

PROGRAM 4

Write program to obtain the Topological ordering of vertices in a given digraph.

ALGORITHM

→ Function Topological sort (Graph) :

Input: Graph with v vertices & adjacency list $Adj[v]$

Output: List representing the topological order or a message if a cycle is detected.

1. Initialize indegree $(0 \dots v-1)$ to 0
2. For each vertex u in Graph :
For each neighbour v in $adj[u]$:
 $indeg[v] = indeg[v] + 1$
3. Initialize an empty queue Q
4. For each vertex i from 0 to $v-1$:
If $indeg[i] == 0$:
Enqueue (Q, i)
5. Initialize count = 0
6. Initialize empty list TopOrder
7. While Q is not empty :
 $u = Dequeue(Q)$
Append u to TopOrder.
count = count + 1
For each neighbour v in $Adj[u]$:
 $indeg[v] = indeg[v] - 1$

If $indeg[v] == 0$:
Enqueue (Q, v)

8. If count $\neq v$:
Output "Cycle detected: Topological sort not possible"

Else :
Output TopOrder.

CODE

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX 100
```

```
int adj[MAX][MAX]; // adjacency matrix
```

```
int indegree[MAX]; // array to store indegree of vertices
```

```
int queue[MAX]; // queue for BFS
```

```
int front = -1, rear = -1;
```

```
// Add edge from u -> v
```

```
void addEdge(int u, int v) {
```

```
    adj[u][v] = 1;
```

```
    indegree[v]++;
```

```
}
```

```
// Enqueue
```

```
void enqueue(int value) {
```

```
    if (rear == MAX - 1)
```

```
        return;
```

```
    if (front == -1)
```

```
        front = 0;
```

```
    queue[++rear] = value;
```

```
}
```

```
// Dequeue
```

```
int dequeue() {
```

```
    if (front == -1 || front > rear)
```

```
        return -1;
```

```

    return queue[front++];
}

// Topological Sort using Kahn's Algorithm
void topologicalSort(int n) {
    // Enqueue vertices with 0 indegree
    for (int i = 0; i < n; i++)
        if (indegree[i] == 0)
            enqueue(i);

    int count = 0;
    int topoOrder[MAX];

    while (front <= rear) {
        int u = dequeue();
        topoOrder[count++] = u;

        for (int v = 0; v < n; v++) {
            if (adj[u][v]) {
                indegree[v]--;
                if (indegree[v] == 0)
                    enqueue(v);
            }
        }
    }

    if (count != n) {
        printf("Cycle detected! Topological sort not possible.\n");
    } else {
        printf("Topological Order: ");
    }
}

```

```

        for (int i = 0; i < count; i++)
            printf("%d ", topoOrder[i]);
        printf("\n");
    }
}

int main() {
    int n, e, u, v;

    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    printf("Enter edges (from to):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d", &u, &v);
        addEdge(u, v);
    }

    topologicalSort(n);

