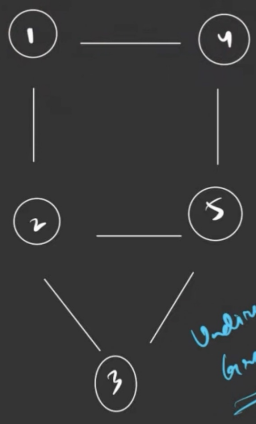
Graph 24/4/23.

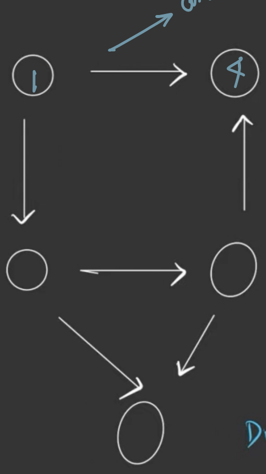
lec 1 → introduction to graph



→ nodes / vertex

v / N=5

Undirected Graph



directed edge

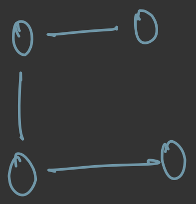
1 → 4

4 → 1 ✗

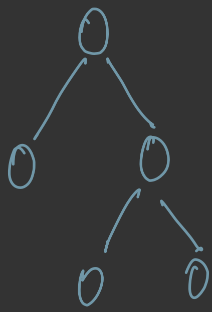
acyclic graph (no cycle)

Directed → all the edges are directed
Graph → it can have multiple direction
→ bidirectional

Cycles in a graph (circular)



graph

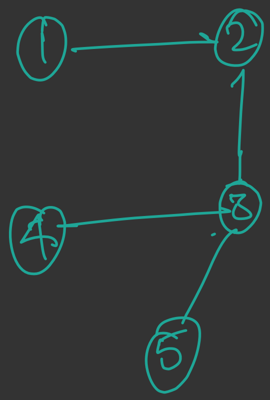


graph

graph → 1. atleast a node
2. edge/s

cycle → start from a node & reach back to the same node

path → contains a lot of nodes & each of them are reachable

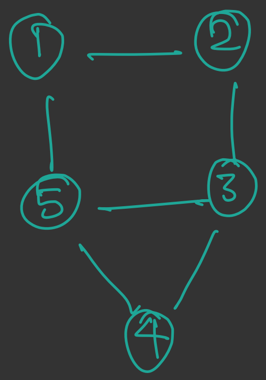


1 2 3 5 → path

1 2 3 2 1 → not path → a node cannot appear more than one in a path

Degrees in Graph

undirected graph ⇒ the num of edges attached to a node



D[3] → 3

D[4] → 2

property:

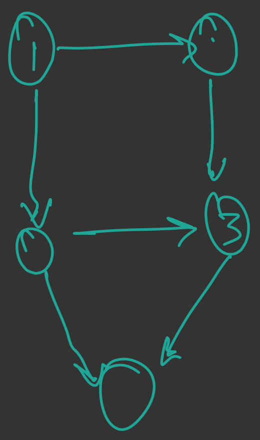
total degrees of graph = 2× num of edges in the graph

bcoz every edge is associated with 2 nodes

Directed Graph :-

indegree(node) → num of incoming edges

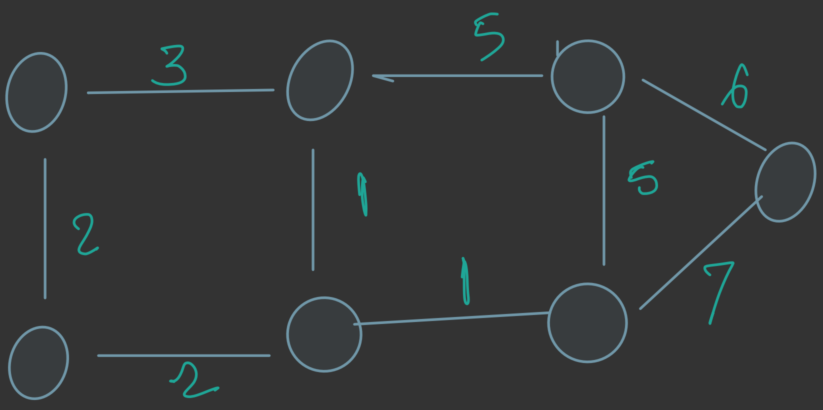
outdegree(node) → num of outgoing edges



indegree(3) → 2

outdegree(3) → 1

edge weight ⇒ weight of an edge, if not given, take 1 unit



lec 2 → Graph representation in C++

input → number of node, numbers of edges, type of graph (directed or not directed)

m lines that will represent edges (this can be given in any order)

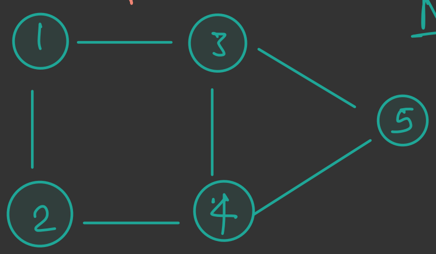
5, 6
↑ ↑
node ← n m edges

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

two ways to store

1. Adjacency matrix
2. Adjacency list

Adjacency matrix



N=5
nodes are 1 based indexing

matrix ⇒ adj [N+1] [N+1] = 0

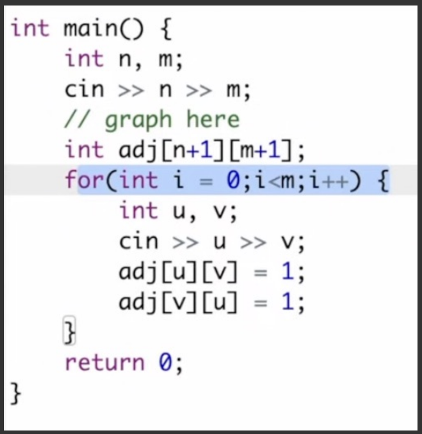
(2,1), (1,2)

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

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

space → NXN
↑
costly.

Time → O(N)



```
int main() {
    int n, m;
    cin >> n >> m;
    // graph here
    int adj[n+1][m+1];
    for(int i = 0;i<m;i++) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1;
    }
    return 0;
}
```

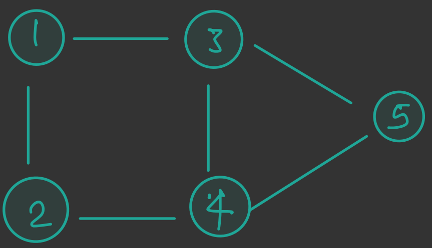
```
function graph() {
    let n = prompt();
    let m = prompt();
    let mat = Array(n+1).fill().map(() => Array(m+1).fill(0));

    for(let i=0; i<m; i++){
        let u = prompt();
        let v = prompt();
        mat[u][v] = 1;
        mat[v][u] = 1;
    }
}
```

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

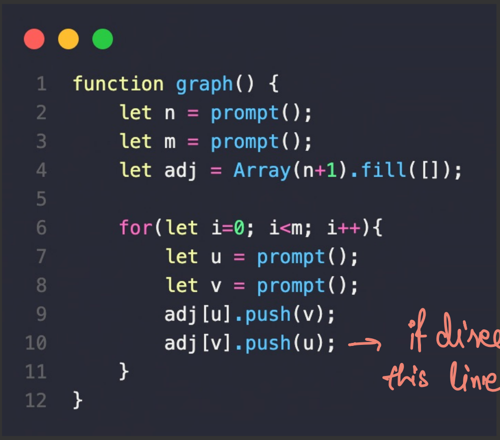
0 →
1 → {2, 3}
2 → {1, 4}
3 → {1, 4, 5}
4 → {2, 3, 5}
5 → {3, 4}

Adjacency List

adj[n+1]



SC → O(2E)

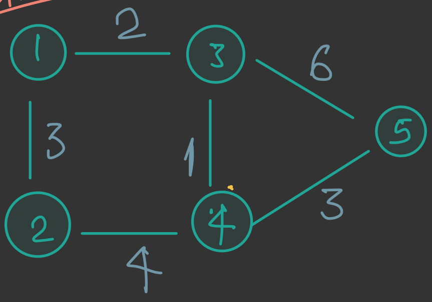


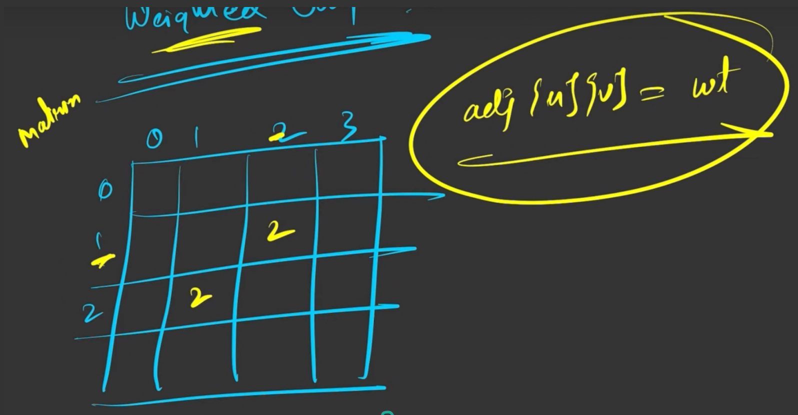
```
function graph() {
    let n = prompt();
    let m = prompt();
    let adj = Array(n+1).fill([]);

    for(let i=0; i<m; i++){
        let u = prompt();
        let v = prompt();
        adj[u].push(v);
        adj[v].push(u);
    }
}
```

→ if directed this line is not needed
↳ SC → O(E)

How to store weighted graph

matrix





adj[u][v] = wt

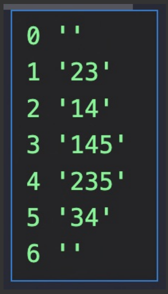
{ (a,b), }
↑ node ↑ weight

List

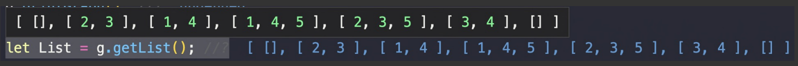
0 →
1 → {(3,2), (2,3)}
2 → {(1,3), (4,4)}
3 → {(1,2), (4,1), (5,6)}
4 → {(2,4), (3,1), (5,3)}
5 → {(3,6), (4,3)}

```javascript
class Graph {
    constructor(noOfVertices) {
        this.noOfVertices = noOfVertices;
        this.List = new Array(noOfVertices+1).fill().map(() => Array());
    }

    addEdge(v, w) {
        this.List[v].push(w);
        this.List[w].push(v);
    }

    printGraph() {
        for (let i=0; i<=this.noOfVertices; i++) {
            console.log(i, this.List[i].join(''));
        }
    }

    getList() {
        return this.List;
    }
}
```

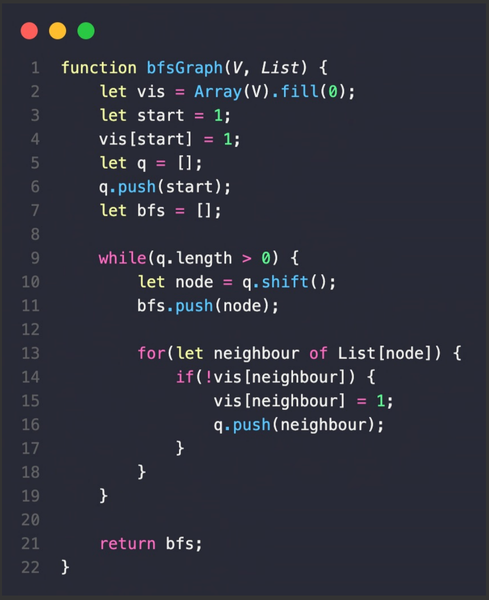
```javascript
// Using the above implemented graph class
var g = new Graph(6);
var vertices = [ '1', '2', '3', '4', '5', '6' ];

// adding edges
g.addEdge(1, 2);
g.addEdge(1, 3);
g.addEdge(2, 4);
g.addEdge(3, 4);
g.addEdge(3, 5);
g.addEdge(4, 5);

g.printGraph(); //?

let List = g.getList(); //?
```



```
0 ''
1 '23'
2 '14'
3 '145'
4 '235'
5 '34'
6 ''
```



```
[ [], [ 2, 3 ], [ 1, 4 ], [ 1, 4, 5 ], [ 2, 3, 5 ], [ 3, 4 ], [] ]
let List = g.getList(); //?  [ [], [ 2, 3 ], [ 1, 4 ], [ 1, 4, 5 ], [ 2, 3, 5 ], [ 3, 4 ], [] ]
```



```javascript
function bfsGraph(V, List) {
    let vis = Array(V).fill(0);
    let start = 1;
    vis[start] = 1;
    let q = [];
    q.push(start);
    let bfs = [];

    while(q.length > 0) {
        let node = q.shift();
        bfs.push(node);

        for(let neighbour of List[node]) {
            if(!vis[neighbour]) {
                vis[neighbour] = 1;
                q.push(neighbour);
            }
        }
    }

    return bfs;
}
```

```javascript
function dfs(node, List, vis, result) {
    vis[node] = 1;
    result.push(node);

    for(let neighbour of List[node]){
        if(!vis[neighbour]){
            dfs(neighbour, List, vis, result);
        }
    }
}

function dfsGraph(V, List){
    let vis = Array(V).fill(0);
    let start = 1;
    let result = [];
    dfs(start, List, vis, result);
    return result;
}
```

convert adjacency matrix to List

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

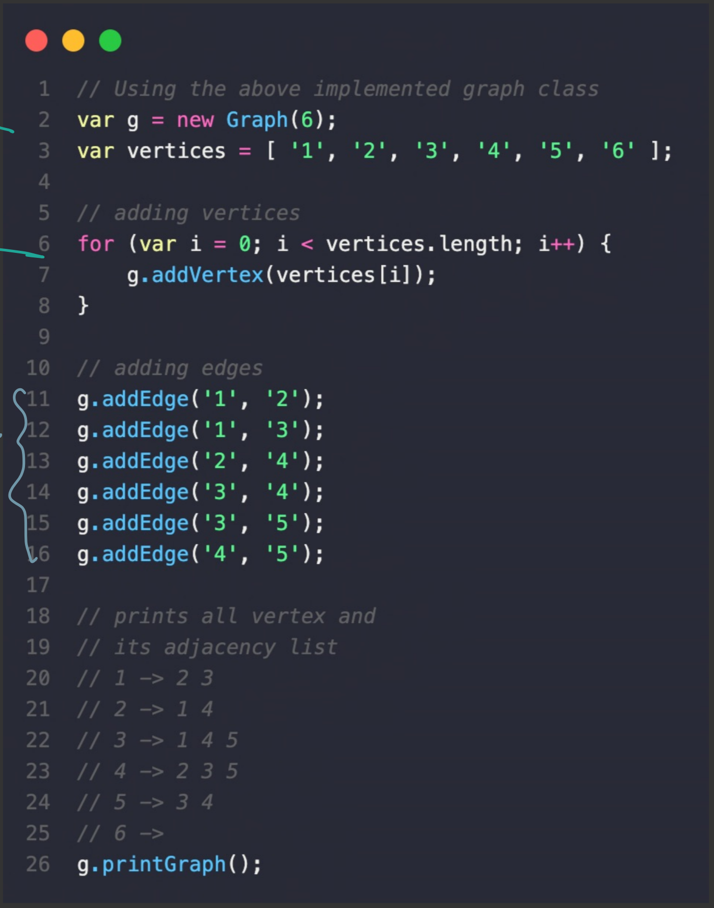
1 → [3]
2 → [ ]
3 → [1]

```javascript
function matToList(mat){
    let V = mat.length;
    let List = Array.from({ length: V+1 }, () => []);

    for(let i=0; i<V; i++){
        for(let j=0; j<V; j++){
            if(mat[i][j] == 1 && i != j) {
                if(!List[i+1].includes(j+1)){
                    List[i+1].push(j+1);
                }
                if(!List[j+1].includes(i+1)){
                    List[j + 1].push(i + 1);
                }
            }
        }
    }
    return List;
}
```

```
class Graph {
    constructor(noOfVertices) {
        this.noOfVertices = noOfVertices;
        this.List = new Map();
    }

    addVertex(v) {
        // initialize the adjacent list with a
        // null array
        this.List.set(v, []);
    }

    addEdge(v, w) {
        // get the list for vertex v and put the
        // vertex w denoting edge between v and w
        this.List.get(v).push(w);

        // Since graph is undirected,
        // add an edge from w to v also
        this.List.get(w).push(v);
    }

    // Prints the vertex and adjacency list
    printGraph() {
        // get all the vertices
        var get_keys = this.List.keys();

        // iterate over the vertices
        for (var i of get_keys) {
            // get the corresponding adjacency list
            // for the vertex
            var get_values = this.List.get(i);
            var conc = "";

            // iterate over the adjacency list
            // concatenate the values into a string
            for (var j of get_values) conc += j + " ";

            // print the vertex and its adjacency list
            console.log(i + " -> " + conc);
        }
    }

    // bfs(v)
    // dfs(v)
}
```

created object
g with vertices 6 ←

list = {
1: [ ] [2,3]
2: [ ] [1,4]
3: [ ] [1,4,5]
4: [ ] [2,3,5]
5: [ ] [3,4]
6: [ ]
}



```
// Using the above implemented graph class
var g = new Graph(6);
var vertices = [ '1', '2', '3', '4', '5', '6' ];

// adding vertices
for (var i = 0; i < vertices.length; i++) {
    g.addVertex(vertices[i]);
}

// adding edges
g.addEdge('1', '2');
g.addEdge('1', '3');
g.addEdge('2', '4');
g.addEdge('3', '4');
g.addEdge('3', '5');
g.addEdge('4', '5');

// prints all vertex and
// its adjacency list
// 1 -> 2 3
// 2 -> 1 4
// 3 -> 1 4 5
// 4 -> 2 3 5
// 5 -> 3 4
// 6 ->
g.printGraph();
```

---

lec3 → Graph representation in Java

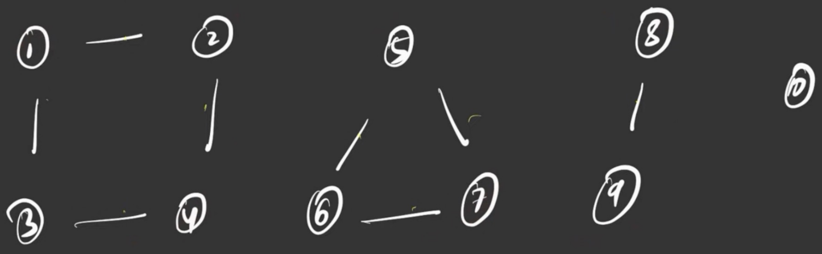
---

lec4 → what are connected components

N = 10, M = 8

edges

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

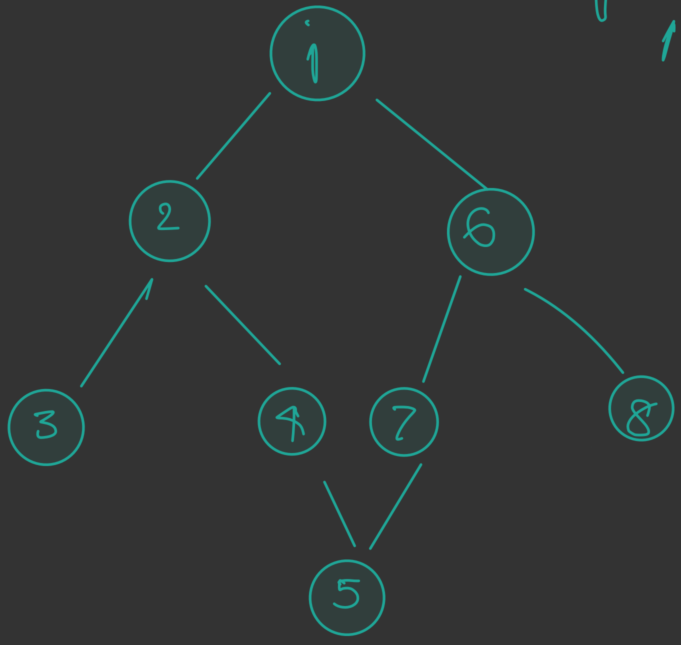


4 components.

⇒ for any traversal, always keep a visited array/map for node
if a node is not visited, run the loop for that component

lec 5 → BFS

always check if given nodes are 0 based or 1 based



N=8

level wise.

if starting node =1

1 2 6 3 4 7 8 5

if starting node = 6

6 1 7 8 2 5 3 4

level means node should be at equal distance from starting node

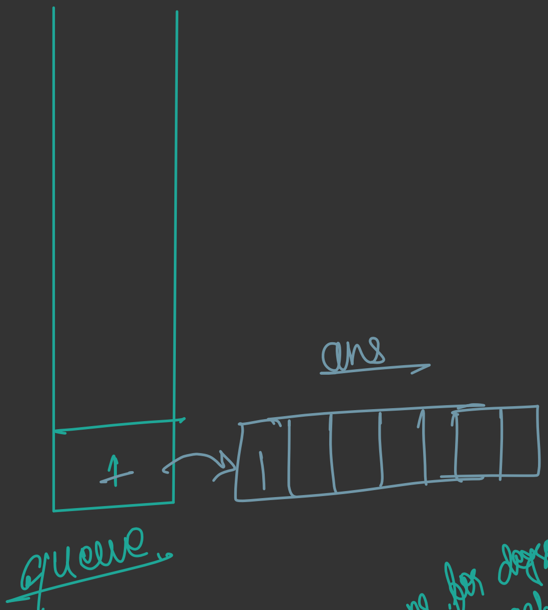
---

starting node = 1

visited = → n+1

starting node

0	1	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8



→ shift from queue, add it to ans

→ traverse the neighbours of shifted node

if it's not visited already push it to the queue

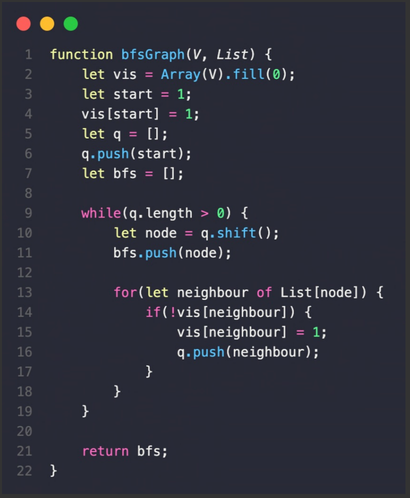
→ when queue is empty return the ans.

runs for degree of each node

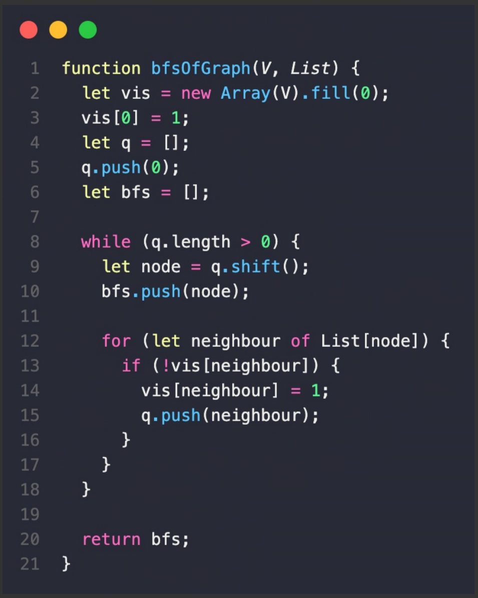
inner for loop

0 → { }
1 → {2,6}
2 → {1,3,4}
3 → {2}
4 → {2,5}
5 → {4,7}
6 → {1,7,8}
7 → {6,5}
8 → {6}

TC → O(N)+O(2E)

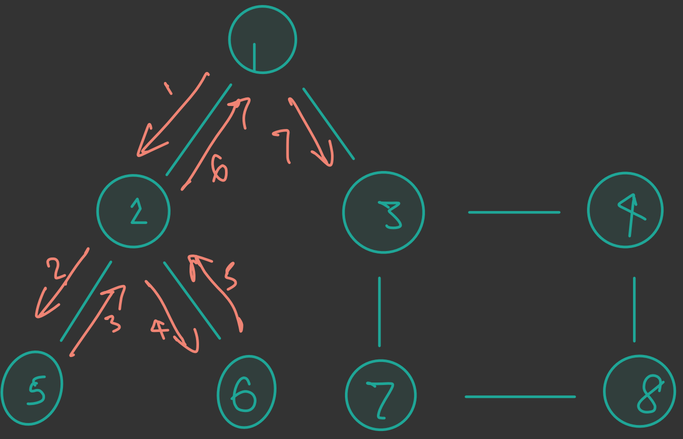


```
function bfsGraph(V, List) {
    let vis = Array(V).fill(0);
    let start = 1;
    vis[start] = 1;
    let q = [];
    q.push(start);
    let bfs = [];

    while(q.length > 0) {
        let node = q.shift();
        bfs.push(node);

        for(let neighbour of List[node]) {
            if(!vis[neighbour]) {
                vis[neighbour] = 1;
                q.push(neighbour);
            }
        }
    }

    return bfs;
}
```



```
function bfsOfGraph(V, List) {
  let vis = new Array(V).fill(0);
  vis[0] = 1;
  let q = [];
  q.push(0);
  let bfs = [];

  while (q.length > 0) {
    let node = q.shift();
    bfs.push(node);

    for (let neighbour of List[node]) {
      if (!vis[neighbour]) {
        vis[neighbour] = 1;
        q.push(neighbour);
      }
    }
  }

  return bfs;
}
```

lec6 DFS



starting node 1

1 2 5 6 3487 ✓

3784 ✓

starting node 3

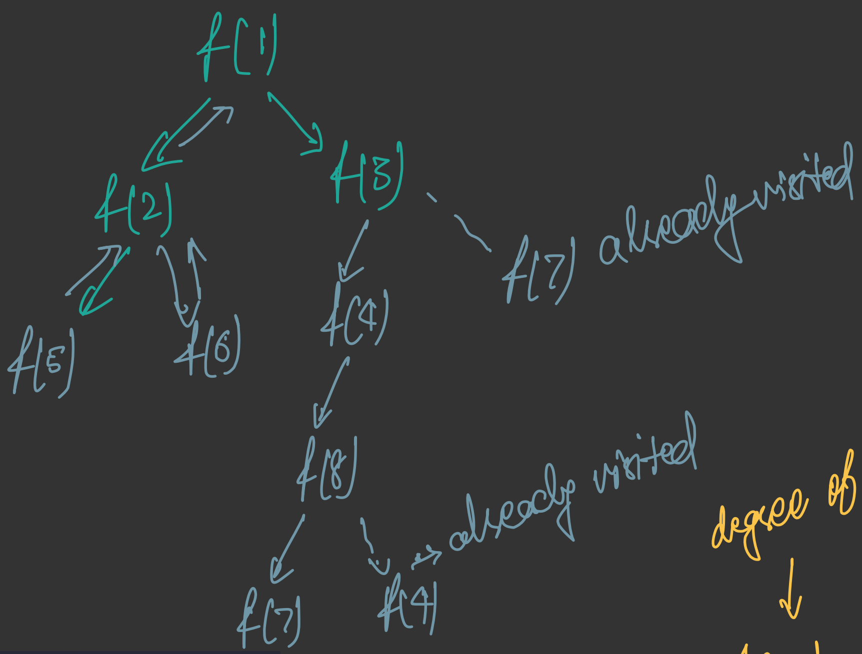
34871256

Recursion

vis =

		1	1	1	1	1	1	1
0	1	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~
0	1	2	3	4	5	6	7	8

starting
↓
f(1)



LIST

1 → {2, 3}

2 → {1, 5, 6}

3 → {1, 4, 7}

4 → {3, 8}

5 → {2}

6 → {2}

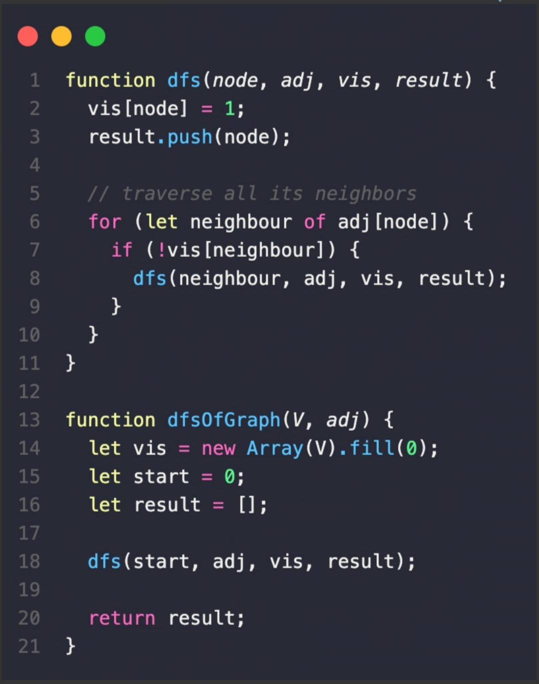
7 → {3, 8}

8 → {4, 7}

degree of each node
↓

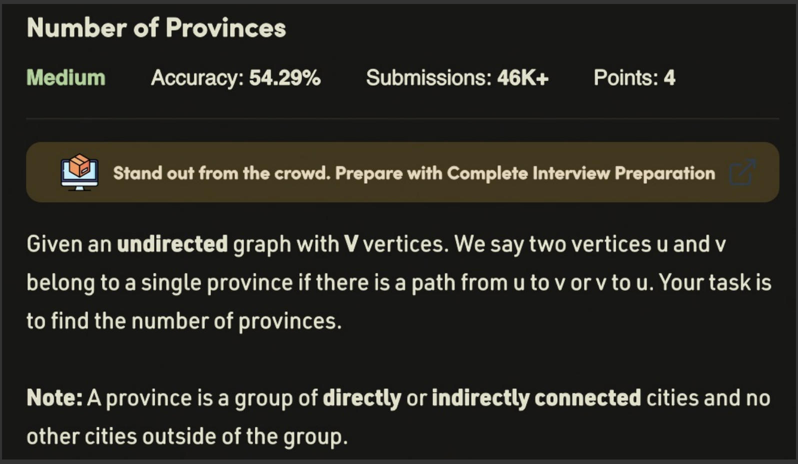
TC → O(N) + O(2E)

SC → O(N) + O(N) + O(N) ≈ O(N)



```
function dfs(node, adj, vis, result) {
  vis[node] = 1;
  result.push(node);

  // traverse all its neighbors
  for (let neighbour of adj[node]) {
    if (!vis[neighbour]) {
      dfs(neighbour, adj, vis, result);
    }
  }
}

function dfsOfGraph(V, adj) {
  let vis = new Array(V).fill(0);
  let start = 0;
  let result = [];

  dfs(start, adj, vis, result);

  return result;
}
```

lec7. number of provinces.



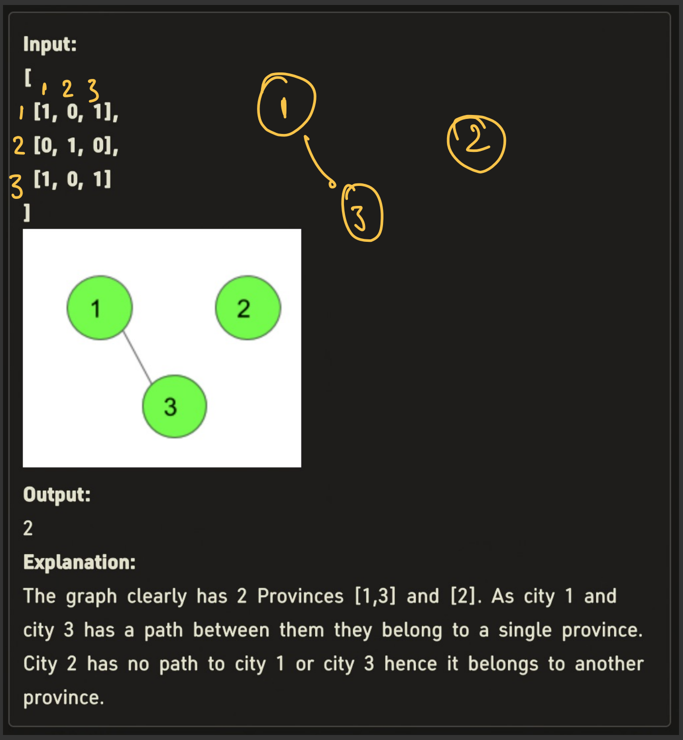
Number of Provinces

Medium Accuracy: **54.29%** Submissions: **46K+** Points: **4**

Stand out from the crowd. Prepare with Complete Interview Preparation

Given an **undirected** graph with **V** vertices. We say two vertices u and v belong to a single province if there is a path from u to v or v to u. Your task is to find the number of provinces.

Note: A province is a group of **directly** or **indirectly connected** cities and no other cities outside of the group.



Input:

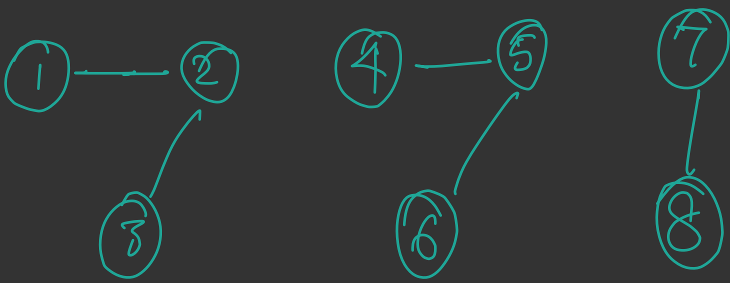
```
[ 1 2 3
1 [1, 0, 1],
2 [0, 1, 0],
3 [1, 0, 1]
]
```

Output:

2

Explanation:

The graph clearly has 2 Provinces [1,3] and [2]. As city 1 and city 3 has a path between them they belong to a single province. City 2 has no path to city 1 or city 3 hence it belongs to another province.



dfs traversal.

starting point = 1/2/3

" " = 4/5/6

= 7/8

dfs() can be called ③ times

ans.

sol^n

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

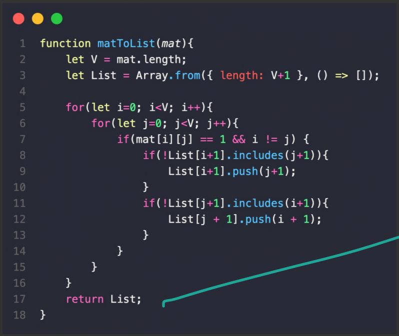
visited =

```
count=0;
for ( i=1; i<=V ; i++){
    if (visited[i]==0)
    {   count++
        dfs(i) || bfs(i)  → dusing this, it will mark the visited
                             node in the visited array.
    }
}
```

$$TC \rightarrow \overset{\text{outter loop}}{O(N)} + \overbrace{O(V+2E)}^{\text{dfs \& neighbours}} \rightarrow O(N)$$

$$\Rightarrow O(N)+O(N) \approx O(N)$$

$$SC \rightarrow O(N)+O(N)$$

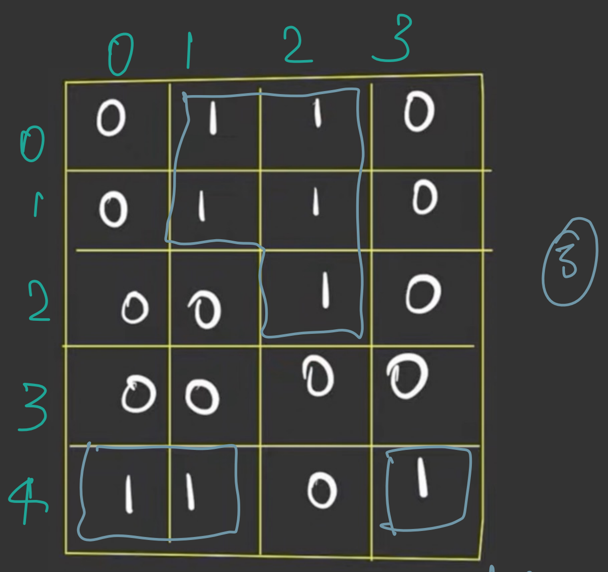


```
function matToList(mat){
    let V = mat.length;
    let List = Array.from({ length: V+1 }, () => []);

    for(let i=0; i<V; i++){
        for(let j=0; j<V; j++){
            if(mat[i][j] == 1 && i != j) {
                if(!List[i+1].includes(j+1)){
                    List[i+1].push(j+1);
                }
                if(!List[j+1].includes(i+1)){
                    List[j + 1].push(i + 1);
                }
            }
        }
    }
    return List;
}
```

[[ ], [3], [ ], [1]]

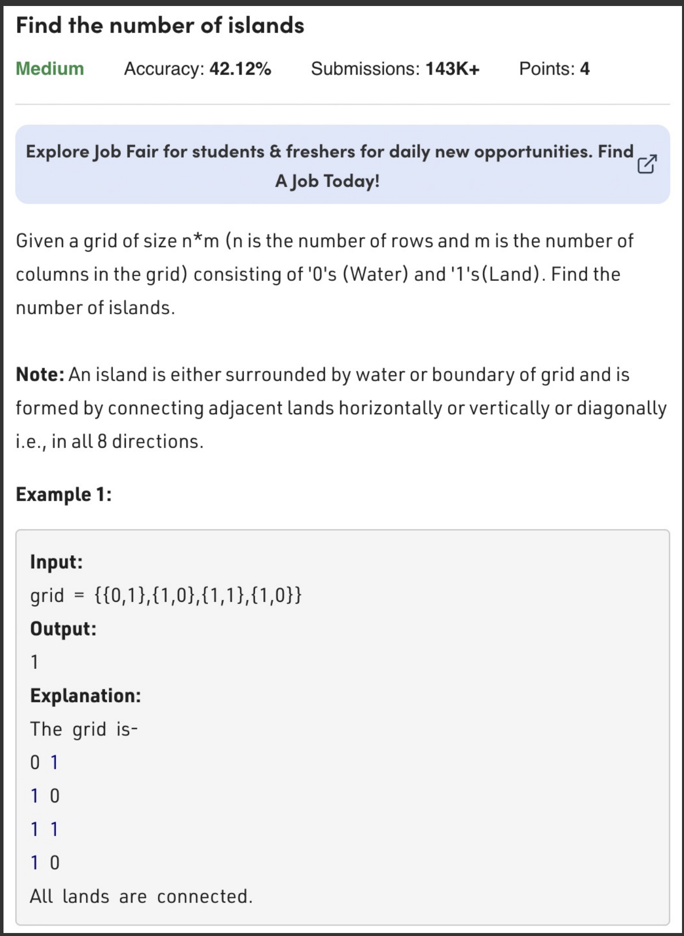
```
const mat = [[1,0,1],[0,1,0],[1,0,1]];
let adjList = matToList(mat);

function dfs(node, List, vis) {
    vis[node] = 1;

    for (let neighbour of List[node]) {
        if (!vis[neighbour]) {
            dfs(neighbour, List, vis);
        }
    }
}

function numOfProvinces(V, adjList){
    let vis = Array(V).fill(0);
    let count = 0;
    for(let i=0; i<V; i++){
        if(!vis[i]){
            count++;
            dfs(i, adjList, vis);
        }
    }
    return count;
}
numOfProvinces(3, adjList); //3
```

lec8. numbers of islands. (connected components)



	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	0	1

find numbers of starting node? → ans



Find the number of islands

Medium Accuracy: **42.12%** Submissions: **143K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a grid of size n*m (n is the number of rows and m is the number of columns in the grid) consisting of '0's (Water) and '1's(Land). Find the number of islands.

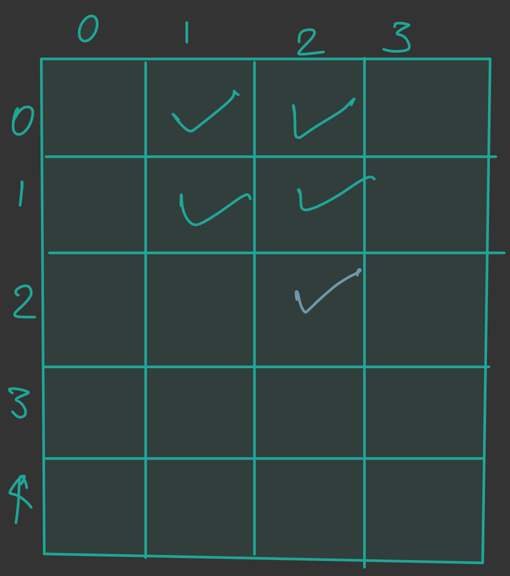
Note: An island is either surrounded by water or boundary of grid and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

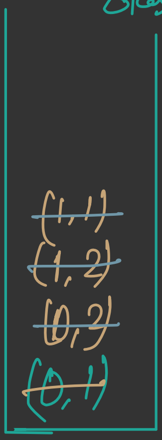
Example 1:

```
Input:
grid = {{0,1},{1,0},{1,1},{1,0}}
Output:
1
Explanation:
The grid is-
0 1
1 0
1 1
1 0
All lands are connected.
```

BFS

creat a visited matrix





Starting point :→ (0,1)

→ start from (0,1)

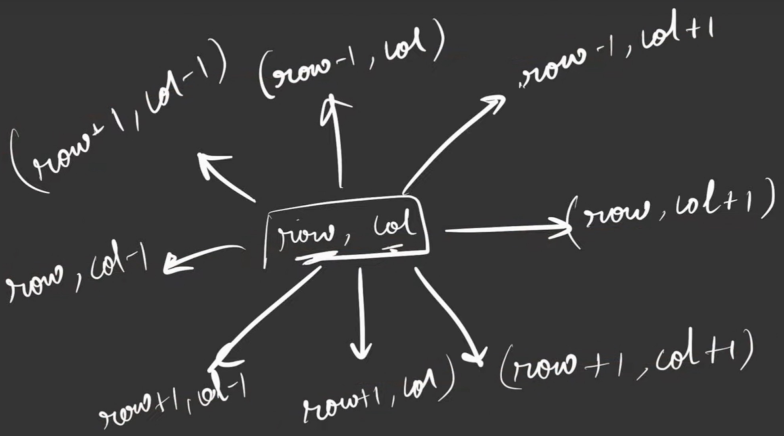
→ traverse it's neighbours

→ push it in queue

→ mark it visited in visited array

traverse the input matrix, if it's 1 & not visited, it is another starting point

```
for (row → 0 to n)
    for (col → 0 to m)
    { if (! vis[row][col])
        { bfs(row, col)
            count ++;
        }
    }
return count;
```



visit neighbours.

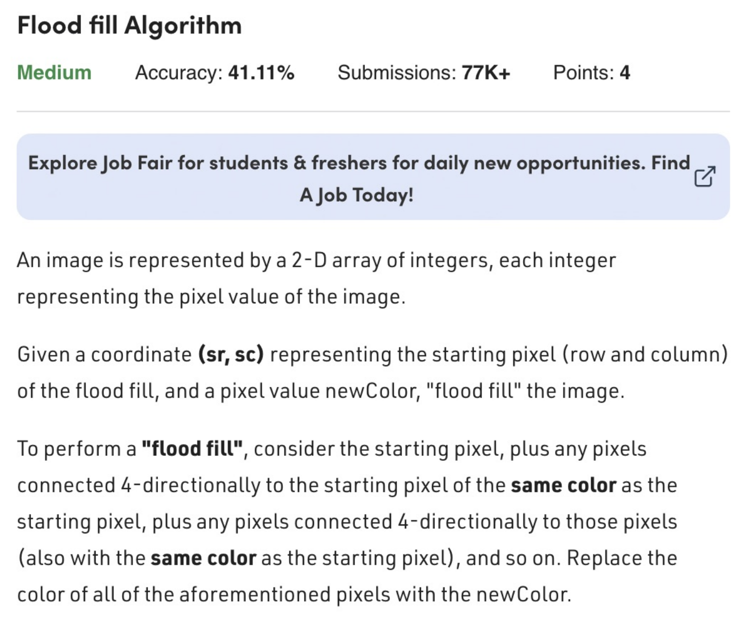
```
for ( deltarow -> -1 to +1)
    for ( deltaCol -> -1 to +1)
    {
        neighboursRow = row + deltarow
        neighboursCol = col + deltaCol
    }
```

```
function bfs(row, col, vis, grid) {
  vis[row][col] = 1;
  let q = [];
  q.push([row, col]);
  while (q.length > 0) {
    let [r, c] = q.shift();
    for (let delRow = -1; delRow <= 1; delRow++) {
      for (let delCol = -1; delCol <= 1; delCol++) {
        let newRow = r + delRow;
        let newCol = c + delCol;
        if (
          newRow >= 0 &&
          newRow < grid.length &&
          newCol >= 0 &&
          newCol < grid[0].length &&
          grid[newRow][newCol] === 1 &&
          vis[newRow][newCol] === 0
        ) {
          q.push([newRow, newCol]);
          vis[newRow][newCol] = 1;
        }
      }
    }
  }
}

function numIslands(grid) {
  let n = grid.length;
  let m = grid[0].length;
  let vis = Array.from({ length: n }, () => Array(m).fill(0));
  let cnt = 0;
  for (let row = 0; row < n; row++) {
    for (let col = 0; col < m; col++) {
      if (grid[row][col] === 1 && vis[row][col] === 0) {
        cnt++;
        bfs(row, col, vis, grid);
      }
    }
  }
  return cnt;
}

const grid = [[0,1,1,1,0,0,0],[0,0,1,1,0,1,0]]
numIslands(grid); //? 2
```

$SC \rightarrow \underset{\uparrow\ \text{visited}}{O(N^2)} + \underset{\uparrow\ \text{queue}}{O(N^2)}$

$TC \rightarrow \underset{\uparrow\ \text{matrix}}{N^2} + \underset{\uparrow\ \text{visited} \times \text{neighbours bfs}}{\underline{N^2 \times 9}} \approx N^2$

lec9 → Flood Fill Algorithm



Flood fill Algorithm

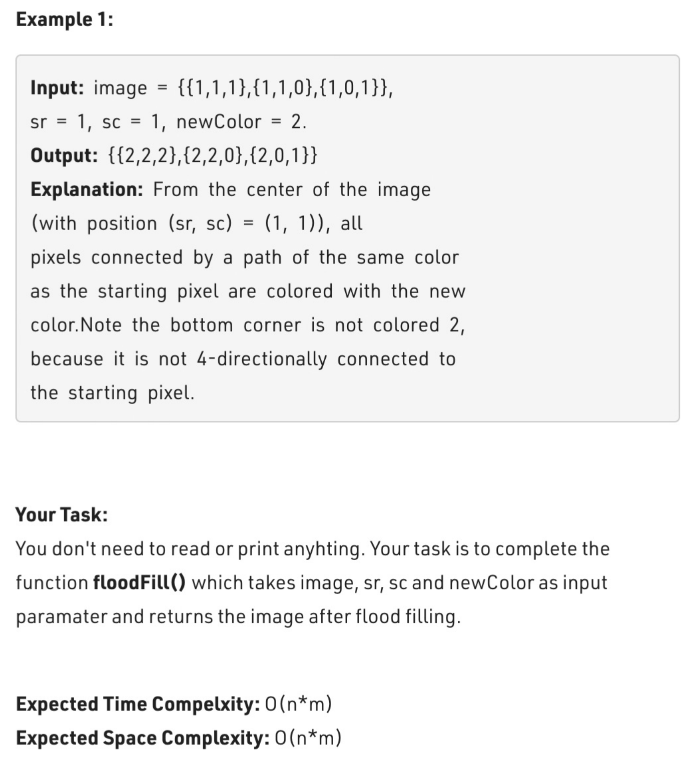
Medium Accuracy: **41.11%** Submissions: **77K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image.

