**TREE QUESTIONS**

Table of Contents

[**Theory** 2](#_Toc157576486)

[What is Tree data structure? 2](#_Toc157576487)

[What are components of tree? 2](#_Toc157576488)

[Binary Tree 3](#_Toc157576489)

[Tree traversal 3](#_Toc157576490)

[Binary Search Tree 4](#_Toc157576491)

[Use cases of Tree and its use in system design 4](#_Toc157576492)

[Array representation of Binary Tree 4](#_Toc157576493)

[**Important Codes** 6](#_Toc157576494)

[Binary Search Tree Implementation: 6](#_Toc157576495)

[Traversals: 7](#_Toc157576496)

[Breadth First : 8](#_Toc157576497)

[Morris Traversal: 9](#_Toc157576498)

[LEVEL 1: **EASY** 12](#_Toc157576499)

[1. Check if two binary trees are same or not. 12](#_Toc157576500)

[3. Merge two binary trees. 12](#_Toc157576501)

[4. Convert a BST to tree with only right nodes. 12](#_Toc157576502)

[5. Average of levels in binary tree. 12](#_Toc157576503)

[6. Sorted array to BST 12](#_Toc157576504)

[7. Binary Tree tilt 12](#_Toc157576505)

[8. Diameter of binary tree 12](#_Toc157576506)

[9. Subtree of another tree 12](#_Toc157576507)

[LEVEL 2: **Medium** 13](#_Toc157576508)

[LEVEL 3: **Difficult** 14](#_Toc157576509)

[**SOLUTIONS:** 15](#_Toc157576510)

[**LEVEL 1:** 15](#_Toc157576511)

# **Theory**

### 1.1 What is a Graph?

* **Definition**: A graph G=(V,E)G = (V, E)G=(V,E) consists of a set of vertices VVV and a set of edges EEE.
* **Types of Graphs**:
  + **Directed vs. Undirected**: In directed graphs, edges have direction, whereas in undirected graphs, edges do not.
  + **Weighted vs. Unweighted**: In weighted graphs, edges have weights associated with them, representing costs, lengths, etc.
  + **Simple vs. Multi Graphs**: Simple graphs do not have multiple edges between the same vertices or self-loops.

### 1.2 Basic Terminology

* **Vertex (Node)**: An individual entity in a graph.
* **Edge (Link)**: Connection between two vertices.
* **Degree**: Number of edges connected to a vertex.
  + **In-degree** and **Out-degree** in directed graphs.
* **Path**: Sequence of edges connecting vertices.
* **Cycle**: Path that starts and ends at the same vertex.
* **Connected Graph**: There is a path between every pair of vertices.
* **Component**: A maximally connected subgraph.

### 1.3 Graph representation

# Adjacency List Representation

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

# Adjacency Matrix Representation

import numpy as np

adj\_matrix = np.array([

    [0, 1, 1, 0, 0, 0],

    [1, 0, 0, 1, 1, 0],

    [1, 0, 0, 0, 0, 1],

    [0, 1, 0, 0, 0, 0],

    [0, 1, 0, 0, 0, 1],

    [0, 0, 1, 0, 1, 0]

])

### Q1. <https://www.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1>

class Solution:

    def printGraph(self, V : int, edges : List[List[int]]) -> List[List[int]]:  #V= number of vertex

        # code here

        adj = [[] for \_ in range(V)]

        for i in edges:

            adj[i[0]].append(i[1])

            adj[i[1]].append(i[0])

        return adj

### 2.1 Depth-First Search (DFS)

* **Theory**: Explore as far as possible along each branch before backtracking.
* **Algorithm**:
  1. Start at the root (or any arbitrary node).
  2. Mark the node as visited.
  3. Recursively visit all unvisited adjacent nodes.

def dfs(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(start)

    for neighbour in graph[start]:

        if neighbour not in visited:

            dfs(graph, neighbour, visited)

    return visited

# Example Usage

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

dfs(graph, 'A')

### 2.2 Breadth-First Search (BFS)

* **Theory**: Explore all neighbors at the present depth prior to moving on to nodes at the next depth level.
* **Algorithm**:
  1. Start at the root (or any arbitrary node).
  2. Mark the node as visited.
  3. Add the node to the queue.
  4. While the queue is not empty:
     + Dequeue a node.
     + Visit all unvisited adjacent nodes and add them to the queue.

