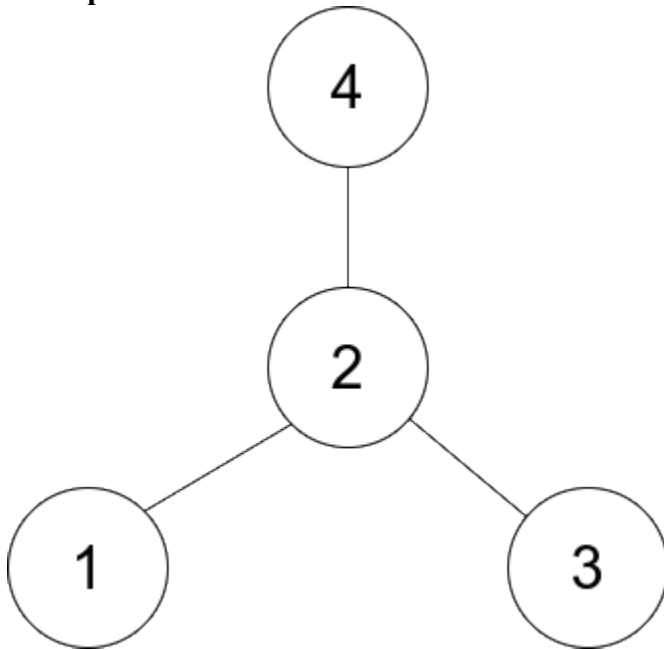


**DAY – 7****Find Center of Star Graph**

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 <= ui, vi <= n`
- `ui != vi`
- The given edges represent a valid star graph.

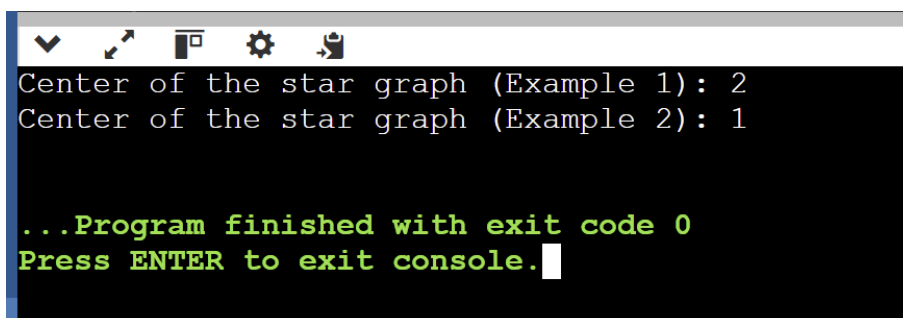
```
#include <iostream>
#include <vector>
using namespace std;
```

```
class Solution
{ public:
    int findCenter(vector<vector<int>>& edges) {
        // Check the first two edges to find the common node
        if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1])
            { return edges[0][0];
          }
        return edges[0][1];
    }
};
```

```
int main()
{ Solution
  solution;

  // Example 1: edges = [[1,2],[2,3],[4,2]]
  vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
  cout << "Center of the star graph (Example 1): " << solution.findCenter(edges1) << endl;
  // Output: 2

  // Example 2: edges = [[1,2],[5,1],[1,3],[1,4]]
  vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}
}
```



```
Center of the star graph (Example 1): 2
Center of the star graph (Example 2): 1

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

## **2Find the Town Judge**

In a town, there are  $n$  people labeled from 1 to  $n$ . There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given an array `trust` where `trust[i] = [ai, bi]` representing that the person labeled  $a_i$  trusts the person labeled  $b_i$ . If a trust relationship does not exist in `trust` array, then such a trust relationship does not exist.

Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.

**Example 1:**

**Input:**  $n = 2$ , `trust = [[1,2]]`

**Output:** 2

**Example 2:**

**Input:**  $n = 3$ , `trust = [[1,3],[2,3]]`

**Output:** 3

**Example 3:**



**Input:**  $n = 3$ ,  $\text{trust} = [[1,3],[2,3],[3,1]]$

**Output:** -1

**Constraints:**

- $1 \leq n \leq 1000$
- $0 \leq \text{trust.length} \leq 1e4$
- $\text{trust}[i].\text{length} == 2$
- All the pairs of trust are unique.
- $a_i \neq b_i$
- $1 \leq a_i, b_i \leq n$

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

