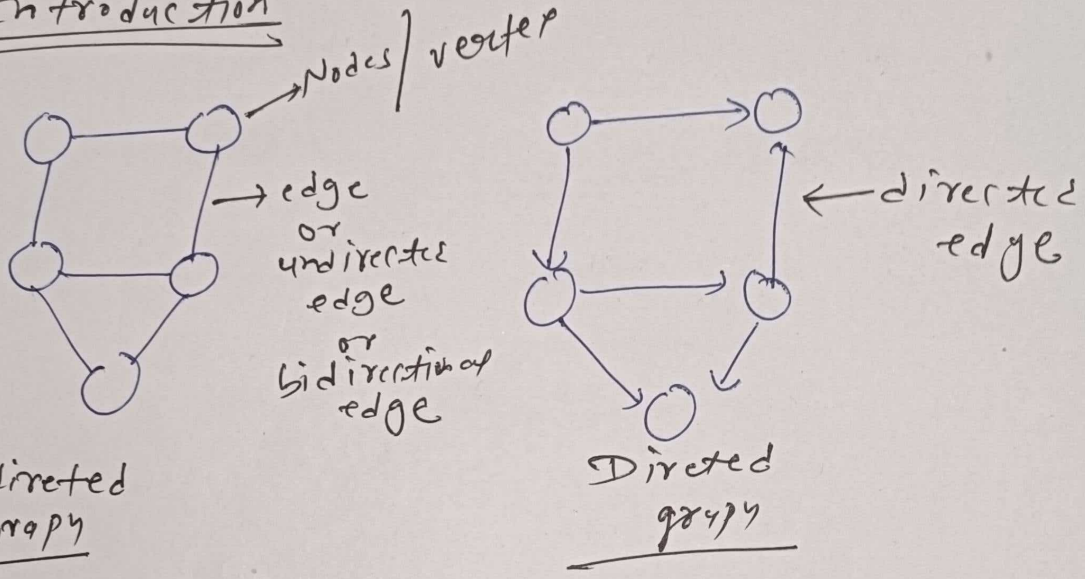
Graph

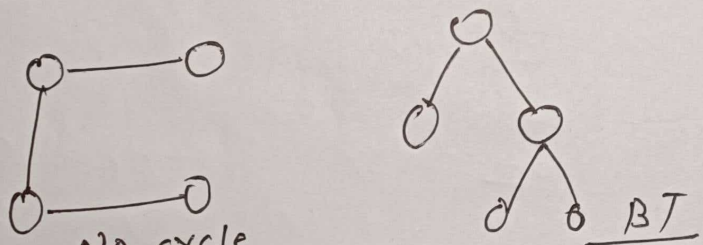
Introduction



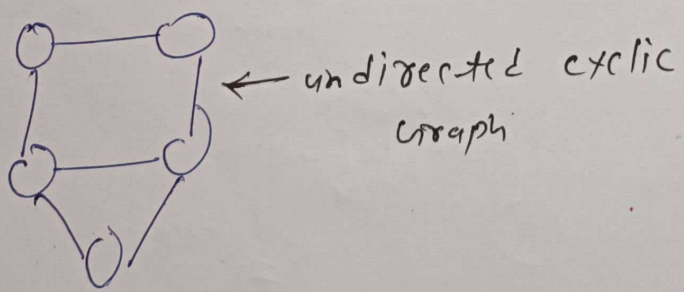
Undirected graph

Directed graph

Cycles in a Graph

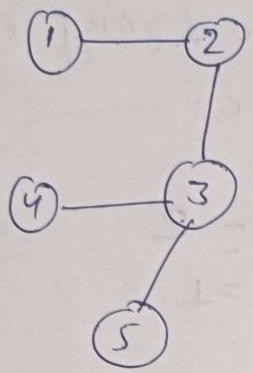


Start from a node & end at that node



Path

contain a lot of nodes/vertices & each of them are reachebale



path → 1 2 3 5 ✓

1 2 3 2 1 ✗

Imp:- A node can't appear twice in a path.

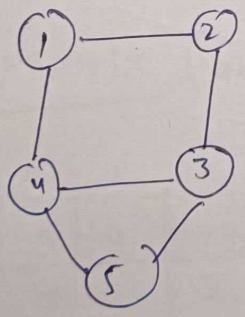
1 3 5 ✗

adjacent node in path must have an edge b/w them.

ex- 1 2 3 5

1 2 → edge ✓
2 3 → edge ✓
3 5 → edge ✓

Degrees in Graph:-



D (3) = 3

D (4) = 2

for undirected graph the degree of node is equal to numbers of edges attached to it.

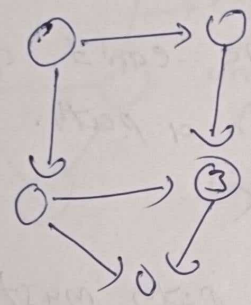
Imp total degree of graph = 2×E

↳ edges.

Directed Graph

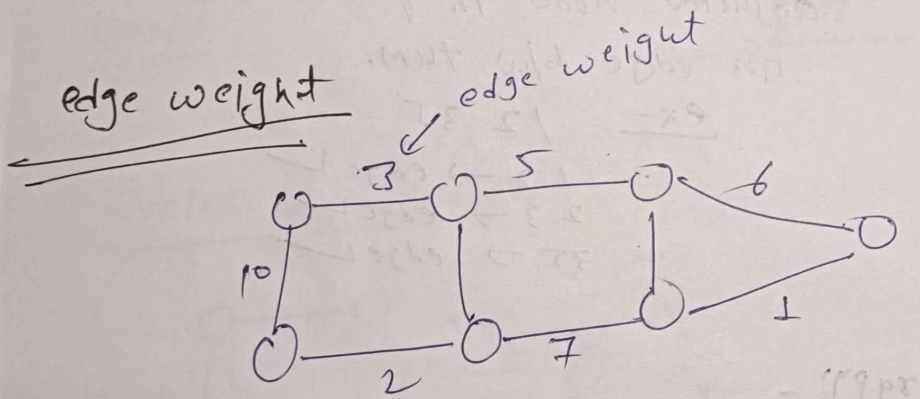
(i) Indegree → The number of incoming edges to that node

(ii) outdegree → The no. of outgoing edges to that node.

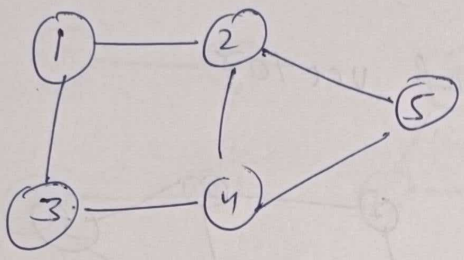


Indegree (3) = 2
outdegree (3) = 1

edge weight



Graph Representation



i/p

```
5 6
2 1
1 3
2 4
3 5
2 5
4 5
```

Input

n = 5 → no. of nodes/vertices

m = 6 → no. of edges in graph.

m lines → represent edges

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

Store

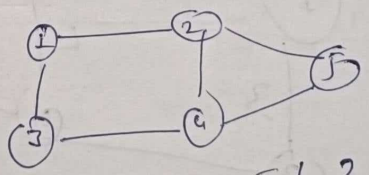
There are two ways to store a graph

(i) Matrix way

(ii) List way

(i) Matrix (adjacency matrix)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	0	0
2	0	1	0	0	1	0
3	0	1	0	0	1	1
4	0	0	1	1	0	1
5	0	0	0	1	1	0



edges

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

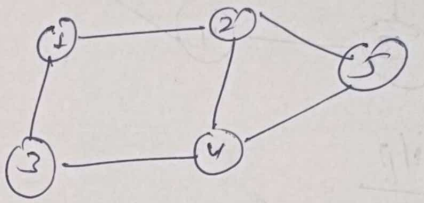
Space - O(N*N)

(ii) Adj List

vector<int> adj[n+1];

↳ array of vector.

