ESO207 Theoretical Assignment 2 Submission

Naman Kumar Jaiswal
220687
namankj22@iitk.ac.in

Fall Semester 2023

Question 1: Search Complicated

Part a:

We need to find an $\mathcal{O}(k\log(k) + q\log(n))$ to find whether an element is present in the array or not. The array elements increase to some maximum value and then decreases and then the last k elements are random. Array $\leftarrow$ [1, 2, 3, $\cdots$ n].

Thinking Process:

Since we have to find a solution in $\mathcal{O}(k\log(k) + q\log(n))$, per query q - taking q to be large - we have on average $\log(n)$ time to find the element. This is only possible if the array is sorted in some way.
The first n - k elements are two sorted portions. We need to take care of the next k elements in some way

Logic:

1. First we will sort the first k elements using any optimized sorting algorithm.
2. Then we will use binary search to find the maximum element in the first n - k elements.
3. Finally we will get two sorted components of the array on which we can use binary search twice to find whether an element is present or not in the array in $\mathcal{O}(log(n))$.

Principle for Optimization:

1. Use of binary search to find peak elements in an array which is a combination of two monotonic sequences in $\mathcal{O}(log(n-k))$.
2. Use of binary search twice to find an element in an array which is a combination of two monotonic sequences in $\mathcal{O}(log(n))$.

Pseudocode:

1. Function to find the peak element index p in the first n - k elements:

```
left ← 1, right ← n - k;
peak_element(left, right, Array){
    while (left <> right){
        mid ← (left + right) / 2;
        if (Array[mid] < Array[mid + 1] && Array[mid] > Array[mid - 1]){
            left ← mid + 1;}
        else if (Array[mid] > Array[mid + 1] && Array[mid] < Array[mid - 1]){
            right ← mid - 1;}
        else {return mid;}
    return mid;}
p ← peak_element(left, right, Array);
```

2. Function to sort the next k elements in decreasing order (from Notes):

```
left ← n - k + 1, right ← n;
QuickSort(Array, left, right){
    if (left < right){
        i ← Partition(Array, left, right);
        QuickSort(Array, left, i - 1); QuickSort(Array, i + 1, right);}
/*
Partition:
x ← Array[left] as a pivot element, permute the subarray Array[l, ··· r] such
   that elements preceding x are smaller than x, Array[i]= x, and elements
   succeeding x are greater than x. */
```

3. Function for finding whether an element z is present in the array which now contains two sorted components - from 1 to p and p + 1 to n:

```
left ← 1, right ← n;
Find (Array, z, left, right) {
    return BinarySearch(Array, z, left, p, 0) || BinarySearch(Array, z, p + 1,
     right, 1);

// Here the fifth input tells how the Search is to be made. 1 represents a
    decreasing array.
}
```

4. Pseudo Code for Binary Search:

```
BinarySearch(Array, left, right, target, order){
    if (left > right) {return -1;}
    mid = left + right / 2;
    if (arr[mid] = target) {return 1;}
    if (order = 1) {if (arr[mid] < target)
        {return BinarySearch(arr, left, mid - 1, target, order);}
    else {return BinarySearch(arr, mid + 1, right, target, order);}}
    else {if (arr[mid] < target)
        {return BinarySearch(arr, mid + 1, right, target, order);}
        else {return BinarySearch(arr, left, mid - 1, target, order);}}}
```

Part b: Proof of Correctness and Complexity Analysis

Proof of Correctness:

It is sufficient to show that if our algorithm returns true then the element surely exists in the array and if it returns false, it surely doesn't exist in the array.
Let us look at the algorithm closely - if an element exists in the array - the step involving sorting the last k elements does not introduce a new element or remove an existing element - neither does the step involving finding the peak element have to do anything with the addition or removal of any element from the array. Hence, if an element exists in the original array, it will also exist in the array involving two sorted components and if it does not, it will still not exist in the final array.
The element will have some position in the final array, thus it will be part of one of the sorted components of the array.
Since the binary search ends only if the element is found - if our algorithm returns true, it is guaranteed that the element exists in the array. Now, since binary search works on divide and conquer approach and if it reduces the potential region of availability of an element, it is trivial that the element can never exist outside the region. Hence, at the end if Binary Search return false, the element can never exist in the array. Thus, if our algorithm reports False, the element does not exist in the original array.
This concludes the proof of correctness.