Given a coordinate **(sr, sc)** representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a **"flood fill"**, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the **same color** as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the **same color** as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

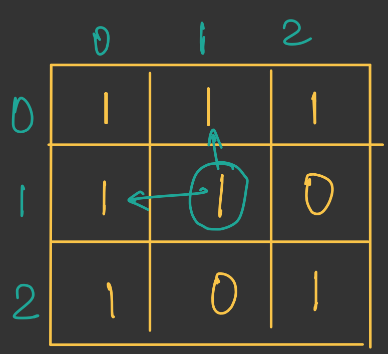


Example 1:

Input: image = {{1,1,1},{1,1,0},{1,0,1}},
sr = 1, sc = 1, newColor = 2.
Output: {{2,2,2},{2,2,0},{2,0,1}}
Explanation: From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected by a path of the same color as the starting pixel are colored with the new color.Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

Your Task:
You don't need to read or print anyhting. Your task is to complete the function **floodFill()** which takes image, sr, sc and newColor as input paramater and returns the image after flood filling.

Expected Time Compelxity: O(n*m)
Expected Space Complexity: O(n*m)



	0	1	2
0	1	1	1
1	1	1	0
2	1	0	1

sr = 1, sc = 1

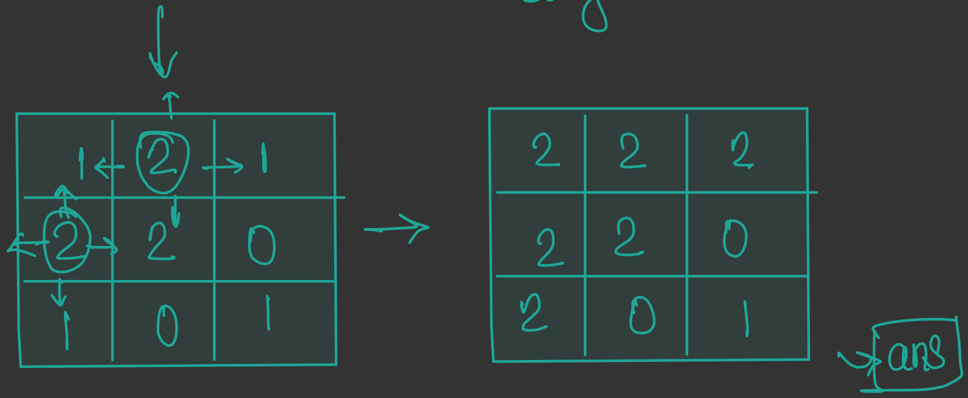
newColor = 2

initially = 1

connected to upper 1 & left 1

right & bottom are 0

diagonal are not allowed



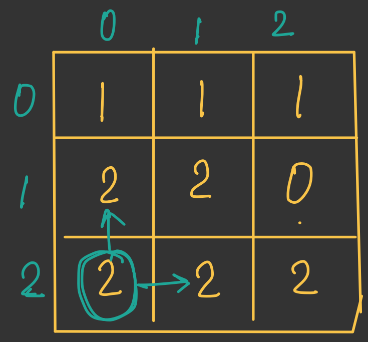
1	2	1
2	2	0
1	0	1

→

2	2	2
2	2	0
2	0	1

→ ans

ex2

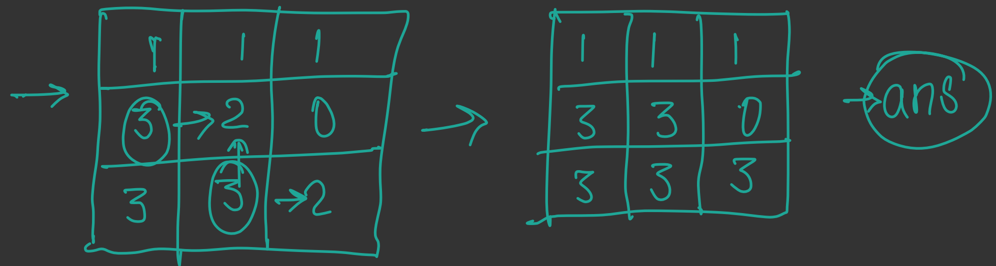


	0	1	2
0	1	1	1
1	2	2	0
2	2	2	2

sr = 2, sc = 0

newColor = 3

initial Color = 2



1	1	1
3	2	0
3	3	2

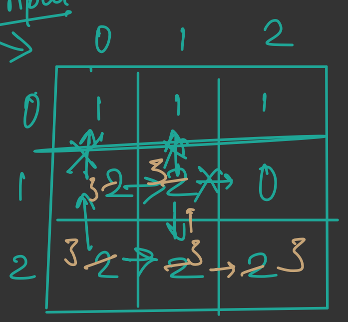
→

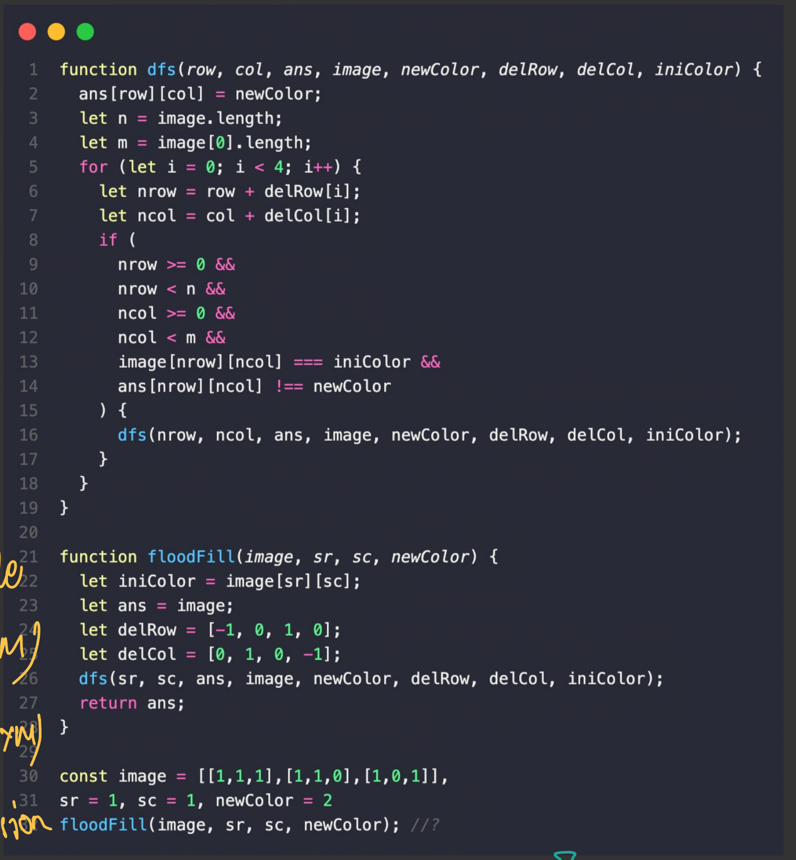
1	1	1
3	3	0
3	3	3

→ ans

sr sc
↑ ↑
dfs(2, 0)

ans => copy of input





```
function dfs(row, col, ans, image, newColor, delRow, delCol, iniColor) {
  ans[row][col] = newColor;
  let n = image.length;
  let m = image[0].length;
  for (let i = 0; i < 4; i++) {
    let nrow = row + delRow[i];
    let ncol = col + delCol[i];
    if (
      nrow >= 0 &&
      nrow < n &&
      ncol >= 0 &&
      ncol < m &&
      image[nrow][ncol] === iniColor &&
      ans[nrow][ncol] !== newColor
    ) {
      dfs(nrow, ncol, ans, image, newColor, delRow, delCol, iniColor);
    }
  }
}

function floodFill(image, sr, sc, newColor) {
  let iniColor = image[sr][sc];
  let ans = image;
  let delRow = [-1, 0, 1, 0];
  let delCol = [0, 1, 0, -1];
  dfs(sr, sc, ans, image, newColor, delRow, delCol, iniColor);
  return ans;
}

const image = [[1,1,1],[1,1,0],[1,0,1]],
sr = 1, sc = 1, newColor = 2
floodFill(image, sr, sc, newColor); //?
```

TC -> N×M ≃ x node

(4x) or O(N×M)

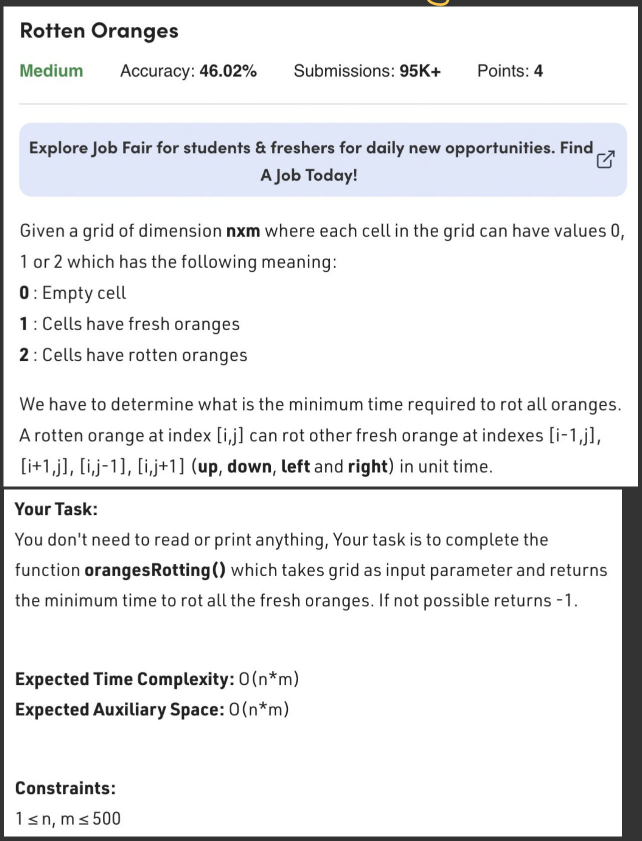
SC -> O(N×M) + O(N×M)

↑

recursion

[[2,2,2],[2,2,0],[2,0,1]]

lec 10. Rotten Oranges



Rotten Oranges

Medium Accuracy: **46.02%** Submissions: **95K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a grid of dimension **nxm** where each cell in the grid can have values 0, 1 or 2 which has the following meaning:

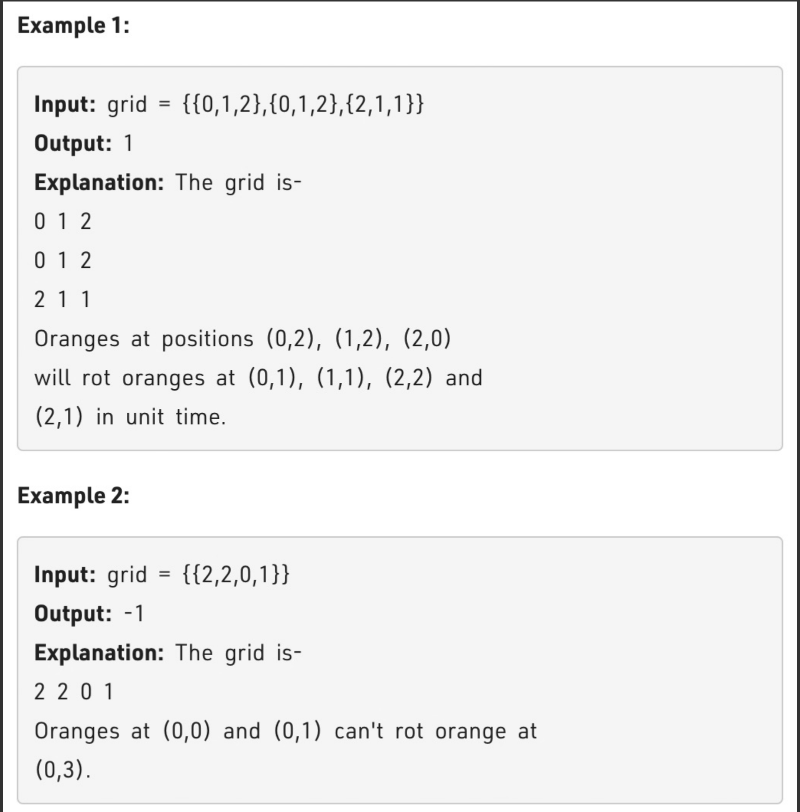
0 : Empty cell
1 : Cells have fresh oranges
2 : Cells have rotten oranges

We have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (**up**, **down**, **left** and **right**) in unit time.

Your Task:
You don't need to read or print anything, Your task is to complete the function **orangesRotting()** which takes grid as input parameter and returns the minimum time to rot all the fresh oranges. If not possible returns -1.

Expected Time Complexity: O(n*m)
Expected Auxiliary Space: O(n*m)

Constraints:
1 ≤ n, m ≤ 500



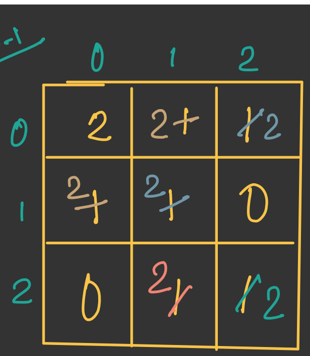
Example 1:

Input: grid = {{0,1,2},{0,1,2},{2,1,1}}
Output: 1
Explanation: The grid is-
0 1 2
0 1 2
2 1 1
Oranges at positions (0,2), (1,2), (2,0) will rot oranges at (0,1), (1,1), (2,2) and (2,1) in unit time.

Example 2:

Input: grid = {{2,2,0,1}}
Output: -1
Explanation: The grid is-
2 2 0 1
Oranges at (0,0) and (0,1) can't rot orange at (0,3).

ex-1



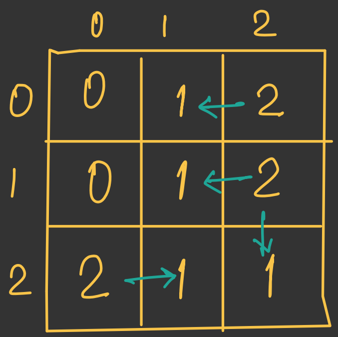
	0	1	2
0	2	2 ~~1~~	~~1~~ 2
1	2 ~~1~~	2 ~~1~~	0
2	0	2 ~~1~~	~~1~~ 2

2 → rot
1 → fresh
0 → doesn't have orange in this cell

t = 1
t = 2
t = 3
t = 4

total min time = 4 unit

ex 2



	0	1	2
0	0	1	← 2
1	0	1	← 2
2	2 →	1	1

if not possible to rot all oranges return -1.

t = 1

ans ⇒ 1

Intuition
each rotten orange is rotting other oranges which are at 1 level distance from it.
level wise → BFS

	0	1	2
0	0	1	2
1	0	1	1
2	2	1	1

input

minTime → keep track of max time

→ in queue, put coordinates of all initially rotten items with time 0

(2,0), 0

(0,2), 0

→ shift from queue

visit it

& add it's neighbours

to the queue

	0	1	2
0		2	2
1		2	2
2	2	2	2

visited

(2,2), 2

(1,1), 2

~~(0,1), 1~~

~~(1,2), 1~~

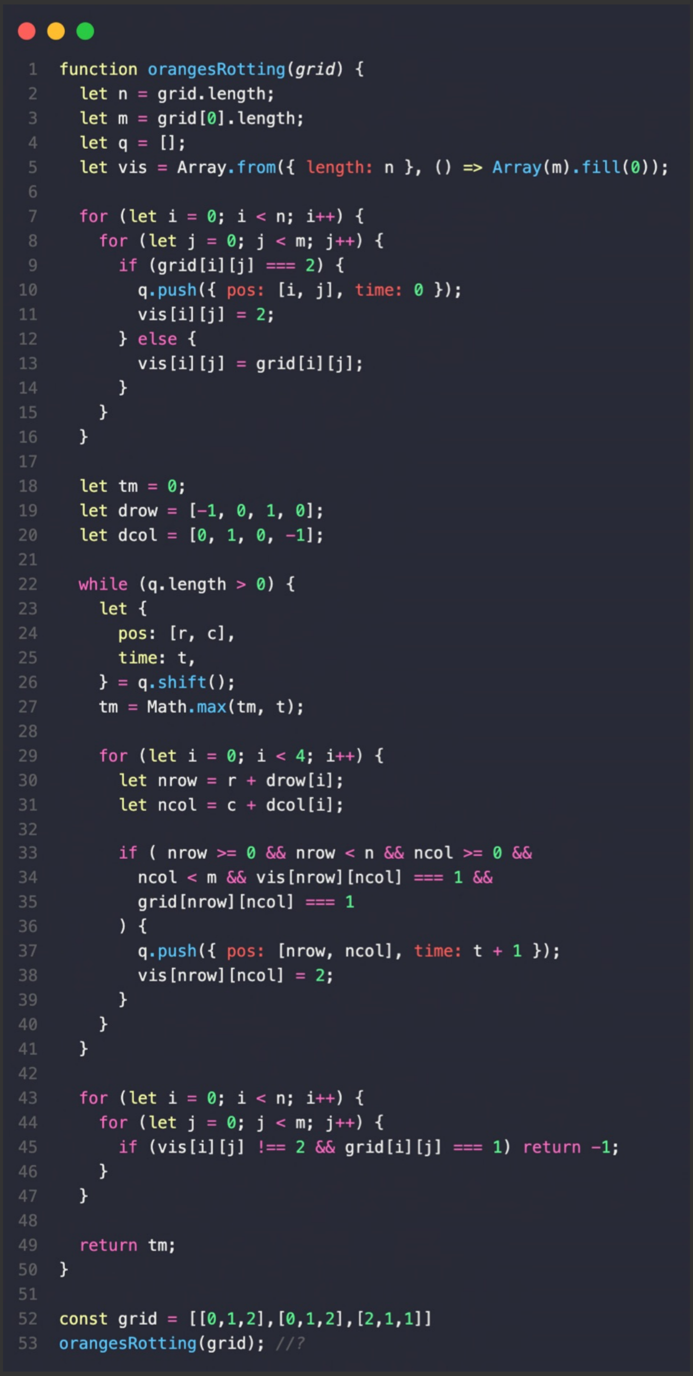
~~(2,1), 1~~

~~(0,2), 0~~

~~(2,0), 0~~

queue

ans → 2



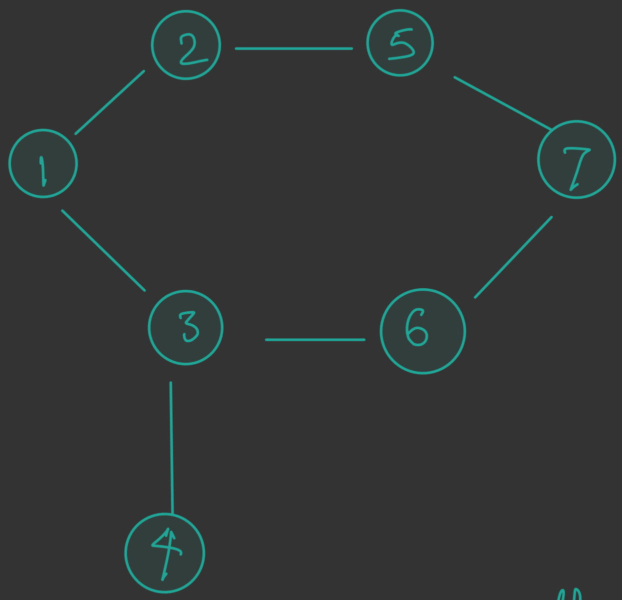
```
function orangesRotting(grid) {
  let n = grid.length;
  let m = grid[0].length;
  let q = [];
  let vis = Array.from({ length: n }, () => Array(m).fill(0));

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (grid[i][j] === 2) {
        q.push({ pos: [i, j], time: 0 });
        vis[i][j] = 2;
      } else {
        vis[i][j] = grid[i][j];
      }
    }
  }

  let tm = 0;
  let drow = [-1, 0, 1, 0];
  let dcol = [0, 1, 0, -1];

  while (q.length > 0) {
    let {
      pos: [r, c],
      time: t,
    } = q.shift();
    tm = Math.max(tm, t);

    for (let i = 0; i < 4; i++) {
      let nrow = r + drow[i];
      let ncol = c + dcol[i];

      if ( nrow >= 0 && nrow < n && ncol >= 0 &&
        ncol < m && vis[nrow][ncol] === 1 &&
        grid[nrow][ncol] === 1
      ) {
        q.push({ pos: [nrow, ncol], time: t + 1 });
        vis[nrow][ncol] = 2;
      }
    }
  }

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (vis[i][j] !== 2 && grid[i][j] === 1) return -1;
    }
  }

  return tm;
}

const grid = [[0,1,2],[0,1,2],[2,1,1]]
orangesRotting(grid); //?
```

✓ neighbours

TC → O(N×M) × 4

SC → O(N×M)

lec11 detect a cycle in an undirected graph using BFS



adj List

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

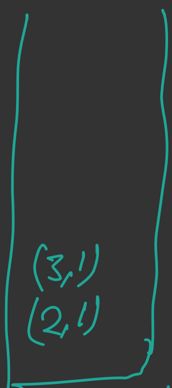
intuition → start in two different direction, if reach / collide at a same node later on, that means it has a cycle

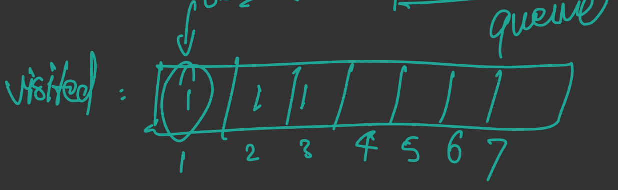
→ push starting in queue

→ go in direction so that 3 won't go to 1 again because it just came from it

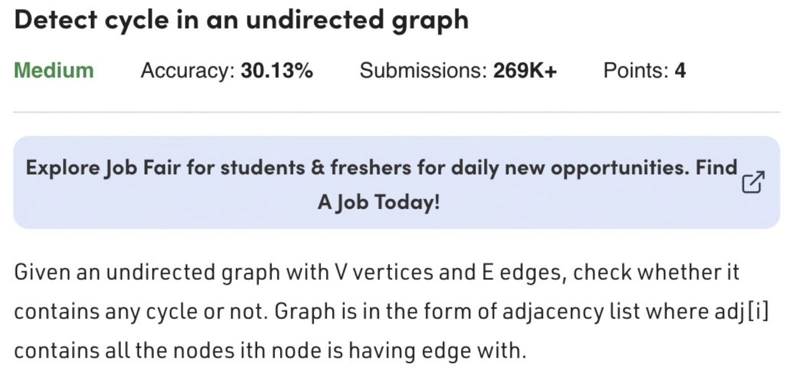
3 1 → which node 3 came frome
↑
next node

& mark it visited in visited





→ if something is already visited, then there is a cycle in it.

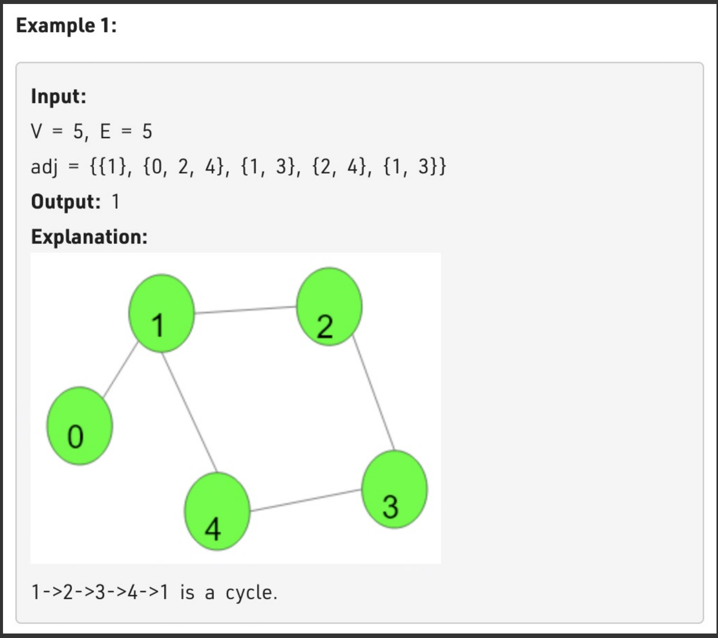


Detect cycle in an undirected graph

Medium Accuracy: **30.13%** Submissions: **269K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not. Graph is in the form of adjacency list where adj[i] contains all the nodes ith node is having edge with.

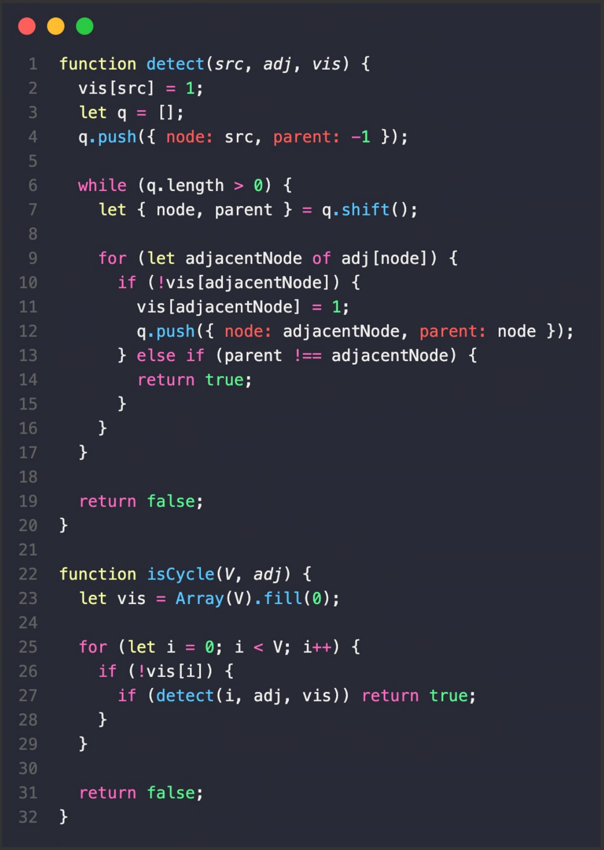


Example 1:

Input:
```
V = 5, E = 5
adj = {{1}, {0, 2, 4}, {1, 3}, {2, 4}, {1, 3}}
```
Output: 1

Explanation:

1->2->3->4->1 is a cycle.



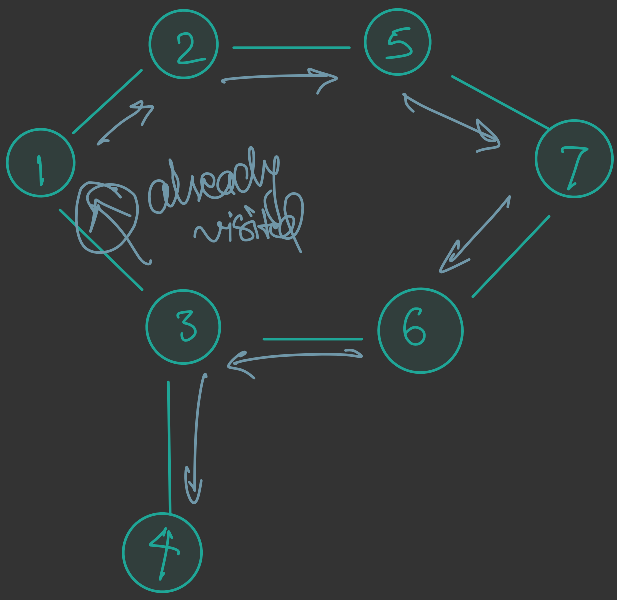
```
function detect(src, adj, vis) {
  vis[src] = 1;
  let q = [];
  q.push({ node: src, parent: -1 });

  while (q.length > 0) {
    let { node, parent } = q.shift();

    for (let adjacentNode of adj[node]) {
      if (!vis[adjacentNode]) {
        vis[adjacentNode] = 1;
        q.push({ node: adjacentNode, parent: node });
      } else if (parent !== adjacentNode) {
        return true;
      }
    }
  }

  return false;
}

function isCycle(V, adj) {
  let vis = Array(V).fill(0);

  for (let i = 0; i < V; i++) {
    if (!vis[i]) {
      if (detect(i, adj, vis)) return true;
    }
  }

  return false;
}
```

TC → $O(N + 2E)$

SC → $O(N) + O(N) \approx O(N)$

---

lec12 → detect a cycle in undirected graph using dfs



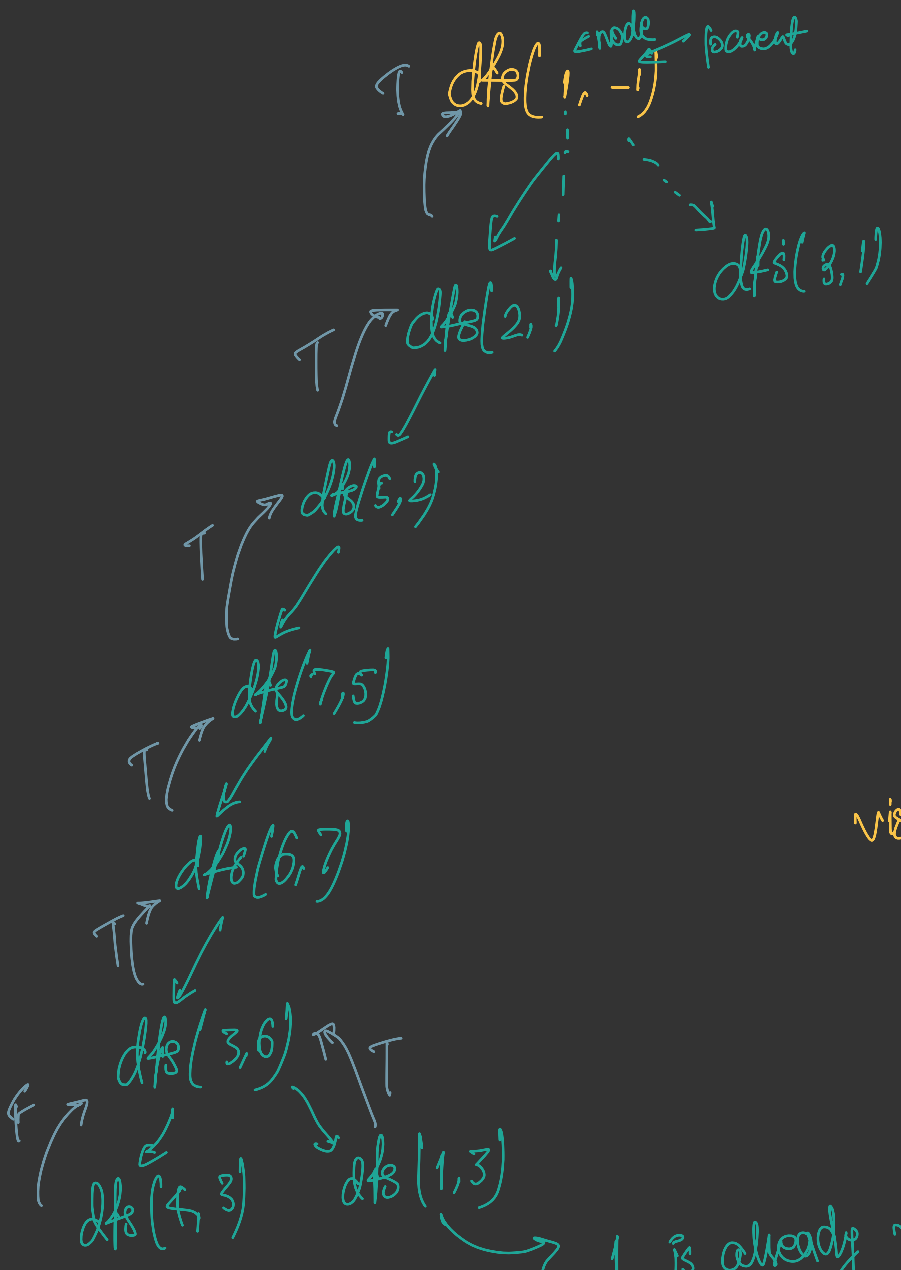
adj List

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

intuition → start dfs from a node, if we encounter any node which has already been visited then there is cycle present in graph

visited =

0	0	0	0	0	0	0
1	2	3	4	5	6	7



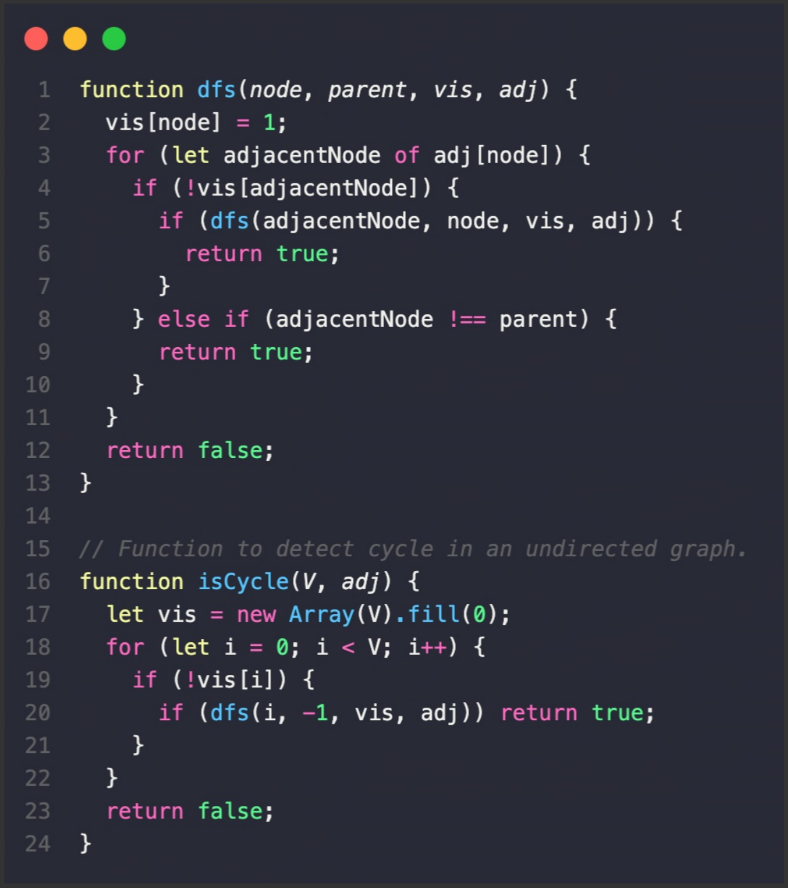
visited =

1	1	1	1	1	1	1
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~
1	2	3	4	5	6	7

3 → 1, 4, 6

1 is already visited & it is not parent of 3, so cycle = true

```
dfs( node, parent){
   vis[node] = 1
   for ( let it of adj[node])
   {  if (vis[it] ==0){
         if (dfs(it, node) == true)
             return true;
      }
      else if ( it != parent)
             return true;
   }
   return false;
}
```



```
function dfs(node, parent, vis, adj) {
  vis[node] = 1;
  for (let adjacentNode of adj[node]) {
    if (!vis[adjacentNode]) {
      if (dfs(adjacentNode, node, vis, adj)) {
        return true;
      }
    } else if (adjacentNode !== parent) {
      return true;
    }
  }
  return false;
}

// Function to detect cycle in an undirected graph.
function isCycle(V, adj) {
  let vis = new Array(V).fill(0);
  for (let i = 0; i < V; i++) {
    if (!vis[i]) {
      if (dfs(i, -1, vis, adj)) return true;
    }
  }
  return false;
}
```

$SC \rightarrow O(N) + O(N)$

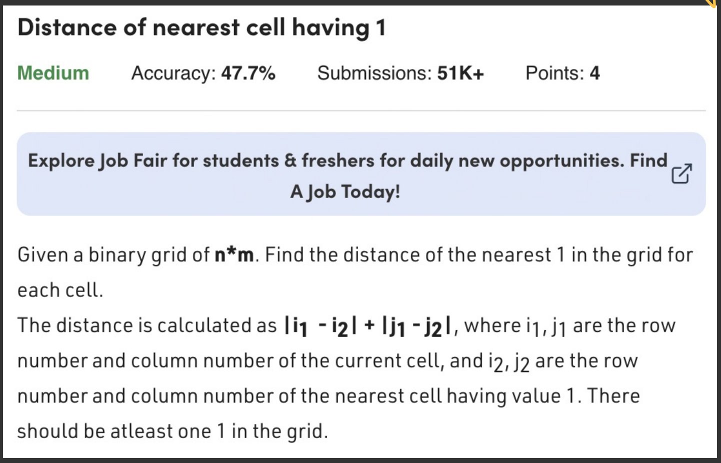
↑ recursion ↑ visited array

$TC \rightarrow O(N+2E) + O(N)$

↑ for loop

---

lec 13. distance of nearest cell having 1 | 0/1 matrix



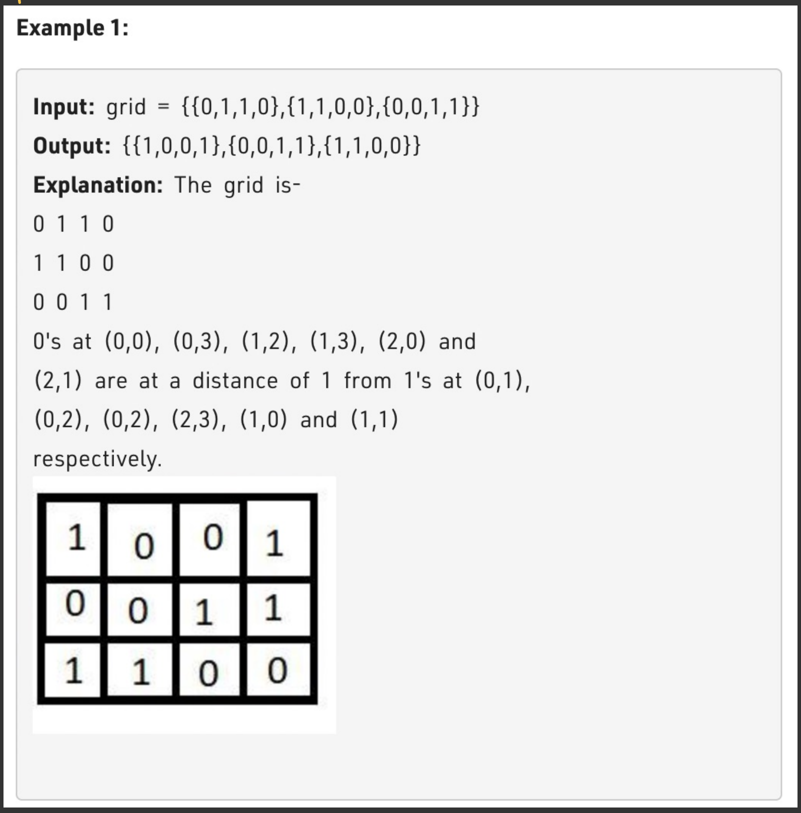
Distance of nearest cell having 1

Medium Accuracy: **47.7%** Submissions: **51K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a binary grid of **n*m**. Find the distance of the nearest 1 in the grid for each cell.

The distance is calculated as **$|i_1 - i_2| + |j_1 - j_2|$**, where i_1, j_1 are the row number and column number of the current cell, and i_2, j_2 are the row number and column number of the nearest cell having value 1. There should be atleast one 1 in the grid.



Example 1:

Input: grid = {{0,1,1,0},{1,1,0,0},{0,0,1,1}}

Output: {{1,0,0,1},{0,0,1,1},{1,1,0,0}}

Explanation: The grid is-

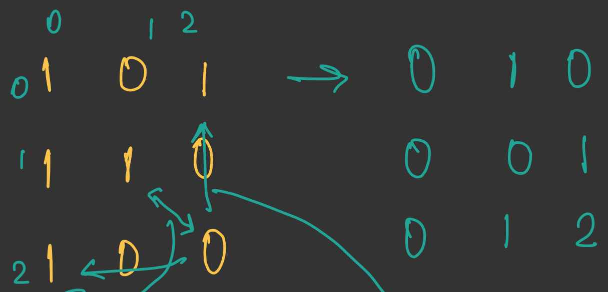
0 1 1 0

1 1 0 0

0 0 1 1

0's at (0,0), (0,3), (1,2), (1,3), (2,0) and (2,1) are at a distance of 1 from 1's at (0,1), (0,2), (0,2), (2,3), (1,0) and (1,1) respectively.

