UNIT–III – Syllabus

Breadth First Traversal and Depth First Traversal: *BFS Introduction, Applications: Find All the Lonely Nodes, Max Area of Island, Number of Distinct Islands. DFS Introduction, Applications: The Maze, Boundary of Binary Tree.*

Trees: *Binary Tree Introduction, Applications: Symmetric Tree, Balanced Binary Tree, Average of Levels in Binary Tree, Find Largest Value in Each Tree Row, Binary Tree Right Side View.*

Backtracking: *General method, Applications: N Queens Problem, Hamiltonian Cycle, Brace Expansion, Gray Code, Path with Maximum Gold, Generalized Abbreviation, Campus Bikes II.*

BFS(Breadth-First Search) Algorithm

Breadth First Search traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

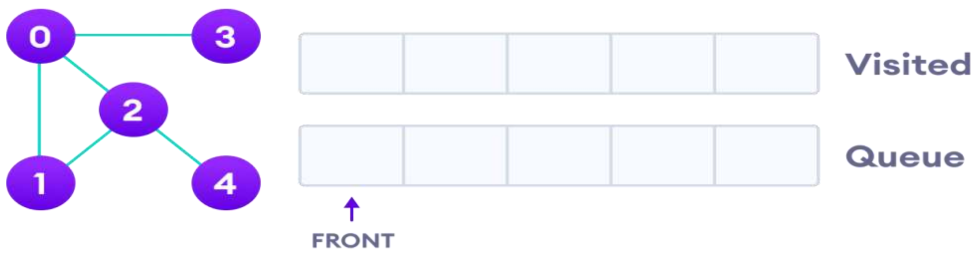
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

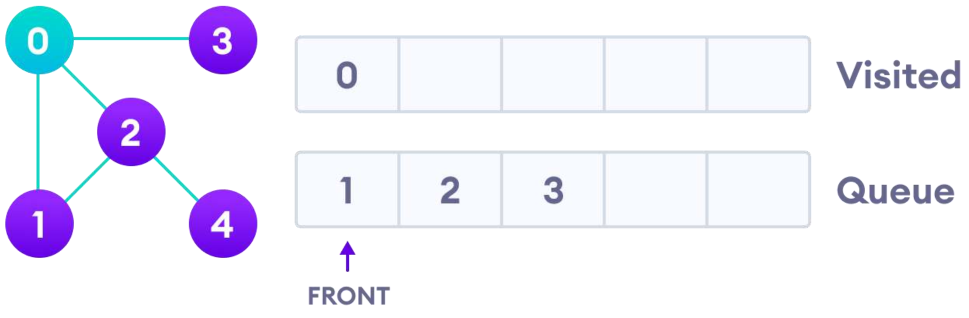
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



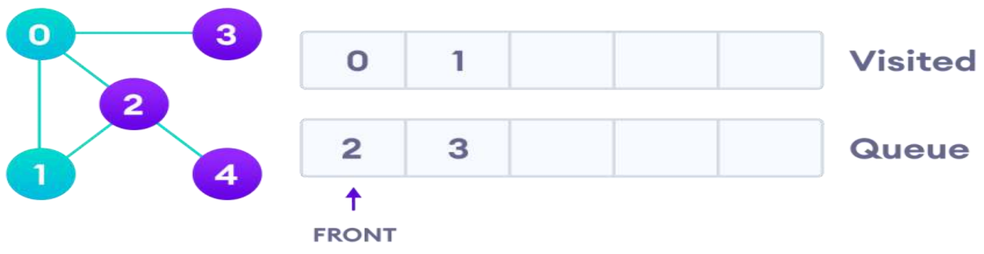
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the Queue.



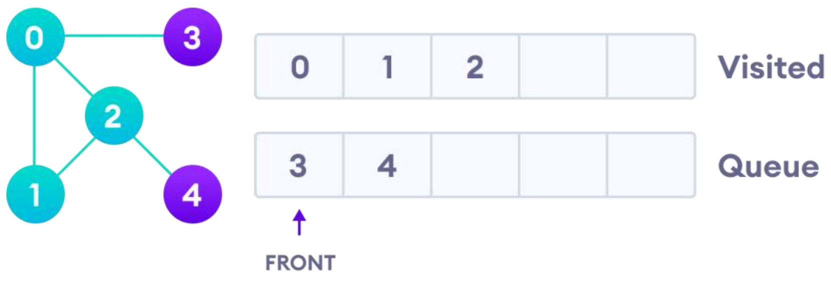
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e., 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

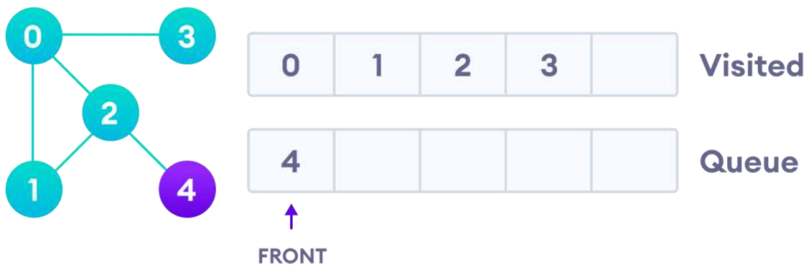


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

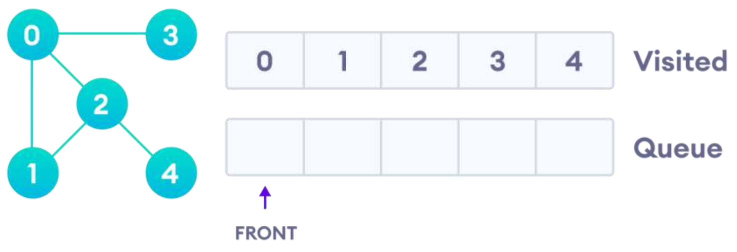


Visit 2 which was added to queue earlier to add its neighbors



4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e., 0 is already visited. We visit it.



Visit last remaining item in the queue to check if it has unvisited neighbours

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

Rules of BFS Algorithm

Here, are important rules for using BFS algorithm:

- A queue (FIFO-First in First Out) data structure is used by BFS.
- You mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
- Removes the previous vertex from the queue in case no adjacent vertex is found.
- BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.
- There are no loops caused by BFS during the traversing of data from any node.

Applications of BFS Algorithm

Let's take a look at some of the real-life applications where a BFS algorithm implementation can be highly effective.

- **Un-weighted Graphs:** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.
- **P2P Networks:** BFS can be implemented to locate all the nearest or neighbouring nodes in a peer-to-peer network. This will find the required data faster.
- **Web Crawlers:** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.
- **Navigation Systems:** BFS can help find all the neighbouring locations from the main or source location.
- **Network Broadcasting:** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

Summary

- A graph traversal is a unique process that requires the algorithm to visit, check, and/or update every single un-visited node in a tree-like structure. BFS algorithm works on a similar principle.
- The algorithm is useful for analysing the nodes in a graph and constructing the shortest path of traversing through these.

- The algorithm traverses the graph in the smallest number of iterations and the shortest possible time.
- BFS selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. BFS accesses these nodes one by one.
- The visited and marked data is placed in a queue by BFS. A queue works on a first in first out basis. Hence, the element placed in the graph first is deleted first and printed as a result.
- The BFS algorithm can never get caught in an infinite loop.
- Due to high precision and robust implementation, BFS is used in multiple real-life solutions like P2P networks, Web Crawlers, and Network Broadcasting.

BFS Traversal Program

```
import java.util.*;
class Graph
{
	private int Num Vertices;
	private LinkedList<Integer>[] adjList;

	public Graph(int numVertices)
	{
		this.numVertices = numVertices;
		adjList = new LinkedList[numVertices];
		for (int i = 0; i < numVertices; i++)
		{
			adjList[i] = new LinkedList<>();
		}
	}
	public void addEdge(int source, int destination)
	{
		adjList[source].add(destination);
	}
	public void BFS(int startVertex)
	{
		boolean[] visited = new boolean[numVertices];
		Queue<Integer> queue = new LinkedList<>();

		visited[startVertex] = true;
		queue.add(startVertex);

		while (!queue.isEmpty())
		{
			int currentVertex = queue.poll();
			System.out.print(currentVertex + " ");

			for (int neighbor : adjList[currentVertex])
			{
				if (!visited[neighbor])
				{
					visited[neighbor] = true;
					queue.add(neighbor);
```

```
                        }
                    }
                }
        }
}

class Main
{
        public static void main(String[] args)
        {
                Graph graph = new Graph(5);
                graph.addEdge(0, 1);
                graph.addEdge(0, 4);
                graph.addEdge(1, 0);
                graph.addEdge(1, 2);
                graph.addEdge(1, 3);
                graph.addEdge(1, 4);
                graph.addEdge(2, 3);
                graph.addEdge(2, 1);
                graph.addEdge(3, 1);
                graph.addEdge(3, 2);
                graph.addEdge(3, 4);
                graph.addEdge(4, 1);
                graph.addEdge(4, 0);
                graph.addEdge(4, 3);

                System.out.println("Breadth-First Traversal (starting from vertex 0):");
                graph.BFS(0);
        }
}
```

Output:

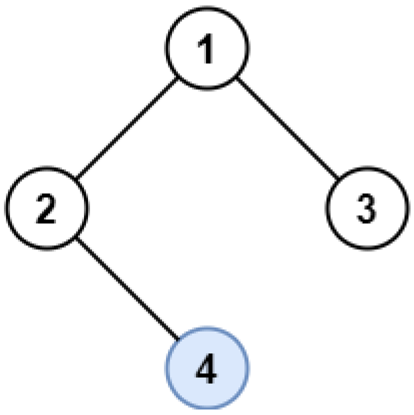
```
Breadth-First Traversal (starting from vertex 0):
0 1 4 2 3
```

1. Find All the Lonely Nodes

In a binary tree, a **lonely** node is a node that is the only child of its parent node. The root of the tree is not lonely because it does not have a parent node.

Given the **root** of a binary tree, return *an array containing the values of all lonely nodes* in the tree. Return the list **in any order**.

Example 1:



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

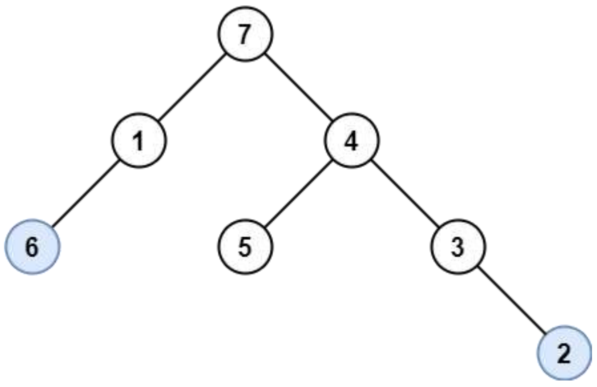
Output: [4]

Explanation: Light blue node is the only lonely node.

Node 1 is the root and is not lonely.

Nodes 2 and 3 have the same parent and are not lonely.

Example 2:



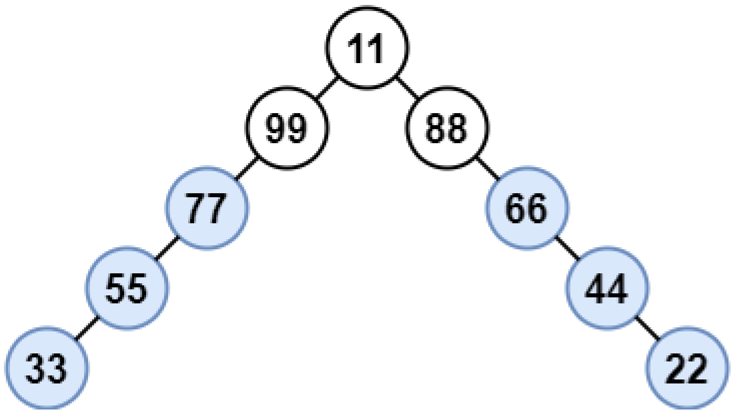
Input: root = [7,1,4,6,null,5,3,null,null,null,null,null,2]

Output: [6,2]

Explanation: Light blue nodes are lonely nodes.

Please remember that order doesn't matter, [2,6] is also an acceptable answer.

Example 3:



Input: root = [11,99,88,77,null,null,66,55,null,null,44,33,null,null,22]

Output: [77,55,33,66,44,22]

Explanation: Nodes 99 and 88 share the same parent. Node 11 is the root.

All other nodes are lonely.

Example 4:

Input: root = [197]

Output: []

Example 5:

Input: root = [31,null,78,null,28]

Output: [78,28]

PROGRAM:

```
import java.util.*;
class BinaryTreeNode
{
        public int data;
        public BinaryTreeNode left, right;
        public BinaryTreeNode(int data)
        {
                this.data = data;
                left = null;
                right = null;
        }
}
```

```java
class Solution
{
	public ArrayList<Integer> getPersonIDs(BinaryTreeNode root)
	{
		ArrayList<Integer> nodes = new ArrayList<>();
		getPersonIDs(root, false, nodes); // root is not lonely
		return nodes;
	}
	private void getPersonIDs(BinaryTreeNode root, boolean isLonely, ArrayList<Integer> nodes)
	{
		if (root == null) return;
		System.out.println("root.data " + root.data);

		if (isLonely && root.data!=-1) {
			nodes.add(root.data);
		}

		getPersonIDs(root.left, root.right == null || root.right.data==-1, nodes);
		getPersonIDs(root.right, root.left == null || root.left.data==-1, nodes);
	}
}
public class LonelyNodes
{
	static BinaryTreeNode root;
	void insert(BinaryTreeNode temp, int key)
	{
		if (temp == null)
		{
			temp = new BinaryTreeNode(key);
			return;
		}
		Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
		q.add(temp);

		// Do level order traversal until we find an empty place.
		while (!q.isEmpty())
		{
			temp = q.peek();
			q.remove();

			if (temp.left == null)
			{
				temp.left = new BinaryTreeNode(key);
				break;
			}
			else
				q.add (temp.left);

			if (temp.right == null)
			{
```

```
                    temp.right = new BinaryTreeNode(key);
                    break;
                }
                else
                    q.add(temp.right);
            }
        }
        public static void main (String args[])
        {
            Scanner sc=new Scanner (System.in);
            LonelyNodes ln=new LonelyNodes();
            Solution sol= new Solution ();

            String str[] = sc.nextLine().split(" ");
            root=new BinaryTreeNode (Integer.parseInt(str[0]));
            for(int i=1; i<str.length; i++)
                ln.insert (root,Integer.parseInt(str[i]));

            System.out.println(sol.getPersonIds (root));
        }
    }
```

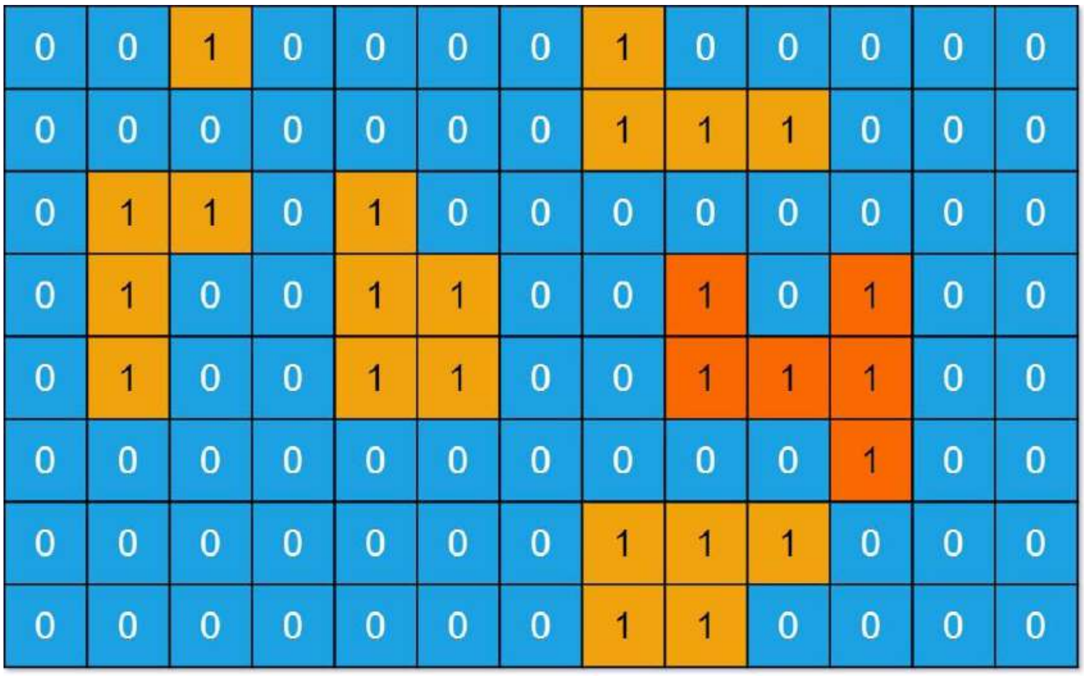
2. Max Area of Island

You are given an m x 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

0

Input: grid = [0,0,1,0,0,0,0,1,0,0,0,0,0], [0,0,0,0,0,0,0,1,1,1,0,0,0], [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]]

Output: 6

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

Example 2:

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

Output: 0

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50

grid[i][j] is either 0 or 1.

Program to Max Area of Island:

```
import java.util.*;
public class MaxArea_BFS
{
	public static int solve(int[][] grid)
	{
		if (grid == null || grid.length == 0)
		return 0;
		int res = 0;
		int curr = 0;
		for (int i = 0; i < grid.length; i++)
		{
			for (int j = 0; j <grid[0].length; j++)
			{
				if (grid[i][j] == 1)
				{
					grid[i][j] = 0;
					curr = bfs(grid, i, j);
					res = Math.max(curr, res);
				}
			}
		}
	return res;
	}
	private static int bfs(int[][] grid, int k, int l)
	{
		Queue<int[]> q = new LinkedList<>();
		q.offer(new int[]{k, l});
		int res = 0;
		while(!q.isEmpty())
		{
			int[] curr = q.poll();
			res++;
			final int[][] neigs = new int[][]{{-1, 0}, {1, 0}, {0, -1}, {0, 1},{-1, 1}, {1, 1}, {-1, -
			1}, {1, -1}};
			for (int[] neig : neigs)
			{
				int i = curr[0] + neig[0];
				int j = curr[1] + neig[1];
				if (i >= 0 && i < grid.length && j >= 0 && j < grid[0].length && grid[i][j] == 1)
				{
					grid[i][j] = 0;
					q.offer(new int[]{i, j});
				}
			}
		}
	return res;
	}

	public static void main(String[] args)
```

```
        {
                Scanner in = new Scanner(System.in);
                int m = in.nextInt();
                int n = in.nextInt();
                int[][] board = new int[m][n];
                for (int i = 0; i < m; i++)
                {
                        for (int j = 0; j < n; j++)
                        {
                                board[i][j] = in.nextInt();
                        }
                }
        System.out.println(solve(board));
        }
}
```

Sample Input-1:

```
5 4
0 0 1 1
0 0 1 0
0 1 1 0
0 1 0 0
1 1 0 0
```

Sample Output-1:

```
8
```

Sample Input-2:

```
5 5
0 1 1 1 1
0 0 0 0 1
1 1 0 0 0
1 1 0 1 1
0 0 0 1 0
```

Sample Output-2:

```
5
```

3. No. of Distinct Islands

Given a Boolean 2D matrix **grid** of size **n** * **m**. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is equal to another (not rotated or reflected).

Example 1:

Input:

```
grid[][] = {{1, 1, 0, 0, 0},
            {1, 1, 0, 0, 0},
            {0, 0, 0, 1, 1},
            {0, 0, 0, 1, 1}}
```

Output:

1

Explanation:

Island 1, 1 at the top left corner is same as island
1, 1 at the bottom right corner.

Example 2:

Input:

```
grid[][] = {{1, 1, 0, 1, 1},
            {1, 0, 0, 0, 0},
            {0, 0, 0, 0, 1},
            {1, 1, 0, 1, 1}}
```

Output:

3

Explanation:

Distinct islands are: 1, 1 at the top left corner; 1, 1 at the top right corner and 1 at the bottom right corner. We ignore the island 1, 1 at the bottom left corner since 1, 1 it is identical to the top right corner.

Your Task:

You don't need to read or print anything. Your task is to complete the function **countDistinctIslands()** which takes the **grid** as an input parameter and returns the total number of **distinct** islands.