```
class Solution
{ public:
    int findJudge(int n, vector<vector<int>>& trust) {
        // Create a trustCounts array initialized to 0
        vector<int> trustCounts(n + 1, 0);

        // Update trustCounts based on trust relationships
        for (const auto& t : trust) {
            int a = t[0], b = t[1];
            trustCounts[a]--; // Person a trusts someone
            trustCounts[b]++; // Person b is trusted by someone
        }

        // Find the town judge
        for (int i = 1; i <= n; ++i) {
            if (trustCounts[i] == n - 1)
                { return i; // Judge found
            }
        }
    }
}
```

```

        return -1; // No judge exists
    }
};

int main()
{
    Solution
    solution;

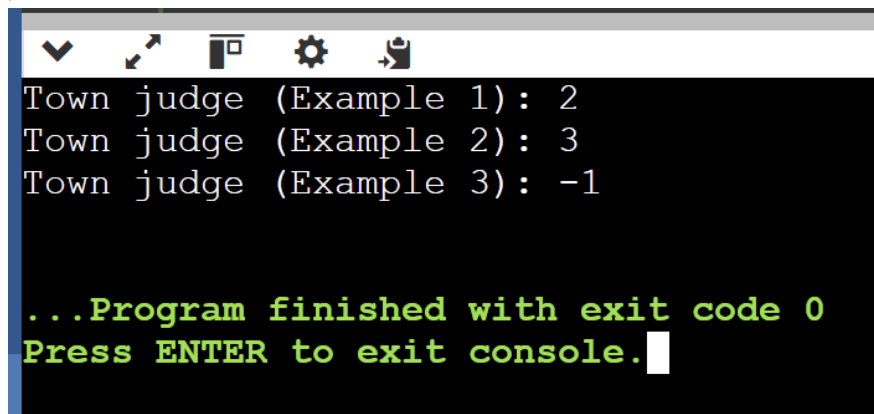
    // Example 1: n = 2, trust = [[1, 2]]
    int n1 = 2;
    vector<vector<int>> trust1 = {{1, 2}};
    cout << "Town judge (Example 1): " << solution.findJudge(n1, trust1) << endl;
    // Output: 2

    // Example 2: n = 3, trust = [[1, 3], [2, 3]]
    int n2 = 3;
    vector<vector<int>> trust2 = {{1, 3}, {2, 3}};
    cout << "Town judge (Example 2): " << solution.findJudge(n2, trust2) << endl;
    // Output: 3

    // Example 3: n = 3, trust = [[1, 3], [2, 3], [3, 1]]
    int n3 = 3;
    vector<vector<int>> trust3 = {{1, 3}, {2, 3}, {3, 1}};
    cout << "Town judge (Example 3): " << solution.findJudge(n3, trust3) << endl;
    // Output: -1

    return 0;
}

```



```

Town judge (Example 1): 2
Town judge (Example 2): 3
Town judge (Example 3): -1

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

```

### 3. Flood Fill - [link](#)

You are given an image represented by an  $m \times n$  grid of integers `image`, where `image[i][j]` represents the pixel value of the image. You are also given three integers `sr`, `sc`, and `color`. Your task is to perform a flood fill on the image starting from the pixel `image[sr][sc]`.

**To perform a flood fill:**





**Begin with the starting pixel and change its color to color.**

**Perform the same process for each pixel that is directly adjacent (pixels that share a side with the original pixel, either horizontally or vertically) and shares the same color as the starting pixel.**

**Keep repeating this process by checking neighboring pixels of the updated pixels and modifying their color if it matches the original color of the starting pixel.**

**The process stops when there are no more adjacent pixels of the original color to update.**

**Return the modified image after performing the flood fill.**

**Example 1:**

**Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2**

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

**Explanation:**

**From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.**

**Note the bottom corner is not colored 2, because it is not horizontally or vertically connected to the starting pixel.**

**Example 2:**

**Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0**

**Output: [[0,0,0],[0,0,0]]**

**Explanation:**

The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

**Constraints:**

- $m == \text{image.length}$
- $n == \text{image}[i].\text{length}$
- $1 \leq m, n \leq 50$
- $0 \leq \text{image}[i][j], \text{color} < 2^{16}$
- $0 \leq \text{sr} < m$
- $0 \leq \text{sc} < n$

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