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



$|2-1| + |2-1| = 2$

$|2-2| + |2-0| = 2$

$|2-0| + |2-2| = 2$

min(all three) = 2

intuition

① starts with all given 1
step = 0 (distance = 0)

② traverse one level
distance = 1

③ traverse two level
distance = 2

BFS

0	0	0
0	1	0
1	0	1

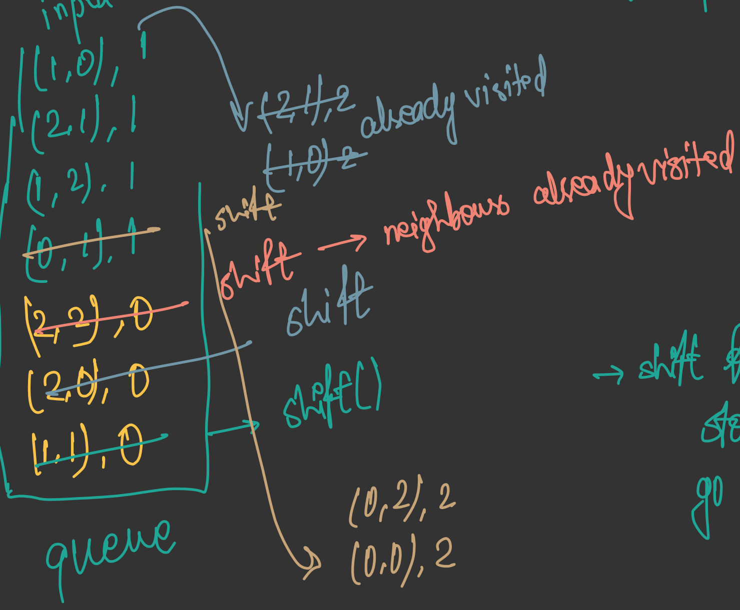
input

	0	1	2
0	~~0~~ 1	~~0~~ 1	~~0~~ 1
1	~~0~~ 1	1	~~0~~ 1
2	1	~~0~~ 1	1

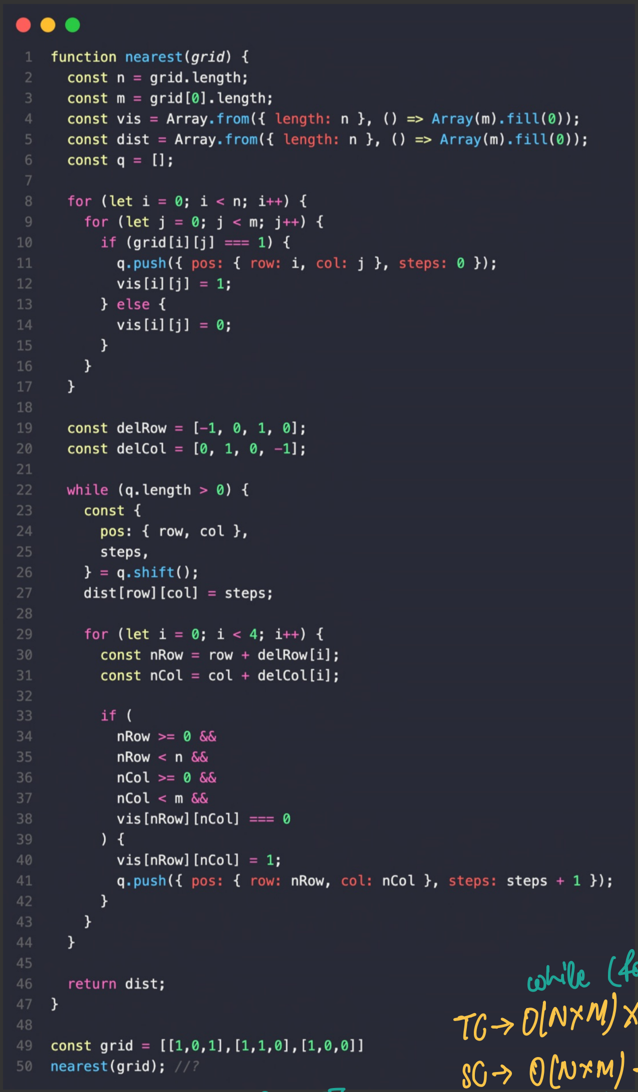
visited

	0	1	2
2		1	
1		0	
0	0		0

distance → ans



→ shift from queue
store the distance in ans matrix
go to it's neighbours
put it in queue if not visited
mark it visited



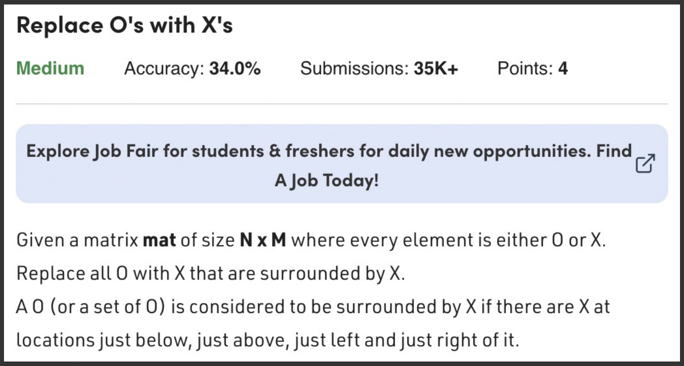
```javascript
function nearest(grid) {
  const n = grid.length;
  const m = grid[0].length;
  const vis = Array.from({ length: n }, () => Array(m).fill(0));
  const dist = Array.from({ length: n }, () => Array(m).fill(0));
  const q = [];

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (grid[i][j] === 1) {
        q.push({ pos: { row: i, col: j }, steps: 0 });
        vis[i][j] = 1;
      } else {
        vis[i][j] = 0;
      }
    }
  }

  const delRow = [-1, 0, 1, 0];
  const delCol = [0, 1, 0, -1];

  while (q.length > 0) {
    const {
      pos: { row, col },
      steps,
    } = q.shift();
    dist[row][col] = steps;

    for (let i = 0; i < 4; i++) {
      const nRow = row + delRow[i];
      const nCol = col + delCol[i];

      if (
        nRow >= 0 &&
        nRow < n &&
        nCol >= 0 &&
        nCol < m &&
        vis[nRow][nCol] === 0
      ) {
        vis[nRow][nCol] = 1;
        q.push({ pos: { row: nRow, col: nCol }, steps: steps + 1 });
      }
    }
  }

  return dist;
}

const grid = [[1,0,1],[1,1,0],[1,0,0]]
nearest(grid); //?
```

[[0,1,0], [0,0,1], [0,1,2]]

$$TC \rightarrow \overset{\text{while (for)}}{O(N\times M)} \times 4 + \overset{\text{grid}}{O(N\times M)} \approx O(N\times M)$$

$$SC \rightarrow \underset{\text{visited}}{O(N\times M)} + \underset{\text{distance}}{O(N\times M)}$$

lec 14 -> surrounded regions.

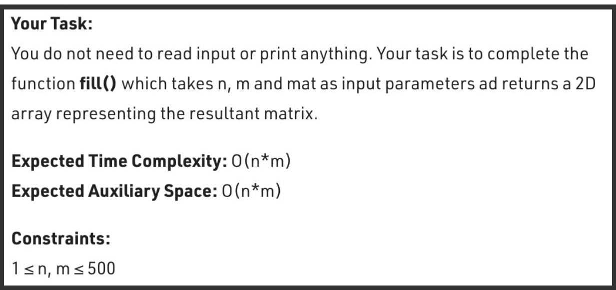


Replace O's with X's

Medium Accuracy: **34.0%** Submissions: **35K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

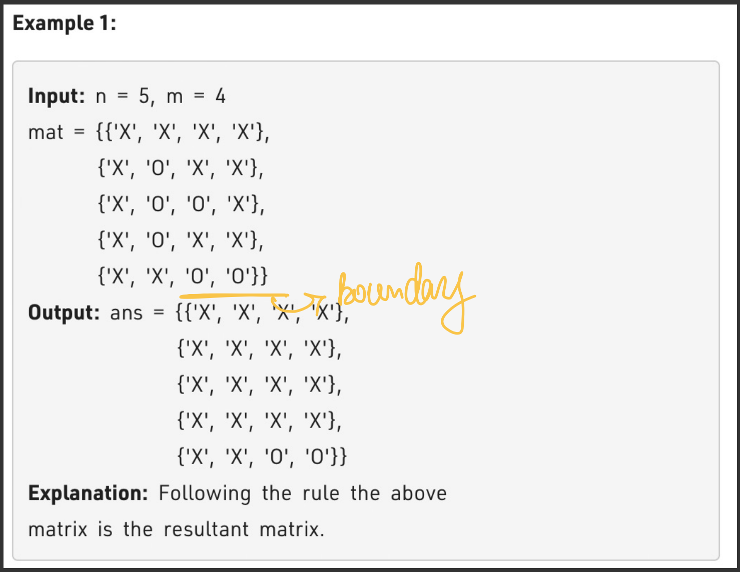
Given a matrix **mat** of size **N x M** where every element is either O or X.
Replace all O with X that are surrounded by X.
A O (or a set of O) is considered to be surrounded by X if there are X at locations just below, just above, just left and just right of it.



Your Task:
You do not need to read input or print anything. Your task is to complete the function **fill()** which takes n, m and mat as input parameters ad returns a 2D array representing the resultant matrix.

Expected Time Complexity: O(n*m)
Expected Auxiliary Space: O(n*m)

Constraints:
1 ≤ n, m ≤ 500



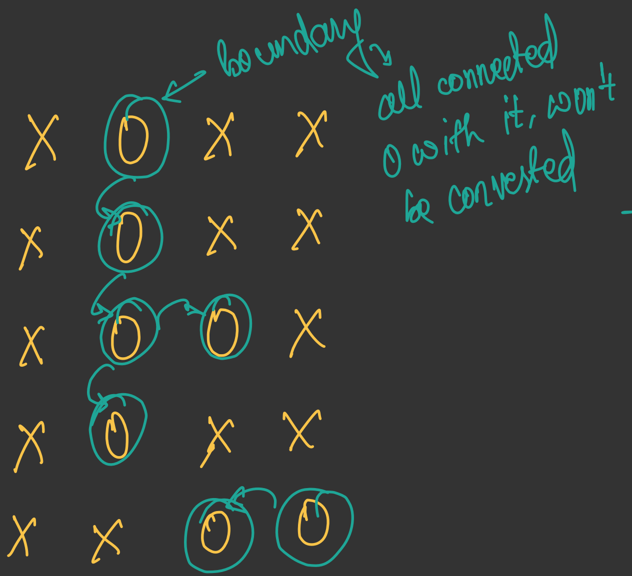
Example 1:

```
Input: n = 5, m = 4
mat = {{'X', 'X', 'X', 'X'},
       {'X', 'O', 'X', 'X'},
       {'X', 'O', 'O', 'X'},
       {'X', 'O', 'X', 'X'},
       {'X', 'X', 'O', 'O'}}
Output: ans = {{'X', 'X', 'X', 'X'},
               {'X', 'X', 'X', 'X'},
               {'X', 'X', 'X', 'X'},
               {'X', 'X', 'X', 'X'},
               {'X', 'X', 'O', 'O'}}
Explanation: Following the rule the above
matrix is the resultant matrix.
```

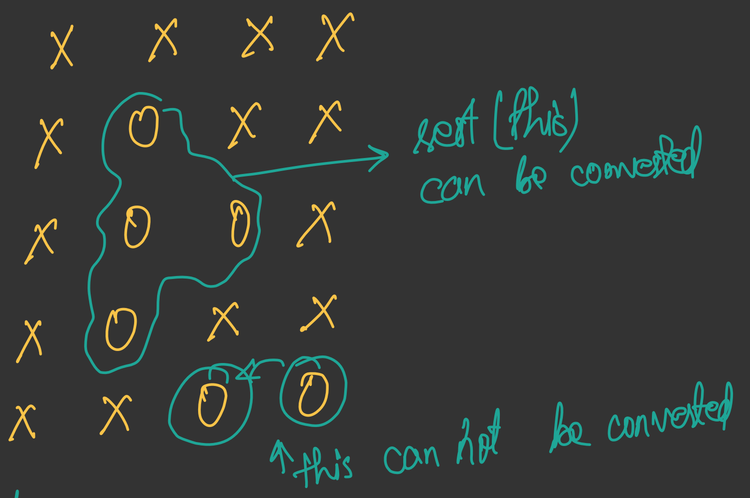
-> boundary

observations.

① if a set of X' connected to the boundary O, it won't be converted



if



intuition

start from the boundary O & mark those that will not be converted and convert the remaining.

	0	1	2	3	4
0	X	X	X	X	X
1	X	O	O	X	O
2	X	X	O	X	O
3	X	O	X	O	X
4	O	O	X	X	X

vis:

0	0	0	0	0
0	0	0	0	~~0~~ 1
0	0	0	0	~~0~~ 1
0	~~0~~ 1	0	0	0
~~0~~ 1	~~0~~ 1	0	0	0

top   no 0

bottom → done

left

0 → (4, 0)

dfs(4, 0)

dfs(4, 1)

dfs(3, 1)

right → 0 → (1, 4)

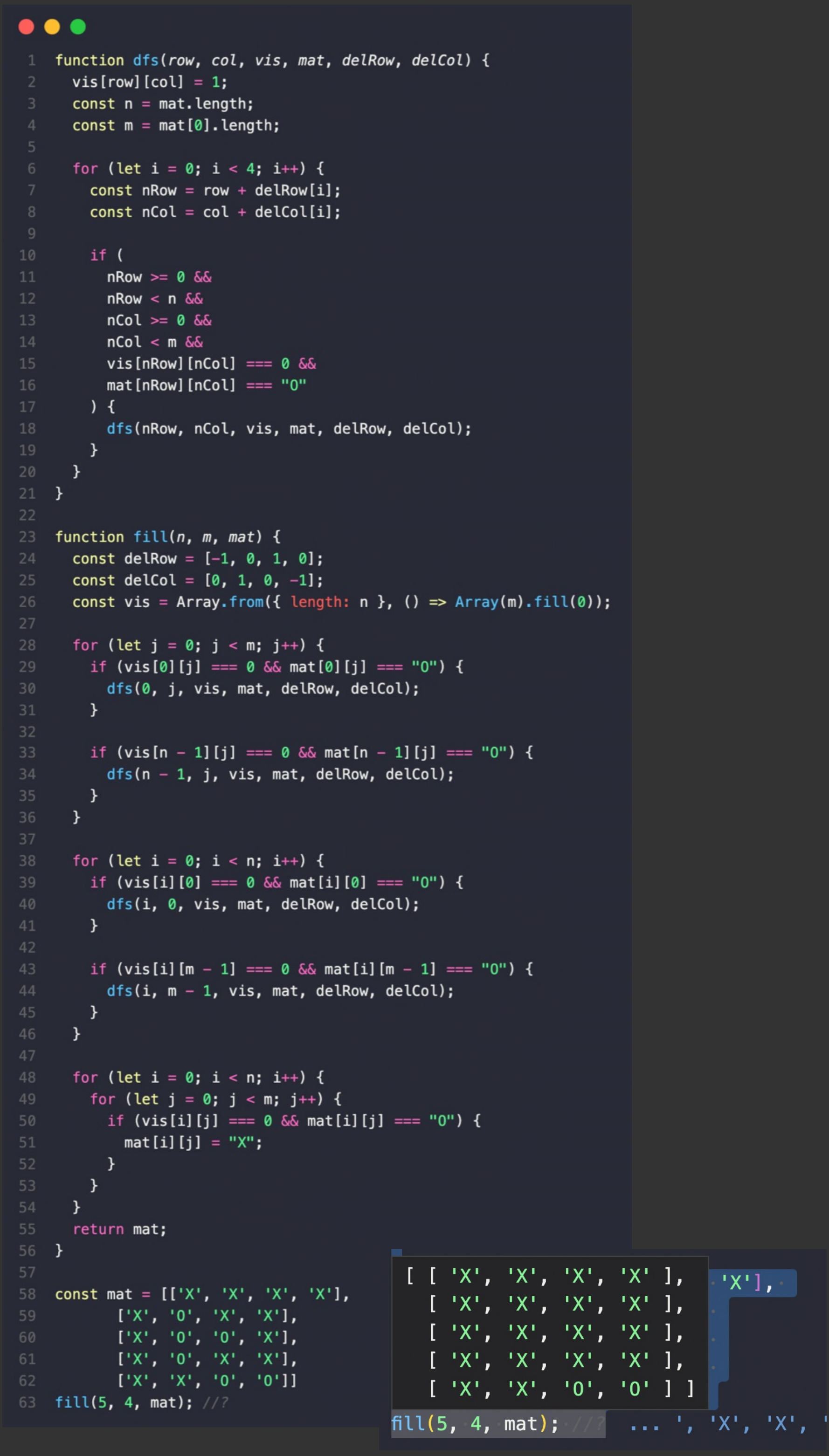
dfs(1, 4)

dfs(2, 4)

$TC \rightarrow O(N \times M) \times 4 + O(N) + O(N)$

$\approx O(N \times M)$

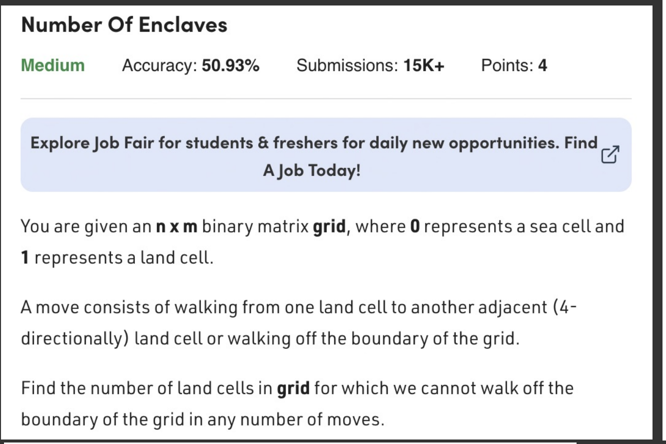
$SC \rightarrow O(N \times M)$



```javascript
function dfs(row, col, vis, mat, delRow, delCol) {
  vis[row][col] = 1;
  const n = mat.length;
  const m = mat[0].length;

  for (let i = 0; i < 4; i++) {
    const nRow = row + delRow[i];
    const nCol = col + delCol[i];

    if (
      nRow >= 0 &&
      nRow < n &&
      nCol >= 0 &&
      nCol < m &&
      vis[nRow][nCol] === 0 &&
      mat[nRow][nCol] === "O"
    ) {
      dfs(nRow, nCol, vis, mat, delRow, delCol);
    }
  }
}

function fill(n, m, mat) {
  const delRow = [-1, 0, 1, 0];
  const delCol = [0, 1, 0, -1];
  const vis = Array.from({ length: n }, () => Array(m).fill(0));

  for (let j = 0; j < m; j++) {
    if (vis[0][j] === 0 && mat[0][j] === "O") {
      dfs(0, j, vis, mat, delRow, delCol);
    }

    if (vis[n - 1][j] === 0 && mat[n - 1][j] === "O") {
      dfs(n - 1, j, vis, mat, delRow, delCol);
    }
  }

  for (let i = 0; i < n; i++) {
    if (vis[i][0] === 0 && mat[i][0] === "O") {
      dfs(i, 0, vis, mat, delRow, delCol);
    }

    if (vis[i][m - 1] === 0 && mat[i][m - 1] === "O") {
      dfs(i, m - 1, vis, mat, delRow, delCol);
    }
  }

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (vis[i][j] === 0 && mat[i][j] === "O") {
        mat[i][j] = "X";
      }
    }
  }
  return mat;
}

const mat = [['X', 'X', 'X', 'X'],
        ['X', 'O', 'X', 'X'],
        ['X', 'O', 'O', 'X'],
        ['X', 'O', 'X', 'X'],
        ['X', 'X', 'O', 'O']]
fill(5, 4, mat); //?
```

```
[ [ 'X', 'X', 'X', 'X' ],
  [ 'X', 'X', 'X', 'X' ],
  [ 'X', 'X', 'X', 'X' ],
  [ 'X', 'X', 'X', 'X' ],
  [ 'X', 'X', 'O', 'O' ] ]
fill(5, 4, mat); //?  ... ', 'X', 'X', '
```

lec15 number of enclaves.



Number Of Enclaves

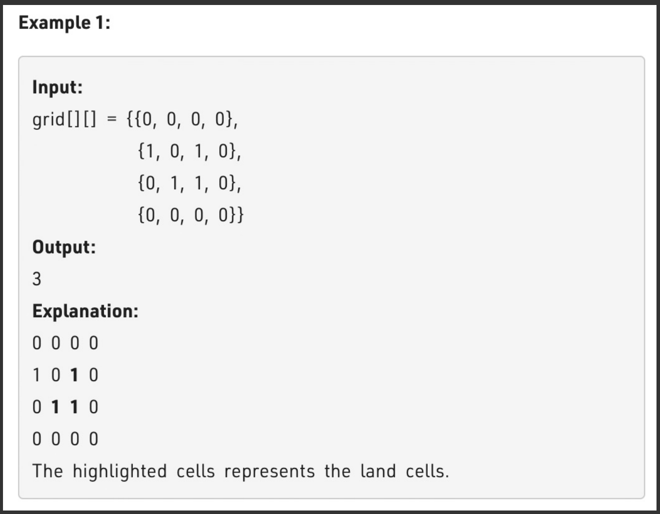
Medium Accuracy: **50.93%** Submissions: **15K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

You are given an **n x m** binary matrix **grid**, where **0** represents a sea cell and **1** represents a land cell.

A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid.

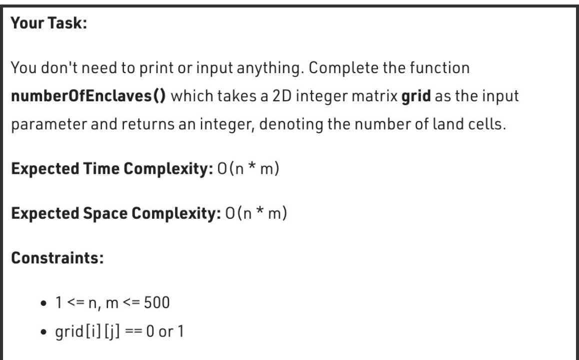
Find the number of land cells in **grid** for which we cannot walk off the boundary of the grid in any number of moves.



Example 1:

Input:
```
grid[][] = {{0, 0, 0, 0},
            {1, 0, 1, 0},
            {0, 1, 1, 0},
            {0, 0, 0, 0}}
```
Output:
3

Explanation:
```
0 0 0 0
1 0 1 0
0 1 1 0
0 0 0 0
```
The highlighted cells represents the land cells.



Your Task:

You don't need to print or input anything. Complete the function **numberOfEnclaves()** which takes a 2D integer matrix **grid** as the input parameter and returns an integer, denoting the number of land cells.

Expected Time Complexity: O(n * m)

Expected Space Complexity: O(n * m)

Constraints:

- 1 <= n, m <= 500
- grid[i][j] == 0 or 1

count num of 1's that can not go out of boundary

0	0	~~1~~ 0	1
0	1	1	0
0	1	1	0
0	0	0	1
0	1	1	0

→ ans => 0

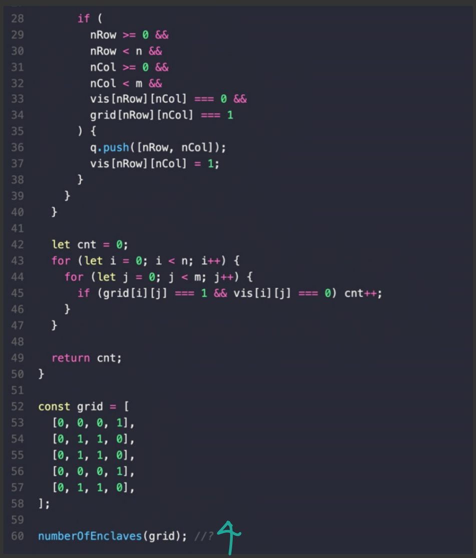
→ ans => 4

→ find all 1 at boundary & other 1's connected with boundary 1 won't be the answer
remaining 1's will be the answer

BFS

queue, visited matrix

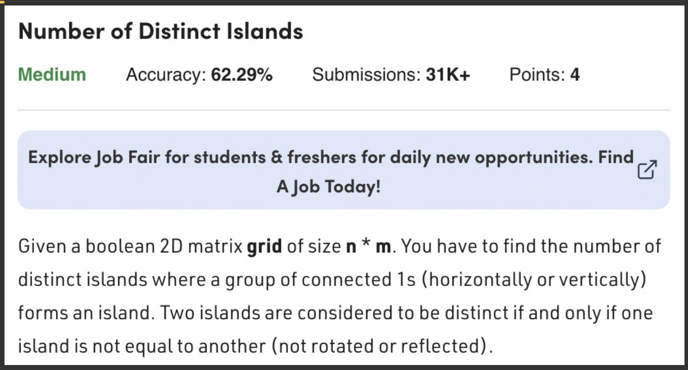
```
function numberOfEnclaves(grid) {
  const q = [];
  const n = grid.length;
  const m = grid[0].length;
  const vis = Array.from({ length: n }, () => Array(m).fill(0));

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (i === 0 || j === 0 || i === n - 1 || j === m - 1) {
        if (grid[i][j] === 1) {
          q.push([i, j]);
          vis[i][j] = 1;
        }
      }
    }
  }

  const delRow = [-1, 0, 1, 0];
  const delCol = [0, 1, 0, -1];

  while (q.length > 0) {
    const [row, col] = q.shift();

    for (let i = 0; i < 4; i++) {
      const nRow = row + delRow[i];
      const nCol = col + delCol[i];

```



```
      if (
        nRow >= 0 &&
        nRow < n &&
        nCol >= 0 &&
        nCol < m &&
        vis[nRow][nCol] === 0 &&
        grid[nRow][nCol] === 1
      ) {
        q.push([nRow, nCol]);
        vis[nRow][nCol] = 1;
      }
    }
  }

  let cnt = 0;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (grid[i][j] === 1 && vis[i][j] === 0) cnt++;
    }
  }

  return cnt;
}

const grid = [
  [0, 0, 0, 1],
  [0, 1, 1, 0],
  [0, 1, 1, 0],
  [0, 0, 0, 1],
  [0, 1, 1, 0],
];

numberOfEnclaves(grid); //?
```

4

lec 16 numbers of distinct islands.

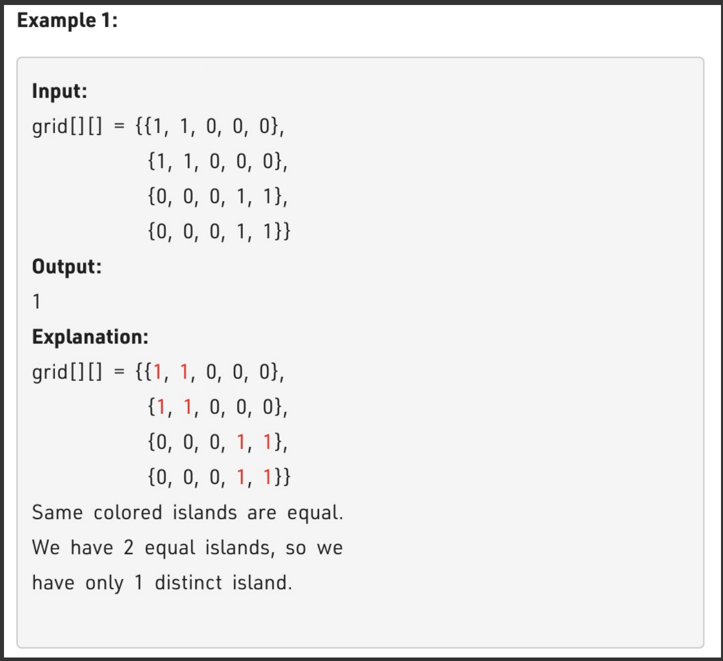


Number of Distinct Islands

Medium Accuracy: **62.29%** Submissions: **31K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a boolean 2D matrix **grid** of size **n** * **m**. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is not equal to another (not rotated or reflected).



Example 1:

Input:

```
grid[][] = {{1, 1, 0, 0, 0},
            {1, 1, 0, 0, 0},
            {0, 0, 0, 1, 1},
            {0, 0, 0, 1, 1}}
```

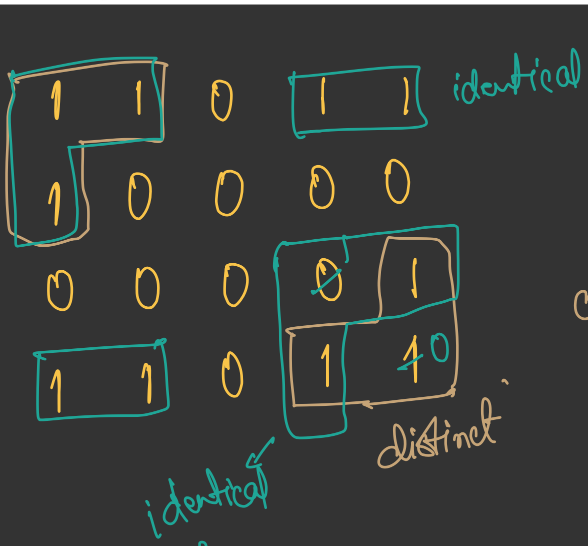
Output:

1

Explanation:

```
grid[][] = {{1, 1, 0, 0, 0},
            {1, 1, 0, 0, 0},
            {0, 0, 0, 1, 1},
            {0, 0, 0, 1, 1}}
```

Same colored islands are equal.
We have 2 equal islands, so we have only 1 distinct island.

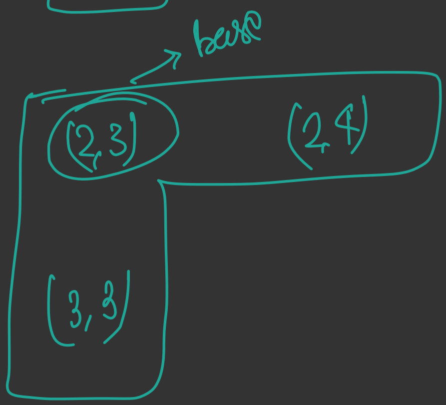


ans = 2

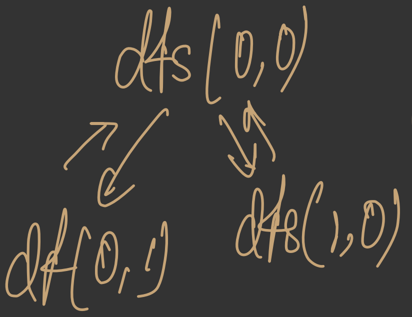
ans => 3 → M1 => store all the shape in set and return the size of set

=> (0,0) - (0,0) => (0,0)
(0,1) - (0,0) => (0,1)
(1,0) - (0,0) => (1,0)

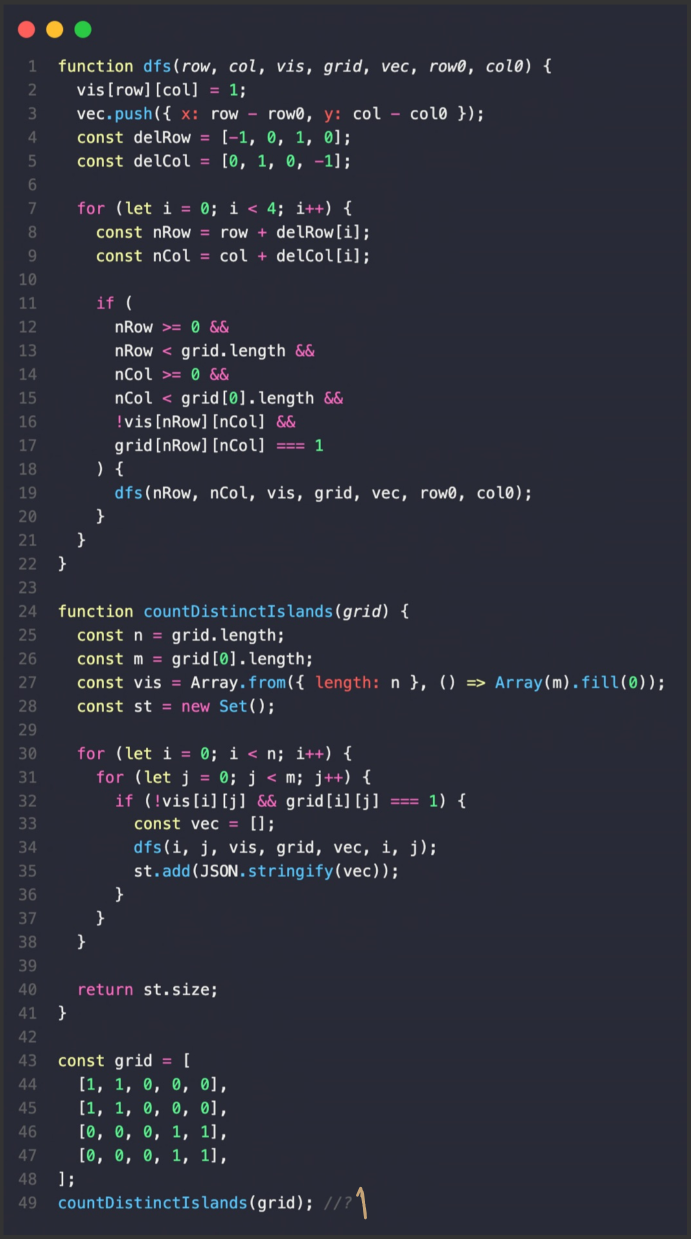
set => {(0,0)(0,1),(1,0)}
{(0,0),(0,1),(1,0)}



(2,3) - (2,3) => (0,0)
(2,4) - (2,3) => (0,1)
(3,3) - (2,3) => (1,0)



once one dfs calls completed store this shape in the set



```
function dfs(row, col, vis, grid, vec, row0, col0) {
  vis[row][col] = 1;
  vec.push({ x: row - row0, y: col - col0 });
  const delRow = [-1, 0, 1, 0];
  const delCol = [0, 1, 0, -1];

  for (let i = 0; i < 4; i++) {
    const nRow = row + delRow[i];
    const nCol = col + delCol[i];

    if (
      nRow >= 0 &&
      nRow < grid.length &&
      nCol >= 0 &&
      nCol < grid[0].length &&
      !vis[nRow][nCol] &&
      grid[nRow][nCol] === 1
    ) {
      dfs(nRow, nCol, vis, grid, vec, row0, col0);
    }
  }
}

function countDistinctIslands(grid) {
  const n = grid.length;
  const m = grid[0].length;
  const vis = Array.from({ length: n }, () => Array(m).fill(0));
  const st = new Set();

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      if (!vis[i][j] && grid[i][j] === 1) {
        const vec = [];
        dfs(i, j, vis, grid, vec, i, j);
        st.add(JSON.stringify(vec));
      }
    }
  }

  return st.size;
}

const grid = [
  [1, 1, 0, 0, 0],
  [1, 1, 0, 0, 0],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 1, 1],
];
countDistinctIslands(grid); //?1
```

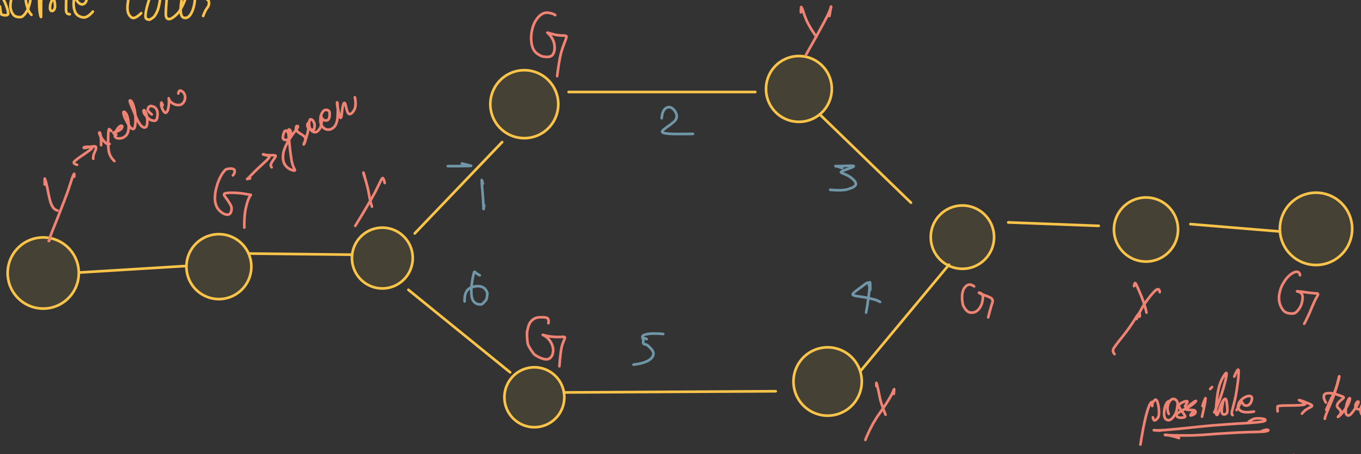
TC→ $O(N \times M) \times 4 + O(N \times M \times \log(N \times M))$

SC→ $O(N \times M)$

---

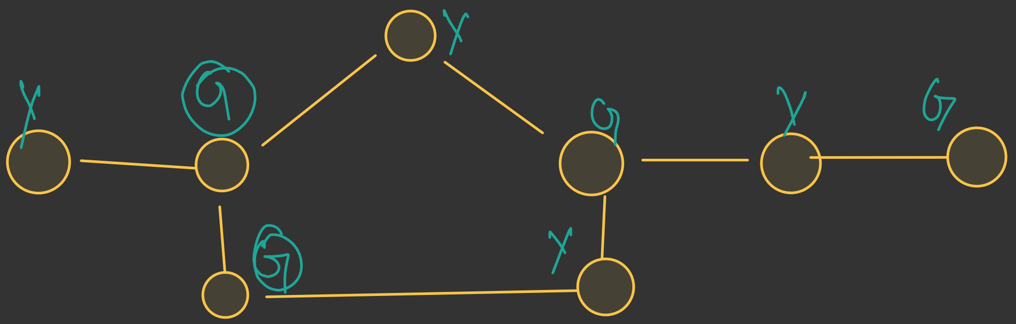
lec17 bipartite graph

Color the graph with 2 colors such that no adjacent nodes have same color



possible → true

bipartite graph



not possible

→ linear graph with no cycle are alway bipartite

→ any graph with even cycle length are bipartite

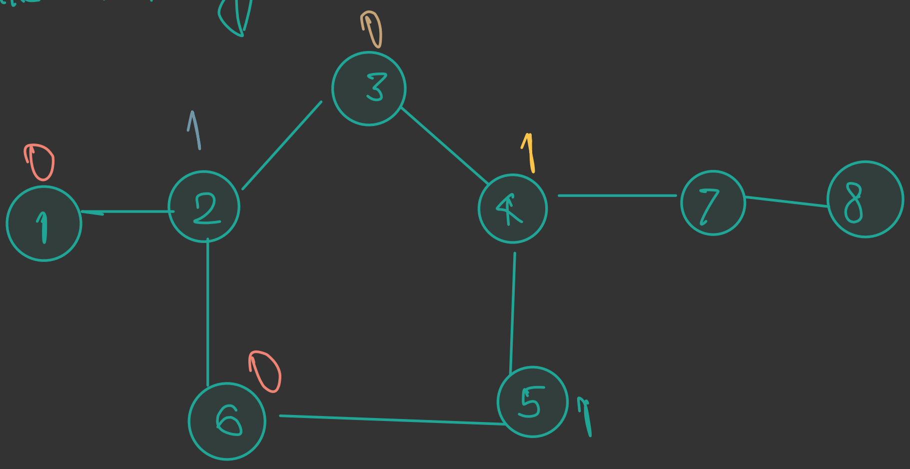
odd cycle length → not bipartite

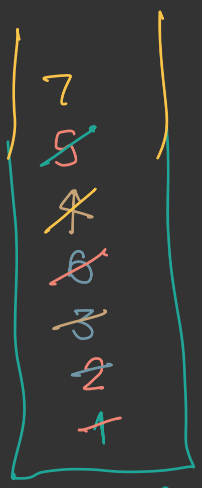
BFS

color =
↓
(like visited array)

-1 → not colored yet
0 → color1
1 → color2

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





queue

color =

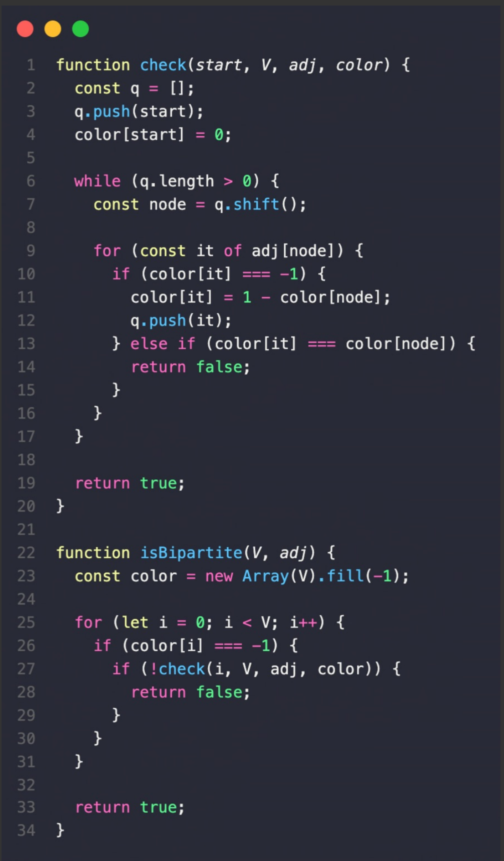
0	1	0	1	1	0		
~~-1~~	~~-1~~	~~-1~~	~~-1~~	~~-1~~	~~-1~~	-1	-1
1	2	3	4	5	6	7	8

intuition

same as BFS

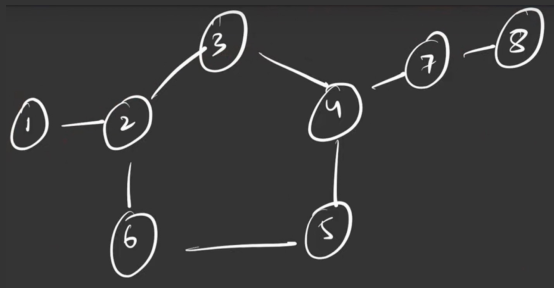
TC →

SC →



```
function check(start, V, adj, color) {
  const q = [];
  q.push(start);
  color[start] = 0;

  while (q.length > 0) {
    const node = q.shift();

    for (const it of adj[node]) {
      if (color[it] === -1) {
        color[it] = 1 - color[node];
        q.push(it);
      } else if (color[it] === color[node]) {
        return false;
      }
    }
  }

  return true;
}

function isBipartite(V, adj) {
  const color = new Array(V).fill(-1);

  for (let i = 0; i < V; i++) {
    if (color[i] === -1) {
      if (!check(i, V, adj, color)) {
        return false;
      }
    }
  }

  return true;
}
```

lec18. Bipartite graph using DFS



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

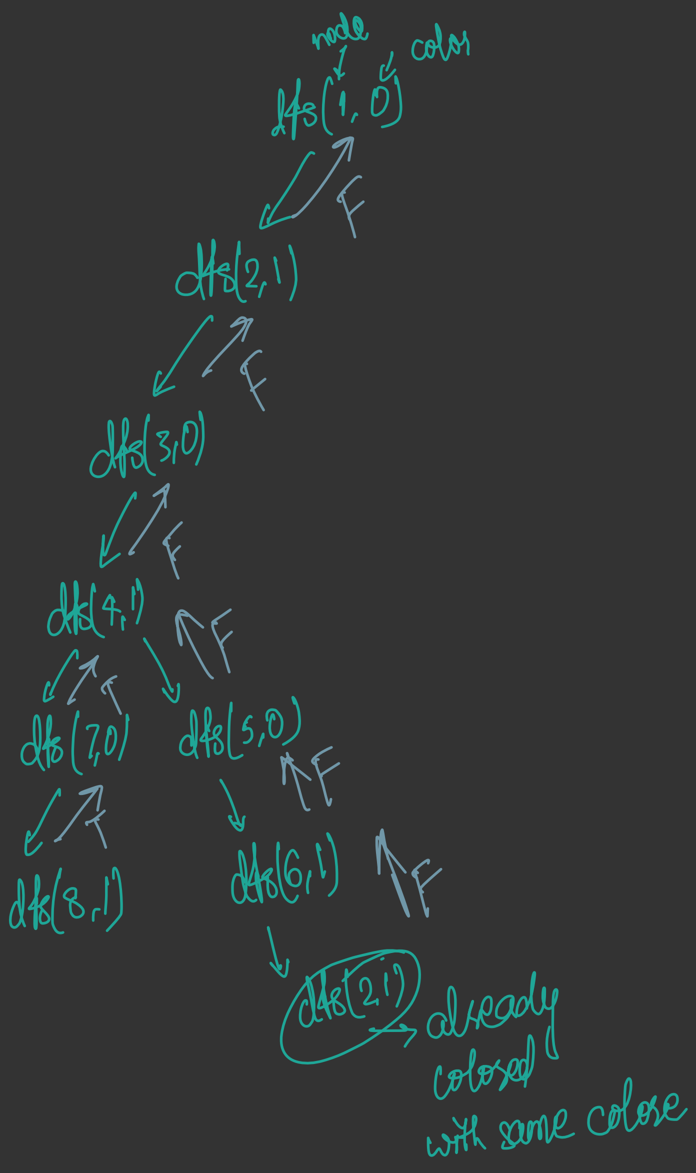
start = 1

color : 0/1

color =

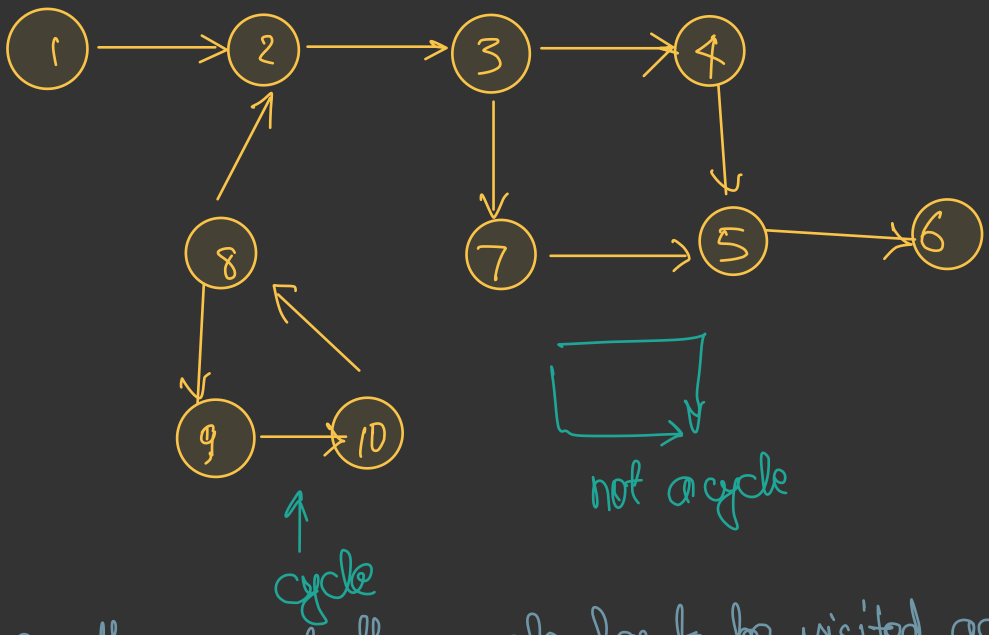
-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

TC → O(V+2E)
SC → O(V) + ↑ recursion stack



```javascript
function dfs(node, col, color, adj) {
  color[node] = col;
  for (let it of adj[node]) {
    if (color[it] === -1) {
      if (dfs(it, !col, color, adj) === false) return false;
    } else if (color[it] === col) {
      return false;
    }
  }
  return true;
}

function isBipartite(V, adj) {
  let color = new Array(V).fill(-1);

  for (let i = 0; i < V; i++) {
    if (color[i] === -1) {
      if (dfs(i, 0, color, adj) === false) return false;
    }
  }
  return true;
}
```

lec19 detect cycle in a directed graph (DFS)

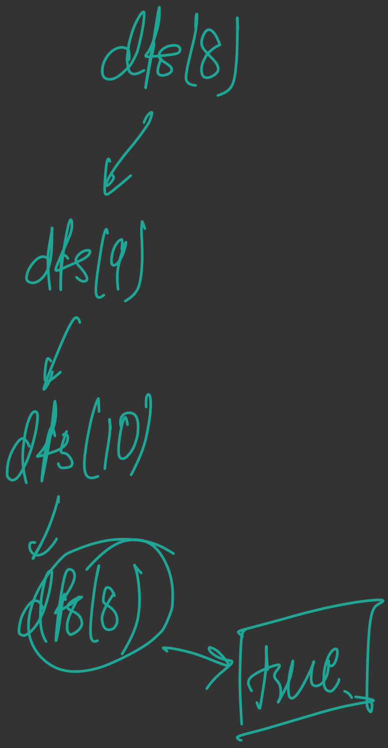


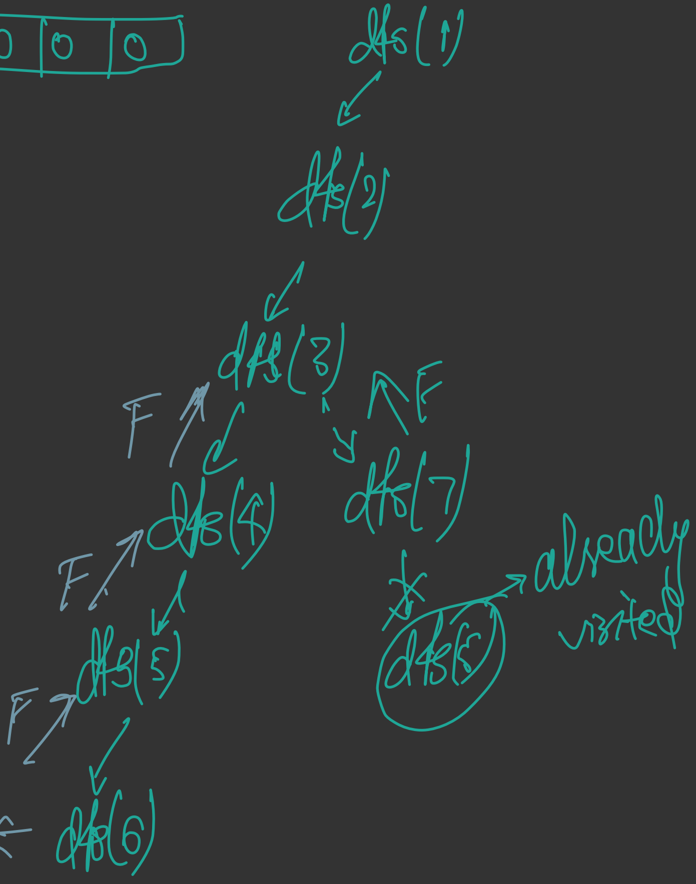
1 → {2}
2 → {3}
3 → {4, 7}
4 → {5}
5 → {6}
6 → {}
7 → {5}
8 → {9}
9 → {10}
10 → {8}

→ on the same path a node has to be visited again to be it a cycle

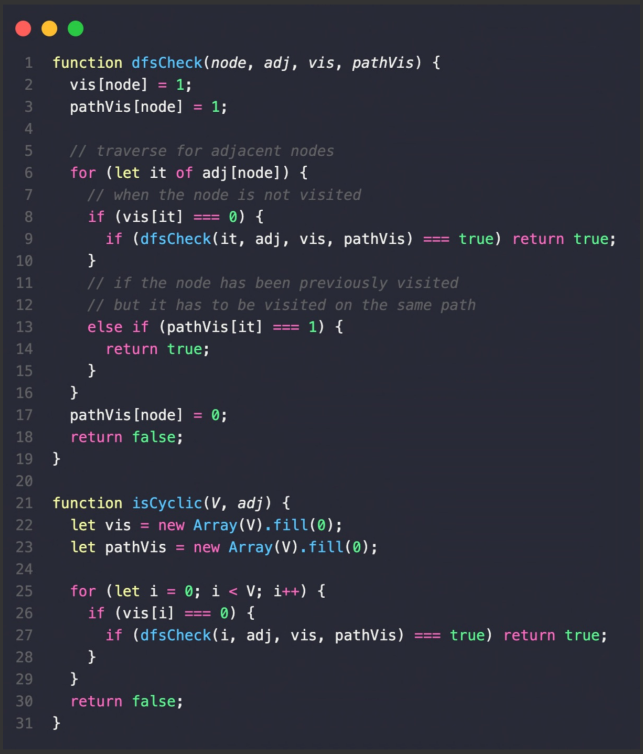
	0	1	2	3	4	5	6	7	8	9	10
vis =	0	0	0	0	0	0	0	0	0	0	0
path vis =	0	0	0	0	0	0	0	0	0	0	0

path vis → visited on the same path





set path visit to 0 again for 6



```
function dfsCheck(node, adj, vis, pathVis) {
  vis[node] = 1;
  pathVis[node] = 1;

  // traverse for adjacent nodes
  for (let it of adj[node]) {
    // when the node is not visited
    if (vis[it] === 0) {
      if (dfsCheck(it, adj, vis, pathVis) === true) return true;
    }
    // if the node has been previously visited
    // but it has to be visited on the same path
    else if (pathVis[it] === 1) {
      return true;
    }
  }
  pathVis[node] = 0;
  return false;
}

function isCyclic(V, adj) {
  let vis = new Array(V).fill(0);
  let pathVis = new Array(V).fill(0);

  for (let i = 0; i < V; i++) {
    if (vis[i] === 0) {
      if (dfsCheck(i, adj, vis, pathVis) === true) return true;
    }
  }
  return false;
}
```

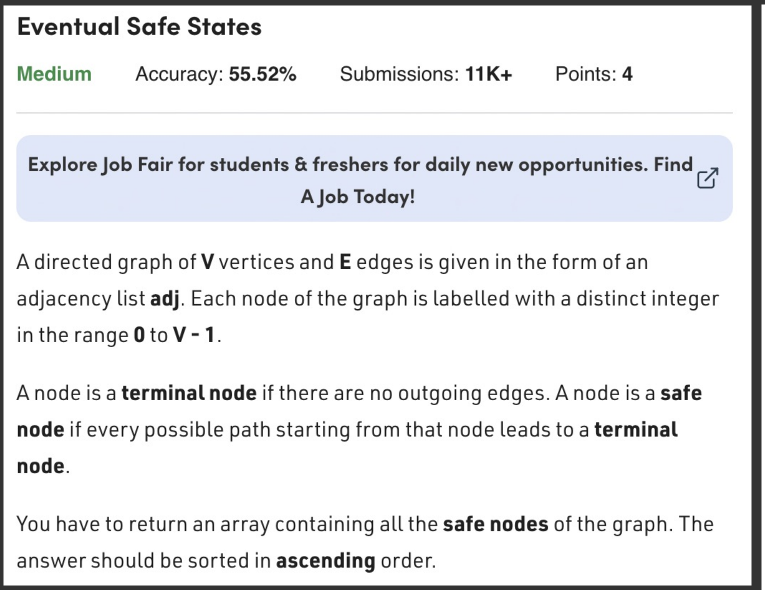
TC→

SC→

same as DFS

---

lec20. find eventual safe states.



