**Roll No…………….. Total No. of Pages:……**

**ST-5 (SET-V)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

1. In a directed graph, back edges are used to detect cycles during:
   1. **Depth-First Search (DFS)**
   2. Breadth-First Search (BFS)
   3. Dijkstra's algorithm
   4. Prim's algorithm
2. The presence of a back edge during DFS traversal indicates the presence of a(n):
   1. Acyclic graph
   2. Bipartite graph
   3. Strongly connected component
   4. **Cycle**
3. Which algorithm can be used to determine if a graph is bipartite?
   1. Dijkstra's algorithm
   2. Prim's algorithm
   3. **Depth-First Search (DFS)**
   4. Breadth-First Search (BFS)
4. In BFS, the data structure typically used for maintaining the vertices is:
   1. Stack
   2. **Queue**
   3. Linked List
   4. Array
5. DFS can be used to find which of the following in a graph?
6. Shortest path between two nodes
7. Minimum spanning tree
8. **Topological ordering**
9. All of the above
10. Which of the following graph algorithms can be used to solve the single-source shortest path problem on a directed acyclic graph (DAG)?

a) Dijkstra's algorithm

b) Bellman-Ford algorithm

**c) Topological sorting**

d) Floyd-Warshall algorithm

1. Which algorithm is used to find the maximum flow in a flow network?

a) Kruskal's algorithm

b) Dijkstra's algorithm

c) Bellman-Ford algorithm

**d) Ford-Fulkerson algorithm**

1. What is the chromatic number of a graph?

a) The maximum degree of a vertex in the graph

b) The minimum degree of a vertex in the graph

**c) The minimum number of colors needed to color the vertices such that no two adjacent vertices share the same color**

d) The maximum number of colors that can be assigned to the vertices

1. In a Trie, what is the maximum number of children a node can have?
   1. 1
   2. 2
   3. **26 (for each alphabet letter)**
   4. Unlimited
2. Which operation is used to find the total number of keys stored in a Trie?
   1. **size()**
   2. count()
   3. length()
   4. keysCount()

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

11) #include <iostream>

#include <vector>

#include <queue>

using namespace std;

void BFS(vector<vector<int>>& graph, int start) {

queue<int> q;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : graph[node]) {

q.push(neighbor);

}

}

}

int main() {

int n = 3;

vector<vector<int>> graph(n + 1);

graph[1] = {2, 3};

graph[2] = {3};

graph[3] = {};

BFS(graph, 1);

return 0;

}

What will be the output of the program?

**a) 1 2 3**

b) 1 3 2

c) 1 2

d) 1 3

12) #include <iostream>

#include <vector>

#include <climits>

using namespace std;

void BellmanFord(vector<vector<pair<int, int>>>& graph, int start) {

int n = graph.size();

vector<int> dist(n, INT\_MAX);

dist[start] = 0;

for (int i = 0; i < n - 1; ++i) {

for (int u = 0; u < n; ++u) {

for (auto edge : graph[u]) {

int v = edge.first;

int weight = edge.second;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

}

for (int i = 1; i < n; ++i) {

cout << dist[i] << " ";

}

}

int main() {

int n = 5;

vector<vector<pair<int, int>>> graph(n + 1);

graph[1].push\_back({2, 6});

graph[1].push\_back({3, 5});

graph[2].push\_back({4, -2});

graph[3].push\_back({2, -2});

graph[2].push\_back({3, 7});

graph[4].push\_back({5, 2});

graph[3].push\_back({5, 3});

BellmanFord(graph, 1);

return 0;

}

What will be the output of the program?

a) 0 1 5 -2 0

b) 0 6 5 -2 0

**c) 0 3 5 1 3**

d) 0 6 7 2 5

13) #include <iostream>

#include <vector>

#include <queue>

using namespace std;

void Dijkstra(vector<vector<pair<int, int>>>& graph, int start) {

int n = graph.size();

vector<int> dist(n, INT\_MAX);

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto edge : graph[u]) {

int v = edge.first;

int weight = edge.second;

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

for (int i = 1; i < n; ++i) {

cout << dist[i] << " ";

}

}

int main() {

int n = 4;

vector<vector<pair<int, int>>> graph(n + 1);

graph[1].push\_back({2, 1});

graph[1].push\_back({3, 4});

graph[2].push\_back({3, 2});

graph[2].push\_back({4, 3});

graph[3].push\_back({4, 1});

Dijkstra(graph, 1);

return 0;

}

What will be the output of the program?

