



GRAPH

Q: Total no of undirected graph from N nodes:

$$= 2^{(n*(n-1))}$$

```
1 ▶  # } Driver Code Ends
6 #User function Template for python3
7
8 class Solution:
9     def count(self, n):
10         N= (n*(n-1))//2
11         return 2**N
12
13 ▶  # } Driver Code Ends
```

Q: Graph representation in adjacency list:

```
class Solution:

    #Function to return the adjacency list for each vertex.
    def printGraph(self, V, adj):
        ans=[]
        for i in range(V):
            ans.append(adj[i])
            ans[i].insert(0,i)
        return ans
```

Q: BFS traversal:

```

from typing import List
from queue import Queue
class Solution:
    #Function to return Breadth First Traversal of given graph.
    def bfsOfGraph(self, V: int, adj: List[List[int]]) -> List[int]:
        ans=[]
        visited=[0]*len(adj)
        queue=[]
        queue.append(0)
        visited[0]=1
        while(len(queue)!=0):
            node= queue.pop(0)
            ans.append(node)
            for i in adj[node]:
                if visited[i]!=1:
                    queue.append(i)
                    visited[i]=1
        return ans

```

Q:DFS traversal:

```

class Solution:
    #Function to return a list containing the DFS traversal of the graph.
    def dfsOfGraph(self, V, adj):
        visited=[0]*V
        ans=[]
        def dfs(node):
            visited[node]=1
            ans.append(node)
            for j in adj[node]:
                if visited[j]!=1:
                    dfs(j)
        dfs(0)
        return ans

```

**Q: Find the no of component in graph:
Using DFS**

```

class Solution:
    def numProvinces(self, adj, V):
        visited=[0]*V

        def traversal(node): # here i use dfs traversal
            visited[node]=1
            for i in range(V):
                if adj[node][i]==1 and visited[i]!=1:
                    traversal(i)

        cnt=0
        for i in range(V):
            if visited[i]!=1:
                traversal(i)
                cnt+=1

        return cnt

```

Using BFS

```

class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        visited=[0]*len(isConnected)
        queue=[]
        def bfs(node):
            queue.append(node)
            visited[node]=1
            while (len(queue)!=0):
                node=queue.pop(0)
                for i in range(len(isConnected)):
                    if isConnected[node][i]==1 and visited[i]!=1:
                        visited[i]=1
                        queue.append(i)

        cnt=0
        for i in range(len(isConnected)):
            if visited[i]!=1:
                bfs(i)
                cnt+=1
        return cnt

```

Q: Rotten oranges using BFS:

```
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        m= len(grid)
        n= len(grid[0])
        visited= [[-1]*n for i in range(m)]
        queue=[]
        fresh=0          # Keep track of the number of fresh oranges

        # Find initial rotten oranges and mark them as visited and append it in queue with time stamp 0
        for i in range(m):
            for j in range(n):
                if grid[i][j]==2:
                    queue.append([i,j,0])
                    visited[i][j]=2
                elif grid[i][j]==1:
                    fresh+=1

        # No fresh oranges initially, so no time needed for rotting
        if fresh==0:
            return 0

        t=0
        drow=[-1,0,1,0]
        dcol=[0,1,0,-1]
        while(len(queue)!=0):
            r= queue[0][0]
            c= queue[0][1]
            t= queue[0][2]

            queue.pop(0)
            for i in range(4):
                nrow= r+drow[i]
                ncol= c+dcol[i]
                if nrow>=0 and nrow<m and ncol>=0 and ncol<n and visited[nrow][ncol]!=2 and grid[nrow][ncol]==1:
                    queue.append([nrow,ncol,t+1])
                    visited[nrow][ncol]=2
                    fresh-=1

        # There are fresh oranges left, impossible to rot all
        if fresh>0:
            return -1
        return t
```

Q: Flood fill:

Same intuition as rotten oranges and solved by BFS or DFS, i am using BFS to solve it.

```

class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) ->
List[List[int]]:
    m= len(image)
    n= len(image[0])
    q=[]
    if image[sr][sc] == color:
        return image
    #initial color on sr, sc
    color1=image[sr][sc]
    visited=[[-1]*n for i in range(m)]
    q.append([sr,sc])
    image[sr][sc]= color
    visited[sr][sc]=1
    row=[-1,0,1,0]
    col=[0,1,0,-1]
    while(len(q)!=0):
        r, c= q.pop(0)
        for i in range(4):
            nr=r+row[i]
            nc= c+col[i]
            if nr>=0 and nr<m and nc>=0 and nc<n and visited[nr][nc]!=1 and image[nr]
[nc]==color1:
                visited[nr][nc]=1
                q.append([nr,nc])
                image[nr][nc]=color
    return image

```

NEW

Q: Cycle detection in graph , graph may be disconnected also so make sure to detect if any cycle exists.

I am using BFS traversal to detect cycle.

[Detect cycle in an undirected graph | Practice | GeeksforGeeks](#)

Here we are checking that if any node is already visited , then it must be parent thats why it visited , or other case is someone else make it visited at some level so this indicates that cycle exists.

```

from typing import List
class Solution:
    #Function to detect cycle in an undirected graph.
    def isCycle(self, V: int, adj: List[List[int]]) -> bool:
        visited=[0]*V
        q=[]
        def cyclic(ind):
            q.append([ind,-1])
            visited[ind]=1
            while(len(q)!=0):
                node,parent= q.pop(0)
                for i in adj[node]:
                    if visited[i]!=1:
                        visited[i]=1
                        q.append([i,node])
                    elif visited[i]==1 and i!=parent:
                        return True
            return False

        for i in range(V):
            if visited[i]!=1:
                if cyclic(i)==True:
                    return True

        return False

```

Cycle detection using DFS:

Note: Always check

```

from typing import List
class Solution:
    #Function to detect cycle in an undirected graph.
    def isCycle(self, V: int, adj: List[List[int]]) -> bool:
        visited=[0]*V

        def dfs(node,parent):
            visited[node]=1
            for i in adj[node]:
                if visited[i]!=1:
                    if dfs(i,node)==True:
                        return True
                elif i!=parent:
                    return True
            return False

        for i in range(V):
            if visited[i]!=1:
                if dfs(i,-1)==True:
                    return True
        return False

```

Q: Finding cycle in Directed graph:

Intuition behind cycle detection:

If there is already visited node on same path, then there is cycle but if already visited node but on different path, its not cycle.

[Detect Cycle In A Directed Graph - Coding Ninjas](#)

Using DFS: We make two array, visited and pathvis, whatever visited we mark 1 and when on same path mark it 1 but whenever move to another path, then mark pathvis[node] to 0

```

def detectCycleInDirectedGraph(n, edges):
    adj=[[] for i in range(n+1)]    #node starts from 1
    adj[0].append(-1)
    visited=[-1]*(n+1)
    pathvis=[-1]*(n+1)

    for i in range(len(edges)):
        adj[edges[i][0]].append(edges[i][1])

    def dfs(node):
        visited[node]=1
        pathvis[node]=1
        for i in adj[node]:
            if visited[i]==1 and pathvis[i]==1:
                return True
            elif visited[i]==-1:
                if dfs(i)==True:
                    return True
        pathvis[node]=0
        return False

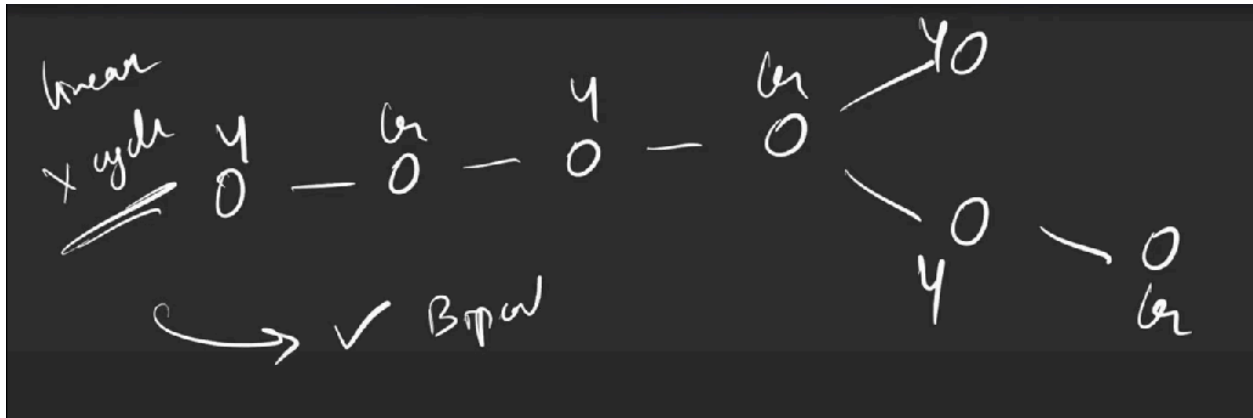
    for i in range(n+1):
        if visited[i]==-1:
            if dfs(i)==True:
                return True
    return False

