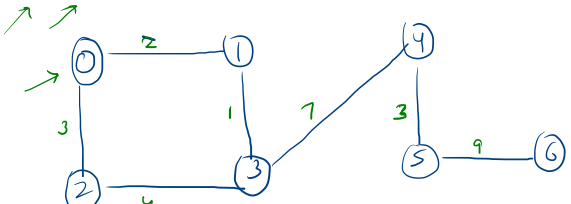
graph

src, des

Vertex => V

8 Vertex

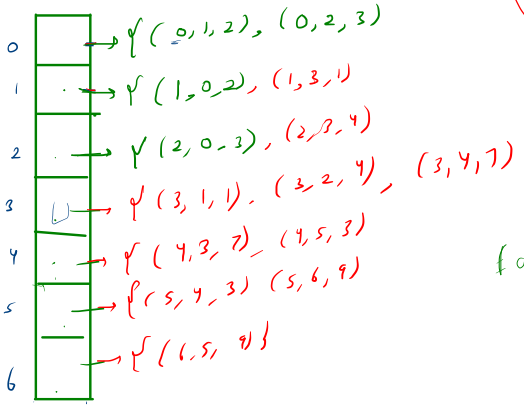
|| nodes



ArrayList<Edge>[] graph = new ArrayList[V]

O(E)

Adjacency list =>

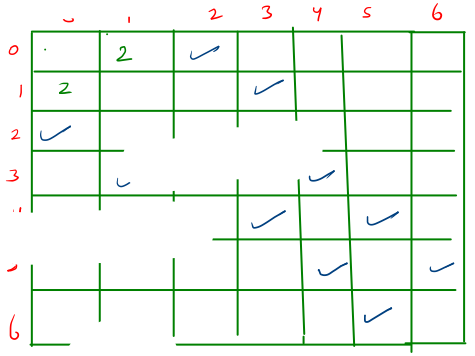


0 → (0,1,2) (0,2,3)

0 → (1,2) (2,3)

Edge d

V

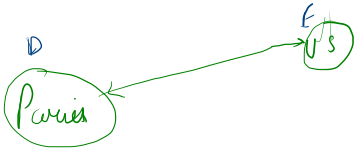


=> adjacency matrix

not efficient

O(V²)

O(V)

