CSE 4304-Data Structures Lab. Winter 23-24
Batch: CSE 22
Date: March 04, 2025
Target Group: All
Topic: Graph BFS DFS

Instructions:

- Regardless of how you finish the lab tasks, you must submit the solutions in Google Classroom. In case I forget to upload the tasks there, CR should contact me. The deadline will always be 11:59 PM on the day the lab took place.
- Task naming format: fullID_T01L01_2A.c/cpp
- If you find any issues in the problem description/test cases, comment in the Google Classroom.
- If you find any tricky test cases that I didn't include but that others might forget to handle, please comment! I'll be happy to add them.
- Use appropriate comments in your code. This will help you recall the solution easily in the future.
- Obtained marks will vary based on the efficiency of the solution.
- Do not use <bits/stdc++.h> library.
- Modified sections will be marked with BLUE color.
- You can use the STL stack unless it's specifically mentioned that you should use manual functions.

Group	**Tasks**
2A	1 2 3
1B	1 2 3
1A	1 2 4 5
2B	1 2 4 5
Assignments	2A/1B: 1A/2B:

Task 1: (10 points) Implement Breadth First Search (BFS) for an undirected unweighted graph.

Given a graph G = (V, E) and a source vertex ‘s’:

- Implement the BFS algorithm to discover every vertex reachable from ‘s’.
- Compute the distance of every reachable vertex from s.
- Print the path of each reachable vertex from s.
- Print the list of edges that produces the BFS tree.

Each nodes has three attributes: distance, predecessor, and color.

Input:
First line will contain V, E, and s representing the total number of vertices, edges, and the source vertex. Following E lines will contain the edge information.

Output:

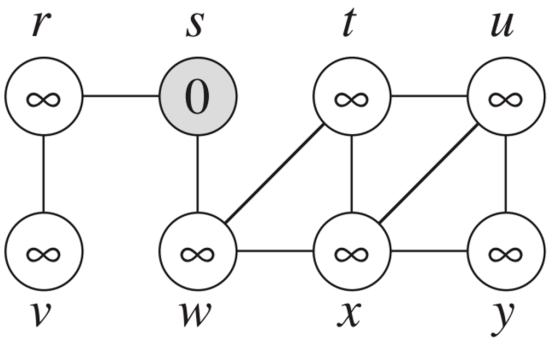
1. (1 points) After storing the Graph, first print the adjacency list
2. (2 points) Print the order in which nodes will be discovered in BFS
3. (2 points) Print the distance of each node from source
4. (3 points) Print the path to reach each vertex from the source
5. (2 points) Print the edges of the BFS tree

Sample Input	**Sample Output**
8 10 2 2 6 2 1 6 3 6 7 1 5 3 4 3 7 7 8 7 4 4 8	Adjacency list: 1: 2 5 2: 6 1 3: 6 4 7 4: 3 7 8 5: 1 6: 2 3 7 7: 6 3 8 4 8: 7 4 BFS order: 2 6 1 3 7 5 4 8 Distance from source: 1(1) 2(0) 3(2) 4(3) 5(2) 6(1) 7(2) 8(3) Paths from source: 1: 2->1 2: 2 3: 2->6->3 4: 2->6->3->4 5: 2->1->5 6: 2->6 7: 2->6->7 8: 2->6->7->8 Edges of BFS tree: (order may vary) 2 6 2 1 6 3 6 7 1 5 3 4 7 8

Steps:

- Use pushback to insert new nodes in adjacency list. This should result into the given output list.
- Don't implement everything at once. First implement the BFS algorithm, then show the distance, then the path, and finally the tree-edge list. There are separate marks for each step.
- Following figure is taken from the lecture slide. We considered {r,s,t,u,v,w,x,y} as {1,2,3,...,8}. Hence simulation should match with the slide.

(a)



Q | s | 0

- Use the following pseucode for BFS algorithm

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Task 2: (10 points) Implement Depth First Search (DFS) for an **Directed** unweighted graph.

Given a graph G = (V, E):

- Implement the DFS algorithm to discover every vertex.
- Classify the type of each edge.

Each nodes has four attributes: discovery time, finishing time, predecessor, and color.

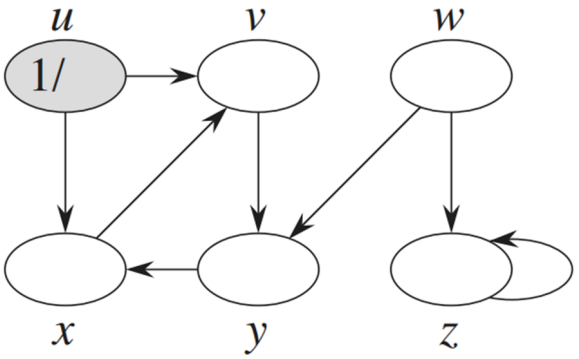
Input:
First line will contain V, and E representing the total number of vertices, and edges. Following E lines will contain the edge information.

Output:

1. (1 point) After storing the Graph, first print the adjacency list
2. (3 points) Print the order in which nodes will be discovered in DFS
3. (2 points) Print the discovery and finishing timestamps of each node
4. (4 points) Print class type of each edge

Sample Input	**Sample Output**
6 8 1 2 1 4 2 5 5 4 4 2 3 5 3 6 6 6	Adjacency list: 1: 2 4 2: 5 3: 5 6 4: 2 5: 4 6: 6 DFS order: 1 2 5 4 3 6 Timestamps of Vertex(discovery/finishing): 1(1/8) 2(2/7) 3(9/12) 4(4/5) 5(3/6) 6(10/11) Edge classification: 1 2: Tree Edge 1 4: Forward Edge 2 5: Tree Edge 5 4: Tree Edge 4 2: Back Edge 3 5: Cross Edge 3 6: Tree Edge 6 6: Back Edge

- Following figure is taken from the lecture slide. We considered {u,v,w,x,y,z} as {1,2,3,...,6}. Hence simulation should match with the slide.



