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

**ST-5 (SET-I)**

**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 an undirected graph, a bridge is an edge that, when removed, increases the number of:
   1. Vertices
   2. Edges
   3. **Connected components**
   4. Cycles
2. Which graph traversal algorithm can be used to detect cycles in an undirected graph?
   1. **DFS**
   2. BFS
   3. Dijkstra's algorithm
   4. Prim's algorithm
3. In Tarjan's algorithm, a vertex is considered part of an SCC when:
   1. It has the smallest index
   2. It has the largest index
   3. **It has the lowest low-link value**
   4. It has the highest low-link value
4. Bellman-Ford algorithm is used to find:
   1. Strongly Connected Components
   2. **Shortest Path in a weighted graph**
   3. Minimum Spanning Tree
   4. Maximum Flow
5. Which traversal algorithm explores the graph layer by layer?

**a) BFS**

b) DFS

c) Both BFS and DFS

d) Neither BFS nor DFS

1. In Dijkstra's algorithm, what is the primary difference between using a priority queue and a simple queue?

a) A priority queue guarantees the shortest paths.

**b) A simple queue ensures that vertices are explored in increasing order of their distances.**

c) A priority queue explores vertices in the order they were inserted.

d) A simple queue is more efficient for large graphs.

1. Which operation in a Trie is used to remove a key from the Trie?
   1. **delete()**
   2. remove()
   3. erase()
   4. pop()
2. What is the advantage of using a Trie over a Hash Table for string keys?
   1. Tries have faster insertion and deletion operations.
   2. **Tries have constant-time lookup for any key length.**
   3. Tries do not require a hash function.
   4. Tries use less memory.
3. Which algorithm is used to find the articulation points (cut vertices) in a graph?

a) Prim's algorithm

b) Kruskal's algorithm

**c) Tarjan's algorithm**

d) Hopcroft-Karp algorithm

1. A Hamiltonian cycle in a graph is a cycle that:

**a) Visits every vertex exactly once and returns to the starting vertex**

b) Visits every edge exactly once

c) Visits some vertices more than once

d) Visits some edges more than once

**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 = 5;

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

graph[1] = {2, 3};

graph[2] = {4, 5};

graph[3] = {};

graph[4] = {};

graph[5] = {};

BFS(graph, 1);

return 0;

}

What will be the output of the program?

**a) 1 2 3 4 5**

b) 1 2 4 5 3

c) 1 2 3

d) 1 2

12) Which of the following functions correctly represent iterative DFS?

a)

void DFS(int s)

{

vector<bool> discovered(V, true);

stack<int> st;

st.push(s);

while (!st.empty())

{

s = st.top();

st.pop();

if (!discovered[s])

{

cout << s << " ";

discovered[s] = true;

}

for (auto i = adjacent[s].begin(); i != adjacent[s].end(); ++i)

if (!discovered[\*i])

st.push(\*i);

}

}

**b)**

**void DFS(int s)**

**{**

**vector<bool> discovered(V, false);**

**stack<int> st;**

**st.push(s);**

**while (!st.empty())**

**{**

**s = st.top();**

**st.pop();**

**if (!discovered[s])**

**{**

**cout << s << " ";**

**discovered[s] = true;**

**}**

**for (auto i = adjacent[s].begin(); i != adjacent[s].end(); ++i)**

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

**st.push(\*i);**

**}**

**}**

c)

void DFS(int s)

{

vector<bool> discovered(V, false);

stack<int> st;

st.push(s);

while (!st.empty())

{

st.pop();

s = st.top();

if (!discovered[s])

{

cout << s << " ";

discovered[s] = true;

}

for (auto i = adjacent[s].begin(); i != adjacent[s].end(); ++i)

if (!discovered[\*i])

st.push(\*i);

}

}

d)

void DFS(int s)

{

vector<bool> discovered(V, false);

stack<int> st;

st.push(s);

while (!st.empty())

{

s = st.top();

st.pop();

if (!discovered[s])

{

cout << s << " ";

discovered[s] = false;

}

for (auto i = adjacent[s].begin(); i != adjacent[s].end(); ++i)

if (discovered[\*i])

st.push(\*i);

}

}

13) #include <iostream>

#include <vector>

using namespace std;

void FloydWarshall(vector<vector<int>>& graph) {

int n = graph.size();

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

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

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

if (graph[i][k] != INT\_MAX && graph[k][j] != INT\_MAX && graph[i][k] + graph[k][j] < graph[i][j]) {

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

}

int main() {

int n = 4;

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

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

graph[i][i] = 0;