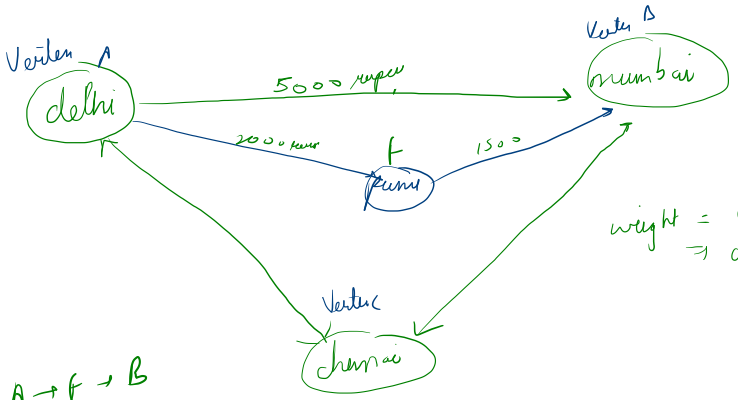


Vertex

edges

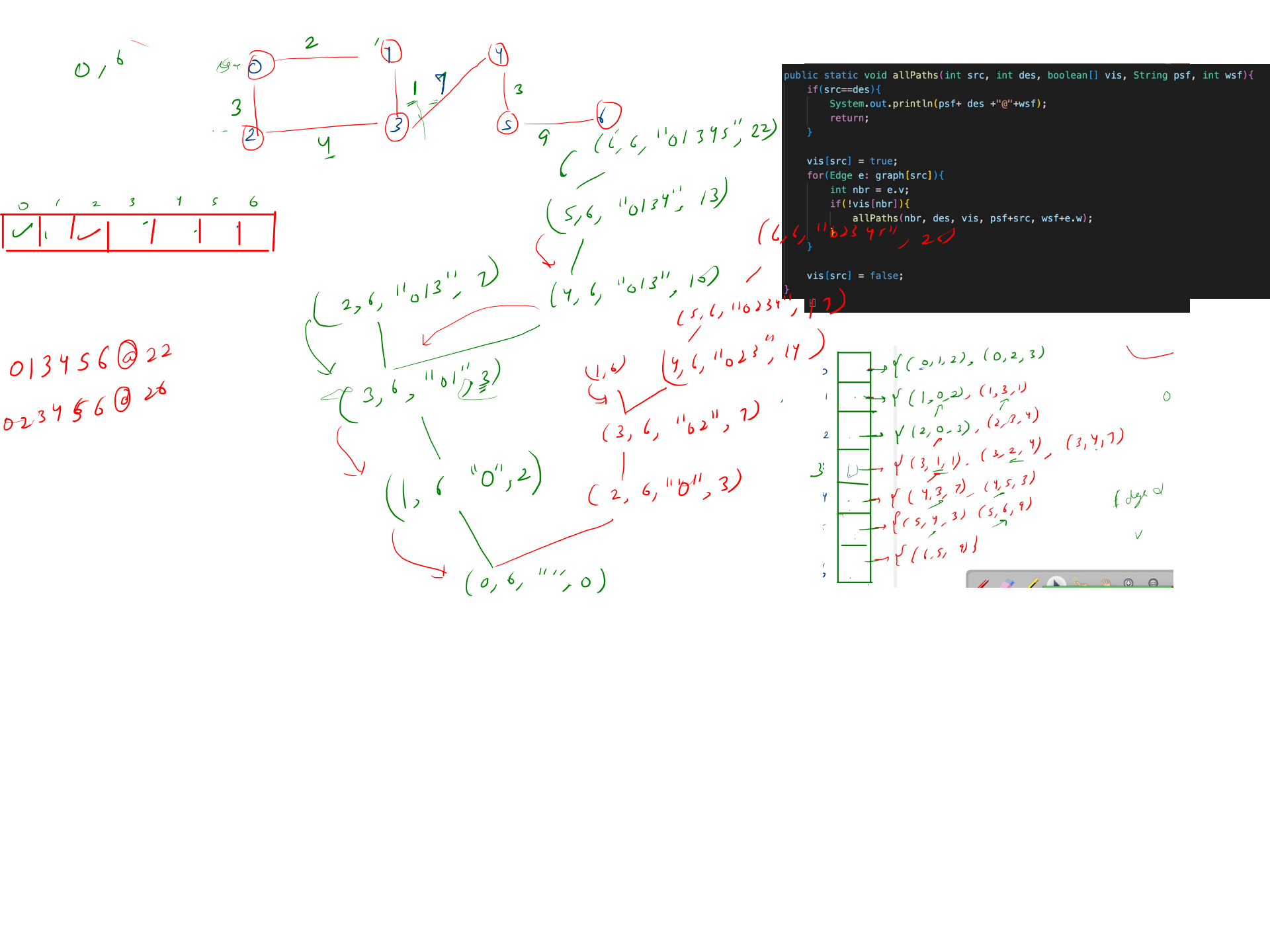
weight

car



weight = cost
=> distance

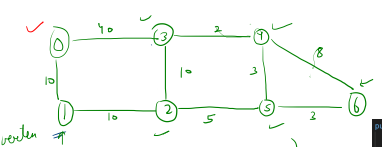
A → F → B

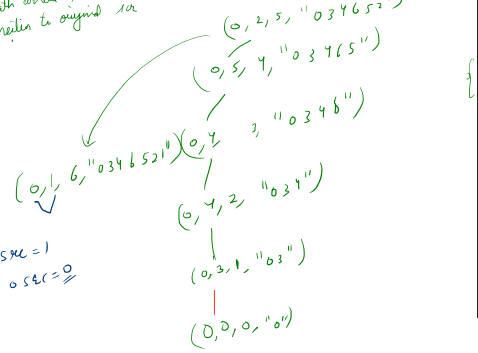

0, 6
0 1 2 3 4 5 6
0 1 3 4 5 6 @ 22
0 2 3 4 5 6 @ 26
(6, 6, "01345", 22)
(5, 6, "0134", 13)
(4, 6, "013", 10)
(2, 6, "013", 7)
(3, 6, "01", 3)
(1, 6, "0", 2)
(0, 6, "", 0)
(6, 6, "02345", 20)
(5, 6, "0234", 14)
(4, 6, "023", 11)
(3, 6, "62", 7)
(2, 6, "0", 3)
public static void allPaths(int src, int des, boolean[] vis, String psf, int wsf){
if(src==des){
System.out.println(psf+ des +"@"+wsf);
return;
}
vis[src] = true;
for(Edge e: graph[src]){
int nbr = e.v;
if(!vis[nbr]){
allPaths(nbr, des, vis, psf+src, wsf+e.w);
}
}
vis[src] = false;
}