Expected Time Complexity: O(n * m)

Expected Space Complexity: O(n * m)

Constraints:

$1 \leq n, m \leq 500$

grid[i][j] == 0 or grid[i][j] == 1

Program

```
import java.util.*;

class DistinctIslands_BFS
{
        public int numDistinctIslands(int[][] grid)
        {
                Set<String> islands = new HashSet<>();
                for(int row = 0; row < grid.length; row++)
                {
                        for(int col = 0; col < grid[0].length; col++)
                        {
                                if(grid[row][col] == 1)
                                {
                                        String island = bfs(grid, row , col);
                                        islands.add(island);
                                }
                        }
                }
        return islands.size();
        }

        private String bfs(int[][] grid, int row, int col)
        {
                Queue<int[]> queue = new LinkedList<>();
                // row, col
                int[][] dirs = {{1, 0},  {0, 1}, {-1, 0}, {0, -1}};
                char direction[] = {'d','r','u','l'};

                queue.offer(new int[]{row, col});

                int rows = grid.length;
                int cols = grid[0].length;

                StringBuilder builder = new StringBuilder();

                grid[row][col] = 0;

                while(!queue.isEmpty())
                {
                        int size = queue.size();
                        for(int i = 0; i < size; i++)
                        {
                                int[] cell = queue.poll();
                                int count=0;
                                for(int[] dir : dirs)
                                {
                                        int r = cell[0] + dir[0];
                                        int c = cell[1] + dir[1];
                        if(inBound(rows, cols, r, c) && grid[r][c] == 1)
                        {
                            grid[r][c] = 0;
                            builder.append(direction[count]);
                            System.out.println("r " + r + " c " + c);
                            queue.offer(new int[]{r, c});
                        }
                        count++;
                    }
                builder.append('b');
                }
            }
            System.out.println(builder.toString());
        return builder.toString();
        }

        private boolean inBound(int rows, int cols, int row, int col)
        {
            return row >= 0 && col >= 0 && row < rows && col < cols;
        }

        public static void main(String args[])
        {
            Scanner sc=new Scanner(System.in);
            int m=sc.nextInt();
            int n=sc.nextInt();
            int grid[][]=new int[m][n];
            for(int i=0;i<m;i++)
                for(int j=0;j<n;j++)
                    grid[i][j]=sc.nextInt();

            System.out.println(new DistinctIslands_BFS().numDistinctIslands(grid));
        }
    }
```

Sample Input-1:
```
---------------
4 5
1 1 0 0 0
1 1 0 0 0
0 0 0 1 1
0 0 0 1 1
```

Sample Output-1:
```
----------------
1
```

Sample Input-2:

```
5 5
1 1 0 1 1
1 0 0 0 1
0 0 0 0 0
1 0 0 0 1
1 1 0 1 1
```

Sample Output-2:

```
4
```

Depth First Search Algorithm:

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1.Visited

2.Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

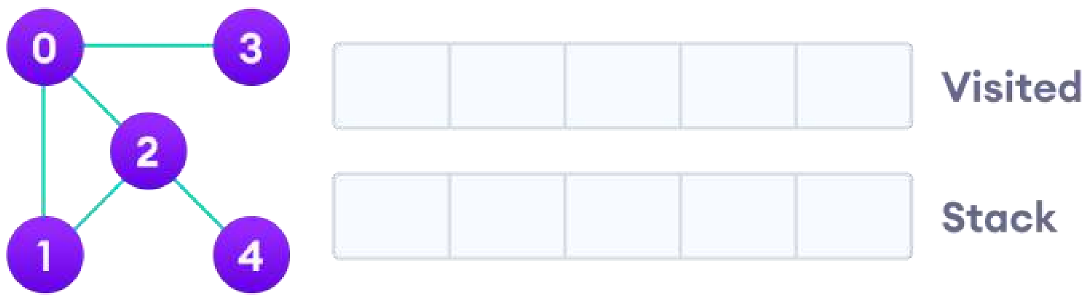
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

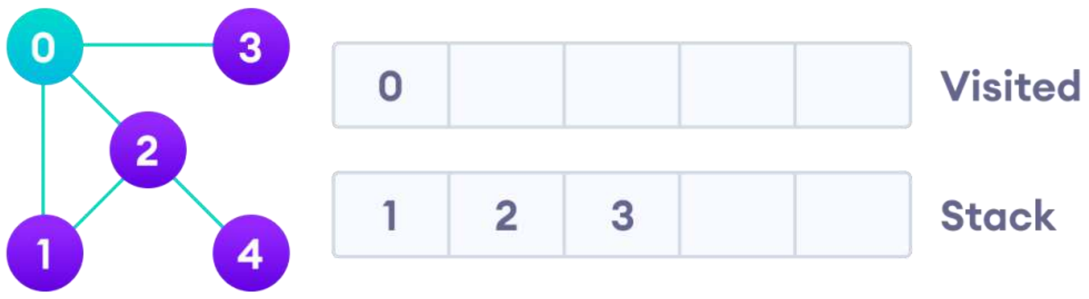
Let's see how the Depth First Search algorithm works with an example.

We use an undirected graph with 5 vertices.



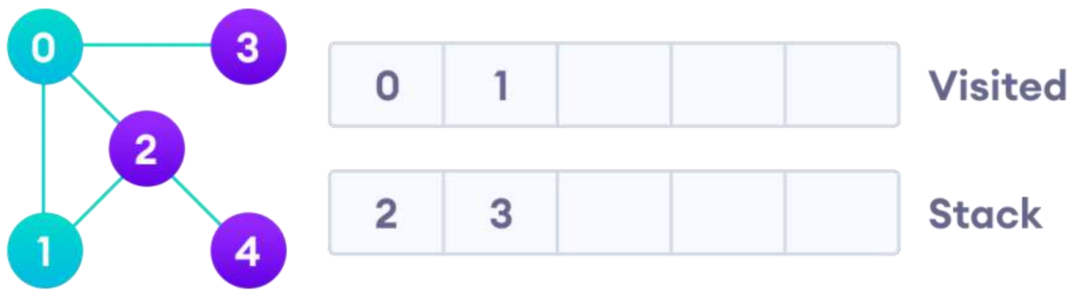
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



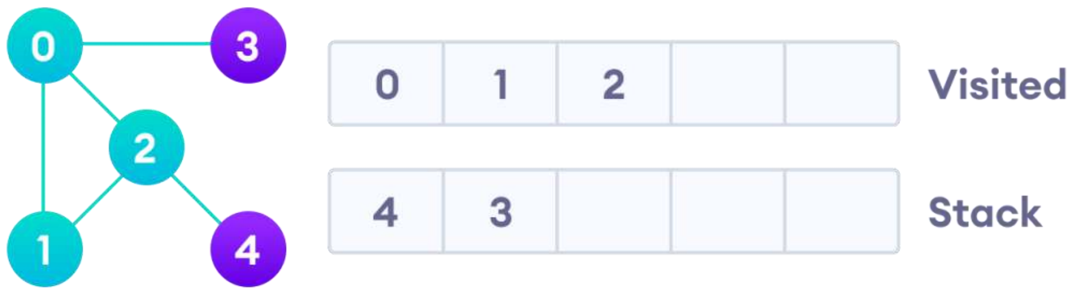
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e., 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

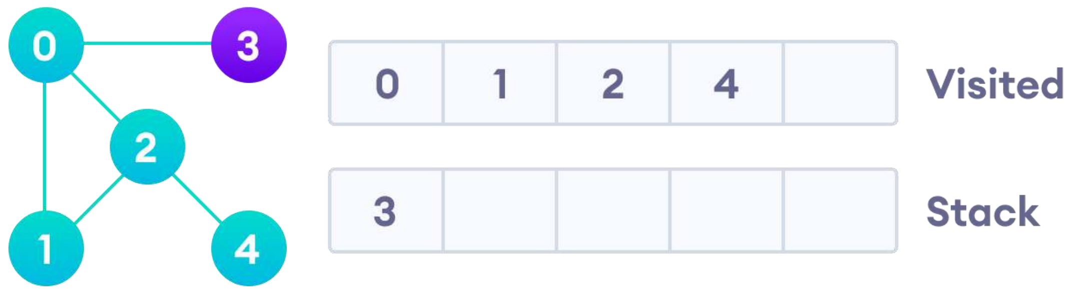


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

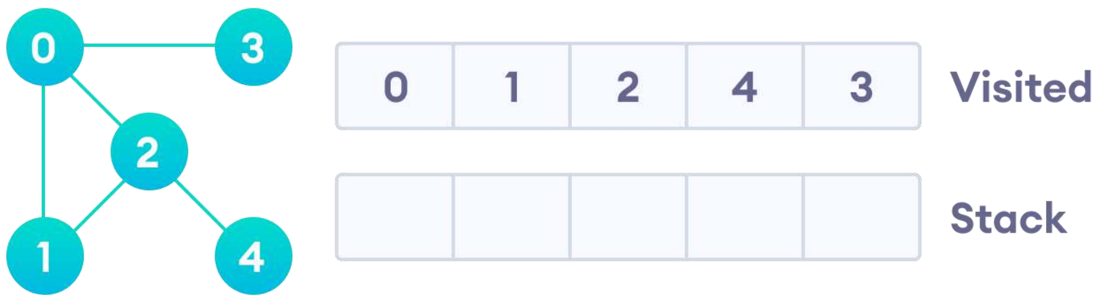


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS (G, u)
{
        u.visited = true
        for each v ∈ G.Adj[u]
                if v.visited == false
                        DFS(G,v)
}
init()
{
        For each u ∈ G
                u.visited = false
        For each u ∈ G
                DFS(G, u)
}
```

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Program:
DFS traversal from a graph

```
import java.io.*;
import java.util.*;
// This class represents a directed graph using adjacency list representation
class Graph
{
	private int V;
	// Array of lists for Adjacency List Representation
	private LinkedList<Integer> adj[];

	// Constructor
	Graph(int v)
	{
		V = v;
		adj = new LinkedList[v];
		for (int i = 0; i < v; ++i)
			adj[i] = new LinkedList();
	}
	// Function to add an edge into the graph
	void addEdge(int v, int w)
	{
		// Add w to v's list
		adj[v].add(w);
	}
	// A function used by DFS
	void DFSUtil(int v, boolean visited[])
	{
		// create a stack used to do iterative DFS
		Stack<Integer> stack = new Stack<>();

		// push the source node into the stack
		stack.push(v);

		// loop till stack is empty
		while (!stack.empty())
		{
			// Pop a vertex from the stack
			v = stack.pop();

			// if the vertex is already visited, ignore it
			if (visited[v])
			continue;

			// we will reach here if the popped vertex `v` is not visited yet;
			// print `v` and process its unvisited adjacent nodes into the stack
			visited[v] = true;
			System.out.print(v + " ");
			// do for every edge (v, u)
```

```
                    List<Integer> adjList = adj[v];
                    for (int i = adjList.size() - 1; i >= 0; i--)
                    {
                            int u = adjList.get(i);
                            if (!visited[u])
                                    stack.push(u);
                    }
            }
    }

    // The function to do DFS traversal. It uses interactive DFSUtil()
    void DFS(int v)
    {
            // Mark all the vertices as not visited(set as false by default in java)
            boolean visited[] = new boolean[V];

            // Call the interactive helper function to print DFS traversal
            DFSUtil(v, visited);
    }

    public static void main(String args[])
    {
            Graph graph = new Graph(5);
            graph.addEdge(0, 1);
            graph.addEdge(0, 4);
            graph.addEdge(1, 0);
            graph.addEdge(1, 2);
            graph.addEdge(1, 3);
            graph.addEdge(1, 4);
            graph.addEdge(2, 3);
            graph.addEdge(2, 1);
            graph.addEdge(3, 1);
            graph.addEdge(3, 2);
            graph.addEdge(3, 4);
            graph.addEdge(4, 1);
            graph.addEdge(4, 0);
            graph.addEdge(4, 3);

            System.out.println("Depth First Traversal (starting from vertex 0)");
            graph.DFS(0);
    }
}
```

Output:

```
Depth First Traversal (starting from vertex 0)
0 1 2 3 4
```

The Maze Problem

You may remember the maze game from childhood where a player starts from one place and ends up at another destination via a series of steps. This game is also known as the rat maze problem.

What is the rat maze problem?

A rat starts at a position (source) and can only move in two directions:

1.Forward

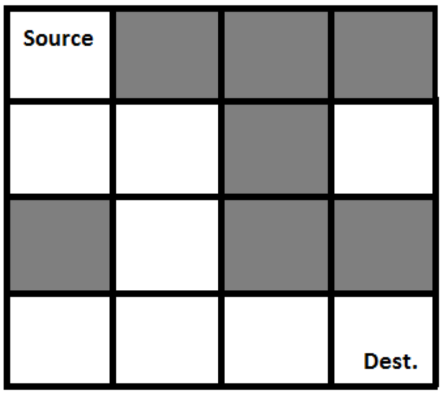
2.Down

The goal is to reach the destination.

In computer science, we can solve the maze game using array indexes or the maze matrix.

First, we generate the corresponding matrix of the maze. White represents the area where the rat can travel and is represented by 1. Grey represents the area where the rat cannot go and is represented by 0.

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



For the given maze the path above should ideally be good to go from beginning to end.

To solve the maze problem, the program is required to generate the following matrix, where 1 represents the path taken by the rat to travel from source to destination.

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

Solution

1.Firstly, create a maze matrix.

2.Make a recursive call with the position of the rat in the matrix

(initial position should be (0,0)).

3.If the position provided is invalid or value at position is 0, return "false".

4.Otherwise, if value at given position is 1 recursively call for position (i+1, j), (i, j+1) till the last cell (size-1, size-1) is reached. If last cell is 1 return "true".

(Note: "i" is row index and "j" is column index)

Program

```
import java.util.Scanner;
public class Maze
{
	private int size;
	Maze(int N)
	{
		this.size = N;
	}

	boolean isSafe( int maze[][], int x, int y)
	{
		return (x >= 0 && x < size && y >= 0 && y < size && maze[x][y] == 1);
	}

	boolean solveMaze(int maze[][])
	{
		if (dfs(maze, 0, 0) == false) {
			return false;
		}
		return true;
	}

	boolean dfs(int maze[][], int x, int y)
	{
		if (x == size - 1 && y == size - 1 && maze[x][y] == 1) {
			return true;
		}

		if (isSafe(maze, x, y) == true)
		{
			if (dfs(maze, x + 1, y))
				return true;

			if (dfs(maze, x, y + 1))
				return true;
			return false;
		}
		return false;
	}

	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		int n=sc.nextInt();
		Maze m = new Maze(n);
		int maze[][] = new int[n][n];
		for(int i=0;i<n;i++)
```

```
                        for(int j=0;j<n;j++)
                                maze[i][j]=sc.nextInt();
                System.out.println(m.solveMaze(maze));
        }
}
```

Input Format:

Line-1 -> An integer N, size of square board.
Next N lines -> N space separated integers

Output Format:

Print a boolean value.

Sample Input-1:

```
4
1 0 0 0
1 1 0 1
0 1 0 0
1 1 1 1
```

Sample Output-1:

```
true
```

Sample Input-2:

```
4
1 1 0 0
1 0 0 1
0 1 1 0
1 0 0 1
```

Sample Output-2:

```
false
```

Boundary Traversal of Binary Tree

Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.

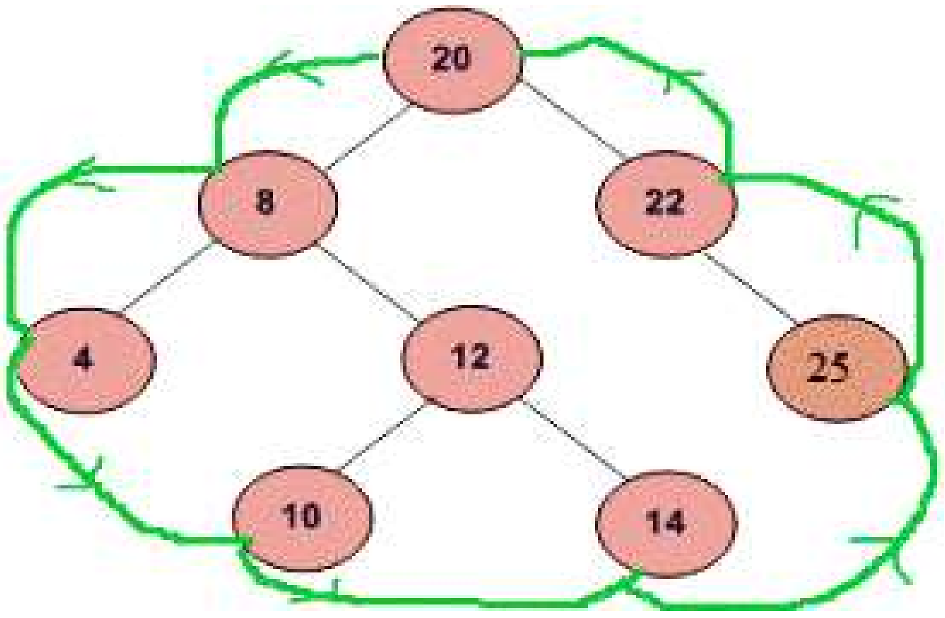
Left boundary is defined as the path from root to the left-most node.

Right boundary is defined as the path from root to the right-most node.

If the root doesn’t have left subtree or right subtree, then the root itself is left boundary or right boundary.

For example, boundary traversal of the following tree is

“20 8 4 10 14 25 22”



Boundary Traversal of binary tree

We break the problem in 3 parts:

1. Print the left boundary in top-down manner.

2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:

    2.1 Print all leaf nodes of left sub-tree from left to right.

    2.2 Print all leaf nodes of right subtree from left to right.

3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g., The left most node is also the leaf node of the tree.

Program

```
import java.util.*;

class BinaryTreeNode
{
	public int data;
	public BinaryTreeNode left, right;
	public BinaryTreeNode(int data)
	{
		this.data = data;
		left = null;
		right = null;
	}
}

class Solution
{
	List<Integer> nodes = new ArrayList<>();
	public List<Integer> boundaryOfBinaryTree(BinaryTreeNode root)
	{
		if(root == null || root.data==-1) return nodes;

		nodes.add(root.data);
		leftBoundary(root.left);
		leaves(root.left);
		leaves(root.right);
		rightBoundary(root.right);

	return nodes;
	}

	public void leftBoundary(BinaryTreeNode root)
	{
		if( (root == null || root.data==-1) || ( (root.left == null || root.left.data==-1) && (root.right
		== null || root.right.data==-1 ) ) )
			return;

		nodes.add(root.data);
		if(root.left == null||root.left.data==-1)
			leftBoundary(root.right);
		else
			leftBoundary(root.left);
	}

	public void rightBoundary(BinaryTreeNode root)
	{
		if((root == null ||root.data==-1)||
		((root.left == null ||root.left.data==-1) && (root.right == null||root.right.data==-1)))
			return;
		if(root.right == null || root.right.data==-1)
			rightBoundary(root.left);
		else
			rightBoundary(root.right);

		nodes.add(root.data); // add after child visit (reverse)
	}

	public void leaves(BinaryTreeNode root)
	{
		if(root == null || root.data==-1) return;
		if((root.left == null ||root.left.data==-1) && (root.right == null||root.right.data==-1))
		{
			nodes.add(root.data);
			return;
		}
		leaves(root.left);
		leaves(root.right);
	}
}

public class BoundaryOfBinaryTree
{
	static BinaryTreeNode root;
	void insert(BinaryTreeNode temp, int key)
	{
		if (temp == null)
		{
			temp = new BinaryTreeNode(key);
			return;
		}
		Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
		q.add(temp);

		// Do level order traversal until we find an empty place.
		while (!q.isEmpty())
		{
			temp = q.remove();

			if (temp.left == null)
			{
				temp.left = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.left);

			if (temp.right == null)
			{
```

```
                    temp.right = new BinaryTreeNode(key);
                    break;
                }
                else
                    q.add(temp.right);
            }
        }

        public static void main(String args[])
        {
            Scanner sc=new Scanner(System.in);
            BoundaryOfBinaryTree bbt=new BoundaryOfBinaryTree();
            Solution sol= new Solution();

            String str[]=sc.nextLine().split(" ");

            root=new BinaryTreeNode(Integer.parseInt(str[0]));

            for(int i=1; i<str.length; i++)
                bbt.insert(root,Integer.parseInt(str[i]));

            System.out.println(sol.boundaryOfBinaryTree(root));
        }
}
```

Sample Input-1:

```
5 2 4 7 9 8 1
```

Sample Output-1:

```
[5, 2, 7, 9, 8, 1, 4]
```

Sample Input-2:

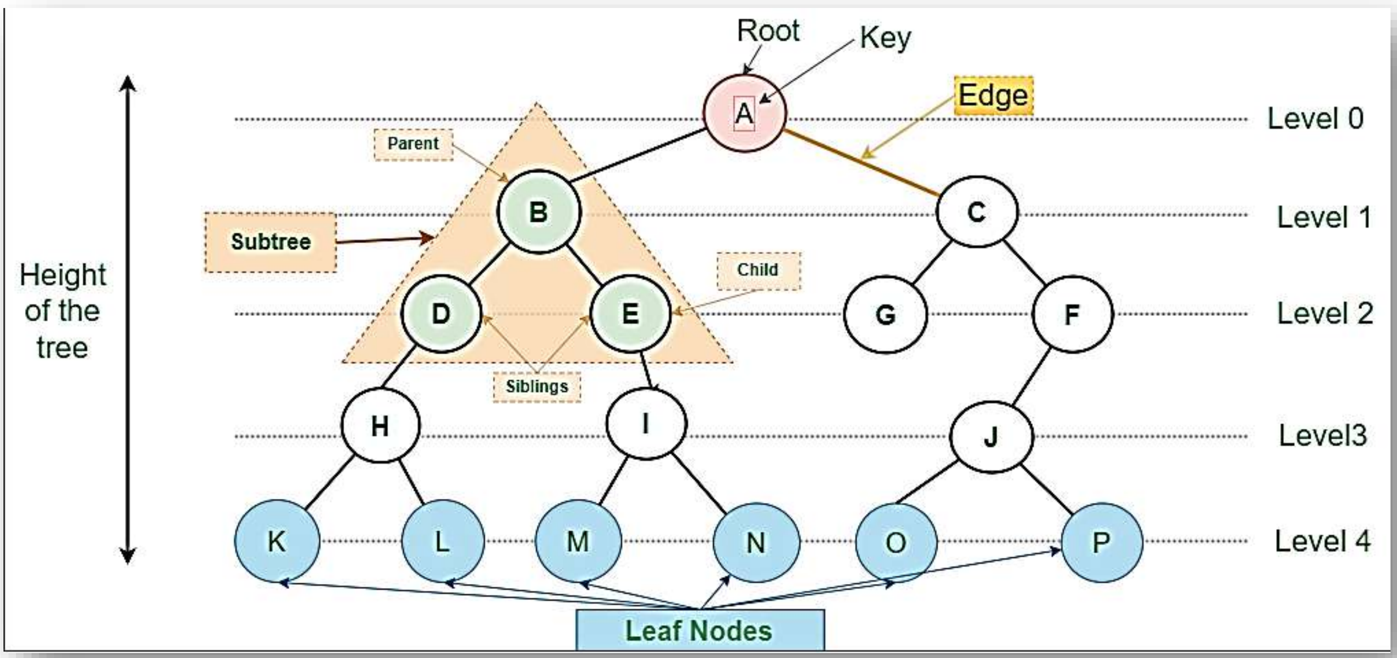
```
11 2 13 4 25 6 -1 -1 -1 7 18 9 10
```

Sample Output-2:

```
[11, 2, 4, 7, 18, 9, 10, 6, 13]
```

Trees

A **tree** is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Why Tree Data Structure?

Tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

<u>Basic Terminology:</u>

1. **Node**: Every element of tree is called as a node. It stores the actual data and links to other nodes.

2. Link / Edge / Branch: It is a connection between 2 nodes.

3. Parent Node: The Immediate Predecessor of a Node is called as Parent Node.

4.Child Node: The Immediate Successor of a Node is called as Child Node.

5. Root Node: Which is a specially designated node, and does not have any parent node.

6. Leaf node or terminal node: The node which does not have any child nodes is called leaf node.

7. Level: It is the rank of the hierarchy and the Root node is present at level 0.

If a node is present at level **L,** then its parent node will be at the level **L-1** and child nodes will present at level **L+1**.

8. Siblings: The nodes which have same parent node are called as siblings.

9. Degree of a node: The number of nodes attached to a particular node.

10. Degree of a tree: The maximum degree of a node is called as degree of tree.

11. Height of a node: It is the length of longest path from the node to leaf. Height of leaf node is zero.

12. Height of a tree: It is the length of longest path from the root node to leaf.

Advantages of Trees:

Trees are so useful and frequently used, because they have some very serious advantages:

1. Trees reflect structural relationships in the data.
2. Trees are used to represent hierarchies.
3. Trees provide an efficient insertion and searching.
4. Trees are very flexible data, allowing to move subtrees around with minimum effort.

Binary Tree

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent.

Every node in a binary tree has a left and right reference along with the data element.

A parent node has two child nodes: the left child and right child.

At every level of **L**, the maximum number allowed for nodes stands at **2*L**.

The height of a binary tree stands defined as the longest path emanating from a root node to the tree's leaf node.

What is Binary Tree, more than the Binary Tree definition?

- The **Maximum** number of nodes possible for **Height H** is **2^{H+1} -1**.

  For Example, if the height of Binary Tree is 3, the highest number of nodes for this height stands equal to 15, that is, 2^{3+1}-1.
- The **Minimum** number of nodes possible at the **Height H** is **H+1**.
- The **Minimum Height** with N nodes will be **$LOG_2(N+1)-1$**
- The **Maximum Height** with N nodes will be **N-1**

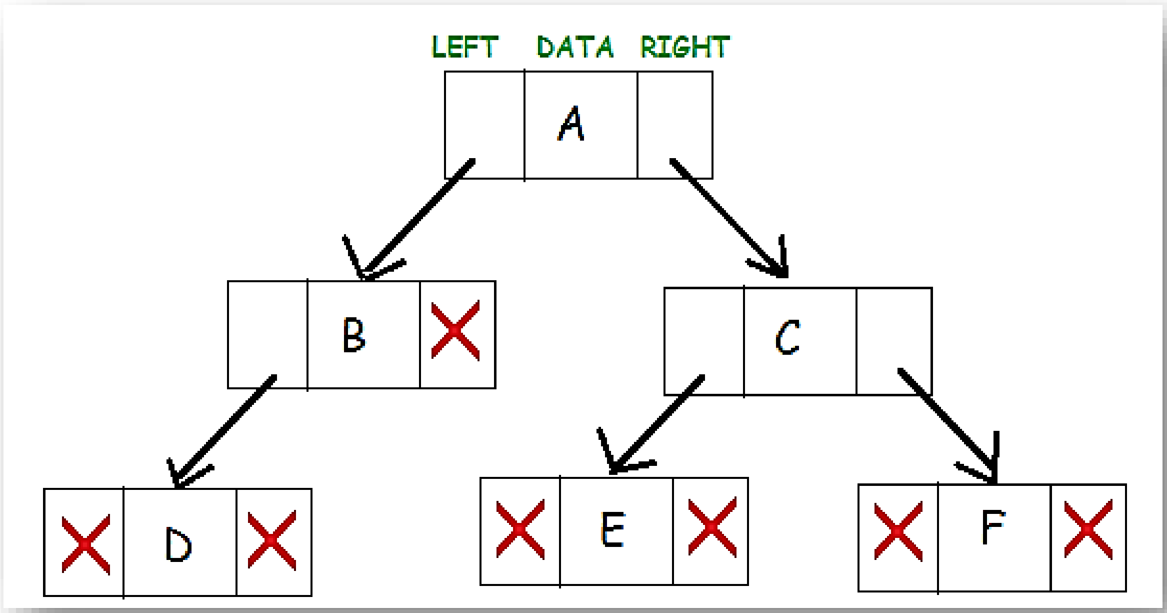
Binary Tree Components:

There are three binary tree components. Every binary tree node has these three components associated with it. It becomes an essential concept for programmers to understand these three binary tree components:

1. Data element

2. Pointer to left subtree

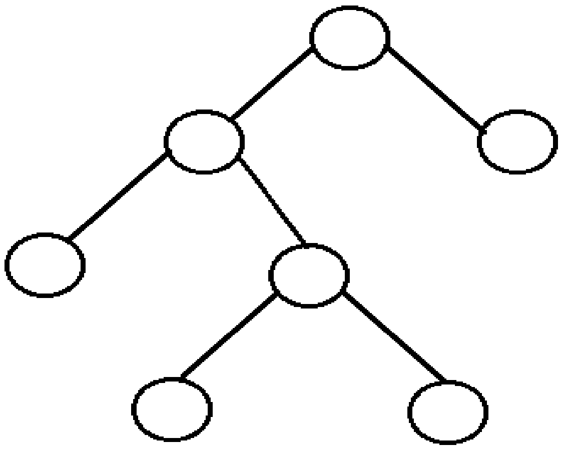
3. Pointer to right subtree



Types of Binary Trees (Based on Structure)

1. Rooted binary tree: It has a root node and every node has at-most two children.

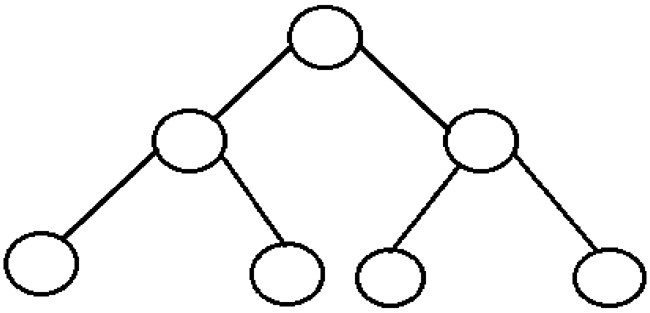
2. Full binary tree: It is a tree in which every node in the tree has either 0 or 2 children.



The number of nodes, n, in a full binary tree is at least $n = 2^h - 1$, and at most $n = 2^{h+1} - 1$, where h is the height of the tree.

The number of leaf nodes l, in a full binary tree is number, No. of internal nodes(L) + 1, i.e., $l = L+1$.

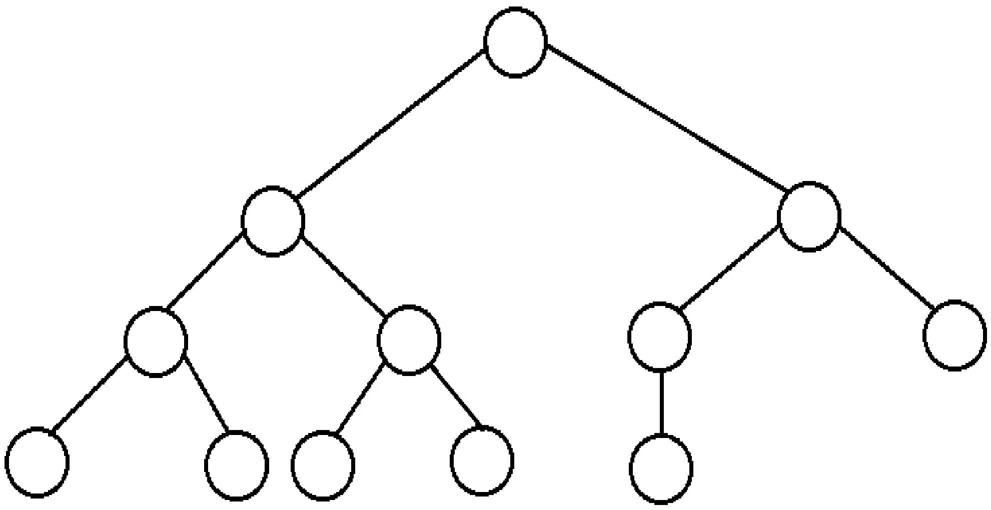
3. Perfect binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



A perfect binary tree with l leaves has $n = 2l-1$ nodes.

In perfect full binary tree, $l = 2h$ and $n = 2^{h+1} - 1$ where, n is number of nodes, h is height of tree and l is number of leaf nodes

4. Complete binary tree: It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



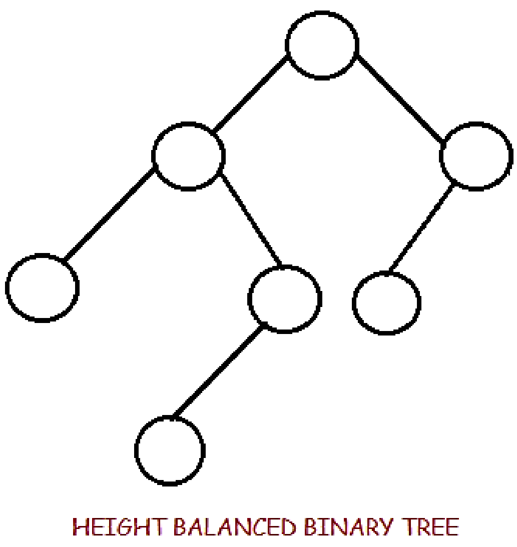
The number of internal nodes in a complete binary tree of ***n*** nodes is ***floor(n/2)*.**

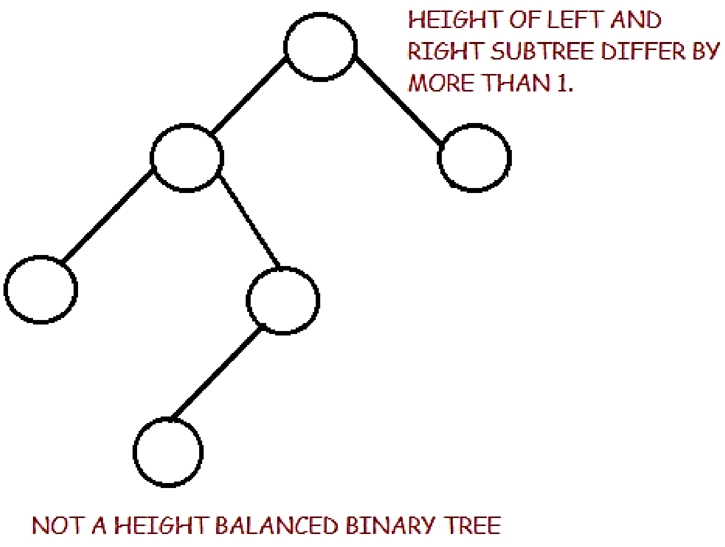
5. Balanced binary tree: A binary tree is height balanced if it satisfies the following constraints:

The left and right subtrees' heights differ by at most one, AND

The left subtree is balanced, AND

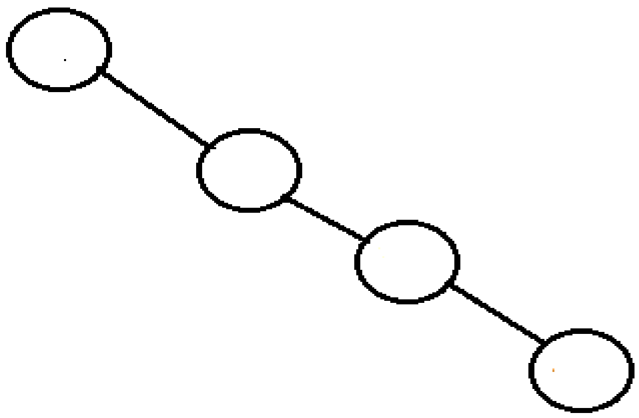
The right subtree is balanced




HEIGHT OF LEFT AND RIGHT SUBTREE DIFFER BY MORE THAN 1.
NOT A HEIGHT BALANCED BINARY TREE

The height of a balanced binary tree is *Log n* where *n* is number of nodes.

6. Degenerate tree: It is a tree is where each parent node has only one child node.



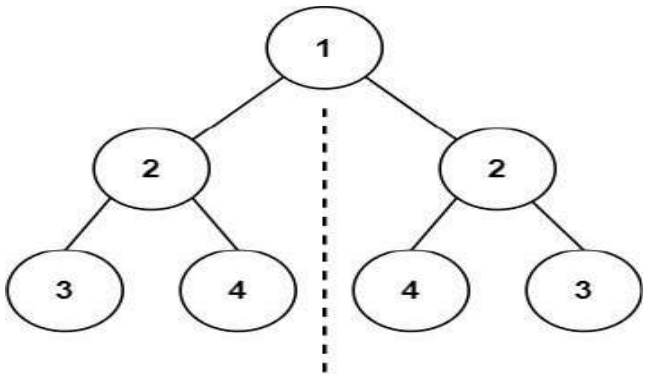
1. Symmetric Tree

A binary tree is considered symmetric if it is a mirror image of itself, i.e., it is symmetric around its root node. Given the root node of a binary tree, determine whether it's symmetric.

Two trees mirror each other if all the following conditions are satisfied:

- Both trees are empty, or both are non-empty.
- The left subtree is the mirror of the right subtree.
- The right subtree is the mirror of the left subtree.

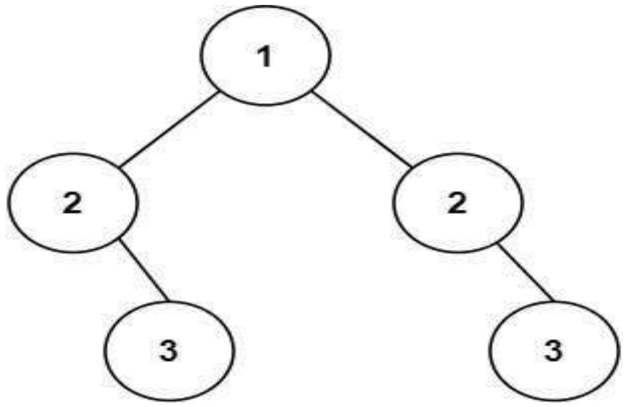
Example:1



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

Output: true

Example:2



Input: root = [1, 2, 2, null, 3, null, 3]

Output: false

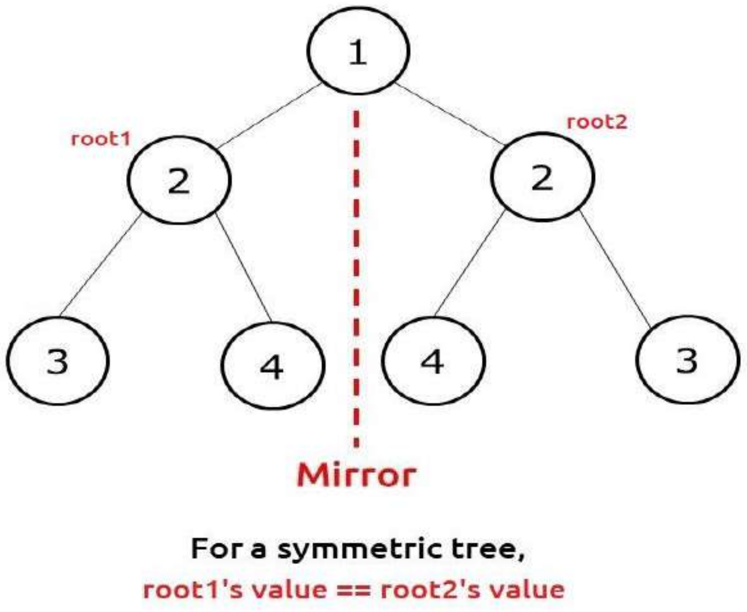
Constraints: The number of nodes in the tree is in the range [1, 1000].
-100 <= Node.val <= 100

Time Complexity: O(N):
Reason: We are doing simple tree traversal and changing both root1 and root2 simultaneously.

Space Complexity: O(N)
Reason: In the worst case (skewed tree), space complexity can be O(N).

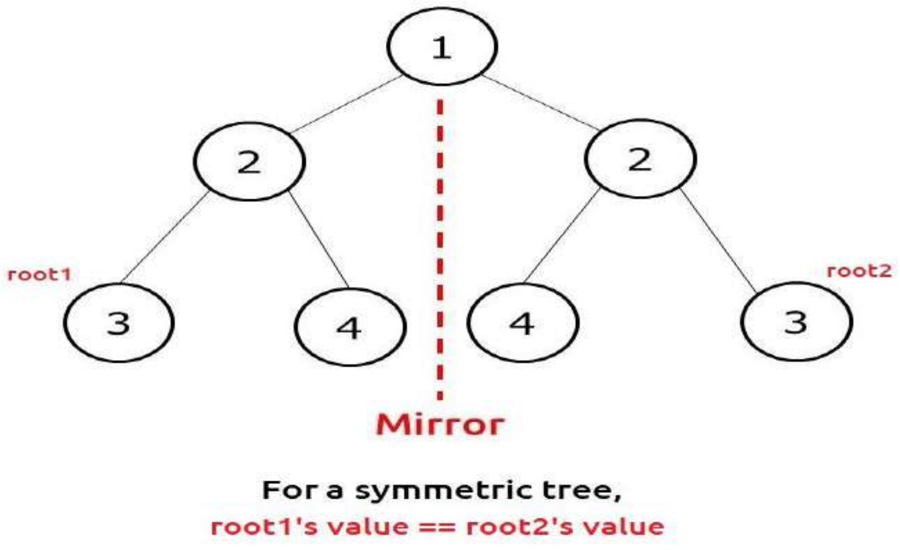
Solution:

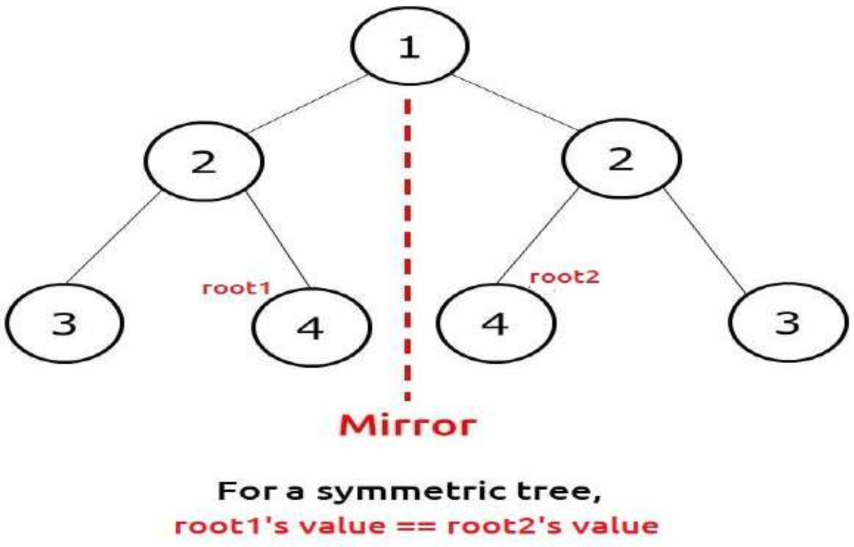
We need to understand the property of the mirror. We can ignore the root node as it is lying on the mirror line. In the next level, for a symmetric tree, the node at the root's left should be equal to the node at the root's right.
If we take two variables root1 and root2 to represent the left child of root and right child of the root, then



Further, we need to understand that when root1's value is equal to root2's value, we need to further check for its children.

As we are concerned about node positions through a mirror, root1's left child should be checked with root2's right child and root1's right child should be checked with root2's left child.





Approach:

The algorithm steps can be summarized as follows:

- We take two variables **root1** and **root2** initially both pointing to the root.
- Then we use any tree traversal to traverse the nodes. We will simultaneously change root1 and root2 in this traversal function.
- For the **base case**, if both are pointing to NULL, we return true, whereas if only one points to NULL and other to a node, we return false.
- If both points to a node, we first compare their value. If it is same, we will check for the lower levels of the tree.
- We will recursively call the function to check the root1's left child with root2's right child;

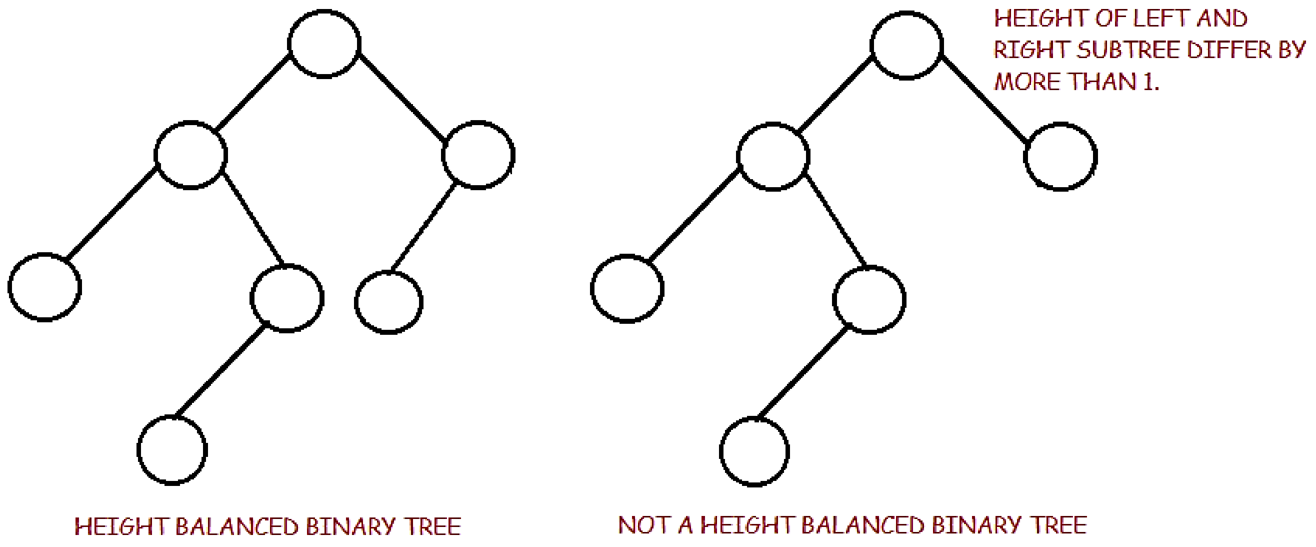
  then we again recursively check the root1's right child with root2's left child.
- When all three conditions (node values of left and right and two recursive calls) return true, we return true from our function else we return false.

Program