Eventual Safe States

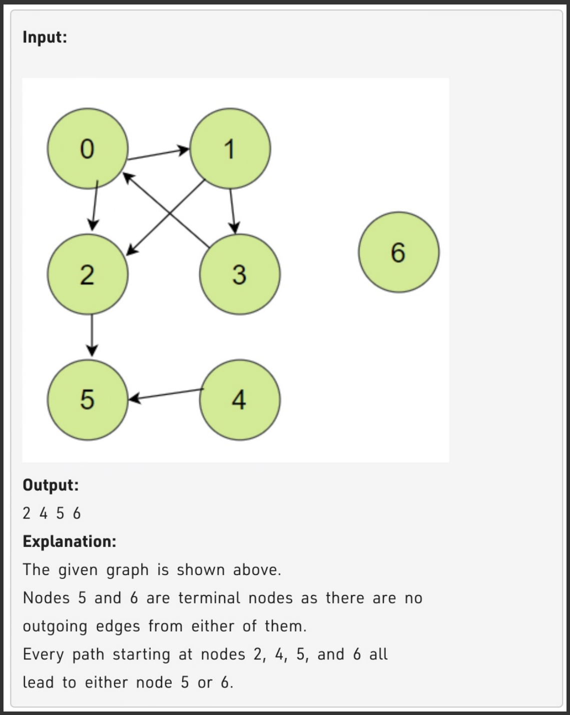
Medium Accuracy: **55.52%** Submissions: **11K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

A directed graph of **V** vertices and **E** edges is given in the form of an adjacency list **adj**. Each node of the graph is labelled with a distinct integer in the range **0** to **V - 1**.

A node is a **terminal node** if there are no outgoing edges. A node is a **safe node** if every possible path starting from that node leads to a **terminal node**.

You have to return an array containing all the **safe nodes** of the graph. The answer should be sorted in **ascending** order.



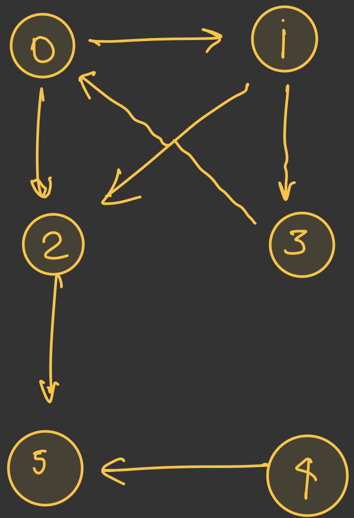
Input:

Output:

2 4 5 6

Explanation:

The given graph is shown above.
Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.
Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.



safe node →?

terminal node

↳ outdegree → 0

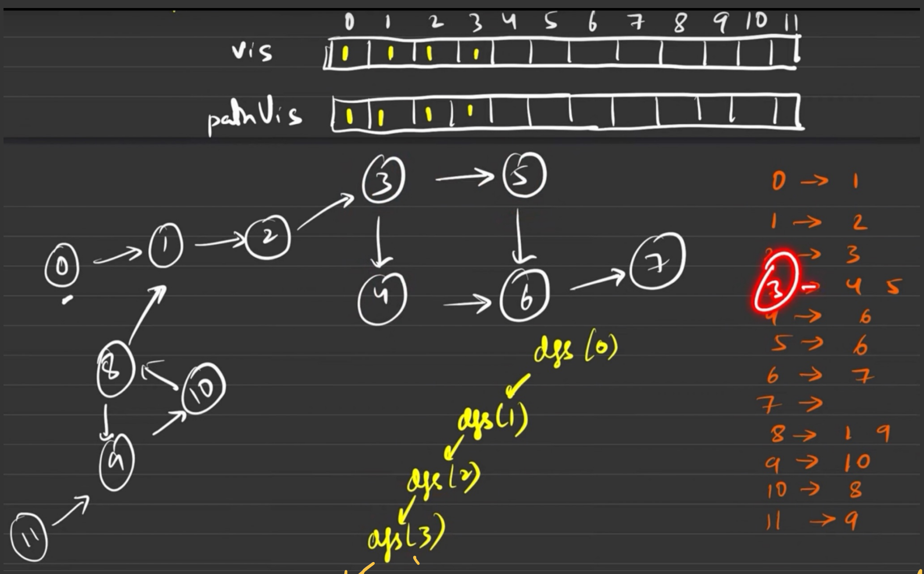
⑥ → outdegree=0
terminal

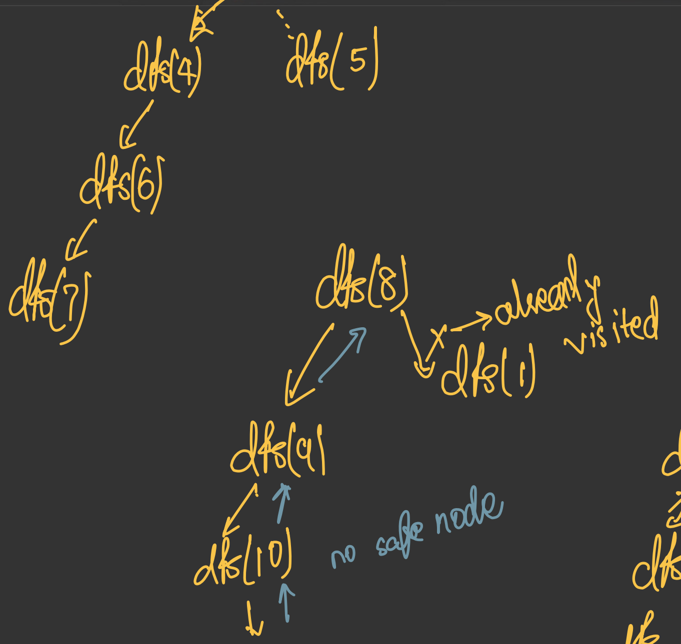
⑤ ✓

safe node → 2, 4

① any node who is part of cycle can not be safe node

② any one who leads to a cycle can not be a safe node





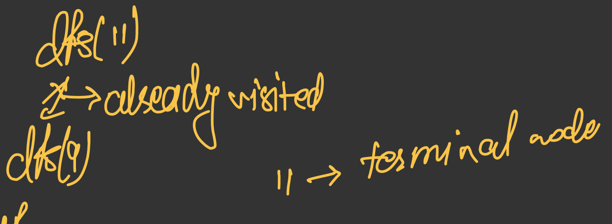
7 → terminal node

6 leads to 7 → safe node

4 —— 7 → safe node

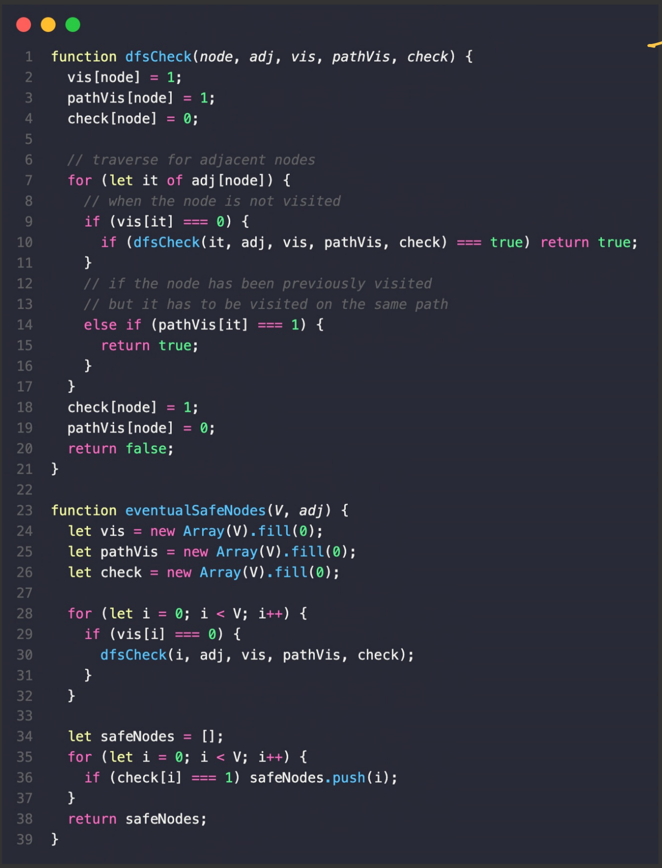
3 has 2 path → check both

0, 1, 2, 3, 4, 5, 6 → safe nodes.

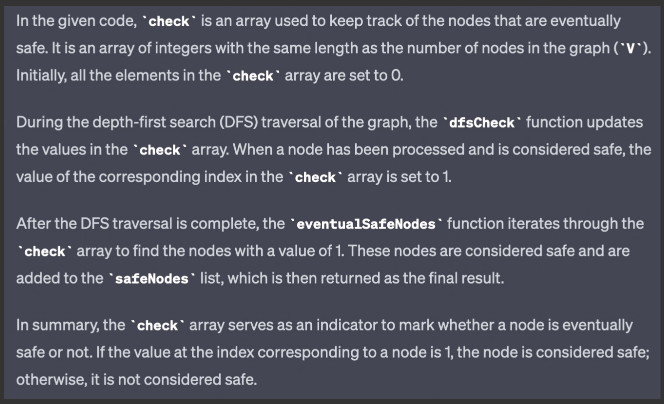


TC → same as dfs.

SC →



```
function dfsCheck(node, adj, vis, pathVis, check) {
  vis[node] = 1;
  pathVis[node] = 1;
  check[node] = 0;

  // traverse for adjacent nodes
  for (let it of adj[node]) {
    // when the node is not visited
    if (vis[it] === 0) {
      if (dfsCheck(it, adj, vis, pathVis, check) === true) return true;
    }
    // if the node has been previously visited
    // but it has to be visited on the same path
    else if (pathVis[it] === 1) {
      return true;
    }
  }
  check[node] = 1;
  pathVis[node] = 0;
  return false;
}

function eventualSafeNodes(V, adj) {
  let vis = new Array(V).fill(0);
  let pathVis = new Array(V).fill(0);
  let check = new Array(V).fill(0);

  for (let i = 0; i < V; i++) {
    if (vis[i] === 0) {
      dfsCheck(i, adj, vis, pathVis, check);
    }
  }

  let safeNodes = [];
  for (let i = 0; i < V; i++) {
    if (check[i] === 1) safeNodes.push(i);
  }
  return safeNodes;
}
```



In the given code, `check` is an array used to keep track of the nodes that are eventually safe. It is an array of integers with the same length as the number of nodes in the graph (`V`). Initially, all the elements in the `check` array are set to 0.

During the depth-first search (DFS) traversal of the graph, the `dfsCheck` function updates the values in the `check` array. When a node has been processed and is considered safe, the value of the corresponding index in the `check` array is set to 1.

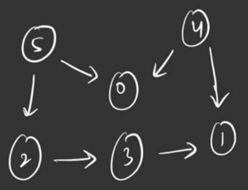
After the DFS traversal is complete, the `eventualSafeNodes` function iterates through the `check` array to find the nodes with a value of 1. These nodes are considered safe and are added to the `safeNodes` list, which is then returned as the final result.

In summary, the `check` array serves as an indicator to mark whether a node is eventually safe or not. If the value at the index corresponding to a node is 1, the node is considered safe; otherwise, it is not considered safe.

lec 21 ⇒ Topological Sort Algorithms.

Topological Sorting (DFS)

↳ linear ordering of vertices such that if there is an edge between u & v, u appears before v in that ordering.

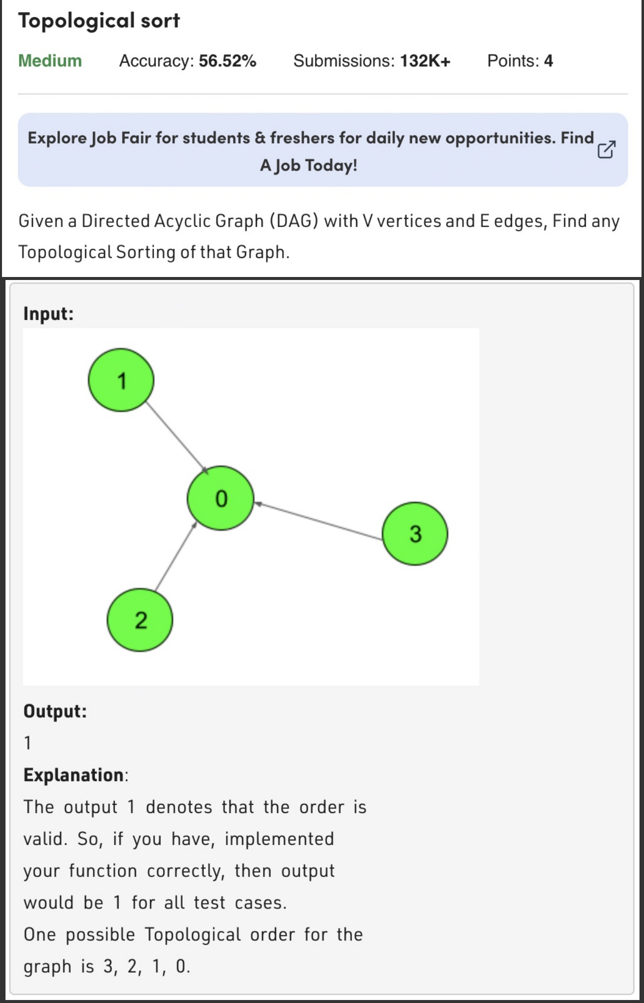


5 4 2 3 1 0 → valid ordering

4 5 2 3 1 0 → valid ordering

only in directed acyclic graph

5→0
4→0
5→2
2→3
3→1
4→1



Topological sort

Medium Accuracy: **56.52%** Submissions: **132K+** Points: **4**

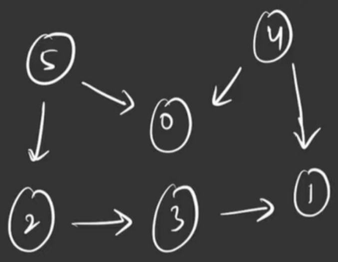
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

Input:

Output:
1

Explanation:
The output 1 denotes that the order is valid. So, if you have, implemented your function correctly, then output would be 1 for all test cases.
One possible Topological order for the graph is 3, 2, 1, 0.



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

~~dfs(0)~~
~~dfs(1)~~
dfs(2)
dfs(3)
already visited dfs(1)

dfs(4)
dfs(5)

visited =

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

5
4
3
2
1
0

stack

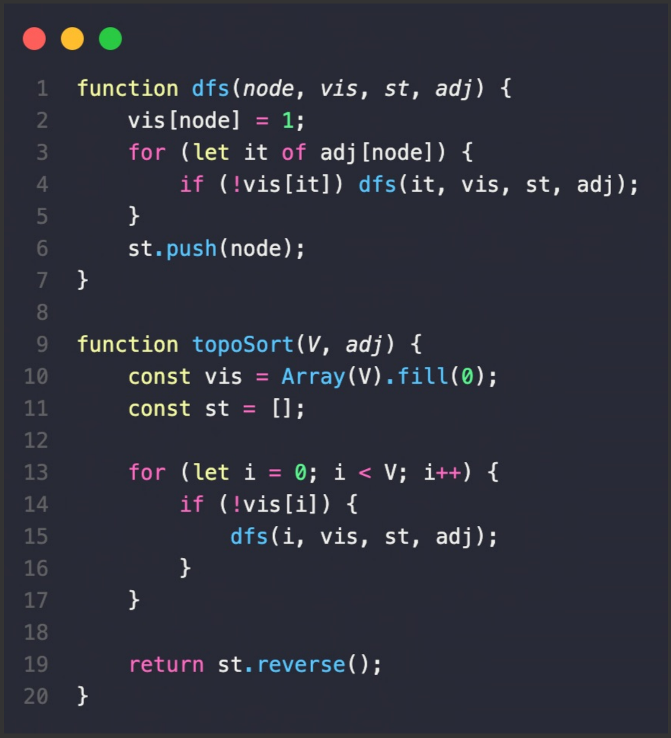
LIFO

```
for (i=0 to 5)
{ if (!vis[i])
      dfs(i)
}
```

5 4 3 2 1 0

ordering ans.

→ before returning from dfs, push the node to stack.



```
function dfs(node, vis, st, adj) {
    vis[node] = 1;
    for (let it of adj[node]) {
        if (!vis[it]) dfs(it, vis, st, adj);
    }
    st.push(node);
}

function topoSort(V, adj) {
    const vis = Array(V).fill(0);
    const st = [];

    for (let i = 0; i < V; i++) {
        if (!vis[i]) {
            dfs(i, vis, st, adj);
        }
    }

    return st.reverse();
}
```

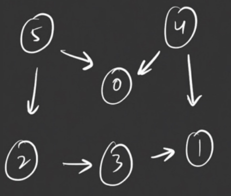
SC → $\underset{\uparrow \atop st}{O(N)} + \underset{\uparrow \atop vis}{O(N)}$

TC → O(V+E)

---

lec22 Kahn's Algorithm → BFS

Topological Sorting (Kahn's Algorithm / BFS) → linear ordering of vertices such that if there is an edge between u & v, u appears before v in the ordering



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

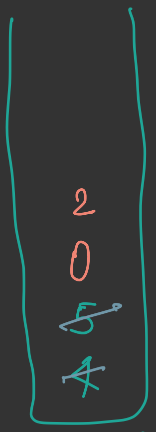
edges.

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

ordering → 5 4 0 2 3 1
os. 4 5 2 3 1 0

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

	0					
	~~1~~	1	0			
indegree:	~~2~~	~~2~~	~~1~~	1	0	0
	0	1	2	3	4	5



queue
FIFO

① insert all nodes with 0 indegree in queue

② shift 4 from queue
reduce the 4 as indegree from connected node, 0, 1

③ shift 5 from queue
reduce the 5 as . . . . . .

④ when indegree becomes 0 for a node put it into queue

How to calculate indegree

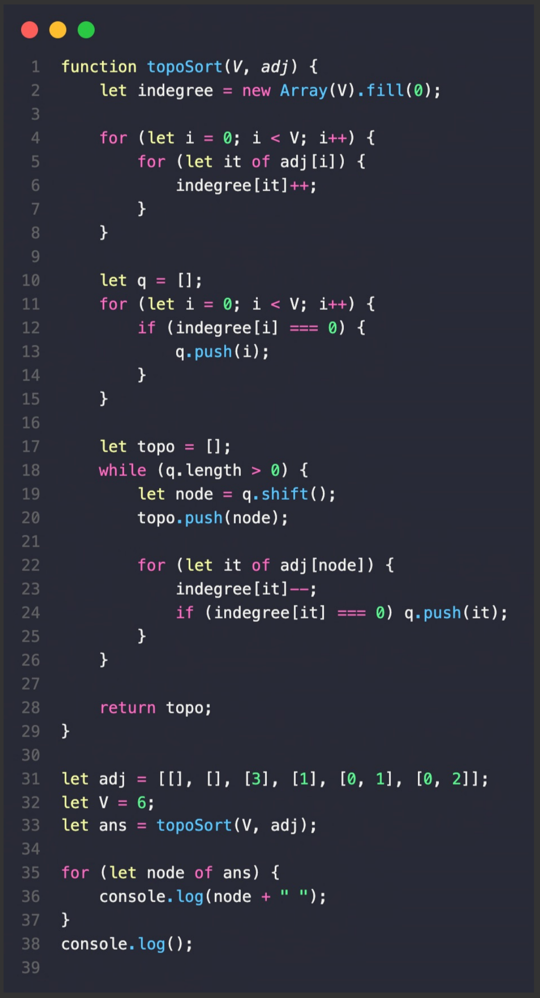
using adjacent List

traverse the List & increase the indegree

SC-> O(N)+O(N)+O(N)

↑ indegree ↑ q ↑ topo

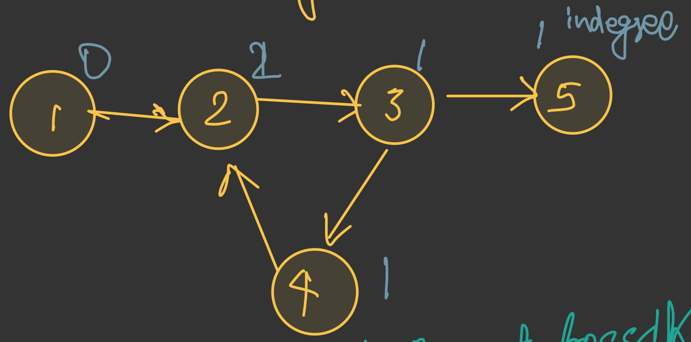
TC-> O(V+E)



```
function topoSort(V, adj) {
    let indegree = new Array(V).fill(0);

    for (let i = 0; i < V; i++) {
        for (let it of adj[i]) {
            indegree[it]++;
        }
    }

    let q = [];
    for (let i = 0; i < V; i++) {
        if (indegree[i] === 0) {
            q.push(i);
        }
    }

    let topo = [];
    while (q.length > 0) {
        let node = q.shift();
        topo.push(node);

        for (let it of adj[node]) {
            indegree[it]--;
            if (indegree[it] === 0) q.push(it);
        }
    }

    return topo;
}

let adj = [[], [], [3], [1], [0, 1], [0, 2]];
let V = 6;
let ans = topoSort(V, adj);

for (let node of ans) {
    console.log(node + " ");
}
console.log();
```

---

lec23. detect a cycle in a directed graph



BFS

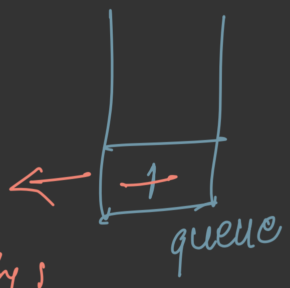
1 -> {2}
2 -> {3}
3 -> {4, 5}
4 -> {2}
5 -> { }

we know topo sort is not possible for this, but we will still apply topo sort

Topo sort -> 1

toposort.len < num of nodes

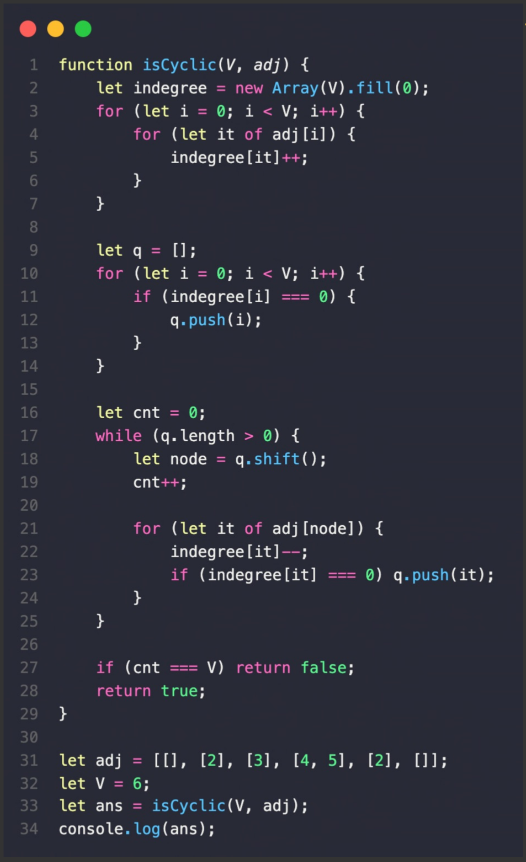
=> there is a cycle in graph



1

2 reduced by 1

but now no 0 indegree node



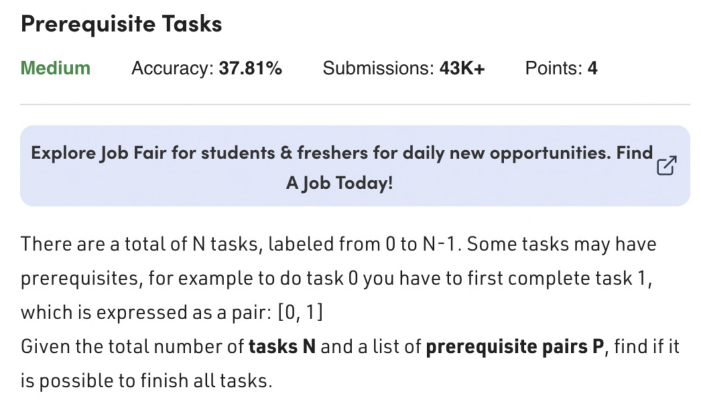
```
function isCyclic(V, adj) {
    let indegree = new Array(V).fill(0);
    for (let i = 0; i < V; i++) {
        for (let it of adj[i]) {
            indegree[it]++;
        }
    }

    let q = [];
    for (let i = 0; i < V; i++) {
        if (indegree[i] === 0) {
            q.push(i);
        }
    }

    let cnt = 0;
    while (q.length > 0) {
        let node = q.shift();
        cnt++;

        for (let it of adj[node]) {
            indegree[it]--;
            if (indegree[it] === 0) q.push(it);
        }
    }

    if (cnt === V) return false;
    return true;
}

let adj = [[], [2], [3], [4, 5], [2], []];
let V = 6;
let ans = isCyclic(V, adj);
console.log(ans);
```

TC → O(V+E)

SC → O(N) + O(N)

---

lec 24 → course schedule 1 & 2 | pre requisite task|



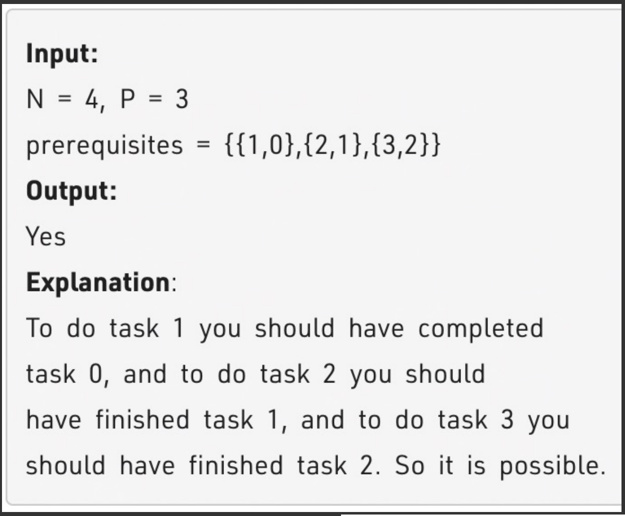
Prerequisite Tasks

Medium Accuracy: **37.81%** Submissions: **43K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for example to do task 0 you have to first complete task 1, which is expressed as a pair: [0, 1]

Given the total number of **tasks N** and a list of **prerequisite pairs P**, find if it is possible to finish all tasks.



Input:

N = 4, P = 3

prerequisites = {{1,0},{2,1},{3,2}}

Output:

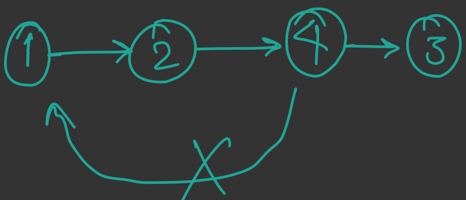
Yes

Explanation:

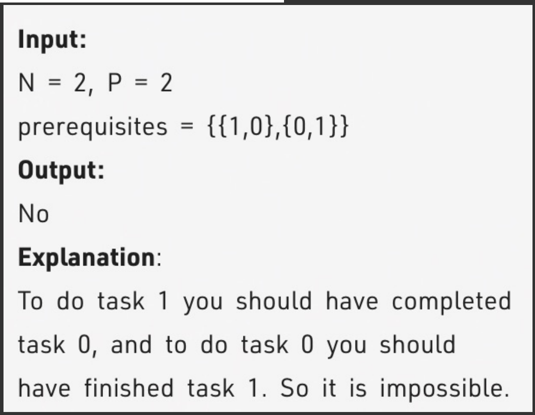
To do task 1 you should have completed task 0, and to do task 2 you should have finished task 1, and to do task 3 you should have finished task 2. So it is possible.

1	2	✓
4	3	✓
2	4	✓
4	1	✗

topo sort



ans → false.



Input:

N = 2, P = 2

prerequisites = {{1,0},{0,1}}

Output:

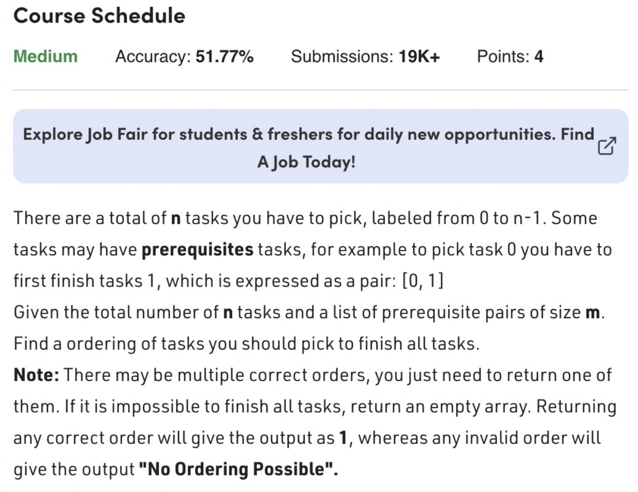
No

Explanation:

To do task 1 you should have completed task 0, and to do task 0 you should have finished task 1. So it is impossible.

way1 → detect if the graph has cycle, if yes return false

way2 → if topo sort not possible, return false ✓



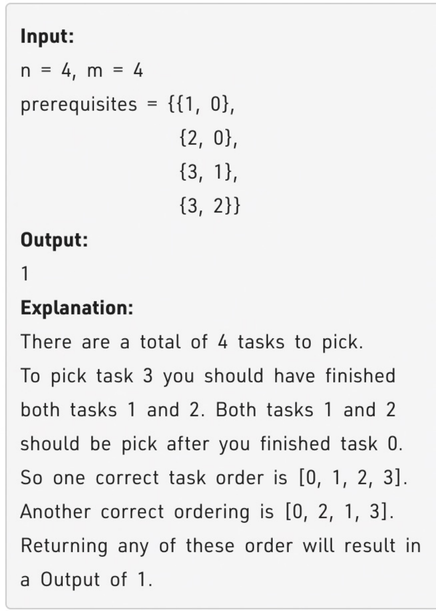
Course Schedule

Medium Accuracy: **51.77%** Submissions: **19K+** Points: **4**

There are a total of **n** tasks you have to pick, labeled from 0 to n-1. Some tasks may have **prerequisites** tasks, for example to pick task 0 you have to first finish tasks 1, which is expressed as a pair: [0, 1]

Given the total number of **n** tasks and a list of prerequisite pairs of size **m**. Find a ordering of tasks you should pick to finish all tasks.

Note: There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all tasks, return an empty array. Returning any correct order will give the output as **1**, whereas any invalid order will give the output **"No Ordering Possible"**.



Input:

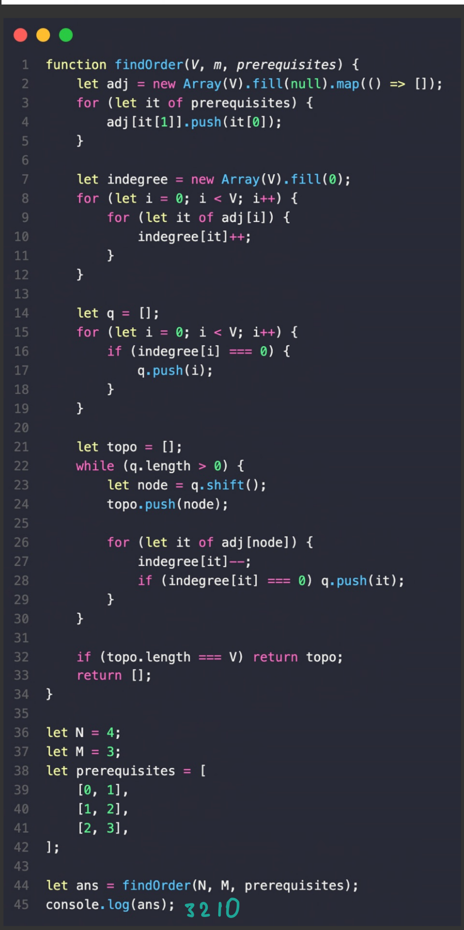
```
n = 4, m = 4
prerequisites = {{1, 0},
                 {2, 0},
                 {3, 1},
                 {3, 2}}
```

Output:

```
1
```

Explanation:

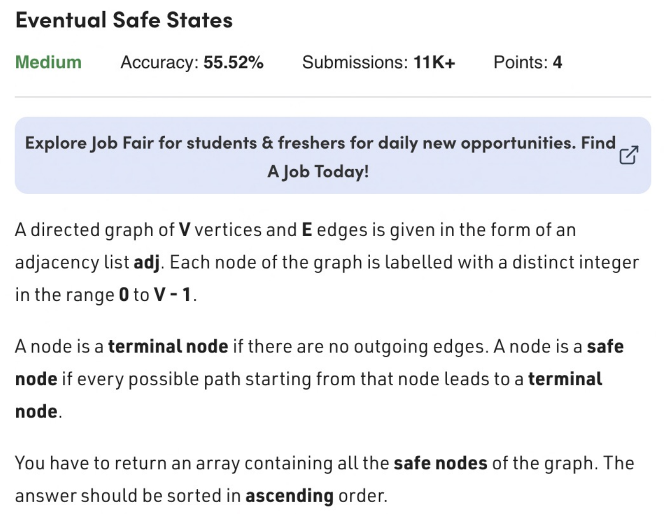
There are a total of 4 tasks to pick. To pick task 3 you should have finished both tasks 1 and 2. Both tasks 1 and 2 should be pick after you finished task 0. So one correct task order is [0, 1, 2, 3]. Another correct ordering is [0, 2, 1, 3]. Returning any of these order will result in a Output of 1.



```javascript
function findOrder(V, m, prerequisites) {
    let adj = new Array(V).fill(null).map(() => []);
    for (let it of prerequisites) {
        adj[it[1]].push(it[0]);
    }

    let indegree = new Array(V).fill(0);
    for (let i = 0; i < V; i++) {
        for (let it of adj[i]) {
            indegree[it]++;
        }
    }

    let q = [];
    for (let i = 0; i < V; i++) {
        if (indegree[i] === 0) {
            q.push(i);
        }
    }

    let topo = [];
    while (q.length > 0) {
        let node = q.shift();
        topo.push(node);

        for (let it of adj[node]) {
            indegree[it]--;
            if (indegree[it] === 0) q.push(it);
        }
    }

    if (topo.length === V) return topo;
    return [];
}

let N = 4;
let M = 3;
let prerequisites = [
    [0, 1],
    [1, 2],
    [2, 3],
];

let ans = findOrder(N, M, prerequisites);
console.log(ans);
```

3210

lec25 find eventual safe states - BFS.



Eventual Safe States

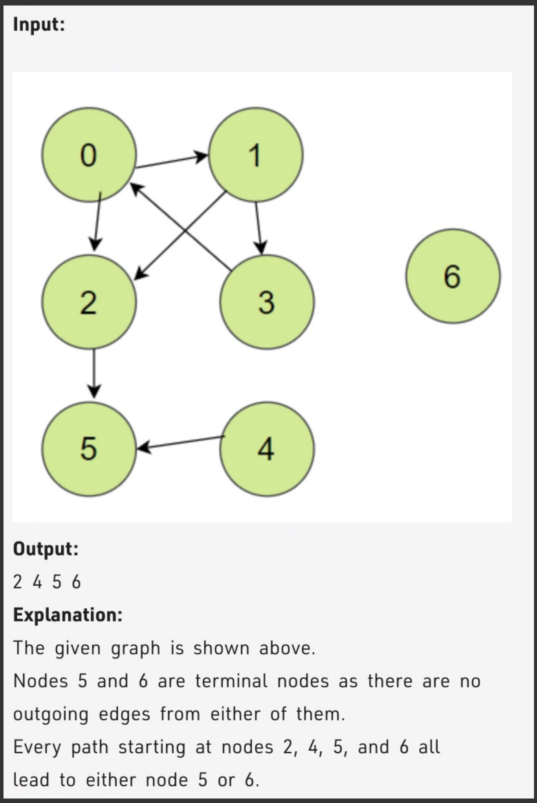
Medium Accuracy: **55.52%** Submissions: **11K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

A directed graph of **V** vertices and **E** edges is given in the form of an adjacency list **adj**. Each node of the graph is labelled with a distinct integer in the range **0** to **V - 1**.

A node is a **terminal node** if there are no outgoing edges. A node is a **safe node** if every possible path starting from that node leads to a **terminal node**.

You have to return an array containing all the **safe nodes** of the graph. The answer should be sorted in **ascending** order.



Input:

Output:

2 4 5 6

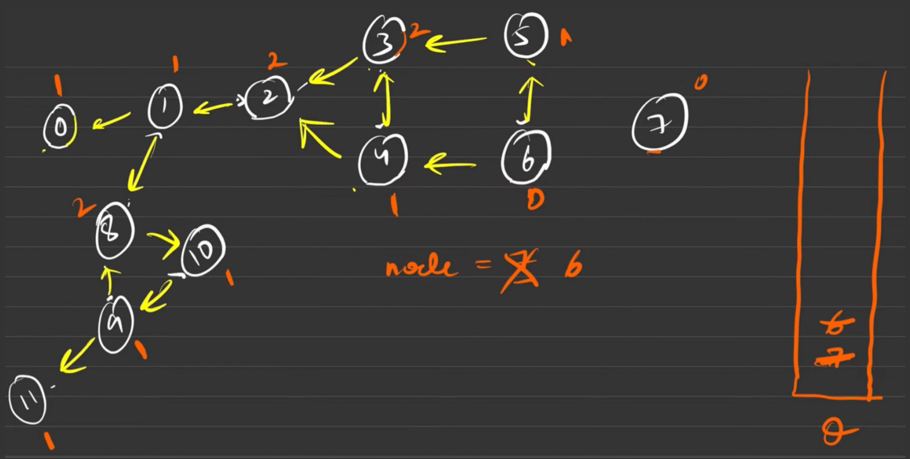
Explanation:

The given graph is shown above.

Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.

Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.