node → 1 time

Hamiltonian path
→ visit every vertex one time

Hamiltonian cycle
⇒ Hamiltonian path where last vertex has a connection to original src






```
public static void hamiltonianPathCycle(int osrc, int src, boolean[] vis, int edgeCount, String psf){
    if(edgeCount == N-1){
        if(checkifEdgeExist(src,osrc)){
            System.out.println("This is a hamiltonian cycle "+psf);
        } else {
            System.out.println("This is a hamiltonian path "+psf);
        }
        return;
    }

    vis[src] = true;

    for(Edge e: graph[src]){
        int nbr = e.v;
        if(!vis[nbr]){
            hamiltonianPathCycle(osrc, nbr, vis, edgeCount + 1, psf+nbr);
        }
    }

    vis[src] = false;
}

public static boolean checkifEdgeExist(int u, int v){
    for(Edge e:graph[u]){
        if(e.v == v){
            return true;
        }
    }

    return false;
}
```

src = 0
dest = 6
criteria = 40
k = 4
lpathwt = 50 66
lpath = 0346 0325476
fpathwt = 28 38
fpath = 0256 0123476

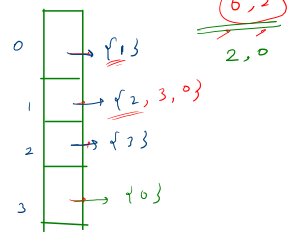
0346 50
03476 48
0325476 66
03256 58

```
public static void multisolver(ArrayList<Edge>[] graph, int src, int dest, boolean[] visited, int criteria, int k, String psf, int wsf){
    if(src==dest){
        pq.add(new Pair(wsf,psf));
        if(pq.size()>k){
            pq.remove();
        }

        if(wsf > lpathwt){
            lpath = psf;
            lpathwt = wsf;
        }

        if(wsf < criteria && wsf > fpathwt){
            fpathwt = wsf;
            fpath = psf ;
        }
    }

    visited[src] = true;

    for(Edge e: graph[src]){
        int nbr = e.nbr;
        if(!visited[nbr]){
            multisolver(graph, nbr, dest, visited, criteria, k, psf+nbr, wsf+e.wt);
        }
    }

    visited[src] = false;
}
```

n = 4

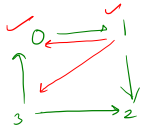
1 2 {0, 1}
2 3 {1, 2}
3 4 {2, 3}
4 1 {3, 0}

① ② ③ ④



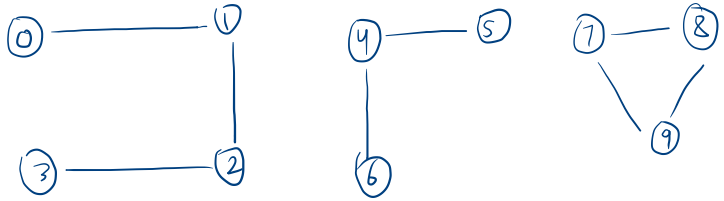
0, 3
6, 2
2, 0

n-1

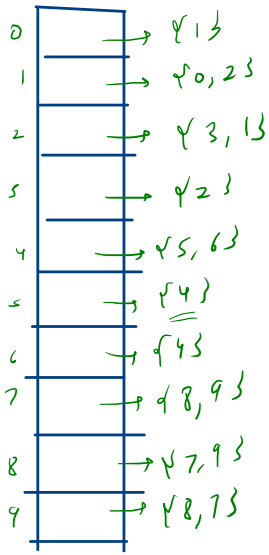


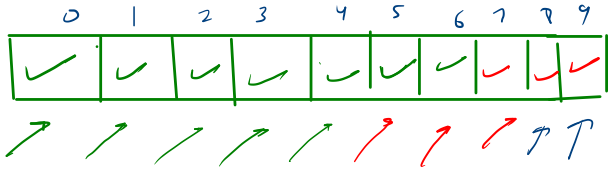

```
boolean hasPath(int src, int des, ArrayList<Integer>[] graph, boolean[] vis){
    if(src == des){
        return true;
    }

    boolean ans = false;
    vis[src]= true;
    for(int nbr: graph[src]){
        if(!vis[nbr]){
            ans = ans || hasPath(nbr, des, graph, vis);
        }
    }

    return ans;
}

boolean check(int N, int M, ArrayList<ArrayList<Integer>> Edges,int U,int V) {
    // N vertices, M edges
    ArrayList<Integer>[] graph = new ArrayList[N];
    for(int i=0; i<N; i++){
        graph[i] = new ArrayList<>();
    }

    for(ArrayList<Integer> edge:Edges){
        int u = edge.get(index: 0);
        int v = edge.get(index: 1);

        graph[u-1].add(v-1);
    }

    return hasPath(U-1, V-1, graph, new boolean[N]) || hasPath(V-1, U-1, graph,new boolean[N]);
}
```

