

DOMAIN WINTER CAMP

(Department of Computer Science and Engineering)

Name: Aman Bansal UID: 22BCS13365 Section: KPIT 901-B

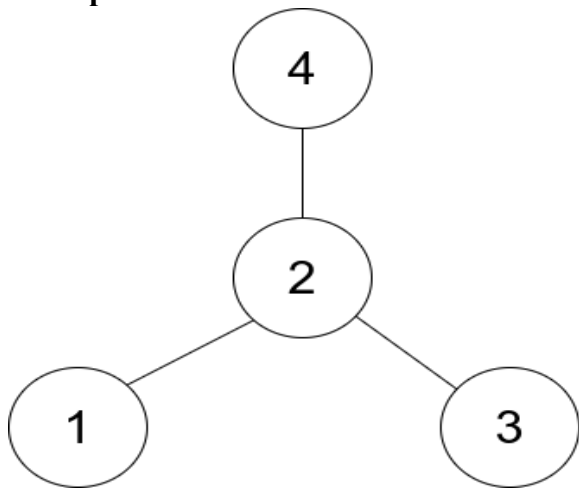
DAY-7

Q1. Find center of the graph

There is an undirected star graph consisting of n nodes labeled from 1 to n . A star graph is a graph where there is one center node and exactly $n - 1$ edges that connect the center node with every other node.

You are given a 2D integer array `edges` where each `edges[i] = [ui, vi]` indicates that there is an edge between the nodes `ui` and `vi`. Return the center of the given star graph.

Example 1:



Input: `edges = [[1,2],[2,3],[4,2]]`

Output: 2

Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

Example 2:**Input:** edges = [[1,2],[5,1],[1,3],[1,4]]**Output:** 1**Constraints:**

- $3 \leq n \leq 1e5$
- `edges.length == n - 1`
- `edges[i].length == 2`
- $1 \leq u_i, v_i \leq n$
- $u_i \neq v_i$
- The given edges represent a valid star graph.

Program code:

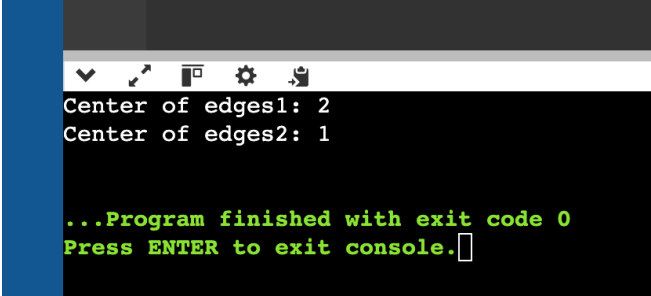
```
#include <iostream>
#include <vector>
using namespace std;

int findCenter(vector<vector<int>>& edges) {
    return (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) ? edges[0][0] :
    edges[0][1];
}

int main() {
    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}};

    cout << "Center of edges1: " << findCenter(edges1) << endl;
    cout << "Center of edges2: " << findCenter(edges2) << endl;

    return 0;
}
```

Output:A screenshot of a terminal window with a dark background and a blue vertical bar on the left. The terminal shows the output of the program: "Center of edges1: 2" and "Center of edges2: 1". Below this, it says "...Program finished with exit code 0" and "Press ENTER to exit console." with a cursor icon.

```
Center of edges1: 2
Center of edges2: 1

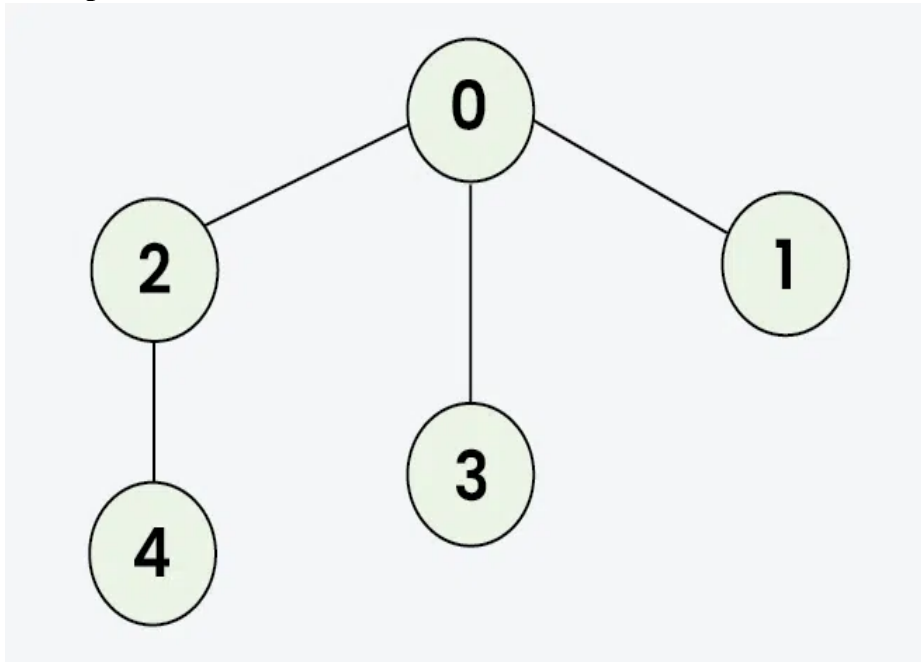
...Program finished with exit code 0
Press ENTER to exit console.
```