→ apply topo sort



→ backtrack from terminal node

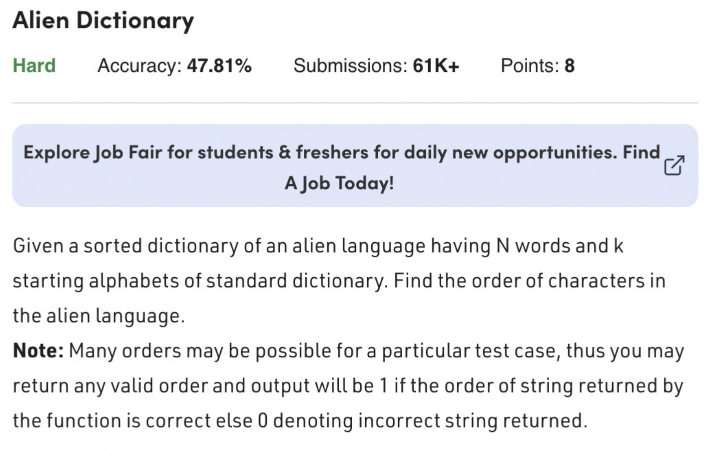
terminal node → T

→ indegree becomes 0
push it in queue

```
function eventualSafeNodes(V, adj) {
    const adjRev = Array.from({ length: V }, () => []);
    const indegree = Array(V).fill(0);

    for (let i = 0; i < V; i++) {
        for (const it of adj[i]) {
            adjRev[it].push(i);
            indegree[i]++;
        }
    }

    const q = [];
    const safeNodes = [];

    for (let i = 0; i < V; i++) {
        if (indegree[i] === 0) {
            q.push(i);
        }
    }

    while (q.length > 0) {
        const node = q.shift();
        safeNodes.push(node);

        for (const it of adjRev[node]) {
            indegree[it]--;
            if (indegree[it] === 0) q.push(it);
        }
    }

    safeNodes.sort((a, b) => a - b);
    return safeNodes;
}

const adj = [[1],[2],[3, 4],[4, 5],[6],[6],[7],[],[1, 9],[10],[8],[9]];
const V = 12;
const safeNodes = eventualSafeNodes(V, adj);

console.log(safeNodes.join(" "));
```

0 1 2 3 4 5 6 7

lec26 → alien dictionary - topological sort



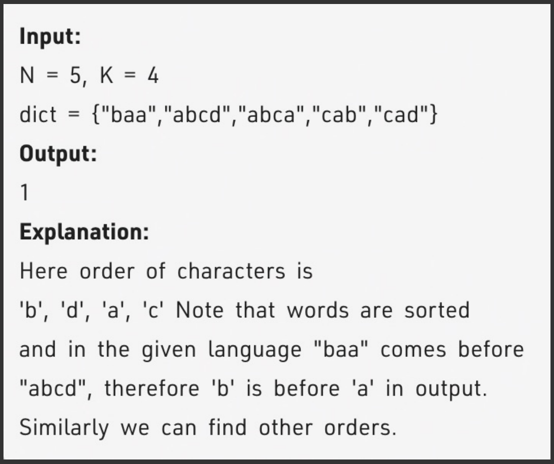
Alien Dictionary

Hard Accuracy: **47.81%** Submissions: **61K+** Points: **8**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a sorted dictionary of an alien language having N words and k starting alphabets of standard dictionary. Find the order of characters in the alien language.

Note: Many orders may be possible for a particular test case, thus you may return any valid order and output will be 1 if the order of string returned by the function is correct else 0 denoting incorrect string returned.



Input:

N = 5, K = 4

dict = {"baa","abcd","abca","cab","cad"}

Output:

1

Explanation:

Here order of characters is 'b', 'd', 'a', 'c' Note that words are sorted and in the given language "baa" comes before "abcd", therefore 'b' is before 'a' in output. Similarly we can find other orders.

K=4 → a,b,c,d

alien dict

b a a

a b c d

a b c a

ac c a b

c a d

b a a
a b c d ⇒ b a

~~a b c~~ d
~~a b c~~ a ⇒ d a

~~c a~~ b
~~c a~~ d ⇒ b d

⇒ b d a c ans.

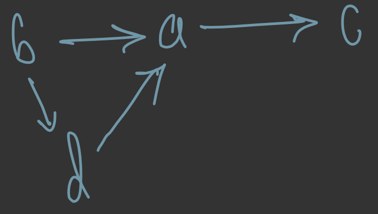
pattern → something before something

K=4

a	b	c	d
0	1	2	3

b a a
a b c d ⇒ b < a

~~a b c~~ d
~~a b c~~ a ⇒ d < a



a b c a
c a b ⇒ a < c

~~c a~~ b
~~c a~~ d ⇒ b < d

i=0 b a a
1 a b c d
2 a b c a
3 c a b
c a d

S1 = arr[i] → b a a
S2 = arr[i+1] → a b c d

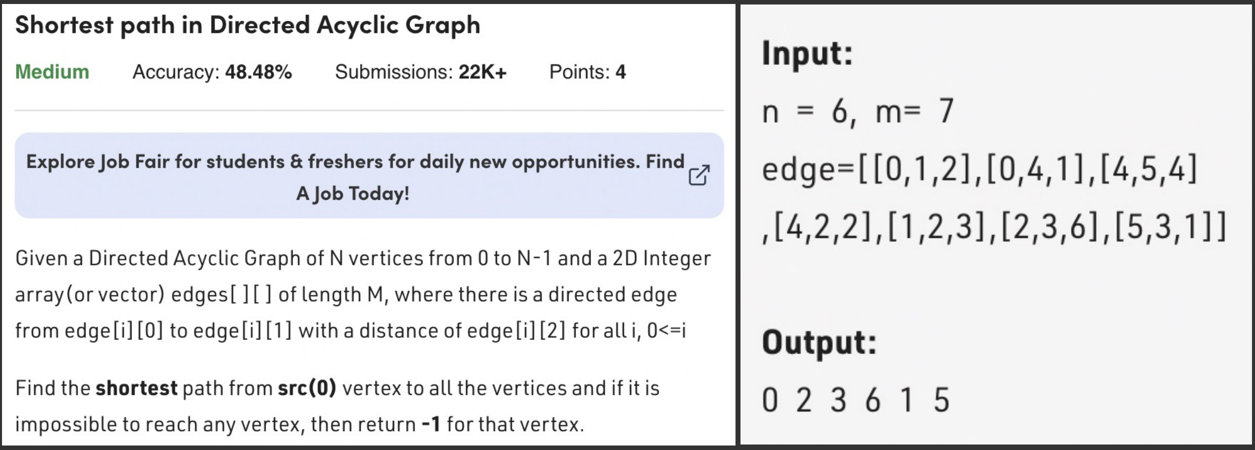
followup what if order is not possible?

① a b c d → S1
a b c → S2
if larger is before smaller, order is not possible

② a b c
b a t
a d e
→ cycle so order not possible

```javascript
function topoSort(V, adj) {
    let indegree = new Array(V).fill(0);
    for (let i = 0; i < V; i++) {
        for (const it of adj[i]) {
            indegree[it]++;
        }
    }

    let q = [];
    for (let i = 0; i < V; i++) {
        if (indegree[i] === 0) {
            q.push(i);
        }
    }
    let topo = [];
    while (q.length > 0) {
        let node = q.shift();
        topo.push(node);

        for (const it of adj[node]) {
            indegree[it]--;
            if (indegree[it] === 0) q.push(it);
        }
    }

    return topo;
}

function findOrder(dict, N, K) {
    let adj = new Array(K).fill(null).map(() => []);

    for (let i = 0; i < N - 1; i++) {
        let s1 = dict[i];
        let s2 = dict[i + 1];
        let len = Math.min(s1.length, s2.length);
        for (let ptr = 0; ptr < len; ptr++) {
            if (s1[ptr] !== s2[ptr]) {
                adj[s1.charCodeAt(ptr) - "a".charCodeAt(0)].push(
                    s2.charCodeAt(ptr) - "a".charCodeAt(0)
                );
                break;
            }
        }
    }

    let topo = topoSort(K, adj);
    let ans = "";
    for (const it of topo) {
        ans = ans + String.fromCharCode(it + "a".charCodeAt(0));
    }
    return ans;
}

let N = 5,
    K = 4;
let dict = ["baa", "abcd", "abca", "cab", "cad"];
let ans = findOrder(dict, N, K);
console.log(ans); // bdac
```

lec 27 shortest path in directed acyclic graph



Shortest path in Directed Acyclic Graph

Medium Accuracy: **48.48%** Submissions: **22K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a Directed Acyclic Graph of N vertices from 0 to N-1 and a 2D Integer array(or vector) edges[ ][ ] of length M, where there is a directed edge from edge[i][0] to edge[i][1] with a distance of edge[i][2] for all i, 0<=i

Find the **shortest** path from **src(0)** vertex to all the vertices and if it is impossible to reach any vertex, then return **-1** for that vertex.

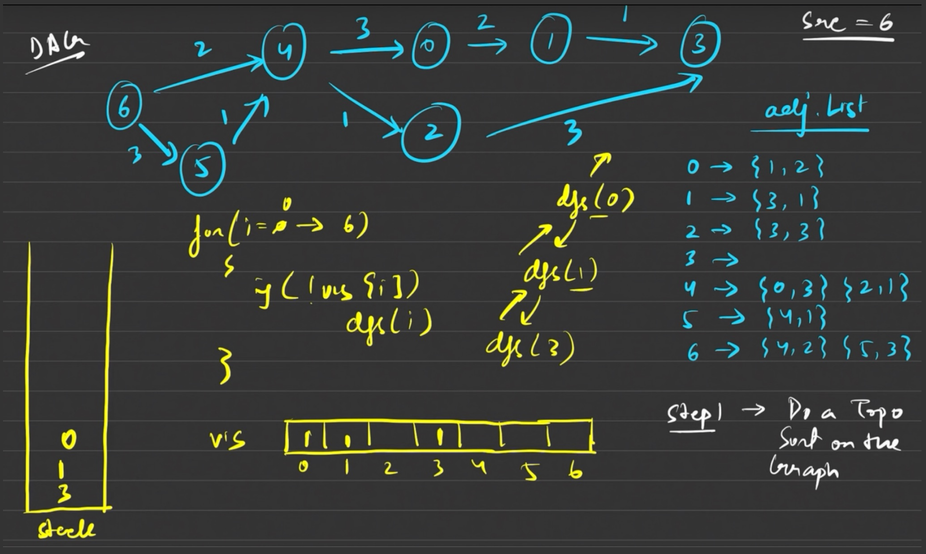
Input:

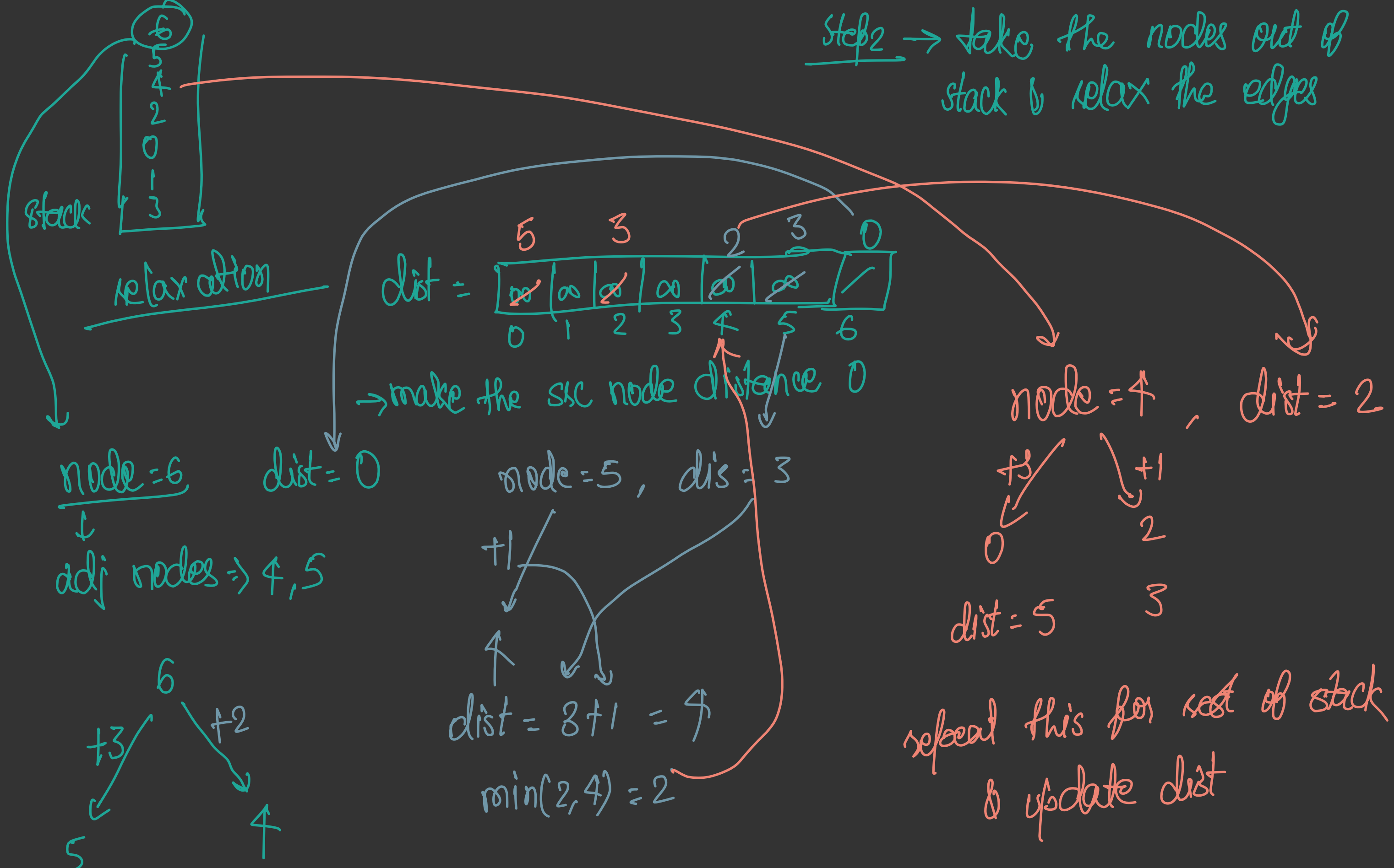
n = 6, m= 7

edge=[[0,1,2],[0,4,1],[4,5,4],[4,2,2],[1,2,3],[2,3,6],[5,3,1]]

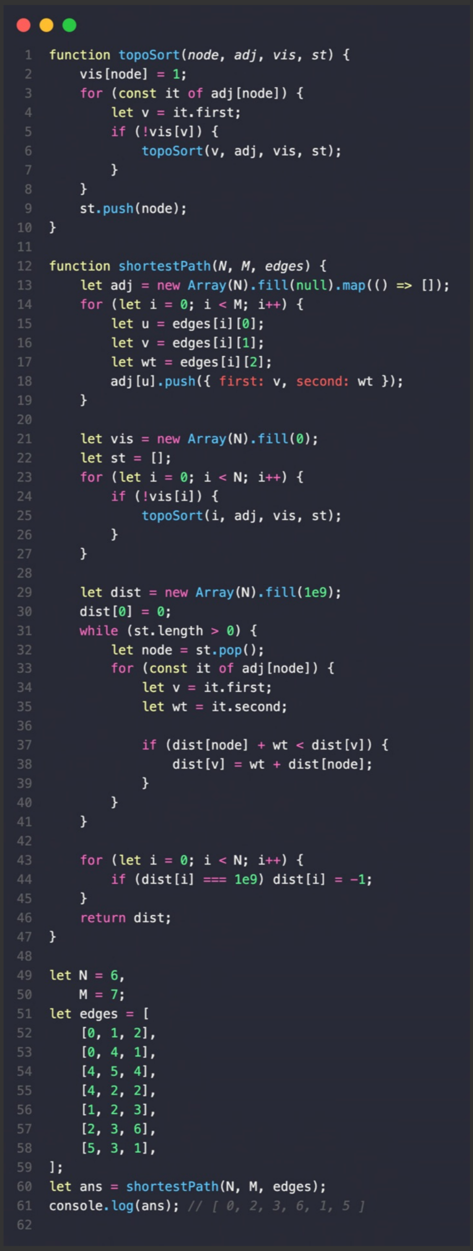
Output:

0 2 3 6 1 5



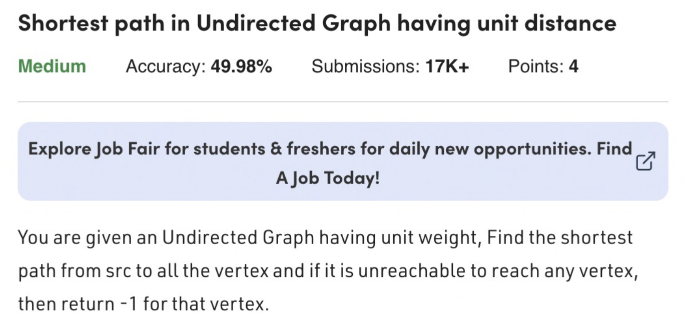


TC ⇒ O(N+M) + O(N+M)



```
function topoSort(node, adj, vis, st) {
    vis[node] = 1;
    for (const it of adj[node]) {
        let v = it.first;
        if (!vis[v]) {
            topoSort(v, adj, vis, st);
        }
    }
    st.push(node);
}

function shortestPath(N, M, edges) {
    let adj = new Array(N).fill(null).map(() => []);
    for (let i = 0; i < M; i++) {
        let u = edges[i][0];
        let v = edges[i][1];
        let wt = edges[i][2];
        adj[u].push({ first: v, second: wt });
    }

    let vis = new Array(N).fill(0);
    let st = [];
    for (let i = 0; i < N; i++) {
        if (!vis[i]) {
            topoSort(i, adj, vis, st);
        }
    }

    let dist = new Array(N).fill(1e9);
    dist[0] = 0;
    while (st.length > 0) {
        let node = st.pop();
        for (const it of adj[node]) {
            let v = it.first;
            let wt = it.second;

            if (dist[node] + wt < dist[v]) {
                dist[v] = wt + dist[node];
            }
        }
    }

    for (let i = 0; i < N; i++) {
        if (dist[i] === 1e9) dist[i] = -1;
    }
    return dist;
}

let N = 6,
    M = 7;
let edges = [
    [0, 1, 2],
    [0, 4, 1],
    [4, 5, 4],
    [4, 2, 2],
    [1, 2, 3],
    [2, 3, 6],
    [5, 3, 1],
];
let ans = shortestPath(N, M, edges);
console.log(ans); // [ 0, 2, 3, 6, 1, 5 ]
```

lec28. shortest path in undirected graph with unit weights.

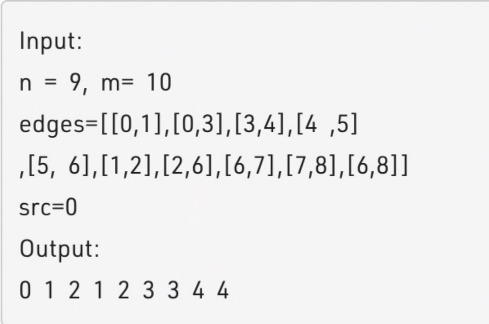


Shortest path in Undirected Graph having unit distance

Medium Accuracy: **49.98%** Submissions: **17K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

You are given an Undirected Graph having unit weight, Find the shortest path from src to all the vertex and if it is unreachable to reach any vertex, then return -1 for that vertex.



Input:

n = 9, m= 10

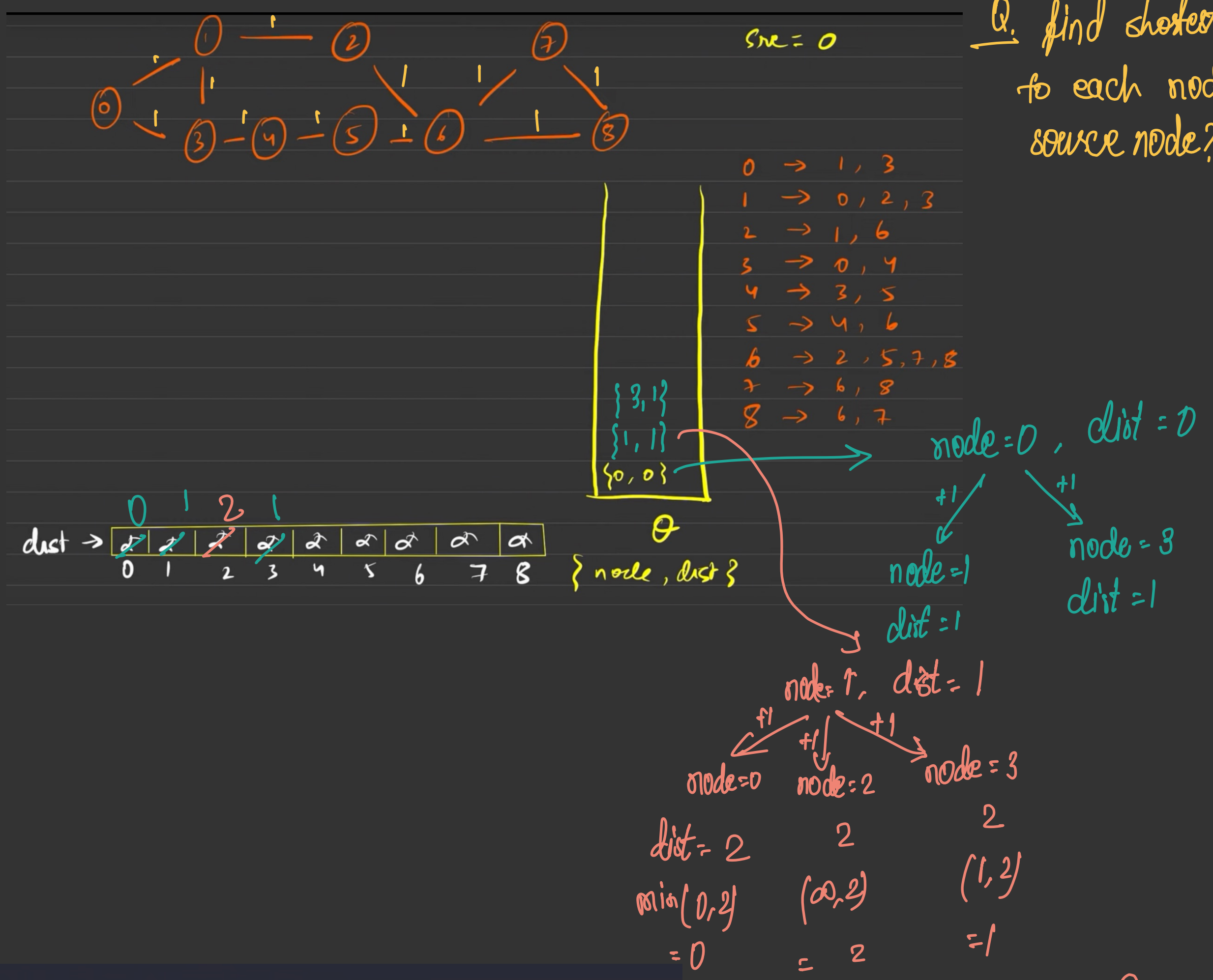
edges=[[0,1],[0,3],[3,4],[4 ,5],[5, 6],[1,2],[2,6],[6,7],[7,8],[6,8]]

src=0

Output:

0 1 2 1 2 3 3 4 4

Q. find shortest distance to each node from the source node?

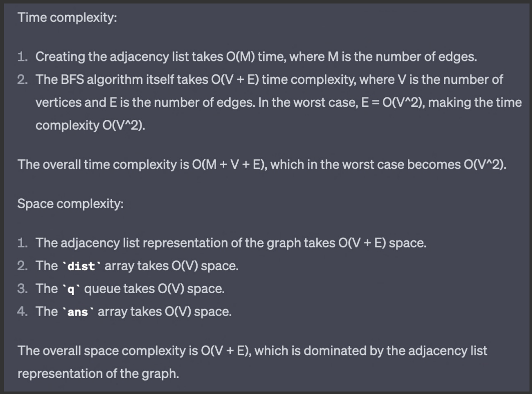


repeat this for all the nodes

TC → O(N + 2E)

SC →

```javascript
function shortestPath(edges, N, M, src) {
    let adj = new Array(N).fill(null).map(() => []);
    for (const it of edges) {
        adj[it[0]].push(it[1]);
        adj[it[1]].push(it[0]);
    }

    let dist = new Array(N).fill(1e9);
    dist[src] = 0;
    let q = [];
    q.push(src);
    while (q.length > 0) {
        let node = q.shift();
        for (const it of adj[node]) {
            if (dist[node] + 1 < dist[it]) {
                dist[it] = 1 + dist[node];
                q.push(it);
            }
        }
    }

    let ans = new Array(N).fill(-1);
    for (let i = 0; i < N; i++) {
        if (dist[i] !== 1e9) {
            ans[i] = dist[i];
        }
    }
    return ans;
}

let N = 9,
    M = 10;
let edges = [[0, 1],[0, 3],[3, 4],[4, 5],[5, 6],[1, 2],[2, 6],[6, 7],[7, 8],[6, 8]];

let ans = shortestPath(edges, N, M, 0);
console.log(ans); //[ 0, 1, 2, 1, 2, 3, 3, 4, 4 ]
```



Time complexity:

1. Creating the adjacency list takes O(M) time, where M is the number of edges.
2. The BFS algorithm itself takes O(V + E) time complexity, where V is the number of vertices and E is the number of edges. In the worst case, E = O(V^2), making the time complexity O(V^2).

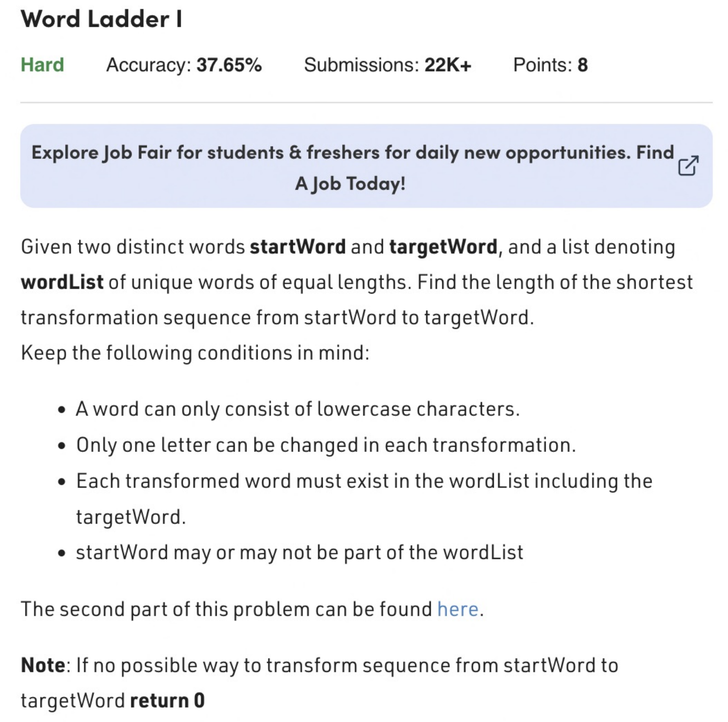
The overall time complexity is O(M + V + E), which in the worst case becomes O(V^2).

Space complexity:

1. The adjacency list representation of the graph takes O(V + E) space.
2. The `dist` array takes O(V) space.
3. The `q` queue takes O(V) space.
4. The `ans` array takes O(V) space.

The overall space complexity is O(V + E), which is dominated by the adjacency list representation of the graph.

lec29. word ladder 1 [shortest path



Word Ladder I

Hard Accuracy: **37.65%** Submissions: **22K+** Points: **8**

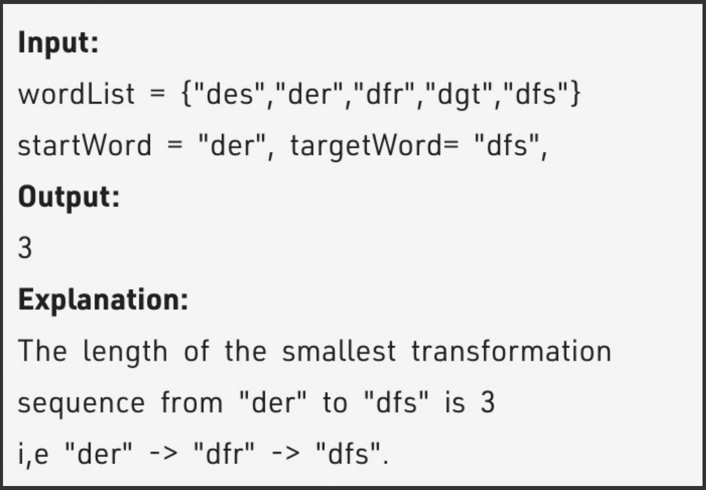
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given two distinct words **startWord** and **targetWord**, and a list denoting **wordList** of unique words of equal lengths. Find the length of the shortest transformation sequence from startWord to targetWord.
Keep the following conditions in mind:

- A word can only consist of lowercase characters.
- Only one letter can be changed in each transformation.
- Each transformed word must exist in the wordList including the targetWord.
- startWord may or may not be part of the wordList

The second part of this problem can be found here.

Note: If no possible way to transform sequence from startWord to targetWord **return 0**



Input:

wordList = {"des","der","dfr","dgt","dfs"}

startWord = "der", targetWord= "dfs",

Output:

3

Explanation:

The length of the smallest transformation sequence from "der" to "dfs" is 3

i,e "der" -> "dfr" -> "dfs".

beginWord = "hit" endWord = "cog"

wordList = [hot, dot, dog, lot, log, cog]

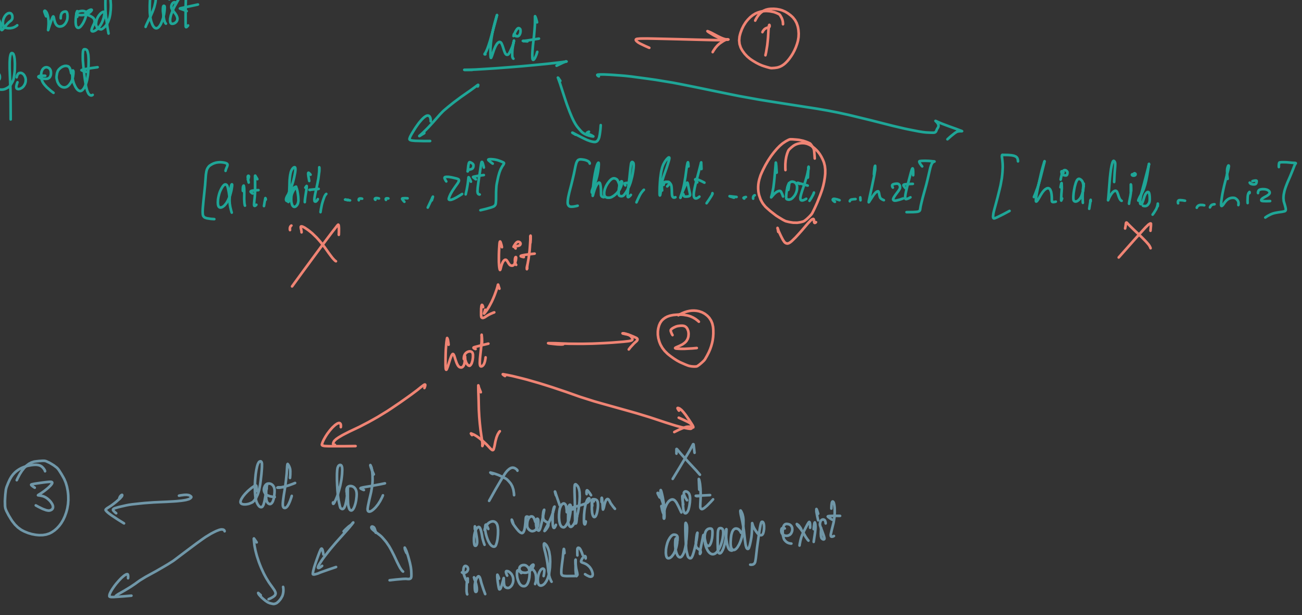
hit → hot → dot → dog → cog

M1 → brute force

→ get the start word

→ change each char to a to z & check which one is valid from the word list

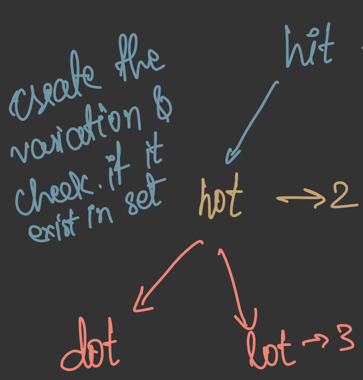
→ repeat



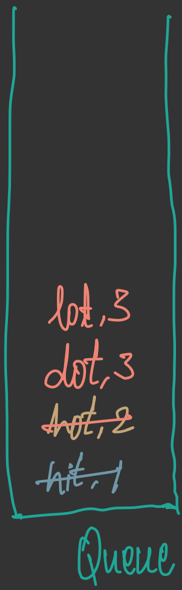
wordList = [hot, dot, dog, lot, log, cog]

beginWord = hit , endWord = cog

set = { ~~hot~~, ~~dot~~, ~~dog~~, ~~lot~~, log, cog }



, when we find a variation in set, remove this variation from set, because if we take this variation again on further level, distance will be longer & we need shortest distance.

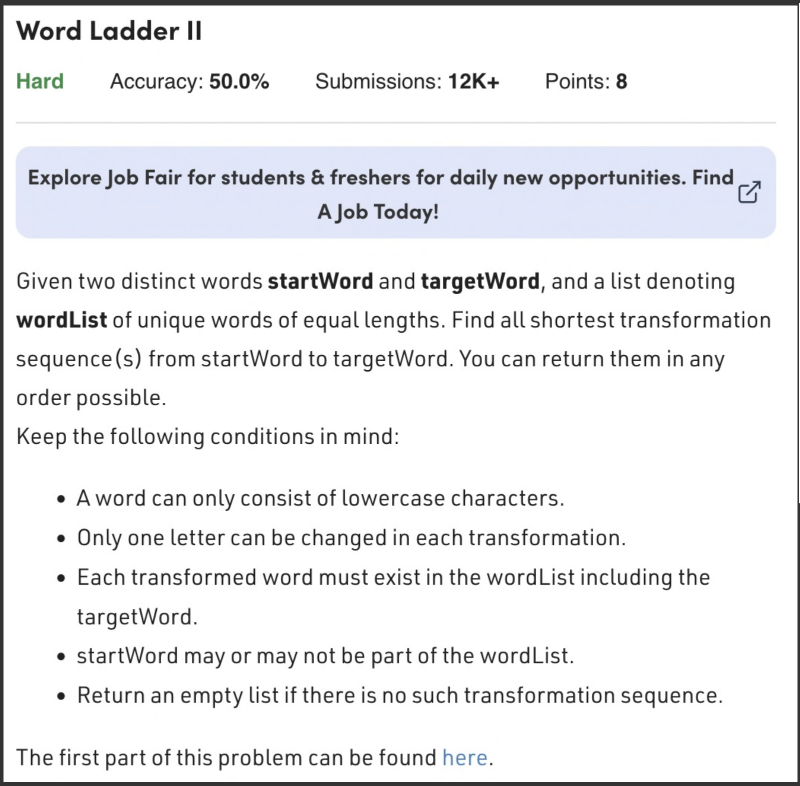


```
function wordLadderLength(startWord, targetWord, wordList) {
    // Creating a queue ds of type [{word, transitions to reach 'word'}].
    const q = [];

    // BFS traversal with pushing values in queue
    // when after a transformation, a word is found in wordList.
    q.push({ word: startWord, steps: 1 });

    // Push all values of wordList into a set
    // to make deletion from it easier and in less time complexity.
    const set = new Set(wordList);
    set.delete(startWord);

    while (q.length !== 0) {
        const { word, steps } = q.shift();

        // we return the steps as soon as
        // the first occurrence of targetWord is found.
        if (word === targetWord) {
            return steps;
        }

        for (let i = 0; i < word.length; i++) {
            // Now, replace each character of 'word' with char
            // from a-z then check if 'word' exists in wordList.
            const original = word[i];
            for (let ch = "a".charCodeAt(0); ch <= "z".charCodeAt(0); ch++) {
                const newWord =
                    word.substring(0, i) +
                    String.fromCharCode(ch) +
                    word.substring(i + 1);
                // check if it exists in the set and push it in the queue.
                if (set.has(newWord)) {
                    set.delete(newWord);
                    q.push({ word: newWord, steps: steps + 1 });
                }
            }
        }
    }
    // If there is no transformation sequence possible
    return 0;
}

const wordList = ["des", "der", "dfr", "dgt", "dfs"];
const startWord = "der";
const targetWord = "dfs";

const ans = wordLadderLength(startWord, targetWord, wordList);

console.log(ans); // 3
```

TC → O(N x word[i].length x 26)

SC → O(N)

lec 30 → word ladder 2.



Word Ladder II

Hard Accuracy: **50.0%** Submissions: **12K+** Points: **8**

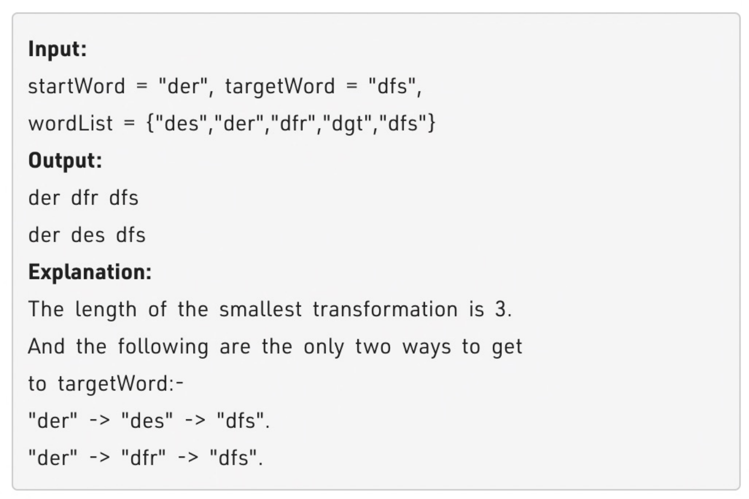
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given two distinct words **startWord** and **targetWord**, and a list denoting **wordList** of unique words of equal lengths. Find all shortest transformation sequence(s) from startWord to targetWord. You can return them in any order possible.

Keep the following conditions in mind:

- A word can only consist of lowercase characters.
- Only one letter can be changed in each transformation.
- Each transformed word must exist in the wordList including the targetWord.
- startWord may or may not be part of the wordList.
- Return an empty list if there is no such transformation sequence.

The first part of this problem can be found here.



Input:

```
startWord = "der", targetWord = "dfs",
wordList = {"des","der","dfr","dgt","dfs"}
```

Output:

```
der dfr dfs
der des dfs
```

Explanation:

The length of the smallest transformation is 3.
And the following are the only two ways to get to targetWord:-
"der" -> "des" -> "dfs".
"der" -> "dfr" -> "dfs".

wordList = [pat, bot, pot, poz, coz]

beginWord = bat endword = coz

set = [~~pat~~, ~~bot~~, ~~pot~~, ~~poz~~, coz]

① bat

bat
↙ ↓
pat bot

② (bat, pat)

pat
↓
pot

(bat, bot)

bot
↓
pot

now entire level 2 is completed
so delete pot from set

(bat, bot, pot, poz, coz) 5

(bat, pat, pot, poz, coz) 5

~~(bat, bot, pot, poz) 4~~

~~(bat, pat, pot, poz) 4~~

(bat, bot, pot) 3

(bat, pat, pot) 3

(bat, bot) 2

~~(bat, pat), 2~~

~~(bat) 1~~

queue

③ (bat, pat, pot)

pot
↓
poz

(bat, bot, pot)

pot
↓
poz

delete poz from set

```
function findSequences(beginWord, endWord, wordList) {
    const st = new Set(wordList);

    const q = [];
    q.push([beginWord]);

    const usedOnLevel = [];
    usedOnLevel.push(beginWord);
    let level = 0;

    const ans = [];
    while (q.length > 0) {
        const vec = q.shift();

        if (vec.length > level) {
            level++;
            for (const it of usedOnLevel) {
                st.delete(it);
            }
        }

        const word = vec[vec.length - 1];

        if (word === endWord) {
            if (ans.length === 0) {
                ans.push(vec);
            } else if (ans[0].length === vec.length) {
                ans.push(vec);
            }
        }
        for (let i = 0; i < word.length; i++) {
            const original = word[i];
            for (let c = "a".charCodeAt(0); c <= "z".charCodeAt(0); c++) {
                const modifiedWord =
                    word.slice(0, i) +
                    String.fromCharCode(c) +
                    word.slice(i + 1);
                if (st.has(modifiedWord)) {
                    vec.push(modifiedWord);
                    q.push([...vec]);
                    usedOnLevel.push(modifiedWord);
                    vec.pop();
                }
            }
        }
    }
    return ans;
}

function comp(a, b) {
    const x = a.join("");
    const y = b.join("");
    return x < y;
}

const wordList = ["des", "der", "dfr", "dgt", "dfs"];
const startWord = "der";
const targetWord = "dfs";
const ans = findSequences(startWord, targetWord, wordList);

if (ans.length === 0) {
    console.log(-1);
} else {
    ans.sort(comp);
    for (const sequence of ans) {
        console.log(sequence.join(" "));
        // der dfr dfs
        // der des dfs
    }
}
```

TC-> vary from examples to examples

lec 81 → word ladder 2 | optimised approach [cp way → interviewer doesn't ask]

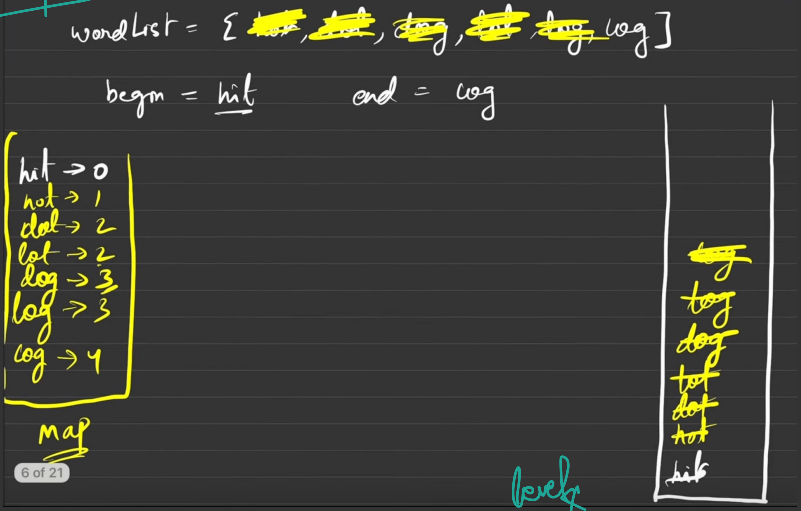
wordList = [hot, dot, dog, lot, log, cog]

begin = hit   end = cog

step1 → follow word ladder 1, & find min steps
& store the steps for each

step2 → backtrack in the map from end → begin
to get the answer

step1



wordList = [~~hot~~, ~~dot~~, ~~dog~~, ~~lot~~ ~~log~~ cog]

begin = hit   end = cog

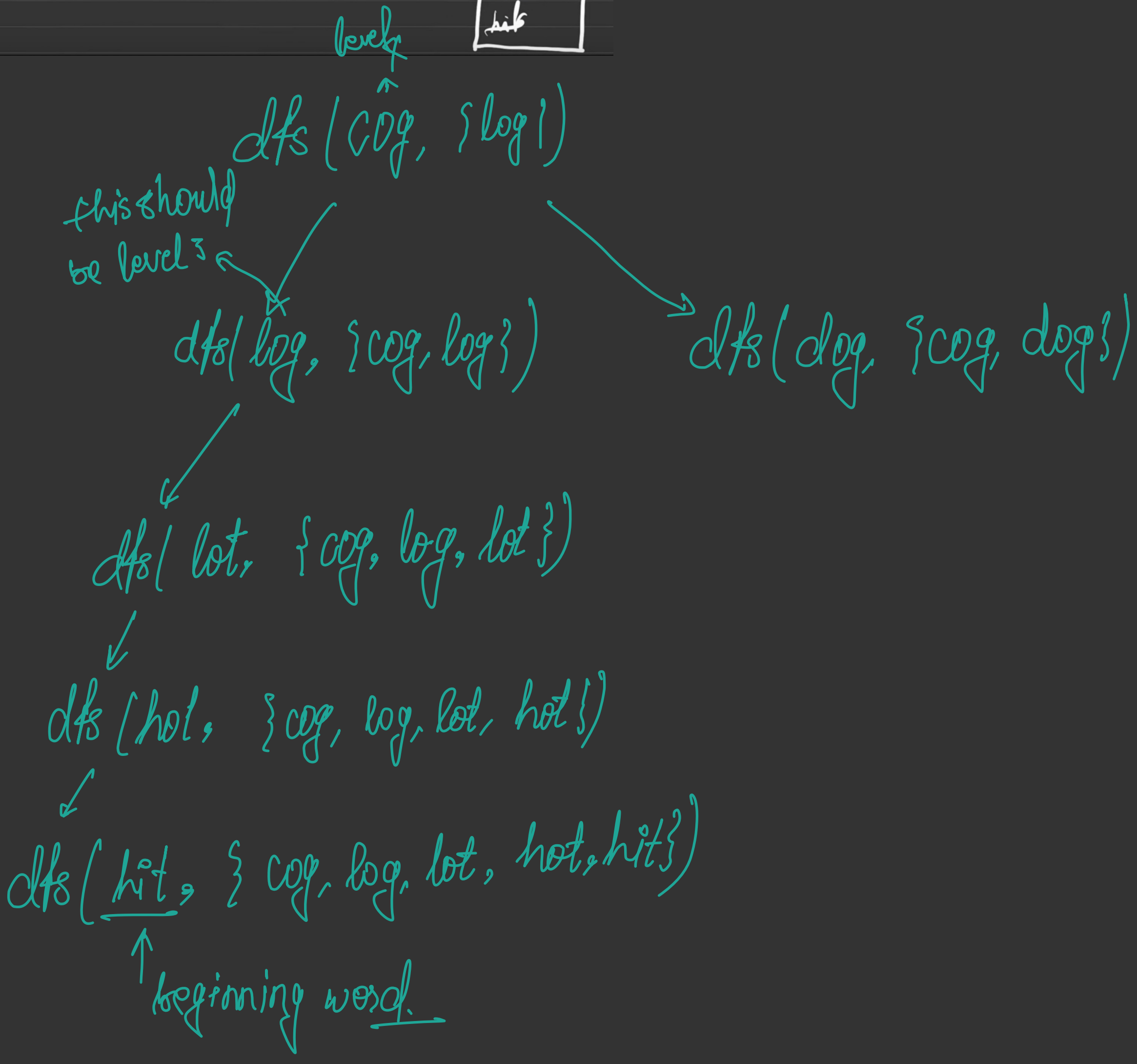
hit → 0
hot → 1
dot → 2
lot → 2
dog → 3
log → 3
cog → 4

map

~~cog~~
~~log~~
~~dog~~
~~lot~~
~~dot~~
~~hot~~
~~hit~~

6 of 21

step2



level

dfs(cog, {log})

this should be level 3

dfs(log, {cog, log})

dfs(dog, {cog, dog})

dfs(lot, {cog, log, lot})

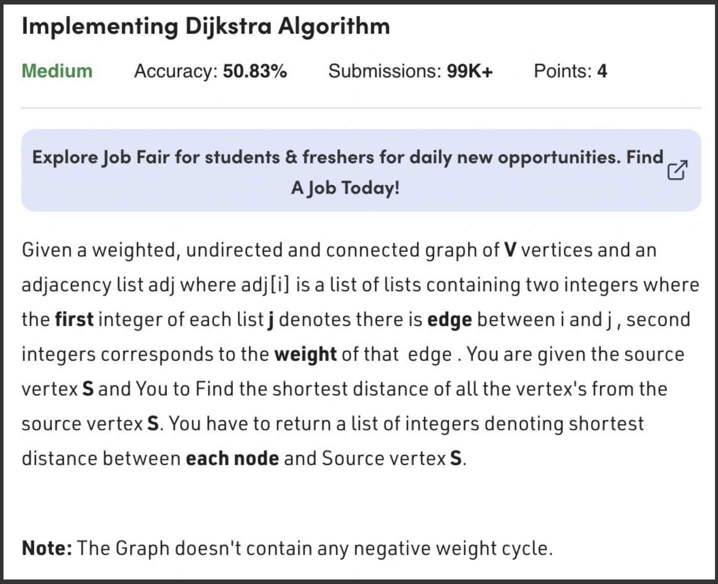
dfs(hot, {cog, log, lot, hot})

dfs(hit, {cog, log, lot, hot, hit})

beginning word.

```
const findLadders = (beginWord, endWord, wordList) => {
    const map = new Map();
    const ans = [];
    let b = beginWord;

    const dfs = (word, seq) => {
        if (word === b) {
            ans.push([...seq].reverse());
            return;
        }
        const sz = word.length;
        const steps = map.get(word);

        for (let i = 0; i < sz; i++) {
            const original = word[i];
            for (let ch = "a".charCodeAt(0); ch <= "z".charCodeAt(0); ch++) {
                word =
                    word.slice(0, i) +
                    String.fromCharCode(ch) +
                    word.slice(i + 1);
                if (map.has(word) && map.get(word) + 1 === steps) {
                    seq.push(word);
                    dfs(word, seq);
                    seq.pop();
                }
            }
            word = word.slice(0, i) + original + word.slice(i + 1);
        }
    };

    const st = new Set(wordList);
    const q = [beginWord];
    map.set(beginWord, 1);
    const sz = beginWord.length;
    st.delete(beginWord);

    while (q.length > 0) {
        const word = q.shift();
        const steps = map.get(word);

        if (word === endWord) break;

        for (let i = 0; i < sz; i++) {
            const original = word[i];
            for (let ch = "a".charCodeAt(0); ch <= "z".charCodeAt(0); ch++) {
                const newWord =
                    word.slice(0, i) +
                    String.fromCharCode(ch) +
                    word.slice(i + 1);
                if (st.has(newWord)) {
                    q.push(newWord);
                    st.delete(newWord);
                    map.set(newWord, steps + 1);
                }
            }
        }
    }

    if (map.has(endWord)) {
        const seq = [endWord];
        dfs(endWord, seq);
    }

    return ans;
};

const comp = (a, b) => {
    const x = a.join("");
    const y = b.join("");
    return x < y;
};

const wordList = ["des", "der", "dfr", "dgt", "dfs"];
const startWord = "der";
const targetWord = "dfs";
const ans = findLadders(startWord, targetWord, wordList);

if (ans.length === 0) {
    console.log(-1);
} else {
    ans.sort(comp);
    ans.forEach((path) => {
        console.log(path.join(" "));
        // der des dfs
        // der dfr dfs
    });
}
```

lec32 Dijkstra's algorithm. using priority queue, part 1 | shortest path



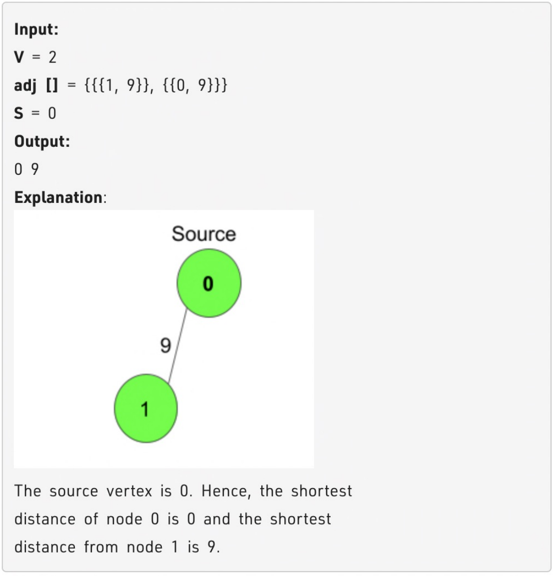
Implementing Dijkstra Algorithm

Medium Accuracy: **50.83%** Submissions: **99K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a weighted, undirected and connected graph of **V** vertices and an adjacency list adj where adj[i] is a list of lists containing two integers where the **first** integer of each list **j** denotes there is **edge** between i and j , second integers corresponds to the **weight** of that edge . You are given the source vertex **S** and You to Find the shortest distance of all the vertex's from the source vertex **S**. You have to return a list of integers denoting shortest distance between **each node** and Source vertex **S**.

