Course ESO207: Data Structures and Algorithms
Attempt by Dhruv - 210338
Date October 2023

Problem Statement 1 - Search Complicated

You are given an array A[0, . . . , 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 $\boldsymbol{\mathcal{O}(klog(k) + qlog(n))}$ time complexity algorithm to do the task.

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

Solution

Algorithm Analysis: done at the end with proof of correctness.
(**Note:** Assuming $n - k >= 3, k >= 1$)
Pseudo Code:

Algorithm 1: *PeakSearch(A, n, k)*

Data: Array A contains n elements where they increase till some index (say idx) then decrease till index $n - k$
Result: Find peak value index (idx) of first $n - k$ elements in A

```
lo = 1;
hi = n - k;
while lo <= hi do
    mid = (lo + hi) / 2;
    if (A[mid] > A[mid + 1]) & (A[mid] > A[mid - 1]) then
        return mid;
    if A[mid] < A[mid + 1] then
        lo = mid;
    else
        hi = mid + 1;
return n - k;
```

Algorithm 2: *LastKSearch(A, n, k, val)*

Data: Array A contains n elements with last k elements sorted in increasing order
Result: Returns if val is present in A

```
lo = n - k + 1;
hi = n;
while lo <= hi do
    mid = (lo + hi) / 2;
    if A[mid] == val then
        return True;
    if A[mid] < $x$ then
        lo = mid + 1;
    else
        hi = mid;
return False;
```

Algorithm 3: *FirstNKSearch(A, n, k, idx, val)*

```
Data: Array $A$ contains $n$ elements where they increase till index $idx$ then decrease till index $n - k$
Result: Returns if $val$ is present in $A$
lo = 1;
hi = idx;
while lo <= hi do
    mid = (lo + hi) / 2;
    if A[mid] == val then
        return True;
    if A[mid] < val then
        lo = mid + 1;
    else
        hi = mid;
lo = idx;
hi = n - k;
while lo <= hi do
    mid = (lo + hi) / 2;
    if A[mid] == val then
        return True;
    if A[mid] > val then
        lo = mid + 1;
    else
        hi = mid;
return False;
```

Algorithm 4: *FindVal(A, n, k, q)*

```
Data: Array $A$ contains $n$ elements with last $k$ elements randomly arranged. $q$ are number of queries
Result: Returns if $val$ is present in $A$
sort(A, n - k + 1, n);
idx = PeakSearch(A, n, k);
for i: 1 → q do
    val = query[i];
    if val < min(A[1], A[n-k]) then
        if LastKSearch(A, n, k, val) == True then
            print("YES");
        else
            print"NO";
    else
        if FirstNKSearch(A, n, k, idx, val) == True then
            print("YES");
        else
            print"NO";
```

Part B:
Time Complexity Analysis:
Algorithm 1: Binary search (discussed in Algorithm Analysis) is applied on first $n-k$ elements of array A

$$\Rightarrow \mathcal{O}(log(n-k)) \Rightarrow \boldsymbol{\mathcal{O}(log(n))}$$

Algorithm 2: Binary search (discussed in class) is applied on last k sorted elements of array A

$$\Rightarrow \mathcal{O}(log(k)) \Rightarrow \boldsymbol{\mathcal{O}(log(n))}$$

Algorithm 3: Binary search is applied two times in two sorted search space i.e $[A[1], A[idx]]\&[A[idx], A[n-k]]$

$$\Rightarrow \mathcal{O}(log(idx)) + \mathcal{O}(log(n-k-idx+1) \Rightarrow \mathcal{O}(log(n-k)) \Rightarrow \boldsymbol{\mathcal{O}(log(n))} \textit{(since idx < n - k < n)}$$

Algorithm 4:

