ESO207A: Data Structures and Algorithms

Theoretical Assignment 2

Due Date: 7th October, 2023

Total Number of Pages: 10 Total Points 100

Instructions-

1. For submission typeset the solution to each problem and compile them in a single pdf file. Hand-written solutions will not be accepted. You can use LaTeX or Word for typesetting.
2. Start each problem from a new page. Write down your Name, Roll number and problem number clearly for each problem.
3. For each question, give the pseudo-code of the algorithm with a clear description of the algorithm. Unclear description will receive less marks. Less optimal solutions will receive only partial marks.
4. Assume that sorting would have $O(n\log(n))$ complexity.

Question 1. Search Complicated

You are given an array $A[0,\ldots,n-1]$ of n distinct integers. The array has following three properties:

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

(a) (10 points) 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 $\mathcal{O}(k\log(k)+q\log(n))$ time complexity algorithm to do the task.
(Higher time complexity correct algorithms will also receive partial credit)

Solution:

Algorithm 1: search_complicated(A, val, n, k)

Data:

- A: The input array of integers in which operations like sorting and binary searching are performed
- val: The value to be searched for
- n: Represents the total number of elements in the array A
- k: The number of elements in the last random segment of the array

Result:

- For each Query print True if val is present in the array, False otherwise

```
1  ReverseSort(A, n − k, n − 1);
2  L = 0;
3  R = n − k;
4  while True do
5      mid = (L+R)/2;
6      if A[mid] > A[mid − 1]  and  A[mid] > A[mid + 1] then
7          maxindex = mid;
8          break;
9      else if A[mid] > A[mid − 1] then
10         L = mid + 1;
11     else
12         R = mid − 1;
13 for i = 0 to q − 1 do
14     val = queries[i];
15     // - - - - - - - - - - - - - BinarySearch returns 1 if val found and 0 if not found - - - - -
           - - - - - - - -
16     search = BinarySearch(A, 0, maxindex, val);
17     if search == 1 then
18         print True;
19         continue;
20     // - - - - - - - - - - - ReverseBinarySearch is BinarySearch on reverse sorted array - -
           - - - - - - - - -
21     search = ReverseBinarySearch(A, maxindex + 1, n − 1, val);
22     if search == 1 then
23         print True;
24         continue;
25     print False;
```

(b) (5 points) Explain the correctness of your algorithm and give the complete time complexity analysis for your approach in part (a).

Solution: Proof of Correctness:

- To ensure the correctness of the algorithm, we need to make sure that it can find any element in the array A and correctly report the absence of any query not present in the array.
- The sorting operation only changes the order of elements in the array and does not affect their presence, therefore, it is sufficient to demonstrate that the algorithm correctly searches after sorting last k elements.
- Given that all elements from $A[n-k]$ to $A[n-1]$ are smaller than all elements from $A[0]$ to $A[n-k-1]$. Thus after reverse sorting step, Array will be increasing till maxima then decreasing.
- Therefore, if *val* is present from 0 to maxima (sorted) we find it in the BinarySearch, if *val* is present from maxima+1 to n-1 (reverse sorted) we find it in ReverseBinarySearch.
- If not found in both implies that it is not in 0 to n-1, i.e, it is absent, thus giving False as output.

Time Complexity Analysis:

- ReverseSort takes $\mathcal{O}(k \log k)$ time
- First while loop takes $\mathcal{O}(\log(n-k))$ time
- In each of q iterations of for loop, both BinarySearch and ReverseBinarySearch takes $\mathcal{O}(\log n)$ time
- Overall Time Complexity is thus $\mathcal{O}(k \log k + \log(n-k) + q \log n) = \mathcal{O}(k \log k + q \log n)$

Question 2. **Perfect Complete Graph**

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

- There is exactly one directed edge between every pair of distinct vertices.
- For any three vertices a, b, c, if (a, b) and (b, c) are directed edges, then (a, c) is present in the graph.

Note: **Outdegree** of a vertex v in a directed graph is the number of edges going out of v.

(a) (20 points) 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 $k \in \{0, 1, \ldots, n-1\}$, there exist a vertex v in the graph, such that **Outdegree**$(v) = k$.

Solution: a) Perfect Complete Graph ⇒ Between any pair of vertices, there is at most one edge

Proof: As per the definition there is exactly one edge between any two vertices,which automatically implies there is at most one edge between any two vertices.

b) Perfect Complete Graph ⇒ ∃ a vertex v in the graph s.t. Outdegree(v) = x ∀ x ∈ {0, 1, ..., n-1}

Proof:

- It can be seen that the range set of values of k {0, 1, ... n-1} contains n elements, and there are n nodes. Also the maximum outdegree possible is n-1 (when a vertex is connected to all nodes except itself).