10 vertices



3 connected components





count = ~~0~~ ~~1~~ ~~2~~ 3

```java
public static void dfs(int src, boolean[] vis){
    vis[src]= true;

    for(Edge e: graph[src]){
        int nbr = e.v;
        if(!vis[nbr]){
            dfs(nbr, vis);
        }
    }
}
```

```java
public static int numberOfComponents(){
    boolean[] visited = new boolean[N];
    int count=0;

    for(int i=0; i<N; i++){
        if(visited[i]==false){
            dfs(i,visited);
            count++;
        }
    }

    return count;
}
```

leetcode 200

2 4

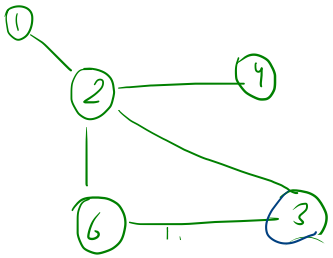
6 2

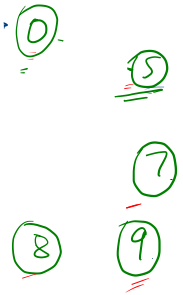
3 6

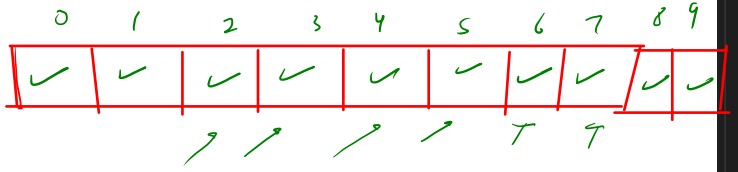
2 1

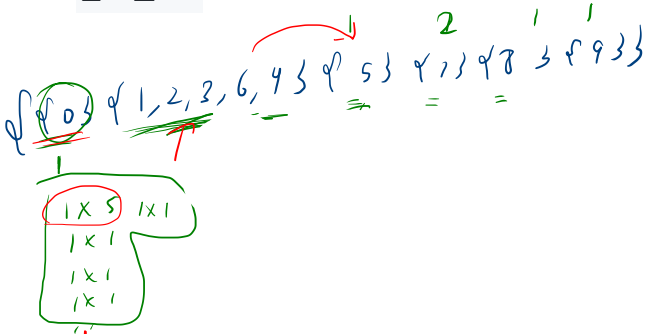
2 3

1 2

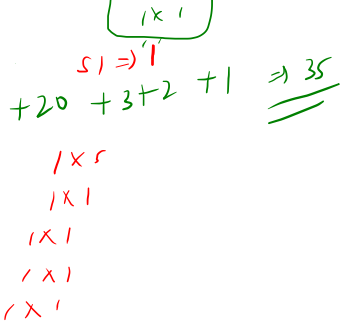




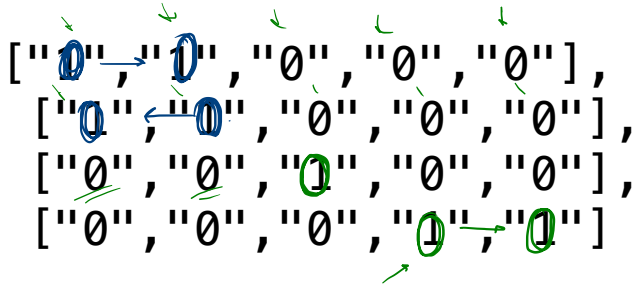




break till 10:20



```
public static int getSize(int src, boolean[] vis, ArrayList<Edge>[] graph){
    vis[src]= true;

    int ans = 0;
    for(Edge e: graph[src]){
        int nbr = e.nbr;
        if(vis[nbr]==false){
            int nbrSize = getSize(nbr, vis, graph);
            ans += nbrSize;
        }
    }

    return ans + 1;
}
public static int perfectStudents(int vtces,  ArrayList<Edge>[] graph){
    ArrayList<Integer> components = new ArrayList<>();

    boolean[] visited = new boolean[vtces];
    for(int i=0; i<vtces; i++){
        if(visited[i]==false){
            int size  = getSize(i,visited,graph);
            components.add(size);
        }
    }

    int pairs=0;
    for(int i=0; i<components.size(); i++){
        int s1 = components.get(i);
        for(int j=i+1; j<components.size(); j++){
            int s2 = components.get(j);
            pairs = pairs +(s1*s2);
        }
    }

    return pairs;
}
```