- *sort(last k elements)* $\Rightarrow \boldsymbol{\mathcal{O}(klog(k))}$
- *Algorithm 1* for finding $idx \Rightarrow \boldsymbol{\mathcal{O}(log(n))}$
- Loop over q queries of type *val.* Each query calls either *Algorithm 2 or Algorithm 3.* $\Rightarrow \boldsymbol{\mathcal{O}(qlog(n))}$

$$\Rightarrow \mathcal{O}(klog(k)) + \mathcal{O}(log(n)) + \mathcal{O}(qlog(n)) \Rightarrow \boldsymbol{\mathcal{O}(klog(k) + qlog(n))}$$

$$\boldsymbol{Total\ Time\ Complexity} = \mathcal{O}(klog(k) + qlog(n)) + 3\mathcal{O}(log(n))) = \boldsymbol{\mathcal{O}(klog(k) + qlog(n))}$$

Proof Of Correctness & Algorithm Analysis
Assertion1: *Algorithm 1 - Peak Search* finds the idx such that $A[0] < A[1] < \ldots < A[idx] > A[idx+1] > \ldots A[n-k]$
Proof: idx always remains in between lo and hi i.e $lo < idx < hi$. This can be proved by induction.

- *Base Case:* $lo = 1, hi = n-k$ with $lo < idx < hi$
- *Assume for some j:* for some $lo_j < idx < hi_j$
- Proof for $j+1: mid = (lo_j + hi_j)/2$, if $A[mid] < A[mid+1] \Rightarrow idx >= (mid+1)$ since $lo_{j+1} = mid, hi_{j+1} = hi_j \Rightarrow lo_{j+1} < idx < hi_{j+1}$. Else if $A[mid] > A[mid+1] \Rightarrow idx <= mid$ since $h_{j+1} = mid+1, lo_{j+1} = lo_j \Rightarrow lo_{j+1} < idx < hi_{j+1}$.

Since search space ($[lo, hi]$) decreases in every iteration we eventually end up at required idx

Assertion2: *Algorithm 3 - FirstNKSearch* search for val in first $n-k$ elements
Proof: Since array A is sorted upto idx in ascending order and sorted in descending order in range $[idx, n-k]$ (using Assertion 1) binary search can be applied to both the sections for searching *val.* Proof of correctness for *binary search* is discussed in lectures.

Assertion3: *Algorithm 4 - FindVal* search for val in whole array A
Proof: There are two exhaustive cases where val can exist:

- $val < min(A[0], A[n-k])$ in which case val lies in last k elements and can be found using binary search
- $val >= min(A[0], A[n-k])$ in which case we apply *Algorithm 3* (Assertion 2)

Problem Statement 2 - Perfect Complete Graph

A directed graph with n vertices is called Perfect Complete Graph if:
Property 1. There is exactly one directed edge between every pair of distinct vertices.
Property 2. For any three vertices a, b, c, if (a, b) and (b, c) are directed edges, then (a, c) is present in the graph.
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 anypair of vertices, there is at most one edge, and for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph, such that $\boldsymbol{Outdegree(v)} = k$.

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

Reinterpretation

Part A.
Statement 1 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a Perfect Complete Graph
Statement 2 Between any pair of vertices, there is at most one edge, and for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph, such that $Outdegree(v) = k$.
Prove:

$$\textbf{Statement 1} \iff \textbf{Statement 2}$$

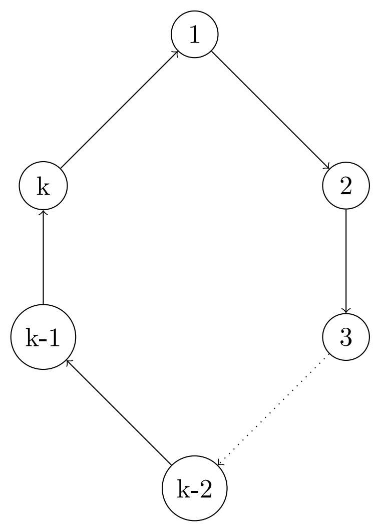
Note:

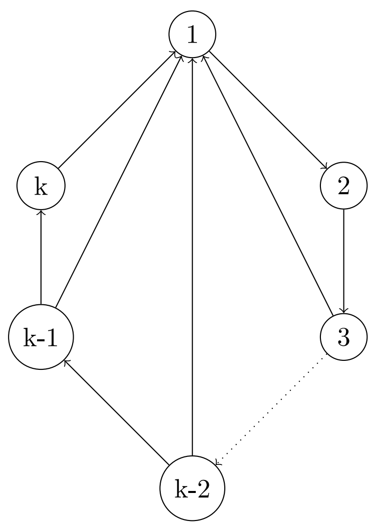
- Edge (u, v) means an directed edge from u to v
- There are no self loops i.e no edge of form (u, u)

Solution

Claim 1. *A Perfect Complete Graph do not contain a cycle.*

Proof. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a Perfect Complete Graph with a cycle of length k.

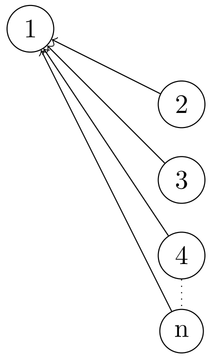




Using Property one and two of Perfect complete graph there should be an edge $(k-1,1)$, $(k-2,1)$ … $(3,1)$. But an edge $(3,1)$ violates the property two of Perfect Complete Graph for nodes $1,2,3$. Hence $\mathcal{G}$ cannot be a Perfect Complete Graph which is a **Contradiction**. Hence $\mathcal{G}$ contains no cycle. □

Claim 2. *A Perfect Complete Graph contains exactly one node with Outdegree zero*

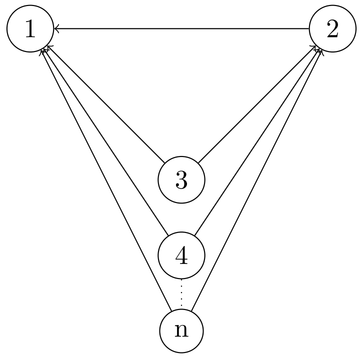
Proof. Let $\forall$ node $v \in \mathcal{G}$, $Outdegree(v) > 0$. Start from any node i, $\exists$ a node $j \in \mathcal{G}$ with an edge (i,j) since $Outdegree(i) > 0$. Keep on repeating and since we have finite nodes we will end upon one of visited nodes. This is a Contradiction to **Claim 1**. Hence $\exists$ a node $v \in \mathcal{G}$ with $Outdegree(v) = 0$.
To satisfy first property of perfect complete graph there can't be more than one node with $outdegree = 0$. Hence there is exactly one node (say 1) with $outdegree(1) = 0$. Using claim 2 and property 1 we get fig 3.



□

Claim 3. *A Perfect Complete Graph contains exactly one node with Outdegree one*

Proof. Let $\forall$ node $v \in \mathcal{G} \setminus 1$, $Outdegree(v) > 1$. Start from any node i other than 1, $\exists$ a node $j \in \mathcal{G} \setminus 1$ with an edge (i,j) since $Outdegree(i) > 1$. Keep on repeating and since we have finite nodes we will end upon one of visited nodes other than 1. This is a Contradiction to **Claim 1**. Hence $\exists$ a node $v \in \mathcal{G}$ with $Outdegree(v) = 1$.
Let consider two nodes $i, j \in \mathcal{G} \setminus 1$ such that $Outdegree(i) = Outdegree(j) = 1$. From fig 3. $\exists$ edges $(i,1)$ and $(j,1)$. Therefore there exist no edge between i, j which is a contradiction to property 1. Hence there exist exactly one node (say 2) with $Outdegree(2) = 1$. Using claim 2, claim 3, and property 1 we get fig 4.