```
import java.util.*;
class BinaryTreeNode
{
	public int data;
	public BinaryTreeNode left, right;
	public BinaryTreeNode(int data)
	{
		this.data = data;
		left = null;
		right = null;
	}
}

class Solution
{
	public boolean isSymmetric(BinaryTreeNode root)
	{
		return root==null || isSymmetricHelp(root.left, root.right);
	}

	private boolean isSymmetricHelp(BinaryTreeNode left, BinaryTreeNode right)
	{
		if(left==null || right==null)
			return left==right;
		if(left.data!=right.data)
			return false;

		return isSymmetricHelp(left.left, right.right) && isSymmetricHelp(left.right, right.left);
	}
}

public class SymmetricTree
{
	static BinaryTreeNode root;
	static BinaryTreeNode temp = root;
	void insert (BinaryTreeNode temp, int key)
	{
		if (temp == null)
		{
			root = new BinaryTreeNode(key);
			return;
		}

		Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
		q.add(temp);
		// Do level order traversal until we find an empty place.
		while (!q.isEmpty())
		{
			temp = q.remove();
			if (temp.left == null)
			{
				temp.left = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.left);

			if (temp.right == null)
			{
				temp.right = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.right);
		}
	}
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String str[]=sc.nextLine().split(" ");
		SymmetricTree st=new SymmetricTree();
		root=new BinaryTreeNode(Integer.parseInt(str[0]));

		for(int i=1; i<str.length; i++)
			st.insert(root,Integer.parseInt(str[i]));
		Solution sol= new Solution();
		System.out.println(sol. isSymmetric (root));
	}
}
```

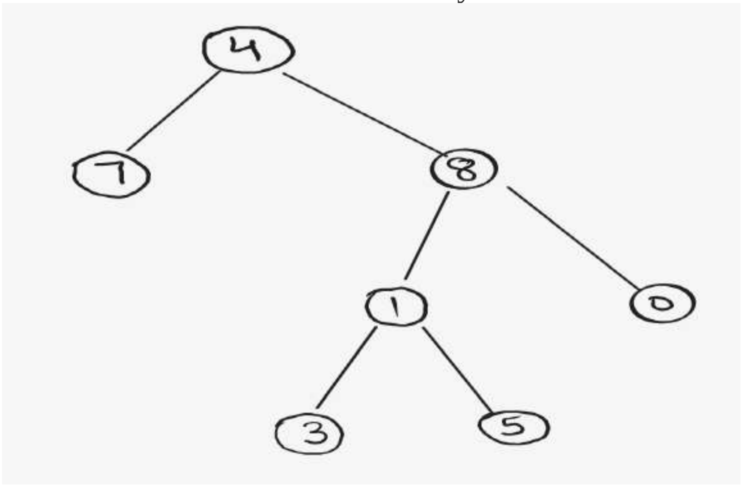
2. Balanced Binary Tree

A binary tree is height balanced if it satisfies the following constraints:
The left and right subtrees' heights differ by at most one, AND the left subtree is balanced, AND the right subtree is balanced.


HEIGHT OF LEFT AND RIGHT SUBTREE DIFFER BY MORE THAN 1.
HEIGHT BALANCED BINARY TREE
NOT A HEIGHT BALANCED BINARY TREE

Example-1:

Input Format: Given the root of Binary Tree

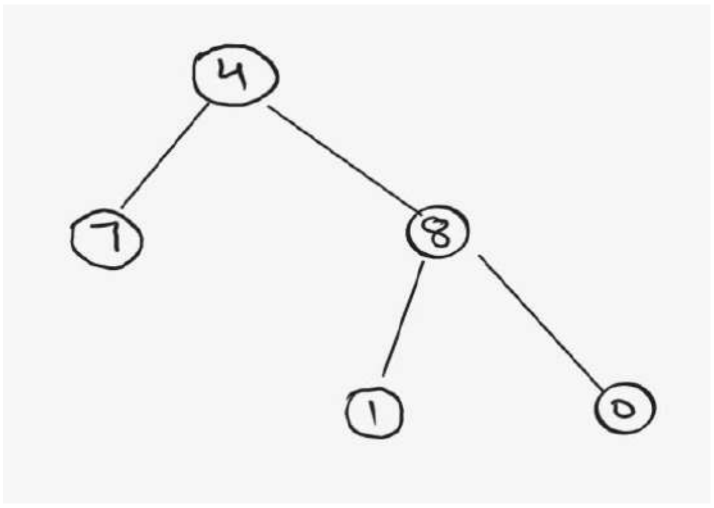


Result: False

Explanation: At Node 4, Left Height = 1 & Right Height = 3, Absolute Difference is 2 which is greater than 1, Hence, not a balanced tree.

Example-2:

Input Format: Given the root of Binary Tree



Result: True

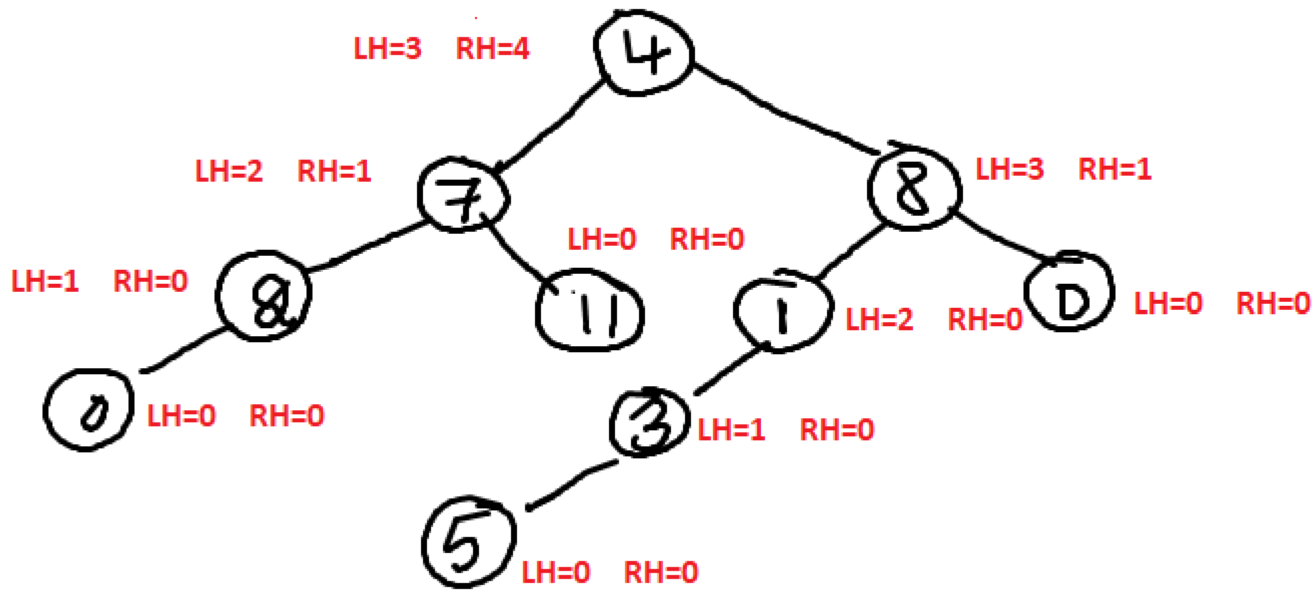
Explanation: All Nodes in the tree have an Absolute Difference of Left Height & Right Height not more than 1.

Solution-1:

For a Balanced Binary Tree, Check left subtree height and right subtree height for every node present in the tree. Hence, traverse the tree recursively and calculate the height of left and right subtree from every node, and whenever the condition of Balanced tree violates, simply return false.

Condition for Balanced Binary Tree

For all Nodes , Absolute(Left Subtree Height – Right Subtree Height) <= 1



Start traversing the tree, the example given in above diagram:

- Reach on **Node 4**, call Height Function, Left height = 3, Right height = 4 so Absolute Difference between two is Abs (3 – 4) = 1.
- Reach on **Node 7**, call Height Function, Left height = 2, Right height = 1 so Absolute Difference between two is Abs(2 – 1) = 1.
- Reach on **Node 2**, call Height Function, Left height = 1, Right height = 0 so Absolute Difference between two is Abs(1 – 0) = 1.
- Reach on **Node 0**, call Height Function, Left height = 0, Right height = 0 so Absolute Difference between two is Abs(0 – 0) = 0.
- Now, on **PostOrder of Node 0,** the left subtree (null) gives true & right subtree (null) gives true, as both are true, return true.
- Now, on **PostOrder of Node 2,** the left subtree (0) gives true & right subtree (null) gives true, as both are true, return true.
- Reach on **Node 11**, call Height Function, Left height = 0, Right height = 0 so Absolute Difference between two is Abs(0 – 0) = 0.
- Now, on **PostOrder of Node 11,** the left subtree (null) gives true & right subtree (null) gives true, as both are true, return true.
- Now, on **PostOrder of Node 7,** the left subtree (2) gives true & right subtree (11) gives true, as both are true, return true.

- Reach on **Node 8**, call Height Function, Left height = 3, Right height = 1 so Absolute Difference between two is Abs(3 – 1) = 2. Here Condition violates, simply return false, no need to call further
- Now, on **PostOrder of Node 4,** the left subtree (7) gives true & right subtree (8) gives false, so any one of subtree gives false, return false.

Time Complexity: O(N*N)
For every node, Height Function is called which takes O(N) Time. Hence for every node it becomes N*N

Space Complexity: O(1) (Extra Space) **+ O(H)** (Recursive Stack Space where **"H"** is height of tree)

<u>Solution-2</u>: Using Post Order Traversal

Intuition: Can we optimize the above brute force solution? Which operation do you think can be skipped to optimize the time complexity?
Aren't we traversing the subtrees again and again in the above example?
Yes, so can we skip the repeated traversals?
What if we can make use of post-order traversal?

So, the idea is to use post-order traversal. Since, in post order traversal, we first traverse the left and right sub trees and then visit the parent node, similarly instead of calculating the height of the left subtree and right subtree every time at the root node, use post-order traversal, and keep calculating the heights of the left and right subtrees and perform the validation.

<u>Approach:</u>

- Start traversing the tree recursively and do work in Post Order.
- For each call, calculate the height of the root node, and return it to previous calls.
- Simultaneously, in the Post Order of every node, Check for condition of balance as information of left and right subtree height is available.
- If it is balanced, simply return height of current node and if not then return -1.
- Whenever the subtree result is -1, simply keep on returning -1.

Time Complexity: O(N)
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space (Where "H" is the height of binary tree)

In Post Order, Start traversing the tree on the example given in below diagram

- Reach on **Node 0**, Left child = null so 0 Height, Right child = null so 0 Height, Difference is 0-0 = 0 , (0 <= 1) so return height , i.e., Max(0,0) + 1 = 1.
- Reach on **Node 2,** Left subtree height = 1 , Right subtree height = 0, Difference is 1-0 = 1 , (1 <= 1) so return height , i.e., Max(1,0) + 1 = 2.
- Reach on **Node 11,** Left child = null so 0 Height , Right child = null so 0 Height , Difference is 0-0 = 0 , (0 <= 1) so return height , i.e., Max(0,0) + 1 = 1.
- Reach on **Node 7,** Left subtree height = 2 , Right subtree height = 1, Difference is 2-1 = 1 , (1 <= 1) so return height , i.e., Max(2,1) + 1 = 3.
- Reach on **Node 5,** Left child = null so 0 Height , Right child = null so 0 Height , Difference is 0-0 = 0 , (0 <= 1) so return height , i.e., Max(0,0) + 1 = 1.
- Reach on **Node 3,** Left subtree height = 1 , Right subtree height = 0, Difference is 1-0 = 1 , (1 <= 1) so return height , i.e., Max(1,0) + 1 = 2.
- Reach on **Node 1,** Left subtree height = 2 , Right subtree height = 0, Difference is 2-0 = 2 , (2 > 1) i.e., Tree is **not Balanced** , so return -1.
- Reach on **Node 8,** Left subtree height = **-1** , indicates that tree is not balanced, simply return -1;
- Reach on **Node 4,** Left subtree height = 3 , Right subtree height = **-1**, therefore indicates that tree is not balanced , simply return -1;

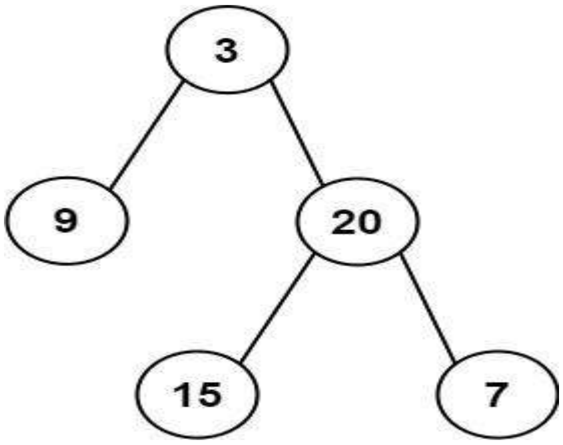
In the Main function, If the final Height of tree is -1 return false as tree is not balanced, else return true.

Program

```
import java.util.*;

class BinaryTreeNode
{
	public int data;
	public BinaryTreeNode left, right;
	public BinaryTreeNode(int data){
		this.data = data;
		left = null;
		right = null;
	}
}

class Solution
{
	public boolean isBalanced(BinaryTreeNode root)
	{
		if(root == null || root.data==-1)
			return true;

		return helper(root) != -1;
	}
	private int helper(BinaryTreeNode root)
	{
		if(root == null || root.data==-1)
			return 0;

		int left = helper(root.left);
		int right = helper(root.right);
		if(left == -1 || right == -1 || Math.abs(left - right) > 1)
			return -1;

		return Math.max(left, right) + 1;
	}
}


public class BalancedBinaryTree
{
	static BinaryTreeNode root;
	void insert(BinaryTreeNode temp, int key)
	{
		if (temp == null) {
			temp = new BinaryTreeNode(key);
			return;
		}
		Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
		q.add(temp);

		// Do level order traversal until we find an empty place.
		while (!q.isEmpty()) {
			temp = q.remove();

			if (temp.left == null) {
				temp.left = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.left);

			if (temp.right == null) {
				temp.right = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.right);
		}
	}

	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		BalancedBinaryTree ln=new BalancedBinaryTree();
		Solution sol= new Solution();

		String str[]=sc.nextLine().split(" ");
		root=new BinaryTreeNode(Integer.parseInt(str[0]));
		for(int i=1; i<str.length; i++)
			ln.insert(root,Integer.parseInt(str[i]));

		System.out.println(sol.isBalanced(root));
	}
}
```

3. Average of Levels in Binary Tree

Given the root of a binary tree, return the average value of the nodes on each level in the form of an array.

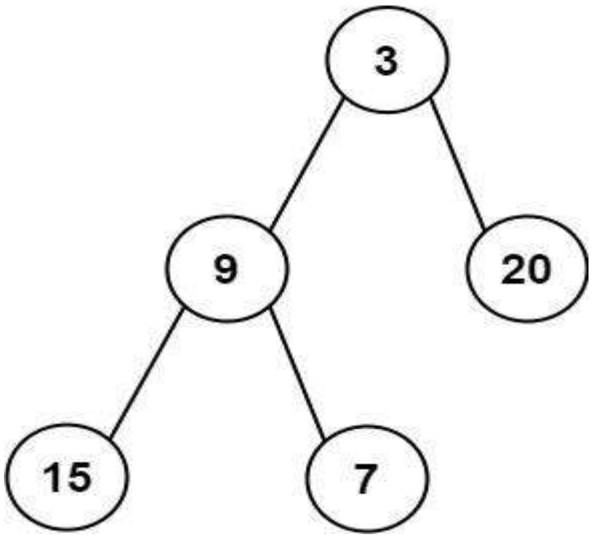


Input: 3 9 20 -1 -1 15 7

Output: [3.0, 14.5, 11.0]

Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11.

Hence return [3, 14.5, 11].



Input: 3 9 20 -1 -1 15 7

Output: [3.0, 14.5, 11.0]

Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11.

Hence return [3, 14.5, 11].

Approach

You must have read about the **Level Order Traversal** in a tree that is used to traverse in a binary tree. The level order traversal is similar to BFS traversal where the starting point is the root node.

How can we use Level Order Traversal Technique to solve the problem?

- We will use a **Queue** that will store the nodes of each level.
- We will enqueue all the nodes of the current level and store their **sum** in a variable.
- The total number of nodes in the current level will be equal to the **queue size**.
- Using them we can find the average of the current level and store it in the **result array**.
- Afterwards, we will dequeue all the nodes from Queue and enqueue the children of the dequeued nodes which will represent the next level of the tree.

Complexity Analysis:

Time Complexity: O(n)
Space Complexity: O(n)

Program:

```
import java.util.*;

class BinaryTreeNode
{
        public int data;
        public BinaryTreeNode left, right;
        public BinaryTreeNode(int data){
                this.data = data;
                left = null;
                right = null;
        }
}

class Solution
{
        public List<Double> averageOfLevels(BinaryTreeNode root)
        {
                List<Double> result = new ArrayList<>();
                Queue<BinaryTreeNode> q = new LinkedList<>();

                if((root == null)||(root.data==-1)) return result;
                q.add(root);
                while(!q.isEmpty())
                {
                        int n = q.size();
                        double sum = 0.0;
                for(int i = 0; i < n; i++)
                {
                    BinaryTreeNode node = q.poll();
                    sum += node.data;
                    if((node.left != null)&&(node.left.data!=-1))
                        q.offer(node.left);
                    if((node.right != null)&&(node.right.data!=-1))
                        q.offer(node.right);
                }
                result.add(sum / n);
            }
            return result;
        }
}

public class AverageOfLevels
{
        static BinaryTreeNode root;
        void insert(BinaryTreeNode temp, int key)
        {
            if (temp == null)
            {
                temp = new BinaryTreeNode(key);
                return;
            }
            Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
            q.add(temp);

