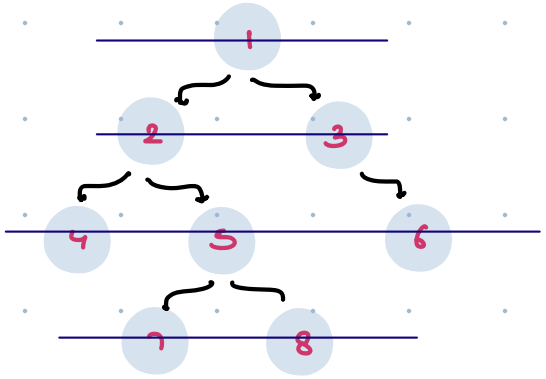
Graphs 2

Good Evening

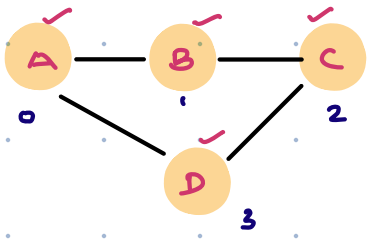
Agenda

- → BFS (Breadth first search)
- → Multisource BFS ✓
- → Rotten Oranges ✓
- → No. of Island ✓
- → Shortest Distance in Maze (Google)

BFS (Breadth first Search)



Level Order Traversal



~~A~~ ~~B~~ ~~D~~ ~~C~~

output : A B D C

Steps

01. Take a queue

→ Add src node to the queue (Mark src node as true)

```
while (q.isEmpty() == false) {
    → Remove nodes from queue → process it
    → Add all the unvisited nbrs inside queue
}
```

mark them visited

* Given graph { Adjacency list }

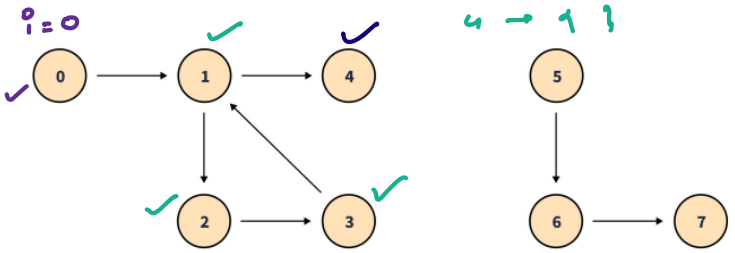
```
Queue<Integer> q = new LinkedList<>();
boolean [] vis = new boolean[n];
q.add(src);
vis[src] = true;
```

```
while (q.size() > 0) {
    int rem = q.remove();
    print(rem);
    for (int nbr : graph[rem]) {
        if (vis[nbr] == false) {
            vis[nbr] = true;
            q.add(nbr);
        }
    }
}
```

TC : O(n+E)

SC : O(n)

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




```
for (i=0; i<n; i++) {
    if (vis[i] == false) {
        bfs(i, graph);
    }
}
```

Output = 0 1 2

q : ~~0~~ ~~1~~ ~~2~~ 4 3

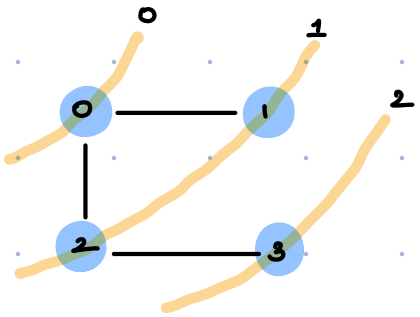
rem = 2

for (int nbr : graph[2]

{3}

* Tree is a special graph where every node will have a single parent & thus for n nodes, we will be having n-1 edges.