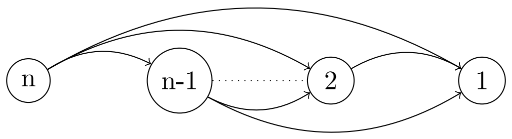
□

Expanding the proof of claim 2 and claim 3 for nodes (say) $3, 4, 5, 6 \ldots$ we conclude that in a Perfect Complete Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ for all $k \in \{0, 1, 2, \ldots\}$, there exist a vertex v in $\mathcal{G}$ such that $Outdegree(v) = k$.
From property 1 we can say "Between any vertices, there is at most one edge" since there is exactly 1. Hence

Statement 1 $\Rightarrow$ Statement 2

Claim 4. *Statement 2 ⇒ Statement1*

Proof. $\mathcal{WLOG}$, let $\forall$ node $i : 1 \to n$ $Outdegree(i) = i - 1$. Since there is atmost one edge between any pair of vertices there cannot be an edge $(n-1, n)$. Expanding on the same argument for a node i there cannot be any edge $(i, n), (i, n-1), \dots (i, i+1)$ and since $Outdegree(i) = i - 1$ this gives us edges $(i, i-1), (i, i-2), \dots (i, 1)$. Hence there is exactly one edge between every pair of nodes. This is shown in figure 4.



Take any three nodes (say) $a < b < c$ then we have edges $(a, b), (b, c), (a, c)$ which equals property 2.
$\mathcal{G}$ satisfies property 1 and 2 therefore $\mathcal{G}$ is a Perfect Complete Graph.

□

Statement 1 $\iff$ Statement 2

Part B.

Algorithm Analysis:

We use the conditions given in statement 2 to check for perfect complete. One is to check if a node have more than one edge with other nodes and second to store outdegree for each node by iterating over Adjacency matrix. Finally a check if array outdegree contains each number from the set $\{0, 1, 2 \dots n-1\}$ or not.

Pseudo Code:

Algorithm 1: *PerfectCompleteGraph(AdjMat, n)*

```
Data: AdjMat represents the adjacency matrix of graph under test, n is total number of nodes
Result: True if graph is Perfect Complete else False
Outdegree(n, 0);
bool flag = true;
for i: 1 → n do
    for j: 1 → n do
        if $AdjMat[i][j]$ & $AdjMat[j][i]$ then
            flag = false;
        if $AdjMat[i][j]$ then
            $Outdegree[i] += 1$;
for k: 0 → n-1 do
    bool check = false;
    for i: 1 → n do
        if $Outdegree[i] == k$ then
            check = true;
    if !check then
        flag = false;
return flag;
```

Time Complexity Analysis:

Initialize: Outdegree array initialization $\Rightarrow \mathcal{O}(n)$
For Loop: 2 nested for loops $\Rightarrow \mathcal{O}(n^2)$

Time Complexity $\Rightarrow \mathcal{O}(n^2)$

Problem Statement 3 - PnC

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 tof $max(a_i, a_j)$. You are allowed to use this operation any number of times. Let $\prod$ be a permutation of 1, 2, . . . , n. For an array A of length n, let $A(\prod)$ be the permuted array $A(\prod) = [a_{\prod(1)}, a_{\prod(2)}, ..., a_{\prod(n)}]$
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 $A(\prod_0) = [a_{\prod_0(1)}, a_{\prod_0(2)}, a_{\prod_0(3)} \dots, a_{\prod_0(n)}]$ of A such that

$$S(A(\prod_0)) = \min_{\prod} S(A(\prod))$$

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, $A(\prod_0)$. 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 $\mathcal{O}(nlogn)$ in the worst case, otherwise you will only be awarded partial marks, if at all.

Solution

Claim 1. *A good permutation is one which arranges A in (always) increasing or decreasing order*

Proof. It suffices to prove that of all the permutation of elements of A, $S(A)$ minimizes when A is always increasing or decreasing.
Consider a array $A = [a, b, c]$ such that $a < b < c$. Six permutation are possible for this array. Lets calculate $S(A)$ for all the cases.