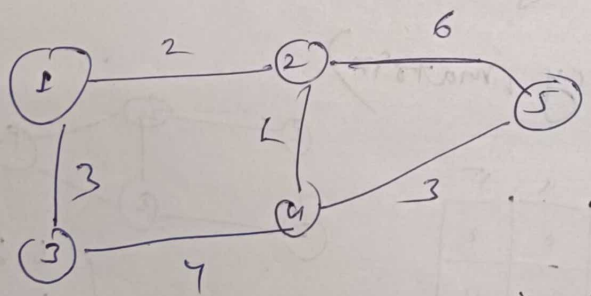
0 → { }
1 → {2, 3}
2 → {1, 4, 5}
3 → {1, 4}
4 → {3, 2, 5}
5 → {2, 4}



S.C. = 2*E

Directed graph → S.C. = E

weighted graph

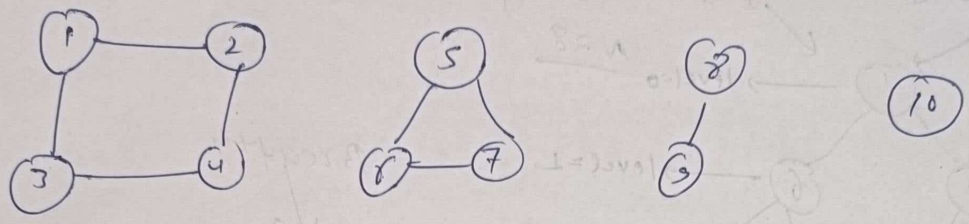


vector<pair<int, int>> adj[n+1];

4 → {(2,1), (3,4), (5,3)}

adj node, weight.

Connected Components



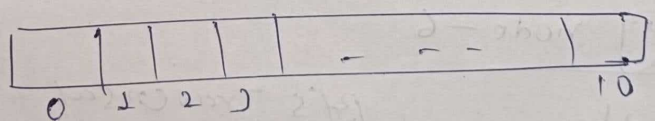
4 different components of a single graph

n = 10 m = 8

edges

1 2
1 3
2 4
3 4
5 6
5 7
6 7
8 9

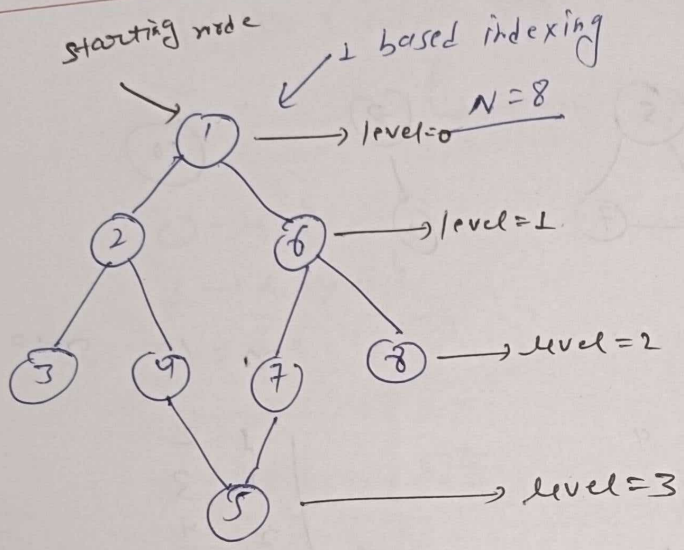
So when we do traversal of graph then we ~~always~~ always used a visited array.

vis = 

N = 11

```
for ( i = 1 ⟶ 10)
{
    if ( ! vis[i]
    {
        traveral (i);
    }
}
```

Breadth - First - Search Traversal



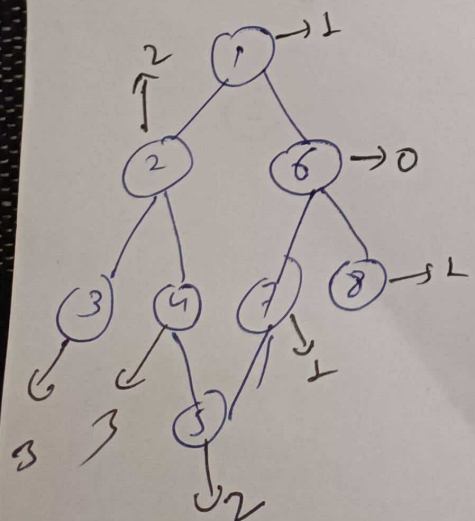
Breadth → level wise traversal

BFS Traversal → 1 2 6 3 4 7 8 5

0 1 2 3

Imp:- There can be multiple BFS traversal for a particular starting Node. Also if starting node changes BFS Traversal algo changes.

Starting node - 6

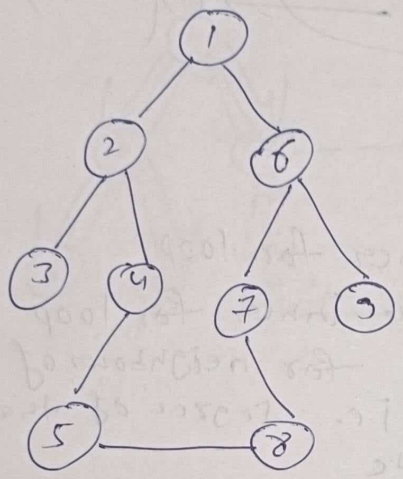


BFS Traversal

6 1 7 8 2 5 3 4

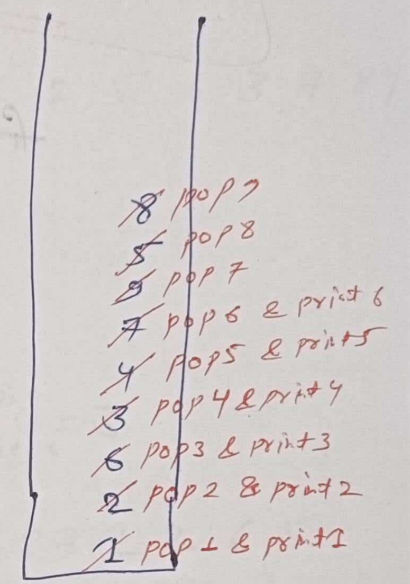
0 1 2 3

BFS Traversal

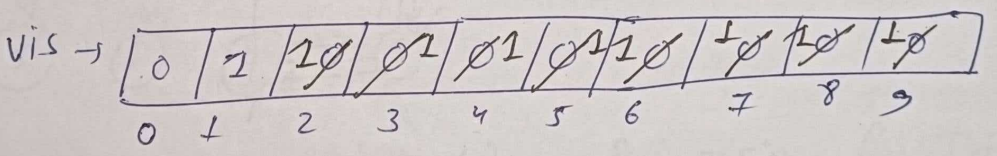


adj -

1 → {2,6}
2 → {1,3,4}
3 → {2}
4 → {2,5}
5 → {4,8}
6 → {1,7,9}
7 → {6,8}
8 → {5,6}
9 → {6}



BFS → 1 2 6 3 4 7 9 5 8



Imp:- initially queue can't contain only starting node = 1 & when any node is visited or is inserted into queue marked as 1 in vis.