}

graph[0][1] = 3;

graph[1][2] = -2;

graph[2][0] = 7;

graph[2][3] = 1;

graph[3][0] = 2;

FloydWarshall(graph);

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

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

cout << graph[i][j] << " ";

}

cout << endl;

}

return 0;

}

What will be the output of the program?

a) 0 3 -2 1

5 0 -4 1

7 10 0 1

2 5 3 0

b) 0 3 1 1

7 0 -2 1

5 8 0 1

2 5 3 0

c) 0 3 1 2

7 0 -2 1

5 8 0 1

2 5 3 0

**d) 0 3 1 2**

**1 0 -2 -1**

**3 6 0 1**

**2 5 3 0**

14) What will be the output if we pass { {0 2 0},{0 0 -1},{-3 0 0 }} as input to following function?

void FloydWarshall(vector<vector<int>>& graph) {

int n = graph.size();

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

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

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

if (graph[i][k] != INT\_MAX && graph[k][j] != INT\_MAX && graph[i][k] + graph[k][j] < graph[i][j]) {

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

}

What will be the output of the program?

a) 0 2 -1

2 0 -1

-3 -1 0

**b) -2 0 -1**

**-4 -2 -3**

**-5 -3 -4**

c) 0 2 -1

2 0 -3

-3 -1 0

d) 0 2 -3

2 0 -3

-3 -1 0

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

a b c

a 0 2 ∞

b 1 0 ∞

c ∞ 3 0

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

a) 0

**b) 4**

c) 1

d) ∞

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

Q16) Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. See following examples for more details.

Note: This is a famous Google interview question, also being asked by many other companies now a days.

Consider the following dictionary

{ i, like, sam, sung, samsung, mobile, ice,

cream, icecream, man, go, mango}

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | ilike | Ilikesamsung | ilikechole |
| **Output** | yes | Yes | No |

Solution :