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



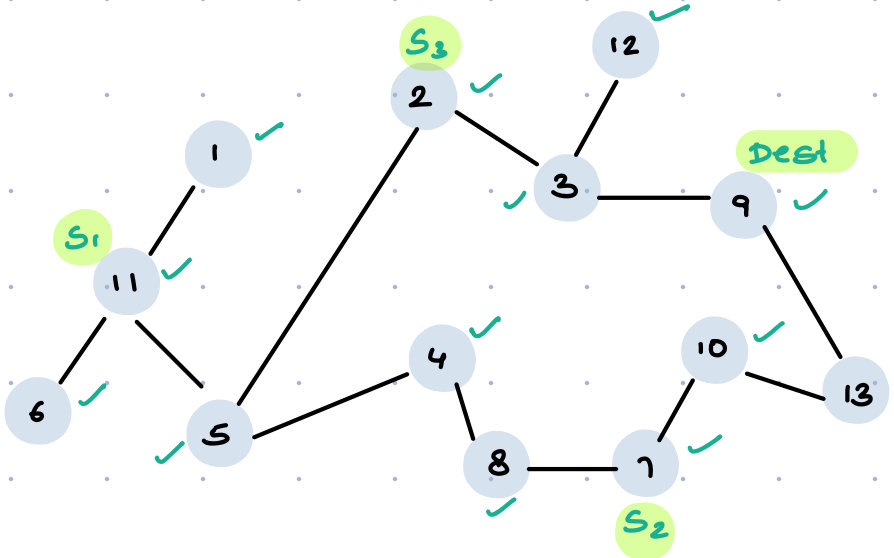
Ans = 0 1 2 3

Ans = 0 2 1 3

Multisource BFS ****

There are N no. of nodes and multisource $\{S_1, S_2, S_3\}$

Find the length of shortest path for given destination to any one of the sources



dest	Shortest path
9	2
13	2
5	1
7	0

Idea → Add all the sources in queue with dist = 0

queue : ~~(0,11)~~ ~~(0,7)~~ ~~(0,2)~~

~~(1,1)~~ ~~(1,6)~~ ~~(1,5)~~ ~~(1,8)~~ ~~(1,10)~~ ~~(1,3)~~

~~(2,4)~~ ~~(2,13)~~ ~~(2,12)~~ (2,9)

→ Ans = 2

```
public class pair {
    int dis;
    int vtx;

    pair (int d, int v) {
        dis = d;
        vtx = v;
    }
}

Queue <pair> q = new LinkedList <> ();

for (i=0; i < sources.length; i++) {
    q.add ( new pair (0, sources[i]));
    vis[ sources[i]] = true;
}

while (q.size () > 0) {

    pair rem = q.remove ();

    if (rem.vtx == destination) {
        ans = rem.dis;
        break;
    }

    for (int nbr : graph [rem.vtx]) {

        if (vis [nbr] == false) {
            vis [nbr] = true;
            pair np = new pair ( rem.dis + 1 , nbr);
            q.add (np);
        }
    }

}
return ans;
```

rem
→ vtx
→ dis

graph [rem.vtx]

Rotten Oranges

Given a matrix[N][M] with

Find the earliest time when all

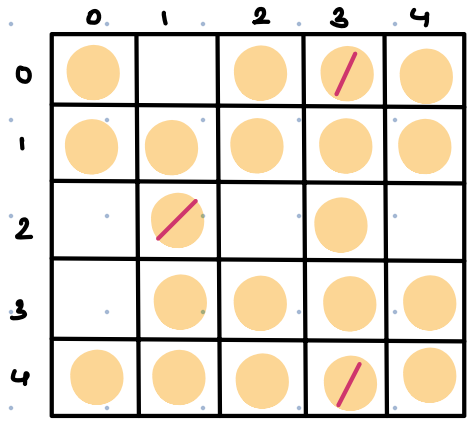
becomes rotten.

0 → Empty

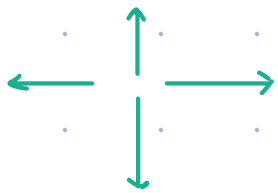
1 → Fresh

2 → Rotten

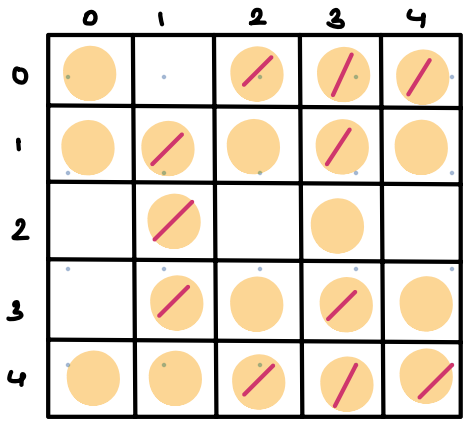
Note :- Return -1 if not possible.