1st iteration of queue → node = q.front() → print(1)
q.pop();
adj[1] = {2,6}
insert (2,6) into queue.

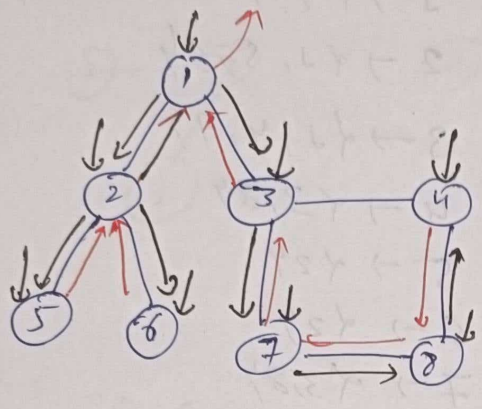
S.C. = O(3N)

T.C. = O(N) + O(2E) → O(N+E)

for queue

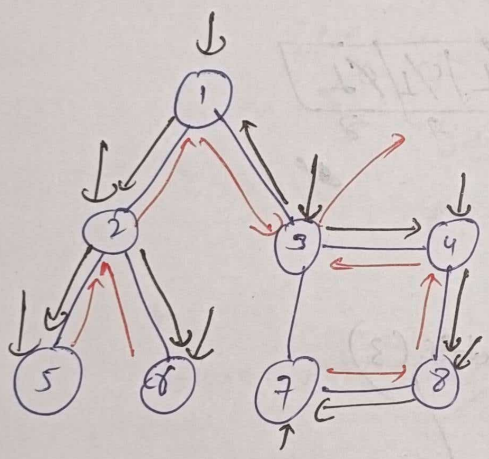
for inner for loop
bcz the inner for loop will run for neighbours of a node i.e. degree of that node
i.e. for all degree of the graph

DFS Traversal in a Graph



Starting node = 1

DFS = 1 2 5 6 3 7 8 4



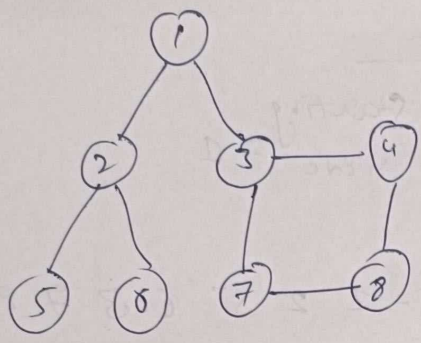
Starting node = 3

DFS = 3 4 8 7 1 2 6 5

⟹ Use Recursion for DFS.

```
for(int i=0; i<V; i++)
{
    if(!vis[i])
    {  dfs(i)
    }
}
```

```
dfs(node)
{
    vis[node] = 1
    dfs.push_back(node);
    for(auto nb: adj[node])
    {
        if(!vis[nb])
        {  dfs(nb);
        }
    }
}
```

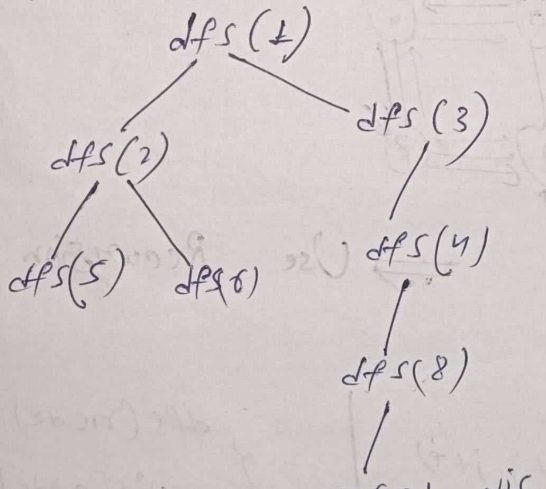


adj list

1 → {2, 3}
2 → {1, 5, 6}
3 → {1, 4, 7}
4 → {3, 8}
5 → {2}
6 → {2}
7 → {3, 8}
8 → {4, 7}

vis → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

dfs = 1 2 5 6 3 4 8 7

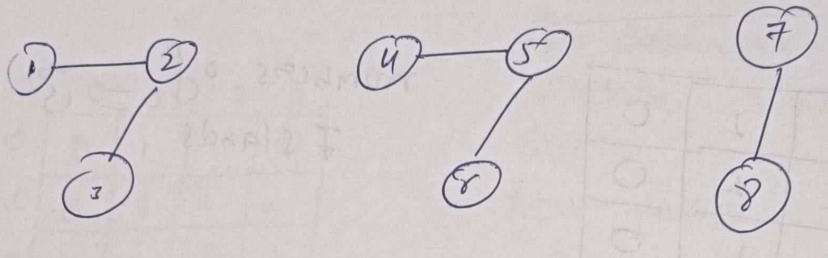


dfs
vis
recursion

S.C. = O(N) + O(N) + O(N)

T.C. = O(N + 2E)

Number of Provinces




```
for(i=0 ; i<=V; i++)
{
    if( !vis[i])
    {
        dfs(i)
        provinces++;
    }
}
return provinces;
}
```

S.C. = O(N) + O(N)

~~T.C. = O(N) + O(2E)~~

T.C. = O(N) + O(V+2E)

for loop

DFS call

200. Number of Islands

ex

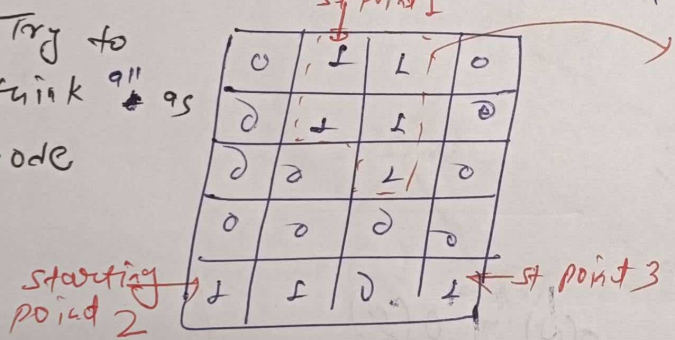
0	1	1	0
0	1	1	0
0	0	1	0
0	0	0	0
1	1	0	1

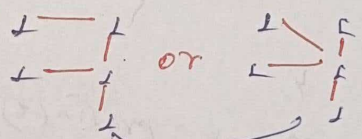
numbers of Islands = 3

Intution :-

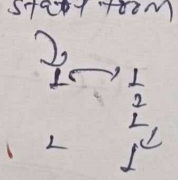
How can we say that it is a graph problem bcz it is a 2D matrix.

Try to think all as node





many way these can be connected to each other

So if I start a traversal  then it will make sure it will traverse the nearest 1 in all 8 direction so from 3 starting point we can traverse all the island.