- Thus if we can prove that the Outdegree of every vertex is different then we are done.
- **Base Case:** If n=1, the outdegree will be 0 as there is no edge which is same as set representing the values possible for k.
- Now, for n>1,
- **Claim:** Given any two vertex v1 and v2, their Outdegree is different.
- Proof by contradiction:

  If possible let Outdegree(v1) = Outdegree(v2)

  Also as there is exactly one edge between any two vertices, without loss of generality let there be an edge from v1 $\rightarrow$ v2, represented as (v1,v2)

  Now for every vertex v s.t. (v2,v) is an edge,

  1) It is counted in the outdegree(v2)

  2) As there are edges (v1,v2) and (v2,v), implies that there is an edge (v1,v)

  Therefore every vertex contributing to outdegree of v2 contributes to that of v1. Also the edge (v1,v2) contributes only to outdegree of v1 and not v2.

  $\Rightarrow$ Outdegree(v1) is at least Outdegree(v2) + 1. Therefore the assumption is wrong.

  Thus by contradiction, Claim is correct.
- Now as for any two vertices v1 and v2 in Graph, Outdegree is different, implies that Outdegree is different for each vertex in the graph and Outdegree(v) is upper bounded by n-1.
- This gives only one possible set of Outdegrees i.e., {0, 1, ..., n-1}

Now for the other direction, given that there exist vertex v s.t. Outdegree(v) = x $\forall$ x $\in$ {0, 1, ..., n-1}.
As there are n vertices and n realised Outdegrees $\Rightarrow$ each of n Outdegree values correspond to a different node and vice versa.
Thus Number of Edges = $\sum$ Outdegree(v) = 0 + 1 + ... + n-1 = n(n-1)/2
Given that between every pair of vertices there is at most one edge
$\Rightarrow$ Maximum possible number of edges = ${}^{n}C_2$ = n(n-1)/2 , which occurs when there is exactly one edge between any two vertices
As we have already seen that number of edges = n(n-1)/2, implies:
There is exactly one directed edge between any two vertices
Now the problem is reduced to prove the following,
Given: A directed graph with exactly one edge between any two vertices and $\exists$ a vertex v s.t. Outdegree(v) = x $\forall$ x $\in$ {0, 1, ..., n-1}
To Prove: For any three vertices a, b, c, if (a, b) and (b, c) are directed edges, then (a, c) is present in the graph.
Proof:

- **Claim:** Vertex with Outdegree k (say v_k) has outward edges to vertices with Outdegree < k, and, inward edges from vertices with Outdegree > k , $\forall$ k.
- Proof:

For every vertex outdegree + indegree = n-1 as there is exactly one edge to or from any other vertex.

As vertices can be one to one mapped with the outdegree, let, Outdegree(v_k) = k, i.e.,

Outdegree(v0) = 0, Outdegree(v1) = 1, ... Outdegree(v_{n-1}) = n-1

Base Case: v_0 ⇒ Outdegree(v_0) = 0 ⇒ Indegree(v_0) = n-1 ⇒ have inward edge from all vertices but itself ⇒ Has inward edges from vertices v_i with i>0. As there is no vertex v_i for i<0, thus v_0 satisfies the claim.

Induction:

Let the claim be true for v_i $\forall$ i $\leq$ k.

⇒ $\forall$ i $\leq$ k, v_i has inward edge from v_{k+1} as k+1 > i

⇒, v_{k+1} has outward edge to v_i, $\forall$ i $\leq$ k

⇒, v_{k+1} has outward edge to v_i, $\forall$ i < k+1

Note that these are k+1 edges that go outward v_{k+1}, and Outdegree(v_{k+1}) is also k+1.

⇒ Rest all edges connected to v_{k+1} are coming inward.

⇒, v_{k+1} has inward edge from v_i, $\forall$ i > k+1

⇒, v_{k+1} has outward edge to vertices with Outdegree < k+1, and, inward edges from vertices with Outdegree > k+1.

Thus claim true for i $\leq$ k implies claim true for i $\leq$ k+1, and Claim is true for k=0.

Therefore, the **Claim is always true**.

Let two known edges be (v_a,v_b) and (v_b,v_c),

⇒ b < a ; c < b ⇒ c < a

⇒ There is an edge (v_a,v_c).

(b) (10 points) Given the adjacency matrix of a directed graph, design an $\mathcal{O}(n^2)$ 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).

Solution:

Store the outdegree of each vertex in an array (say A) of length n.

This process involves iterating over all edges which is O(n^2)

For each value in {0, 1, ..., n-1} iterate over each index of A to find it. If any value is not found implies it is not a Perfect Complete Graph.

This process also takes O(n^2) time

If above for loop didn't stop implies that all of outdegrees {0, 1, ..., n-1} are present implying it is a Perfect Complete Graph.

