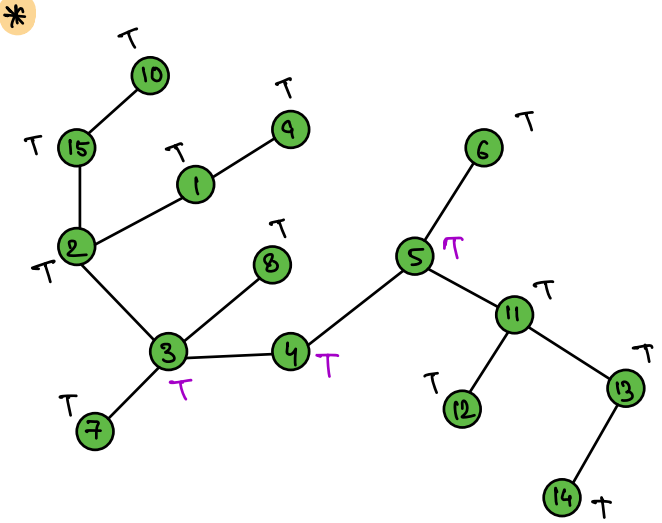
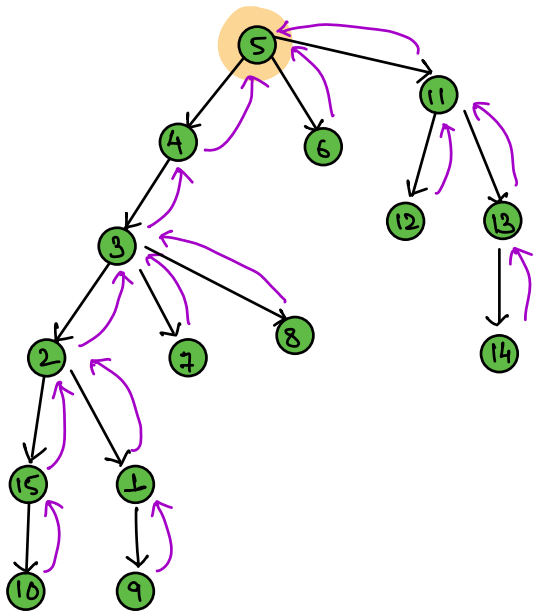
*



S = 5 , D = 14 ⇒ True.



⇒ DFS
(Depth First Search)

```
bool isPath(N, E, u[], v[], src, dest) {
    list<int> g[N+1]; // TODO. (Adj. List)
    bool vis[N+1] = {false};
    dfs(g, vis, src);
    return vis[dest];
}
void dfs(list<int> g, bool vis[], int s) {
    if(vis[s] == true) return;
    vis[s] = true;
    for(i = 0; i < g[s].size(); i++) {
        v = g[s][i];
        dfs(g, vis, v);
    }
}
```

TC : O(E)

SC : O(E + N + recursive stack)

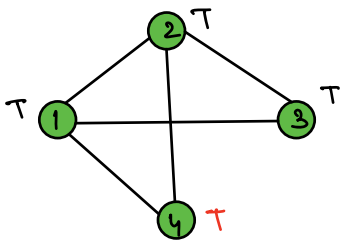
E → Adj list

N → Vis Array

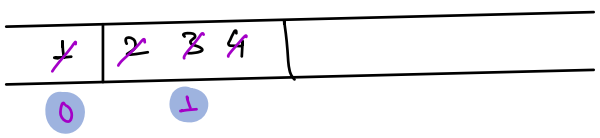
recursive stack → N

: O(N+E) ⇒ O(E) {E >> N}

Eg

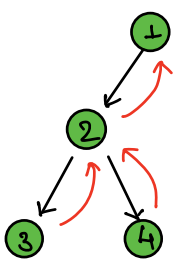


* BFS



→ from 1 shortest distance to reach 2, 3, 4 ⇒ (1)

* DFS



length of path from 1 to 4 = 2

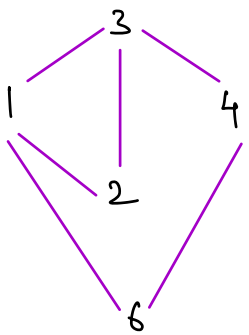
Obs :- DFS won't give us the shortest path from src.

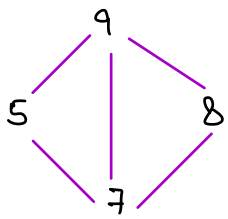
Only visit ⇒ DFS

Shortest path ⇒ BFS

Q. Given an undirected graph, find the number of connected components.

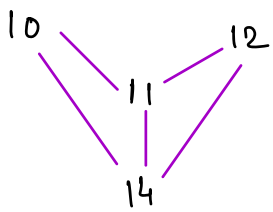
connected component ⇒ If from every node, we can visit all the nodes of a component.