            // Do level order traversal until we find an empty place.
            while (!q.isEmpty())
            {
                temp = q.remove();

                if (temp.left == null)
                {
                    temp.left = new BinaryTreeNode(key);
                    break;
                }
                else
                    q.add(temp.left);

                if (temp.right == null)
                {
                    temp.right = new BinaryTreeNode(key);
                    break;
                }
                else
                    q.add(temp.right);
            }
        }
```

```
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		AverageOfLevels ln=new AverageOfLevels();
		Solution sol= new Solution();

		String str[]=sc.nextLine().split(" ");
		root=new BinaryTreeNode(Integer.parseInt(str[0]));
		for(int i=1; i<str.length; i++)
			ln.insert(root,Integer.parseInt(str[i]));

		System.out.println(sol.averageOfLevels(root));
	}
}
```

Sample Input-1:

```
3 8 4 3 5 -1 7
```

Sample Output-1:

```
[3.0, 6.0, 5.0]
```

Sample Input-2:

```
1 10 4 3 -1 7 9 12 8 6 -1 -1 2
```

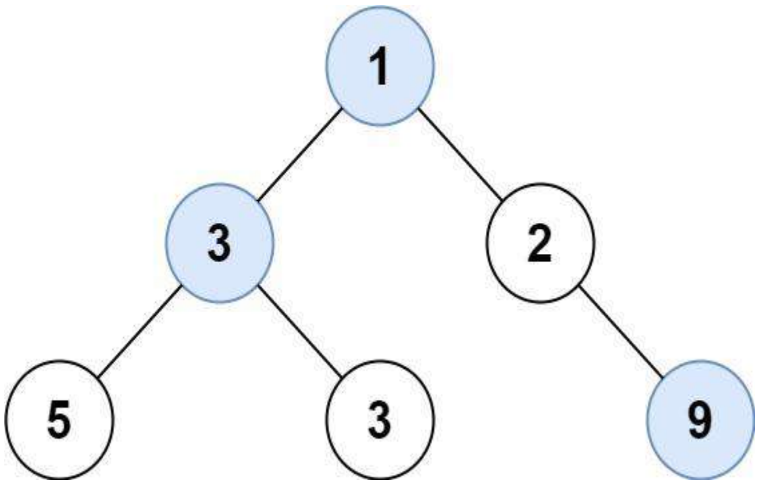
Sample Output-2:

```
[1.0, 7.0, 6.333333333333333, 7.333333333333333]
```

4. Find Largest Value in Each Tree Row

The **Root** of a Binary tree will be given. We should find the largest value in each level of the given Binary tree.

For example, let us take following binary



Example 1:

Input: 1 3 2 5 3 -1 9

Output: [1,3,9]

Example 2:

input = 1 10 4 3 -1 7 9 12 8 -1 -1 6 2 9 13

output = [1, 10, 9, 13]

Time Complexity: O(n) — We touch every node once
Space Complexity: O(n) — The last level of a full and complete binary tree can have N/2 nodes. The queue stores every level.

Approach:

We should input a Tree and pass Root of the Tree.

The idea is to recursively traverse tree in a pre-order fashion. Root is considered to be at 0th level.

While traversing, keep track of the level of the element and if its current level is not equal to the number of elements present in the list, update the maximum element at that level in the list.

```
import java.util.*;

class BinaryTreeNode
{
	public int data;
	public BinaryTreeNode left, right;
	public BinaryTreeNode(int data)
	{
		this.data = data;
		left = null;
		right = null;
	}
}

class LargestValue
{
	public List<Integer> largestValues(BinaryTreeNode root)
	{
		List<Integer> res = new ArrayList<Integer>();
		helper(root, res, 0);
		return res;
	}

	private void helper(BinaryTreeNode root, List<Integer> res, int level)
	{
		if(root == null)		return;
		//expand list size
		if(level == res.size())
			res.add(root.data);
		else
			//or set value
			res.set(level, Math.max(res.get(level), root.data));

		helper(root.left, res, level+1);
		helper(root.right, res, level+1);
	}
}

public class LargestValue
{
	static BinaryTreeNode root;
	void insert(BinaryTreeNode temp, int key)
	{
		if (temp == null) {
			temp = new BinaryTreeNode(key);
			return;
		}
		Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
		q.add(temp);
		// Do level order traversal until we find an empty place.
		while (!q.isEmpty())
		{
			temp = q.remove();

			if (temp.left == null) {
				temp.left = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.left);

			if (temp.right == null)
			{
				temp.right = new BinaryTreeNode(key);
				break;
			}
			else
				q.add(temp.right);
		}
	}

	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		LargestValue ln=new LargestValue();
		Solution sol= new Solution();

		String str[]=sc.nextLine().split(" ");
		root=new BinaryTreeNode(Integer.parseInt(str[0]));
		for(int i=1; i<str.length; i++)
			ln.insert(root,Integer.parseInt(str[i]));

		System.out.println(sol.largestValues(root));
	}
}
```

Example:

Sample Input-1:

```
2 4 3 6 4 -1 9
```

Sample Output-1:

```
[2, 4, 9]
```

Sample Input-2:

```
3 4 7 7 3 8 4
```

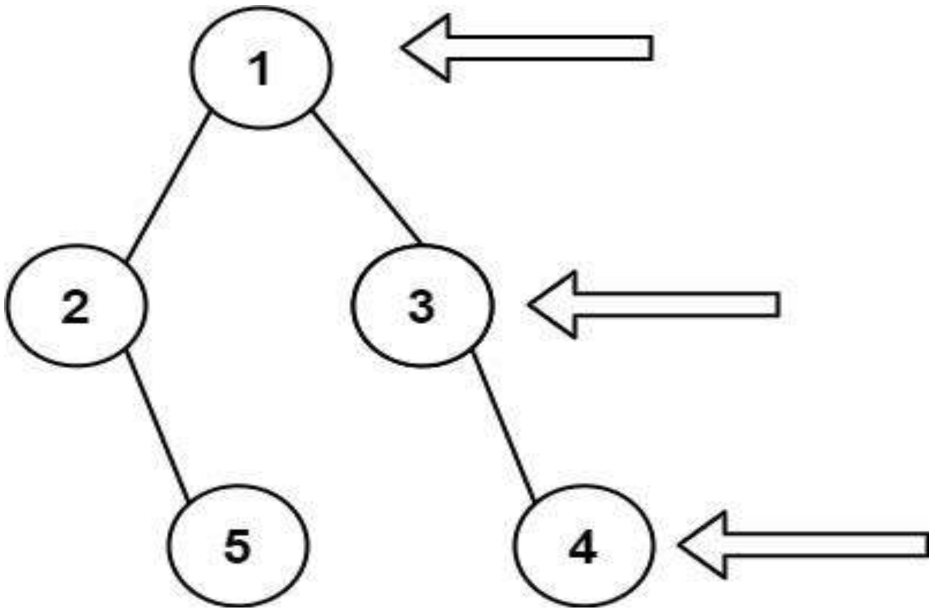
Sample Output-2:

```
[3, 4, 8]
```

5. Binary Tree Right Side View

The **Root** of a binary tree is Given. We should find the values of the nodes in the right-side view ordered from top to bottom.

For example, let us take following binary



Example 1:

Input: 1 2 3 -1 5 -1 4

Output: [1,3,4]

Example 2:

Input: 1 -1 3

Output: [1,3]

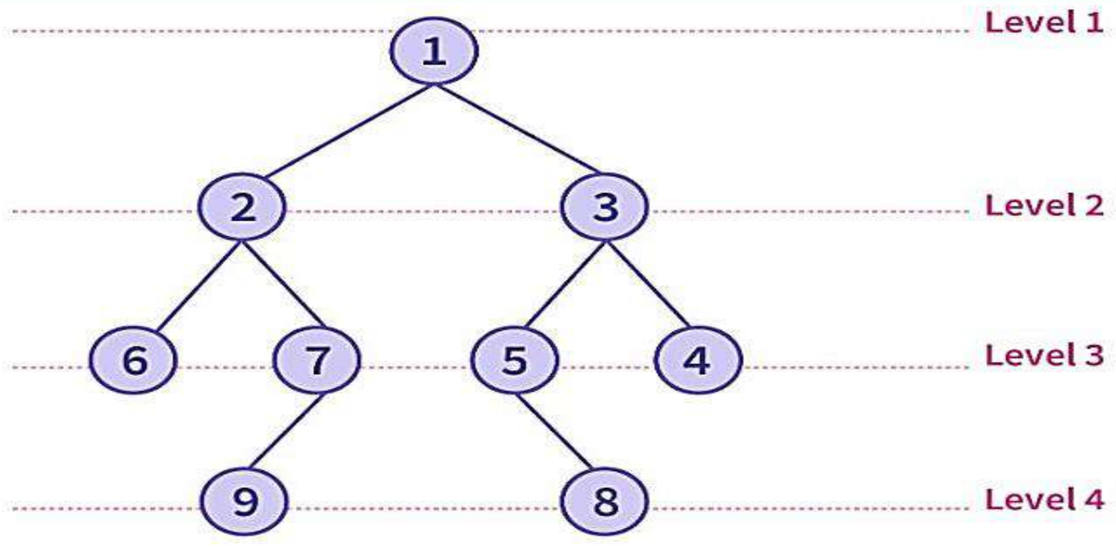
Time Complexity: O(n)

Space Complexity : O(n)

Explanation:

We can define the level of a binary tree as the set of nodes that have an equal number of edges between themselves and the root node (that is, nodes equidistant from the root).

For example, consider the tree with root node as 1:



Example of Right View of Binary Tree

Consider again the tree we used above. As we said earlier, the rightmost node of each level will be part of our right view of the binary tree. That is:

Level	Nodes	Rightmost Node
0	1	1
1	2, 3	3
2	6, 7, 5, 4	4
3	9, 8	8

Therefore, the right view of the binary tree is
1 3 4 8

Program:

```
import java.util.*;

class BinaryTreeNode
{
	public int data;
	public BinaryTreeNode left, right;
	public BinaryTreeNode(int data)
	{
		this.data = data;
		left = null;
		right = null;
	}
}
```

```
class Solution
{
        public List<Integer> rightSideView(BinaryTreeNode root)
        {
                List<Integer> result = new ArrayList<Integer>();
                rightView(root, result, 0);
                return result;
        }

        public void rightView(BinaryTreeNode curr, List<Integer> result, int currDepth)
        {
                if((curr == null)||(curr.data==-1))     return;

                if(currDepth == result.size())          result.add(curr.data);

                rightView(curr.right, result, currDepth + 1);
                rightView(curr.left, result, currDepth + 1);
        }
}

public class RightSideView
{
        static BinaryTreeNode root;
        static BinaryTreeNode temp = root;
        void insert(BinaryTreeNode temp, int key)
        {
                if (temp == null) {
                        root = new BinaryTreeNode(key);
                        return;
                }
                Queue<BinaryTreeNode> q = new LinkedList<BinaryTreeNode>();
                q.add(temp);

                // Do level order traversal until we find an empty place.
                while (!q.isEmpty())
                {
                        temp = q.remove();
                        if (temp.left == null)
                        {
                                temp.left = new BinaryTreeNode(key);
                                break;
                        }
                        else
                                q.add(temp.left);
                        if (temp.right == null) {
                                temp.right = new BinaryTreeNode(key);
                                break;
                        }
                        else
                                q.add(temp.right);
                }
        }

        public static void main(String args[])
        {
                Scanner sc=new Scanner(System.in);
                String str[]=sc.nextLine().split(" ");
                RightSideView bt=new RightSideView();
                root=new BinaryTreeNode(Integer.parseInt(str[0]));

                for(int i=1; i<str.length; i++)
                        bt.insert(root,Integer.parseInt(str[i]));

                Solution sol= new Solution();
                System.out.println(sol.rightSideView(root));
        }
}
```

Sample Input-1:

```
1 2 3 5 -1 -1 5
```

Sample Output-1:

```
[1, 3, 5]
```

Sample Input-2:

```
3 2 -1 1 -1 -1 -1 4 5
```

Sample Output-2:

```
[3, 2, 1, 5]
```

Backtracking

General method:

Backtracking can be defined as a general algorithmic technique that considers searching every possible solution in order to solve a computational problem.

This technique involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem.

For example, to search for a path in a maze or solving puzzles like Sudoku, when a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

For Backtracking problems, the algorithm finds a path sequence for the solution of the problem that keeps some check-points, i.e., a point from where the given problem can take a backtrack if no workable solution is found out for the problem.

Types of Backtracking Problems

Problems associated with backtracking can be categorized into 3 categories:

Decision Problems: Here, we search for a feasible solution.

Optimization Problems: For this type, we search for the best solution.

Enumeration Problems: We find set of all possible feasible solutions to the problems of this type.

Basic Terminologies

Candidate: A candidate is a potential choice or element that can be added to the current solution.

Solution: The solution is a valid and complete configuration that satisfies all problem constraints.

Partial Solution: A partial solution is an intermediate or incomplete configuration being constructed during the backtracking process.

Decision Space: The decision space is the set of all possible candidates or choices at each decision point.

Decision Point: A decision point is a specific step in the algorithm where a candidate is chosen and added to the partial solution.

Feasible Solution: A feasible solution is a partial or complete solution that adheres to all constraints.

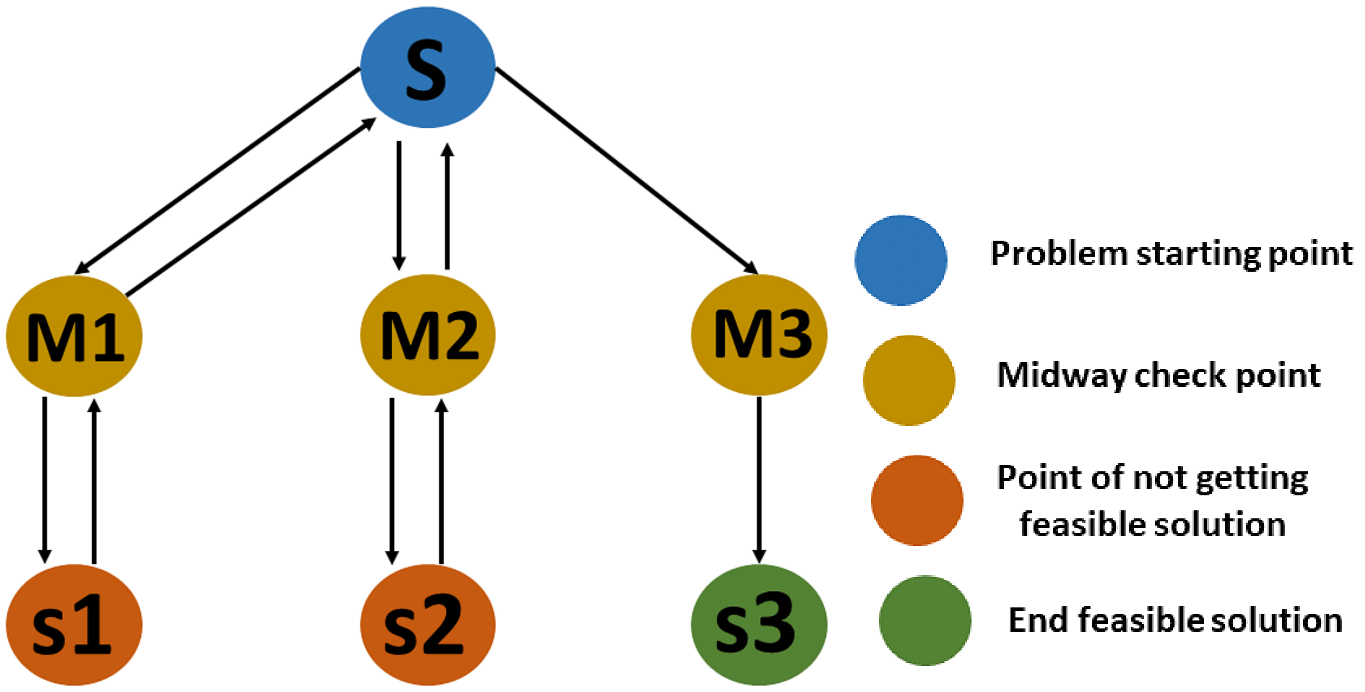
Dead End: A dead end occurs when a partial solution cannot be extended without violating constraints.

Backtrack: Backtracking involves undoing previous decisions and returning to a prior decision point.

Search Space: The search space includes all possible combinations of candidates and choices.

Optimal Solution: In optimization problems, the optimal solution is the best possible solution.

Backtracking can be easily understood using the following diagram



1. In this case, S represents the problem's starting point. You start at S and work your way to solution S1 via the midway point M1. However, you discovered that solution S1 is not a viable solution to our problem. As a result, you **backtrack** (return) from S1, return to M1, return to S, and then look for the feasible solution S2. This process is repeated until you arrive at a workable solution.

2. S1 and S2 are not viable options in this case. According to this example, only S3 is a viable solution. When you look at this example, you can see that we go through all possible combinations until you find a viable solution. As a result, you refer to **backtracking** as a brute-force algorithmic technique.

3. A "**space state tree**" is the above tree representation of a problem. It represents all possible states of a given problem (*solution or non-solution*).

The final algorithm is as follows:

Step 1: Return success if the current point is a viable solution.

Step 2: Otherwise, if all paths have been exhausted (i.e., the current point is an endpoint), return failure because there is no feasible solution.

Step 3: If the current point is not an endpoint, backtrack and explore other points, then repeat the preceding steps.

Applications:

1. **N Queens Problem.**
2. **Hamiltonian Cycle.**
3. **Brace Expression.**
4. **Gray Code.**
5. **Path with Maximum Gold.**
6. **Generalized Abbreviation.**
7. **Campus Bikes II.**

1. N Queens Problem:

N Queen problem is the classical Example of backtracking.

N-Queen problem is defined as, **"**given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal**"**.

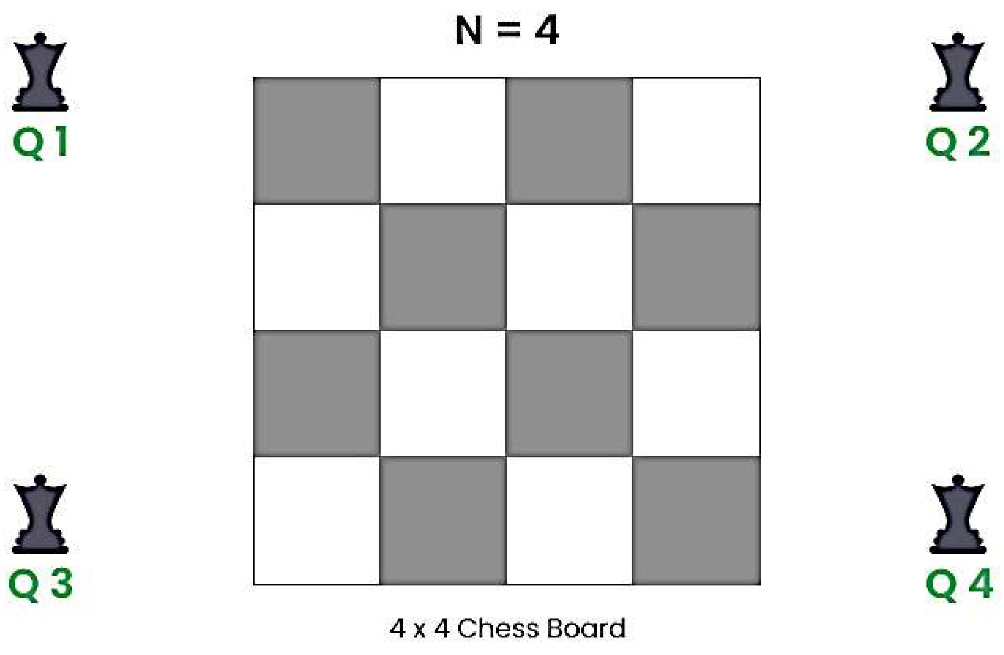
Example:

4-Queen Problem

For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible.

So, we start with N = 4 and we will generalize it for N queens.

Problem : Given 4 x 4 chessboard, arrange 4 - queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.

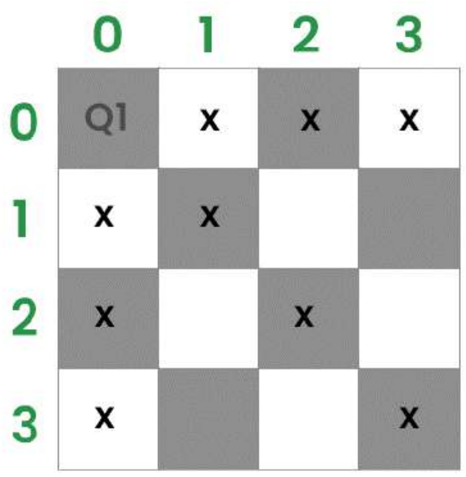


Step 1:

Put our first Queen (Q1) in the (0,0) cell .

'x' represents the cells which is not safe, i.e., they are under attack by the Queen (Q1).

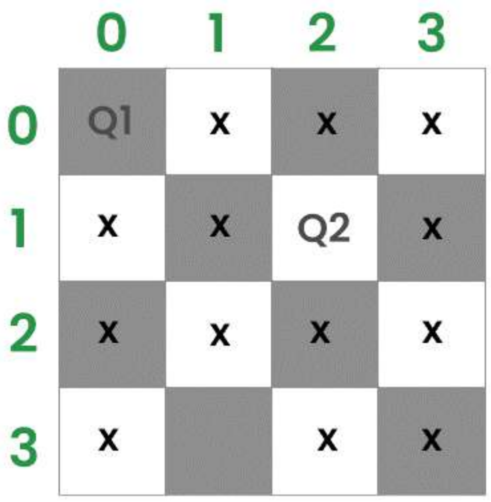
After this move to the next row [0 -> 1].



Step 2:

Put our next Queen (Q2) in the (1,2) cell .

After this move to the next row [1 -> 2].



Step 3:

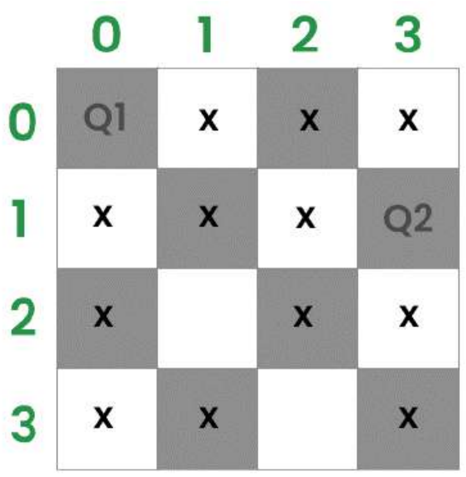
At row 2 there is no cell which are safe to place Queen (Q3) .

So, backtrack and remove queen Q2 queen from cell (1, 2) .

Step 4:

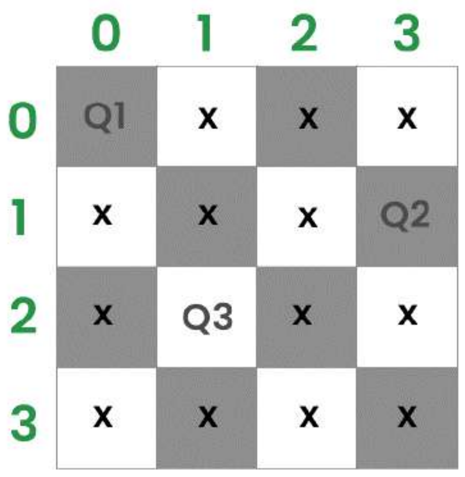
There is still a safe cell in the row 1 i.e., cell (1, 3).

Put Queen (Q2) at cell (1, 3).



Step 5:

Put queen (Q3) at cell (2, 1).



Step 6:

There is no any cell to place Queen (Q4) at row 3.

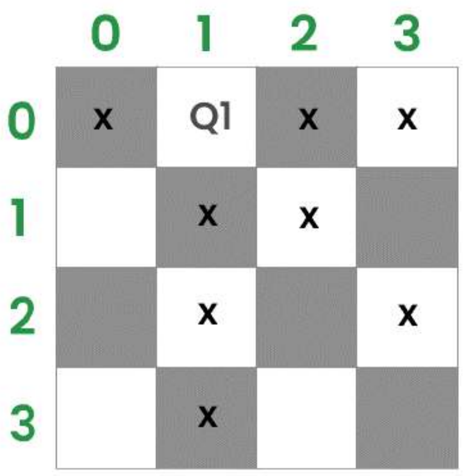
Backtrack and remove Queen (Q3) from row 2.

Again, there is no other safe cell in row 2, So backtrack again and remove queen (Q2) from row 1.

Queen (Q1) will be removed from cell (0,0) and moved to next safe cell i.e. (0 , 1).

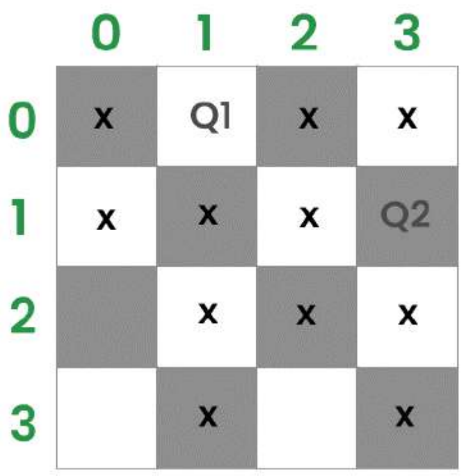
Step 7:

Place Queen Q1 at cell (0 , 1), and move to next row.



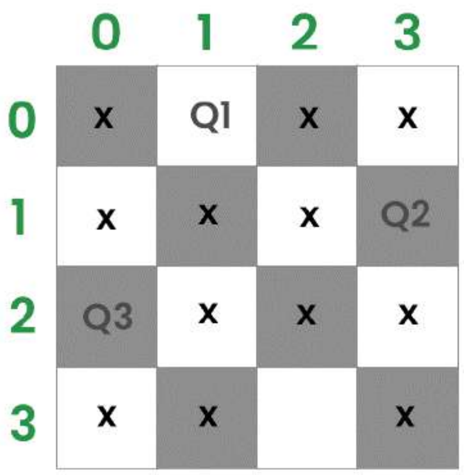
Step 8:

Place Queen Q2 at cell (1 , 3), and move to next row.



Step 9:

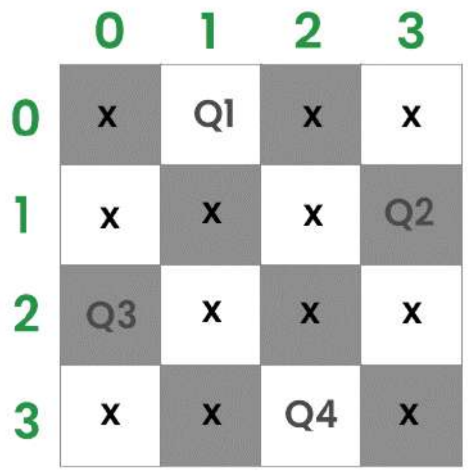
Place Queen Q3 at cell (2 , 0), and move to next row.



Step 10:

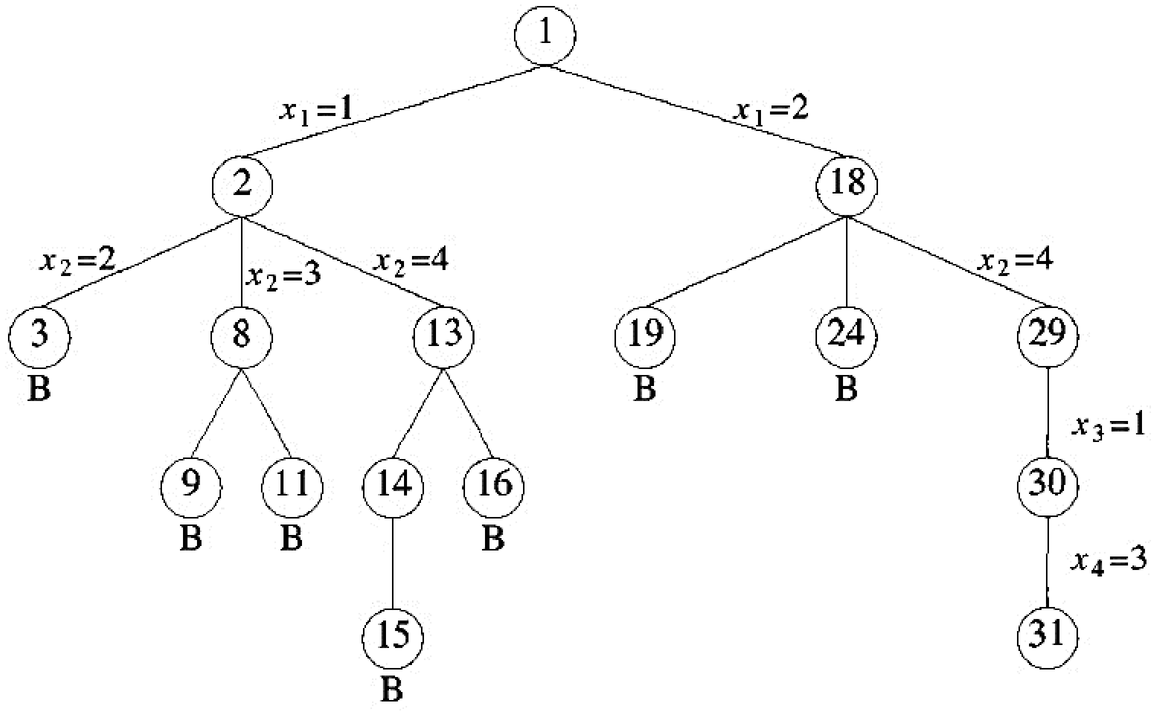
Place Queen Q4 at cell (3 , 2), and move to next row.

This is one possible configuration of solution



	0	1	2	3
0	x	Q1	x	x
1	x	x	x	Q2
2	Q3	x	x	x
3	x	x	Q4	x

The bellow diagram describes the backtracking sequence for the 4-queen problem.



➢ The solution of the 4-queen problem can be seen as four tuples (x_1, x_2, x_3, x_4), where x_i represents the column number of queen Q_i. Two possible solutions for the 4-queen problem are **(2, 4, 1, 3) and (3, 1, 4, 2).**

Time Complexity: O(N!)
Auxiliary Space: O(N^2)

<u>**Java Program for N-Queen Application:**</u> **NQueenProblem.java**

```
import java.util.*;
public class NQueenProblem {
    int N;
    void printSolution(int board[][])
```

```
{
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++)
                        System.out.print(board[i][j]);
                System.out.println();
        }
}

boolean isSafe(int board[][], int row, int col)
{
        int i, j;
        for (i = 0; i < col; i++)
                if (board[row][i] == 1)
                        return false;

        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
                if (board[i][j] == 1)
                        return false;

        for (i = row, j = col; j >= 0 && i < N; i++, j--)
                if (board[i][j] == 1)
                        return false;
        return true;
}

boolean solveNQUtil(int board[][], int col)
{
        if (col >= N)
                return true;
        for (int i = 0; i < N; i++) {
                if (isSafe(board, i, col)) {
                        board[i][col] = 1;

                        if (solveNQUtil(board, col + 1) == true)
                                return true;

                        board[i][col] = 0;
                }
        }

        return false;
}

boolean solveNQ()
{
        int board[][] = new int[N][N];
        if (solveNQUtil(board, 0) == false) {
                System.out.print("No Solution");
                return false;
        }
```

```
			printSolution(board);
			return true;
	}

	public static void main(String args[])
	{
			Scanner sc=new Scanner(System.in);
			NQueenProblem Queen = new NQueenProblem();
			Queen.N=sc.nextInt();
			Queen.solveNQ();
	}
}
```

Sample Input-1:

```
4
```

Sample Output-1:

```
0010
1000
0001
0100

(Or)
0100
0001
1000
0010
```

Sample Input-2:

```
3
```

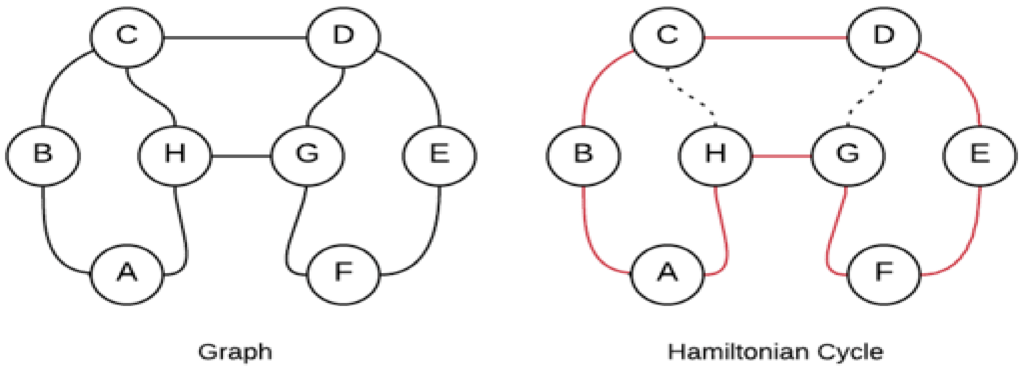
Sample Output-2:

```
No Solution
```

Hamiltonian Cycle

*The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges*

- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – C – D – E – F—G—H—A forms a Hamiltonian cycle. It visits all the vertices exactly once.



- The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, “Given a path, is it a Hamiltonian cycle of the graph?”.
- The optimization problem is stated as, “Given graph G, find the Hamiltonian cycle for the graph.”
- We can define the constraint for the Hamiltonian cycle problem as follows:
  - In any path, vertex i and (i + 1) must be adjacent.
  - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
  - Vertex i must not appear in the first (i – 1) vertices of any path.
- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Complexity Analysis:

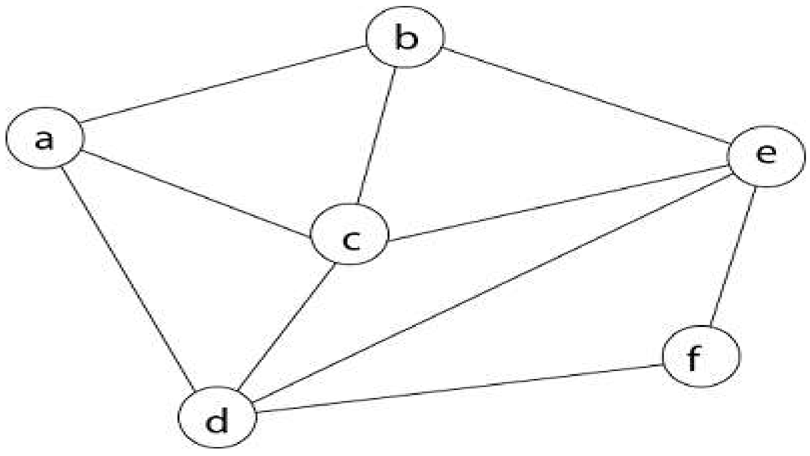
Looking at the state space graph, in worst case, total number of nodes in tree would be,

$T(n) = 1 + (n – 1) + (n – 1)^2 + (n – 1)^3 + \dots + (n – 1)^{n – 1}$

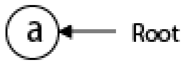
= frac{(n-1)^n - 1}{n - 2}=*frac*(*n*−1)*n*−1*n*−2

$T(n) = O(n^n)$. Thus, the Hamiltonian cycle algorithm runs in exponential time.

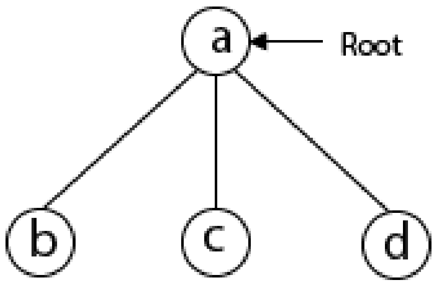
Example: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



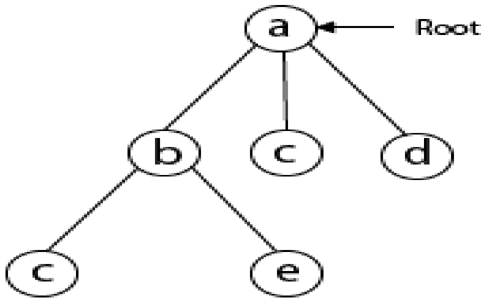
Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



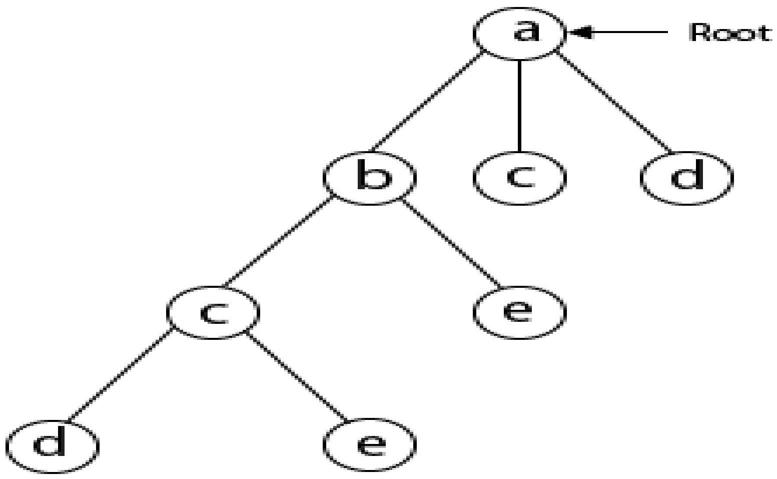
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



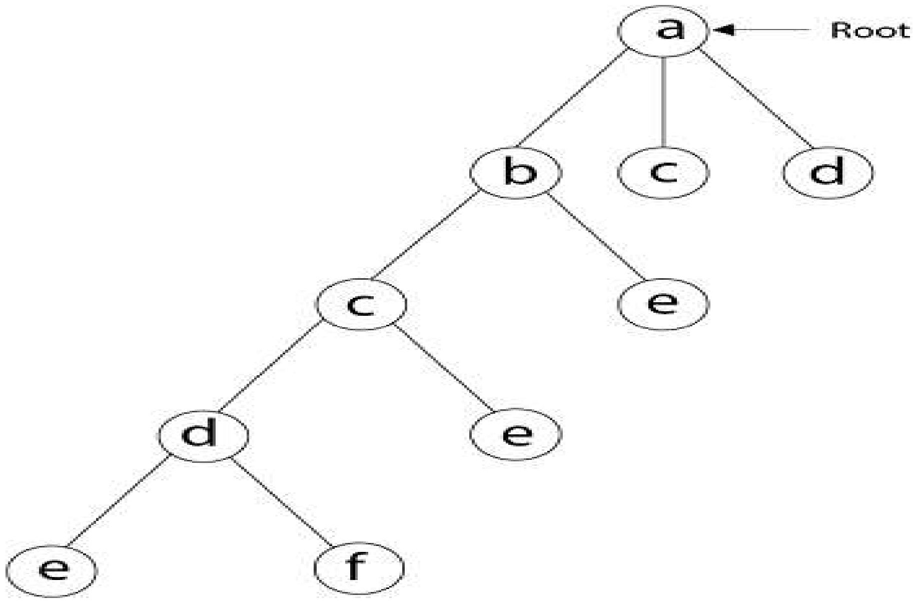
Next, we select 'c' adjacent to 'b.'



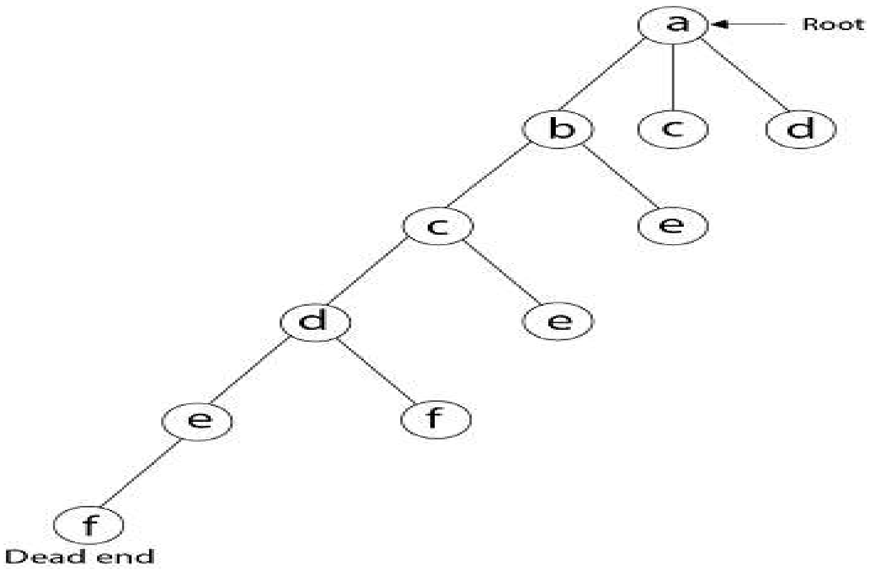
Next, we select 'd' adjacent to 'c.'



Next, we select 'e' adjacent to 'd.'

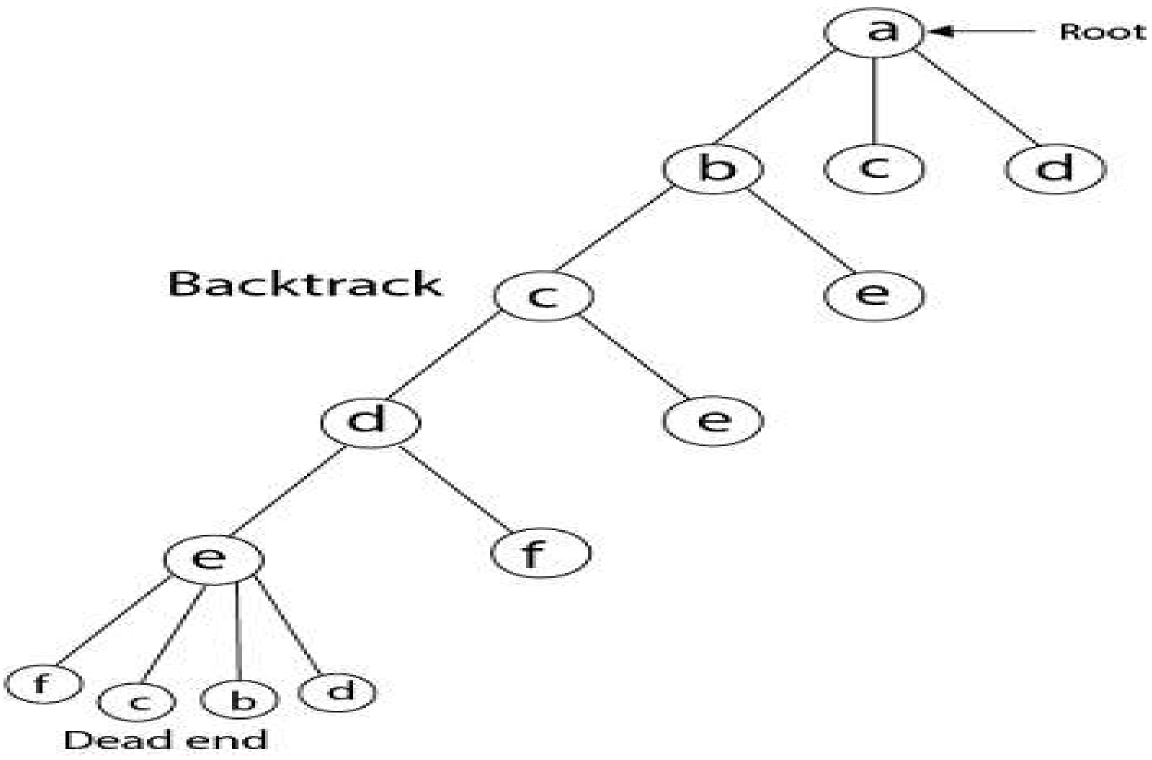


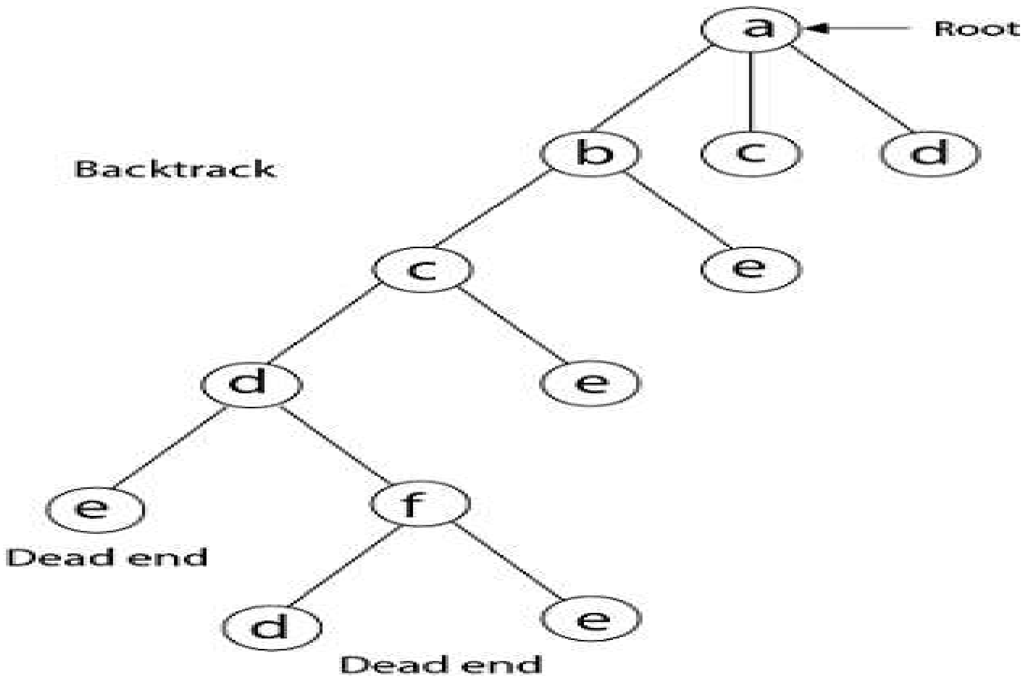
Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



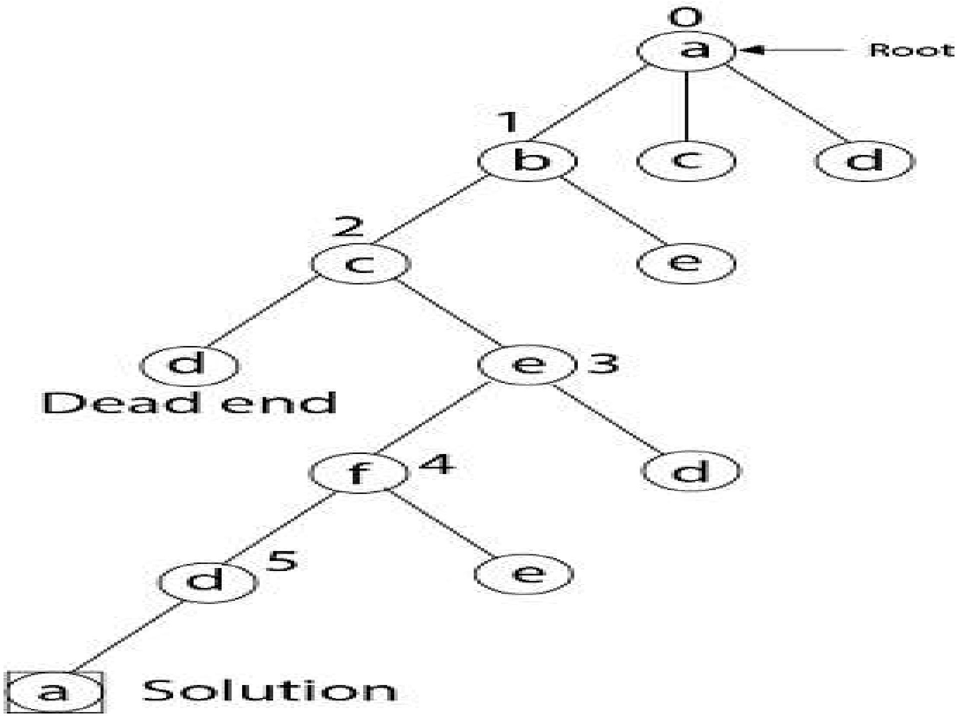
From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).





Again Backtrack



Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Program