→ There by there are 3 starting point so 3 island

BFS Traversal

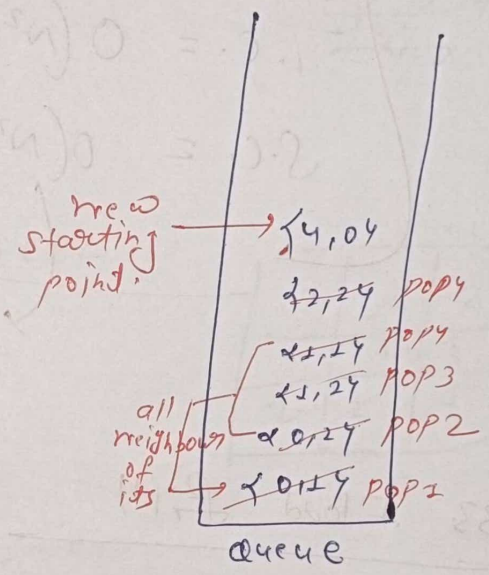
	0	1	2	3
0	0	1	1	0
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	1	1	1	1

for BFS we need a starting node into our queue.

here take row col.
starting → {0, 1}

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

vis



- Initially insert the starting node into queue & mark them as visited.

1-st iteration

BFS said that while traversal pickup the element from queue & insert all its neighbour into queue.

node = {0,1}

neighbour → {0,1}

Q How to decide a starting point?

⇒ Traverse the matrix & if you find a land that is not visited then this is your starting point.

T.C. = $O(N^2)$

S.C. = $O(N^2) + O(N^2)$

queue

vis

733. Flood fill

Similar is above problem.

imp in the question they don't want you to do color in minimum time.

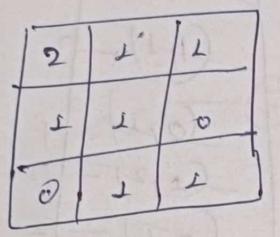
T.C. = $O(n \times m)$

S.C. = $O(n \times m)$ + stack space

$\sim O(n \times m)$

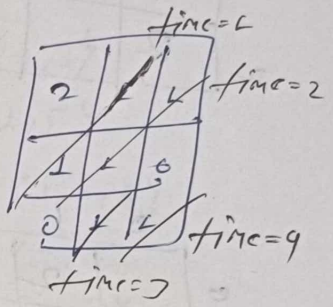
~~733. Flood Fill~~

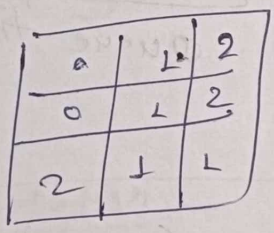
994. Rotting Oranges



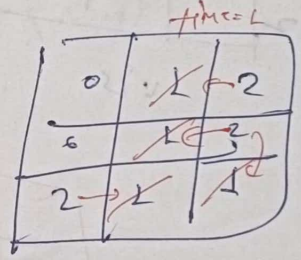
time = 4

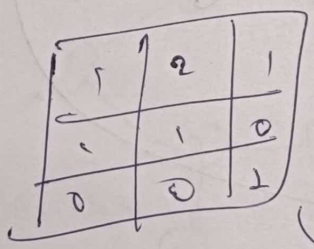
bcz



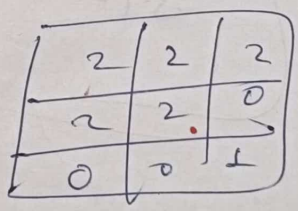


time = 2

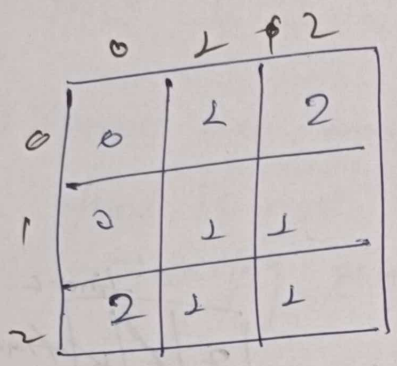


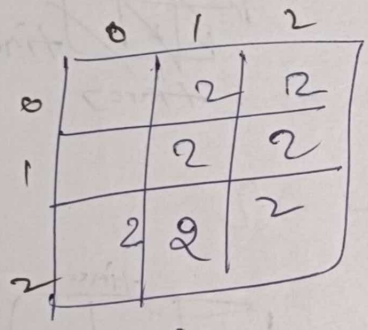


time = 1

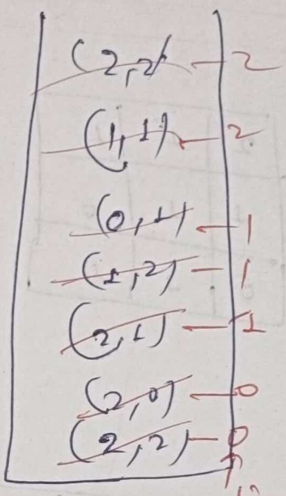


To solve this problem we use BFS Algorithm & their can be many starting points.





vis



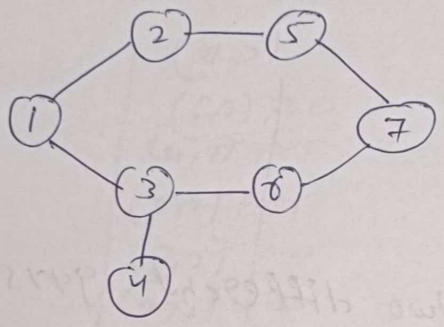
Queue time

$T.C. = O(n \times m) + O(n \times m \times 4)$

$S.C. = O(n \times m)$

Detect Cycle in a Undirected Graph

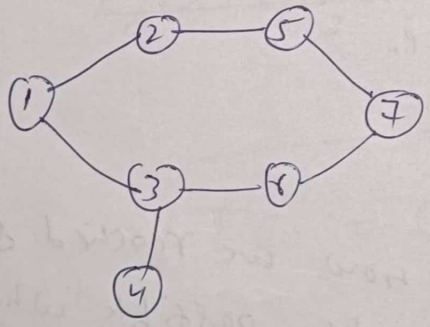
BFS



adj-List

1 → {2,3}
2 → {1,5}
3 → {1,4,6}
4 → {3}
5 → {2,7}
6 → {3,7}
7 → {5,6}

Intution :-



BFS is a Traversal technique now start from '1' & go level wise.

from 1 we can go → 2 & 3 at same time & remember that you came from 1

Now from 2 you can go → 5 & 1 but you not go to 1 bez you came frome 1. so go at 5.

from 3 ⟶ 6, 4, 1 but not go to 1 bcz you came from 1

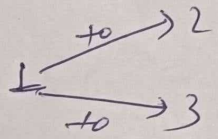
3 ⟶ 6 & 4
&
2 ⟶ 5

from 5 ⟶ 7 & 2^x

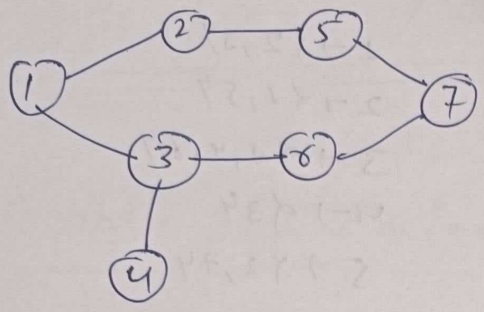
from 6 ⟶ 7 & 3^x

↳ so from two different guys 5 & 6 we can go to 7 i.e. at same place.