**a) 0 1 3 4**

b) 0 1 2 3

c) 0 3 4 5

d) 0 1 4 5

14) Consider the following adjacency matrix for a directed graph:

a b c

a 0 ∞ 1

b ∞ 0 ∞

c ∞ 2 0

After applying the Floyd-Warshall algorithm, what will be the distance from vertex 'c' to vertex 'b'?

a) 0

b) 1

**c) 2**

d) ∞

15) A sink in a directed graph is a vertex i such that there is an edge from every vertex j ≠ i to i and there is no edge from i to any other vertex. A directed graph G with n vertices is represented by its adjacency matrix A, where A[i] [j] = 1 if there is an edge directed from vertex i to j and 0 otherwise. The following algorithm determines whether there is a sink in the graph G.

i = 0

do {

j = i + 1;

while ((j < n) && E1) j++;

if (j < n) E2;

} while (j < n);

flag = 1;

for (j = 0; j < n; j++)

if ((j! = i) && E3)

flag = 0;

if (flag)

printf("Sink exists");

else

printf("Sink does not exist");

Choose the correct expressions for E3

a) (A[i][j] && !A[j][i])

b) (!A[i][j] && A[j][i])

c) (!A[i][j] | | A[j][i])

**d) (A[i][j] | | !A[j][i])**

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) A C++ program to check if a given graph is Eulerian or not

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | [ (1, 0),(0, 2),(2, 1)(0, 3),(3, 4)] | [(0, 1),(1, 2),(2, 0)] | [(1,0),(0,2),(2,1),(0,3),(3,4),(1,3)] |
| **Output** | graph has a Euler path | graph has a Euler cycle | graph is not Eulerian |

Solution :

**#include<iostream>**

**#include <list>**

**using namespace std;**

**// A class that represents an undirected graph**

**class Graph**

**{**

**int V; // No. of vertices**

**list<int> \*adj; // A dynamic array of adjacency lists**

**public:**

**// Constructor and destructor**

**Graph(int V) {this->V = V; adj = new list<int>[V]; }**

**~Graph() { delete [] adj; } // To avoid memory leak**

**// function to add an edge to graph**

**void addEdge(int v, int w);**

**// Method to check if this graph is Eulerian or not**

**int isEulerian();**

**// Method to check if all non-zero degree vertices are connected**

**bool isConnected();**

**// Function to do DFS starting from v. Used in isConnected();**

**void DFSUtil(int v, bool visited[]);**

**};**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w);**

**adj[w].push\_back(v); // Note: the graph is undirected**

**}**

**void Graph::DFSUtil(int v, bool visited[])**

**{**

**// Mark the current node as visited and print it**

**visited[v] = true;**

**// Recur for all the vertices adjacent to this vertex**

**list<int>::iterator i;**

**for (i = adj[v].begin(); i != adj[v].end(); ++i)**

**if (!visited[\*i])**

**DFSUtil(\*i, visited);**

**}**

**// Method to check if all non-zero degree vertices are connected.**

**// It mainly does DFS traversal starting from**

**bool Graph::isConnected()**

**{**

**// Mark all the vertices as not visited**

**bool visited[V];**

**int i;**

**for (i = 0; i < V; i++)**

**visited[i] = false;**

**// Find a vertex with non-zero degree**

**for (i = 0; i < V; i++)**

**if (adj[i].size() != 0)**

**break;**

**// If there are no edges in the graph, return true**

**if (i == V)**

**return true;**

**// Start DFS traversal from a vertex with non-zero degree**

**DFSUtil(i, visited);**

**// Check if all non-zero degree vertices are visited**

**for (i = 0; i < V; i++)**

**if (visited[i] == false && adj[i].size() > 0)**

**return false;**

**return true;**

**}**

**/\* The function returns one of the following values**

**0 --> If graph is not Eulerian**

**1 --> If graph has an Euler path (Semi-Eulerian)**

**2 --> If graph has an Euler Circuit (Eulerian) \*/**

**int Graph::isEulerian()**

**{**

**// Check if all non-zero degree vertices are connected**

**if (isConnected() == false)**

**return 0;**

**// Count vertices with odd degree**

**int odd = 0;**

**for (int i = 0; i < V; i++)**

**if (adj[i].size() & 1)**

**odd++;**

**// If count is more than 2, then graph is not Eulerian**

**if (odd > 2)**

**return 0;**

**// If odd count is 2, then semi-eulerian. If odd count is 0, then eulerian**

**// Note that odd count can never be 1 for undirected graph**

**return (odd)? 1 : 2;**

**}**

**// Function to run test cases**

**void test(Graph &g)**

**{**

**int res = g.isEulerian();**

**if (res == 0)**

**cout << "graph is not Eulerian\n";**

**else if (res == 1)**

**cout << "graph has a Euler path\n";**

**else**

**cout << "graph has a Euler cycle\n";**

**}**

**int main()**

**{**

**// Let us create and test graphs shown in above figures**

**Graph g1(5);**

**g1.addEdge(1, 0);**

**g1.addEdge(0, 2);**

**g1.addEdge(2, 1);**

**g1.addEdge(0, 3);**

**g1.addEdge(3, 4);**

**test(g1);**

**Graph g2(5);**

**g2.addEdge(1, 0);**

**g2.addEdge(0, 2);**

**g2.addEdge(2, 1);**

**g2.addEdge(0, 3);**

**g2.addEdge(3, 4);**

**g2.addEdge(4, 0);**

**test(g2);**

**Graph g3(5);**

**g3.addEdge(1, 0);**

**g3.addEdge(0, 2);**

**g3.addEdge(2, 1);**

**g3.addEdge(0, 3);**

**g3.addEdge(3, 4);**

**g3.addEdge(1, 3);**

**test(g3);**

**// Let us create a graph with 3 vertices**

**// connected in the form of cycle**

**Graph g4(3);**

**g4.addEdge(0, 1);**

**g4.addEdge(1, 2);**

**g4.addEdge(2, 0);**

**test(g4);**

**// Let us create a graph with all vertices**

**// with zero degree**

**Graph g5(3);**

**test(g5);**

**return 0;**

**}**

Q17) Given a Directed Acyclic Graph of n nodes (numbered from 1 to n) and m edges. The task is to find the number of sink nodes.