    return 0;
}

```

OUTPUT

```
Enter number of vertices: 6
Enter number of edges: 6
Enter edges (from to):
5 2
5 0
4 0
4 1
2 3
3 1
Topological Order: 4 5 0 2 3 1
```

TRACING

Tracing:

- 1) Calculate indegrees
 $\text{indeg}[0] = 2$ (from 4, 5)
 $\text{indeg}[1] = 2$ (from 3, 4)
 $\text{indeg}[2] = 1$ (from 5)
 $\text{indeg}[3] = 1$ (from 2)
 $\text{indeg}[4] = 0$
 $\text{indeg}[5] = 0$
- 2) Queue: [4, 5] ($\rightarrow \text{indeg} = 0$)
- 3) * Remove 4 \rightarrow Add to result \rightarrow Result: [4]
 $\rightarrow 4 \rightarrow 0 \rightarrow \text{indeg}[0] = 1$
 $\rightarrow 4 \rightarrow 1 \rightarrow \text{indeg}[1] = 1$
Queue = [5]
- * Remove 5 \rightarrow Add to result \rightarrow Result: [4, 5]
 $\rightarrow 5 \rightarrow 2 \rightarrow \text{indeg}[2] = 0 \rightarrow$ Add 2 to queue
 $\rightarrow 5 \rightarrow 0 \rightarrow \text{indeg}[0] = 0 \rightarrow$ Add 0 to queue
Queue: [2, 0]
- * Remove 2 \rightarrow Result: [4, 5, 2]
 $\rightarrow 2 \rightarrow 3 \rightarrow \text{indeg}[3] = 0 \rightarrow$ Add 3 to queue
Queue: [0, 3]
- * Remove 0 \rightarrow Result: [4, 5, 2, 0].
 $\rightarrow 0$ has no outgoing edges Queue [3]

* Remove 3 \rightarrow Result: [4, 5, 2, 0, 3]

$\rightarrow 3 \rightarrow 1 \rightarrow \text{indeg}[1] = 0 \rightarrow \text{Add } 1 \text{ to queue}$

Queue: [1]

* Remove 1 \rightarrow Result: [4, 5, 2, 0, 3, 1]

$\rightarrow 1$ has no outgoing edges

Queue: []

Final Topological Order: [4, 5, 2, 0, 3, 1]

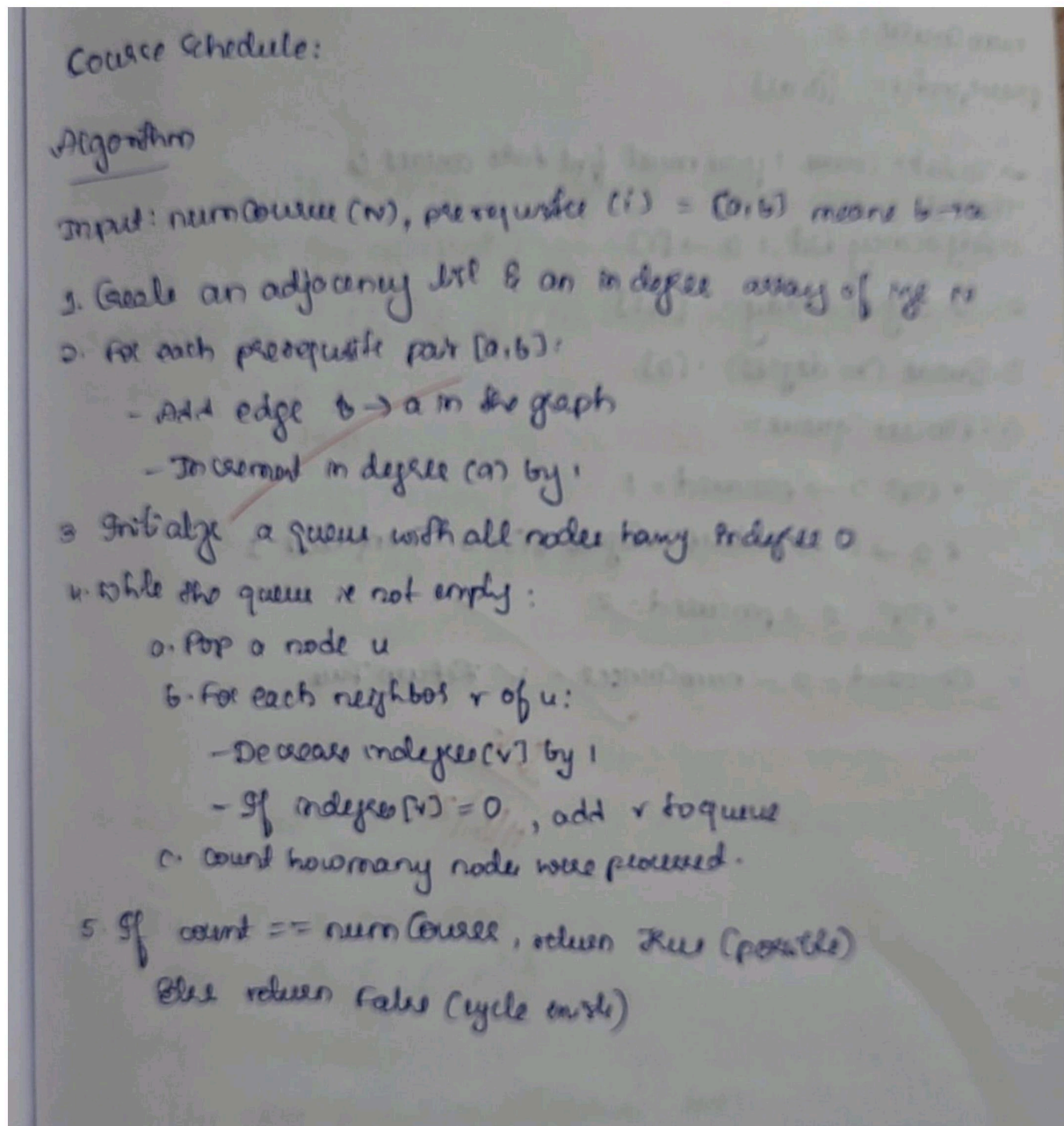
~~Soln~~
17/5/25

6/6

LEETCODE 3

COURSE SCHEDULE

ALGORITHM



CODE