```
import java.util.*;

class HamiltonianCycle
{
    static int V;
    int path[];

    /* A utility function to check if the vertex v can be added at index 'pos'
    in the Hamiltonian Cycle constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
            // Check if this vertex is an adjacent vertex of the previously added vertex.
            if (graph[path[pos - 1]][v] == 0)
                    return false;

            /* Check if the vertex has already been included.
            This step can be optimized by creating an array of size V */
            for (int i = 0; i < pos; i++)
                    if (path[i] == v)
                            return false;

            return true;
    }

    // A recursive utility function to solve hamiltonian cycle problem
    boolean hamCycleUtil(int graph[][], int path[], int pos)
    {
            // base case: If all vertices are included in Hamiltonian Cycle
            if (pos == V)
            {
                    // And if there is an edge from the last included vertex to the first vertex
                    if (graph[path[pos - 1]][path[0]] == 1)
                            return true;
                    else
                            return false;
            }

            // Try different vertices as a next candidate in Hamiltonian Cycle.
            // We don't try for 0 as we included 0 as starting point in hamCycle()
            for (int v = 1; v < V; v++)
            {
                    // Check if this vertex can be added to Hamiltonian  Cycle
                    if (isSafe(v, graph, path, pos))
                    {
                            path[pos] = v;

                            // recur to construct rest of the path
                            if (hamCycleUtil(graph, path, pos + 1) == true)
                                    return true;
                            // If adding vertex v doesn't lead to a solution, then remove it
                            path[pos] = -1;
                    }
            }
    /* If no vertex can be added to Hamiltonian Cycle constructed so far, then return false */
            return false;
    }

            /* This function solves the Hamiltonian Cycle problem using Backtracking. It mainly uses
            hamCycleUtil() to solve the problem. It returns false if there is no Hamiltonian Cycle possible,
            otherwise return true and prints the path. Please note that there may be more than one solutions,
            this function prints one of the feasible solutions. */
    int hamCycle(int graph[][])
    {
            path = new int[V];
            for (int i = 0; i < V; i++)
                    path[i] = -1;

            /* Let us put vertex 0 as the first vertex in the path. If there is a Hamiltonian Cycle, then the path
            can be started from any point of the cycle as the graph is undirected */
            path[0] = 0;
            if (hamCycleUtil(graph, path, 1) == false)
            {
                    System.out.println("\nSolution does not exist");
                    return 0;
            }

            printSolution(path);
            return 1;
    }

    /* A utility function to print solution */
    void printSolution(int path[])
    {
            System.out.println("Solution Exists: Following is one Hamiltonian Cycle");
            for (int i = 0; i < V; i++)
                    System.out.print(path[i] + " ");

            // Let us print the first vertex again to show the complete cycle
            System.out.println(path[0]);
    }

    public static void main(String args[])
    {
            Scanner sc=new Scanner(System.in);
            V = sc.nextInt();
            int graph[][]=new int[V][V];

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

```
                    for(int j=0;j<V;j++)
                            graph[i][j]=sc.nextInt();

            HamiltonianCycle obj = new HamiltonianCycle();
            obj.hamCycle(graph);
    }
}
```

```
Sample Input-1:
---------------
5
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 1
0 1 1 1 0

Sample Output-1:
----------------
0  1  2  4  3  0

Sample Input-2:
---------------
5
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 0
0 1 1 0 0

Sample Output-2:
----------------
No Solution
```

Brace Expansion

Under the grammar given below, strings can represent a set of lowercase words. Let R(expr) denote the set of words the expression represents.

The grammar can best be understood through simple examples:

- Single letters represent a singleton set containing that word.
  - R("a") = {"a"}
  - R("w") = {"w"}
- When we take a comma-delimited list of two or more expressions, we take the union of possibilities.
  - R("{a,b,c}") = {"a","b","c"}
  - R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains each word at most once)
- When we concatenate two expressions, we take the set of possible concatenations between two words where the first word comes from the first expression and the second word comes from the second expression.
  - R("{a,b}{c,d}") = {"ac","ad","bc","bd"}
  - R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg", "acefh"}

Formally, the three rules for our grammar:

- For every lowercase letter x, we have R(x) = {x}.
- For expressions e_1, e_2, ... , e_k with k >= 2, we have R({e_1, e_2, ...}) = R(e_1) ∪ R(e_2) ∪ ...
- For expressions e_1 and e_2, we have R(e_1 + e_2) = {a + b for (a, b) in R(e_1) × R(e_2)}, where + denotes concatenation, and × denotes the cartesian product.

Given an expression representing a set of words under the given grammar, return *the sorted list of words that the expression represents*.

Example 1:

Input: expression = "{a,b}{c,{d,e}}"

Output: ["ac","ad","ae","bc","bd","be"]

Example 2:

Input: expression = "{{a,z},a{b,c},{ab,z}}"

Output: ["a","ab","ac","z"]

Explanation: Each distinct word is written only once in the final answer.

Constraints:

- 1 <= expression.length <= 60
- expression[i] consists of '{', '}', ','or lowercase English letters.
- The given expression represents a set of words based on the grammar given in the description.

<u>Java Program for BraceExpansion</u>:   BraceExpression.java

```
case =1
input =[a,b,c,d]e[x,y,z]
output =[aex, aey, aez, bex, bey, bez, cex, cey, cez, dex, dey, dez]

case =2
input =[ab,cd]x[y,z]
output =[abxy, abxz, cdxy, cdxz]
```

Program:

```java
import java.util.*;

public class BraceExpression
{
	public static  String[] expand(String expr)
	{
		// TreeSet to sort
		TreeSet<String> set = new TreeSet<>();
		if (expr.length() == 0)
		{
			return new String[]{""};
		}
		else if (expr.length() == 1)
		{
			return new String[]{expr};
		}
		if (expr.charAt(0) == '[')
		{
			int i = 0; // keep track of content in the "[content]"
			while (expr.charAt(i) != ']')
			{
				i++;
			}
			String sub = expr.substring(1, i);
			String[] subs = sub.split(",");

			String[] strs = expand(expr.substring(i + 1)); // dfs
			for (int j = 0; j < subs.length; j++)
			{
				for (String str : strs) {
					set.add(subs[j] + str);
				}
			}
		}
		else
		{
			String[] strs = expand(expr.substring(1));
			for (String str : strs)
```

```
                        {
                                    set.add(expr.charAt(0) + str);
                        }
                }
        return set.toArray(new String[0]);
        }

        public static void main(String args[] )
        {
                Scanner sc = new Scanner(System.in);
                String str=sc.next();
                System.out.println(Arrays.deepToString(expand(str)));
        }
}
```

Sample Input-1:
[b]c[e,g]k

Sample Output-1:
[bcek, bcgk]

Sample Input-2:
[a,b][c,d]

Sample Output-2:
[ac, ad, bc, bd]

Sample Input-3:
[xyz]a[b,c]

Sample Output-3:
[xyzab, xyzac]

Gray Code

An **n-bit gray code sequence** is a sequence of 2^n integers where:

- Every integer is in the **inclusive** range $[0, 2^n - 1]$,
- The first integer is 0,
- An integer appears **no more than once** in the sequence,
- The binary representation of every pair of **adjacent** integers differs by **exactly one bit**, and
- The binary representation of the **first** and **last** integers differs by **exactly one bit**.

Given an integer n, return any valid **n-bit gray code sequence**.

Example 1:

Input: n = 2

Output: [0,1,3,2]

Explanation:

The binary representation of [0,1,3,2] is [00,01,11,10].

- 0$\underline{0}$ and 0$\underline{1}$ differ by one bit

- $\underline{0}$1 and $\underline{1}$1 differ by one bit

- 1$\underline{1}$ and 1$\underline{0}$ differ by one bit

- $\underline{1}$0 and $\underline{0}$0 differ by one bit

[0,2,3,1] is also a valid gray code sequence, whose binary representation is [00,10,11,01].

- $\underline{0}$0 and $\underline{1}$0 differ by one bit

- 1$\underline{0}$ and 1$\underline{1}$ differ by one bit

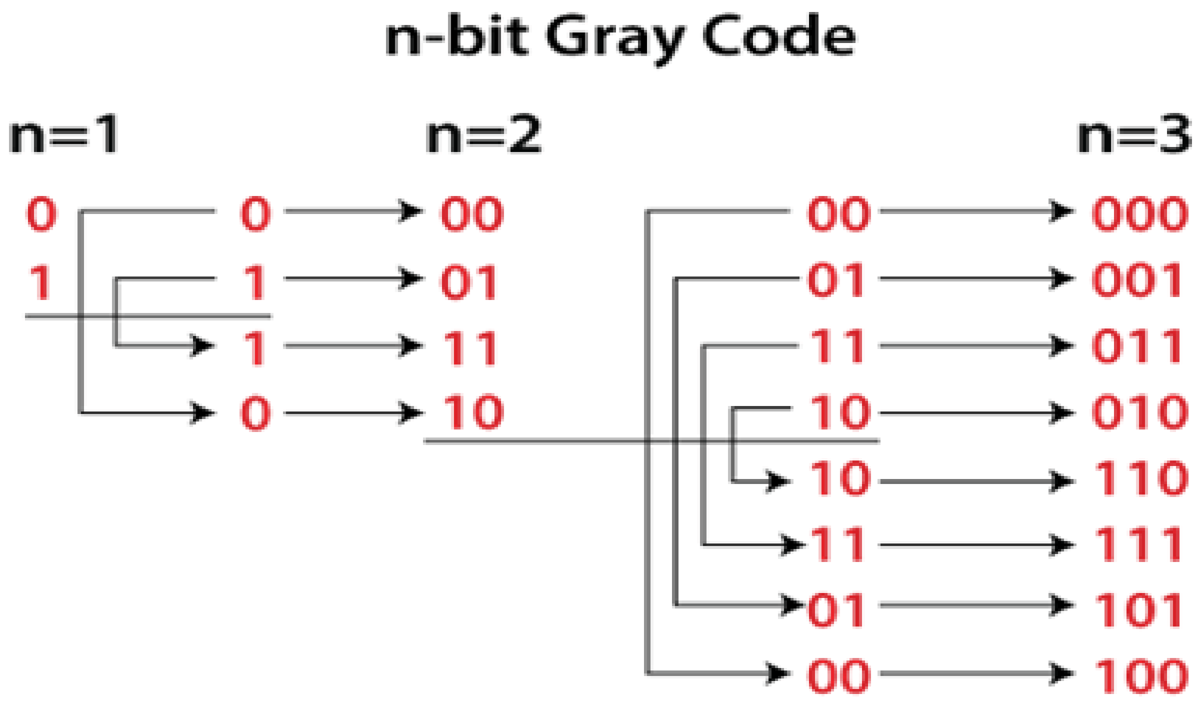
- $\underline{1}$1 and $\underline{0}$1 differ by one bit

- 0$\underline{1}$ and 0$\underline{0}$ differ by one bit

Example 2:

Input: n = 1

Output: [0,1]


n-bit Gray Code
n=1
n=2
n=3
0
1
0
1
1
0
00
01
11
10
00
01
11
10
10
11
01
00
000
001
011
010
110
111
101
100

Program

```
import java.util.*;
class GrayCode
{
	int nums = 0;
	public List<Integer> grayCode(int n)
	{
		List<Integer> result = new ArrayList<>();
		result.add(0);
		// Keeps track of the numbers present in the current sequence.
		Set<Integer> isPresent = new HashSet<>();

		// All Gray code sequence starts with 0
		isPresent.add(0);
		grayCodeHelper(result, n, isPresent);
		return result;
	}

	/*Find each possible number (next) that can be generated by changing one bit of the last number in the
	   result list (current). We do so by toggling the ith bit at each iteration. Since the maximum possible
	   number of bits present in any number of the sequence is n, we need to toggle n bits.*/
	private boolean grayCodeHelper(List<Integer> result, int n, Set<Integer> isPresent)
	{
		if (result.size() == (1 << n))
		return true;

		int current = result.get(result.size() - 1);

		for (int i = 0; i < n; i++)
		{
			int next = current ^ (1 << i);
			if (!isPresent.contains(next))
			{
				isPresent.add(next);
				result.add(next);
				// If valid sequence found no need to search any further
				if (grayCodeHelper(result, n, isPresent))
						return true;
			}
		}
		return false;
	}
	public static void main( String args[])
	{
		Scanner sc=new Scanner(System.in);
		int N=sc.nextInt();
		System.out.println("\n" + new GrayCode().grayCode(N));
	}
}
```

Path with Maximum Gold

In a gold mine grid of size m x n, each cell in this mine has an integer representing the amount of gold in that cell, 0 if it is empty.

Return the maximum amount of gold you can collect under the conditions:

- Every time you are located in a cell you will collect all the gold in that cell.
- From your position, you can walk **one step to the left, right, up, or down.**
- You can't visit the same cell more than once.
- Never visit a cell with 0 gold.
- You can start and stop collecting gold from **any** position in the grid that has some gold.

Example 1:

Input: grid = [[0,6,0],[5,8,7],[0,9,0]]

Output: 24

Explanation:

```
[[0,6,0],
 [5,8,7],
 [0,9,0]]
```

Path to get the maximum gold, 9 -> 8 -> 7.

Example 2:

Input: grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]

Output: 28

Explanation:

```
[[1,0,7],
 [2,0,6],
 [3,4,5],
 [0,3,0],
 [9,0,20]]
```

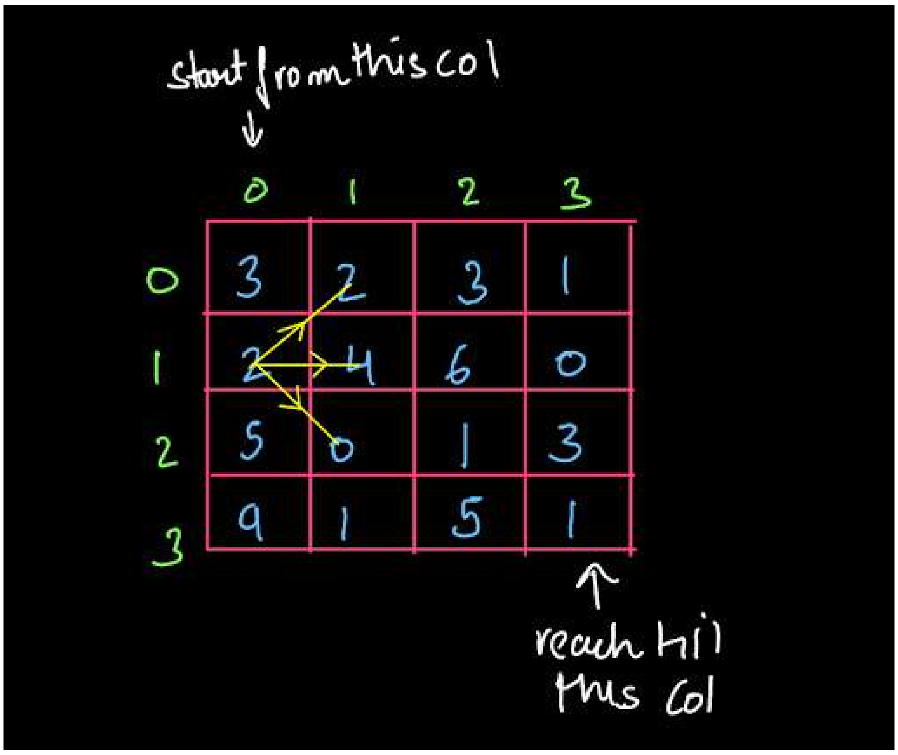
Path to get the maximum gold, 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7.