**#include <iostream>**

**using namespace std;**

**const int ALPHABET\_SIZE = 26;**

**// trie node**

**struct TrieNode {**

**struct TrieNode\* children[ALPHABET\_SIZE];**

**// isEndOfWord is true if the node represents**

**// end of a word**

**bool isEndOfWord;**

**};**

**// Returns new trie node (initialized to NULLs)**

**struct TrieNode\* getNode(void)**

**{**

**struct TrieNode\* pNode = new TrieNode;**

**pNode->isEndOfWord = false;**

**for (int i = 0; i < ALPHABET\_SIZE; i++)**

**pNode->children[i] = NULL;**

**return pNode;**

**}**

**// If not present, inserts key into trie**

**// If the key is prefix of trie node, just**

**// marks leaf node**

**void insert(struct TrieNode\* root, string key)**

**{**

**struct TrieNode\* pCrawl = root;**

**for (int i = 0; i < key.length(); i++) {**

**int index = key[i] - 'a';**

**if (!pCrawl->children[index])**

**pCrawl->children[index] = getNode();**

**pCrawl = pCrawl->children[index];**

**}**

**// mark last node as leaf**

**pCrawl->isEndOfWord = true;**

**}**

**// Returns true if key presents in trie, else**

**// false**

**bool search(struct TrieNode\* root, string key)**

**{**

**struct TrieNode\* pCrawl = root;**

**for (int i = 0; i < key.length(); i++) {**

**int index = key[i] - 'a';**

**if (!pCrawl->children[index])**

**return false;**

**pCrawl = pCrawl->children[index];**

**}**

**return (pCrawl != NULL && pCrawl->isEndOfWord);**

**}**

**// returns true if string can be segmented into**

**// space separated words, otherwise returns false**

**bool wordBreak(string str, TrieNode\* root)**

**{**

**int size = str.size();**

**// Base case**

**if (size == 0)**

**return true;**

**// Try all prefixes of lengths from 1 to size**

**for (int i = 1; i <= size; i++) {**

**// The parameter for search is str.substr(0, i)**

**// str.substr(0, i) which is prefix (of input**

**// string) of length 'i'. We first check whether**

**// current prefix is in dictionary. Then we**

**// recursively check for remaining string**

**// str.substr(i, size-i) which is suffix of**

**// length size-i**

**if (search(root, str.substr(0, i))**

**&& wordBreak(str.substr(i, size - i), root))**

**return true;**

**}**

**// If we have tried all prefixes and none**

**// of them worked**

**return false;**

**}**

**int main()**

**{**

**string dictionary[]**

**= { "mobile", "samsung", "sam", "sung", "ma\n",**

**"mango", "icecream", "and", "go", "i",**

**"like", "ice", "cream" };**

**int n = sizeof(dictionary) / sizeof(dictionary[0]);**

**struct TrieNode\* root = getNode();**

**// Construct trie**

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

**insert(root, dictionary[i]);**

**wordBreak("ilikesamsung", root) ? cout << "Yes\n": cout << "No\n";**

**wordBreak("iiiiiiii", root) ? cout << "Yes\n"cout << "No\n";**

**wordBreak("", root) ? cout << "Yes\n" : cout << "No\n";**

**wordBreak("ilikelikeimangoiii", root) ? cout << "Yes\n": cout << "No\n";**

**wordBreak("samsungandmango", root) ? cout << "Yes\n": cout << "No\n";**

**wordBreak("samsungandmangok", root) ? cout << "Yes\n": cout << "No\n";**

**return 0;**

**}**

Q17) A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other.

Given a Graph, find if it can be divided into two Cliques.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | G[][] = {{0, 1, 1, 0, 0}  {1, 0, 1, 1, 0}  {1, 1, 0, 0, 0}  {0, 1, 0, 0, 1}  {0, 0, 0, 1, 0}} | G[][] = {{0, 1, 1, 0, 0}  {1, 0, 1, 1, 0}  {1, 0, 0, 0, 0}  {0, 1, 0, 0, 1}  {0, 0, 0, 1, 0}} | G[][] = {{0, 1, 1, 0, 0}  {1, 0, 1, 1, 0}  {1, 1, 0, 0, 0}  {0, 1, 0, 0, 1}  {1, 0, 0, 1, 0}} |
| **Output** | Yes | No | Yes |

Solution :

**// C++ program to find out whether a given graph can be converted to two Cliques or not.**

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

**using namespace std;**

**const int V = 5;**

**// This function returns true if subgraph reachable from src is Bipartite or not.**

**bool isBipartiteUtil(int G[][V], int src, int colorArr[])**

**{**

**colorArr[src] = 1;**

**// Create a queue (FIFO) of vertex numbers and enqueue source vertex for BFS traversal**

**queue <int> q;**

**q.push(src);**

**// Run while there are vertices in queue (Similar to BFS)**

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

**{**

**// Dequeue a vertex from queue**

**int u = q.front();**

**q.pop();**

**// Find all non-colored adjacent vertices**

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

**{**

**// An edge from u to v exists and destination v is not colored**

**if (G[u][v] && colorArr[v] == -1)**

**{**

**// Assign alternate color to this adjacent v of u**

**colorArr[v] = 1 - colorArr[u];**

**q.push(v);**

**}**

**// An edge from u to v exists and destination v is colored with same color as u**

**else if (G[u][v] && colorArr[v] == colorArr[u])**

**return false;**

**}**

**}**

**// If we reach here, then all adjacent vertices can be colored with alternate color**

**return true;**

**}**

**// Returns true if a Graph G[][] is Bipartite or not. Note that G may not be connected.**

**bool isBipartite(int G[][V])**

**{**

**// Create a color array to store colors assigned to all vertices. Vertex number is used as index in**

**// this array. The value '-1' of colorArr[i] is used to indicate that no color is assigned to**

**// vertex 'i'. The value 1 is used to indicate first color is assigned and value 0 indicates second color is assigned.**

**int colorArr[V];**

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

**colorArr[i] = -1;**

**// One by one check all not yet colored vertices.**

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

**if (colorArr[i] == -1)**

**if (isBipartiteUtil(G, i, colorArr) == false)**

**return false;**

**return true;**

**}**

**// Returns true if G can be divided into two Cliques, else false.**

**bool canBeDividedinTwoCliques(int G[][V])**

**{**

**// Find complement of G[][] All values are complemented except diagonal ones**

**int GC[V][V];**

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

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

**GC[i][j] = (i != j)? !G[i][j] : 0;**

**// Return true if complement is Bipartite else false.**

**return isBipartite(GC);**

**}**

**int main()**

**{**

**int G[][V] = {{0, 1, 1, 1, 0},**

**{1, 0, 1, 0, 0},**

**{1, 1, 0, 0, 0},**

**{0, 1, 0, 0, 1},**

**{0, 0, 0, 1, 0}**

**};**

**canBeDividedinTwoCliques(G) ? cout << "Yes" : cout << "No";**

**return 0;**

**}**

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

Q18) Write a C++ program to find single source longest distances in a DAG

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | graph= [(0, 1, 5),(0, 2, 3),(1, 3, 6),(1, 2, 2),(2, 4, 4),(2, 5, 2),(2, 3, 7),(3, 5, 1),(3, 4, -1),(4, 5, -2)] | graph= [(0, 1, 5),(0, 2, 1),(1, 3, 2),(1, 2, 3),(2, 4, 4),(2, 5, 2),(2, 3, 7),(3, 5, 1),(3, 4, -1),(4, 5, -2)] | graph= [(0, 1, 5),(0, 2, 3),(1, 3, 6),(1, 2, 2),(2, 4, 4),(2, 5, -2),(2, 3, 7),(3, 5, 1),(3, 4, -6),(4, 5, -9)] |
| **Output** | Following are longest distances from source vertex 1:  INF 0 2 9 8 10 | Following are longest distances from source vertex 1  INF 0 3 10 9 11 | Following are longest distances from source vertex 1  INF 0 3 10 7 11 |