Q 2. BFS of graph

Given a connected undirected graph represented by an adjacency list `adj`, which is a vector of vectors where each `adj[i]` represents the list of vertices connected to vertex `i`. Perform a Breadth First Traversal (BFS) starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

Note: Do traverse in the same order as they are in the adjacency list.

Example 1:



Input: `adj = [[2,3,1], [0], [0,4], [0], [2]]`

Output: `[0, 2, 3, 1, 4]`

Explanation: Starting from 0, the BFS traversal will follow these steps:

Visit 0 → Output: 0

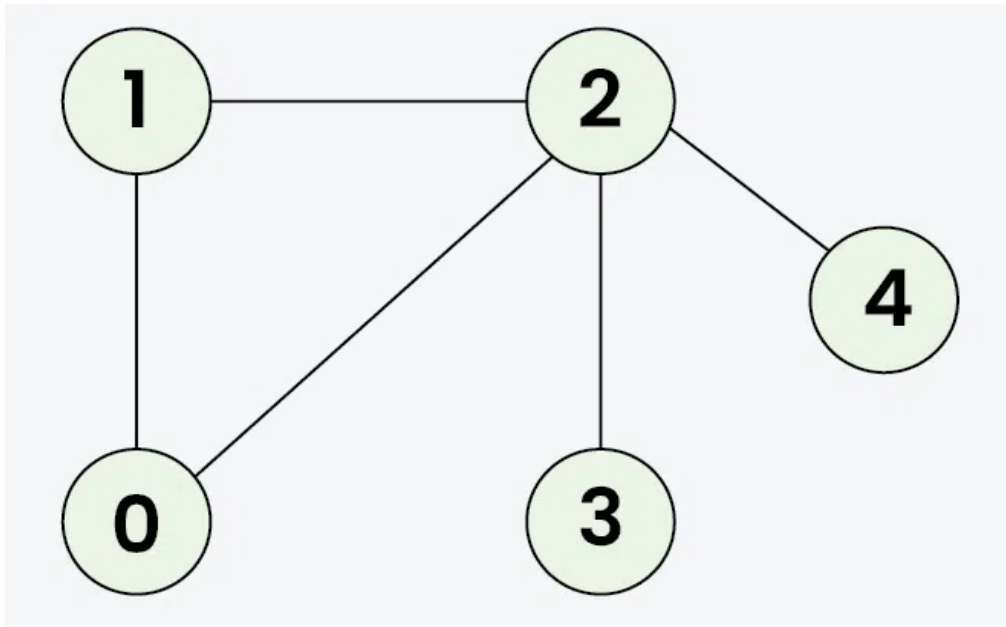
Visit 2 (first neighbor of 0) → Output: 0, 2

Visit 3 (next neighbor of 0) → Output: 0, 2, 3

Visit 1 (next neighbor of 0) → Output: 0, 2, 3,

Visit 4 (neighbor of 2) → Final Output: 0, 2, 3, 1, 4

Example 2



Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]

Output: [0, 1, 2, 3, 4]