```
class Solution
```

```
{ public:
```

```
void dfs(vector<vector<int>>& image, int sr, int sc, int newColor, int originalColor) {
    // Check for boundaries and if the pixel color is not the original color
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() || image[sr][sc] != originalColor)
    { return;
    }
}
```

```
    // Change the current pixel's color to the new color image[sr]
    [sc] = newColor;
```

```
    // Recursively visit all four adjacent pixels
```

```
    dfs(image, sr + 1, sc, newColor, originalColor); // Down
```

```
    dfs(image, sr - 1, sc, newColor, originalColor); // Up
```

```
    dfs(image, sr, sc + 1, newColor, originalColor); // Right
```

```
    dfs(image, sr, sc - 1, newColor, originalColor); // Left
```

```
}
```

```
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color)
```

```
{ int originalColor = image[sr][sc];
```

```
    // If the starting pixel's color is already the target color, return the image as-is
```

```
if (originalColor != color) {
```

```

        dfs(image, sr, sc, color, originalColor);
    }
    return image;
}
};

```

```

int main()
{
    Solution
    solution;

    // Example 1
    vector<vector<int>> image1 = {{1, 1, 1}, {1, 1, 0}, {1, 0, 1}};
    int sr1 = 1, sc1 = 1, color1 = 2;
    vector<vector<int>> result1 = solution.floodFill(image1, sr1, sc1, color1);


    cout << "Example 1 Output: " << endl;
    for (const auto& row : result1) {
        for (int pixel : row)
            { cout << pixel << "
              ";
            }
        cout << endl;
    }

    // Example 2
    vector<vector<int>> image2 = {{0, 0, 0}, {0, 0, 0}};
    int sr2 = 0, sc2 = 0, color2 = 0;
    vector<vector<int>> result2 = solution.floodFill(image2, sr2, sc2, color2);

    cout << "Example 2 Output: " << endl;
    for (const auto& row : result2) {
        for (int pixel : row)
            { cout << pixel << "
              ";
            }
        cout << endl;
    }

    return 0;
}

```



```
2 2 2
2 2 0
2 0 1
Example 2 Output:
0 0 0
0 0 0

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

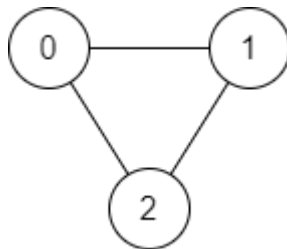
#### 4 [Find if Path Exists in Graph](#)

There is a bi-directional graph with  $n$  vertices, where each vertex is labeled from  $0$  to  $n - 1$  (inclusive). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [ui, vi]` denotes a bi-directional edge between vertex `ui` and vertex `vi`. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` if there is a valid path from `source` to `destination`, or `false` otherwise.

**Example 1:**



**Input:** `n = 3`, `edges = [[0,1],[1,2],[2,0]]`, `source = 0`, `destination = 2`

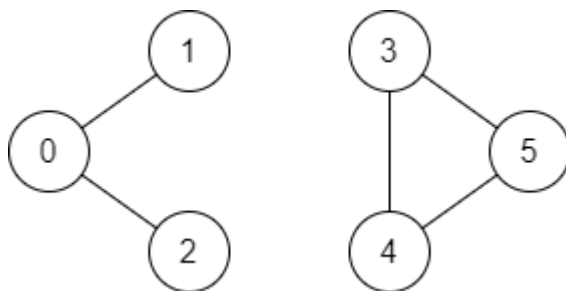
**Output:** `true`

**Explanation:** There are two paths from vertex `0` to vertex `2`:

- `0 → 1 → 2`

- `0 → 2`

**Example 2:**



**Input:** `n = 6`, `edges = [[0,1],[0,2],[3,5],[5,4],[4,3]]`, `source = 0`, `destination = 5`

**Output:** `false`

**Explanation:** There is no path from vertex `0` to vertex `5`.

**Constraints:**

- $1 \leq n \leq 2 * 10^5$
- $0 \leq \text{edges.length} \leq 2 * 10^5$
- `edges[i].length == 2`