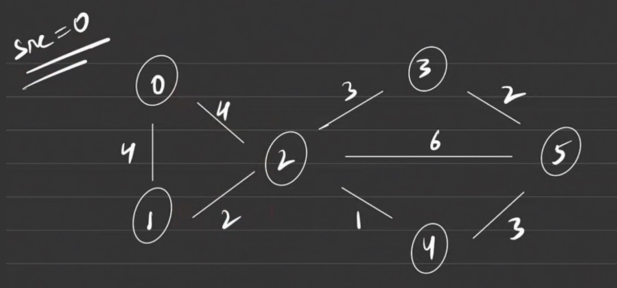
Note: The Graph doesn't contain any negative weight cycle.



Input:
V = 2
adj [] = {{{1, 9}}, {{0, 9}}}
S = 0
Output:
0 9
Explanation:

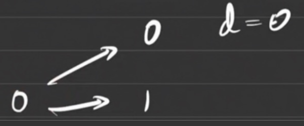
The source vertex is 0. Hence, the shortest distance of node 0 is 0 and the shortest distance from node 1 is 9.

given the src node, find the shortest path to other nodes from src



Adj. List

0 → {1, 4} {2, 4}
1 → {0, 4} {2, 2}
2 → {0,4} {1, 2} {3,3} {4,1} {5,6}
3 → {2, 3} {5, 2}
4 → {2, 1} {5, 3}
5 → {2, 6} {3, 2} {4, 3}

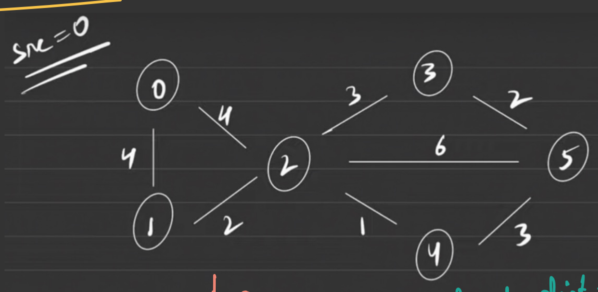


dist(1) = min(4, (4+2)) = 4

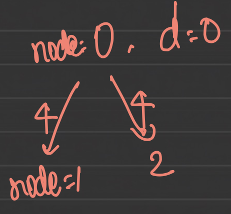
⇒ this algo can be implemented using two ways →

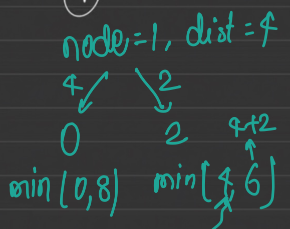
① priority queue
fastest ← ② set data structure
③ queue

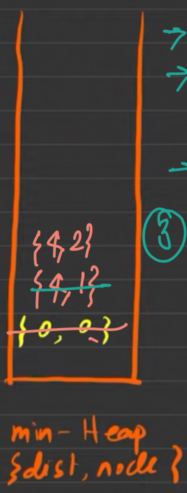
Priority Queue.



① put dist. src in PQ initially.
② shift from PQ
→ visit it's neighbours
→ calculate & compare distance
→ update the distance
③ put the nodes in PQ.
if not in PQ already from current node







dist[] =

	4	4			
0	~~∞~~	~~∞~~	∞	∞	∞
0	1	2	3	4	5

```
const { MinPriorityQueue } = require("@datastructures-js/priority-queue");

const dijkstra = (V, adj, S) => {
    const pq = new MinPriorityQueue({ priority: (pair) => pair[0] });
    const distTo = Array(V).fill(Number.MAX_SAFE_INTEGER);

    distTo[S] = 0;
    pq.enqueue([0, S]);
    pq; //?

    while (!pq.isEmpty()) {
        const [dis, node] = pq.dequeue().element;
        for (const it of adj[node]) {
            const [v, w] = it;
            if (dis + w < distTo[v]) {
                distTo[v] = dis + w;
                pq.enqueue([dis + w, v]);
            }
        }
    }

    return distTo;
};

const V = 3;
const S = 2;
const adj = [
    [
        [1, 1],
        [2, 6],
    ],
    [
        [2, 3],
        [0, 1],
    ],
    [
        [1, 3],
        [0, 6],
    ],
];

const res = dijkstra(V, adj, S);

for (let i = 0; i < V; i++) {
    console.log(res[i]);
}
```

TODO → MinPriorityQueue in JS with objects.

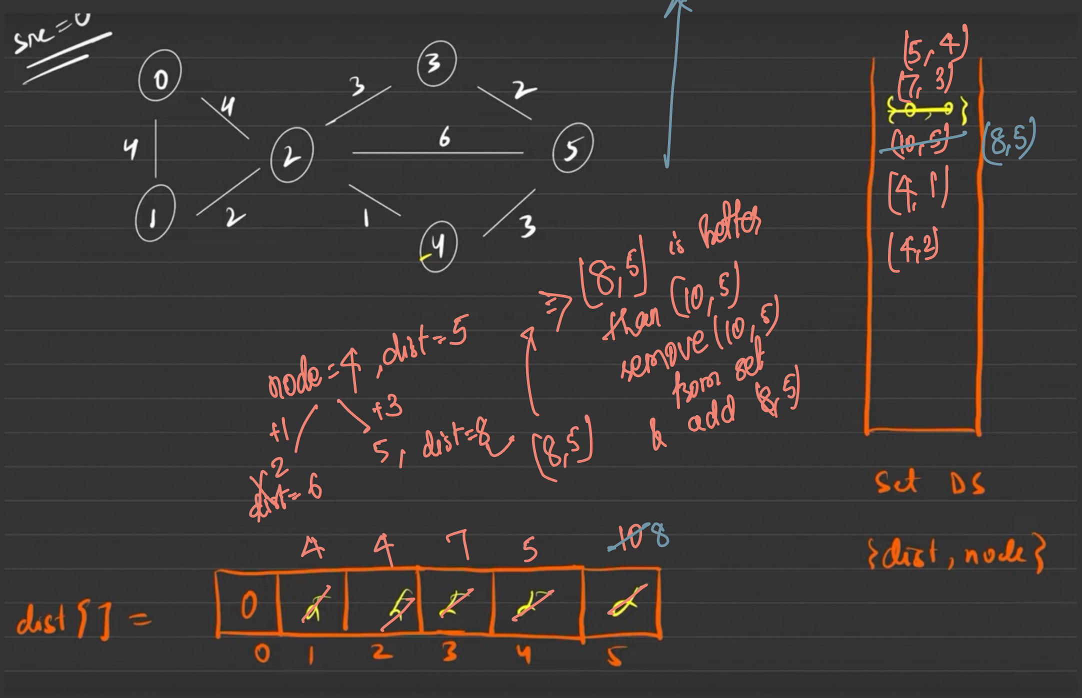
djikstra does not works in → negative weight
→ negative cycle

TC → E log V → num of nodes
↑ edges

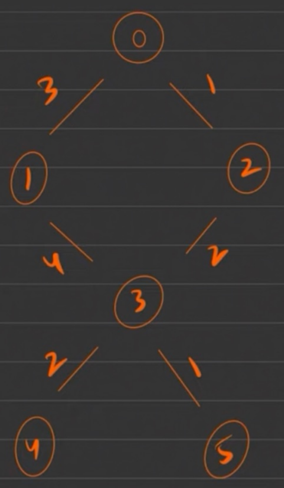
lec 83 using Set → this sol^n not possible in JS

⇒ set always stores data in ascending order.
↳ not true in JavaScript

erase already existing path



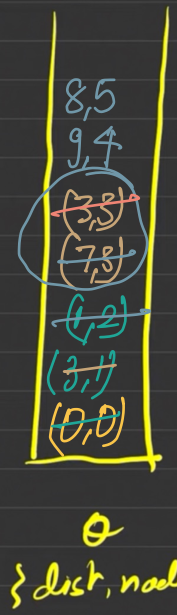
lec34 Why Priority Queue & not queue for Dijkstra algo



adj list

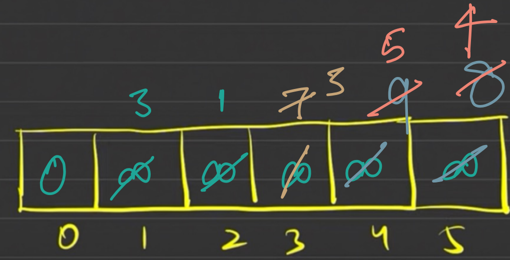
0 → {1,3} {2,1}
1 → {0,3} {3,4}
2 → {0,1} {3,2}
3 → {1,4} {2,2} {4,2} {5,1}
4 → {3,2}
5 → {3,1}

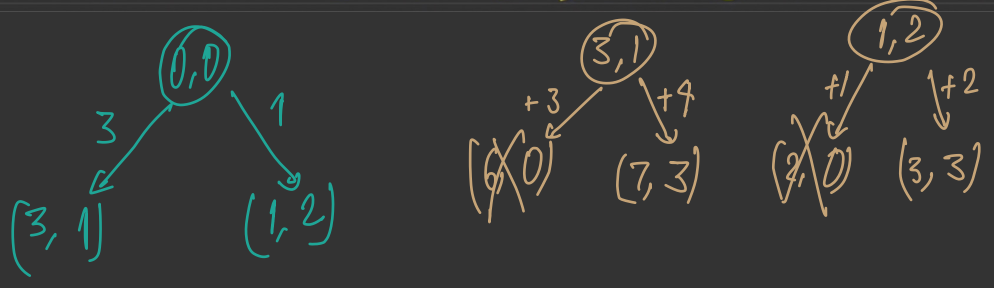
src = 0

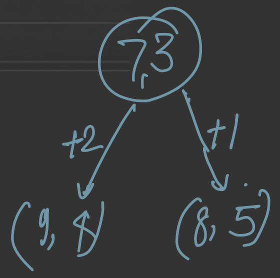


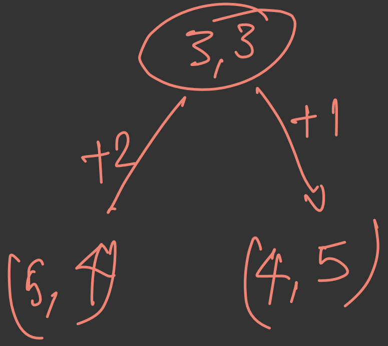
if it were PQ we would have dealt with (7,3) 1st but here we have to deal with both

dist[] =



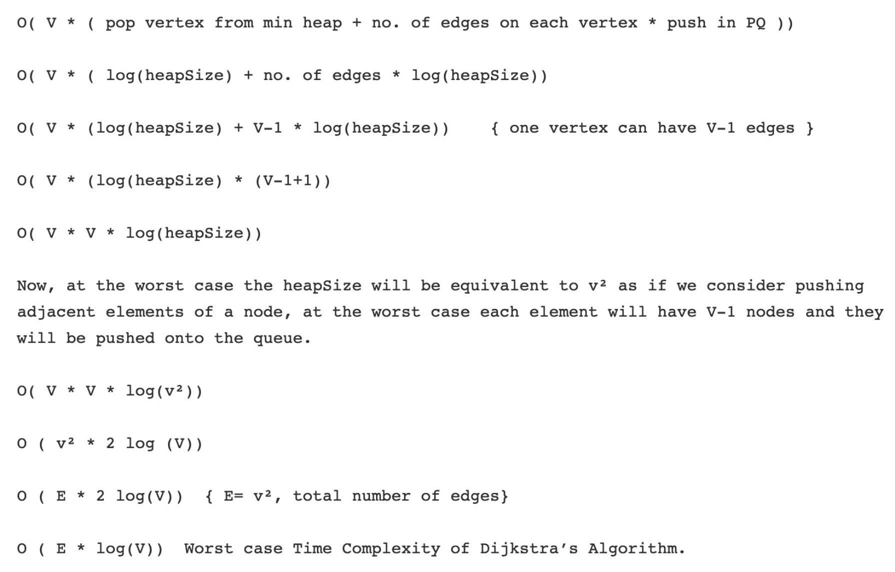






⇒ using Queue is brute force approach, because it is trying each possible path

TC in PQ



O(V * (pop vertex from min heap + no. of edges on each vertex * push in PQ))

O(V * (log(heapSize) + no. of edges * log(heapSize))

O(V * (log(heapSize) + V-1 * log(heapSize))   { one vertex can have V-1 edges }

O(V * (log(heapSize) * (V-1+1))

O(V * V * log(heapSize))

Now, at the worst case the heapSize will be equivalent to v^2 as if we consider pushing adjacent elements of a node, at the worst case each element will have V-1 nodes and they will be pushed onto the queue.

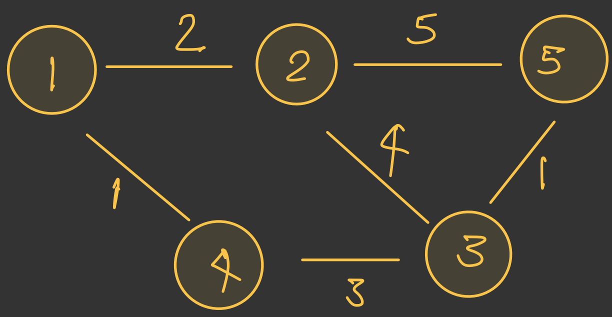
O(V * V * log(v^2))

O (v^2 * 2 log (V))

O (E * 2 log(V)) { E= v^2, total number of edges}

O (E * log(V)) Worst case Time Complexity of Dijkstra's Algorithm.

lec 35 → print shortest path | Dijkstra's algo



src = 1

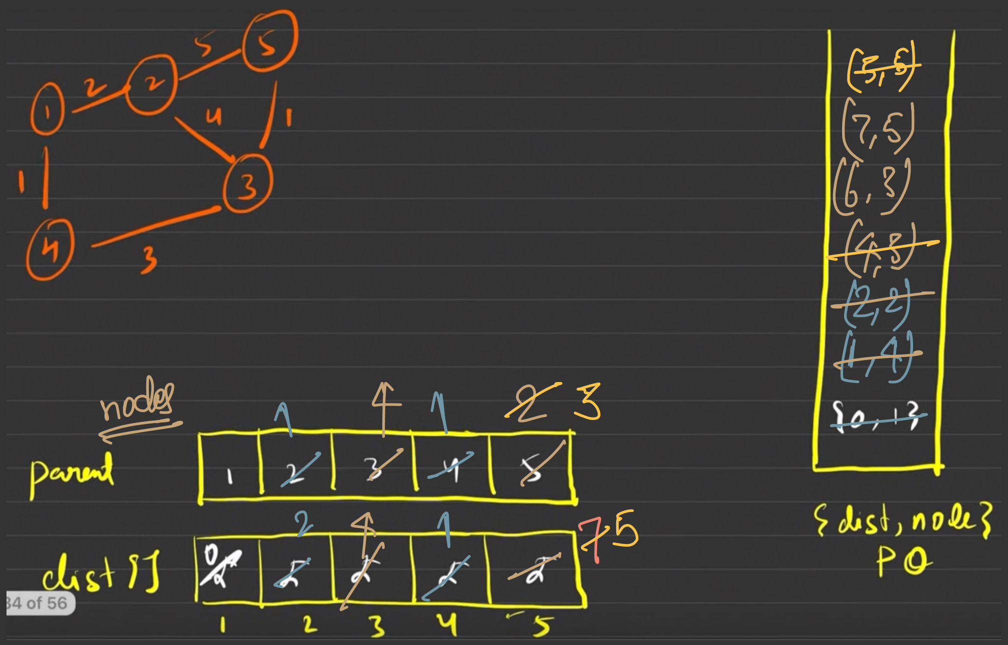
dist = 5

1 → 4 → 3 → 5

[1, 4, 3, 5] ans

if not possible return [-1]

⇒ remember where I am coming from

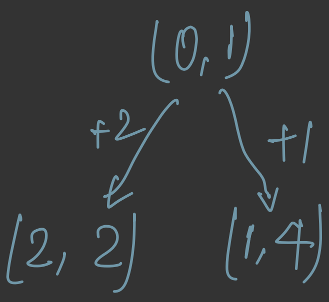


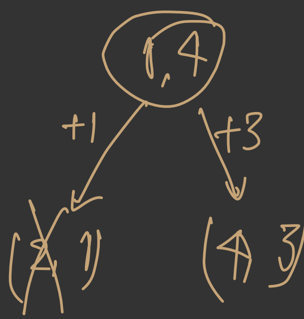
	1	2	3	4	5
parent	1	~~2~~ 1	~~3~~ 4	~~4~~ 1	~~5~~ ~~2~~ 3
dist[]	~~∞~~ 0	~~∞~~ 2	~~∞~~ 4	~~∞~~ 1	~~∞~~ 7 5

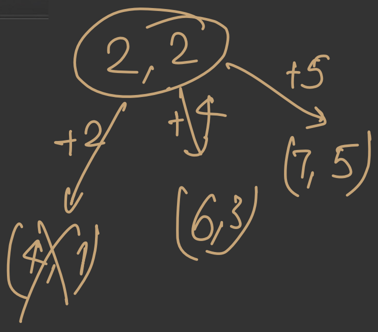
~~(5, 5)~~
(7, 5)
(6, 3)
~~(4, 3)~~
~~(2, 2)~~
~~(1, 4)~~
~~{0, 1}~~

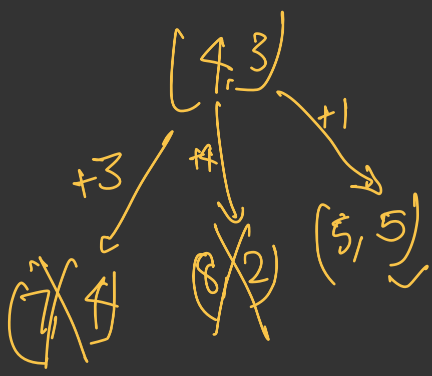
{dist, node}
PQ

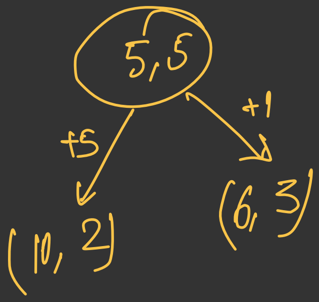
min PQ.







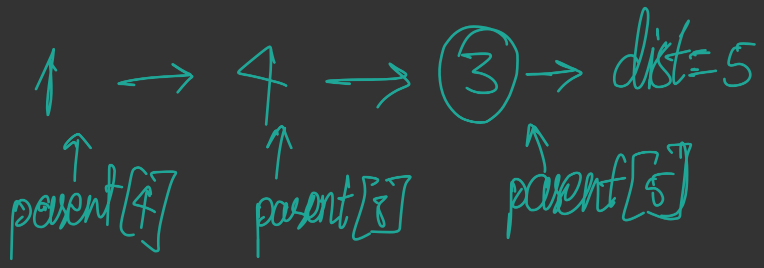




⇒ when PQ is empty we got the shortest dist = 5

⇒ using parent, backtrack the path of shortest final parent = | 1 | 1 | 4 | 1 | 3 |

↑ 5



```javascript
const shortestPath = (n, m, edges) => {
    const adj = Array(n + 1)
        .fill()
        .map(() => []);
    edges.forEach((edge) => {
        adj[edge[0]].push([edge[1], edge[2]]);
        adj[edge[1]].push([edge[0], edge[2]]);
    });

    const pq = [];
    const dist = Array(n + 1).fill(1e9);
    const parent = Array.from({ length: n + 1 }, (_, i) => i);

    dist[1] = 0;
    pq.push([0, 1]);

    while (pq.length) {
        pq.sort(([a], [b]) => a - b);
        const [dis, node] = pq.shift();

        adj[node].forEach(([adjNode, edW]) => {
            if (dis + edW < dist[adjNode]) {
                dist[adjNode] = dis + edW;
                pq.push([dis + edW, adjNode]);
                parent[adjNode] = node;
            }
        });
    }

    if (dist[n] === 1e9) {
        return [-1];
    }

    const path = [];
    let node = n;
    while (parent[node] !== node) {
        path.push(node);
        node = parent[node];
    }
    path.push(1);
    path.reverse();

    return path;
};

const V = 5;
const E = 6;
const edges = [
    [1, 2, 2],
    [2, 5, 5],
    [2, 3, 4],
    [1, 4, 1],
    [4, 3, 3],
    [3, 5, 1],
];

const path = shortestPath(V, E, edges);
console.log(path.join(" ")); // 1 4 3 5
```

TODO:-> instead of queue & sorting
use Min PriorityQueue

TC -> O(E log V)

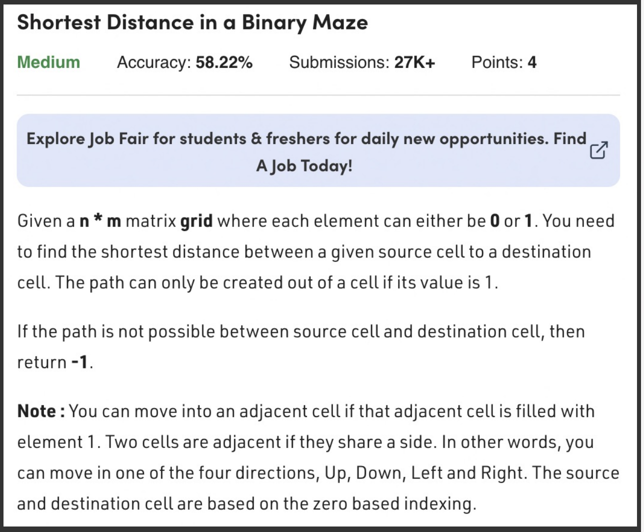
E -> edges
V -> nodes

SC -> O(|E|+|V|) + O(|V|)

↑ pq & dist array

↑ final path

lec36. shortest distance in a binary maze



Shortest Distance in a Binary Maze

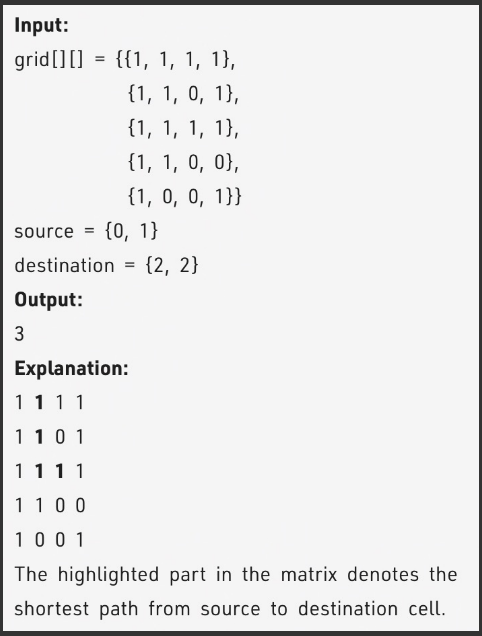
Medium Accuracy: **58.22%** Submissions: **27K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a **n * m** matrix **grid** where each element can either be **0** or **1**. You need to find the shortest distance between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

If the path is not possible between source cell and destination cell, then return **-1**.

Note : You can move into an adjacent cell if that adjacent cell is filled with element 1. Two cells are adjacent if they share a side. In other words, you can move in one of the four directions, Up, Down, Left and Right. The source and destination cell are based on the zero based indexing.



Input:

```
grid[][] = {{1, 1, 1, 1},
            {1, 1, 0, 1},
            {1, 1, 1, 1},
            {1, 1, 0, 0},
            {1, 0, 0, 1}}
source = {0, 1}
destination = {2, 2}
```

Output:

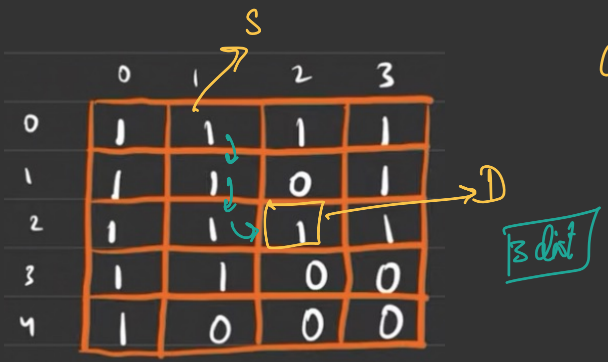
3

Explanation:

```
1 1 1 1
1 1 0 1
1 1 1 1
1 1 0 0
1 0 0 1
```

The highlighted part in the matrix denotes the shortest path from source to destination cell.

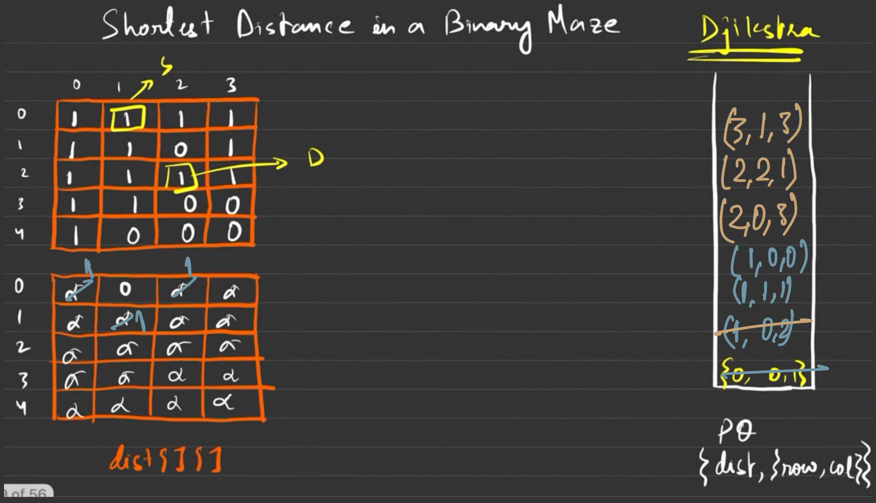
src = [0, 1]
dest = [2, 2]



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

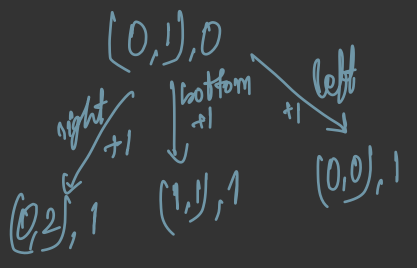
can move in four dirn

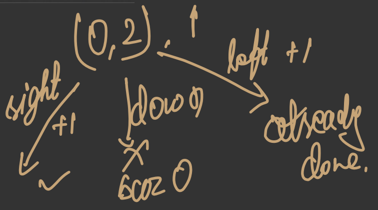
Djikstra algo

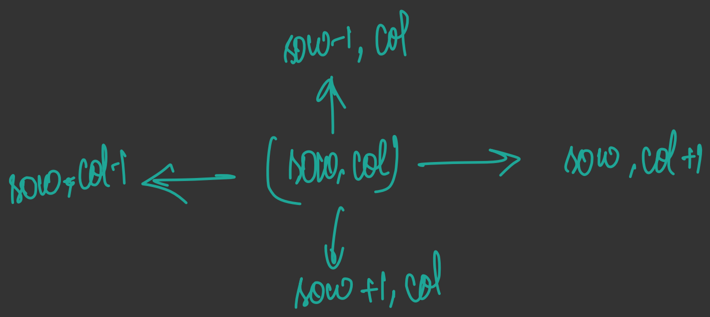


here, queue already store the values in increasing order due to unit distance

so, we don't need a priority queue.







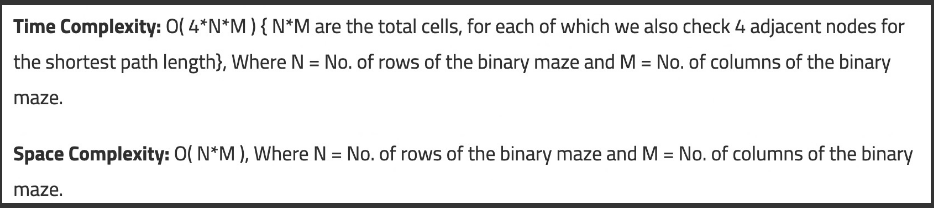
adjList

drc[] = [-1, 0, 1, 0]

dc[] = [0, 1, 0, -1]

index → 0, 1, 2, 3

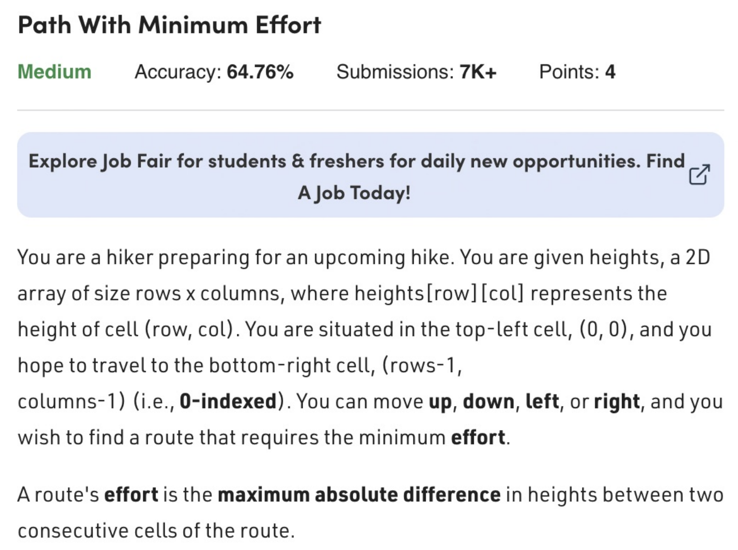
```javascript
function shortestPath(grid, source, destination) {
    if (source[0] === destination[0] && source[1] === destination[1]) {
        return 0;
    }

    const q = [];
    const n = grid.length;
    const m = grid[0].length;

    const dist = Array.from({ length: n }, () =>
        Array.from({ length: m }, () => 1e9)
    );
    dist[source[0]][source[1]] = 0;
    q.push([0, source]);

    const dr = [-1, 0, 1, 0];
    const dc = [0, 1, 0, -1];

    while (q.length > 0) {
        const [dis, [r, c]] = q.shift();

        for (let i = 0; i < 4; i++) {
            const newr = r + dr[i];
            const newc = c + dc[i];

            if (
                newr >= 0 &&
                newr < n &&
                newc >= 0 &&
                newc < m &&
                grid[newr][newc] === 1 &&
                dis + 1 < dist[newr][newc]
            ) {
                dist[newr][newc] = 1 + dis;

                if (newr === destination[0] && newc === destination[1]) {
                    return dis + 1;
                }
                q.push([1 + dis, [newr, newc]]);
            }
        }
    }
    return -1;
}

const source = [0, 1];
const destination = [2, 2];

const grid = [
    [1, 1, 1, 1],
    [1, 1, 0, 1],
    [1, 1, 1, 1],
    [1, 1, 0, 0],
    [1, 0, 0, 1],
];

const res = shortestPath(grid, source, destination);
console.log(res); // 3
```



Time Complexity: O(4*N*M) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the shortest path length}, Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M), Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

lec37 Path with min efforts



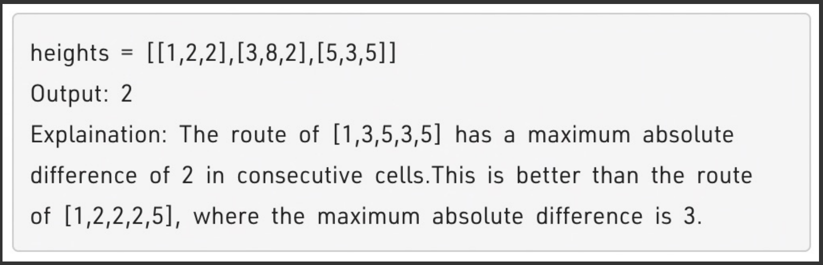
Path With Minimum Effort

Medium Accuracy: **64.76%** Submissions: **7K+** Points: **4**

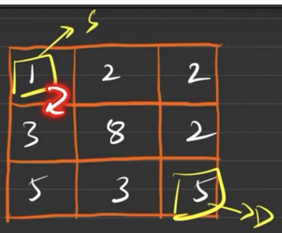
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

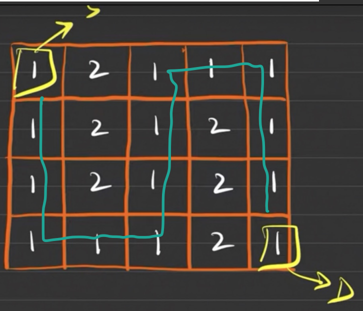
A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.



heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2
Explaination: The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.



1	2	2
3	8	2
5	3	5



1	2	1	1	1
1	2	1	2	1
1	2	1	2	1
1	1	1	2	1

path effort = 0

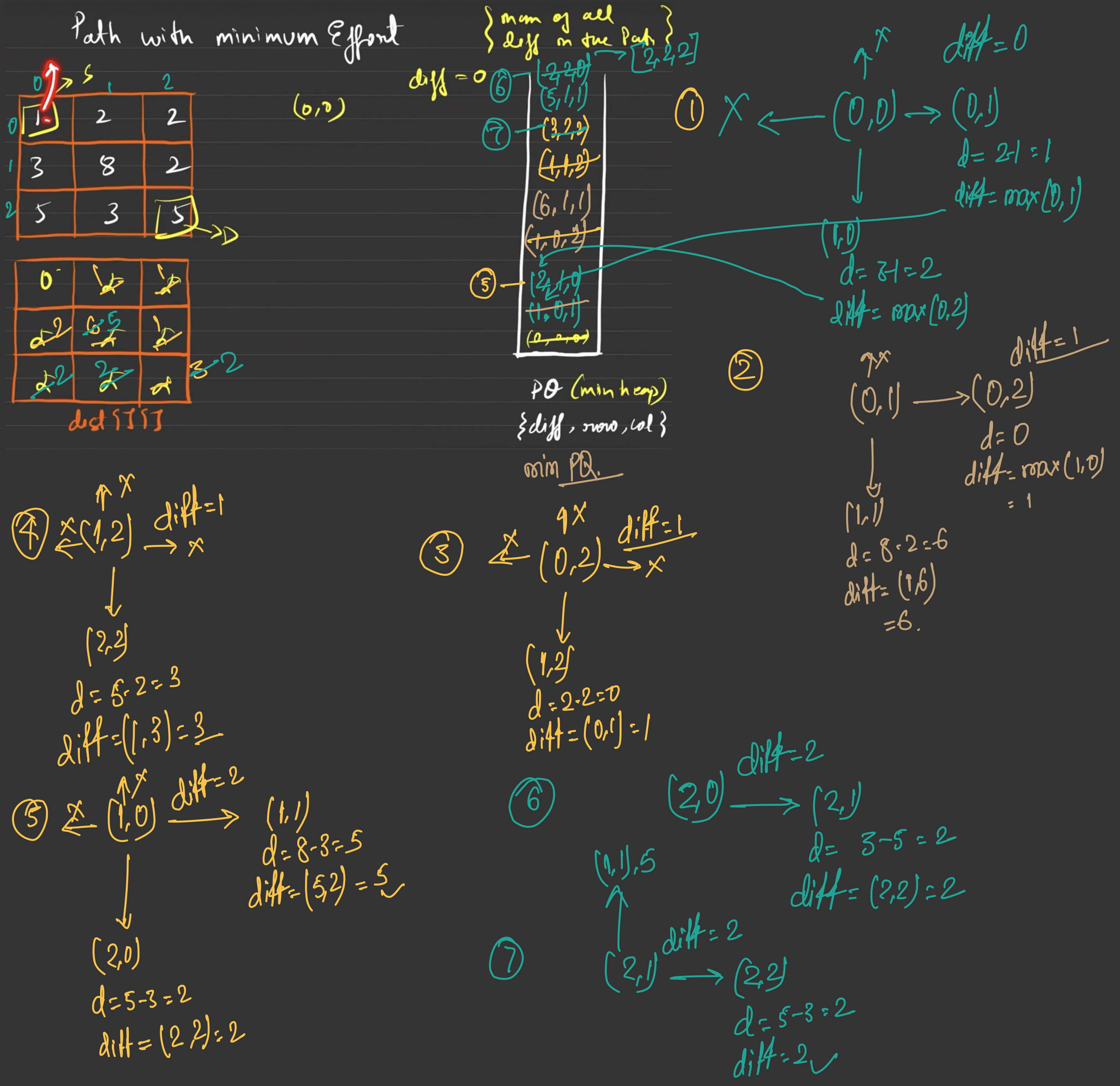
Dijkstra algo

1 2 2 2 5
1 0 0 3 = max → 3 effort

1 3 8 2 5
2 5 6 3 → max → 6 effort

1 3 5 3 5
2 2 2 2 → max → 2 effort
↳ min = 2 ans = 2

Path with minimum Effort



Time Complexity: O(4*N*M * log(N*M)) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the minimum effort and additional log(N*M) for insertion-deletion operations in a priority queue }

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M) { Distance matrix containing N*M cells + priority queue in the worst case containing all the nodes (N*M) }.

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

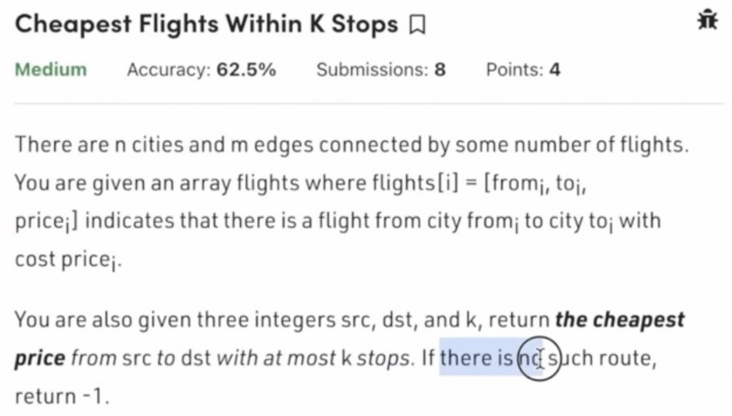
```javascript
const { MinPriorityQueue } = require("@datastructures-js/priority-queue");

function minimumEffort(heights) {
    const pq = new MinPriorityQueue({ priority: (cell) => cell.diff });
    const n = heights.length;
    const m = heights[0].length;

    const dist = Array.from({ length: n }, () => Array(m).fill(1e9));
    dist[0][0] = 0;
    pq.enqueue({ diff: 0, row: 0, col: 0 });

    const dr = [-1, 0, 1, 0];
    const dc = [0, 1, 0, -1];

    while (!pq.isEmpty()) {
        const it = pq.dequeue().element;
        const { diff, row, col } = it;

        if (row === n - 1 && col === m - 1) {
            return diff;
        }

        for (let i = 0; i < 4; i++) {
            const newr = row + dr[i];
            const newc = col + dc[i];

            if (newr >= 0 && newc >= 0 && newr < n && newc < m) {
                const newEffort = Math.max(
                    Math.abs(heights[row][col] - heights[newr][newc]),
                    diff
                );

                if (newEffort < dist[newr][newc]) {
                    dist[newr][newc] = newEffort;
                    pq.enqueue({ diff: newEffort, row: newr, col: newc });
                }
            }
        }
    }

    return 0;
}

const heights = [
    [1, 2, 2],
    [3, 8, 2],
    [5, 3, 5],
];

const ans = minimumEffort(heights);

console.log(ans);
```

TODO

MinPriorityQueue →

4x

lec 35:- cheapest flights within K stops

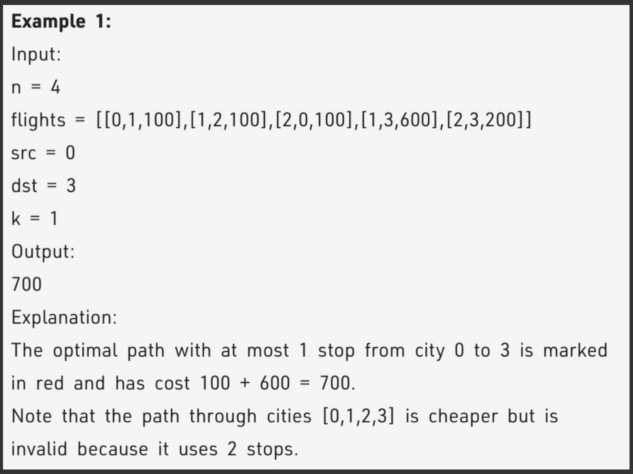


Cheapest Flights Within K Stops

Medium Accuracy: **62.5%** Submissions: **8** Points: **4**

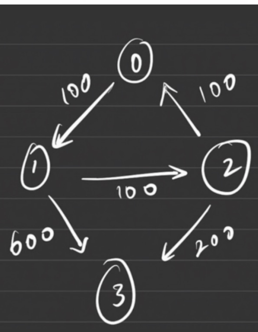
There are n cities and m edges connected by some number of flights. You are given an array flights where flights[i] = [from$_i$, to$_i$, price$_i$] indicates that there is a flight from city from$_i$ to city to$_i$ with cost price$_i$.