count = 2 + 2 = 3

```java
public void dfs(int i, int j, char[][] grid){
    grid[i][j] = '0'; // changing land to water

    int[][] dirs = {{0,1},{1,0},{0,-1},{-1,0}};

    for(int[] dir:dirs){
        int x = i+dir[0];
        int y =j + dir[1];

        if(x>=0 && y>=0 && x<grid.length && y<grid[0].length && grid[x][y]=='1'){
            dfs(x,y,grid);
        }
    }
}

public int numIslands(char[][] grid) {
    int n = grid.length;
    int m = grid[0].length;

    int count = 0;
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(grid[i][j]=='1'){
                dfs(i,j,grid);
                count++;
            }
        }
    }

    return count;
}
```

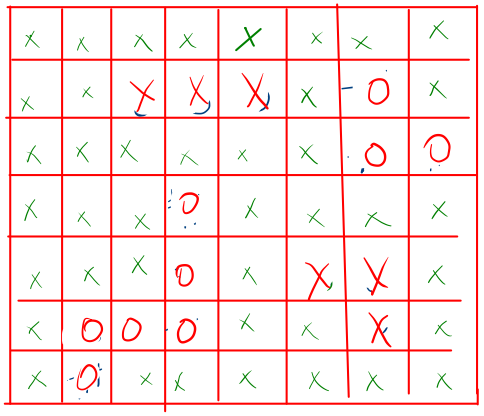
(0,1)

(0,4)

surrounded regions

'O' → '#'

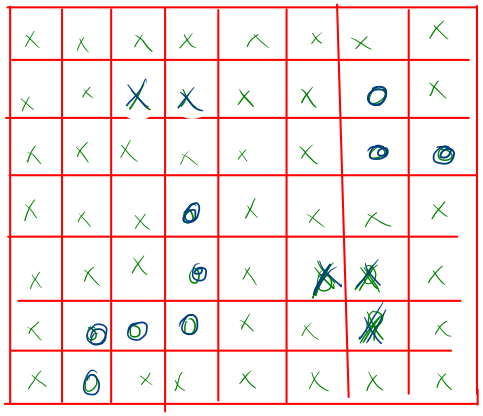
src → dest
dest → src



⇒

'O' → 'X'

'#' → 'O'



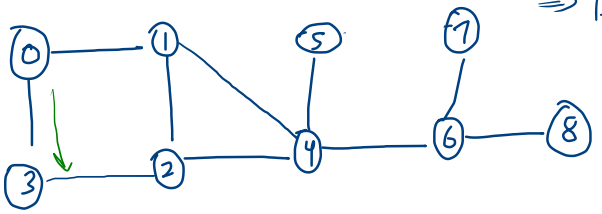
(i, j) → boundary ⇒ not convert to 'X'

boundary

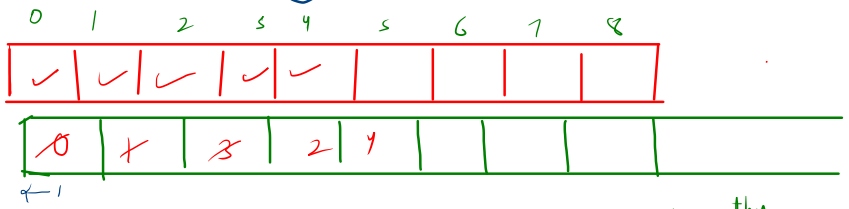
```java
public void solve(char[][] board) {
    int n = board.length;
    int m = board[0].length;

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(i==0 || i==n-1 || j==0 || j==m-1){
                if(board[i][j]=='O')
                    convert(i,j,board);
            }
        }
    }

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(board[i][j]=='O'){
                board[i][j]='X';
            } else if(board[i][j]=='#'){
                board[i][j]='O';
            }
        }
    }
}
```

BFS ⇒ breadth first search

⇒ BFS ensures shortest path



src = 0



level =

removal time → you will discover all the paths
↳ slow ↳ shortest to longest

insertion time → shortest path
↳ fast

```
que.add(src);
int level = 0;

while(que.size()>0){
    int s = que.size();
    System.out.print("This is level " + level+ " -> ");
    while(s-->0){
        int vtx = que.removeFirst();

        if(vis[vtx]==true){
            System.out.println("There is a cycle" + vtx);
            continue;
        }

        vis[vtx] = true;
        System.out.print(vtx+" ");
        for(Edge e:graph[vtx]){
            int nbr = e.v;
            if(!vis[nbr]){
                que.addLast(nbr);
            }
        }
    }
    level++;
    System.out.println();
}
}
```