Imp from starting we are going in different direction i.e.

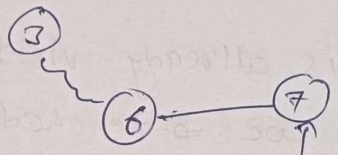


but some how we reached same place it is only possible when the graph is connected.



1 → { 2, 3, 4 }
2 → { 1, 5 }
3 → { 1, 2, 4, 6 }
4 → { 3 }
5 → { 2, 7 }
6 → { 3, 7 }
7 → { 5, 6 }

(7, 5)
(5, 2) pop5
(4, 3) pop4
(2, 1) pop2
(3, 1) pop3
(1, -1) pop1



already visited
i.e. cycle.

vis → | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
0 1 2 3 4 5 6 7

BFS

for loop of BFS

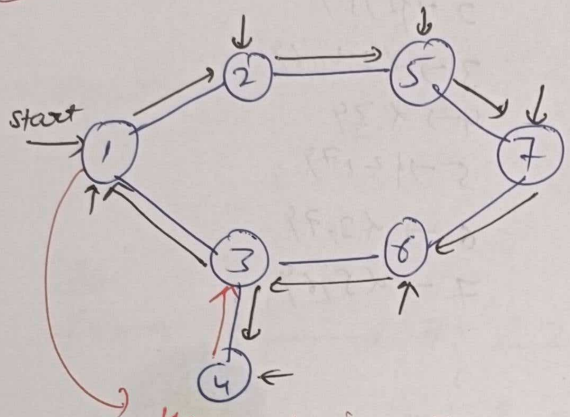
$$T.C. = O(N+2E) + O(N)$$

$$S.C. = O(N) + O(N)$$

Queue

vis

DFS



1 → {2, 3}
2 → {1, 5}
3 → {1, 4, 6}
4 → {3}
5 → {2, 7}
6 → {3, 7}
7 → {5, 6}

Here 1 is already visited that means we reached at a point where we started it is only possible in cycle.

vis = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

node parent

DFS(1, -1)

DFS(2, -1)

DFS(5, 2)

DFS(7, 5)

DFS(6, 7)

DFS(3, 6) → DFS(1, 3)

DFS(4, 3)

Dfs

T.C. = O(N + 2E) + O(N)
(loop)

S.C. = O(N) + O(N)
(vis) (stack)

Distance of nearest cell having 1

Number of Enclaves

0 → sea cell
1 → land cell

n x m ← matrix

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

→ ans = 4

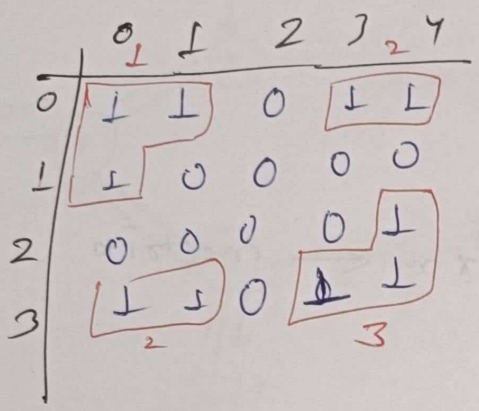
Return the number of land cell in grid for which we can't walk off the boundary of the grid in any number of moves.

⟹ Call dfs/bfs for boundary cell that having '1's and mark visited that all cell which are connected to these.

T.C. = O(n*m*4)

S.C. = O(n*m) + O(max(n,m))

Number of Distinct Islands



no. of islands = 4

distinct islands = 3

Intution :- To solve this problem one thing which we can do is we can store the shap of Island into a set data structure bcz set always store the unique valuse.

⇒ How do we can store the shap?

Shap-1 : (0,0) (0,1) (1,0) → {(0,0), (0,1), (1,0)}

These are not same.

Shap-2 : (2,3) (2,4) (3,3) → {(2,3), (2,4), (3,3)}

Base → (2,3) so (2,3) - (2,3) = (0,0)
(2,4) - (2,3) = (0,1)
(3,3) - (2,3) = (1,0)

Now the st are same. → {(0,0), (0,1), (1,0)}

(0,3) (0,4) ⟶ (0,3) − (0,3) = (0,0)

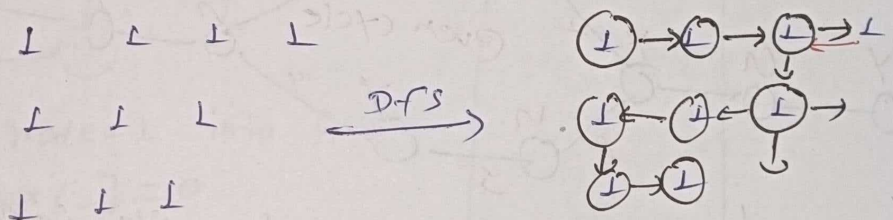
Shape → 3 ⊥ ⊥ (0,4) − (0,3) = (0,1)

Shape 4 ⊥ ⊥ ⟶ (3,0) − (3,0) = (0,0)

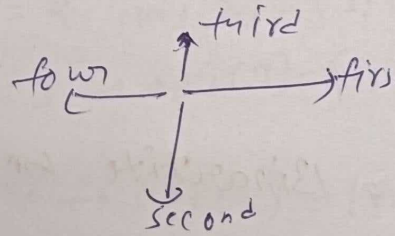
(3,0) (3,1) (3,1) − (3,0) = (0,1)

Same Shape

Note:-

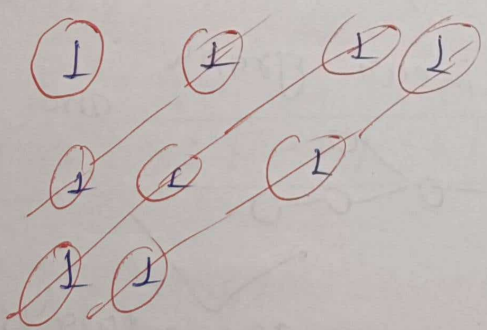


follow a particular order for entire BFS / DFS traversal so that list contains values in an order for every Island.

i.e. 

← this order should be follow for everyone.

BFS



DFS

T.C = O(n * m * 4)

loop → O(n * m)

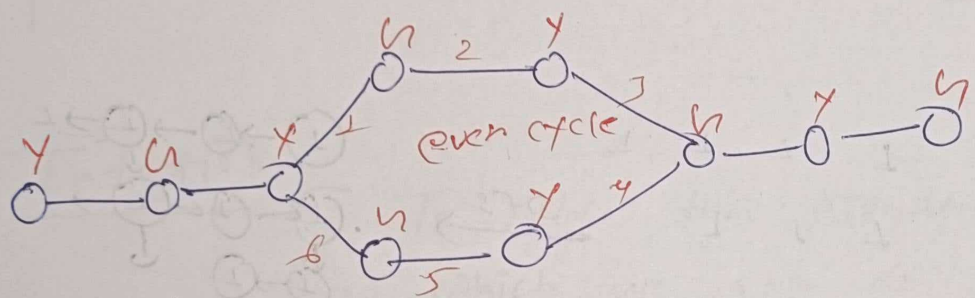
S.C = O(n * m)

set of vis

Bipartite Graph

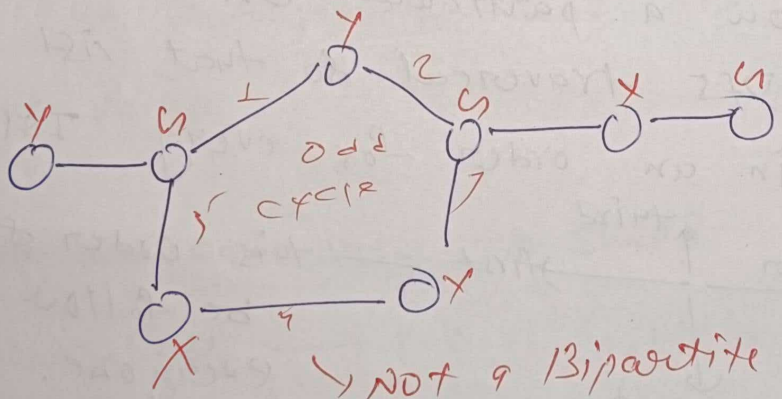
Bipartite Graph :- Colour the graph with 2 colour such that no adjacent nodes have same color.