```

Bipartite Graph:

The graph which can color by two color is known as bipartite graph.

Note: Linear Graph with no cycle is always bipartite.



Q: To check whether a graph is bipartite or not using DFS:

Intuition: If it is colored by 2 color.

We use dfs traversal to check if any node which is traversed and its adjacent is already colored by the same color as the element then return false. It is not bipartite.

```
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n=len(graph)
        color=[-1]*n

        s=[]
        # we will color it by 0 and 1 in adjacent fashion,
        #if we are able to color all node with 0 and 1 so that no two adjacent node have same color
        #then it is bipartite graph
        def dfs(node,col):
            color[node]=col
            for i in graph[node]:
                if color[i] == -1: #yet not being colored
                    if dfs(i,not col)==False:
                        return False
                elif color[i]==col:
                    return False
            return True

        #this for loop check for all the component of graph
        for i in range(n):
            if color[i]==-1:
                if(dfs(i,0)==False):
                    return False

        return True
```

Bipartite using BFS:

Since we perform the BFS traversal for each node, the overall time complexity is $O(V + E)$.

```

class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n=len(graph)
        color=[-1]*n
        q=[]
        # we will color it by 0 and 1 in adjacent fashion,
        #if we are able to color all node with 0 and 1 so that no two adjacent node have same color
        #then it is bipartite graph
        def bfs(node,col):
            q.append([node,col])
            while(len(q)!=0):
                node,col= q.pop(0)
                for i in graph[node]:
                    if color[i] == -1: #yet not being colored
                        color[i]= not col
                        q.append([i,color[i]])
                    elif color[i]==col:
                        return False
                return True
            return True
        #this for loop check for all the component of graph
        for i in range(n):
            if color[i]==-1:
                if(bfs(i,0)==False):
                    return False

        return True

```

Q: Finding no of island:

Intuition: Find the no of island by finding no of component , this can be find by any traversal. I m using bfs to traverse it.

```

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        m=len(grid)
        n=len(grid[0])
        c=0
        #try to solve it by checking no of connected component

        q=[]
        visited= [[-1]*n for i in range(m)]

        def bfs(x,y):
            q.append([x,y])
            row=[-1,1,0,0]
            col=[0,0,-1,1]
            while(len(q)!=0):
                i,j= q.pop(0)
                visited[i][j]=1
                for k in range(4):
                    nrow=row[k]+i
                    ncol=col[k]+j
                    if nrow>=0 and nrow<m and ncol>=0 and ncol<n and visited[nrow][ncol]!=1 and grid[nrow]
[ncol]=='1':
                        q.append([nrow,ncol])
                        visited[nrow][ncol]=1

        for i in range(m):
            for j in range(n):
                if grid[i][j]!='0' and visited[i][j]==-1:
                    bfs(i,j)
                    c+=1

        return c

```

In the worst case, where the grid contains all '1's, each cell will be visited exactly once. Each BFS traversal takes $O(M * N)$ time in the worst case. Since the BFS traversal is performed for each cell, the total time complexity for BFS traversals is $O(M * N)$.

TOPOLOGICAL SORT:

Problem Statement: Given a DAG(Directed Acyclic Graph), print all the vertex of the graph in a topologically sorted order. If there are multiple solutions, print any.