- $0 \leq u_i, v_i \leq n - 1$



- **ui != vi**
- **0 <= source, destination <= n - 1**
- **There are no duplicate edges.**
- **There are no self edges.**

```
##include <iostream>
```

```
#include <vector>
```

```
#include <queue>
```

```
#include <unordered_map>
```

```
#include <unordered_set>
```

```
using namespace std;
```

```
class Solution
```

```
{ public:
```

```
    bool validPath(int n, vector<vector<int>>& edges, int  
source, int destination) {
```

```
        if (source == destination) return true; // If source is the  
same as destination, path exists
```

```
    // Build an adjacency list for the graph
```

```
    unordered_map<int, vector<int>> graph;
```

```
    for (const auto& edge : edges) {
```

```
        graph[edge[0]].push_back(edge[1]);
```

```
        graph[edge[1]].push_back(edge[0]);
```

```
    }
```

```

// Use BFS to find the path

queue<int> q;

unordered_set<int> visited;


q.push(source);

visited.insert(source);


while (!q.empty())

    { int node =

    q.front(); q.pop();

    for (int neighbor : graph[node]) {

        if (neighbor == destination) return true; // If we
reach the destination, return true

        if (visited.find(neighbor) == visited.end()) {

            visited.insert(neighbor);

            q.push(neighbor);

        }

    }

}

return false; // No valid path found

}

```



```

int main()

{ Solution

solution;


// Example 1

int n1 = 3;

vector<vector<int>> edges1 = {{0, 1}, {1, 2}, {2, 0}};

int source1 = 0, destination1 = 2;

cout << (solution.validPath(n1, edges1, source1,
destination1) ? "true" : "false") << endl;


// Example 2

int n2 = 6;

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

int source2 = 0, destination2 = 5;

cout << (solution.validPath(n2, edges2, source2,
destination2) ? "true" : "false") << endl;


return 0;

}

```

A terminal window with a dark background and a light gray title bar. The title bar contains four icons: a checkmark, a cursor, a window, and a gear. The terminal displays the text 'true' and 'false' on two separate lines. Below these, a green message reads: '...Program finished with exit code 0' followed by 'Press ENTER to exit console.' with a white cursor block at the end.

```
true
false

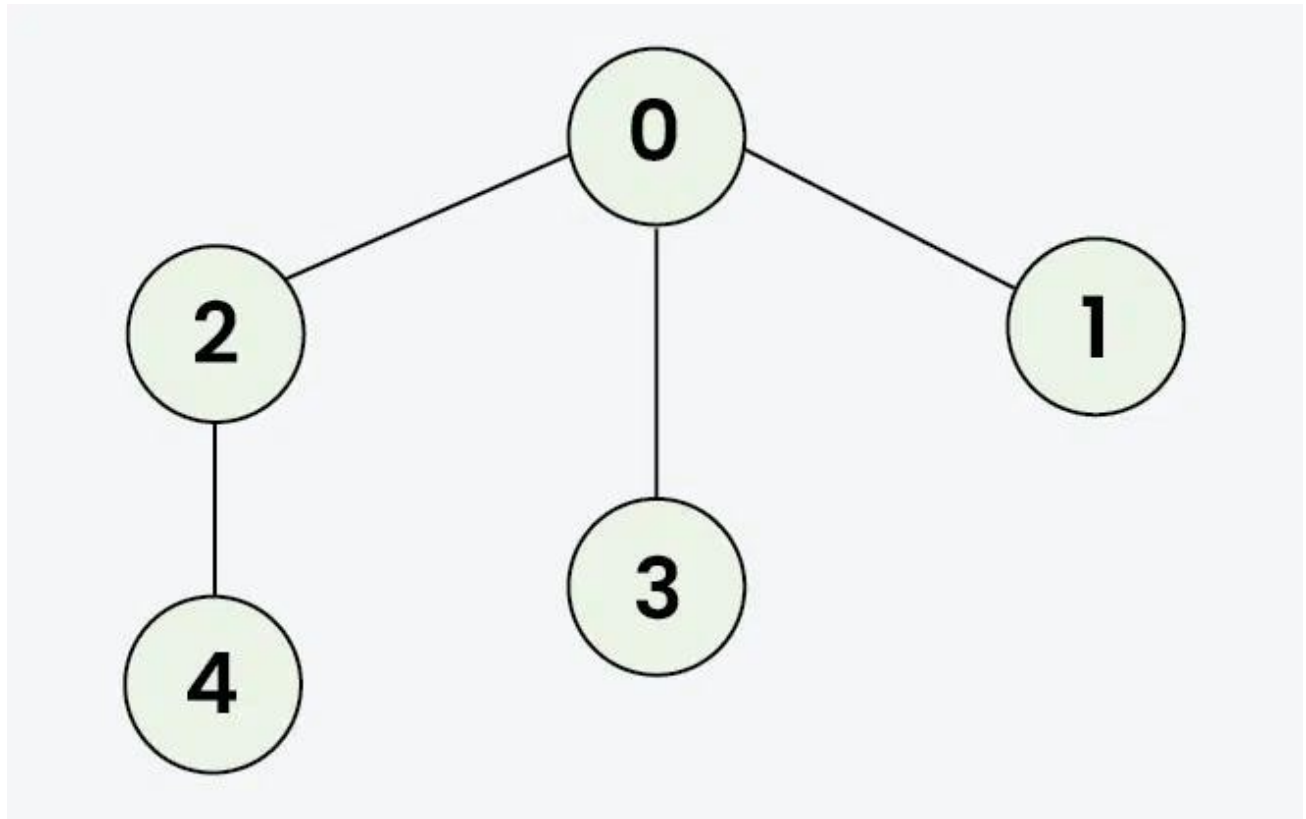
...Program finished with exit code 0
Press ENTER to exit console.
```