from collections import deque

def bfs(graph, start):

    visited = set()

    queue = deque([start])

    visited.add(start)

    while queue:

        vertex = queue.popleft()

        print(vertex)

        for neighbor in graph[vertex]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

# Example Usage

bfs(graph, 'A')

Q1 . <https://www.geeksforgeeks.org/problems/depth-first-traversal-for-a-graph/1>

class Solution:

    #Function to return a list containing the DFS traversal of the graph.

    def dfsOfGraph(self, V, adj):

        def helper(v,visited):

            global ans

            visited.add(v)

            ans.append(v)

            for neighbour in adj[v]:

                if neighbour not in visited:

                    helper(neighbour,visited)

        global ans

        ans =[]

        visited=set()

        helper(0,visited)  #starting for vertex 0

        return ans

Q2. <https://www.geeksforgeeks.org/problems/bfs-traversal-of-graph/1>

#User function Template for python3

from typing import List

from queue import Queue

from collections import deque

class Solution:

    #Function to return Breadth First Traversal of given graph.

    def bfsOfGraph(self, V: int, adj: List[List[int]]) -> List[int]:

        # start from vertex 0

        queue  = deque([0])

        visited=set()

        ans=[]

        visited.add(0)

        while queue:

            temp = queue.popleft()

            ans.append(temp)

            for neighbour in adj[temp]:

                if neighbour not in visited:

                    visited.add(neighbour)

                    queue.append(neighbour)

        return ans

### 2.1 Bipartite Graph

#### Definition:

A bipartite graph is a type of graph in which the set of vertices can be divided into two disjoint sets UUU and VVV such that every edge connects a vertex in UUU to a vertex in VVV. In other words, no two vertices within the same set are adjacent.

#### Conditions:

1. **Two Colorability**: A graph is bipartite if and only if it is possible to color the vertices using two colors such that no two adjacent vertices have the same color.
2. **No Odd-Length Cycles**: A graph is bipartite if and only if it does not contain any odd-length cycles.(means all graph without cycle or graph with even cycle length are bipartite)

#### Logic to Check if a Graph is Bipartite:

To check if a graph is bipartite, we can use a breadth-first search (BFS) or depth-first search (DFS) approach to attempt to color the graph using two colors. If we can successfully color the graph such that no two adjacent vertices share the same color, then the graph is bipartite. If we encounter a situation where an adjacent vertex needs to be the same color, the graph is not bipartite.

from collections import deque

def is\_bipartite(graph):

    color = {}

    for node in graph:

        if node not in color:

            queue = deque([node])

            color[node] = 0

            while queue:

                current = queue.popleft()

                for neighbor in graph[current]:

                    if neighbor not in color:

                        color[neighbor] = 1 - color[current]

                        queue.append(neighbor)

                    elif color[neighbor] == color[current]:

                        return False

    return True

# Example Usage

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D'],

    'C': ['A', 'D'],

    'D': ['B', 'C']

}

print(is\_bipartite(graph))  # Output: True

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'C'],

    'C': ['A', 'B']

}

print(is\_bipartite(graph))  # Output: False

### Number of provinces

Link: <https://www.geeksforgeeks.org/problems/number-of-provinces/1>

from collections import deque

class Solution:

    def numProvinces(self, adj, V):

        # code here

        #approach: in visited array try to visit all, if not able to visit from sigle vertex, then it is new province

        #here we are given adjacency matrix. lets convert it to adjacency list first

        adj\_l = [[] for \_ in range(V)]

        for i in range(len(adj)):

            for j in range(len(adj[0])):

                if adj[i][j]==1 and i!=j:

                    adj\_l[i].append(j)

        visited = [0]\*V

        count=0

        for i in range(V):

            if visited[i] !=1:

                count+=1

                visited[i]=1

                queue = deque([i])

                while(queue):  #travel all connected nodes of node(i)

                    t = queue.popleft()

                    for neighbour in adj\_l[t]:

                        if visited[neighbour]==0:

                            visited[neighbour]=1

                            queue.append(neighbour)

        return count

### Find number of islands

Link: <https://www.geeksforgeeks.org/problems/find-the-number-of-islands/1>

#User function Template for python3

import sys

sys.setrecursionlimit(10\*\*8)

class Solution:

    #arrays as passed by reference, so changes in visited array made in dfs reflects here too

    def dfs(self,grid,visited,i,j,n,m):

        visited[i][j]=1

        delta = [-1,0,1]  #all neighbour are of type [i+delta][j+delta] , 0-0 is that node itself

        for del\_row in delta:

            for del\_col in delta:

                new\_row = i + del\_row

                new\_col = j + del\_col

                #means neighbour value in grid is 1 and it is not visited

                if(new\_row>=0 and new\_row<n and new\_col>=0 and new\_col<m and grid[new\_row][new\_col]==1 and visited[new\_row][new\_col]==0):

                    visited[new\_row][new\_col]=1

                    self.dfs(grid,visited,new\_row,new\_col,n,m)

    def numIslands(self,grid):

        #we can consider matrix as graph, all connected 1 as one component, and

        #finally we want number of connected components

        n=len(grid)

        m=len(grid[0])

        visited=[[0]\*m for \_ in range(n)]

        count=0

        for i in range(n):

            for j in range(m):

                if grid[i][j]==1 and visited[i][j]!=1:

                    count+=1

                    self.dfs(grid,visited,i,j,n,m)

        return count

### Bipartite Graph

Link: <https://www.geeksforgeeks.org/problems/bipartite-graph/1>

from collections import deque

class Solution:

    def isBipartite(self, V, adj):

        color = {}

        #there can be many disconnected components of graph, therefore running loops to cover all

        for i in range(V):

            if i not in color:

                color[i]=0

                queue = deque([i])

                while queue:

                    node = queue.popleft()

                    for neighbour in adj[node]:

                        if neighbour not in color:

                            color[neighbour] = 1-color[node]

                            queue.append(neighbour)

                        else:

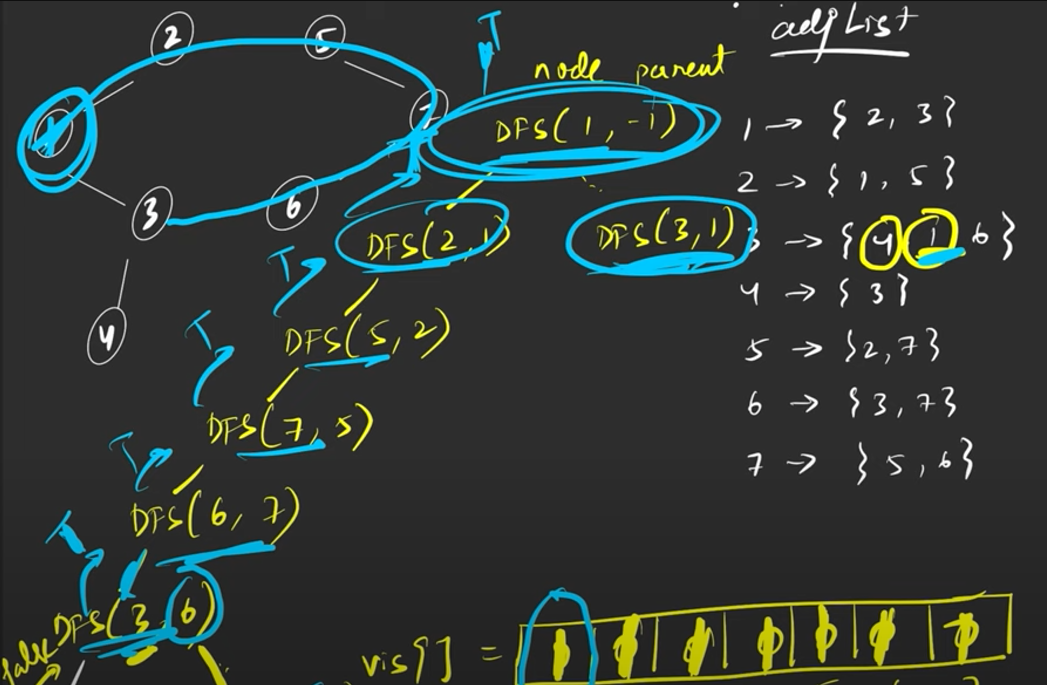
                            if color[neighbour]==color[node]:

                                return False

        return True

### Undirected Graph cycle

Link: <https://www.geeksforgeeks.org/problems/detect-cycle-in-an-undirected-graph/1>

****

from typing import List

class Solution:

    #Function to detect cycle in an undirected graph.