Explanation: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 → Output: 0

Visit 1 (the first neighbor of 0) → Output: 0, 1

Visit 2 (the next neighbor of 0) → Output: 0, 1, 2

Visit 3 (the first neighbor of 2 that hasn't been visited yet) → Output: 0, 1, 2, 3

Visit 4 (the next neighbor of 2) → Final Output: 0, 1, 2, 3, 4

Input: adj = [[1], [0, 2, 3], [1], [1, 4], [3]]

Output: [0, 1, 2, 3, 4]

Explanation: Starting the BFS from vertex 0:

Visit vertex 0 → Output: [0]

Visit vertex 1 (first neighbor of 0) → Output: [0, 1]

Visit vertex 2 (first unvisited neighbor of 1) → Output: [0, 1, 2]

Visit vertex 3 (next neighbor of 1) → Output: [0, 1, 2, 3]

Visit vertex 4 (neighbor of 3) → Final Output: [0, 1, 2, 3, 4]

Constraints:

- $1 \leq \text{adj.size()} \leq 1e4$
- $1 \leq \text{adj}[i][j] \leq 1e4$

Program code:

```
#include <iostream>
```

```
#include <vector>
```

```
#include <queue>
```

```
using namespace std;
```

```
vector<int> bfsTraversal(int V, vector<vector<int>>& adj) {
```

```

vector<int> bfs;
vector<bool> visited(V, false);
queue<int> q;
q.push(0);
visited[0] = true;

while (!q.empty()) {
    int node = q.front();
    q.pop();
    bfs.push_back(node);

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
        }
    }
}
return bfs;
}

int main() {
    vector<vector<int>> adj1 = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<vector<int>> adj2 = {{1, 2}, {0, 2}, {0, 1, 3, 4}, {2}, {2}};
    vector<vector<int>> adj3 = {{1}, {0, 2, 3}, {1}, {1, 4}, {3}};

    vector<int> result1 = bfsTraversal(5, adj1);
    vector<int> result2 = bfsTraversal(5, adj2);
    vector<int> result3 = bfsTraversal(5, adj3);

    cout << "BFS Traversal of adj1: ";
    for (int node : result1) cout << node << " ";
    cout << endl;

    cout << "BFS Traversal of adj2: ";
    for (int node : result2) cout << node << " ";
    cout << endl;

    cout << "BFS Traversal of adj3: ";
    for (int node : result3) cout << node << " ";
}

```

```

    cout << endl;

    return 0;
}

```

Output:

```

51
BFS Traversal of adj1: 0 2 3 1 4
BFS Traversal of adj2: 0 1 2 3 4
BFS Traversal of adj3: 0 1 2 3 4

...Program finished with exit code 0
Press ENTER to exit console.

```

Q 3. Minimum height trees

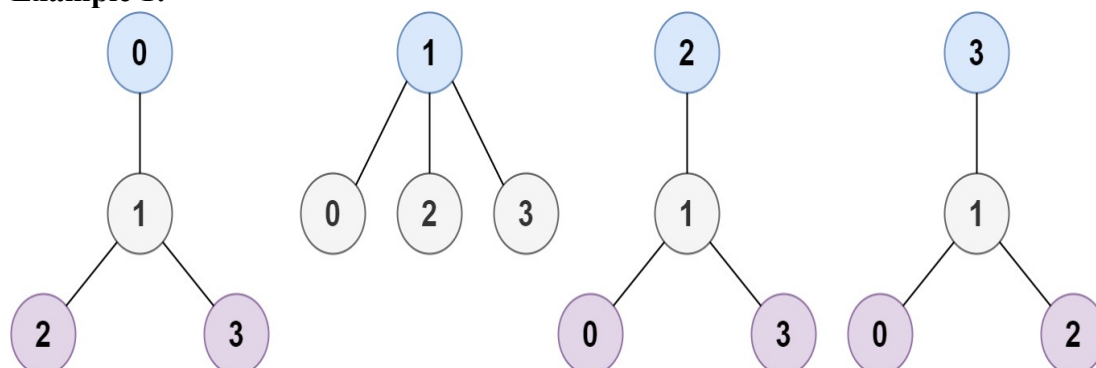
A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to $n - 1$, and an array of $n - 1$ edges where $\text{edges}[i] = [a_i, b_i]$ indicates that there is an undirected edge between the two nodes a_i and b_i in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h . Among all possible rooted trees, those with minimum height (i.e. $\min(h)$) are called minimum height trees (MHTs).