### 5. BFS of graph [link](#)

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

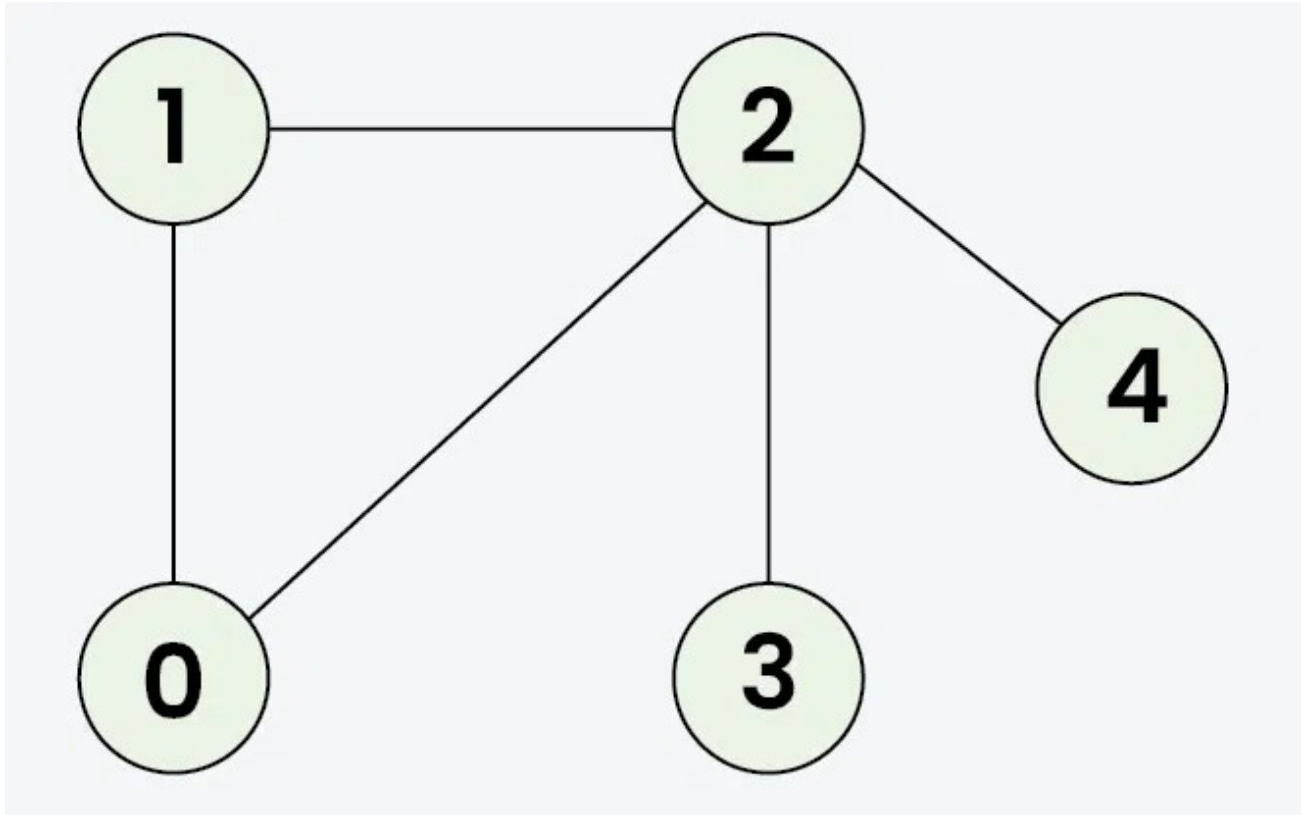
**Visit 2** (first neighbor of 0) → **Output:** 0, 2