ex color → Y/G



↳ This is a Bipartite Graph.

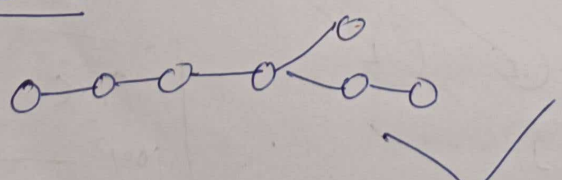
ex



↳ Not a Bipartite Graph

Note :-

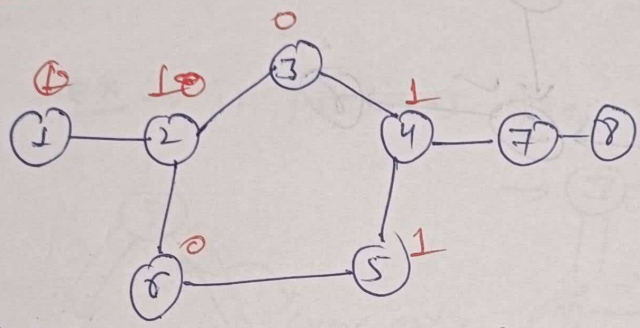
case-1 → Linear graph and graph having even length cycle



are always the Bipartite graph

⇒ The graph which have odd length cycle are not Bipartite graph.

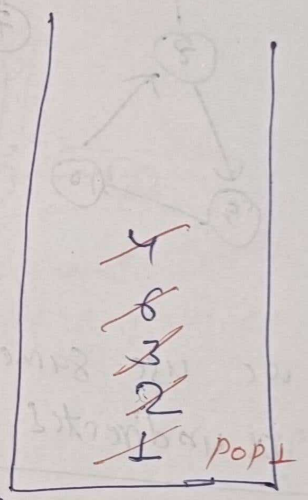
BFS



initially put node = 1 into queue
& assign color[1] = 0

→ Now itrate on queue

node = q.front() = 1
adj(1) = 2
color[2] = -1
color[2] = !color(1)



queue

→ node = 2 adj[2] = {3, 6} {3 = -1, 6 = -1} ⇒ 3 = !color[2] = 0, 6 = !color[2] = 0

→ node = 3 → adj[3] = {2, 4} → 2 → 1 & 3 → 0 ✓
4 → -1 & 3 → 0 ⇒ 4 = 0

→ node = 6 → adj[6] = {2, 5} → 2 → 0 & 6 → 0 ✓
5 → -1 & 6 → 0 → 5 = 1

→ node = 4 → adj(4) = {5, 7} → 5 → 1 & 4 → 1 → Not a Bipartite.

	1	2	3	4	5	6	7	8
color →	0	-1	-1	-1	-1	-1	-1	-1

-1 ← not colored yet

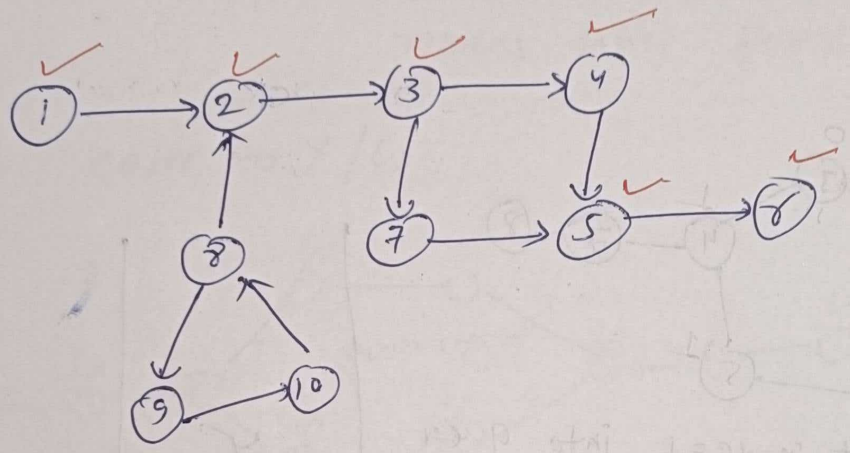
color = 0 / 1

T.C. = O(n) + O(n+2E)

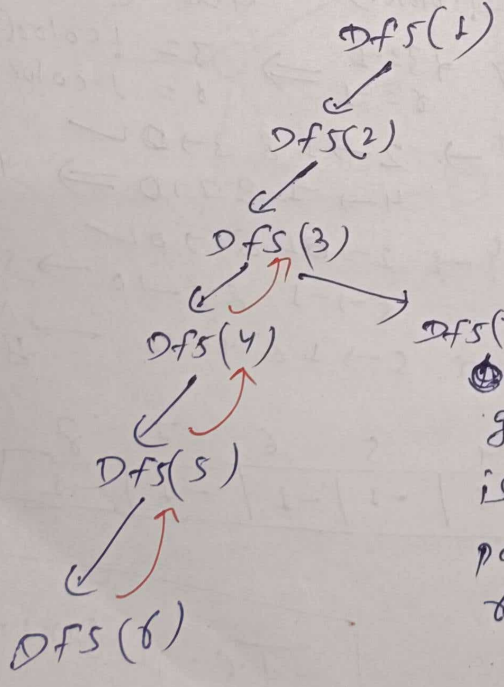
S.C. = O(n) + O(n)

Detect a Cycle in a Directed Graph

use Dfs



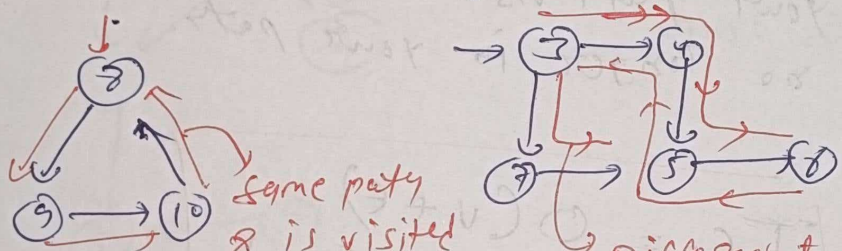
Can we use same approch as detect cycle in a undirected graph → No.