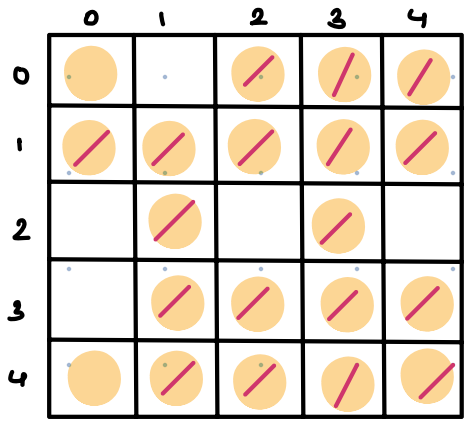
t = 0



valid directions

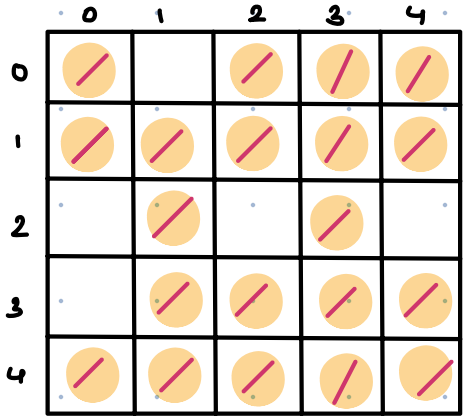


t = 1



t = 2

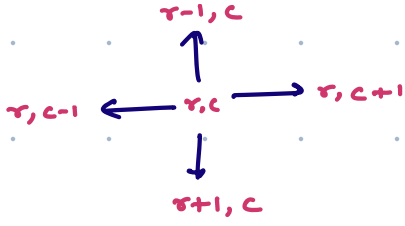
t = 3



Ans = 3

```
class pair {
    int i, j, time

    pair (int r, int c, int t) {
        r = i
        c = j
        time = t;
    }
}
```

```
Queue <pair> q = new LinkedList <> ();

int mintime = 0

for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
        if (A[i][j] == 2) {
            q.add (new pair (i, j, 0);
        }
    }
}
```



	Top	Left	D	R	
DR = {	-1	0	1	0	}
DC = {	0	-1	0	1	}

```
while (q.size () > 0) {
    pair rem = q.remove ();
    int t = rem.time;
    int r = rem.i;
    int c = rem.j;

    mintime = t;
```

TC : O(n*m)

SC : O(n*m)

```
        for (d=0; d<4; d++){
            int nr = r + DR[d]
            int nc = c + DC[d]
            if (nr≥0 && nc≥0 && nr<n && nc<m && A[nr][nc]==1)
                A[nr][nc] = 2;
                q.add(new pair(t+1, nr, nc);
            }
        }
    }
// check for fresh orange
for (i=0; i<n; i++){
    for(j=0; j<m; j++){
        if (A[i][j]==1){
            return -1;
        }
    }
}
return mintime;
```

10:22 – 10:35 pm

Flipkart Delivery Optimization

Problem :

Flipkart Grocery has several warehouses spread across the country and in order to minimize the delivery cost, whenever an order is placed we try to deliver the order from the nearest warehouse.

Therefore, each Warehouse is responsible for a certain number of localities which are closest to it for deliveries, this **minimizes the overall cost for deliveries**, effectively managing the distribution workload and minimizing the overall delivery expenses.

Problem statement:-

You are given a 2D matrix **A** of size **NxM** representing the map, where each cell is marked with either a **0 or a 1**. Here, a **0** denotes a locality, and a **1** signifies a warehouse. The objective is to calculate a new **2D matrix** of the same dimensions as **A**.

In this new matrix, the value of each cell will represent the minimum distance to the nearest warehouse. For the purpose of distance calculation, you are allowed to move to any of the **eight adjacent cells** directly surrounding a given cell.

Map =

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

Answer

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

Exactly rotten orange

No. of island

Q Given a matrix of integers with 1 & 0 in each cell.

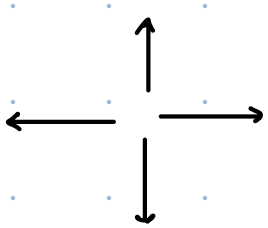
1 → land

0 → water

A set of connected 1's ⟶ island

Find the no. of island in the matrix

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

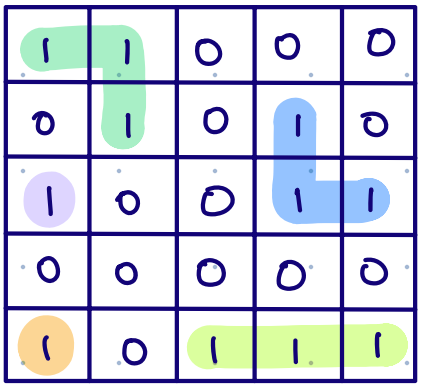


```
ans = 0
for (i=0; i<n; i++){
    for (j=0; j<m; j++){
        if (A[i][j]==1){
            ans++;
            dfs(A,i,j);
        }
    }
}
return ans;

void dfs (int[][]A, int i, int j){
    A[i][j] = 0

    for (d=0; d<4; d++){
        ni = i + DR[d];
        nj = j + DC[d];
        if (ni≥0 && nj≥0 && ni<n && nj<m && A[ni][nj]=1){
            dfs(A, ni, nj);
        }
    }
}
```

