GRAPHS - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://practice.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1
- **Difficulty level**: Easy

Explanation:
Iterate through the given adjacency list and for each vertex, initialize a vector to store its adjacent vertices. Iterate through the list and push back the vertices to the new list. Push the vectors of all vertices to a new vector and return it.

Solution:

```
vector<vector<int>>printGraph(int V, vector<int> adj[])
{
      vector<vector<int>> result;
      for (int v = 0; v < V; ++v)
       {
           vector<int> it;
           it.push_back(v);
           for (auto x : adj[v])
              it.push_back(x);
           result.push_back(it);
       }
       return result;
}
```

Complexity:

- Time: O(V+E)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/find-center-of-star-graph/
- **Difficulty level**: Easy

Explanation:

Iterate through all the edges and find the degree for each vertex. In a star graph, the center is the vertex which has degree the same as the number of edges in the graph. So return the vertex with degree = number of edges. Another optimized solution is, consider any two edges and return the common vertex since the center is connected to every edge.

Solution:

```
class Solution {
private:
    unordered_map<int, int> edgesOrignatingFrom;
public:
    int findCenter(vector<vector<int>>& edges)
    {
        for ( int i=0; i<edges.size(); i++ ) {
            edgesOrignatingFrom[edges[i][0]] += 1;
            edgesOrignatingFrom[edges[i][1]] += 1;
        }

        for ( int i=1; i<=edges.size()+1; i++ ) {
            if ( edgesOrignatingFrom[i] == edges.size() ) {
                return i;
            }
        }

        return -1;
    }
};
```

Optimized Solution:

```
class Solution {
public:
    int findCenter(vector<vector<int>>& edges)
    {
        int a1 = edges[0][0];
        int a2 = edges[0][1];
        int b1 = edges[1][0];
        int b2 = edges[1][1];
        //cout<<a1<<" "<<a2<<" "<<b1<<" "<<b2<<endl;
        if(a1 == b1 || a2 == b1) {
            return b1;
        }
        else if(a1 == b2 || a2 == b2) {
            return b2;
        }
        else return -1;
    }
};
```

Complexity:

- Time: O(E)
- Extra space: O(1)

Question 3:

- **Problem link**: https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1
- **Difficulty level**: Easy

Explanation:
Maintain an array named visited to keep note of all the vertices that have been visited. As we are starting from vertex 0, visit that vertex and push all its neighbors to the queue and keep doing the same until all vertices are visited. The queue is used to ensure that we go to the next level only after all the nodes at the same level are visited.

Solution:

```
vector<int>bfsOfGraph(int V, vector<int> adj[])
    {
      vector<int>ans;
      bool *visited = new bool[V];
      for(int i = 0; i < V; i++)
           visited[i] = false;
      list<int> queue;
     visited[0] = true;
      queue.push_back(0);

      int s;

      while(!queue.empty())
      {
           s = queue.front();
           ans.push_back(s);
           queue.pop_front();

           // Get all adjacent vertices of the dequeued
```

```
            // vertex s. If a adjacent has not been visited,
            // then mark it visited and enqueue it
            for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
            {
                  if (!visited[*i])
                  {
                  visited[*i] = true;
                  queue.push_back(*i);
                  }
            }
      }
      return ans;
    }
```

Complexity:

- Time: O(V+E)
- Space: O(V)

Question 4:

- **Problem link**: https://practice.geeksforgeeks.org/problems/depth-first-traversal-for-a-graph/1
- **Difficulty level**: Easy

Explanation:

In DFS, For every node, we visit all the nodes that are adjacent to it .Hence every time we visit a node, we start recursively performing DFS for its adjacent nodes until we either reach the end or all nodes are visited.

Solution:

```
     vector<int>ans;
      vector<int>visited;
      public:
      void DFSFunction(int vertex,vector<int> adj[])
      {
      visited[vertex] = true;
      ans.push_back(vertex);

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

      for (auto i = adj[vertex].begin(); i != adj[vertex].end(); ++i)
           if (!visited[*i])
                DFSFunction(*i,adj);
      }
    vector<int>dfsOfGraph(int V, vector<int> adj[])
    {
      visited.resize(V);
      DFSFunction(0,adj);
      return ans;}
```

Complexity:

- Time: O(V+E)
- Space: O(V)

Question 5:

- **Problem link**: https://practice.geeksforgeeks.org/problems/detect-cycle-in-a-directed-graph/1
- **Difficulty level**: Medium

Explanation:

1. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
2. Mark the current node as visited and also mark the index in the recursion stack.
3. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
4. If the adjacent vertices are already marked in the recursion stack then return true.
5. Create a wrapper class that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