Time Complexity Analysis:

For Time Complexity we need to look at the algorithm step by step:

1. Sorting last k elements using Quick Sort $\implies \mathcal{O}(k \log(k))$
2. Finding peak of the array $\implies \mathcal{O}(\log(n-k))$ (The code is an updated version of Binary Search)
3. q queries - each query requiring 2 binary searches $\implies q * \mathcal{O}(\log(p) + \log(n-p)) \implies q * \mathcal{O}(\log(n) \implies \mathcal{O}(q \log(n)$ (in worst case of p = n / 2)

So, Overall Time Complexity is $\mathcal{O}(q \log(n) + k \log(k) + \log(n-k)) \equiv \mathcal{O}(q \log(n) + k \log(k))$.

Question 2: Perfect Complete Graph

Part a: Proving Out-Degree Property of Perfect Complete Graph

To 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, . . . , n - 1}, there exists a vertex v in the graph, such that Outdegree(v) = k. We need to prove the both-way statement.

Logic:

Since both the statements can be defined in n, we can use the induction hypothesis to prove the statements.

Proof: Any Perfect Complete Graph Satisfies the given property

Let us define S(n):= In any Perfect Complete Directed Graph, if between any pair of vertices, there is at most one edge, then for all k in 0, 1, . . . , n - 1, there exists a vertex v in the graph, such that Outdegree(v) = k - 1.

1. Claim 1: S(2) holds true:
   Proof:
   The graph contains only 2 vertices and since there is exactly 1 edge between this pair of vertices, either (v_1, v_2) or (v_2, v_1) belongs to E. In both cases - one of v_1, v_2 has an out-degree 1 and the other 0. This is the exact statement of S(2). Hence, we can say S(2) holds.
2. Claim 2: Given S(n-1) holds - S(n) holds:
   Proof:

   Here, we use the nomenclature that v_k is the vertex with out-degree k. We are given that such a vertex exists.
   Also, let us assume that there are two vertices with the same out-degree. Since there exists at least one vertex v for each k from 0 to n - 2 for which the out-degree(v) = k. This makes the count of total vertices n, but the total number of vertices is n - 1. This leads to a contradiction. Hence, there is exactly 1 vertex v whose out-degree = k for each k in 0 to n-2.
   The graph of S(n-1) has every vertex with out-degrees from 0 to n-2. We are introducing another vertex v into the graph. Now from the definition either (v, v_{n-1}) or (v_{n-1}, v) belongs to E. This leads to 2 cases:

   1. If (v, v_{n-1}) belongs to E, then $(v, v_{n-2}), (v, v_{n-3}) \cdots (v, v_1)$ also belong to E from definition point 2 of perfect complete graph as v_{n-1} has outward edges to each of $v_{n-2}, v_{n-3} \cdots v_1$. S(n) holds.

   2. If (v_{n-1}, v) belongs to E, the out-degree of v_{n-1} becomes n-1 which is the maximum - there can only be 1 edge between any pair of vertices and excluding v_{n-1} there are only n - 1 vertices.
   Let us look at the out-degrees of the vertices - v_k has out-degree = k - 1 for k = 1 to n - 2 and k for k = n - 1. v has an incoming edge from v_{n-1} and we to draw edges from v to v_k for k = 1 to n - 2.

   This situation is similar to the situation we were in at the start of the proof.
   We will once again divide the cases as:
   2.1 If (v, v_{n-2}) belongs to E, then $(v, v_{n-3}), (v, v_{n-4}) \cdots (v, v_1)$ also belong to E from definition point 2 of perfect complete graph as v_{n-2} has outward edges to each of $v_{n-3}, v_{n-4} \cdots v_1$. S(n) holds.
   2.2 If (v_{n-2}, v) belongs to E, the out-degree of v_{n-2} becomes n-2 which is the maximum - there can only be 1 edge between any pair of vertices and excluding v_{n-1}, v_{n-2} there are only n - 2 vertices.
   Let us look at the out-degrees of the vertices - v_k has out-degree = k - 1 for k = 1 to n - 3 and k for k = n - 1, n-2. v has an incoming edge from v_{n-2} and we to draw edges from v to v_k for k = 1 to n - 3.