**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:**  $\text{adj} = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]$

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

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

**Visit 0**  $\rightarrow$  **Output:** 0

**Visit 1** (the first neighbor of 0)  $\rightarrow$  **Output:** 0, 1

**Visit 2** (the next neighbor of 0)  $\rightarrow$  **Output:** 0, 1, 2

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

**Visit 4** (the next neighbor of 2)  $\rightarrow$  **Final Output:** 0, 1, 2, 3, 4

**Input:**  $\text{adj} = [[1], [0, 2, 3], [1], [1, 4], [3]]$

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

**Explanation:** Starting the BFS from vertex 0:

**Visit vertex 0**  $\rightarrow$  **Output:** [0]

**Visit vertex 1** (first neighbor of 0)  $\rightarrow$  **Output:** [0, 1]

**Visit vertex 2** (first unvisited neighbor of 1)  $\rightarrow$  **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:**

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

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

class Solution
{ public:
    vector<int> bfsOfGraph(int V, vector<vector<int>>& adj)
    { vector<int> bfs;          // Stores the BFS traversal result
      vector<bool> visited(V, false); // To keep track of visited nodes
      queue<int> q;             // Queue for BFS traversal

      // Start BFS from vertex 0
      q.push(0);
      visited[0] = true;

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

        // Visit all the neighbors of the current node
        for (int neighbor : adj[node]) {
            if (!visited[neighbor])
            { q.push(neighbor);
              visited[neighbor] = true;
            }
        }
      }
      return bfs;
    }
};

int main() {
    // Example 1
    Solution solution;
    vector<vector<int>> adj1 = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
```

```
vector<int> result1 = solution.bfsOfGraph(5, adj1);
```

```
cout << "Example 1 Output: ";
```

```

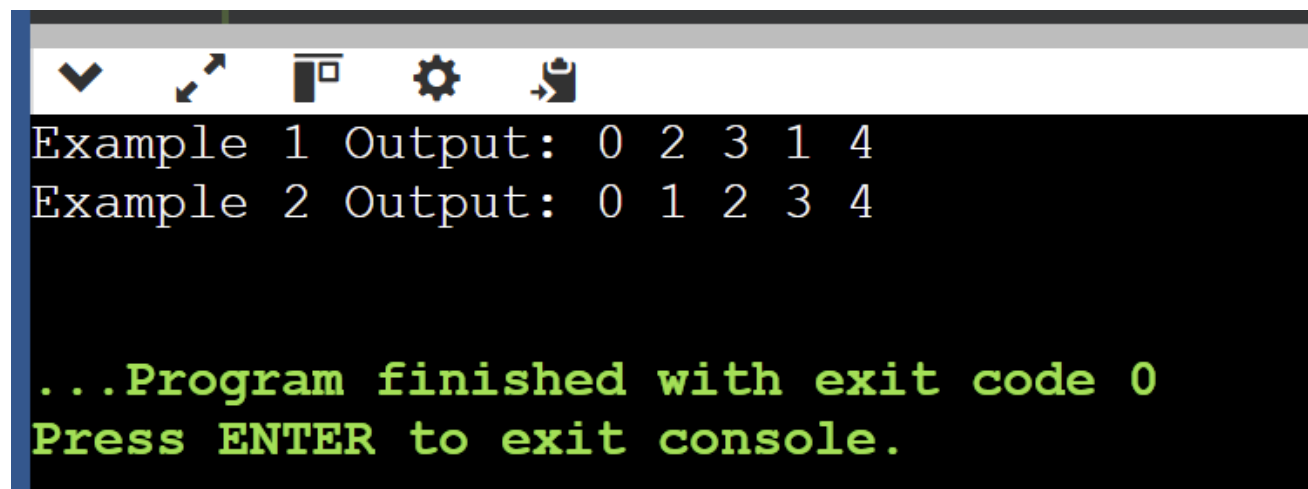
for (int node : result1)
    { cout << node << " ";
    }
cout << endl;