Solution :

**#include <iostream>**

**#include <limits.h>**

**#include <list>**

**#include <stack>**

**#define NINF INT\_MIN**

**using namespace std;**

**// Graph is represented using adjacency list. Every node of adjacency list contains vertex number of**

**// the vertex to which edge connects. It also contains weight of the edge**

**class AdjListNode {**

**int v;**

**int weight;**

**public:**

**AdjListNode(int \_v, int \_w)**

**{**

**v = \_v;**

**weight = \_w;**

**}**

**int getV() { return v; }**

**int getWeight() { return weight; }**

**};**

**// Class to represent a graph using adjacency list**

**// representation**

**class Graph {**

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

**// Pointer to an array containing adjacency lists**

**list<AdjListNode>\* adj;**

**// A function used by longestPath**

**void topologicalSortUtil(int v, bool visited[],**

**stack<int>& Stack);**

**public:**

**Graph(int V); // Constructor**

**~Graph(); // Destructor**

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

**void addEdge(int u, int v, int weight);**

**// Finds longest distances from given source vertex**

**void longestPath(int s);**

**};**

**Graph::Graph(int V) // Constructor**

**{**

**this->V = V;**

**adj = new list<AdjListNode>[V];**

**}**

**Graph::~Graph() // Destructor**

**{**

**delete [] adj;**

**}**

**void Graph::addEdge(int u, int v, int weight)**

**{**

**AdjListNode node(v, weight);**

**adj[u].push\_back(node); // Add v to u's list**

**}**

**// A recursive function used by longestPath. See below**

**// link for details**

**// https:// www.geeksforgeeks.org/topological-sorting/**

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

**stack<int>& Stack)**

**{**

**// Mark the current node as visited**

**visited[v] = true;**

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

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

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

**AdjListNode node = \*i;**

**if (!visited[node.getV()])**

**topologicalSortUtil(node.getV(), visited, Stack);**

**}**

**// Push current vertex to stack which stores topological**

**// sort**

**Stack.push(v);**

**}**

**// The function to find longest distances from a given vertex.**

**// It uses recursive topologicalSortUtil() to get topological**

**// sorting.**

**void Graph::longestPath(int s)**

**{**

**stack<int> Stack;**

**int dist[V];**

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

**bool\* visited = new bool[V];**

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

**visited[i] = false;**

**// Call the recursive helper function to store Topological**

**// Sort starting from all vertices one by one**

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

**if (visited[i] == false)**

**topologicalSortUtil(i, visited, Stack);**

**// Initialize distances to all vertices as infinite and**

**// distance to source as 0**

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

**dist[i] = NINF;**

**dist[s] = 0;**

**// Process vertices in topological order**

**while (Stack.empty() == false) {**

**// Get the next vertex from topological order**

**int u = Stack.top();**

**Stack.pop();**

**// Update distances of all adjacent vertices**

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

**if (dist[u] != NINF) {**

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

**if (dist[i->getV()] < dist[u] + i->getWeight())**

**dist[i->getV()] = dist[u] + i->getWeight();**

**}**

**}**

**}**

**// Print the calculated longest distances**

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

**(dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";**

**delete [] visited;**

**}**

**int main()**

**{**

**// Create a graph**

**// Here vertex numbers are 0, 1, 2, 3, 4, 5 with**

**// following mappings: 0=r, 1=s, 2=t, 3=x, 4=y, 5=z**

**Graph g(6);**

**g.addEdge(0, 1, 5);**

**g.addEdge(0, 2, 3);**

**g.addEdge(1, 3, 6);**

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

**g.addEdge(2, 4, 4);**

**g.addEdge(2, 5, 2);**

**g.addEdge(2, 3, 7);**

**g.addEdge(3, 5, 1);**

**g.addEdge(3, 4, -1);**

**g.addEdge(4, 5, -2);**

**int s = 1;**

**cout << "Following are longest distances from source vertex "<< s << " \n";**

**g.longestPath(s);**

**return 0;**

**}**