```
bool canFinish(int numCourses, int** prerequisites, int prerequisitesSize, int* prerequisitesColSize){
```

```
    // Build graph using adjacency list
```

```
    int* adj[numCourses];
```

```

int adjSize[numCourses];
int indegree[numCourses];

for (int i = 0; i < numCourses; i++) {
    adj[i] = (int*)malloc(sizeof(int) * numCourses); // worst case
    adjSize[i] = 0;
    indegree[i] = 0;
}

// Fill graph and indegree
for (int i = 0; i < prerequisitesSize; i++) {
    int to = prerequisites[i][0];
    int from = prerequisites[i][1];
    adj[from][adjSize[from]++] = to;
    indegree[to]++;
}

// Queue for BFS
int* queue = (int*)malloc(sizeof(int) * numCourses);
int front = 0, rear = 0;

// Push all courses with 0 indegree
for (int i = 0; i < numCourses; i++) {
    if (indegree[i] == 0)
        queue[rear++] = i;
}

int visited = 0;

while (front < rear) {

```



```

    int curr = queue[front++];
    visited++;

    for (int i = 0; i < adjSize[curr]; i++) {
        int neighbor = adj[curr][i];
        indegree[neighbor]--;
        if (indegree[neighbor] == 0)
            queue[rear++] = neighbor;
    }
}

free(queue);
for (int i = 0; i < numCourses; i++) {
    free(adj[i]);
}

return visited == numCourses;
}

```

OUTPUT:

Accepted

Runtime: 0 ms

• Case 1

• Case 2

Input

```
numCourses =  
2
```

```
prerequisites =  
[[1,0]]
```

Output

```
true
```

Expected

```
true
```

• Case 1

• Case 2

Input

```
numCourses =  
2
```

```
prerequisites =  
[[1,0],[0,1]]
```

Output

```
false
```

Expected

```
false
```

TRACING

Example:

numCourses = 2
prerequisites = [[1,0]]

→ To take course 1 you must first take course 0

Tracing:

1. Adjacency List : 0 → [1]
2. In degree Array : [0,1]
3. Queue (in degree) : [0]
4. Process queue:
 - * POP 0 → processed = 1
 - * 0 → 1 → reduce in degree [1] = 0 → enqueue 1
 - * POP 1 → processed = 2
5. Processed = 2 = numCourses = ✓ Return True