   We keep on peeling the layers of cases for further n - 3 times and keep on proving the cases where there is an edge (v_k, v). We will finally be left the case where v has an incoming edge from all vertices from the graph and out-degree of v_k is k for each k from 0 to n. Thus, S(n) holds.

Since, S(n) holds always we can claim S(n) holds for all n.

Proof: Any Graph Satisfying the given property is a Perfect Complete Graph

Let us define S(n):= A Graph, if between any pair of vertices, there is at most one edge and for all k in 0, 1, . . . , n - 1, there exists a vertex v in the graph, such that Outdegree(v) = k, is a Perfect Complete Graph. Also, let us assume that there are two vertices with the same out-degree. Since there exists at least one vertex v for each k from 0 to n - 1 for which the out-degree(v) = k. This makes the count of total vertices n + 1, but the total number of vertices is n. This leads to a contradiction. Hence, there is exactly 1 vertex v whose out-degree = k for each k in 0 to n-1.

1. Claim 1: S(n) holds true:
   Proof:
   Here, we use the nomenclature that v_k is the vertex with out-degree k - 1. We are given that such a vertex exists.
   The graph of S(n) has every vertex with out-degrees from 0 to n-1. Now, v_n has out edges to all other vertices. v_{n-1} has n - 2 out edges, since it can't have out edge to v_n, it will have out edges to all other vertices. Similarly, v_{n-2} has out edges to all vertices except v_n and v_{n-1}. Similarly, going on every vertex v_k an have out edges to vertices which have lower out degree than it. Hence, we can define that v_i and v_j have an edge from i to j if and only if i is greater than j. Now, greater than is a transitive relation in Natural numbers. Hence, so is the edge relation. Implying Property 2 is satisfied.

   Hence proved.

Part b: Algorithm to find whether a given graph is a Perfect Complete Graph

To find a Perfect Complete Graph we are going to use the characterisation in part a.

Thinking Process:

Since we have to find an $\mathcal{O}(n^2)$ algorithm we can traverse the adjacency matrix a constant finite number of times. Since there must be exactly one vertex having out-degree = k - 1 for all k in 1,2,3 $\cdots$ n. We can count the out-degree of each vertex and compare.

Logic:

1. First we traverse the adjacency matrix once and count the outdegrees for each matrix and if we get the value of count as c - we will increment A[c] by 1. Where A is an array whose i_th index stores the number of vertices in the graph that have out-degree = i.
2. Then we will check all the index elements of A and we find any value other than 1 - we return false, else true.

Principle for Optimization:

1. Using the characterization given in part (a).

Pseudocode:

1. Code to traverse the adjacency matrix adj once and making an array A:

```
for (i = 0; i < n; i++) { count ← 0; for (j = 0; j < n; j++) {
    if (adj[i][j]) {count++;}}
    A[count]++;}

for (k = 0; k < n; k++){
    if (count[k] <> 1) {return False;}
    return True;}
```

Time Complexity Analysis:

For Time Complexity we need to look at the algorithm step by step:

1. Time to traverse the adj matrix and make the A matrix $\implies \mathcal{O}(n^2)$
2. Traversing the A matrix $\implies \mathcal{O}(n)$

So, Overall Time Complexity is $\mathcal{O}(n^2 + n) \equiv \mathcal{O}(n^2)$

Question 3: PnC

Part a: Characterization:

Claim: Every permutation $A(\Pi)$ having the score S(A) minimized is monotonic in either of two directions. Either $A(\Pi)$ is in increasing order or in decreasing order.
Proof:

$$\sum_{i=1}^{n-1} |a_{i+1} - a_i| \geq |\sum_{i=1}^{n-1} (a_{i+1} - a_i)| \geq |a_n - a_1|$$

The equality holds only when either $a_{i+1} \geq a_i$ or $a_{i+1} \leq a_i \ \forall i \in \{1, 2, \cdots n-1\}$

Part b:

We have to find a $\mathcal{O}(n \log(n))$ to find the minimum cost to have a good permutation.

