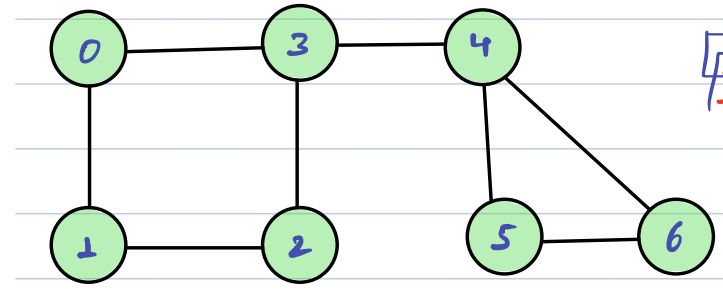
Today's agenda

- BFS Problem
- Rotting oranges
- dfs

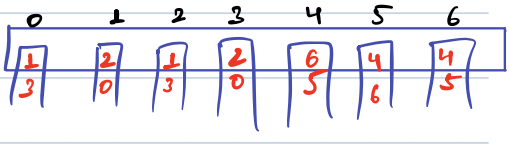
Q) Given undirected graph, source node and destination node. Check if destination node can be visited from source node or not?

SN: 0 DN: 2



→ 2E

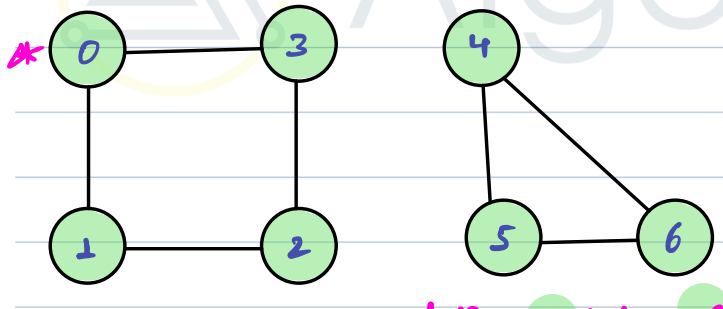
graph



N: 7 M: 7

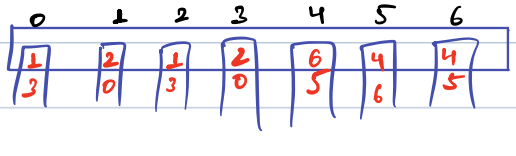
0 1
1 2
2 3
0 3
4 5
5 6
4 6

SN: 0 DN: 4



Disconnected graph

graph



queue: ~~0~~ ~~1~~ ~~2~~ ~~3~~

vis : T T T T F F F

0 1 2 3 4 5 6

sem = 2

// Pseudo Code

```
void BFS ( int N , int M, int[][] edges, int sn, int dn){

    List<List<Integer>> graph = construction(N,M, edges);

    Queue<Integer> q;
    boolean[] vis = new int[N];

    q.add(sn);
    vis[sn] = true;
    while (q.size() > 0 ){
        int rem = q.remove();
        S.O.P(rem);

        // add all unvisited nbrs.
        List<Integer> nbrs = graph.get(rem);

        for (int v: nbrs){
            if (vis[v] == false){
                q.add(v);
                vis[v] = true;
            }
        }
    }
    if (vis[dn] == false) {return false;}
    else { return true;}
}
```

T.C: O(V+E)

S.C: O(V)

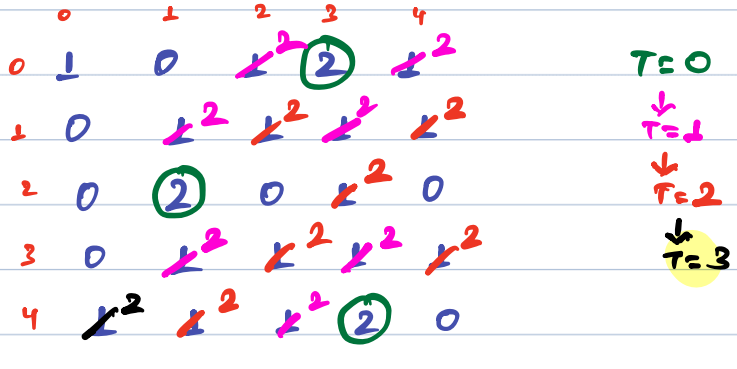
Q) Rotting oranges

mat[N][M] →
- 0 {empty}
- 1 {fresh orange}
- 2 {rotten orange}

↳ Every minute any fresh orange adjacent to rotten orange becomes rotten. Find min time when all oranges become rotten. if not possible to rott all oranges, return -1

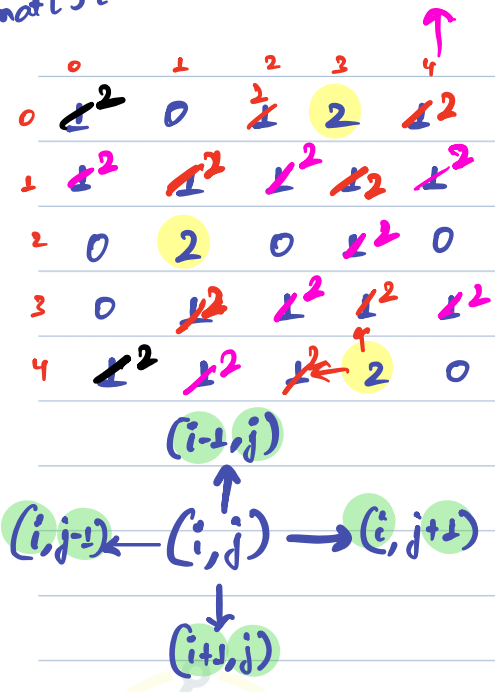
Ex1:

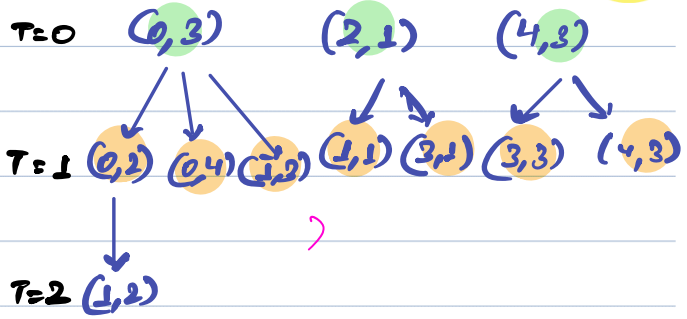
Ex2:



multisource bfs

mat[][]





i j

```
Class Pair {
    int i;
    int j;
    int t;
}
```

q:

rem: {1, 0, 2}

//Pseudo code

```
int rottenOranges(int mat[n][m]){
    Queue<Pair> q;

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(mat[i][j]==2){
                Pair p = new Pair(i,j,0);
                q.add(p);
            }
        }
    }
    int ans=-1;
    while(q.size()>0){
        Pair rem = q.remove();
        int crow = rem.i;
        int ccol = rem.j;
        int ctime = rem.t;

        ans = ctime;
        // crow-1, ccol
        if(crow-1>=0 && mat[crow-1][ccol]==1){
            Pair p = new Pair(crow-1, ccol, ctime+1);
            q.add(p);
            mat[crow-1][ccol] = 2
        }
```

T.C: O(n*m)

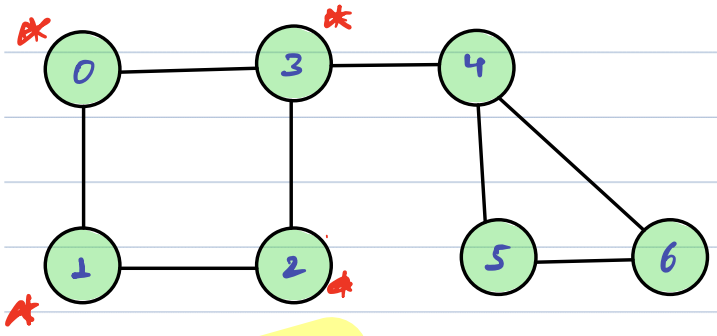
S.C: O(n*m)

```
// crow, ccol+1
if(ccol+1<m && mat[crow][ccol+1]==1){
  Pair p = new Pair(crow, ccol+1, ctime+1);
      q.add(p);
      mat[crow][ccol+1] =2
}

// crow+1, ccol

  if (crow+1<n && mat[crow+1][ccol]==1){
     Pair p= new Pair(crow+1, ccol, ctime+1);
        q.add(p);
        mat[crow+1][ccol] = 2;
  }

// crow, ccol-1
if(ccol-1>=0 && mat[crow][ccol-1]==1){
  Pair p = new Pair(crow, ccol-1, ctime+1);
      q.add(p);
      mat[crow][ccol-1] =2
}

}

for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){
```

```
        if (mat[i][j] == 1){
            return -1;
        }
    }
}
return ans;
}
```

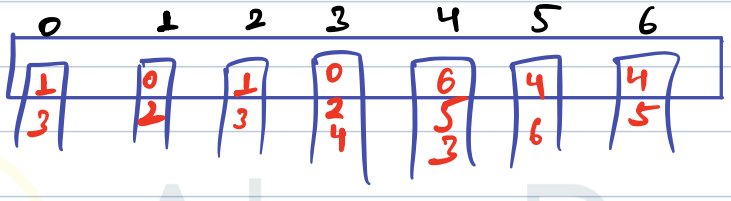
Break till 9:53 PM

DFS → Depth first search Traversal



→ traversal in any order.

graph



Src

0

nbrs

1 3

v=1

1

i

0 2

v=0 2

2

i

1 3

v=1 3

3

//Pseudo Code

```
void dfs (List<List<Integer>> graph, boolean vis[], int src) {

    List<Integer> nbrs = graph.get(src);

    for (int v: nbrs) {
        if (vis[v] == false) {
            vis[v] = true;
            dfs (graph, vis, v);
        }
    }

}
```

T.C: O(V+E)

S.C: O(V)

```
void dfs (List<List<Integer>> graph, boolean vis[], int src) {
    1 List<Integer> nbrs = graph.get(src);
    2 for (int v: nbrs) {
        if (vis[v] == false) {
            vis[v] = true;
            dfs (graph, vis, v);
        }
    }
}
```

graph