1. $A = [a, b, c] \Rightarrow sum_1 = b - a + c - a = c - a$
2. $A = [a, c, b] \Rightarrow sum_2 = c - a + c - b = 2c - a - b$
3. $A = [b, a, c] \Rightarrow sum_3 = b - a + c - a = b + c - 2a$
4. $A = [b, c, a] \Rightarrow sum_4 = c - b + c - a = 2c - a - b$
5. $A = [c, a, b] \Rightarrow sum_5 = c - a + b - a = b + c - 2a$
6. $A = [c, b, a] \Rightarrow sum_6 = c - b + b - a = c - a$

It is easy to see that $S(A)$ is minimum for $A = [a, b, c]$ or $A = [c, b, a]$. This makes sense since for $S(A)$ to be minimum we should keep b in middle of a and c to avoid adding extra length.
Extending the above idea for n element array A, Consider three continuous elements and rearrange them such that they are in increasing (or decreasing order). Keep doing for any three indices you find in different order. Finally we get array $A = [a_1, a_2, \dots, a_n]$ such that $a_1 < a_2 < \dots < a_n$ with $S(A) = a_n - a_1$ (or $a_1 > a_2 > \dots > a_n$ with $S(A) = a_1 - a_n$) □

Part A: Given an array $A = [a_1, a_2, \dots, a_n]$ and an index array $idx = [1, 2, \dots, n]$. Sort A (increasing and decreasing) using *merge sort (Algorithm 1)* such that $idx[i]$ stores the index of $A[i]'s$ new position. This results in $\prod_0 = idx$ such that

$$S(A(\prod_0)) = \min_{\prod} S(A(\prod))$$

We get two permutation for A which minimizes $S(A(\prod))$

Part B:
Algorithm Analysis:
$idx[i]$ stores the position ($1 <= idx[i] <= n$) where $A[i]$ is required to be present thus forming a graph with simple cycles. We iterate over elements of cycle using *bfs* and the cost to do correct position of all its element is sum of all its elements excluding minimum element in the cycle. Add up the minimum cost for each cycle in the array to get final answer.
(Note: We are required to do the above for two types (increasing and decreasing) of *idx*)

Pseudo Code:

Algorithm 1: *mergeSort(A, idx, lo, hi)*

```
Data: A is array containing n distinct positive elements, idx is initial permutation
Result: Returns sorted array A with correspoding index (idx) intact
if lo >= hi then
    return A, idx;
mid = (lo + hi) / 2;
mergeSort(A, idx, lo, mid);
mergeSort(A, idx, mid + 1, hi);
temp(hi - lo + 1);
i = lo;
j = mid + 1;
k = 1;
while i <= mid & j <= hi do
    if A[i] <= A[j] then
        temp[k] = A[i];
        idx[i] = lo + k - 1;
        i++;
    else
        temp[k] = A[j];
        idx[j] = lo + k - 1;
        j++;
    k++;
while i <= mid do
    temp[k] = A[i];
    idx[i] = lo + k - 1;
    i++;
    k++;
while j <= hi do
    temp[k] = A[j];
    idx[j] = lo + k - 1;
    j++;
    k++;
for i : lo → hi do
    A[i] = temp[i - lo + 1];
return A, idx;
```

Algorithm 2: *GoodPermutation(A, n)*

```
Data: A is array containing n distinct positive elements
Result: Returns minimum cost to transform A to good permuatation
temp(n);
for i: 1 → n do
    temp[i] = A[i];
idx(n);
for i: 1 → n do
    idx[i] = i;
mergeSort(temp, idx, 1, n);
cost1 = 0;
vis(n, False);
for i: 1 → n do
    if vis[i] == True then
        continue;
    prev = i;
    mn = A[i];
    while vis[i] == False do
        cost1 = cost1 + A[i];
        mn = min(A[i], mn);
        vis[i] = True;
        i = idx[i];
    cost1 = cost1 - mn;
    i = prev;
cost2 = 0; vis(n, False); reverse(idx);
/*
Same code of for loop with only idx reversed and cost2;
*/
return min(cost1, cost2);
```

Time Complexity Analysis:
mergeSort(temp, idx): (discussed in lectures) $\Rightarrow \boldsymbol{\mathcal{O}(nlogn)}$
For & While loop: Each index i enters while loops once i.e each index gets visited once. Therefore total number of times **while** loops runs over all the **for** loops is n. $\Rightarrow \boldsymbol{\mathcal{O}(n)}$

$$TimeComplexity = \mathcal{O}(nlogn) + \mathcal{O}(n) = \boldsymbol{\mathcal{O}(nlogn)}$$

Problem Statement 4 - Mandatory Batman Question

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

(b) (4 points) He also wants you to provide him with proof of runtime of your algorithm, i.e., a TimeComplexity Analysis of the algorithm you provide.