    def dfs(self,node,parent,visited,adj):

        visited.add(node)

        for neighbour in adj[node]:

            if neighbour == parent:

                continue

            elif neighbour in visited:

                return True

            #if by chance we get true, means found cycle, so return it

            #if this subpart is returning false, we have to check for other place. So don't return anything from this subpart

            if self.dfs(neighbour,node,visited,adj):

                return True

        return False

    def isCycle(self, V: int, adj: List[List[int]]) -> bool:

        visited = set()

        for i in range(V):

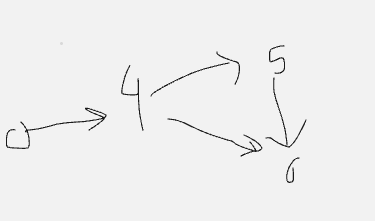
            if i not in visited:

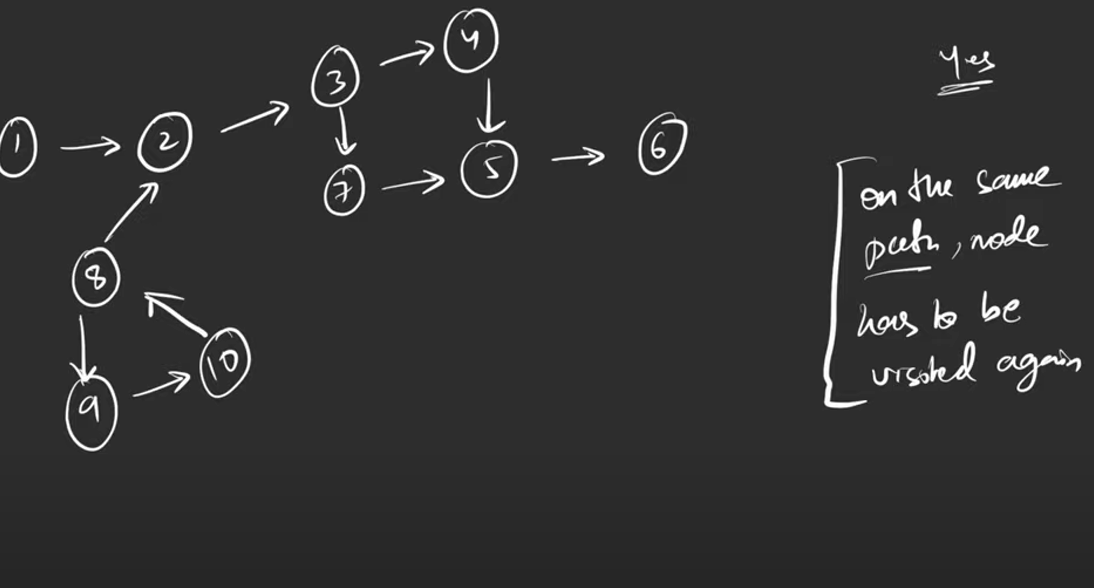
                if self.dfs(i,-1,visited,adj)==True:

                    return True

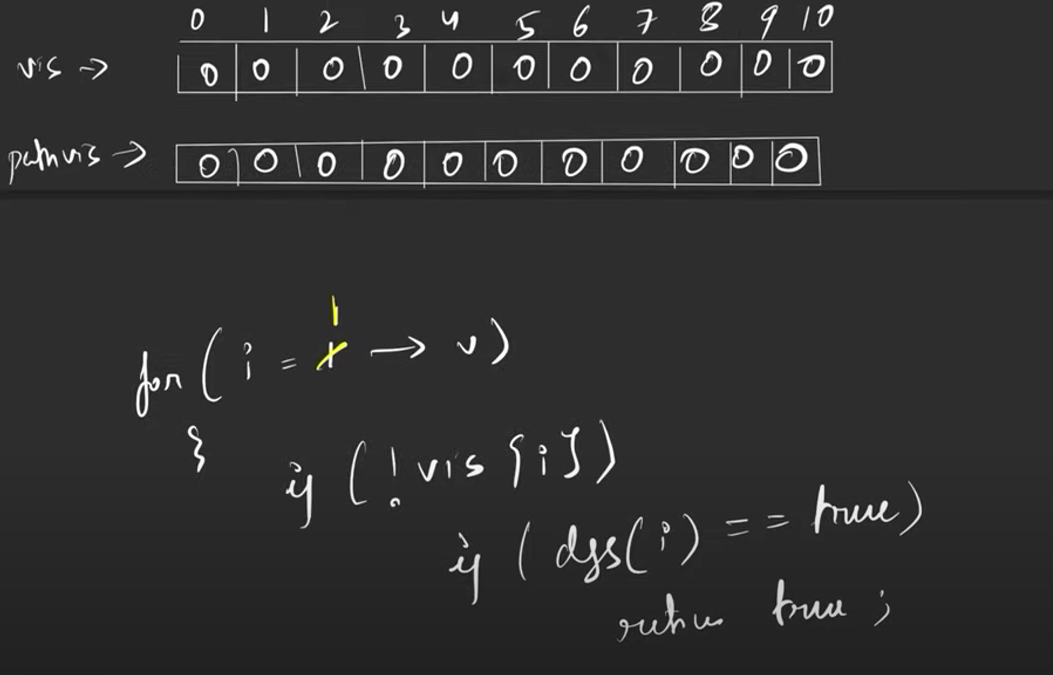
        return False

**For directed graph can’t use this: as for below it gives true though it is not cycle**

****

****

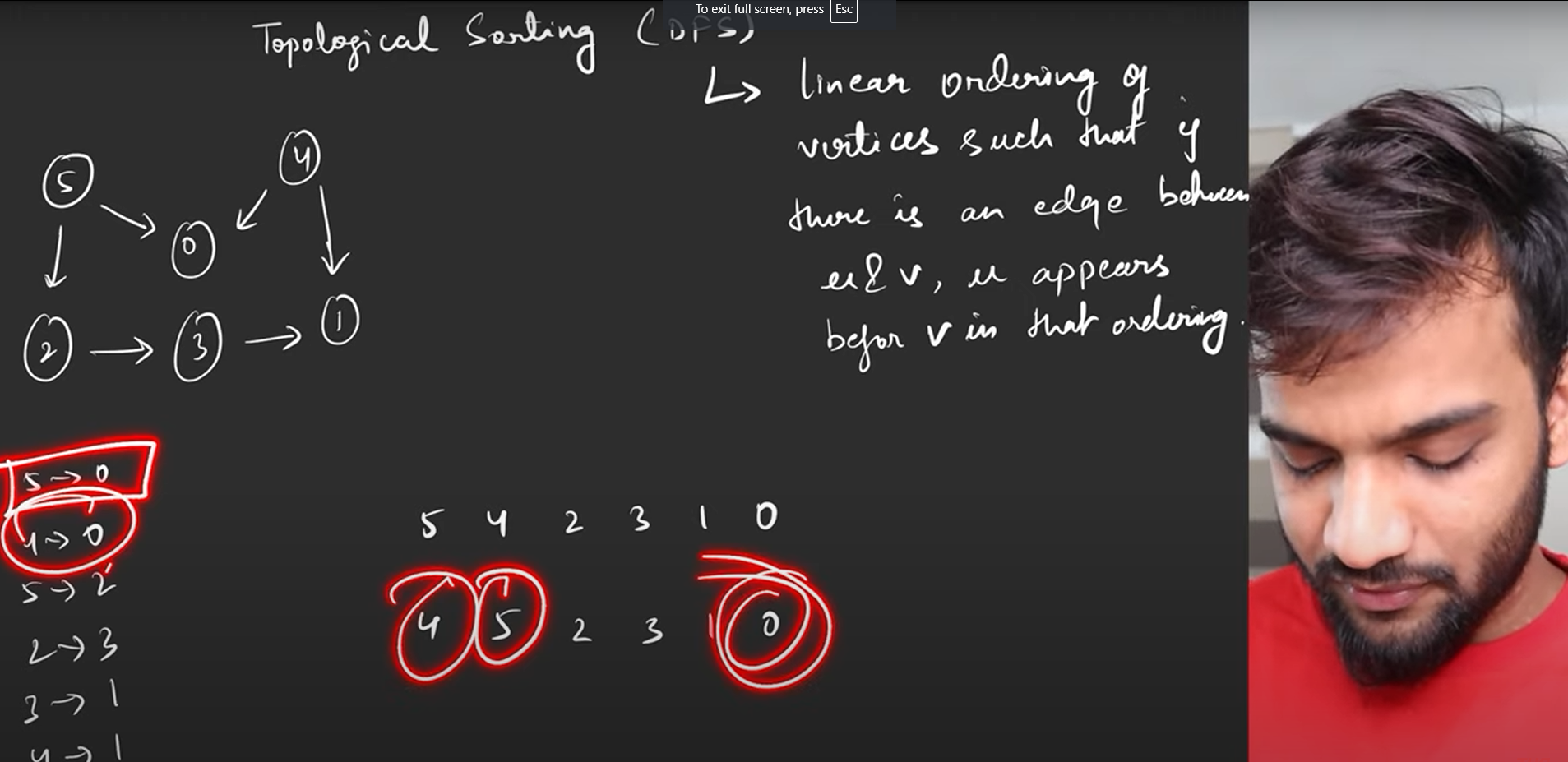
**FOR directed, node should visit in same path again,**