Question 3. PnC (20 points)

You are given an array $A = [a_1, a_2, a_3, \ldots, a_n]$ 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 a permutation of an array of length n be defined as $A(\Pi)$, where $\Pi : [1, n] \to [1, n]$ is a bijective function.
We define the score of an array A of length n as

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

(a) (5 points) Explicitly characterise **all** the permutations $A(\Pi_0) = [a_{\Pi_0(1)}, a_{\Pi_0(2)}, a_{\Pi_0(3)} \ldots, a_{\Pi_0(n)}]$ of A such that

$$S(A(\Pi_0)) = \min_{\Pi} S(A(\Pi))$$

We call such permutations, a *"good permutation"*. In short, a *good permutation* of an array has minimum score over all possible permutations.

(b) (15 points) Provide an algorithm which computes the minimum cost required to transform the given array A into a *good permutation*, $A(\Pi_0)$.
The cost of a transformation is defined as the sum of costs of each individual operation used in the transformation.
You will only be awarded full marks if your algorithm works correctly in $\mathcal{O}(n \log n)$ in the worst case, otherwise you will only be awarded partial marks, if at all.

(c) (0 points) Bonus: Prove that your algorithm computes the minimum cost of converting any array A into a *good permutation*.

Some examples are given below for the sake of clarity:

- Regarding the operation:

Array	(i, j)	Cost
$A = [7, 2, 5, 4, 1]$	$(1, 3)$	$\max(a_1, a_3) = \max(7, 5) = 7$
$P_1 = [5, 2, 7, 4, 1]$	$(2, 5)$	$\max(2, 1) = 2$
$P_2 = [5, 1, 7, 4, 2]$	$(3, 5)$	$\max(7, 2) = 7$
Final Array $= [5, 1, 2, 4, 7]$	–	–

  In essence, the order of operations contributes significantly to the cost of a transformation.

- Regarding the cost of a transformation:
  The cost of transforming the array $A = [7, 2, 5, 4, 1]$ to $P_3 = [5, 1, 2, 4, 7]$ using the **exact** sequencce of operations mentioned above is $7 + 2 + 7 = 16$.

- Regarding permutations:
  Let Π be such that $[1, 2, 3, 4, 5] \rightarrowtail [5, 3, 1, 4, 2]$ and $A = [7, 2, 5, 4, 1]$, then $A(\Pi) = [5, 1, 2, 4, 7]$.

Solution:

(a) The permutation $A(\Pi_0)$ should be sorted.
Or: *good permutations* can only be ascending or descending.

(b) We need to find the minimum cost to make A either ascending (call the cost C_a) or descending (call the cost C_d). The answer shall be the minimum of these.

To convert an array A into a given permutation $A(\Pi)$, the algorithm is simple (greedy in nature):

1. Initialize the cost: $c \leftarrow 0$.
2. Pick the largest element that is not in the same place in both A and $A(\Pi)$. Call it A_l.
3. If there is no such element, return c.
4. Otherwise, let its index in $A(\Pi)$ be p.
5. Update the cost: $c \leftarrow c + \max(A_l, A_p)$.
6. Update the array A by swapping A_l and A_p. Return to Step 2.

There are many ways to implement this algorithm, for example using maps, using an array of pairs of the form (A_i, i), etc. Such solutions will be accepted if the data structure is used in lectures or is explained by the student in sufficient detail.

An implementation that only uses arrays is provided:

1. Initialize the array $S \leftarrow A$. Let G be another array of length n.
2. Sort S.
3. Iterate over A and let the current index be i in A. After the complete array has been iterated over, go to Step 5.
4. Binary search for $a_i = A[i]$ in S. Let this index be j. Update $G[j] \leftarrow i$.
5. Initialize $c \leftarrow 0$.
6. Iterate over S in descending order (depending on how S was sorted initially). Let the current index in S be j. If the complete array has been iterated over, return c.
7. If $S[j] = j$, continue to the next iteration.
8. If $S[j] \neq j$, make the following updates:

$$c \leftarrow c + \max(A[S[j]], A[j]) \qquad \text{swap}(A[j], A[S[j]]) \qquad \text{swap}(S[j], S[S[j]])$$

The above algorithm is run on both ascending S and descending S once, and the minimum of their answers is reported. The exact pseudo-code of the above is left as an exercise.

(c) Lemma 1: It is suboptimal to move the maximum element of the array more than once.
Corollary: The maximum element is moved at most once.
Lemma 2: The maximum element is swapped atleast once iff it is not in the correct order initially.
Lemma 3: Let a_j be the maximum element in the current array, whose position does not match with its position in P. Then, it is never suboptimal to move a_j to its correct place in A before moving any other element.
The proofs of the above are left as an exercise.

As a hint, try to prove them using induction. You only need to consider three indices in your inductive steps.