No. of connected components :-

⇒ 4



Src

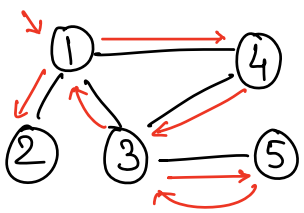
bool vis[N+1]

0	
1	→ 3, 2, 6
2	→ 1, 3
3	→ 1, 2, 4
4	→ 3, 6
5	→ 7, 9
6	→ 1, 4
7	→ 5, 9, 8
8	→ 7, 9
9	→ 5, 7, 8
10	→ 11, 14
11	→ 10, 12, 14
12	→ 11, 14
13	→ 15
14	→ 10, 11, 12
15	→ 13

0	
1	~~F~~ T ⇒ Apply DFS
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T ⇒ Apply DFS
6	~~F~~ T
7	~~F~~ T
8	~~F~~ T
9	~~F~~ T
10	~~F~~ T ⇒ Apply DFS
11	~~F~~ T
12	~~F~~ T
13	~~F~~ T ⇒ Apply DFS
14	~~F~~ T
15	~~F~~ T

No. of connected components = No. of DFS calls.

Eg



1 connected component.

```
int components(N, E, u[], v[]) {
    list<int> g[N+1]; // Create
    bool vis[N+1];
    ans = 0
    for(i = 1; i <= N; i++) {
        if (!vis[i]) {
            dfs(g, vis, i);
            ans++;
        }
    }
    return ans;
}
```

```
void dfs(list<int> g, bool vis[], int s) {
    if (vis[s] == true) return;
    vis[s] = true;
    for(i = 0; i < g[s].size(); i++) {
        v = g[s][i];
        dfs(g, vis, v);
    }
}
```

TC :- O(E), SC: O(N+E)

↳ O(E)

if E >> N

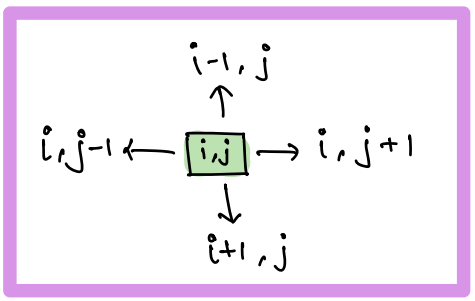
Q: Number of Islands.

→ Given a matrix of 1's & 0's. find the no. of islands present.

mat[ ][ ] → 0 Water

mat[ ][ ] → 1 land

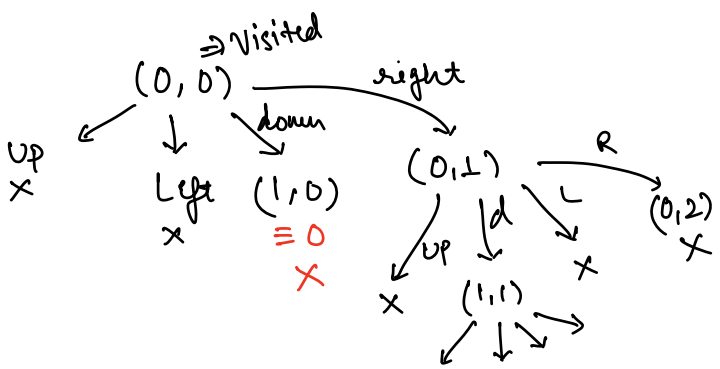
	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1



Islands = 5

	0	1	2	3	4
0	1 → ~~0~~	~~0~~	0	0	~~0~~
1	0	~~0~~	0	~~0~~	0
2	~~0~~	0	0	~~0~~	~~0~~
3	~~0~~	~~0~~	0	0	0
4	~~0~~	0	~~0~~ → ~~0~~ → ~~0~~		

no. of islands = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5



```
int islands( int mat[][], N, M){
    C=0
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            if(mat[i][j]==1){
                // with i,j call DFS.
                C++;
                dfs(mat, i,j, N,M);
            }
        }
    }
    return c;
}
```

T B R L

```
int x[] = {-1, 1, 0, 0}
int y[] = {0, 0, 1, -1}
```

```
void dfs( mat[][], i, j, N, M){
    if( i<0 || j<0 || i>=N || j>=M || mat[i][j]==0){
        return;
    }
    mat[i][j] = 0
    dfs(mat, i+1, j, N, M);
    dfs(mat, i-1, j, N, M);
    dfs(mat, i, j+1, N, M);
    dfs(mat, i, j-1, N, M);
}
```

(i,j) is Invalid index.