Thinking Process:

we have to find a solution in $\mathcal{O}(n \log(n))$ and we need to find two costs and compare them to find the minimum costs. We need to find each cost in $\mathcal{O}(n \log(n))$.
Also, the cost of the swap is dependent on the largest element, we have to look at the position of the element in the sorted array and compare. Since, anyhow we have to swap an element to its correct location, we have to make at least one swap.

Logic:

1. We will sort the given array and compare where the largest element is present. We will swap these two elements and then add the cost.
2. The only way in which the cost is reduced is when an element is already in it's place or there are two elements who are in each other's spots. We will need to check these conditions.

Principle of Optimization:

1. Use of Quick Sort for sorting. And looking for the cost-saving cases on the go to get the correct cost.

Pseudocode:

1. Code for quicksort:

```
QuickSort(Array, left, right){
    if (left < right){
        i ← Partition(Array, left, right);
        QuickSort(Array, left, i - 1); QuickSort(Array, i + 1, right);}
/*
Partition:
x ← Array[left] as a pivot element, permute the subarray Array[l, ··· r] such
   that elements preceding x are smaller than x, Array[i]= x, and elements
   succeeding x are greater than x. */
```

2. Code to get minimum cost

```
// Arr is the given array. Arr1 is the sorted array
Arr1 ← Arr; QuickSort(Arr1, 0, n-1);
Table[2][n]; // This table stores all the elements in sorted manner along with
     their index in the original array
Table[0] ← Arr1;
```

```
// This binary search will return the index of an element in its sorted array
for (i = 0; i < n; i++) {Table[1][BinarySearch (Arr[i], Arr1)] ← i;}

// After building the table, we can proceed to cost calculation
cost ← 0;
for (i = n - 1; i >= 0; i--) {if (Table[1][i] <> i) {cost += Table[0][i];
// Since we have swapped indices of the (n - i)^th largest element
temp ← Table[1][i]; Table[1][i] ← Table[1][BinarySearch (Table[0][i], Arr1)];
    Table[1][BinarySearch (Table[0][i], Arr1)] ← temp;}}
```

3. Similarly we can find the cost for the descending order. We will return the smaller of the two.

```
if (cost_ascending > cost_descending) {return cost_descending;}
else {return cost_ascending;}
```

Proof of Correctness:

Also, the cost of the swap is dependent on the largest element, we have to look at the position of the element in the sorted array and compare. Since, anyhow we have to swap an element to its correct location, we have to make at least one swap. Making more than 1 swap will lead to a wrong answer. The only way in which the cost is reduced is when an element is already in it's place or there are two elements who are in each other's spots. We have checked these conditions.

Question 4: Mandatory Batman Question

Part a:

We need to find the distance between the two given vertices s,t after the edge (u, v) is destroyed, for each edge (u, v) in the graph in $\mathcal{O}(n*(m+n))$, where n $\leftarrow |V|$ and m $\leftarrow |E|$.

Thinking Process:

To find the shortest distance between two points s and t in a graph, the best method is using BFS - breadth-first search. The running time of BFS is $\mathcal{O}(n+m)$ so we can apply BFS on the graph only kn number of times, where k is a constant. Applying BFS after removing each and every edge and calculating distances is a wrong approach as the time complexity will shoot up to $\mathcal{O}(m*(n+m))$.

Logic:

1. First we will apply BFS from s to t in the original graph and then find the shortest path by updating the BFS algorithm and storing it in a vector (in C++).
2. Now we will traverse all the edges and check whether the edge lies on the shortest path.
   (a) If the edge does not lie on the shortest path $\rightarrow$ the initial distance is the distance this time as well.
   (b) If the edge lies on the shortest path, we will have to reapply the BFS in the new graph to get our distance.
3. Finally we can store our distance in the matrix form as given in the question.

Principle for Optimization:

1. Being conservative about the number of times we are applying BFS and handling the cases when the shortest distance can not change anyhow separately.