VIDEO

<https://www.youtube.com/watch?v=9twcmtQj4DU&list=PLgUwDviBIf0oE3gA41TKO2H5bHpPd7fzn&index=19>

**TOPOLOGICAL Sort**

****

**Dijkstra Algorithm for shortest path**

import heapq

def dijkstra(graph, start):

    # Initialize distances with infinity for all vertices except the start

    distances = {vertex: float('infinity') for vertex in graph}

    distances[start] = 0

    # Initialize priority queue with the start vertex and distance 0

    priority\_queue = [(0, start)]

    while priority\_queue:

        # Pop the vertex with the smallest distance from the priority queue

        current\_distance, current\_vertex = heapq.heappop(priority\_queue)

        # If the popped distance is greater than the current known distance, skip it

        if current\_distance > distances[current\_vertex]:

            continue

        # Check and update distances for all neighboring vertices

        for neighbor, weight in graph[current\_vertex].items():

            distance = current\_distance + weight

            if distance < distances[neighbor]:

                distances[neighbor] = distance

                heapq.heappush(priority\_queue, (distance, neighbor))

    return distances

# Example Usage

graph = {

    'A': {'B': 1, 'C': 4},

    'B': {'A': 1, 'D': 2, 'E': 5},

    'C': {'A': 4, 'F': 3},

    'D': {'B': 2},

    'E': {'B': 5, 'F': 1},

    'F': {'C': 3, 'E': 1}

}

print(dijkstra(graph, 'A'))

### Explanation:

1. **Initialization**: Distances to all vertices are set to infinity except the start vertex, which is set to 0.
2. **Priority Queue**: A priority queue (min-heap) is initialized with the start vertex.
3. **Processing**: The vertex with the smallest known distance is extracted from the priority queue.
4. **Distance Update**: For each neighbor of the current vertex, the distance is updated if a shorter path is found. The neighbor and the updated distance are then pushed onto the priority queue.
5. **Check and Skip**: If the current distance of the extracted vertex is greater than the known shortest distance, it is skipped.

### Implement Dijkstra Algorithm.

Link: <https://www.geeksforgeeks.org/problems/implementing-dijkstra-set-1-adjacency-matrix/1>

import heapq

class Solution:

    #Function to find the shortest distance of all the vertices

    #from the source vertex S.

    def dijkstra(self, V, adj, S):

        #code here

        distances = [float('inf') for i in range(V)]

        distances[S]=0

        priority\_queue = [(0,S)]

        while priority\_queue:

            curr\_distance, curr\_node = heapq.heappop(priority\_queue)

            if curr\_distance > distances[curr\_node]:

                continue

            #adj list here = (node,weight)

            for neighbour in adj[curr\_node]:

                n\_node, n\_weight = neighbour[0], neighbour[1]

                n\_distance = curr\_distance + n\_weight

                if n\_distance < distances[n\_node]:

                    distances[n\_node] = n\_distance

                    heapq.heappush(priority\_queue,(n\_distance, n\_node))

        return distances

# 

# LEVEL 1: **EASY**

### Check if two binary trees are same or not.

Link: <https://leetcode.com/problems/same-tree/description/>

# LEVEL 2: **Medium**

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **LEVEL 1:**

1. Same tree

class Solution:

    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:

        self.ans=True

        def helper(p,q):

            if not p and not q:

                return

            elif(p is None) or (q is None):

*self*.ans=False

                return

            else:

                helper(p.left,q.left)

                if(p.val!=q.val):

