B array using binary seach: 5 th position (4 in 0 based indexing),

Similarly finding other elements and filling temp array:}

temp[5]= 4,1,3,2,0

pos[5]=4,1,3,2,0

verifying: B[0]=1, pos[0]=4, yes 1 is A[4]

B[1]=2, pos[1]=1, yes 2 is A[1] and so on..

Sorting takes nlogn time
binary search for each element takes logn time and nlogn for all
Now for our algorithm, we need to move from right to left in our B array and see where this element should have been, If it is at its position than ok, else swap it and add the maximum value to cost.
This takes O(n) time since for each B[i] we are taking constant time to swap

Overall Time complexity = O(nlogn)

Swapping

Say, temp[i]=k and pos[i]=j
implies pos[k]=i and temp[j]=i
To swap we need to bring A[j] to A[i]
We can't just use the swap function because then our pos and temp array will not be functional as they don't map value to index they map index to index.

Now DO,
temp[j]=k and temp[i]=i
pos[k]=j and pos[i]=i

A: A[0] , A[1], A[2]….A[j]…. A[i]….A[n-1]

B: B[0], B[1], B[2] ….B[k]…..B[i],…..B[n-1]

NEED TO CHECK FOR BOTH INCREASING AND DECREASING ORDER:

We will do the same analysis for decreasing order also and report
min(cost_increasing, cost_decreasing)

Question 4: Mandatory Batman Question

Divyansh Mathur,210357

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

dist(s,t): (shortest) distance path between s and t.

Point to note is that the shortest distance between 2 nodes doesn't change if the edge not lying in the shortest path is removed. So we need to find shortest distance between s and t only when edges in this shortest path is removed.

To find dist(s,t): To find the shortest path between s and t simply start bfs from s and terminate it when it reaches t.

To store shortest path: Once we are pushing a node's neighbor to queue, we know that this neighbor comes just after the node and hence we can say the neighbor's parent is the node. Using this-

We make another array parent[n] where we store the parent of each vertex (node),

Pseudocode to record parent[v] and distance(s,t) when none edgeis deleted:

```
->Start BFS from s Until we reach t
->Initialising boolean visited array visited[n]=false  all false initially
->mark visited [s]=true
->Initialising Distance array Distance[n]=infinty(very high value)
->mark Distance [s]=0
->Initialising parent array parent[n]=-1 ( random not to be used value)
->Initialising empty queue Q
->then pushing s(starting node) to it
Q.push(s)
while(Q is not empty){
   u= Q.front
   Q.pop
   if(u == t) end BFS
   for( each neighbor v of u){
      if( visited[v]==false){
         if(Distance[v]>Distance[u]+1){
            Distance[v]=Distance[u]+1
         }
         visited[v]=true
         parent [v]=u
         Q.push(v)
      }
   }
}
```

Data Structure to store the ans: We are given with matrix (nxn), such that M[u,v] stores the dist(s,t) when (u,v) edge is removed. From above analysis we can infer that (u,v) affects dist(s,t) only if it is

part of the shortest path and edges in shortest path are of the form (u,parent(u)), Since dealing with undirected graph we will do same operation on (parent(u),u). We need to alter values of (u,parent(u)) and (parent(u),u) starting from t.

We will run BFS each time to find distance (s,t) removing (u,parent(u)) edge

Pseudocode to update matrix M:

```
UpdateMatrix{
   node=t
   while(parent[node]!=-1){
      M[node][parent[node]]=newdistance(node, parent[node])
      M[parent[node]][node]=M[node][parent[node]]
      node= parent [node]
   }
}

newdistance(node , parent [node]){
   ->Start BFS from s Until we reach t
   ->Initialising boolean visited array visited[n]=false  all false initially
   ->mark visited [s]=true
   ->Initialising Distance array Distance[n]=infinty(very high value)
   ->mark Distance [s]=0
   ->Initialising empty queue Q
   ->then pushing s(starting node to it)
   Q.push(s)
   while(Q is not empty){
      u= Q.front
      Q.pop
      if(u == t) end BFS
      for( each neighbor v of u){
         if(u== node and v== parent[node] || v==node and u== parent [node]) skip;
         else if( visited[v]==false){
            Distance[v]=Distance[u]+1
            visited[v]=true
            parent [v]=u
            Q.push(v)
         }
      }
   }
   return Distance[t]
}
```

b.) **Time Complexity Analysis:** We know that BFS takes O(n+m) running time. In our case also BFS will take worst case O(n+m) running time. Also we are calling BFS each time for each

node in the shortest path from s to t. There can be at max n-2 nodes in between and n-1 (u,parent(u)) pairs. Hence overall time complexity = O(n(n+m))

c.) Proof of correctness:

It is obvious that the edges that don't constitute the shortest path don't disturb the dist(s,t)

Assertion: The algorithm correctly calculates the shortest distance once edge (u, parent(u)) is removed such that this edge lies in the shortest path.

Proof:

A[u]-Since we are running BFS and while running BFS we are skiping the edge encountered , we are in sense destroying the edge (which is required), then we are calculating new distance, which implies that the new distance is the new shortest path.

A[parent[u]]- When the edge (parent(u), parent (parent (u))) is destroyed, again the shortest path cant be taken since it is destroyed now by again running BFS we find the new shortest path.