Tc: O(n*m)

Sc: O(n*m)

* BFS Approach




```
ans = 0
for (i=0; i<n; i++){
    for (j=0; j<m; j++){
        if (A[i][j]==1){
            ans++;
            bfs(A,i,j);
        }
    }
}
return ans;
```

```
void bfs (int[][]A, int r, int c){
    Queue <pair> q = new LL<>();
    q.add( new pair(r,c));
    A[i][j]=0;
    while (q.size() >0){
        pair rem= q.remove();
        int i = rem.r;
        int j= rem.c;
```

```
        for (d=0; d<4; d++){
            ni = i + DR[d];
            nj = j + DC[d];
            if (ni≥0 && nj≥0 && ni<n && nj<m && A[ni][nj]=1){
                A[ni][nj]=0;
                q.add(new pair(ni,nj));
            }
        }
    }
}
```

* Only focused on Traversal → BFS or DFS

* Shortest path, Minimum time } BFS

* Search for something, Path } DFS

When to Use BFS:

1. **Finding the Shortest Path in an Unweighted Graph**:
   - BFS explores nodes layer by layer, so the first time it encounters the target node, it's guaranteed to have found the shortest path in terms of the number of edges.
   - **Use case**: Pathfinding in unweighted graphs, like finding the shortest route in a maze or social network connections.
2. **Level-Order Traversal**:
   - BFS naturally processes nodes level-by-level, which makes it useful for applications that require a level-order traversal.
   - **Use case**: Finding nodes at a specific depth, hierarchical processing, or graph distances from a source node.
3. **Graphs with Finite Nodes or Limited Depth**:
   - BFS is suitable when the graph has a limited number of nodes or shallow depth, as it can be memory-intensive due to its need to store a large number of nodes in the queue.
4. **Finding the Connected Components**:
   - BFS can be used to find all nodes in a connected component in an undirected graph. It helps to explore all nodes reachable from a starting node.

When to Use DFS:

1. **Exploring All Paths or Searching for a Path Without Guaranteeing Shortest**:
   - DFS explores nodes as far down as possible, making it useful when you need to explore all possible paths from a starting node.
   - **Use case**: Solving puzzles or games, like finding all possible solutions in a maze.
2. **Tree Traversals (Inorder, Preorder, Postorder)**:
   - DFS is ideal for traversing binary trees and other hierarchical structures due to its depth-first nature.
   - **Use case**: Working with syntax trees in compilers, decision trees, and recursive structures.
3. **Detecting Cycles in a Graph**:
   - DFS can be used to detect cycles by marking nodes as they are explored and checking if any node is revisited in a single DFS path.
   - **Use case**: Deadlock detection in concurrent systems or detecting cycles in dependency graphs.
4. **Handling Large Graphs with Limited Memory**:
   - DFS typically uses less memory than BFS because it doesn't need to keep track of an entire level at once, only the path it's currently exploring.
   - **Use case**: Large, sparse graphs where storing the entire BFS queue would be memory-intensive.
5. **Topological Sorting and Strongly Connected Components**:
   - DFS is suited for tasks like topological sorting in directed acyclic graphs (DAGs) and finding strongly connected components.

* Shortest distance in maze

Problem Statement:

You are given a maze represented by a matrix A of size NxM where:

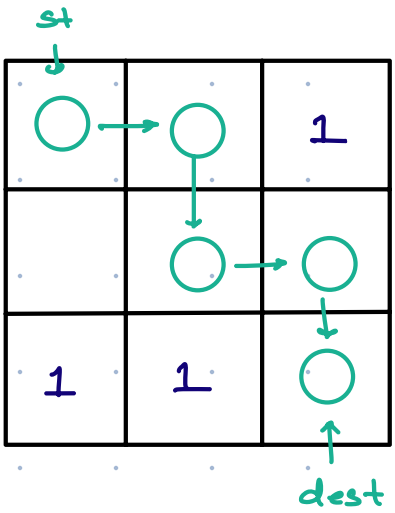
- 1 represents a wall.
- 0 represents an empty space.

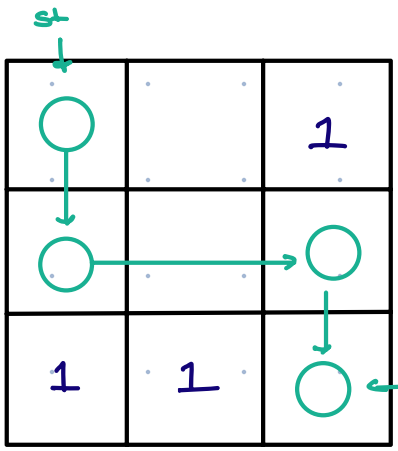
The maze has a ball that can roll through empty spaces in four directions: up, down, left, or right. The ball rolls until it hits a wall or boundary, at which point it can choose another direction.