*# print("vd")*

*self*.ans=False

                helper(p.right,q.right)

        helper(p,q)

        return *self*.ans

1. Range sum of bst

[Solution](https://leetcode.com/problems/range-sum-of-bst/solutions/4558641/python-fastest-optimized-with-explanation/)

class Solution:

    def rangeSumBST(self, root: Optional[TreeNode], low: int, high: int) -> int:

        def travel(root):

            global ans

            if (root is None) :

                return

            else:

                if root.val>=low and root.val<=high:

                    ans+=root.val

                if root.val >low:

                    travel(root.left)

                if root.val <high:

                    travel(root.right)

        global ans

        ans=0

        travel(root)

        return ans

1. Pair sum divisible by 5

class Solution:

    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:

        if root1 is None:

            return root2

        if root2 is None:

            return  root1

        root1.val = root1.val + root2.val

        root1.left = *self*.mergeTrees(root1.left,root2.left)

        root1.right = *self*.mergeTrees(root1.right,root2.right)

        return root1

1. Convert BST to inorder tree with only right node

[Solution](https://leetcode.com/problems/increasing-order-search-tree/solutions/4558733/best-python-solutions-2-approach-optimization/)

class Solution:

    def increasingBST(self, root: TreeNode) -> TreeNode:

        def convert\_to\_inorder(root):

            global node

            if root is None:

                return

            else:

                convert\_to\_inorder(root.left)

                x=node

                while(x.right):

                    x=x.right

                x.right=TreeNode(root.val)

                convert\_to\_inorder(root.right)

        global node

        node=TreeNode()

        convert\_to\_inorder(root)

        return node.right

*#Approach 2*

class Solution:

    def increasingBST(self, root: TreeNode) -> TreeNode:

        dummy = curr =TreeNode(None)

        def dfs(root):

            if not root: return

            nonlocal curr

            dfs(root.left)

            curr.right = root

            curr = root

            curr.left = None

            dfs(root.right)

        dfs(root)

        return dummy.right

1. Average of levels in binary tree

[Solution](https://leetcode.com/problems/average-of-levels-in-binary-tree/solutions/4563843/python-code-explanation-general-approach/)

import collections

class Solution:

    def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:

        res=[]

        q = collections.deque()

        q.append(root)

        while(q):

            level=[]

            qlen =len(q)

            for i in range(qlen):

                node = q.popleft()

                if node:

                    level.append(node.val)

                    q.append(node.left)

                    q.append(node.right)

            if level:

                res.append(sum(level)/len(level))

        return res

1. Sorted Array to binary Search tree

[Solution](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/solutions/4565640/python-code-fastest-solution-explanation/)

[](https://www.youtube.com/watch?v=0K0uCMYq5ng)

class Solution:

    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

        def helper(l,r):

            if l>r:

                return

            else:

                m = (l+r)//2

                root = TreeNode(nums[m])

                root.left = helper(l,m-1)

                root.right = helper(m+1,r)

                return root

        return helper(0,len(nums)-1)

1. Binary tree tilt

[Solution](https://leetcode.com/problems/binary-tree-tilt/solutions/4578246/python-2-solutions-easy-one-optimized-version-with-explanation/)

class Solution:

    def findTilt(self, root: TreeNode) -> int:

        def helper(root):

            if not root: return 0

            lv, rv = helper(root.left), helper(root.right)

*self*.ans += abs(lv - rv)

            return root.val + lv + rv

*self*.ans = 0

        helper(root)

        return *self*.ans

1. Diameter Of Binary tree

[Solution](https://leetcode.com/problems/diameter-of-binary-tree/solutions/1143907/python-thought-process/)

[](https://www.youtube.com/watch?v=bkxqA8Rfv04)

def diameterOfBinaryTree(self, root):

    def recurse(node):

        if not node: return 0

        left, right = recurse(node.left), recurse(node.right)

*self*.result = max(*self*.result, left+right)

        return 1 + max(left, right)

*self*.result = 0

    recurse(root)

    return *self*.result

Similar way try to code for finding height of binary tree.

1. Subtree of another tree

class Solution:

    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:

        if subRoot == None :

            return True

        if root == None :

            return False

        if *self*.same(root , subRoot):

            return True

        return *self*.isSubtree(root.left , subRoot) or *self*.isSubtree(root.right , subRoot)

*#This is code to check if one tree is equal to other*

*#check root vals and left and right subtree vals, can't do r==s(this don't work)*

    def same(self , r , s):

        if r == None and s == None :

            return True

        if r and s and r.val == s.val:

            return *self*.same(r.right , s.right) and *self*.same(r.left , s.left)

        return False