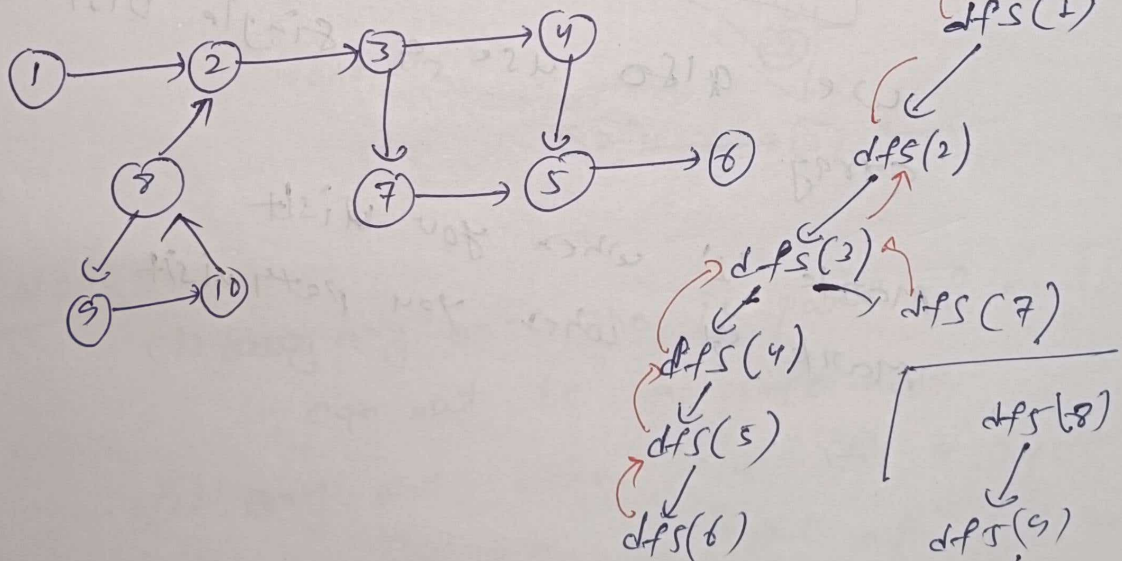


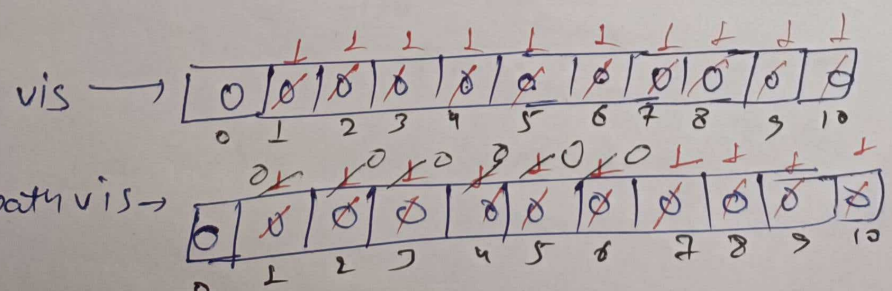
↳ now at 7 when we go for adjacent then 5 is visited && not the parent of 7 so this will return a cycle as the above approz

Cycle in Directed graph ⟶ If on the same path we found a node that is previously visited then we can say that graph contain a cycle.

ex → in previous graph.







→ when you call dfs then mark visited that vertex & also path visited of that vertex.

And

when you come back from that node then you have to remove that node from your path visited vector bcz it is no longer in your path.

T.C. = O(V+E)

S.C. = O(2V)

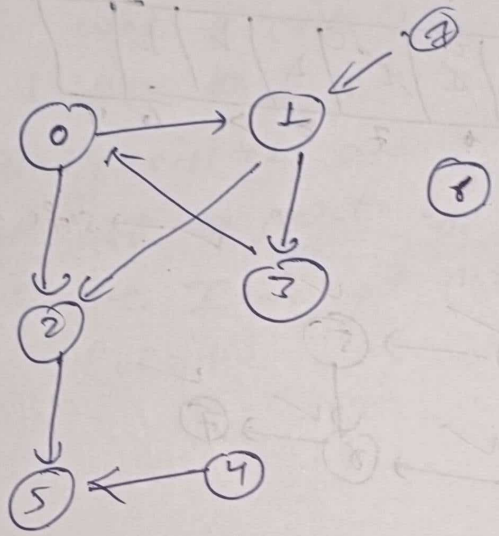
we also use a single visited array

Mark '1' when you visit

Mark '2' when you path visit

Eventual Safe path

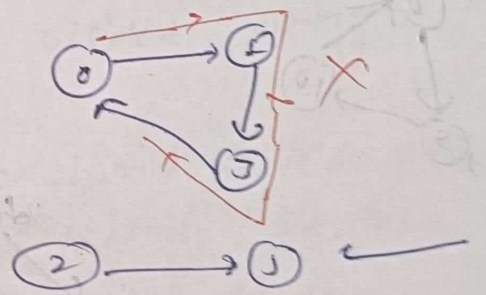
Safe node → every path from that node ends up at terminal node