Same

```
for(k=0; k<4; k++){
    dfs(mat, i+x[k], j+y[k], N, M)
}
```

TC: $O(N \cdot M)$ SC: (1 + Rec. stack)

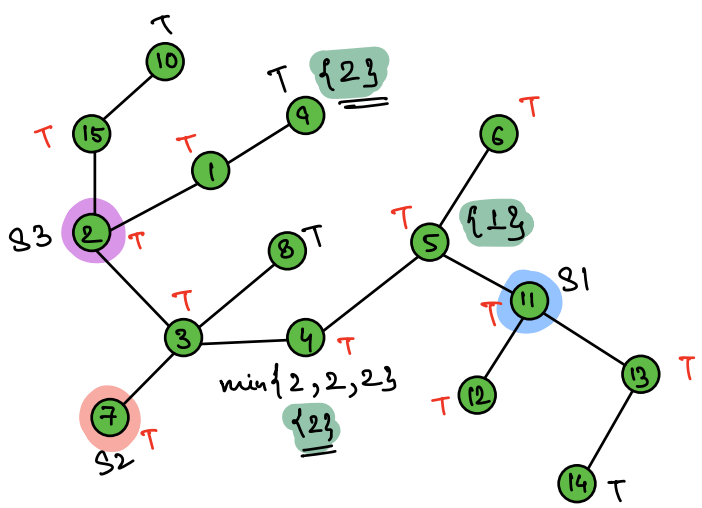
⇓

$O(N \cdot M)$ {wc}

SC: $O(NM)$

Multisource BFS

→ Given N nodes & multiple sources $\{s_1, s_2, s_3\}$. find the length of shortest path for all the nodes to any of the source nodes.



S ⇒ 11, 7, 2

D = 5

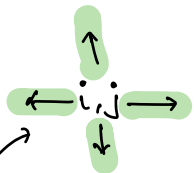
~~11~~, ~~7~~, 2	~~5~~, ~~12~~, ~~15~~, ~~3~~, ~~1~~, ~~15~~	~~4~~, ~~6~~, ~~14~~, ~~8~~, ~~9~~, ~~10~~
0	level = 1	level = 2

Idea :- Only change is to push all the given source nodes in the Queue & apply BFS.

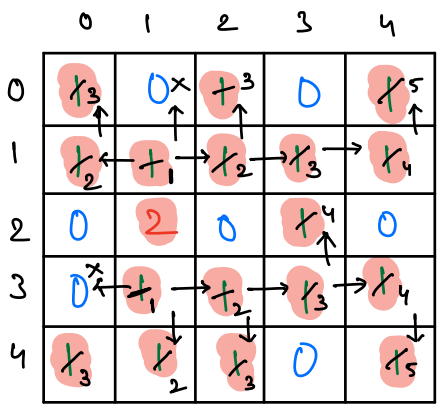
⇒ TODO

Q. Rotten Oranges

mat[N][M] → 0 Empty
→ 1 Orange
→ 2 Rotten Orange

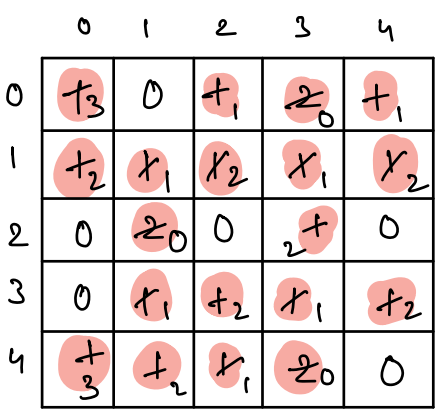


Every minute any fresh orange, adjacent to a rotten orange becomes rotten.
Find min. time when all the oranges will become rotten.



⇒ ans = 5

Idea ⇒ BFS.



→ Multisource BFS

⇒ ans = max(mat[][])
= 3

level 0: (0,3) (2,1) (4,3)

level 1: (0,2) (0,4) (1,3) (1,1) (3,1) (4,2) (3,3)

level 2: (1,2) (1,4) (2,3) (1,0) (4,1) (3,2) (3,4)

```
int minTime(mat[][], N, M){
    Queue<pair<int, int>> q;
    int time[N][M] = {-1};
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            if(mat[i][j] == 2){
                q.enqueue({i,j});
                time[i][j] = 0
            }
        }
    }
    int x[] = {-1, 1, 0, 0}
    int y[] = {0, 0, 1, -1}
    while(q.size() > 0){
        pair<int, int> d = q.front();
        q.dequeue();
        i = d.first;
        j = d.second;
        for(k=0; k<4; k++){
            a = i + x[k], b = j + y[k];
            if(a>=0 && a<N && b>=0 && b<M &&
               mat[a][b] == 1){
                mat[a][b] = 2
                q.enqueue({a,b});
                time[a][b] = time[i][j] + 1;
            }
        }
    }
}
```

if this is True, (a,b) is a valid index

⇒ // Check mat[][], if even a single (1) is // present, return -1;

⇒ // Iterate over time[][], & find max elem.

}

TC: O(N·M)

SC: O(N·M)

———— * ————

Doubts

N×M

⇒ N-1, M-1

i, j

Invalid i<0 || j<0 || i>=N || j>=M

Valid i>=0 && j>=0 && i<N && j<M