Return a list of all MHTs' root labels. You can return the answer in any order.

The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Example 1:

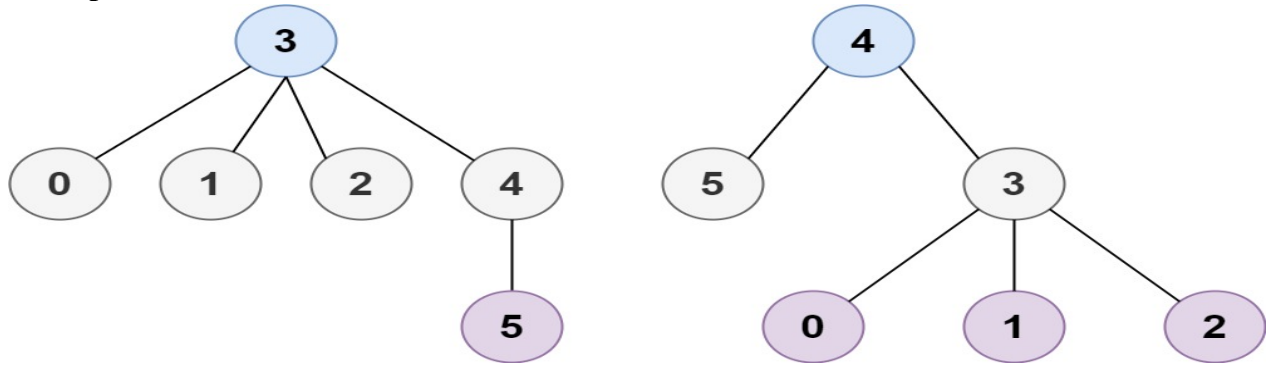


Input: n = 4, edges = [[1,0],[1,2],[1,3]]

Output: [1]

Explanation: As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

Example 2:



Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]

Output: [3,4]

Constraints:

- $1 \leq n \leq 2 \cdot 10^4$
- $\text{edges.length} == n - 1$
- $0 \leq a_i, b_i < n$
- $a_i \neq b_i$
- All the pairs (a_i, b_i) are distinct.
- The given input is guaranteed to be a tree and there will be no repeated edges.

Program Code:

```
#include <vector>
#include <queue>
#include <iostream>
using namespace std;

vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    if (n == 1) return {0};
    vector<int> degree(n, 0);
    vector<vector<int>> adj(n);
    for (auto& edge : edges) {
        adj[edge[0]].push_back(edge[1]);
        adj[edge[1]].push_back(edge[0]);
    }
```

```

        degree[edge[0]]++;
        degree[edge[1]]++;
    }
    queue<int> leaves;
    for (int i = 0; i < n; i++) {
        if (degree[i] == 1) leaves.push(i);
    }
    int remainingNodes = n;
    while (remainingNodes > 2) {
        int leafCount = leaves.size();
        remainingNodes -= leafCount;
        for (int i = 0; i < leafCount; i++) {
            int leaf = leaves.front();
            leaves.pop();
            for (int neighbor : adj[leaf]) {
                degree[neighbor]--;
                if (degree[neighbor] == 1) leaves.push(neighbor);
            }
        }
    }
    vector<int> result;
    while (!leaves.empty()) {
        result.push_back(leaves.front());
        leaves.pop();
    }
    return result;
}

int main() {
    vector<vector<int>> edges1 = {{1, 0}, {1, 2}, {1, 3}};
    vector<vector<int>> edges2 = {{3, 0}, {3, 1}, {3, 2}, {3, 4}, {5, 4}};

    vector<int> result1 = findMinHeightTrees(4, edges1);
    vector<int> result2 = findMinHeightTrees(6, edges2);

    cout << "MHT Roots for edges1: ";
    for (int root : result1) {
        cout << root << " ";
    }
    cout << endl;

    cout << "MHT Roots for edges2: ";