A sink node is a node such that no edge emerges out of it.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | n = 4, m = 2  Edges[] = {{2, 3}, {4, 3}} | n = 4, m = 2  Edges[] = {{3, 2}, {3, 4}} | n = 4, m = 3  Edges[] = {{1, 2}, {2, 3},{3,4}} |
| **Output** | 2 | 3 | 0 |

Solution :

**#include<bits/stdc++.h>**

**using namespace std;**

**// Return the number of Sink NOdes.**

**int countSink(int n, int m, int edgeFrom[],**

**int edgeTo[])**

**{**

**// Array for marking the non-sink node.**

**int mark[n];**

**memset(mark, 0, sizeof mark);**

**// Marking the non-sink node.**

**for (int i = 0; i < m; i++)**

**mark[edgeFrom[i]] = 1;**

**// Counting the sink nodes.**

**int count = 0;**

**for (int i = 1; i <= n ; i++)**

**if (!mark[i])**

**count++;**

**return count;**

**}**

**int main()**

**{**

**int n = 4, m = 2;**

**int edgeFrom[] = { 2, 4 };**

**int edgeTo[] = { 3, 3 };**

**cout << countSink(n, m, edgeFrom, edgeTo) << endl;**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given a matrix of dimension M \* N where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

0: Empty cell

1: Cells have fresh oranges

2: Cells have rotten oranges

Determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index (i,j ) can rot other fresh oranges which are

its neighbours (up, down, left and right). If it is impossible to rot every orange then simply return -1.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | { {2, 1, 0, 2, 1}, {1, 0, 1, 2, 1}, {1, 0, 0, 2, 1}} | { {2, 1, 0, 2, 1}, {0, 0, 1, 2, 1}, {1, 0, 0, 2, 1}} | { { 1, 1, 2 }, { 1, 1, 2 }, { 2, 1, 1 } } |
| **Output** | 2 | -1 | 2 |

Solution :

**#include <bits/stdc++.h>**

**#define R 3**

**#define C 5**

**using namespace std;**

**// This function finds if it is possible to rot all oranges or not. If possible, then it returns minimum time required to rot all, otherwise returns -1**

**int rotOranges(vector<vector<int> >& grid)**

**{**

**int n = grid.size(); // row size**

**int m = grid[0].size(); // column size**

**// delrow and delcol are used to traverse in**

**// up,right,bottom and left respectively.**

**int delrow[] = { -1, 0, 1, 0 };**

**int delcol[] = { 0, 1, 0, -1 };**

**// visited matrix to keep track of the visited cell.**

**int vis[n][m];**

**// queue stores rowIndex,colIndex and time taken to rot**

**// respectively.**

**queue<pair<pair<int, int>, int> > q;**

**// counter to keep track of fresh cells.**

**int cntfresh = 0;**

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < m; j++) {**

**if (grid[i][j] == 2) {**

**q.push({ { i, j },**

**0 }); // already rotten hence 0**

**// time to rot.**

**vis[i][j]**

**= 2; // visited cell marked as rotten.**

**}**

**else {**

**vis[i][j] = 0; // unvisited**

**}**

**if (grid[i][j] == 1)**

**cntfresh++; // maintaining count for fresh**

**// oranges.**

**}**

**}**

**int cnt = 0, tm = 0;**

**while (!q.empty()) {**

**int row = q.front().first.first; // row index**

**int col = q.front().first.second; // col index**

**int t = q.front().second; // time an orange at a**

**// cell takes to rot.**

**q.pop();**

**tm = max(tm, t);**

**// checking for adjacent nodes in 4 directions.**

**for (int i = 0; i < 4; i++) {**

**int nrow = row + delrow[i];**

**int ncol = col + delcol[i];**

**// checking the validity of a node and also**

**// vis[nrow][ncol] !=2**

**if (nrow >= 0 && nrow < n && ncol >= 0**

**&& ncol < m && grid[nrow][ncol] == 1**

**&& vis[nrow][ncol] != 2) {**

**vis[nrow][ncol] = 2; // adj orange is rotten**

**q.push({ { nrow, ncol },**

**t + 1 }); // incrementing time for**

**// that orange by 1**

**cnt++;**

**}**

**}**

**}**

**return (cnt == cntfresh) ? tm : -1;**

**}**

**int main()**

**{**

**vector<vector<int> > arr**

**= { { 0, 1, 2 }, { 0, 1, 2 }, { 2, 1, 1 } };**

**int ans = rotOranges(arr);**

**if (ans == -1)**

**cout << "All oranges cannot rotn";**

**else**

**cout << "Time required for all oranges to rot => "**

**<< ans << endl;**

**return 0;**

**}**