Topological sort using DFS

```
class Solution:
    #Function to return list containing vertices in Topological order.
    def topoSort(self, V, adj):
        stack=[]
        ans=[]
        visited=[-1]*V
        def dfs(node):
            visited[node]=1
            for i in adj[node]:
                if visited[i]!=1:
                    dfs(i)
            stack.append(node)
        for i in range(V):
            if visited[i]!=1:
                dfs(i)
        while(len(stack)!=0):
            ans.append(stack.pop())
        return ans
```

Implementing Dijkstra Algorithm

```

import heapq
class Solution:

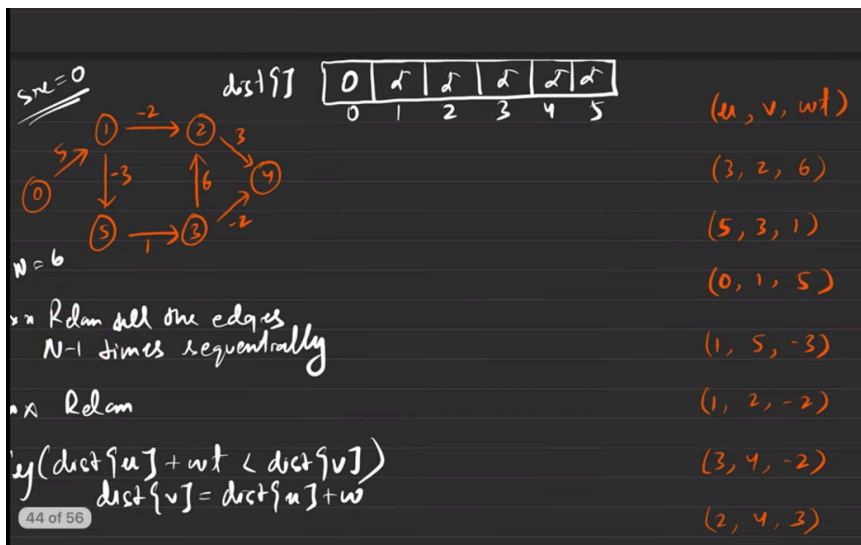
    #Function to find the shortest distance of all the vertices
    #from the source vertex S.
    def dijkstra(self, V, adj, S):
        dist=[float('inf')]*V
        pq=[]
        dist[S]=0
        heapq.heappush(pq,(0,S))
        while pq:
            d,node= heapq.heappop(pq)
            for i in adj[node]:
                adjnode=i[0]
                edgewt=i[1]
                if d+edgewt<dist[adjnode]:
                    dist[adjnode]= d+edgewt
                    heapq.heappush(pq,(dist[adjnode],adjnode))
        return dist

```

Q: BELLMAN-FORD(DISTANCE FROM SOURCE):

- Works for negative edge wt .
- Detect negative cycle in graph

Runs for $V-1$ time to find the distance from source to node and to check if cycle exists or not then run for one more time. If it still relaxes then cycle exists otherwise not.



Intuition → why $N-1$??

xx edges can be in any order...

src $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

(u, v, wt)

1 2 3 4

✓ ✓ ✓ ✓

1 2 3 4

Since in a graph of N nodes, in worst case, you will take N-1 edges to reach from the first to the last, thereby we iterate for N-1 iterations.

Try drawing a graph which takes more than N-1 edges for any path, it is not possible.

$dist[3] + 1 < dist[4]$ (3, 4, 1)

$dist[2] + 1 < dist[3]$ (2, 3, 1)

$dist[1] + 1 < dist[2]$ (1, 2, 1)

$dist[0] + 1 < dist[1]$ (0, 1, 1)

u/v

```
def bellman_ford(self, V, edges, S):
    dist=[100000000]*V
    dist[S]=0
    iteration=V-1
    while(iteration):
        for i in edges:
            sor=i[0]
            des=i[1]
            wt=i[2]
            d = dist[sor]+wt
            if d < dist[des]:
                dist[des]= d
        iteration-=1

    #to check if negative cycle
    #if path still update that mean there is negative cycle exists
    for i in edges:
        sor=i[0]
        des=i[1]
        wt=i[2]
        d = dist[sor]+wt
        if d < dist[des]:
            return [-1]
    return dist
```

Time Complexity: $O(V \cdot E)$, where V = no. of vertices and E = no. of Edges.

Space Complexity: $O(V)$ for the distance array which stores the minimized distances.