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

Solution

Part A:

Algorithm Analysis: is discuussed in Part C

Pseudo Code:

Algorithm 1: *ShortestPath(AdjList, s, t, n)*

```
Data: AdjList is adjacency list representation of graph of n nodes. s, r are start and end nodes repectively
Result: Returns M_{nxn} matrix
par(n, -1);
queue q;
q.push(s);
par[s] = 0;
while !q.empty() do
    currNode = q.top();
    q.pop();
    for auto child : AdjList[currNode] do
        if par[child] == -1 then
            par[child] = currNode;
            q.push(child);
path;
currNode = t;
while currNode <> 0 do
    path.push(currNode);
    currNode = par[currNode];
return path;
```

Algorithm 2: *ShortestDistance(AdjList, s, t, u, v)*

```
Data: AdjList is adjacency list representation of graph of $n$ nodes. $s, r$ are start and end nodes repectively.
      Edge $(u, v)$ is deleted
Result: Returns shortest distance between $s$ and $t$
dis(n, -1);
queue q;
q.push(s);
dis[s] = 0;
while !q.empty() do
    currNode = q.top();
    q.pop();
    for auto child : AdjList[currNode] do
        if (child == u & currNode == v) || (child == v & currNode == u) then
            continue;
        if dis[child] == -1 then
            q.push(child);
            dis[child] = dis[currNode] + 1;
return dis[t];
```

Algorithm 3: *MinDeletion(AdjList, s, t, n)*

```
Data: AdjList is adjacency list representation of graph of $n$ nodes. $s, r$ are start and end nodes repectively
Result: Returns $M_{nxn}$ matrix
path = ShortestPath(AdjList, s, t, n);
M(n, n);
for i: 1 → n do
    for j: 1 → n do
        M[i][j] = path.size() - 1;
for u: 1 → n do
    for auto v : AdjList[u] do
        flag = True;
        for i: 1 → path.size() - 1 do
            if (u == path[i] & v == path[i + 1]) || (v == path[i] & u == path[i + 1]) then
                flag = False;
                break;
        if flag == False then
            M[u][v] = ShortestDistance(AdjList, s, t, u, v);
return M;
```

Part B:

Time Complexity Analysis:
Algorithm1 & Algorithm 2: are varations of ***bfs*** as taught in class ⇒ $\boldsymbol{\mathcal{O}(n+m)}$
Algorithm 3:

- *path* calculation ⇒ $\boldsymbol{\mathcal{O}(n+m)}$
- M_{nxn} initialization ⇒ $\boldsymbol{\mathcal{O}(n^2)}$

- Nested ***for(u){for(v)}*** iterates over *AdjList* i.e number of edges m. Each iteration runs a *for(n)* and out of m iterations $2(path.size()-1)$ iteration calls $Algorithm2$. (**Note:** $path.size() <= n$)
  $\Rightarrow \boldsymbol{\mathcal{O}(mn + (m+n) * 2(path.size()-1))} \Rightarrow \boldsymbol{\mathcal{O}(mn + (m+n) * 2(n-1))} \Rightarrow \boldsymbol{\mathcal{O}(n(m+n))}$

$$\boldsymbol{TimeComplexity} = \mathcal{O}(n(n+m)) + \mathcal{O}(n+m) + \mathcal{O}(n^2) = \boldsymbol{\mathcal{O}(n(n+m))}$$

Part C:

Proof Of Correctness & Algorithm Analysis

Assertion1: Algorithm 2 finds the minimum distance over **all the possible paths** between s, t in a undirected, unweighted, connected graph $\mathcal{G}$ without including (u, v)
Proof: This algorithm is just a simple *bfs* on graph $\mathcal{G}_{\prime} = (\mathcal{V}, \mathcal{E}_{\prime})$ where $\mathcal{E}_{\prime} = \mathcal{E} \setminus (u, v)$. Proof of correctness is same as discussed in lectures.