Solution:

```
class Solution
{
    public:
      //Function to detect cycle in a directed graph.
      bool isCyclicUtil(int v, bool visited[], bool *recStack, vector<int> adj[])
     {
        if(visited[v] == false)
        {
            // Mark the current node as visited and part of recursion stack
            visited[v] = true;
            recStack[v] = true;

            // Recur for all the vertices adjacent to this vertex
            vector<int>::iterator i;
            for(i = adj[v].begin(); i != adj[v].end(); ++i)
            {
                if ( !visited[*i] && isCyclicUtil(*i, visited, recStack, adj) )
                    return true;
                else if (recStack[*i])
                    return true;
            }

        }
        recStack[v] = false;  // remove the vertex from recursion stack
        return false;
    }
    bool isCyclic(int V, vector<int> adj[])
    {
         bool *visited = new bool[V];
         bool *recStack = new bool[V];
         for(int i = 0; i < V; i++)
         {
             visited[i] = false;
             recStack[i] = false;
         }

        // Call the recursive helper function to detect cycle in different
        // DFS trees
           for(int i = 0; i < V; i++)
               if (isCyclicUtil(i, visited, recStack, adj))
                   return true;

           return false;
    }
};
```

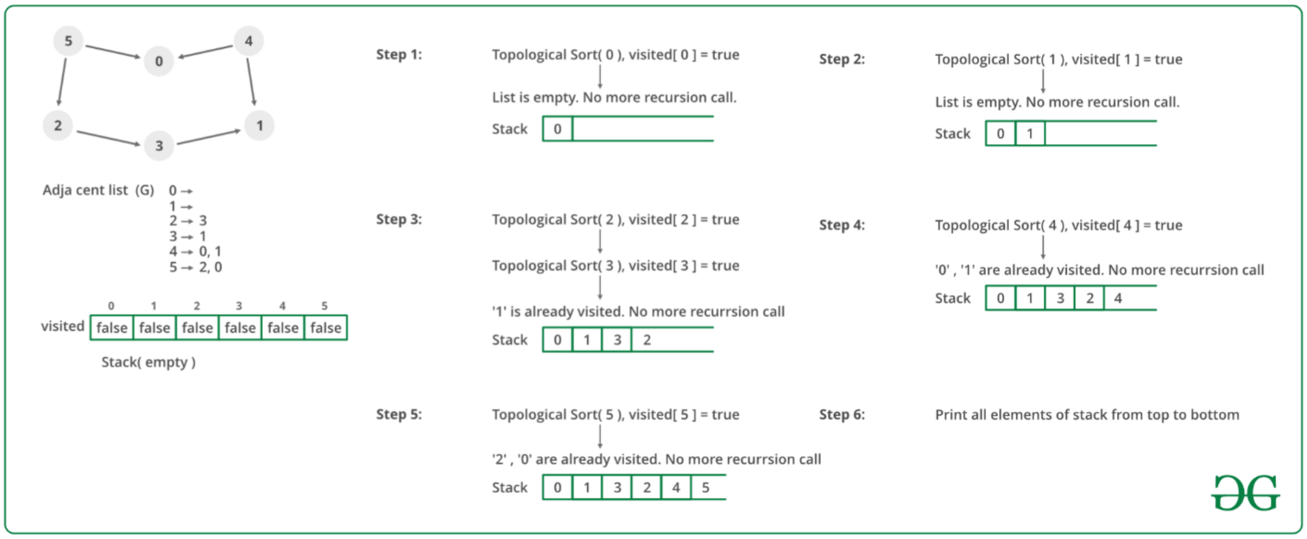
Complexity:

- Time: O(V+E)
- Space: O(V)

Question 6:

- **Problem link**: https://practice.geeksforgeeks.org/problems/topological-sort/1
- **Difficulty level**: Medium

Explanation:
We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print the contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.
Below image is an illustration of the above approach:


5
0
4
2
3
1
Adja cent list (G) 0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0
visited
0 1 2 3 4 5
false false false false false false
Stack(empty)
Step 1:
Topological Sort(0), visited[0] = true
List is empty. No more recursion call.
Stack 0
Step 2:
Topological Sort(1), visited[1] = true
List is empty. No more recursion call.
Stack 0 1
Step 3:
Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack 0 1 3 2
Step 4:
Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack 0 1 3 2 4
Step 5:
Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack 0 1 3 2 4 5
Step 6:
Print all elements of stack from top to bottom

Solution:

```
class Solution
{
      public:
      //Function to return list containing vertices in Topological order.
      void topologicalSortUtil(int v, bool visited[],stack<int>& Stack,
vector<int> adj[])
    {
        // Mark the current node as visited.
        visited[v] = true;

        // Recur for all the vertices
        // adjacent to this vertex
        vector<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                topologicalSortUtil(*i, visited, Stack, adj);

        // Push current vertex to stack
        // which stores result
        Stack.push(v);
    }

    // The function to do Topological Sort.
    // It uses recursive topologicalSortUtil()
      vector<int> topoSort(int V, vector<int> adj[])
      {
          stack<int> Stack;
          vector<int> result;

        // 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, adj);

        // Print contents of stack
        while (Stack.empty() == false) {
            result.push_back(Stack.top());
            Stack.pop();
        }
        return result;
    }
};
```

Complexity:

- Time: O(V+E)
- Space: O(V)

Question 7:

- **Problem link**: https://leetcode.com/problems/all-paths-from-source-to-target/
- **Difficulty level**: Medium

Explanation:

1. The idea is to do Depth First Traversal of the given directed graph.
2. Start the DFS traversal from source.
3. Keep storing the visited vertices in an array or HashMap say 'path[]'.
4. If the destination vertex is reached, print contents of path[].