// Example 2
vector<vector<int>> adj2 = {{1, 2}, {0, 2}, {0, 1, 3, 4}, {2}, {2}};
vector<int> result2 = solution.bfsOfGraph(5, adj2);

cout << "Example 2 Output: ";
for (int node : result2) {
    cout << node << " ";
}
cout << endl;

return 0;
}

```



The screenshot shows a terminal window with a dark background. At the top, there is a toolbar with icons for a dropdown menu, a cursor, a window, a gear, and a document. The terminal output is as follows:

```

Example 1 Output: 0 2 3 1 4
Example 2 Output: 0 1 2 3 4

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

```

6. We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

**Example 1:**

**Input:** asteroids = [5,10,-5]

**Output:** [5,10]

**Explanation:** The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

**Example 2:**

**Input:** asteroids = [8,-8]

**Output:** []

**Explanation:** The 8 and -8 collide exploding each other.

**Example 3:**

**Input:** asteroids = [10,2,-5]

**Output:** [10]

```
#include <iostream>
```

```
#include <vector>
```

```
#include <stack>
```

```
using namespace std;
```

```
vector<int> asteroidCollision(vector<int>& asteroids)
```

```
{ stack<int> st; // Stack to simulate asteroid collisions
```

```
for (int asteroid : asteroids)
```

```
{ bool destroyed = false;
```

```
while (!st.empty() && asteroid < 0 && st.top() > 0)
```

```
{ if (abs(asteroid) == st.top()) {
```

```
    st.pop(); // Both asteroids explode
```

```
    destroyed = true;
```

```
    break;
```

```
} else if (abs(asteroid) > st.top()) {
```

```
    st.pop(); // Top asteroid is smaller and explodes
```

```
} else {
```

```
    destroyed = true; // Current asteroid is smaller and explodes
```

```
    break;
```

```
}
```

```
}
```

```
if (!destroyed) st.push(asteroid); // Push the current asteroid if not destroyed
```

```
}
```

```

vector<int> result(st.size());

for (int i = st.size() - 1; i >= 0; i--) {

    result[i] = st.top(); // Transfer stack to result in reverse order

    st.pop();

}

return result;

}

int main() {

    vector<int> asteroids1 = {5, 10, -5};

    vector<int> asteroids2 = {8, -8};

    vector<int> asteroids3 = {10, 2, -5};

    vector<int> result1 = asteroidCollision(asteroids1);

    vector<int> result2 = asteroidCollision(asteroids2);

    vector<int> result3 = asteroidCollision(asteroids3);

    for (int x : result1) cout << x << " "; // Output: 5 10

    cout << endl;

    for (int x : result2) cout << x << " "; // Output: (empty)

    cout << endl;

```

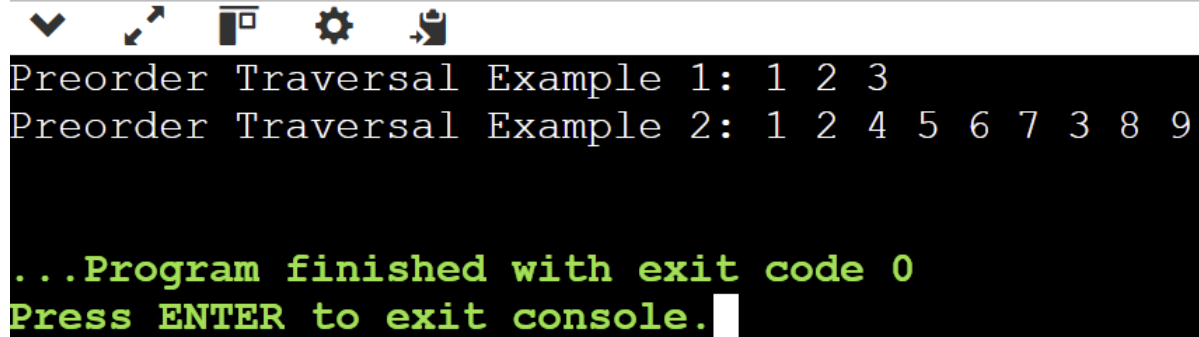
```

for (int x : result3) cout << x << " "; // Output: 10

cout << endl;

return 0;
}

```



```

Preorder Traversal Example 1: 1 2 3
Preorder Traversal Example 2: 1 2 4 5 6 7 3 8 9