You are also given three integers src, dst, and k, return ***the cheapest price*** *from* src *to* dst *with at most* k *stops.* If there is no such route, return -1.



Example 1:

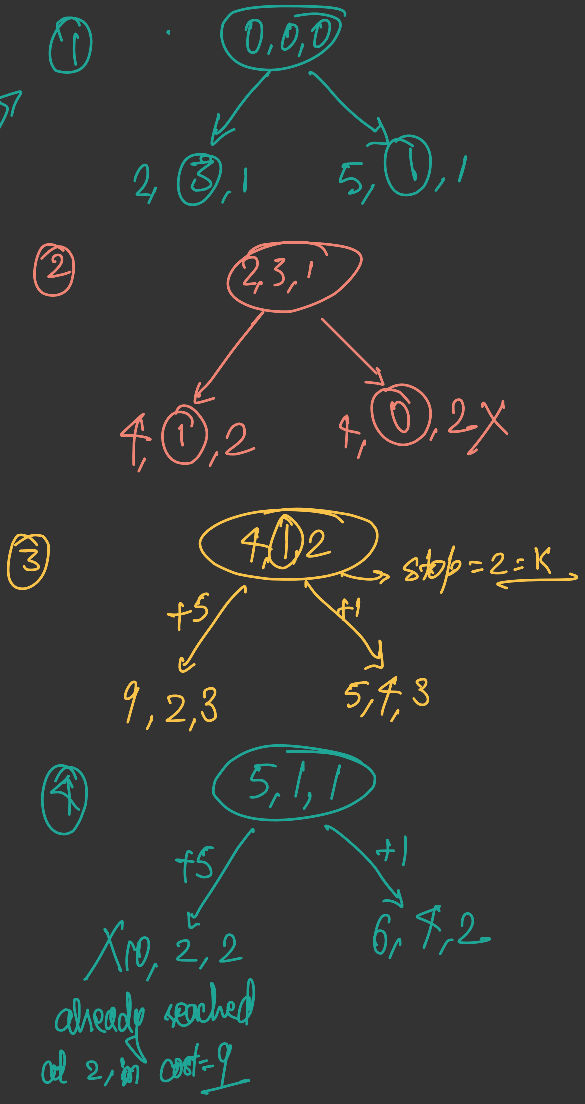
```
Input:
n = 4
flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]
src = 0
dst = 3
k = 1
Output:
700
Explanation:
The optimal path with at most 1 stop from city 0 to 3 is marked
in red and has cost 100 + 600 = 700.
Note that the path through cities [0,1,2,3] is cheaper but is
invalid because it uses 2 stops.
```



src = 0 dst = 3 k = 1

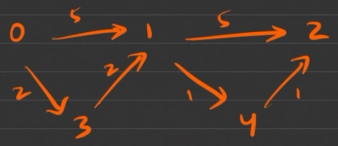
src

700



1. Create Graph ~~Cheapest flight within K steps~~

src = 0
dst = 2
k = 2



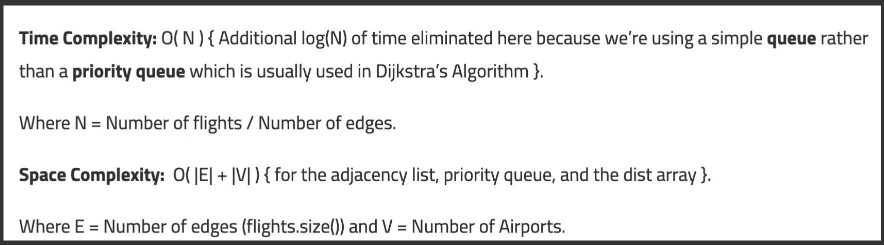
dist[] =	0 ~~∞~~	~~5~~ 4 ~~∞~~	9 ~~∞~~	2 ~~∞~~	5 ~~∞~~
	0	1	2	3	4

(5, 4, 3)
(9, 2, 3)
~~(4, 1, 2)~~
~~(2, 3, 1)~~
~~(5, 1, 1)~~
~~(0, 0, 0)~~

{dist, node, stops}
PQ
minHeap

Priority Queue doesn't work here because it is not considering stops

```javascript
function cheapestFlight(n, flights, src, dst, K) {
    // Create the adjacency list to depict airports and flights in
    // the form of a graph.
    const adj = new Array(n).fill(null).map(() => []);
    for (const flight of flights) {
        adj[flight[0]].push([flight[1], flight[2]]);
    }

    // Create a queue which stores the node and their distances from the
    // source in the form of {stops, {node, dist}} with 'stops' indicating
    // the no. of nodes between src and current node.
    const q = [];

    q.push([0, [src, 0]]);

    // Distance array to store the updated distances from the source.
    const dist = new Array(n).fill(1e9);
    dist[src] = 0;

    // Iterate through the graph using a queue like in Dijkstra with
    // popping out the element with min stops first.
    while (q.length > 0) {
        const [stops, [node, cost]] = q.shift();

        // We stop the process as soon as the limit for the stops reaches.
        if (stops > K) continue;

        for (const [adjNode, edW] of adj[node]) {
            // We only update the queue if the new calculated dist is
            // less than the prev and the stops are also within limits.
            if (cost + edW < dist[adjNode] && stops <= K) {
                dist[adjNode] = cost + edW;
                q.push([stops + 1, [adjNode, cost + edW]]);
            }
        }
    }
    // If the destination node is unreachable return '-1'
    // else return the calculated dist from src to dst.
    if (dist[dst] === 1e9) return -1;
    return dist[dst];
}

// Driver Code
const n = 4,
    src = 0,
    dst = 3,
    K = 1;

const flights = [
    [0, 1, 100],
    [1, 2, 100],
    [2, 0, 100],
    [1, 3, 600],
    [2, 3, 200],
];

const ans = cheapestFlight(n, flights, src, dst, K);
console.log(ans); // 700
```



Time Complexity: O(N) { Additional log(N) of time eliminated here because we're using a simple **queue** rather than a **priority queue** which is usually used in Dijkstra's Algorithm }.

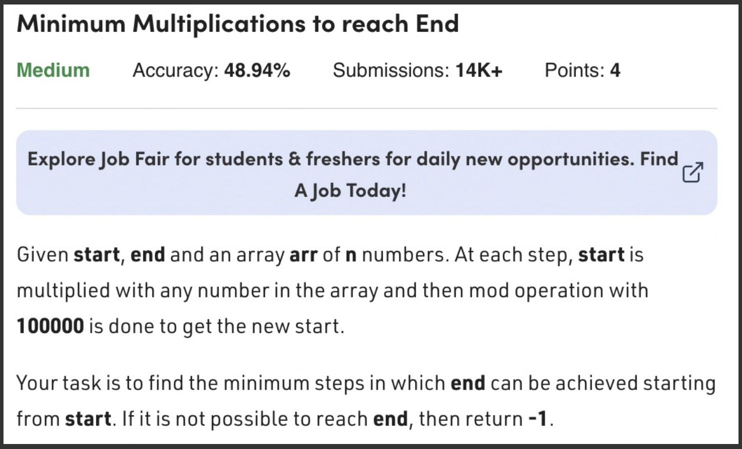
Where N = Number of flights / Number of edges.

Space Complexity: O(|E| + |V|) { for the adjacency list, priority queue, and the dist array }.

Where E = Number of edges (flights.size()) and V = Number of Airports.

---

lec84. min multiplication to reach end.



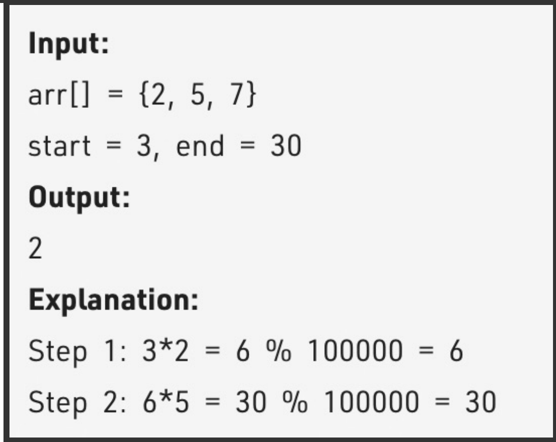
Minimum Multiplications to reach End

Medium Accuracy: **48.94%** Submissions: **14K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given **start**, **end** and an array **arr** of **n** numbers. At each step, **start** is multiplied with any number in the array and then mod operation with **100000** is done to get the new start.

Your task is to find the minimum steps in which **end** can be achieved starting from **start**. If it is not possible to reach **end**, then return **-1**.



Input:

arr[] = {2, 5, 7}

start = 3, end = 30

Output:

2

Explanation:

Step 1: 3*2 = 6 % 100000 = 6

Step 2: 6*5 = 30 % 100000 = 30

MOD = 100000 = 10^5

start = 3   end = 30

arr[] = {2, 5, 7}

start = 3 $\xrightarrow{\times 2}$ 6 $\xrightarrow{\times 5}$ 30

266178 % 10^5

start = 7   end = 66175

arr[] = {3, 4, 65}

7 × 3 = 21

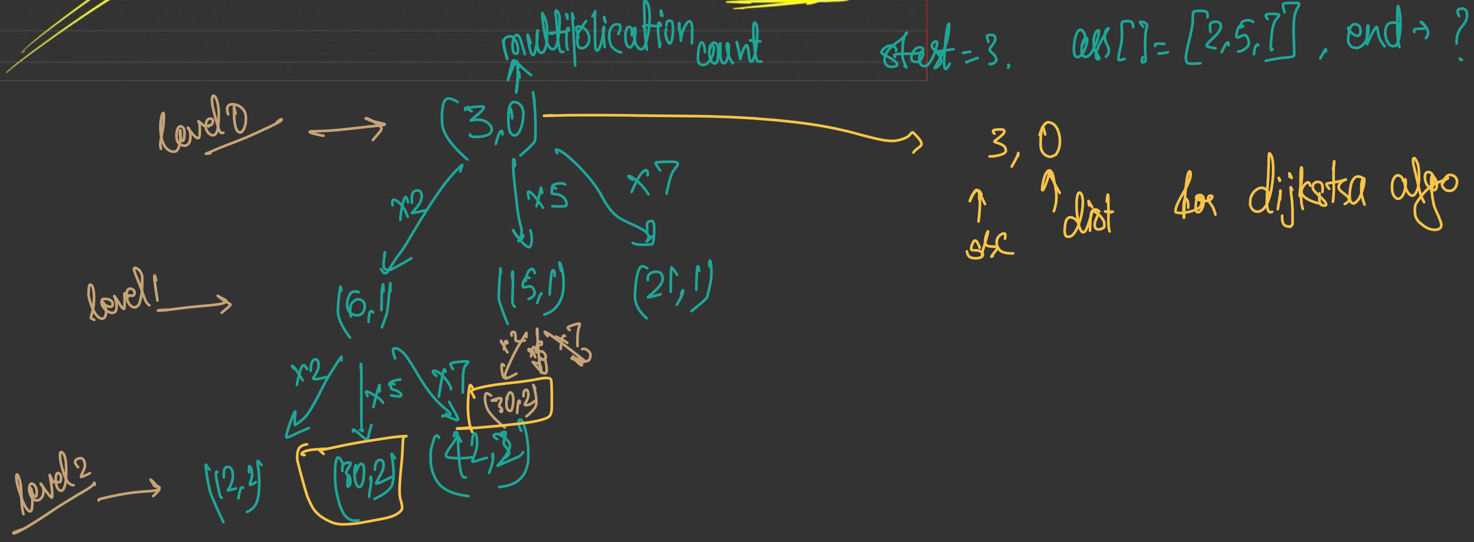
21×3 = 63

63×65 = 4095

4095×65 = 266175

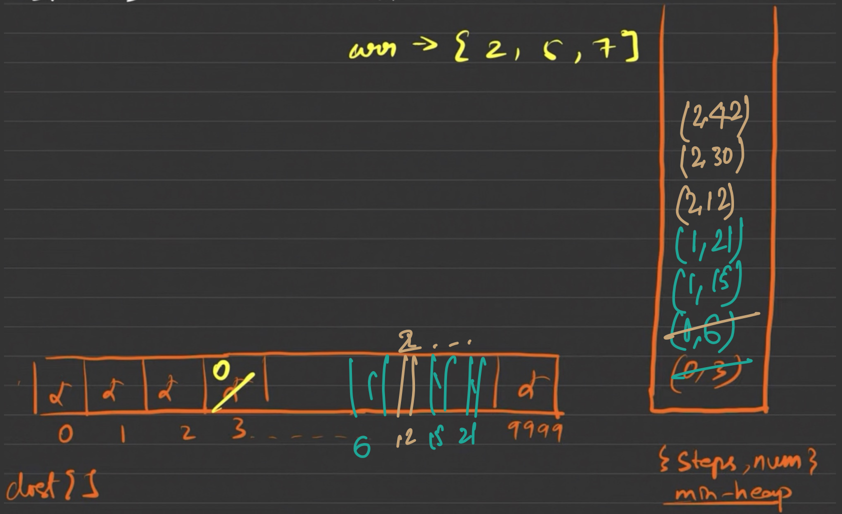
= 66175

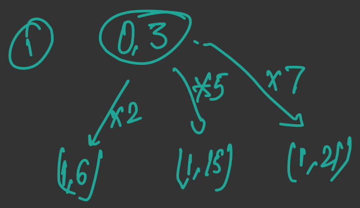
start = 3. arr[] = [2,5,7], end → ?

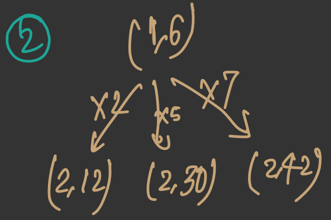


start = 3   end = 75   mod = 100000

arr → {2, 5, 7]



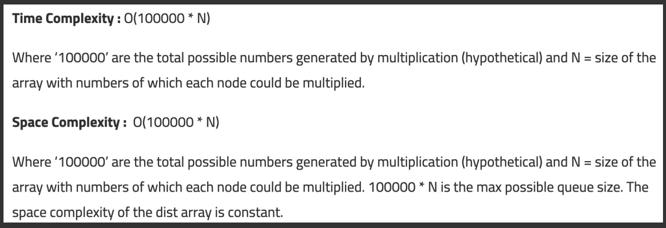




repeat

note → if something has already added in the PQ, there is no need to add it again with higher steps/distance

```javascript
function minimumMultiplications(arr, start, end) {
    // Create a queue for storing the numbers as a result of multiplication
    // of the numbers in the array and the start number.
    let q = [];
    q.push([start, 0]);

    // Create a dist array to store the no. of multiplications to reach
    // a particular number from the start number.
    let dist = Array(100000).fill(1e9);
    dist[start] = 0;
    let mod = 100000;

    // Multiply the start no. with each of numbers in the arr
    // until we get the end no.
    while (q.length) {
        let [node, steps] = q.shift();

        for (let item of arr) {
            let num = (item * node) % mod;

            // If the no. of multiplications are less than before
            // in order to reach a number, we update the dist array.
            if (steps + 1 < dist[num]) {
                dist[num] = steps + 1;

                // Whenever we reach the end number
                // return the calculated steps
                if (num === end) return steps + 1;
                q.push([num, steps + 1]);
            }
        }
    }
    // If the end no. is unattainable.
    return -1;
}

// Driver Code.
let start = 3, end = 30;
let arr = [2, 5, 7];

let ans = minimumMultiplications(arr, start, end);
console.log(ans); // 2
```



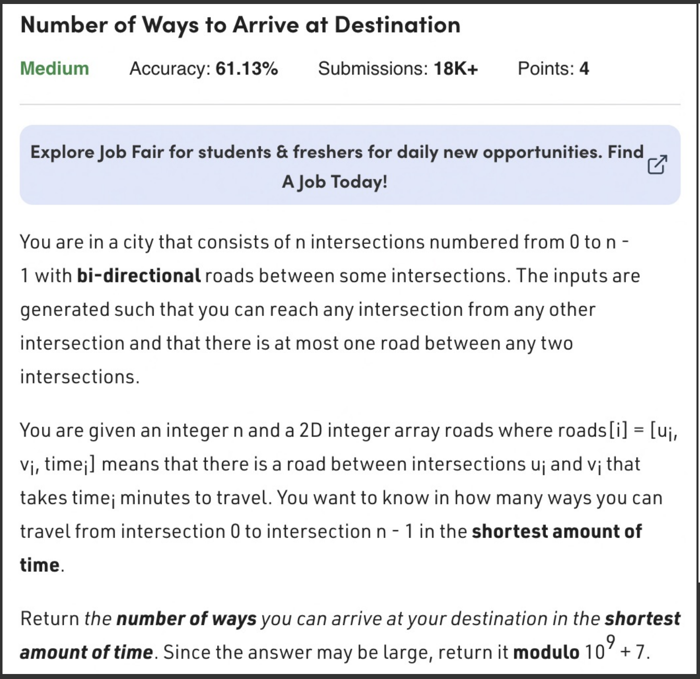
Time Complexity : O(100000 * N)

Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied.

Space Complexity : O(100000 * N)

Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied. 100000 * N is the max possible queue size. The space complexity of the dist array is constant.

lec40 number of ways to arrive at destination



Number of Ways to Arrive at Destination

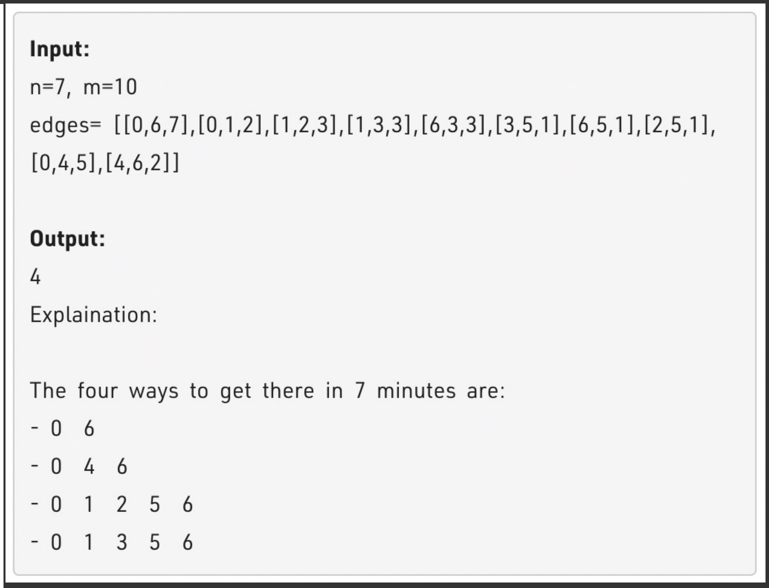
Medium Accuracy: **61.13%** Submissions: **18K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

You are in a city that consists of n intersections numbered from 0 to n - 1 with **bi-directional** roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.

You are given an integer n and a 2D integer array roads where roads[i] = [u_i, v_i, $time_i$] means that there is a road between intersections u_i and v_i that takes $time_i$ minutes to travel. You want to know in how many ways you can travel from intersection 0 to intersection n - 1 in the **shortest amount of time**.

Return *the* ***number of ways*** *you can arrive at your destination in the* ***shortest amount of time***. Since the answer may be large, return it **modulo** $10^9 + 7$.



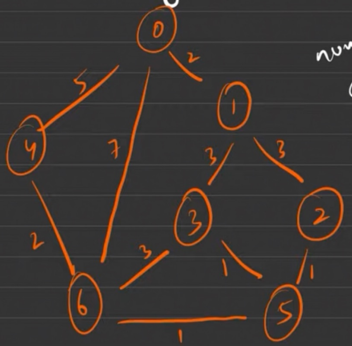
Input:

```
n=7, m=10
edges= [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],[3,5,1],[6,5,1],[2,5,1],
[0,4,5],[4,6,2]]
```

Output:

```
4
Explaination:

The four ways to get there in 7 minutes are:
- 0  6
- 0  4  6
- 0  1  2  5  6
- 0  1  3  5  6
```

Number of ways to reach a destination



number of shortest path

n = 7

src = 0 dst = 6

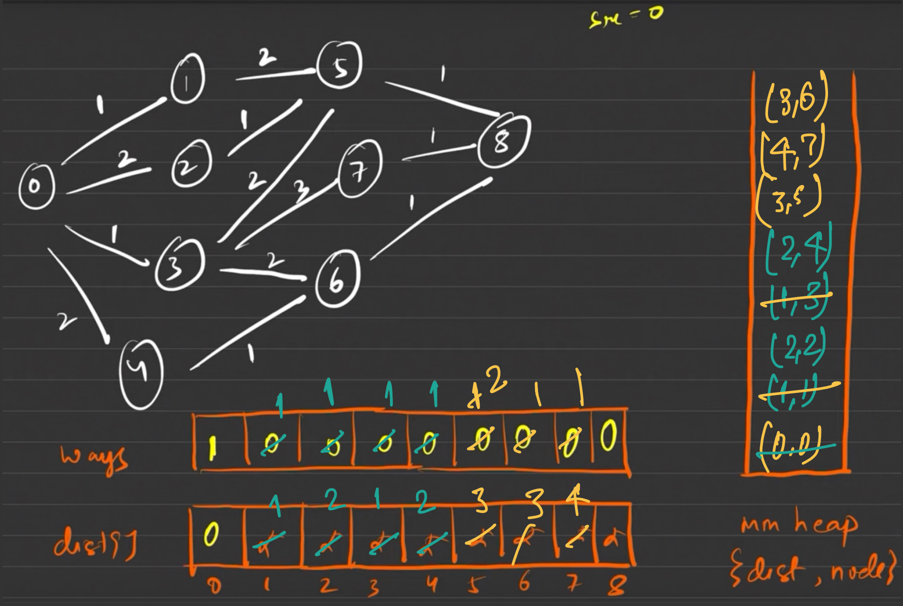
0 → 6

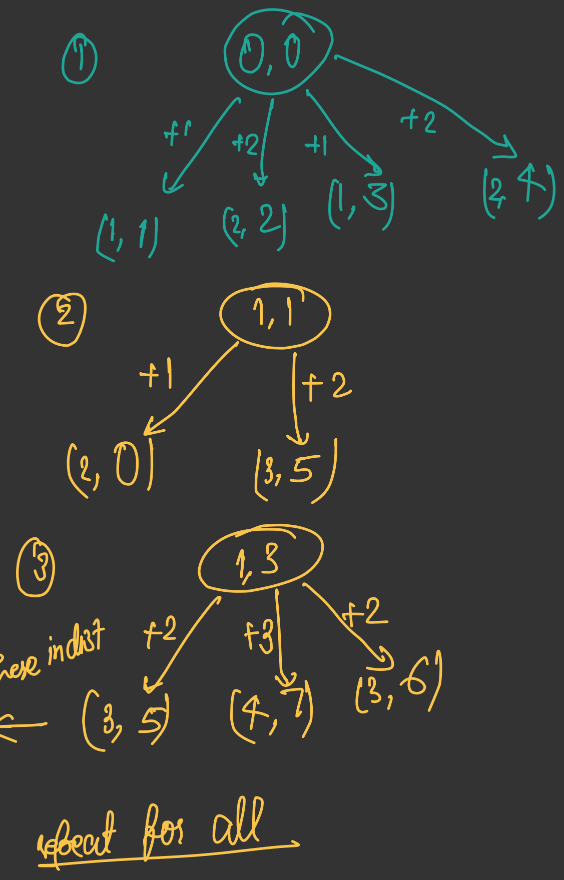
0 → 4 → 6

0 → 1 → 2 → 5 → 6

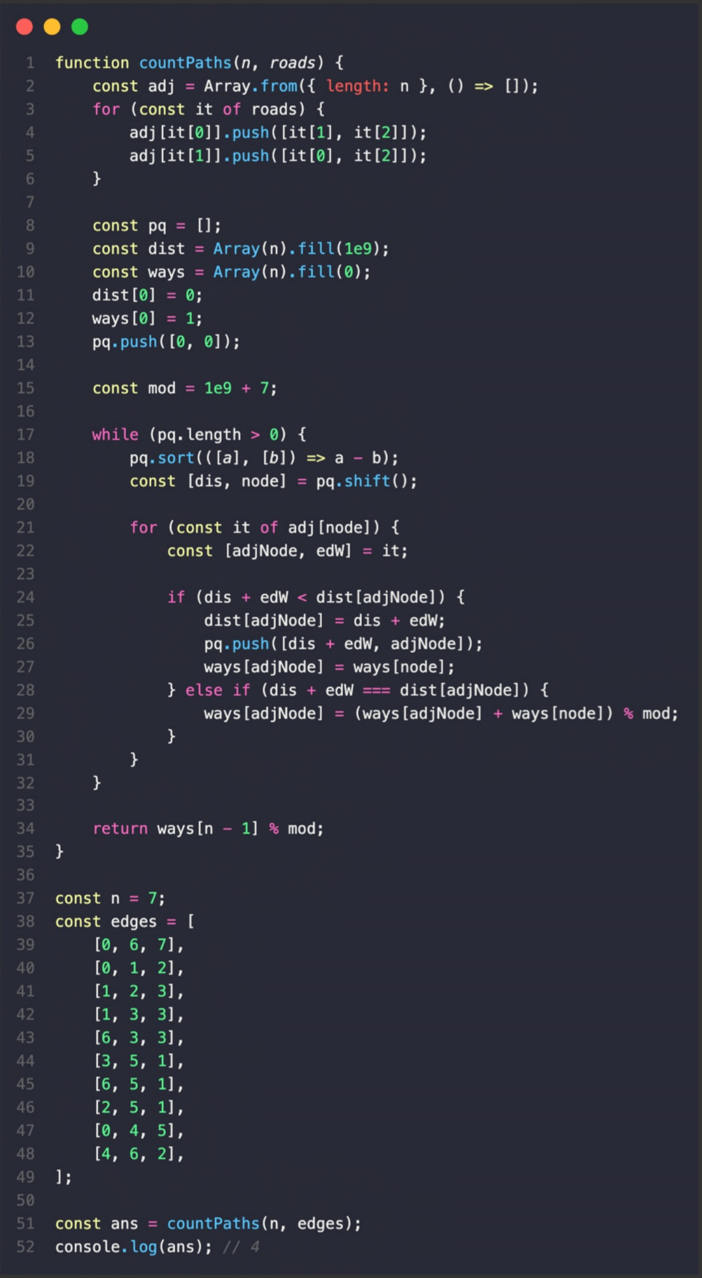
0 → 1 → 3 → 5 → 6

4 ans

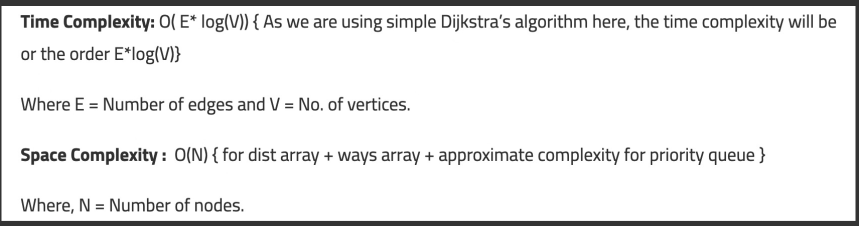




(3,5) already there in dist so, increase ways



```
function countPaths(n, roads) {
    const adj = Array.from({ length: n }, () => []);
    for (const it of roads) {
        adj[it[0]].push([it[1], it[2]]);
        adj[it[1]].push([it[0], it[2]]);
    }

    const pq = [];
    const dist = Array(n).fill(1e9);
    const ways = Array(n).fill(0);
    dist[0] = 0;
    ways[0] = 1;
    pq.push([0, 0]);

    const mod = 1e9 + 7;

    while (pq.length > 0) {
        pq.sort(([a], [b]) => a - b);
        const [dis, node] = pq.shift();

        for (const it of adj[node]) {
            const [adjNode, edW] = it;

            if (dis + edW < dist[adjNode]) {
                dist[adjNode] = dis + edW;
                pq.push([dis + edW, adjNode]);
                ways[adjNode] = ways[node];
            } else if (dis + edW === dist[adjNode]) {
                ways[adjNode] = (ways[adjNode] + ways[node]) % mod;
            }
        }
    }

    return ways[n - 1] % mod;
}

const n = 7;
const edges = [
    [0, 6, 7],
    [0, 1, 2],
    [1, 2, 3],
    [1, 3, 3],
    [6, 3, 3],
    [3, 5, 1],
    [6, 5, 1],
    [2, 5, 1],
    [0, 4, 5],
    [4, 6, 2],
];

const ans = countPaths(n, edges);
console.log(ans); // 4
```



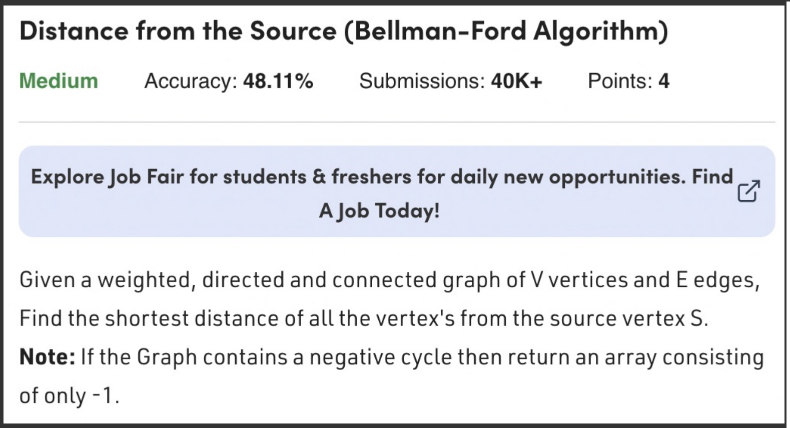
Time Complexity: O(E* log(V)) { As we are using simple Dijkstra's algorithm here, the time complexity will be or the order E*log(V)}

Where E = Number of edges and V = No. of vertices.

Space Complexity : O(N) { for dist array + ways array + approximate complexity for priority queue }

Where, N = Number of nodes.

lec 41 Bellman ford algorithm (shortest distance (works with negative cycle negative value



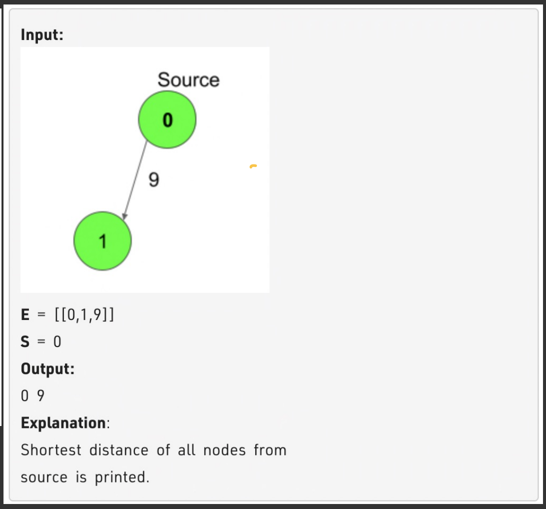
Distance from the Source (Bellman-Ford Algorithm)

Medium Accuracy: **48.11%** Submissions: **40K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

Given a weighted, directed and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S.

Note: If the Graph contains a negative cycle then return an array consisting of only -1.



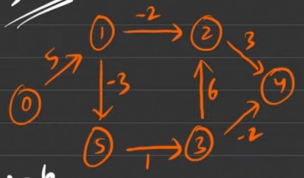
Input:

E = [[0,1,9]]
S = 0
Output:
0 9
Explanation:
Shortest distance of all nodes from source is printed.

→ works only with directed graph

→ convert undirected to directed graph to apply Bellman

src = 0



N = 6

(u, v, wt)
(3, 2, 6)
(5, 3, 1)
(0, 1, 5)
(1, 5, -3)
(1, 2, -2)
(3, 4, -2)
(2, 4, 3)

* Relax all the edges N-1 times sequentially

* Relax

if (dist[u] + wt < dist[v])
  dist[v] = dist[u] + w

* edges can be in any order

0	~~∞~~ 5	~~∞~~ 3	∞	~~∞~~ 6	~~∞~~ 2
0	1	2	3	4	5

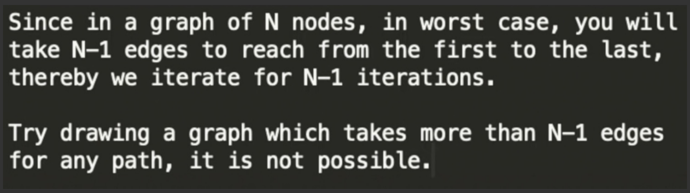
dist =

1st iteration

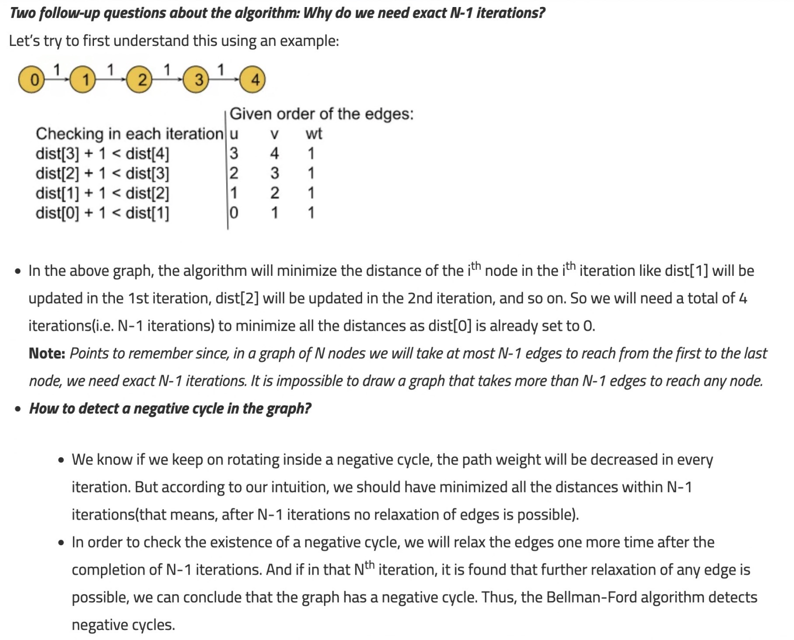
dist[3] + 6 < dist[2]
dist[5]+1 < dist[3]
dist[0] + 5 < dist[1] ⇒ 5 < ∞ ✓
dist[1] − 3 < dist[5] ⇒ 5-3 < ∞ ✓
dist[1] − 2 < dist[2] ⇒ 5-2 < ∞ ✓
dist[3] - 2 < dist[4] ⇒
dist[2] +3 < dist[4] ⇒ 6 < ∞ ✓

repeat this N-1 times & keep updating dist with min distance

why N-1 iterations.



```
Since in a graph of N nodes, in worst case, you will
take N-1 edges to reach from the first to the last,
thereby we iterate for N-1 iterations.

Try drawing a graph which takes more than N-1 edges
for any path, it is not possible.
```



Two follow-up questions about the algorithm: Why do we need exact N-1 iterations?

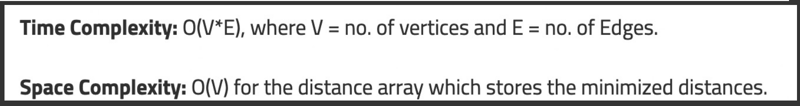
Let's try to first understand this using an example:

Given order of the edges:

Checking in each iteration	u	v	wt
dist[3] + 1 < dist[4]	3	4	1
dist[2] + 1 < dist[3]	2	3	1
dist[1] + 1 < dist[2]	1	2	1
dist[0] + 1 < dist[1]	0	1	1

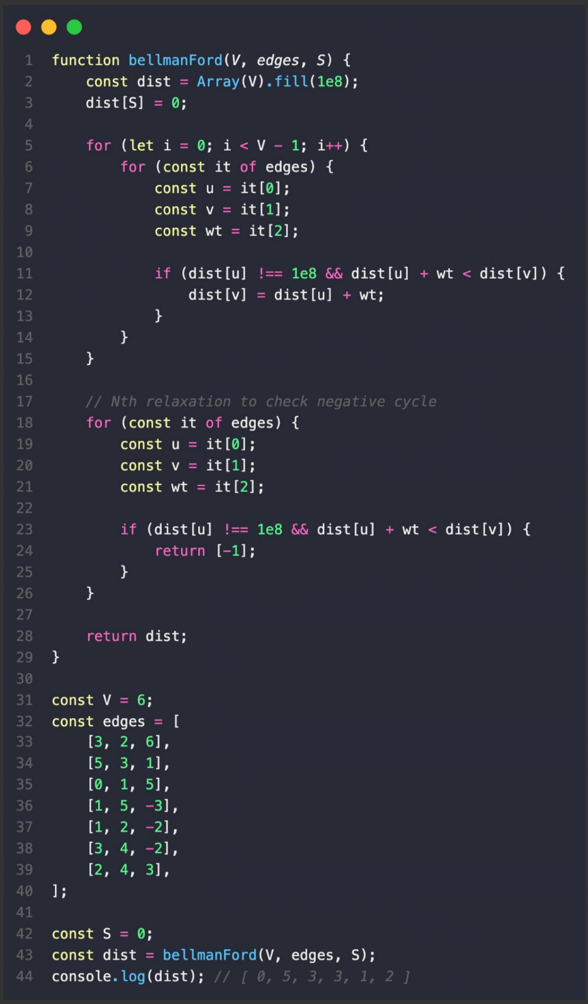
- In the above graph, the algorithm will minimize the distance of the i^{th} node in the i^{th} iteration like dist[1] will be updated in the 1st iteration, dist[2] will be updated in the 2nd iteration, and so on. So we will need a total of 4 iterations(i.e. N-1 iterations) to minimize all the distances as dist[0] is already set to 0.

  Note: *Points to remember since, in a graph of N nodes we will take at most N-1 edges to reach from the first to the last node, we need exact N-1 iterations. It is impossible to draw a graph that takes more than N-1 edges to reach any node.*
- ***How to detect a negative cycle in the graph?***
  - We know if we keep on rotating inside a negative cycle, the path weight will be decreased in every iteration. But according to our intuition, we should have minimized all the distances within N-1 iterations(that means, after N-1 iterations no relaxation of edges is possible).
  - In order to check the existence of a negative cycle, we will relax the edges one more time after the completion of N-1 iterations. And if in that N^{th} iteration, it is found that further relaxation of any edge is possible, we can conclude that the graph has a negative cycle. Thus, the Bellman-Ford algorithm detects negative cycles.



Time Complexity: O(V*E), where V = no. of vertices and E = no. of Edges.

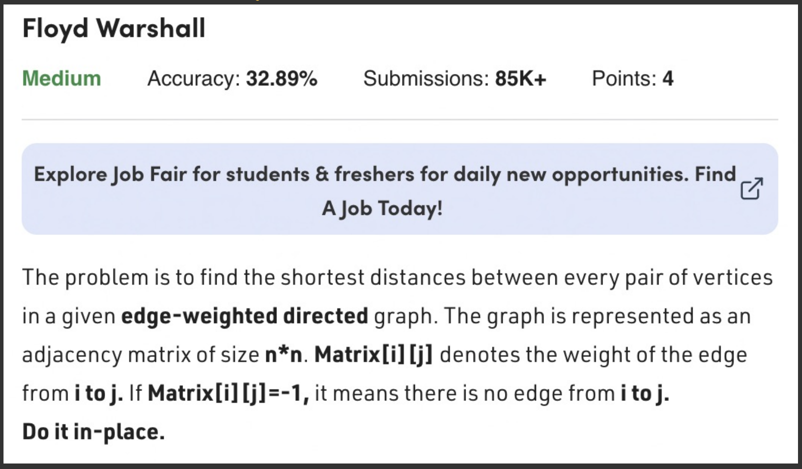
Space Complexity: O(V) for the distance array which stores the minimized distances.



```javascript
function bellmanFord(V, edges, S) {
    const dist = Array(V).fill(1e8);
    dist[S] = 0;

    for (let i = 0; i < V - 1; i++) {
        for (const it of edges) {
            const u = it[0];
            const v = it[1];
            const wt = it[2];

            if (dist[u] !== 1e8 && dist[u] + wt < dist[v]) {
                dist[v] = dist[u] + wt;
            }
        }
    }

    // Nth relaxation to check negative cycle
    for (const it of edges) {
        const u = it[0];
        const v = it[1];
        const wt = it[2];

        if (dist[u] !== 1e8 && dist[u] + wt < dist[v]) {
            return [-1];
        }
    }

    return dist;
}

const V = 6;
const edges = [
    [3, 2, 6],
    [5, 3, 1],
    [0, 1, 5],
    [1, 5, -3],
    [1, 2, -2],
    [3, 4, -2],
    [2, 4, 3],
];

const S = 0;
const dist = bellmanFord(V, edges, S);
console.log(dist); // [ 0, 5, 3, 3, 1, 2 ]
```

---

lec 42 Floyd Warshall Algorithm → multisource shortest path, negative cycle as well

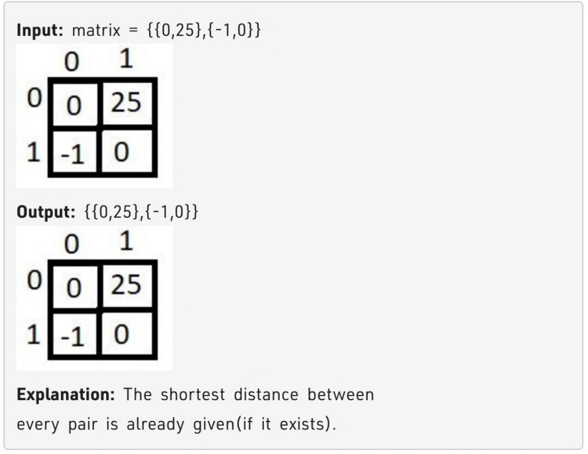


Floyd Warshall

Medium Accuracy: **32.89%** Submissions: **85K+** Points: **4**

Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

The problem is to find the shortest distances between every pair of vertices in a given **edge-weighted directed** graph. The graph is represented as an adjacency matrix of size **n*n**. **Matrix[i][j]** denotes the weight of the edge from **i to j**. If **Matrix[i][j]=-1,** it means there is no edge from **i to j**.
Do it in-place.



Input: matrix = {{0,25},{-1,0}}

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 25 \\ 1 & -1 & 0 \end{array}$$

Output: {{0,25},{-1,0}}

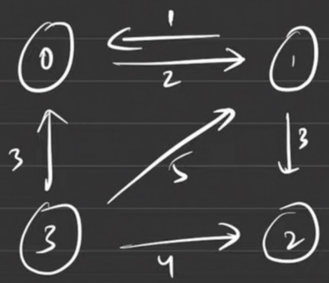
$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 25 \\ 1 & -1 & 0 \end{array}$$

Explanation: The shortest distance between every pair is already given(if it exists).

shortest path from each node to every other node

note → go via every vertex/node

ex $[3][1] = [3][0] + [0][1]$
$= 3 + 2$
$= 5$



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 2 & \alpha & \alpha \\ 1 & 1 & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \xrightarrow{0} \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 2 & \alpha & \alpha \\ 1 & 1 & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array}$$

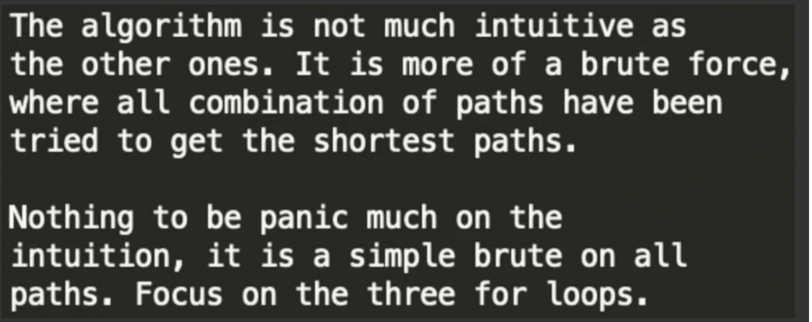
cost

```
for (via=0; via<n; via++)
    for ( i=0; i<n; i++)
        for ( j=0; j<n; j++)
            cost[i][j] = min( cost[i][j],
                              cost [i][via] +cost[via][j])
```

↓ via 1

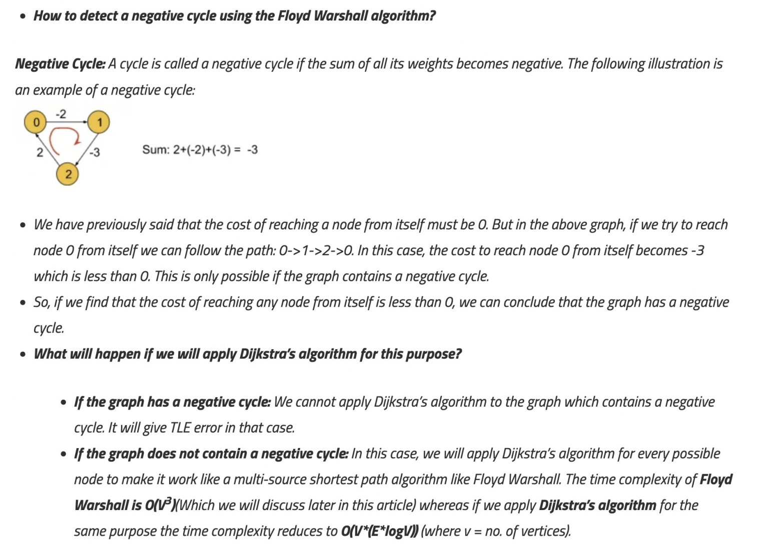
$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 2 & 5 & \infty \\ 1 & 1 & 0 & 3 & \infty \\ 2 & & \infty & 0 & \\ 3 & & 5 & & 0 \end{array}$$

$[0][2] = [0][1] + [1][2]$
$= 2 + 3$
$= 5$



The algorithm is not much intuitive as the other ones. It is more of a brute force, where all combination of paths have been tried to get the shortest paths.