```



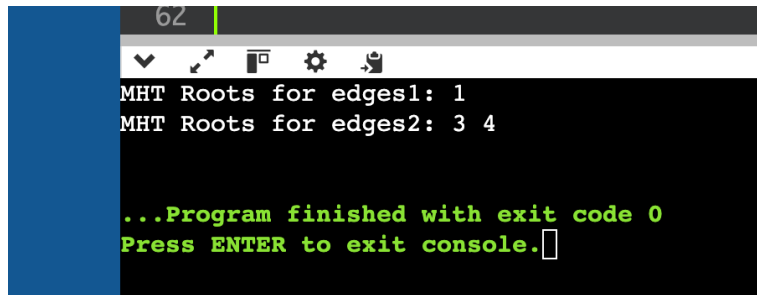
```

for (int root : result2) {
    cout << root << " ";
}
cout << endl;

return 0;
}

```

Output:



```

62
MHT Roots for edges1: 1
MHT Roots for edges2: 3 4

...Program finished with exit code 0
Press ENTER to exit console.

```

Q 4 Max area of island

You are given an $m \times n$ binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

Example 1:

0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0

Input: grid =

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],  
1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,  
0,0,0,0,1,1,0,0,0,0]]
```

Output: 6

Explanation: The answer is not 11, because the island must be connected 4-directionally.

Example 2:

Input: grid = [[0,0,0,0,0,0,0]]

Output: 0

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50
- grid[i][j] is either 0 or 1.

Program Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int dfs(vector<vector<int>>& grid, int i, int j) {
    if (i < 0 || j < 0 || i >= grid.size() || j >= grid[0].size() || grid[i][j] == 0)
        return 0;
    grid[i][j] = 0;
    return 1 + dfs(grid, i + 1, j) + dfs(grid, i - 1, j) + dfs(grid, i, j + 1) + dfs(grid, i, j - 1);
}

int maxAreaOfIsland(vector<vector<int>>& grid) {
    int maxArea = 0;
    for (int i = 0; i < grid.size(); i++) {
        for (int j = 0; j < grid[0].size(); j++) {
            if (grid[i][j] == 1) {
                maxArea = max(maxArea, dfs(grid, i, j));
            }
        }
    }
    return maxArea;
}
```

```

int main() {
    vector<vector<int>> grid = {
        {0,0,1,0,0,0,0,1,0,0,0,0},
        {0,0,0,0,0,0,0,1,1,0,0,0},
        {0,1,1,0,1,0,0,0,0,0,0,0},
        {0,1,0,0,1,1,0,0,1,0,1,0},
        {0,1,0,0,1,1,0,0,1,1,1,0},
        {0,0,0,0,0,0,0,0,0,0,1,0},
        {0,0,0,0,0,0,0,1,1,0,0,0},
        {0,0,0,0,0,0,0,1,1,0,0,0}
    };

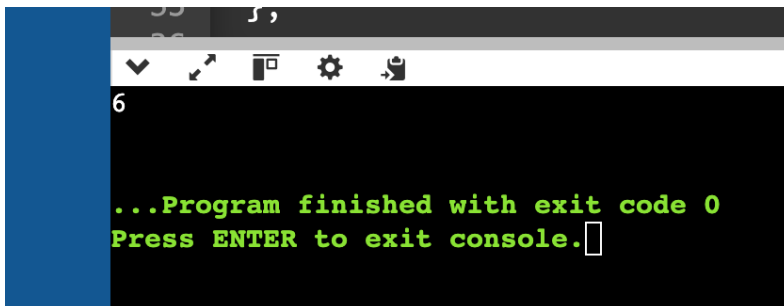
    int result = maxAreaOfIsland(grid);

    // Output the result for the single grid
    cout << result << endl;

    return 0;
}

```

Output:

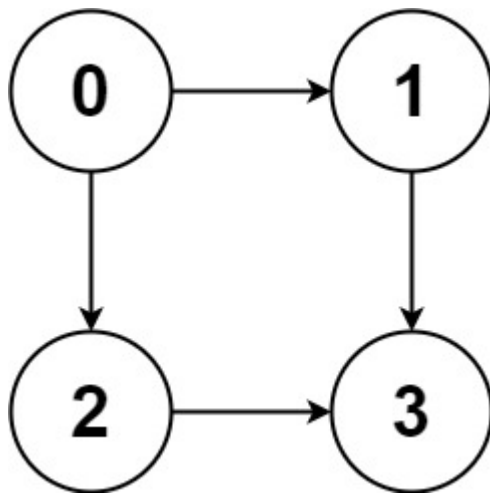


Q 5. All path from source to target

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to $n - 1$, find all possible paths from node 0 to node $n - 1$ and return them in any order.