Pseudocode:

1. Function to find the shortest path and shortest distance between s and t (updated version of the algorithm from the notes):

```
1  // adj is an array of pointers to linked lists containing neighbors of every
       vertex.
2  // visited stores whether a vertex is visited, dist stores the distance of
       every vertex from s - source
3  // predecessor stores the vertex from which a given vertex is traversed,
       parent in BFS tree.
4  // path array stores the shortest path from s to t.
5
6  queue q, visited[n], dist[n], predecessor[n], vector path;
7  Global Initial_Distance;
8  for (i ← 0; i < v; i++) {visited[i] ← false;
9          dist[i] ← INT_MAX; pred[i] ← -1;}
10
11 BFS(adj[], s, t, visited[], predecessor[], dist[]){
12      visited[s] ← true; dist[s] ← 0; q.push(s);
13      while (!q.empty()) {u ← q.pop();
14      for (neighbor ← adj[u] -> next -> vertex_number; neighbor <> nullptr;
       neighbor ← neighbor -> next) {
15          int vertex ← neighbor -> vertex_number;
```

```
        if (!visited[vertex]) { visited[vertex] ← true; dist[vertex] ← dist[u
    ] + 1; pred[vertex] ← u;
        q.push(vertex);}
        if (vertex = dest) { return dist[u]; }}}}

Initial_Distance ← BFS(adj[], s, t, visited[], predecessor[], dist[]);

ShortestPath(adj[], s, t, v)
{
    pred[v], dist[v], vector path; temp → t;
    path.push_back(temp);
    while (pred[temp] != -1) {
        path.push_front(pred[temp]); temp → pred[temp];}}

\\path vector contains the final shortest path from s to t
```

2. Final Code to fill the matrix given:

```
// Take out each edge from the shortest path and apply BFS to update the
    matrix
// remove is assumed as the function to remove an element from a Linked List
// Matrix be given matrix

temp_path
for (i = 0; i < path.size - 1; i++) {
adj[path[i]].remove(path[i+1]);     // remove edge
    Matrix[path[i+1]][path[i]] ←Matrix[path[i]][path[i+1]] ← BFS(adj, s, t,
    visited[], predecessor[], dist[]);
    adj[path[i]].insert(i, path[i+1]);}     // add edge in linked list

// Traverse the adjacency list
for (i = 0; i < n; i++) { for (j = adj[i]; j -> next <> nullptr; j ← j ->
    next) {
    if (Matrix[i][j -> vertex_number] != -1) {Matrix[j -> vertex_number][i] ←
     Matrix[i][j -> vertex_number] ← Initial_Distance;}}}

// Matrix is assumed to be initialized with -1
```

Part b: Complexity Analysis

Time Complexity Analysis:

Let s be the number of edges in the shortest path from s to t in the original graph. For Time Complexity we need to look at the algorithm step by step:

1. Initial BFS to calculate Initial_Distance and to find shortest distance $\implies \mathcal{O}(m+n)$ (from Notes)
2. Traversing the adjacency list
   (a) Edges that are on the shortest path require another BFS $\implies s * \mathcal{O}(m+n) \implies \mathcal{O}(s*(m+n))$
   (b) Edges that are not on the shortest path require directly substituting Initial_Distance $\implies (m-s) * \mathcal{O}(1) \implies \mathcal{O}(m-s)$

So, Overall Time Complexity is $\mathcal{O}(m+n+s*(m+n)+(m-s)) \equiv \mathcal{O}(s*(m+n)) \equiv \mathcal{O}(n*(m+n))$. (Since the vertices on the shortest path can not repeat, the maximum value of s is n)

Part c: Proof of Correctness

Proof of Correctness:

It is sufficient to show that for each edge removed, we are entering the correct shortest distance in the matrix. For a connected graph G, it is confirmed that the shortest path exists between any two vertices of the connected component. Any edge belonging to E, will either be a part of the shortest path or not be a part of the path. There are no other cases:

Case 1: The edge is not a part of the shortest path from s to t. In that case, the removal of the edge can only increase the shortest distance between any two vertices and never decrease. In that case, since we have the shortest path remaining intact, we can easily conclude that the shortest distance will remain the same as the original graph, which is correctly input as Initial_distance in the program.