Constraints:

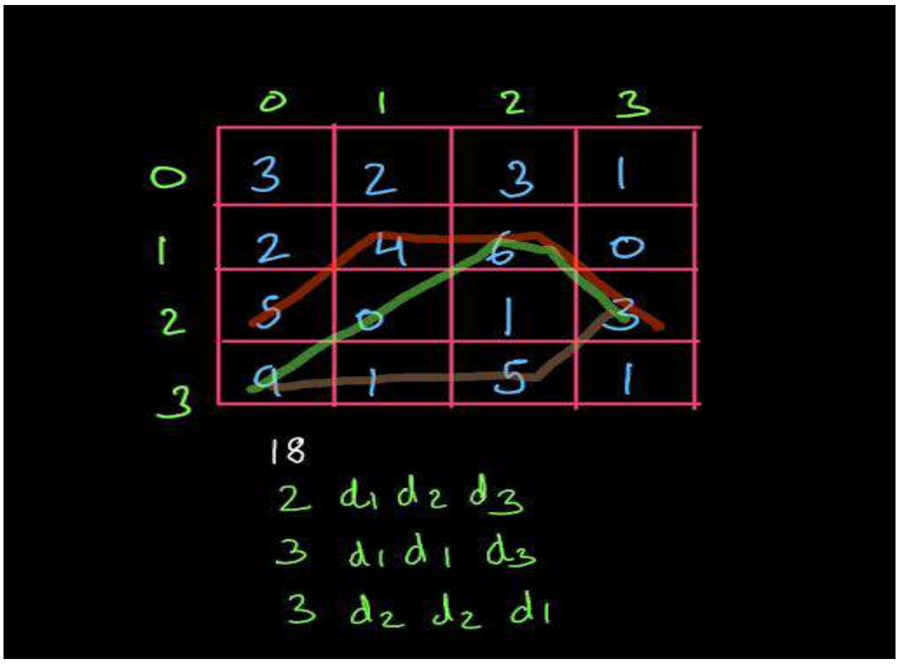
- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 15
- 0 <= grid[i][j] <= 100
- There are at most **25** cells containing gold.

Time Complexity: O(n x m) where n is the number of rows and m is the number of columns of the given input matrix.

Space Complexity: O(n x m)



- Each cell in this matrix has an amount of gold present at it. We have to start digging from the 0th column and reach the last column and in the procedure, we can move either diagonally one step upward (called "d1") or one step forward in the same row (called "d2") or diagonally one step downward (called "d3") collecting the gold at the cell we reach. So, we have to find the paths with the maximum gold.
- For instance, in the matrix shown above, the maximum gold that we can collect is 18 and there are 3 different paths to reach 18 as shown in the image below:



- As you can see, the maximum amount of gold is 18. The first path is "2 d1 d2 d3". This means that we are starting from the 2nd row and the path is "d1 d2 d3".
- We have already discussed the meaning of "d1", "d2" and "d3". Why are we mentioning only the starting row and not the starting column in the paths? This is because we have already been told in the question that we have to start from the 0th column only.

<u>Java program for Path with Maximum Gold</u>: **GetMaximumGold.java**

```
import java.util.*;
class GetMaximumGold
{
	public int getMaximumGold(int[][] grid)
	{
		int maxGold = 0;
		for (int i = 0; i < grid.length; i++)
		{
			for (int j = 0; j < grid[0].length; j++)
			{
			   maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j, 0));
			}
		}
		return maxGold;
	}
	    // i = row, j = column
	private int getMaximumGoldBacktrack(int[][] grid, int i, int j, int curGold)
	{
		if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0)
			       return curGold;
		curGold += grid[i][j];
		int temp = grid[i][j];
		int maxGold = curGold;

		grid[i][j] = 0;
		maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i+1, j, curGold));
		// next row
		maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j+1, curGold));
		// next column
		maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i-1, j, curGold));
		 // previous row
		maxGold = Math.max(maxGold, getMaximumGoldBacktrack(grid, i, j-1, curGold));
		// previous column
		grid[i][j] = temp;

		return maxGold;
	}
	    public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		int m=sc.nextInt();
		int n=sc.nextInt();
		int grid[][]=new int[m][n];
		for(int i=0;i<m;i++)
			for(int j=0;j<n;j++)
					grid[i][j]=sc.nextInt();
		System.out.println(new GetMaximumGold().getMaximumGold(grid));
	}
}
```

Output:

```
input =3 3
0 6 0
5 8 7
0 9 0
output =24

case =2
input =5 3
1 0 7
2 0 6
3 4 5
0 3 0
9 0 20
output =28
```

Generalized Abbreviation:

Write a function to generate the generalized abbreviations of a word.

- **Note:** The order of the output does not matter.
- **Example:**
- **Input:** "word"
  Output:
  ["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Understanding the problem:

1. You are given a word.
2. You have to generate all abbreviations of that word.

For example:
Sample Input-> pep

Sample Output-> pep   pe1 p1p   p2   1ep 1e1 2p 3 (in different lines)
HOW?
First of all, generate all the binaries of the length equal to the length of the input string.

Binaries -

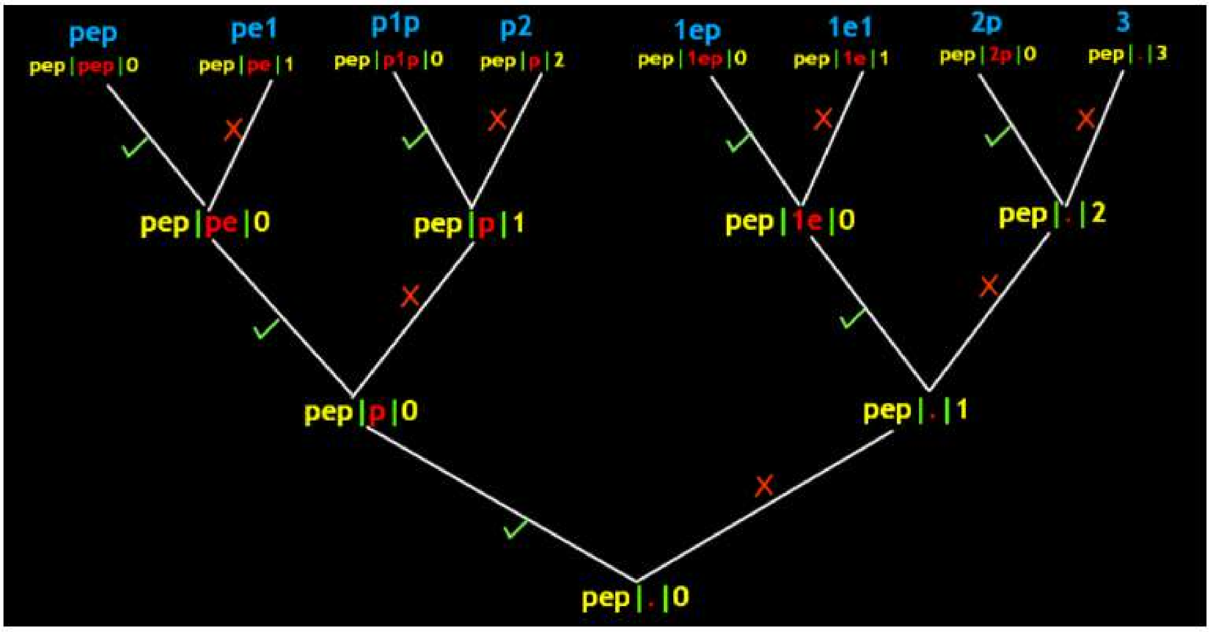
000 -> pep
001 -> pe1
010 -> p1p
011 -> p2
100 -> 1ep
101 -> 1e1
110 -> 2p
111 -> 3

Approach:

- We can observe that whenever the bit is OFF (zero), we will use the character of the string at that position and whenever the bit is ON (one), we will count all the ON (one) bits that are together and then replace them with the count of the ON (one) bits.
- The number of abbreviations will be equal to 2^n, where n = length of the string (as the number of binaries with given n is equal to 2^n).
- In the above example n = 3, therefore the number of abbreviations will be 8 (2^3 = 8).

- We can see that this question is like the subsequence problem, we must maintain one more variable which will count the characters that were not included in the subsequence.

Let's try to draw the recursion tree diagram for the word, "pep". In recursion diagram, (a | b | c) a represents the test string, b represents the answer so far and c represents the count.



- In the above diagram, (pep|.|0) has two options, either it can include the first p in the answer or not. If it includes the p in the answer then p is concatenated in the answer so far string and count remains 0.
- Then at (pep|p|0), we further explore the options for character e. It has two options, either it can include e in the answer or not. If it includes e in the answer then e is concatenated in the answer so far string and count remains 0.
- Then at (pep|pe|0), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If it includes p in the answer then p is concatenated in the answer so far string and count remains 0. **Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. pep is the first possible output.**
- Then we come back at (pep|pe|0) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 1. **And again, we have reached the end of the string, but count is 1 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, pe1.**
- We have explored both the possibilities via (pep|pe|0) therefore, we go back at (pep|p|0), and see what happens if e was not added to asf.
- First of all count becomes 1, and we reach (pep|p|1).
- Then at (pep|p|1), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If p is to be included in the answer then we see that our previous count is not 0, so we concatenate the previous count in the asf and then second p. **Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. p1p is the next possible output.**
- Then we come back at (pep|p|1) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 2.

And again, we have reached the end of the string, but count is 2 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, p2.

- We have explored both the possibilities via (pep|p|1) therefore; we go back at (pep|p|0).
- (pep|p|0) has also explored both the options, if "e" were added or not) therefore; we go back at (pep|.|0).
- At (pep|.|0), now we explore the possibilities of, if first p were not added in the asf. If "p" is not added in the answer so far then count becomes 1.
- Then at (pep|.|1), we further explore the options for character e. It has two options, either it can include e in the answer or not. If it includes e in the answer then e is concatenated with the current count in the answer so far string and count is set to 0.
- Then at (pep|1e|0), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If it includes p in the answer then p is concatenated in the answer so far string and count remains 0.
- Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. 1ep is the next possible output.
- Then we come back at (pep|1e|0) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 1. **And again, we have reached the end of the string, but count is 1 this time, therefore, this count will be concatenated at the end of the asf, which will give us the next possible output, 1e1.**
- We have explored both the possibilities via (pep|1e|0) therefore, we go back at (pep|.|1), and see what happens if e was not added to asf.
- Then at (pep|.|2), we further explore the options for character p (second p). It has two options, either it can include p in the answer or not. If p is to be included in the answer then we see that our previous count is not 0, so we concatenate the previous count (2) in the asf and then second p. **Since we have reached the end of the string, and the count is 0, therefore, the asf i.e. 2p is the next possible output.**
- Then we come back at (pep|.|2) and explore the path if the second p is not added in the asf. In this case, p will not be added in psf and therefore count will become 3. **And again, we have reached the end of the string, but count is 3 this time, therefore, this count will be concatenated at the end of the asf (empty at this stage), which will give us the next possible output, 3.**
- We have explored both the possibilities via (pep|.|2) therefore; we go back at (pep|.|1).
- (pep|.|1) has also explored both the options, if "e" were added or not) therefore; we go back at (pep|.|0).
- At (pep|.|0), we have explored both the possibilities that is if first p were not added in the asf.

Java Program for Generalized Abbreviations: **GenerateAbbreviations.java**

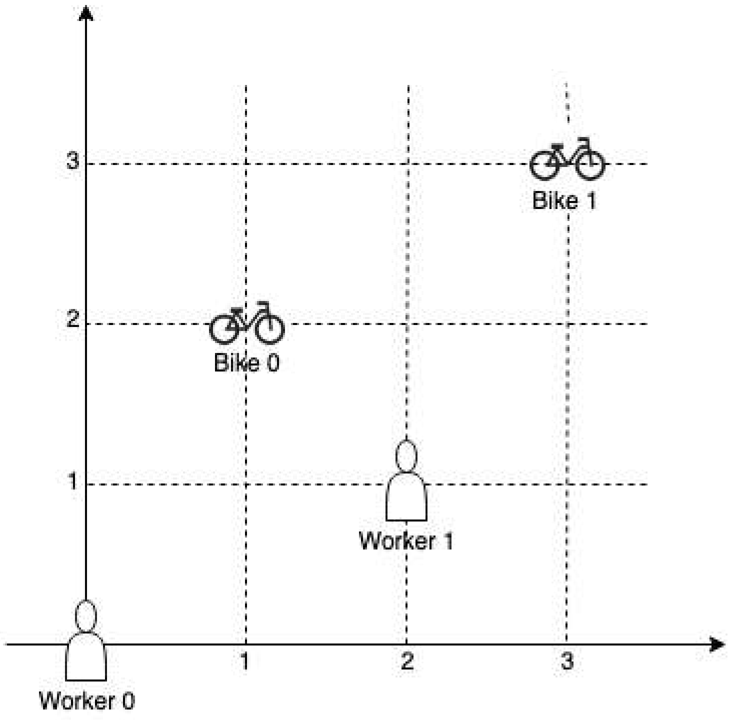
```
import java.util.*;
class GenerateAbbreviations
{
        public List<String> makeShortcutWords(String word)
        {
                List<String> ans = new ArrayList<String>();
                backtrack(ans, new StringBuilder(), word, 0, 0);
                return ans;
        }

            /*i is the current position, k is the count of consecutive abbreviated characters*/
        private void backtrack(List<String> ans, StringBuilder builder, String word, int i, int k)
        {
                int len = builder.length(); // keep the length of builder
                if(i == word.length())
                {
                        if (k != 0)
                                    builder.append(k); // append the last k if non zero
                        ans.add(builder.toString());
                }
                else
                {
                        // the branch that word.charAt(i) is abbreviated
                        backtrack(ans, builder, word, i + 1, k + 1);

                        // the branch that word.charAt(i) is kept
                        if (k != 0)
                                    builder.append(k);
                        builder.append(word.charAt(i));
                        backtrack(ans, builder, word, i + 1, 0);
                    }
                    builder.setLength(len); // reset builder to the original state
        }

        public static void main(String args[])
        {
                Scanner sc=new Scanner(System.in);
                String s=sc.next();
                System.out.println(new GenerateAbbreviations().makeShortcutWords(s));
        }
}
```

```
case =1
input =kmit
output =[1m1t, 1m2, 1mi1, 1mit, 2i1, 2it, 3t, 4, k1i1, k1it, k2t, k3, km1t, km2, kmi1, kmit]

case =2
input =cse
output =[1s1, 1se, 2e, 3, c1e, c2, cs1, cse]

case =3
input =elite
output =[1l1t1, 1l1te, 1l2e, 1l3, 1li1e, 1li2, 1lit1, 1lite, 2i1e, 2i2, 2it1, 2ite, 3t1, 3te, 4e, 5, e1i1e, e1i2,
    e1it1, e1ite, e2t1, e2te, e3e, e4, el1t1, el1te, el2e, el3, eli1e, eli2, elit1, elite]

case =4
input =r
output =[1, r]
```

Campus Bikes II

- On a campus represented as a 2D grid, there are N workers and M bikes, with N <= M. Each worker and bike is a 2D coordinate on this grid.
- We assign one unique bike to each worker so that the sum of the Manhattan distances between each worker and their assigned bike is minimized.
- The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.
- Return the minimum possible sum of Manhattan distances between each worker and their assigned bike.

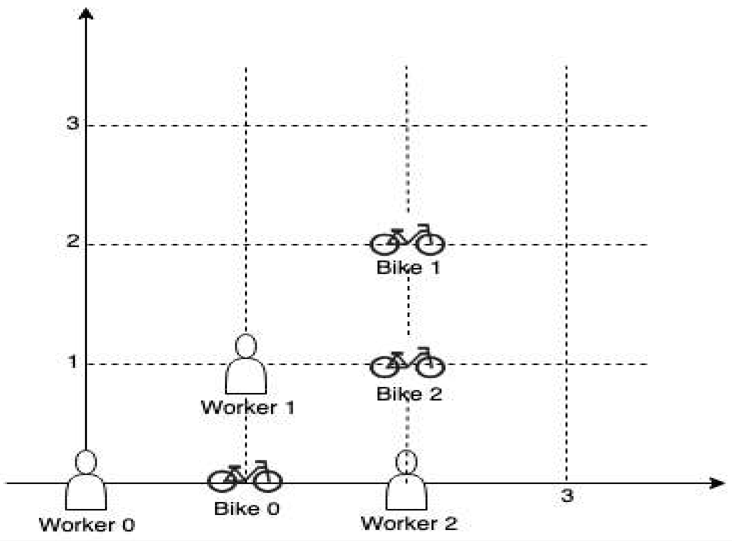
Example 1:



Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]
Output: 6
Explanation:

We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan distance of both assignments is 3, so the output is 6.

Example 2:



Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

Output: 4

Explanation:

We first assign bike 0 to worker 0, then assign bike 1 to worker 1 or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead to sum of the Manhattan distances as 4.

Note:

1. 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000
2. All worker and bike locations are distinct.
3. 1 <= workers.length <= bikes.length <= 10

Solution:

To solve this, we will follow these steps −

```
➢ Define a function helper(). This will take a,b
    o return |a[0]-b[0]| + |a[1] - b[1]|
➢ Define a function solve(). This will take bikes, workers,bikev,i:= 0
➢ info := a list with i and bikev
➢ if info is present in memo, then
    o return memo[info]
➢ if i is same as size of workers, then
    o return 0
➢ temp := infinity
➢ for j in range 0 to size of bikes, do
    o if not bikev[j] is non-zero, then
        ▪ bikev[j]:= 1
        ▪ temp := minimum of temp, helper(workers[i], bikes[j]) +solve(bikes, workers, bikev, i+1)
        ▪ bikev[j]:= 0
➢ memo[info]:= temp
```

- return temp
- Define a function assignBikes(). This will take workers, bikes
- bikev := a list whose size is same as the size of bikes, fill this with false
- memo:= a new map
- return solve(bikes, workers, bikev)

<u>Java program for Campus Bikes-II</u>: **CampusBikes.java**

```
import java.util.*;
class CampusBikes
{
	int min = Integer.MAX_VALUE;
	public int assignBikes(int[][] officers, int[][] bikes)
	{
		backtrack(new boolean[bikes.length], 0, officers, bikes, 0);
		return min;
	}

	void backtrack(boolean[] visited, int i, int[][] officers, int[][] bikes, int distance)
	{
		if (i == officers.length && distance < min)
			min = distance;

		if (i >= officers.length)
			return;

		if (distance > min)
			return;

		for (int j=0; j<bikes.length; j++)
		{
			if (visited[j]) continue;
			visited[j] = true;
			    backtrack(visited, i+1, officers, bikes, distance + calcdist(i, j, officers, bikes));
			visited[j] = false;
		}
	}

	int calcdist(int i, int j, int[][] officers, int[][] bikes)
	{
		return Math.abs(officers[i][0]-bikes[j][0])+Math.abs(officers[i][1]-bikes[j][1]);
	}

    public static void main(String[] args)
    {
	    Scanner sc=new Scanner(System.in);
	    int m=sc.nextInt();
	    int n=sc.nextInt();
	    int bikes[][]=new int[n][2];
	    int men[][]=new int[m][2];
	    for(int i=0;i<m;i++)
```

```
                {
                men[i][0]=sc.nextInt();
                men[i][1]=sc.nextInt();
            }
            for(int i=0;i<n;i++)
                {
                bikes[i][0]=sc.nextInt();
                bikes[i][1]=sc.nextInt();
            }
            System.out.println(new CampusBikes().assignBikes(men,bikes));
    }
}
```

Sample Input-1:

```
3 3    //No of workers and vehicles
0 1     // co-ordinates of workers
1 2
1 3
4 5     // co-ordinates of vehicles
2 5
3 6
```

Sample Output-1:

```
17
```

Sample Input-2:

```
2 2 //No of workers and vehicles
0 0     // co-ordinates of workers
2 1
1 2     // co-ordinates of vehicles
3 3
```

Sample Output-2:

```
6
```