You are also given:

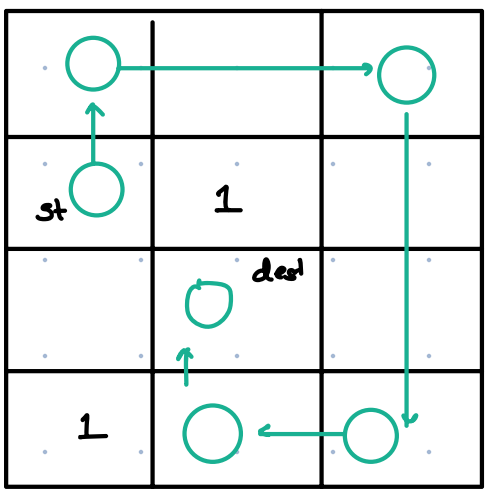
- Start position: A 1D array of Size 2, B with the starting coordinates of the ball.
- Destination position: A 1D array of Size 2, C with the destination coordinates of the ball.

Find the shortest distance for the ball to reach the destination from the starting position. The distance is defined as the number of empty spaces the ball travels through. If it's not possible to reach the destination, return -1.

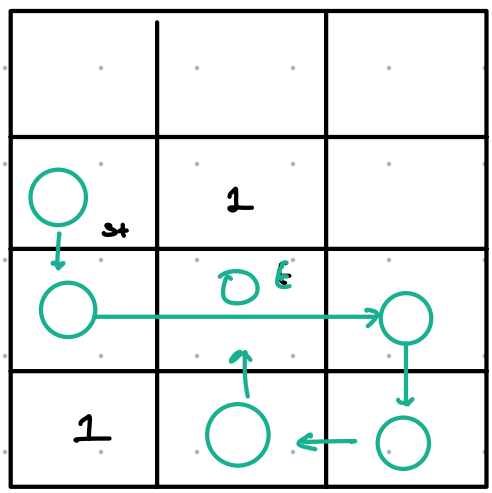




Ans = 4

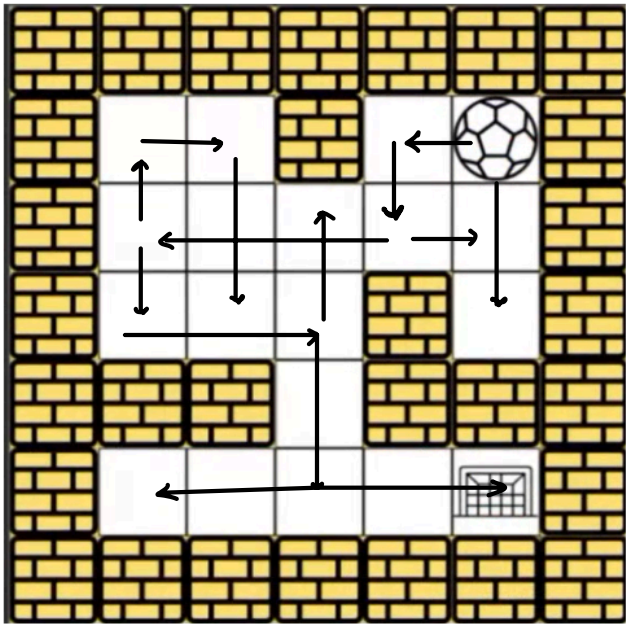


Ans = 8



Ans = 6

Ans = 6



Distance

6	7		1	START 0
5	-1	9	2	3
6	9	8		2
		-1		
12	-1	10	-1	12 END

Ans = 12

↑ ← → ↓

Dx = { -1 0 0 1 }

Dy = { 0 -1 1 0 }

```
// fill Dis[][] with value -1

q.add ( new pair (start_x, start_y));

while ( q.size() > 0) {

    pair rem = q.remove();

    r = rem.x ;
    c = rem.y ;

    for ( int i=0 ; i<4 ; i++) {

        x = r, y = c, cnt = 0

        while (x >= 0 && x < N && y >= 0 && y < M && A[x][y] == 0)

            x = x + Dx[i] ;
```

```
            y = y + Dy[i];
            cnt++;
        }
        x = x - dx[i];
        y = y - dy[i];
        cnt = cnt - 1
        dist = Dis[r][c] + cnt;
        if (Dis[x][y] == -1){
            Dis[x][y] = dist;
            q.add(new pair(x,y));
        }
    }
}
return A[end-x][end-y];
```