(a)

- Use the following pseudocode for DFS:

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1            // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]      // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK            // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Task 3: (10 points) Budget Friendly Caraxes in Budget Friendly Westeros

In the budget-friendly version of Westeros, King (Consort) Daemon Targaryen is set on a journey riding his dragon Caraxes. However, in this version, Caraxes doesn't fly; instead, it hops across cities in Westeros. Each time Caraxes hops, it can hop over exactly k cities in one move. Given a starting location and an ending location, your task is to determine if Daemon can reach his destination and also print the path he must follow to get there, if possible.

Input:

- The first line contains two integers V, E and Q representing the total number of cities, edge information and Queries respectively.(undirected)
- The next line will contain V strings representing names of cities in Westeros.
- The following E lines contain two city names (strings) *city1* and *city2* which represents an edge from *city1 to city2*.
- The following Q lines contain src, dst and k which represents source city, destination city and number of cities Caraxes cover in each hop respectively.

Output:

- If it is possible to reach the destination city from the source, print the path.
- Otherwise print **NO**.

Sample Input	**Sample Output**
12 10 KingsLanding Blackwood Winterfell Dorne Dragonstone Wall Essos Lannisport Harrenhal Riverrun RedKeep StormsEnd	
KingsLanding Blackwood KingsLanding Winterfell KingsLanding Dorne Winterfell Wall Winterfell Dragonstone Dragonstone Harrenhal Dragonstone Riverrun Dorne Essos Dorne Lannisport Essos RedKeep	
KingsLanding Harrenhal 1	KingsLanding Winterfell Dragonstone Harrenhal
KingsLanding Harrenhal 3	NO
Harrenhal RedKeep 2	Harrenhal Winterfell Dorne RedKeep
RedKeep Riverrun 3	RedKeep KingsLanding Riverrun
StormsEnd Winterfell 1	NO
Essos Dragonstone 5	NO

Note that –

1. A city can’t appear twice in a path
2. There are no cycles in the graph
3. There might be some disconnected components in the graph

Task 4: (10 points) Mr. Unascendable

The brilliant (but slightly eccentric) engineering students of the Institute of Unending Torment have created a wondrous robot named "Unascendable". This marvel of engineering can perform a dazzling array of tasks, from completing DS lab tasks in seconds to answering COA questions correctly. However, due to a design flaw involving a misplaced spring and an overabundance of enthusiasm for optimizing "downward mobility", Unascendable **cannot climb**. Not even a little bit.

Unascendable has been placed on a grid-based testing ground with varying terrain height. The grid is represented by an m x n integer matrix called map, where map[r][c] represents the height of the block at coordinate (r, c). The top and left edges of the grid border the "Green Zone", a lush and verdant area filled with cooling pods and repair tools. The bottom and right edges border the "Red Zone", a fiery volcanic region with magma powered fast charging stations.

Unascendable can move from one block to an adjacent block (north, south, east, or west) only if the **height of the destination block is less than or equal to the height of its current block**. In other words, it can jump down or move across to blocks of the same height, but climbing is strictly off-limits.

You are given the map matrix. Determine the starting coordinates (r, c) of Unascendable from which the robot can reach both the Green Zone and the Red Zone. There can be multiple starting coordinates from which it can reach both zones. You have to find all of those coordinates and represent them in a similar map.

Input:

- The first line contains two integers m and n representing dimension of the map.
- The following lines will contain the map where each position map[r][c] denotes the height of the terrain in (r,c) position.

Output:

- An m x n character matrix with each cell containing either 'O' or 'X'. 'O' in the cell (i,j) means Unascendable can reach both Green and Red zones starting from (i,j) and 'X' means it can't.

Sample Input	**Sample Output**
3 3 2 3 3 2 1 2 3 2 2	 X O O X X X O X X
5 5 1 2 2 3 5 3 2 3 4 4 2 4 5 3 1 6 7 1 4 5 5 1 1 2 4	 X X X X O X X X O O X X O X X O O X X X O X X X X
4 3 1 4 1 2 3 2 3 2 3 4 1 4	 X O O X O O O X X O X O

1 1 1	0

Test case 1 explanation

Input

	0	1	2	
0	2	3	3	0
1	2	1	2	1
2	3	2	2	2
	0	1	2	

Output

	0	1	2	
0	X	O (U,RR)	O(U,R)	0
1	X	X	X	1
2	O (L,D)	X	X	2
	0	1	2	

X – Not reachable O – Reachable
(G,R) – G → moves to reach Green Zone, R → moves to reach Red Zone
L → Left R → Right
U → Up D → Down

Task 5: (5 points) Island Tally

An ancient pirate map has been discovered, revealing the location of a hidden treasure buried within a vast archipelago. The map is a 2D grid where '1' marks islands and '0' marks the surrounding sea. To plan your treasure-hunting expedition, you need to determine the exact number of islands in this archipelago.

Input:

- The first line will contain two integers, m and n, separated by a space, representing the number of rows and columns of the grid, respectively.
- The following m lines will each contain a string of length n consisting of '1's and '0's. Each string represents a row in the grid.

Output:
The output will be a single integer representing the number of islands in the grid.

Note: Islands can't be diagonally connected.

Sample Input	**Sample Output**
5 5 11000 11000 00100 00011 10001	4
3 3 101 010 101	5
4 4 1100 1100 0011 0011	2
4 4 1010 0101 1010 0101	8
7 7 1111111 1000001 1011101 1010101 1011101 1000001 1111111	1