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

```

### 8 Binary Tree - Sum of All Nodes

Given the root of a binary tree, you need to find the sum of all the node values in the binary tree.

**Example 1:**

**Input:** root = [1, 2, 3, 4, 5, null, 6]

**Output:** 21

**Explanation:** The sum of all nodes is  $1 + 2 + 3 + 4 + 5 + 6 = 21$ .

**Example 2:**

**Input:** root = [5, 2, 6, 1, 3, 4, 7]

**Output:** 28

**Explanation:** The sum of all nodes is  $5 + 2 + 6 + 1 + 3 + 4 + 7 = 28$ .

**Reference:** <http://leetcode.com/problems/sum-of-left-leaves/>



```
#include <iostream>
```

```
using namespace std;
```

```
// Definition for a binary tree node.
```

```
struct TreeNode {
```

```
    int val;
```

```
    TreeNode* left;
```

```
    TreeNode* right;
```

```
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
```

```
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
```

```
};
```

```
class Solution
```

```
{ public:
```

```
    int sumOfNodes(TreeNode* root) {
```

```
        if (!root) return 0; // Base case: if the tree is empty, sum is 0
```

```
        return root->val + sumOfNodes(root->left) + sumOfNodes(root->right);
```

```
    }
```

```
};
```

```
int main() {
```

```
    // Example 1: [1, 2, 3, 4, 5, null, 6]
```

```
    TreeNode* root1 = new TreeNode(1);
```

```
root1->left = new TreeNode(2);
```

```
root1->right = new TreeNode(3);  
root1->left->left = new TreeNode(4);  
root1->left->right = new TreeNode(5);  
root1->right->right = new TreeNode(6);
```

Solution solution;

```
cout << "Sum of all nodes (Example 1): " << solution.sumOfNodes(root1) << endl;  
  
// Output: 21
```

```
// Example 2: [5, 2, 6, 1, 3, 4, 7]
```

```
TreeNode* root2 = new TreeNode(5);  
root2->left = new TreeNode(2);  
root2->right = new TreeNode(6);  
root2->left->left = new TreeNode(1);  
root2->left->right = new TreeNode(3);  
root2->right->left = new TreeNode(4);  
root2->right->right = new TreeNode(7);
```

```
cout << "Sum of all nodes (Example 2): " << solution.sumOfNodes(root2) << endl;  
  
// Output: 28
```

```
return 0;
```

```
Sum of all nodes (Example 1): 21
Sum of all nodes (Example 2): 28

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

## 10. Same Tree

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

**Example 1:**

**Input:** p = [1,2,3], q = [1,2,3]

**Output:** true

**Example 2:**

**Input:** p = [1,2], q = [1,null,2]

**Output:** false

**Constraints:**

The number of nodes in both trees is in the range [0, 100].

-104 <= Node.val <= 104

**Reference:** <https://leetcode.com/problems/same-tree/description/?envType=study-plan-v2&envId=top-interview-150>

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

```
// Definition for a binary tree node.
```

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
```

```
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
```

```
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
```

};

```

class Solution
{ public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        // If both trees are empty, they are the same
        if (!p && !q) return true;
        // If one is empty and the other is not, they are not the same
        if (!p || !q) return false;
        // If the values of the current nodes are different, they are not the same
        if (p->val != q->val) return false;

        // Recursively check the left and right subtrees
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};

int main() {
    // Example 1: p = [1, 2, 3], q = [1, 2, 3]
    TreeNode* p1 = new TreeNode(1);
    p1->left = new TreeNode(2);
    p1->right = new TreeNode(3);

    TreeNode* q1 = new TreeNode(1);
    q1->left = new TreeNode(2);
    q1->right = new TreeNode(3);

    Solution solution;
    cout << "Are trees p1 and q1 the same? " << (solution.isSameTree(p1, q1) ? "true" : "false") <<
endl;
    // Output: true

    // Example 2: p = [1, 2], q = [1, null, 2]
    TreeNode* p2 = new TreeNode(1);
    p2->left = new TreeNode(2);

    TreeNode* q2 = new TreeNode(1);
    q2->right = new TreeNode(2);

    cout << "Are trees p2 and q2 the same? " << (solution.isSameTree(p2, q2) ? "true" : "false") <<
endl;
    // Output: false

    return 0;
}

```

```
2 -1 2
```

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