Assertion2: Algorithm 3 correctly fill M_{nxn} where $M[u][v]$ stores minimum distance over all the possible path between s, t when (u, v) is deleted.
Proof: Algorithm 3 calculates a minimum distance path (say *path*) between s, t when no edge is deleted (using Assertion 1). When (u, v) is deleted there are two possible exhaustive cases

- (u, v) doesn't lie on *path* in which case we don't need to go over all the possible minimum distance paths between s, t since we know one i.e *path*.
- (u, v) lie on *path* in which case we need to find new minimum distance path using algorithm 2 (Assertion 1)

Problem Statement 5 - No Sugar in this Coat

You are given an undirected, unweighted and connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and a vertex $s \in \mathcal{V}$, with $|\mathcal{V}| = n$, $|\mathcal{E}| = m$ 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 $\mathcal{V}_d \subset \mathcal{V}$ be the set of vertices that are at a distance equal to d from s in $\mathcal{G}$, then

$$\forall\, i >= 0 : u \in \mathcal{V}_i, v \in \mathcal{V}_{i+1} \Rightarrow (u, v) \in \mathcal{E}$$

Provide the following:

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

$$min(dist(u, s), dist(u, t)) <= k$$

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

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

Solution

Algorithm Analysis: is discussed at the end along with the proof of correctness.

Pseudo Code:

Algorithm 1: *ShortestDistance(AdjList, s, t, u, v)*

```
Data: AdjList is adjacency list representation of graph of n nodes. s, r are start and end nodes repectively.
      Edge (u, v) is deleted
Result: Returns shortest distance between s and t
dis(n, -1);
height = 0;
queue q;
q.push(s);
dis[s] = 0;
while !q.empty() do
    currNode = q.top();
    height = max(height, dis[currNode]);
    q.pop();
    for auto child : AdjList[currNode] do
        if dis[child] == -1 then
            q.push(child);
            dis[child] = dis[currNode] + 1;
if height <= k then
    return s;
reqHeight = height - k;
for i: 1 → n do
    if dis[i] == reqHeight then
        return i;
return -1;
```

Time Complexity Analysis:

- *bfs* (while loop) for calculating distance of all nodes from $s \Rightarrow \mathcal{O}(n+m)$ (discussed in class)
- *for* loop to find the *reqHeight* $\Rightarrow \mathcal{O}(n)$

$$\text{Total Time Complexity} = \boldsymbol{\mathcal{O}(n+m) = \mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)}$$

Part B:
Proof of Correctness & Algorithm Analysis:
Applying bfs on $\mathcal{G}$ with root s we get a bfs tree. Further discussion will on the bfs tree (say $\mathcal{T}$).

- Since $n = 3k \Rightarrow height(\mathcal{T}) <= 3k - 1$
- From property of given graph any two nodes, one from i^{th} layer and other from $(i+1)^{th}$ layer have a edge between them. Hence any two nodes on same level have equal distacne with any node from any other level.

Required node t can be found by below two cases based on $height(\mathcal{T})$ (say h)

- $h <= k$: in which case $t = s$ satisfy required conditions.
- $k < h < 3k$: in which case any $t \in \mathcal{V}_{h-k}$ satisfy the conditions because
  - for any node $u \in \mathcal{V}_{h >= d > h-k}$, $dis(u,t) <= k$
  - for any node $u \in \mathcal{V}_{h-k > d > k}$, $dis(u,t) <= k$
  - for any node $u \in \mathcal{V}_{k >= d >= 0}$, $dis(u,s) <= k$
  - for any node $u \in \mathcal{V}_{d=h-k}$, $dis(u,t) <= 2 <= k$ $(since\ k = 1 \Rightarrow h = 1 \Rightarrow h <= k\,(case1))$ (Using Property)
  - for any node $u \in \mathcal{V}$, $min(dis(u,t), dis(u,s)) <= k$