5. The important thing is to mark current vertices in the path[] as visited also so that the traversal doesn't go in a cycle.

Solution:

```
class Solution {
public:
    vector<vector<int>>ans;
    void dfs(vector<vector<int>>& graph,int u,int v,vector<int>&vis)
    {
        vis.push_back(u);

        if(u==v)
        {
            ans.push_back(vis);
            //return;
        }
        else
        {
            for(auto node:graph[u])
            {
                dfs(graph,node,v,vis);
            }
        }
        vis.pop_back();

    }

    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph)
{
        vector<vector<int>>arr={};
        ans=arr;
        int n=graph.size();
        vector<int>vis={};
        dfs(graph,0,n-1,vis);
        return ans;
    }
};
```

Complexity:

- Time: O(2^n.n)
- Space: O(2^n.n)

Question 8:

- **Problem link**: https://practice.geeksforgeeks.org/problems/m-coloring-problem-1587115620/1
- **Difficulty level**: Hard

Explanation:

We'll have a function named isSafe that would help in checking if any adjacent nodes of have the same color c if the vertex V is colored with the color c. Then by taking each vertex, we assign it one color and check if the other nodes in the graph can be assigned with other colors and still satisfy the condition of proper coloring.

Solution:

```
bool isSafe(int v, bool graph[101][101], int color[101], int c, int V)
{
    //checking if the connected nodes to v have same colour as c.
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
        return false;

    //returning true if no connected node has same colour.
    return true;
}

bool graphColoringUtil(bool graph[101][101], int m, int color[101], int v,
                       int V)
{
    //if all vertices have been assigned colour then we return true.
    if (v == V)
    return true;

    for (int c = 1; c <= m; c++)
    {
        //checking if this colour can be given to a particular node.
        if (isSafe(v, graph, color, c, V))
        {
            //assigning colour to the node.
            color[v] = c;

            //calling function recursively and checking if other nodes
            //can be assigned other colours.
            if (graphColoringUtil(graph, m, color, v + 1, V) == true)
                //returning true if the current allocation was successful.
                return true;

            //if not true, we backtrack and remove the colour for that node.
            color[v] = 0;
        }
    }
    //if no colour can be assigned, we return false.
    return false;
}

//Function to determine if graph can be coloured with at most M colours such
//that no two adjacent vertices of graph are coloured with same colour.
bool graphColoring(bool graph[101][101], int m, int V)
```

```
{
    int *color = new int[V];
    for (int i = 0; i < V; i++) color[i] = 0;

    //checking if colours can be assigned.
    if (graphColoringUtil(graph, m, color, 0, V) == false) {
        return false;
    }

    return true;
}
```

Complexity:

- Time: O(M^N)
- Space: O(N)

Question 9:

- **Problem link**: https://practice.geeksforgeeks.org/problems/alien-dictionary/1
- **Difficulty level**: Hard

Explanation:

The idea is to create a graph of characters and then find topological sorting of the created graph. Following are the detailed steps.

1) Create a graph *g* with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.

2) Do following for every pair of adjacent words in given sorted array.

.a) Let the current pair of words be *word1* and *word2*. One by one compare characters of both words and find the first mismatching characters.

b) Create an edge in *g* from mismatching character of *word1* to that of *word2*.

3) Print topological sorting of the above created graph.

Solution:

```
class graph {
  public:
    int V;
    list<int> *adj;

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

    void addedge(int v, int u) { adj[v].push_back(u); }
};
class Solution{
    public:
    void dictorder(string str1, string str2, graph *g, int *exist) {
        int n1 = str1.size();
        int n2 = str2.size();

        for (int i = 0; i < n1; i++) exist[(int)str1[i]] = 1;
        for (int i = 0; i < n2; i++) exist[(int)str2[i]] = 1;

        int i = 0;
        while (i < n1 && i < n2) {
            if (str1[i] != str2[i]) {
                g->addedge((int)str1[i], (int)str2[i]);
                return;
            }
            i++;
        }
    }
    void topsort(list<int> *adj, bool *visited, stack<char> &st, int v,
                 int *exist) {
        if (exist[v]) {
            visited[v] = true;
            for (auto u : adj[v])
                if (!visited[u]) topsort(adj, visited, st, u, exist);
            st.push((char)v);
        }
    }

    string findOrder(string dict[], int N, int K) {
        graph *g = new graph(256);
        int exist[256] = {0};
        for (int i = 1; i < N; i++) {
            dictorder(dict[i - 1], dict[i], g, exist);
        }

        bool visited[256] = {0};
        stack<char> st;
        for (int i = 0; i < 256; i++) {
            if (!visited[i]) topsort(g->adj, visited, st, i, exist);
        }

        string final = "";
        while (!st.empty()) {
            final += st.top();
            st.pop();
        }

        return final;
    }
};
```

Complexity:

- Time: O(N+K)
- Space: O(N)