Case 2: The edge is a part of the shortest path from s to t. Here, the shortest distance might increase. In this case, in the algorithm, we are applying BFS to find the shortest distance between s and t in the reduced graph. Since we know BFS correctly computes the shortest distance between any two points in the graph, we can claim that the value being entered in the matrix is correct.

The proof of correctness of BFS Traversal is given in the notes and hence is directly assumed.

Question 5: No Sugar in this Coat

Part a:

We need to find a vertex t such that for every vertex u ∈ V such that min(dist(u, s), dist(u, t)) ≤ k.

Thinking Process:

Since we have to comment on distances of various vertices from vertex s, it is kind of mandatory to have the use of BFS in the algorithm. Since, we need an algorithm in $\mathcal{O}(m+n)$ we can apply the BFS constant finite number of times only.

Logic:

1. First we will apply BFS from s in the original graph and find the largest distance of a vertex from s - call it l.
2. Now depending on the value of l - we can divide the problem into cases.
   (a) If l is less than 2k - we will report the element that is farthest from s. In case of multiple vertices - reporting any one of them will suffice.
   (b) If l is greater than 2k - we will report any element that is at a distance 2k from s. Any element from V_{2k} will suffice.

Principle for Optimization:

1. Exploiting the critical position of the elements belonging to the V_{2k}.
2. Using BFS only once to compute distances.

Pseudocode:

1. Pseudocode to fill the dist array to get all the elements segregated into sets corresponding to their distance from s:

```
BFS(adj[], s, visited[], dist[]){
    visited[s] ← true; dist[s] ← 0; q.push(s);
    while (!q.empty()) {u ← q.pop();
    for (neighbor ← adj[u] -> next -> vertex_number; neighbor <> nullptr;
    neighbor ← neighbor -> next) {
        int vertex ← neighbor -> vertex_number;
        if (!visited[vertex]) { visited[vertex] ← true; dist[vertex] ← dist[u
    ] + 1;
        q.push(vertex);}
        }return dist[vertex];}}

// We are assuming adjacency list representation
// We are returning the maximum distance from s along with updated dist array
```

2. Pseudocode to return the element t:

```
l ← BFS(adj[], s, visited[], dist[]);
if (l > 2k) { for ( i = 0; i < n; i++ ) {if (dist[i] = 2k) {return i;}}
else {for ( i = 0; i < n; i++ ) {if (dist[i] = l) {return i;}}
```

Part b: Proof of Correctness

Proof of Correctness:

It is sufficient to show that for each possible value of l - the distance of the farthest element from s - our output vertex t satisfies the property for all vertices in the graph.

Let us consider the cases as:

Case 1: l $\leq$2k $\implies$ Our output is vertex t belonging to V_l. For the vertices belonging to V_i for i $\leq$ k, the distance from s itself is less than k.
For the vertices in which i > k, we will consider the distance from t. For i $\neq$ l, we confirmed a path length less than k, as $\forall i \geq 0 : u \in V_i, v \in V_{i+1} \rightarrow (u, v) \in E$. Every vertex is connected to all the elements in the layer above it. We can take any path from t to a vertex u as long as we are making sure to go a layer above in the BFS tree every turn.
For i = l, we will go one layer above to V_{l-1} from t and then return to the vertex v in the layer containing t. We are confirmed to have a path of length 2. We can decrease it to 1 for special cases.

Case 2: l $\geq$ 2k $\rightarrow$ For the vertices which are less than a distance k + 1 is a trivial case.
For the vertices whose distance from s is between k + 1 and 3k - 1, we will consider the distance from t - element from V_{2k}. Same-level elements can be reached with a path length less than or equal to 2 using the method in Case 1. For the rest elements, if we take a path that penetrates through the layers - using the property that every element is connected to all the elements in the above or below layer in the BFS tree - we can reach the elements from V_k to V_{3k} with a pathlength of k.

Thus, all elements in the graph can be reached from either s or t with a distance of less than or equal to k.