Question 4. Mandatory Batman Question (20 points)

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

Some constraints:

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

(a) (12 points) Batman expects an algorithm that works in $\mathcal{O}(|V| \cdot (|V| + |E|) = \mathcal{O}(n \cdot (n + m))$.

(b) (4 points) 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.

(c) (4 points) Lastly, you also need to provide proof of correctness for your algorithm.

Solution:

(a) The algorithm is as follows:

1. Execute a BFS from s.
2. Let the (shortest) path from s to t be P, and its length be denoted $|P|$. It can be found using the BFS done above.
3. Iterate over the matrix, let the first vertex be u, and the second vertex be v.
4. If neither (u, v), nor (v, u) is present in P, then update $M[u, v]$ and $M[v, u]$ to $|P|$.
5. If (u, v) or (v, u) is present in P, then run another BFS from s in the graph $G' = (V, E \backslash \{(u, v)\}$. Let the shortest path from s to t in G' be P'. Update $M[u, v]$ and $M[v, u]$ to $|P'|$.

The only non-trivial components of the algorithm are finding a path and checking if an edge is in a path. Some ideas for implementing the same are provided below:

- Finding a path from s to t: An array of length n, called *prev*, can be used such that $prev[u]$ stores the vertex due to which u entered the BFS queue.
  $prev[s]$ can be put s.
  Now, to find a path from s to t, we can initialize u with t, and keep updating u as $u \leftarrow prev[u]$ until $u = s$. This helps us find a path P from t to s (and consequently from s to t since G is undirected). As P is found using BFS, it is of shortest length.
  Finding a path using this method takes $\mathcal{O}(n)$, which is permissible since we only need to find one such path, and since the array *prev* is accessed at most $\mathcal{O}(n + m)$ times during the BFS (depending on implementation).
- Checking if an edge is present in P: A boolean matrix $B[n, n]$ can be used to store if an edge is used in the path or not. Initially, each entry in B can be initialized to *False*.
  Now, while finding the path P, while updating u, we can update $B[u, prev[u]] \leftarrow True$, and $B[prev[u], u] \leftarrow True$.
  This step takes $\mathcal{O}(n^2)$ in total, which is permissible given the expected runtime.

Pseudo-code for the above is left as an exercise.

(b) The maximum number of vertices in P can be n, therefore $|P| < n$, i.e., there can be atmost $n-1$ edges for which another BFS is run. A BFS is also run initially for finding a path P.

As the runtime of one BFS is $\mathcal{O}(|V|+|E|)$, the runtime of the provided algorithm is:

$$\mathcal{O}(n+m) + \mathcal{O}((n-1)\cdot(n+(m-1))) = \mathcal{O}(n\cdot(n+m))$$

(c) If an edge is included in the path P, then the provided algorithm simulates exactly what happens, destroys the edge, and runs a BFS to find the shortest path. The matrix entry is trivially true in this case.
If however, an edge is not included in the path P, then even after deleting the edge, we have the path P in the graph G'. Since the deletion of an edge only increases shortest path length, we have that $|P|$ is "no longer than the shortest path in G'", i.e, it is the shortest path in G' as well.

Question 5. No Sugar in this Coat (15 points)

You are given an **undirected**, **unweighted** and **connected** graph $G = (V, E)$, and a vertex $s \in V$, with $|V| = n$, $|E| = m$ and $n = 3k$ for some integer k. 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

$$\forall\ i \geq 0: \quad u \in V_i, v \in V_{i+1} \Rightarrow (u, v) \in E$$

Provide the following:

(a) (10 points) An $\mathcal{O}(|V|+|E|)$ time algorithm to find a vertex $t \in V$, such that the following property holds for every vertex $u \in V$:

$$\min(dist(u, s), dist(u, t)) \leq k$$

Note that your algorithm can report s as an answer if it satisfies the statement above.

(b) (5 points) Proof of correctness for your algorithm.

Solution:

(a) A description of the algorithm has been provided below:

1. Execute a BFS from s, and for each vertex, store the length of minimum path from s to that vertex.
2. Let the maximum distance from s to any vertex be d.
3. Find any vertex at a distance of $\min(d, 2k)$ from s and return it.

(b) Lemma: Distance from any vertex in V_i to any vertex in V_j is exactly equal to $|j-i|$ for all $i \neq j$. If $i = j$, then this distance is less than or equal to 2.
Try proving the above using the given property of G and induction.
Let $d \geq 2k$, and u be a vertex in the set V_i. Since, $t \in V_{2k}$ and $s \in V_0$, we have:

$$\begin{aligned}\min(dist(u, s), dist(u,\ t)) &\leq \min(|i-0|, |i-2k|) \\ &\leq \min(i, |i-2k|) \\ &\leq k \qquad \forall\ 0 \leq i \leq 3k\end{aligned}$$

The rest of the proof has been left as an exercise.