The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node `graph[i][j]`).

Example 1:

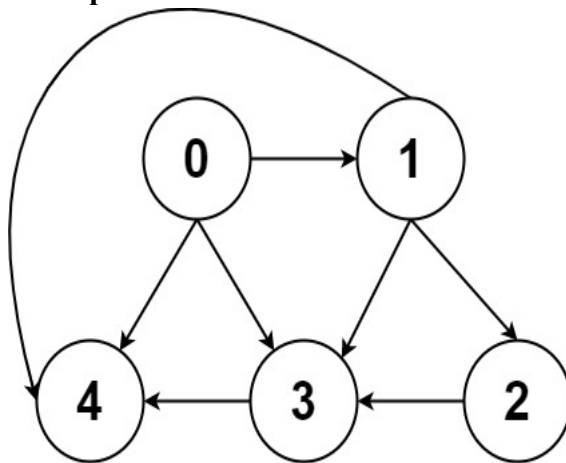


Input: graph = [[1,2],[3],[3],[]]

Output: [[0,1,3],[0,2,3]]

Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

Example 2:



Input: graph = [[4,3,1],[3,2,4],[3],[4],[]]

Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

Constraints:

- $n == \text{graph.length}$
- $2 \leq n \leq 15$
- $0 \leq \text{graph}[i][j] < n$
- $\text{graph}[i][j] \neq i$ (i.e., there will be no self-loops).
- All the elements of $\text{graph}[i]$ are unique.
- The input graph is guaranteed to be a DAG.

Program Code:

```

#include <iostream>
#include <vector>
using namespace std;

void dfs(int node, vector<vector<int>>& graph, vector<int>& path,
vector<vector<int>>& result) {
    path.push_back(node);
    if (node == graph.size() - 1) {
        result.push_back(path);
    } else {
        for (int neighbor : graph[node]) {
            dfs(neighbor, graph, path, result);
        }
    }
    path.pop_back();
}

vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
    vector<vector<int>> result;
    vector<int> path;
    dfs(0, graph, path, result);
    return result;
}

int main() {
    vector<vector<int>> graph1 = {{1,2},{3},{3},{}};
    vector<vector<int>> graph2 = {{4,3,1},{3,2,4},{3},{4},{}};

    vector<vector<int>> result1 = allPathsSourceTarget(graph1);
    vector<vector<int>> result2 = allPathsSourceTarget(graph2);

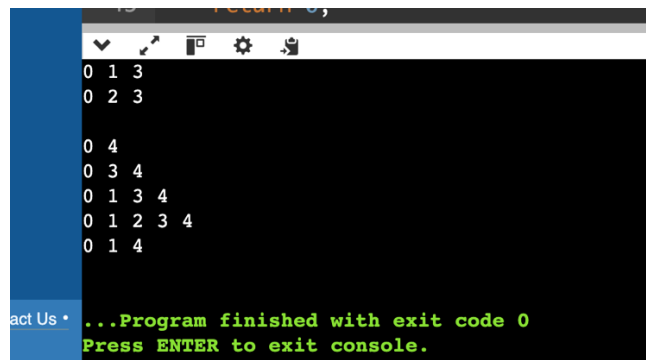
    for (const auto& path : result1) {
        for (int node : path) {
            cout << node << " ";
        }
        cout << endl;
    }
    cout << endl;
    for (const auto& path : result2) {
        for (int node : path) {
            cout << node << " ";
        }
    }
}

```

```
        cout << endl;
    }

    return 0;
}
```

Output:

A screenshot of a console window showing the output of a C++ program. The window has a title bar with standard icons. The output consists of several lines of numbers: "0 1 3", "0 2 3", "0 4", "0 3 4", "0 1 3 4", "0 1 2 3 4", and "0 1 4". At the bottom, a green message states "...Program finished with exit code 0" and "Press ENTER to exit console.".

```
0 1 3
0 2 3

0 4
0 3 4
0 1 3 4
0 1 2 3 4
0 1 4

...Program finished with exit code 0
Press ENTER to exit console.
```