ans = 2, 4, 5, 6

check for node = 0

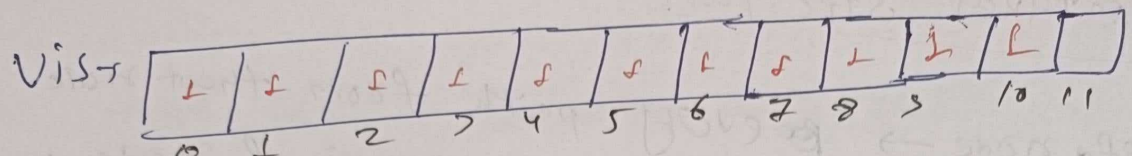
node = 2

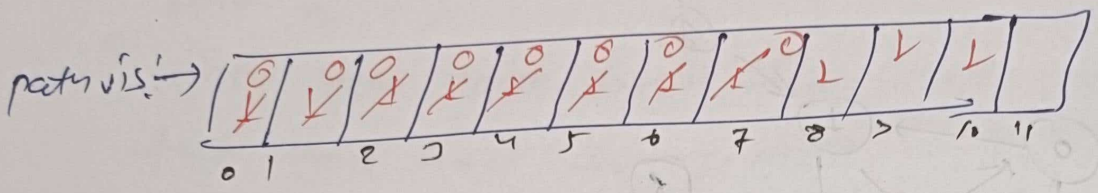


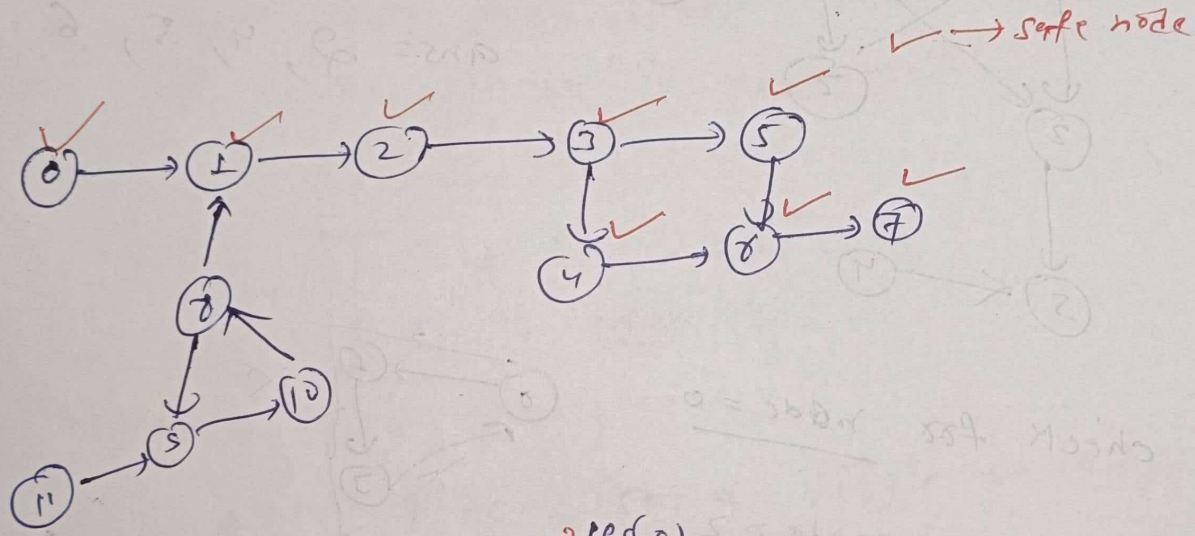
Note -

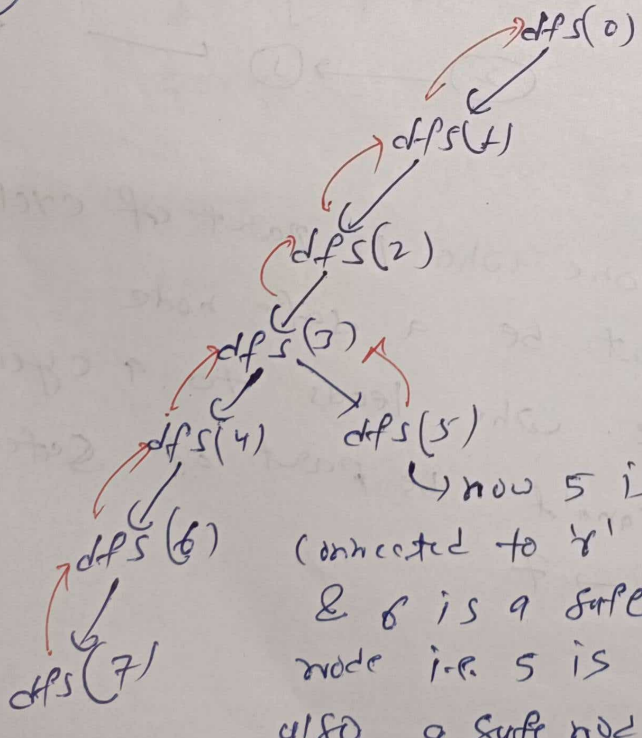
(i) any one who is part of cycle can not be a safe node

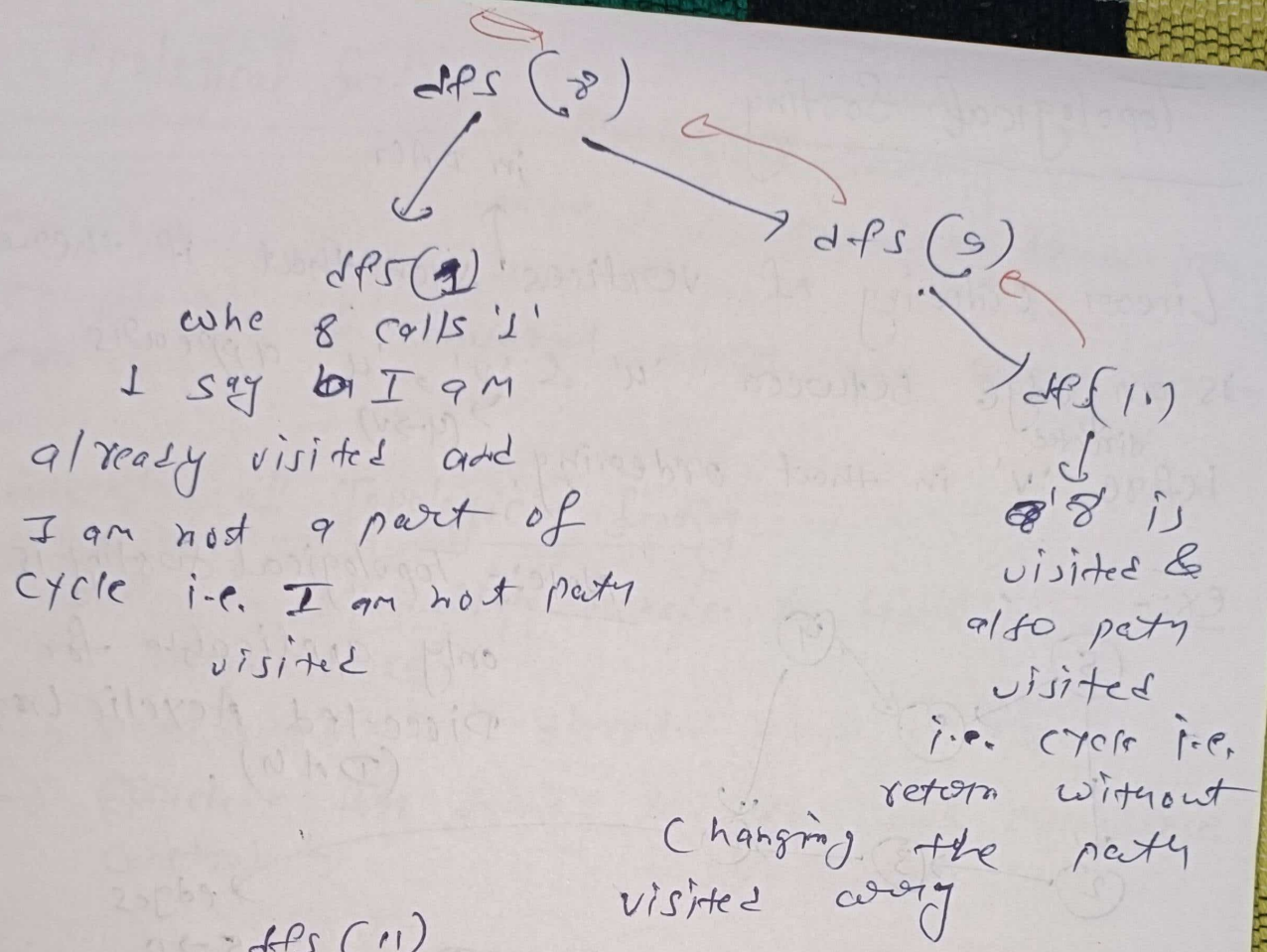
(ii) any one who leads to a cycle also can't be part of safe node like → 7


vis →
0 1 2 3 4 5 6 7 8 9 10 11


path vis →
0 1 2 3 4 5 6 7 8 9 10 11


→ safe node


dfs(0)
dfs(1)
dfs(2)
dfs(3)
dfs(4)
dfs(5)
dfs(6)
dfs(7)
↳ now 5 is connected to '8' & 8 is a safe node i.e. 5 is also a safe node



dfs(1) when 8 calls '1' I say I am already visited and I am not a part of cycle i.e. I am not path visited

dfs(11) 8 is visited & also path visited i.e. cycle i.e. return without changing the path visited array

dfs(7) ← 8 is visitid & also path visited i.e. 11 is not a safe node.