Hence Prooved.

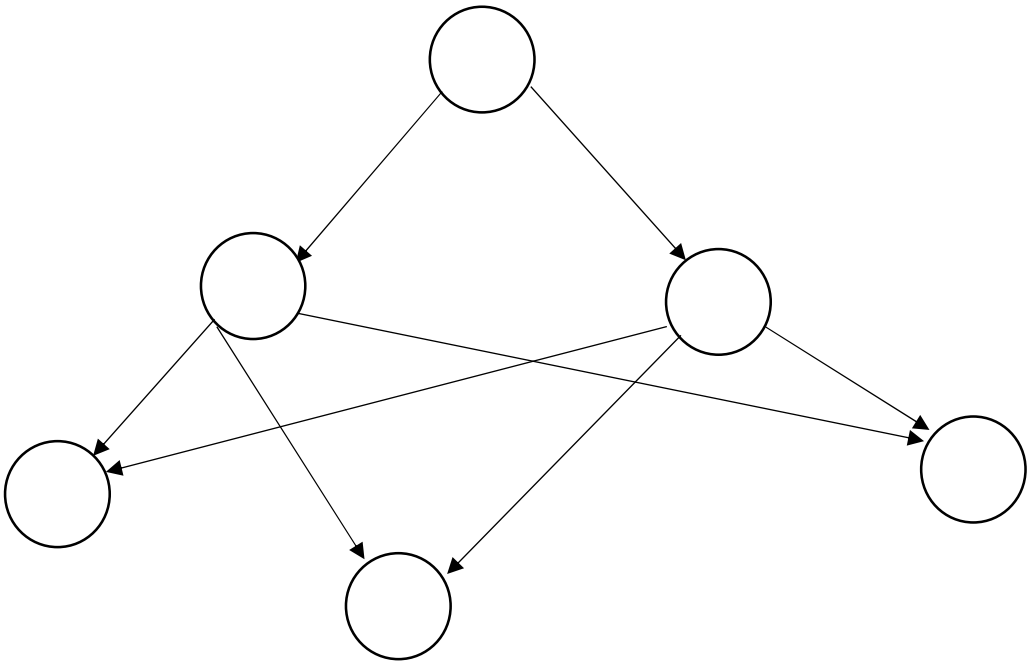
Question 5. No Sugar in this Coat

Divyansh Mathur,210357

• Let $V_d \subseteq V$ be the set of vertices that are at a distance equal to d from s in G, then $\forall\, i \geq 0$:
$u \in V_i$, $v \in V_{i+1} \Rightarrow (u, v) \in E$

This implies that if we start BFS from s, then each node at a level L will be linked to all the nodes at level L+1.

a.) **An O(|V |+|E|) time algorithm to find a vertex $t \in V$, such that the following property holds for every vertex $u \in V$: $\min(dist(u, s), dist(u, t)) \leq k$**
It is obvious that the last nodes (nodes which has maximum) will have the maximum distance from s. Now this last node say M, has the highest tendency to have dist(M,s) >k, we wish to find t such that it satisfies the condition with M just in range, i.e. dist(M,t)=k
Idea is to do BFS once and find the farthest level nodes. Once found we just have to move to its parent k-1 times to find t.



In this figure arrows are just for direction of BFS
use the following algorithm to find last node M:

```
BFS_TO_FIND_M
->initiate empty queue Q
->push s
-> iniate empty boolean visited[n]=false
-> make visited[s]=true
-> iniate empty array parent[n]=-1 random nonachievable value
while(Q is not empty){
   M=Q.top
   Q.pop
   for(all neighbours v of M){
      if(visited[v]==false){
```

```
                visited[v]=true
                parent[v]=M
            }
        }
    }
```

Now, to find t we just need to move up to its parent k-1 times

```
t=M
for(i=1 to k-1){
  t=parent[t]
}
```

b.) Proof of correctness for the algorithm:

If we consider a graph where each node is assigned to a level based on its distance,this graph has at 3k layers selecting a node from a level no greater than 2k+1 will ensure that the minimum distance between this selected node and both s and t is less than or equal, to k**.**

Dist(t,M)= k, dist(s,M)=?

Upper limit of dist(s,M)=n=3k (given)

$Dist(s,M)_{max}$=3k

Therefore $dist(s,t)_{max}$=2k, So any element lying in between will have min(dist(u,s),dist(u,t))<=k

Assertion : min(dist(s,u), dist(t,u)) <=k

Proof:

Let the Assertion is false i.e. there exist a u such that min(dist(s,u), dist(t,u)) > k

All the nodes below t are at max distance =k, hence this u don't lie below t

Nodes at the same level as t are at distance =2 from t, here arises 2 case:

1. k>=2 : here also u cant exist
2. k<2 or k=1 if k=1 total nodes= 3, t=s dist(u,s) for both neighbours of s=1 hence u cant exist here also

u can only exist in middle of s and t,

there are minimum k nodes below t (dist(t,M)=k)

Maximum no. of nodes between s and t = 2k-2

If we take the median elements at distance distance from top = k

And distance from t =k

Hence u cant lie here also

Since U cant lie anywhere in the graph our assumption was wrog,

Contradiction

Hence proved.