Nothing to be panic much on the intuition, it is a simple brute on all paths. Focus on the three for loops.



- ***How to detect a negative cycle using the Floyd Warshall algorithm?***

Negative Cycle: *A cycle is called a negative cycle if the sum of all its weights becomes negative. The following illustration is an example of a negative cycle:*

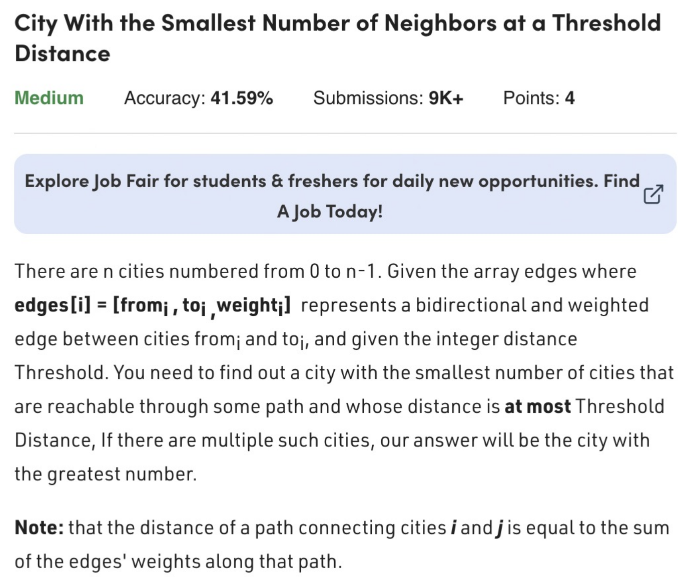
- *We have previously said that the cost of reaching a node from itself must be 0. But in the above graph, if we try to reach node 0 from itself we can follow the path: 0->1->2->0. In this case, the cost to reach node 0 from itself becomes -3 which is less than 0. This is only possible if the graph contains a negative cycle.*
- *So, if we find that the cost of reaching any node from itself is less than 0, we can conclude that the graph has a negative cycle.*
- ***What will happen if we will apply Dijkstra's algorithm for this purpose?***
  - ***If the graph has a negative cycle:*** *We cannot apply Dijkstra's algorithm to the graph which contains a negative cycle. It will give TLE error in that case.*
  - ***If the graph does not contain a negative cycle:*** *In this case, we will apply Dijkstra's algorithm for every possible node to make it work like a multi-source shortest path algorithm like Floyd Warshall. The time complexity of* ***Floyd Warshall is O(V^3)****(Which we will discuss later in this article) whereas if we apply* ***Dijkstra's algorithm*** *for the same purpose the time complexity reduces to* ***O(V*(E*logV))*** *(where v = no. of vertices).*

```
function shortestDistance(matrix) {
    const n = matrix.length;
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            if (matrix[i][j] === -1) {
                matrix[i][j] = 1e9;
            }
            if (i === j) matrix[i][j] = 0;
        }
    }

    for (let k = 0; k < n; k++) {
        for (let i = 0; i < n; i++) {
            for (let j = 0; j < n; j++) {
                matrix[i][j] = Math.min(
                    matrix[i][j],
                    matrix[i][k] + matrix[k][j]
                );
            }
        }
    }

    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            if (matrix[i][j] === 1e9) {
                matrix[i][j] = -1;
            }
        }
    }
}

const V = 4;
const matrix = Array.from({ length: V }, () => Array(V).fill(-1));
matrix[0][1] = 2;
matrix[1][0] = 1;
matrix[1][2] = 3;
matrix[3][0] = 3;
matrix[3][1] = 5;
matrix[3][2] = 4;

shortestDistance(matrix);
/* Output matrix
[
    [0, 2, 5, -1],
    [1, 0, 3, -1],
    [-1, -1, 0, -1],
    [3, 5, 4, 0],
];
*/
```

TC → $O(V^3)$ V → no of vertices

SC → $O(V^2)$ → store adjacency matrix

lec43 find the city with smalles number of neighbours at a threshold distance



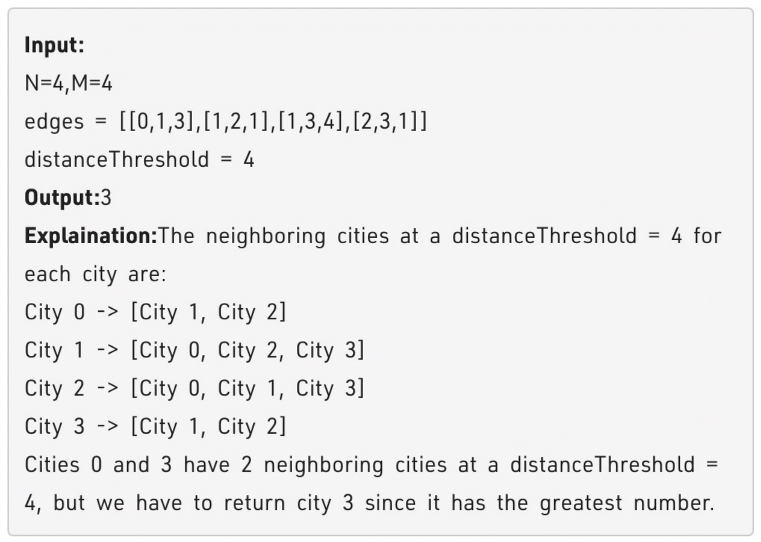
City With the Smallest Number of Neighbors at a Threshold Distance

Medium Accuracy: **41.59%** Submissions: **9K+** Points: **4**

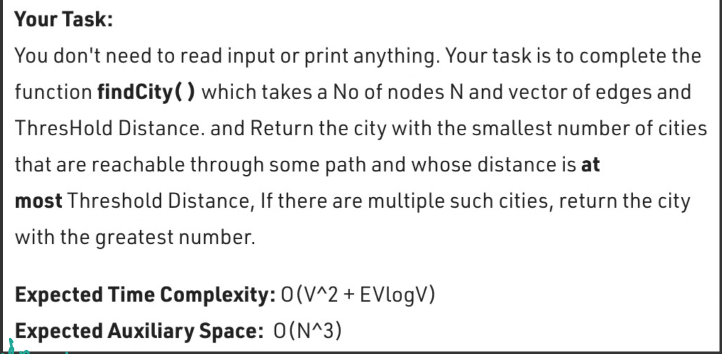
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

There are n cities numbered from 0 to n-1. Given the array edges where **edges[i] = [from$_i$, to$_i$,weight$_i$]** represents a bidirectional and weighted edge between cities from$_i$ and to$_i$, and given the integer distance Threshold. You need to find out a city with the smallest number of cities that are reachable through some path and whose distance is **at most** Threshold Distance, If there are multiple such cities, our answer will be the city with the greatest number.

Note: that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

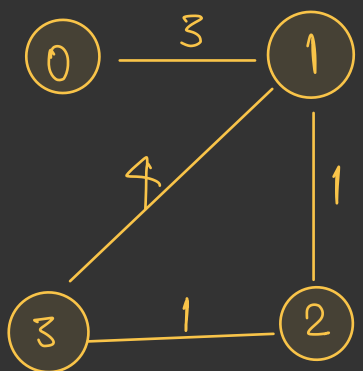


Input:
N=4,M=4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold = 4
Output:3
Explaination: The neighboring cities at a distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.



Your Task:
You don't need to read input or print anything. Your task is to complete the function **findCity( )** which takes a No of nodes N and vector of edges and ThresHold Distance. and Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** Threshold Distance, If there are multiple such cities, return the city with the greatest number.

Expected Time Complexity: O(V^2 + EVlogV)
Expected Auxiliary Space: O(N^3)



max allowed path weight
threshold = 4

src city   dest cities within threshold

0 → 1, 2
1 → 0, 2, 3
2 → 0, 1, 3
3 → 1, 2

lowest num of cities we can travel to

multiple
so select largest src city

③ ans

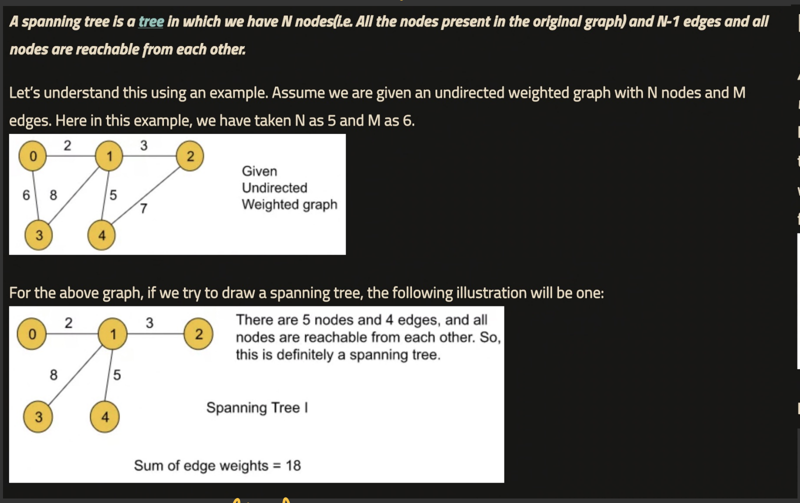
⇒ Floyd warshall algo

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

```javascript
function findCity(n, m, edges, distanceThreshold) {
    const dist = Array.from({ length: n }, () => Array(n).fill(Infinity));
    for (const it of edges) {
        dist[it[0]][it[1]] = it[2];
        dist[it[1]][it[0]] = it[2];
    }
    for (let i = 0; i < n; i++) dist[i][i] = 0;
    for (let k = 0; k < n; k++) {
        for (let i = 0; i < n; i++) {
            for (let j = 0; j < n; j++) {
                if (dist[i][k] === Infinity || dist[k][j] === Infinity)
                    continue;
                dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }

    let cntCity = n;
    let cityNo = -1;
    for (let city = 0; city < n; city++) {
        let cnt = 0;
        for (let adjCity = 0; adjCity < n; adjCity++) {
            if (dist[city][adjCity] <= distanceThreshold) cnt++;
        }

        if (cnt <= cntCity) {
            cntCity = cnt;
            cityNo = city;
        }
    }
    return cityNo;
}

const n = 4;
const m = 4;
const edges = [
    [0, 1, 3],
    [1, 2, 1],
    [1, 3, 4],
    [2, 3, 1],
];
const distanceThreshold = 4;

const cityNo = findCity(n, m, edges, distanceThreshold);
console.log("The answer is node:", cityNo);
// The answer is node: 3
```

TC → $O(V^3)$
SC → $O(V^2)$

lec 44 min spanning tree - theory.



A spanning tree is a tree in which we have N nodes(i.e. All the nodes present in the original graph) and N-1 edges and all nodes are reachable from each other.

Let's understand this using an example. Assume we are given an undirected weighted graph with N nodes and M edges. Here in this example, we have taken N as 5 and M as 6.

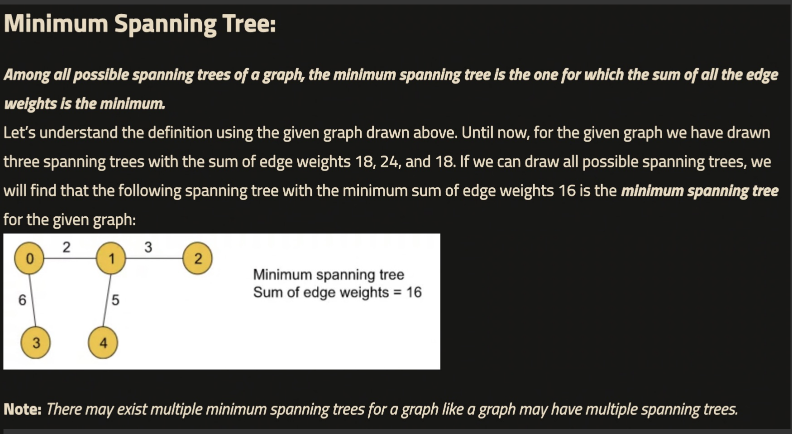
Given Undirected Weighted graph

For the above graph, if we try to draw a spanning tree, the following illustration will be one:

There are 5 nodes and 4 edges, and all nodes are reachable from each other. So, this is definitely a spanning tree.

Spanning Tree I

Sum of edge weights = 18



Minimum Spanning Tree:

Among all possible spanning trees of a graph, the minimum spanning tree is the one for which the sum of all the edge weights is the minimum.

Let's understand the definition using the given graph drawn above. Until now, for the given graph we have drawn three spanning trees with the sum of edge weights 18, 24, and 18. If we can draw all possible spanning trees, we will find that the following spanning tree with the minimum sum of edge weights 16 is the ***minimum spanning tree*** for the given graph:

Minimum spanning tree
Sum of edge weights = 16

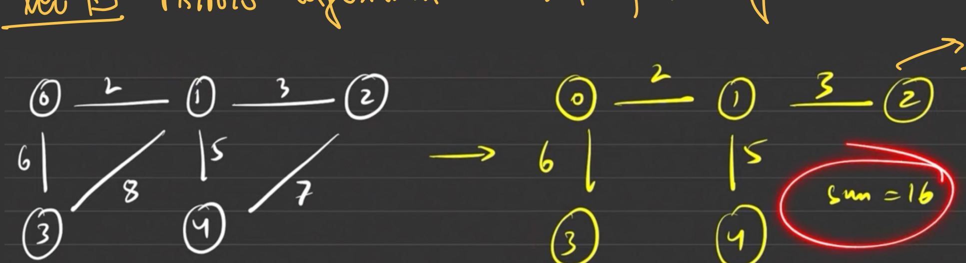
Note: *There may exist multiple minimum spanning trees for a graph like a graph may have multiple spanning trees.*

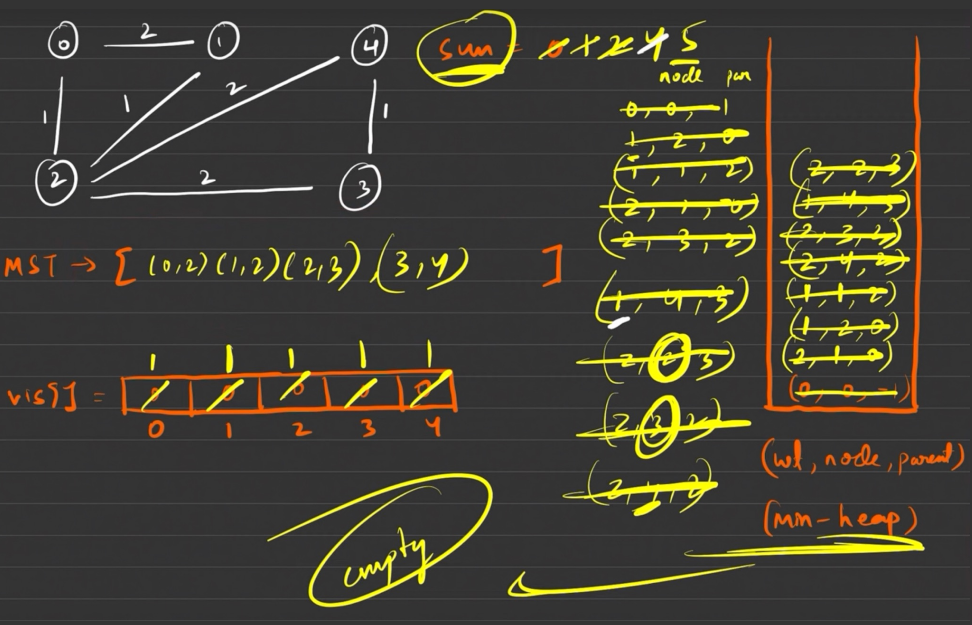
ways to find MST

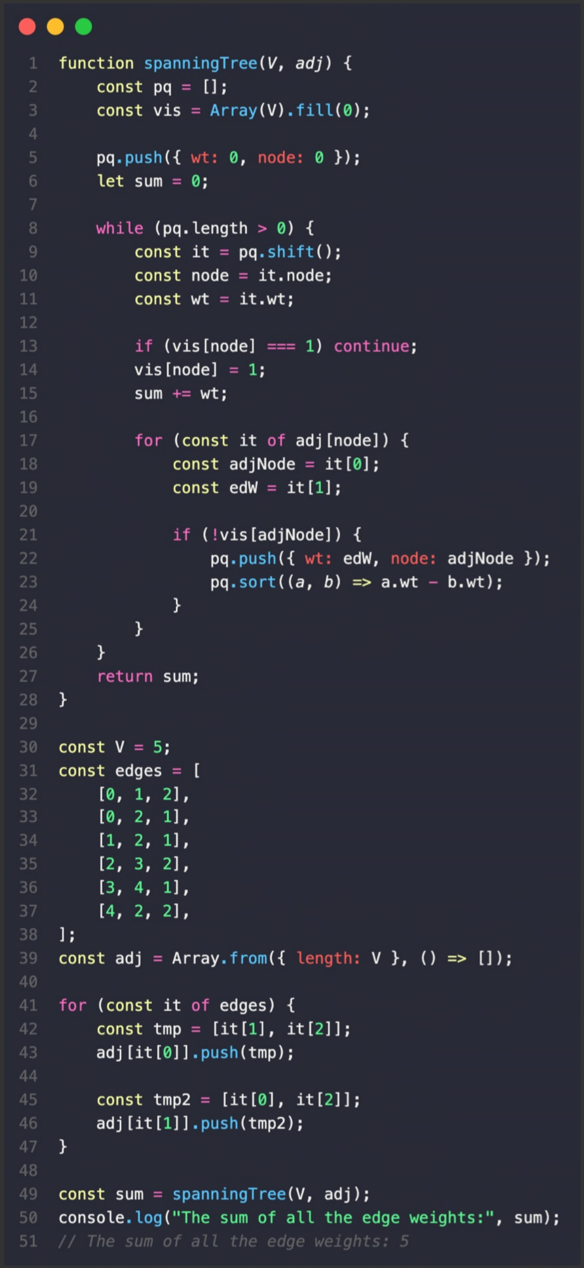
1. Prim's algorithm
2. Kruskal's algorithm

---

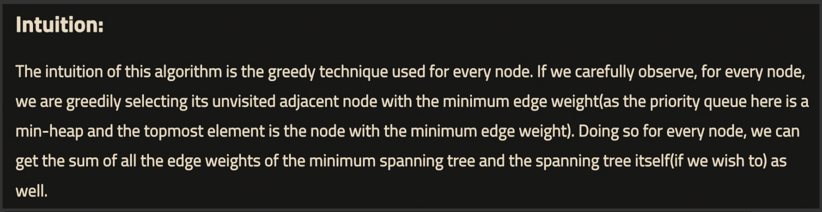
lec 45 Prim's algorithm — min spanning tree





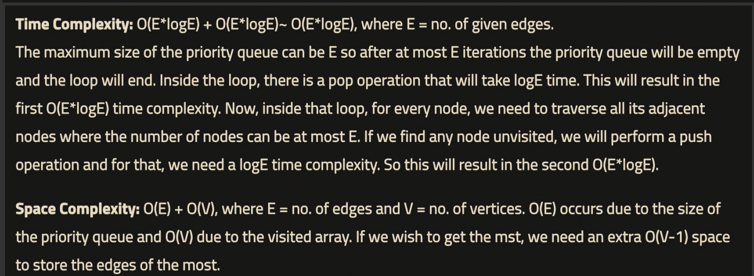


```
function spanningTree(V, adj) {
    const pq = [];
    const vis = Array(V).fill(0);

    pq.push({ wt: 0, node: 0 });
    let sum = 0;

    while (pq.length > 0) {
        const it = pq.shift();
        const node = it.node;
        const wt = it.wt;

        if (vis[node] === 1) continue;
        vis[node] = 1;
        sum += wt;

        for (const it of adj[node]) {
            const adjNode = it[0];
            const edW = it[1];

            if (!vis[adjNode]) {
                pq.push({ wt: edW, node: adjNode });
                pq.sort((a, b) => a.wt - b.wt);
            }
        }
    }
    return sum;
}

const V = 5;
const edges = [
    [0, 1, 2],
    [0, 2, 1],
    [1, 2, 1],
    [2, 3, 2],
    [3, 4, 1],
    [4, 2, 2],
];
const adj = Array.from({ length: V }, () => []);

for (const it of edges) {
    const tmp = [it[1], it[2]];
    adj[it[0]].push(tmp);

    const tmp2 = [it[0], it[2]];
    adj[it[1]].push(tmp2);
}

const sum = spanningTree(V, adj);
console.log("The sum of all the edge weights:", sum);
// The sum of all the edge weights: 5
```



Intuition:

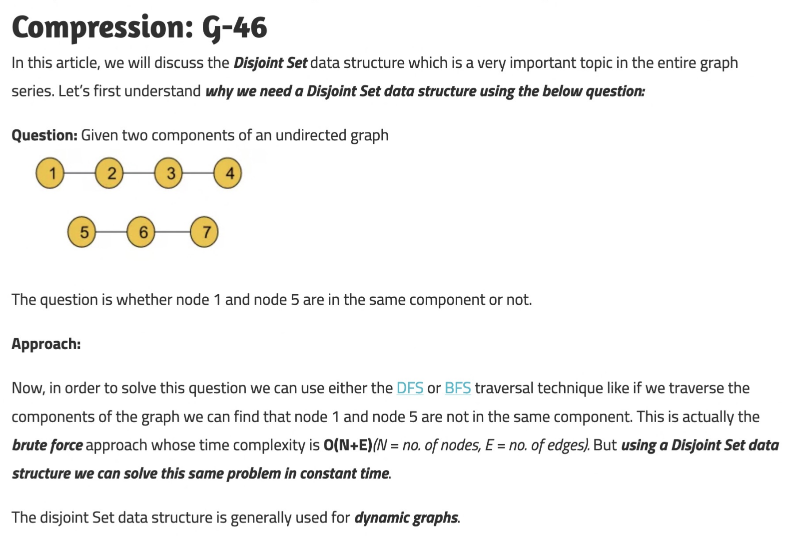
The intuition of this algorithm is the greedy technique used for every node. If we carefully observe, for every node, we are greedily selecting its unvisited adjacent node with the minimum edge weight(as the priority queue here is a min-heap and the topmost element is the node with the minimum edge weight). Doing so for every node, we can get the sum of all the edge weights of the minimum spanning tree and the spanning tree itself(if we wish to) as well.



Time Complexity: O(E*logE) + O(E*logE)~ O(E*logE), where E = no. of given edges.
The maximum size of the priority queue can be E so after at most E iterations the priority queue will be empty and the loop will end. Inside the loop, there is a pop operation that will take logE time. This will result in the first O(E*logE) time complexity. Now, inside that loop, for every node, we need to traverse all its adjacent nodes where the number of nodes can be at most E. If we find any node unvisited, we will perform a push operation and for that, we need a logE time complexity. So this will result in the second O(E*logE).

Space Complexity: O(E) + O(V), where E = no. of edges and V = no. of vertices. O(E) occurs due to the size of the priority queue and O(V) due to the visited array. If we wish to get the mst, we need an extra O(V-1) space to store the edges of the most.

lec46 disjoint set ****



Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46

In this article, we will discuss the ***Disjoint Set*** data structure which is a very important topic in the entire graph series. Let's first understand ***why we need a Disjoint Set data structure using the below question:***

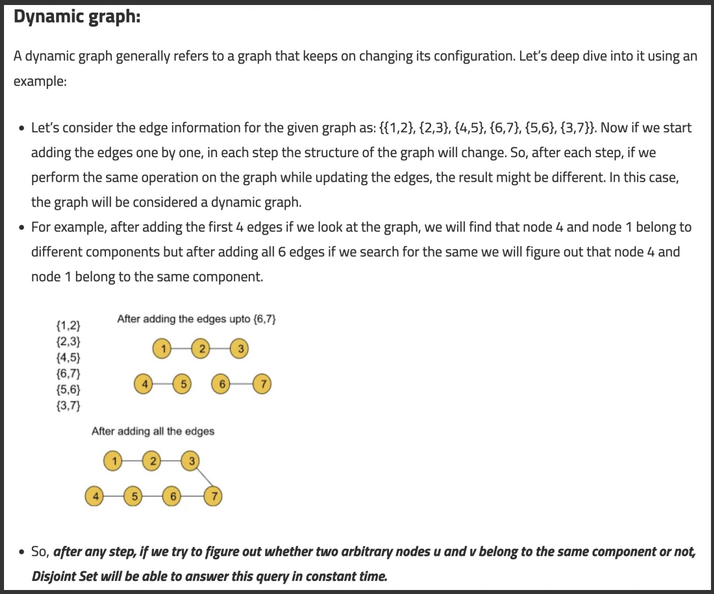
Question: Given two components of an undirected graph

The question is whether node 1 and node 5 are in the same component or not.

Approach:

Now, in order to solve this question we can use either the DFS or BFS traversal technique like if we traverse the components of the graph we can find that node 1 and node 5 are not in the same component. This is actually the ***brute force*** approach whose time complexity is **O(N+E)***(N = no. of nodes, E = no. of edges)*. But ***using a Disjoint Set data structure we can solve this same problem in constant time***.

The disjoint Set data structure is generally used for ***dynamic graphs***.

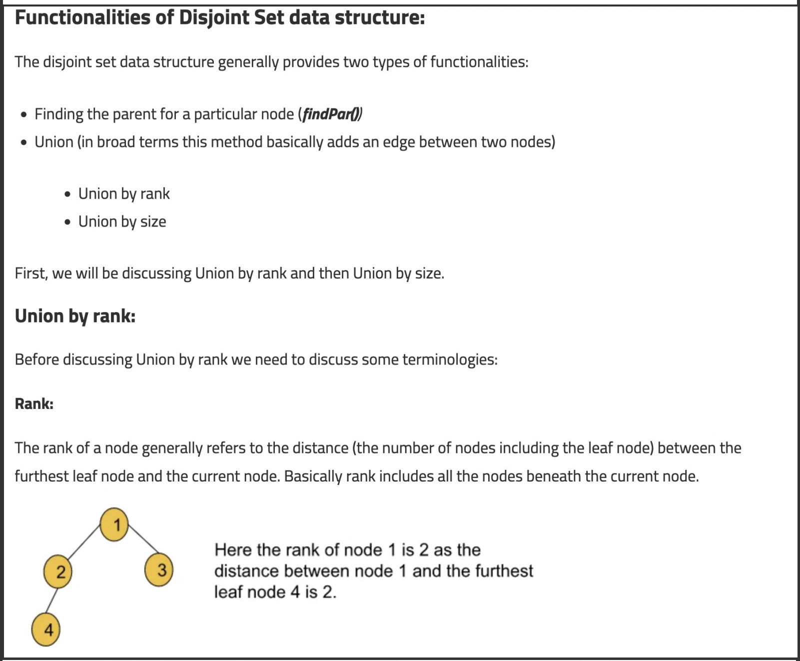


Dynamic graph:

A dynamic graph generally refers to a graph that keeps on changing its configuration. Let's deep dive into it using an example:

- Let's consider the edge information for the given graph as: {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}, {3,7}}. Now if we start adding the edges one by one, in each step the structure of the graph will change. So, after each step, if we perform the same operation on the graph while updating the edges, the result might be different. In this case, the graph will be considered a dynamic graph.
- For example, after adding the first 4 edges if we look at the graph, we will find that node 4 and node 1 belong to different components but after adding all 6 edges if we search for the same we will figure out that node 4 and node 1 belong to the same component.

- So, ***after any step, if we try to figure out whether two arbitrary nodes u and v belong to the same component or not, Disjoint Set will be able to answer this query in constant time.***



Functionalities of Disjoint Set data structure:

The disjoint set data structure generally provides two types of functionalities:

- Finding the parent for a particular node (***findPar()***)
- Union (in broad terms this method basically adds an edge between two nodes)
    - Union by rank
    - Union by size

First, we will be discussing Union by rank and then Union by size.

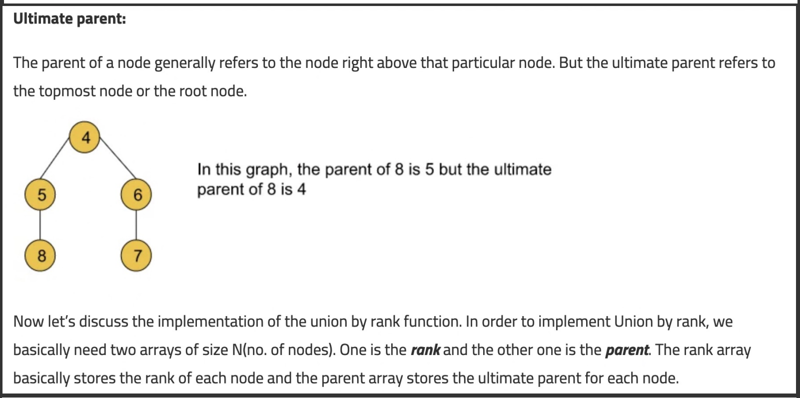
Union by rank:

Before discussing Union by rank we need to discuss some terminologies:

Rank:

The rank of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node.

Here the rank of node 1 is 2 as the distance between node 1 and the furthest leaf node 4 is 2.

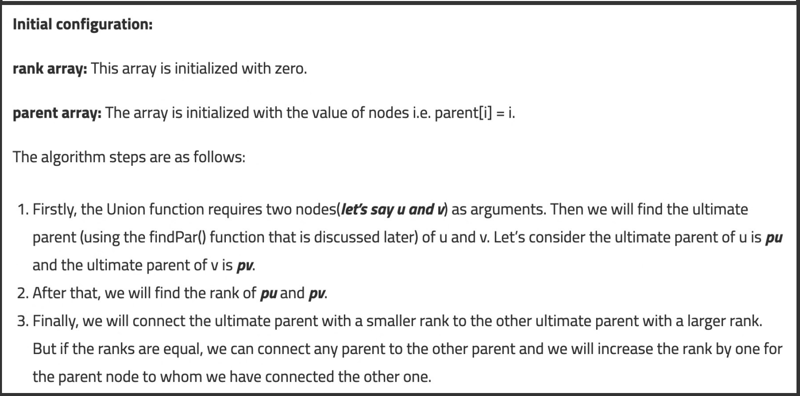


Ultimate parent:

The parent of a node generally refers to the node right above that particular node. But the ultimate parent refers to the topmost node or the root node.

In this graph, the parent of 8 is 5 but the ultimate parent of 8 is 4

Now let's discuss the implementation of the union by rank function. In order to implement Union by rank, we basically need two arrays of size N(no. of nodes). One is the ***rank*** and the other one is the ***parent***. The rank array basically stores the rank of each node and the parent array stores the ultimate parent for each node.



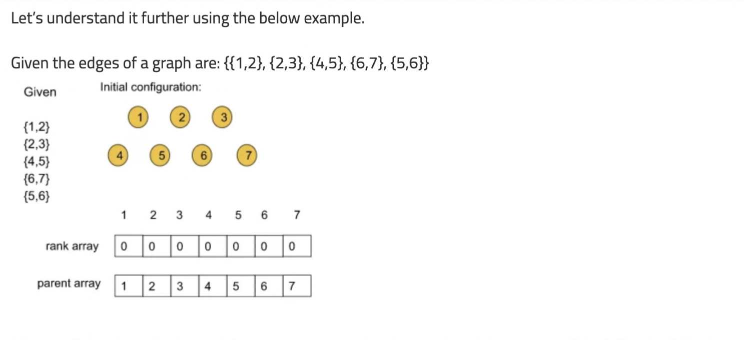
Initial configuration:

rank array: This array is initialized with zero.

parent array: The array is initialized with the value of nodes i.e. parent[i] = i.

The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(***let's say u and v***) as arguments. Then we will find the ultimate parent (using the findPar() function that is discussed later) of u and v. Let's consider the ultimate parent of u is ***pu*** and the ultimate parent of v is ***pv***.
2. After that, we will find the rank of ***pu*** and ***pv***.
3. Finally, we will connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. But if the ranks are equal, we can connect any parent to the other parent and we will increase the rank by one for the parent node to whom we have connected the other one.

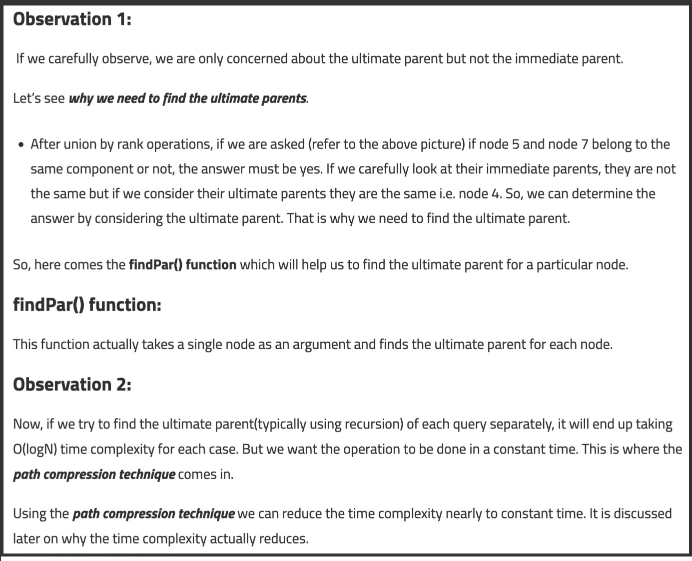


Let's understand it further using the below example.

Given the edges of a graph are: {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}}

	1	2	3	4	5	6	7
rank array	0	0	0	0	0	0	0
parent array	1	2	3	4	5	6	7

After applying the union by rank function to every edge the graph and the arrays will look like the following:



Observation 1:

If we carefully observe, we are only concerned about the ultimate parent but not the immediate parent.

Let's see ***why we need to find the ultimate parents***.

- After union by rank operations, if we are asked (refer to the above picture) if node 5 and node 7 belong to the same component or not, the answer must be yes. If we carefully look at their immediate parents, they are not the same but if we consider their ultimate parents they are the same i.e. node 4. So, we can determine the answer by considering the ultimate parent. That is why we need to find the ultimate parent.

So, here comes the **findPar() function** which will help us to find the ultimate parent for a particular node.

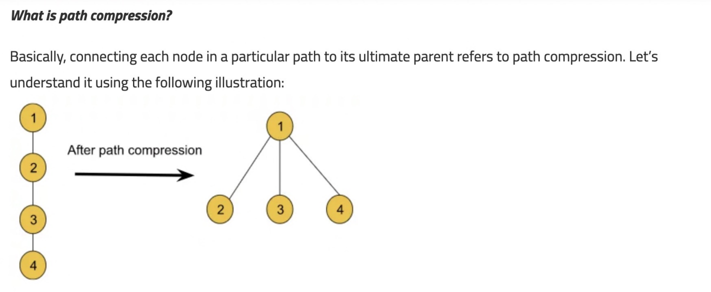
findPar() function:

This function actually takes a single node as an argument and finds the ultimate parent for each node.

Observation 2:

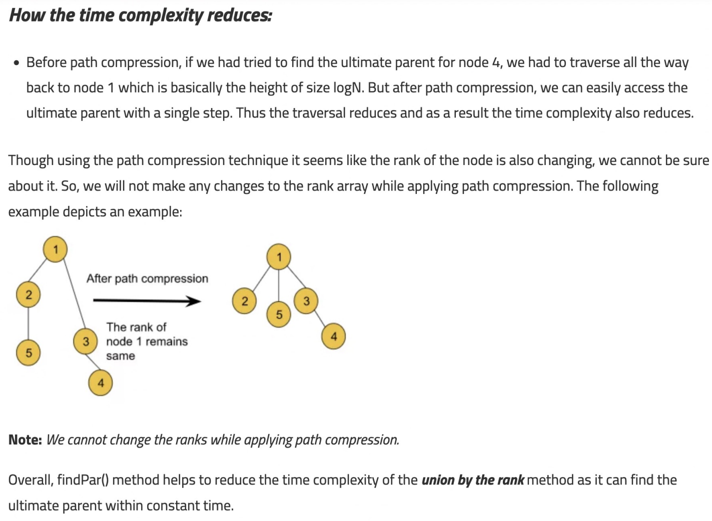
Now, if we try to find the ultimate parent(typically using recursion) of each query separately, it will end up taking O(logN) time complexity for each case. But we want the operation to be done in a constant time. This is where the ***path compression technique*** comes in.

Using the ***path compression technique*** we can reduce the time complexity nearly to constant time. It is discussed later on why the time complexity actually reduces.



What is path compression?

Basically, connecting each node in a particular path to its ultimate parent refers to path compression. Let's understand it using the following illustration:



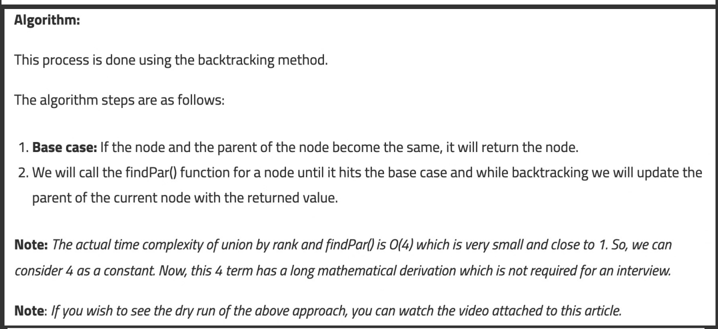
How the time complexity reduces:

- Before path compression, if we had tried to find the ultimate parent for node 4, we had to traverse all the way back to node 1 which is basically the height of size logN. But after path compression, we can easily access the ultimate parent with a single step. Thus the traversal reduces and as a result the time complexity also reduces.

Though using the path compression technique it seems like the rank of the node is also changing, we cannot be sure about it. So, we will not make any changes to the rank array while applying path compression. The following example depicts an example:

Note: *We cannot change the ranks while applying path compression.*

Overall, findPar() method helps to reduce the time complexity of the ***union by the rank*** method as it can find the ultimate parent within constant time.



Algorithm:

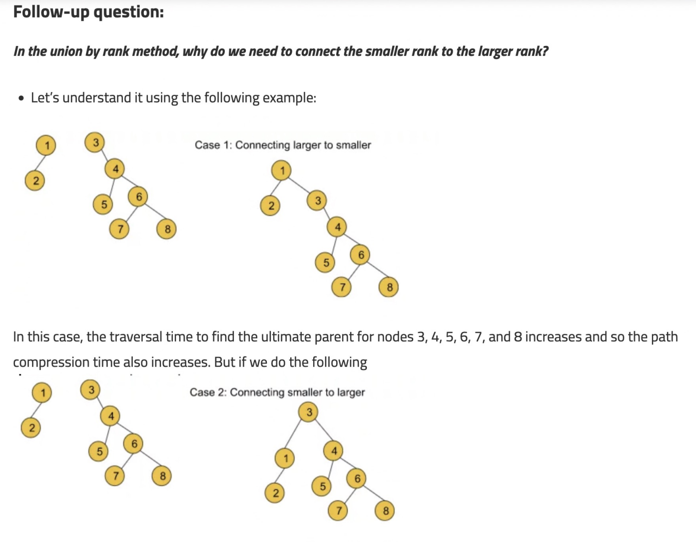
This process is done using the backtracking method.

The algorithm steps are as follows:

1. **Base case:** If the node and the parent of the node become the same, it will return the node.
2. We will call the findPar() function for a node until it hits the base case and while backtracking we will update the parent of the current node with the returned value.

Note: *The actual time complexity of union by rank and findPar() is O(4) which is very small and close to 1. So, we can consider 4 as a constant. Now, this 4 term has a long mathematical derivation which is not required for an interview.*

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*



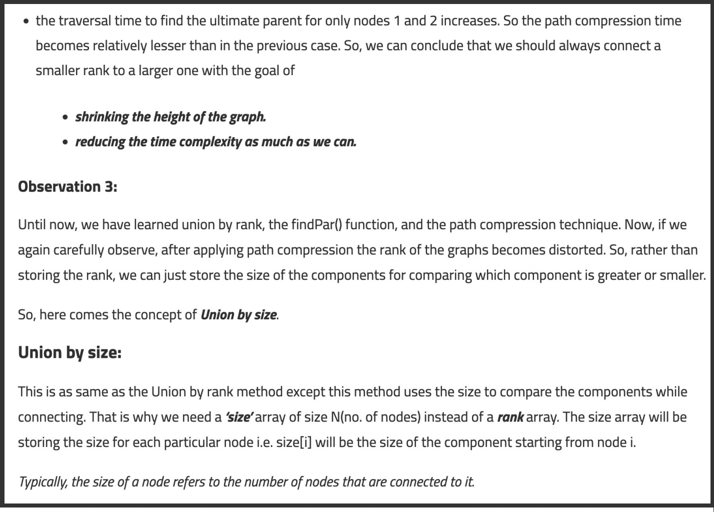
Follow-up question:

In the union by rank method, why do we need to connect the smaller rank to the larger rank?

- Let's understand it using the following example:

In this case, the traversal time to find the ultimate parent for nodes 3, 4, 5, 6, 7, and 8 increases and so the path compression time also increases. But if we do the following

```javascript
class DisjointSet {
    constructor(n) {
        this.rank = new Array(n + 1).fill(0);
        this.parent = new Array(n + 1).fill(0).map((_, i) => i);
    }

    findUPar(node) {
        if (node === this.parent[node]) {
            return node;
        }
        return (this.parent[node] = this.findUPar(this.parent[node]));
    }

    unionByRank(u, v) {
        const ulp_u = this.findUPar(u);
        const ulp_v = this.findUPar(v);
        if (ulp_u === ulp_v) return;
        if (this.rank[ulp_u] < this.rank[ulp_v]) {
            this.parent[ulp_u] = ulp_v;
        } else if (this.rank[ulp_v] < this.rank[ulp_u]) {
            this.parent[ulp_v] = ulp_u;
        } else {
            this.parent[ulp_v] = ulp_u;
            this.rank[ulp_u]++;
        }
    }
}

const main = () => {
    const ds = new DisjointSet(7);
    ds.unionByRank(1, 2);
    ds.unionByRank(2, 3);
    ds.unionByRank(4, 5);
    ds.unionByRank(6, 7);
    ds.unionByRank(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) === ds.findUPar(7)) {
        console.log("Same");
    } else {
        console.log("Not same");
    }

    ds.unionByRank(3, 7);

    if (ds.findUPar(3) === ds.findUPar(7)) {
        console.log("Same");
    } else {
        console.log("Not same");
    }
};

main();
// Not same
// Same
```



- the traversal time to find the ultimate parent for only nodes 1 and 2 increases. So the path compression time becomes relatively lesser than in the previous case. So, we can conclude that we should always connect a smaller rank to a larger one with the goal of
  - ***shrinking the height of the graph.***
  - ***reducing the time complexity as much as we can.***

Observation 3:

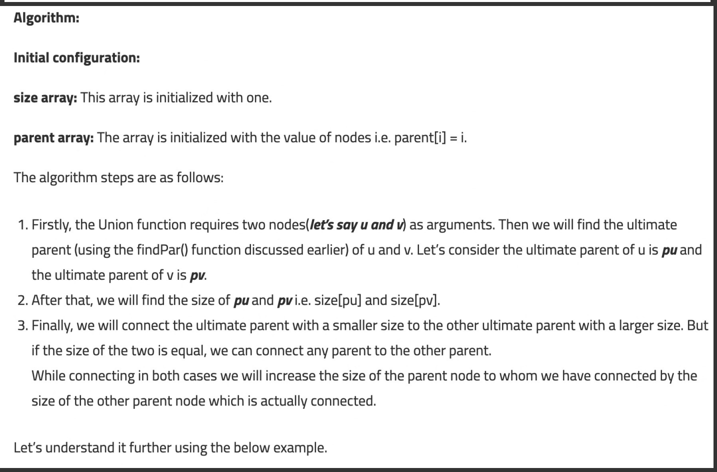
Until now, we have learned union by rank, the findPar() function, and the path compression technique. Now, if we again carefully observe, after applying path compression the rank of the graphs becomes distorted. So, rather than storing the rank, we can just store the size of the components for comparing which component is greater or smaller.

So, here comes the concept of ***Union by size***.

Union by size:

This is as same as the Union by rank method except this method uses the size to compare the components while connecting. That is why we need a ***'size'*** array of size N(no. of nodes) instead of a ***rank*** array. The size array will be storing the size for each particular node i.e. size[i] will be the size of the component starting from node i.

Typically, the size of a node refers to the number of nodes that are connected to it.



Algorithm:

Initial configuration:

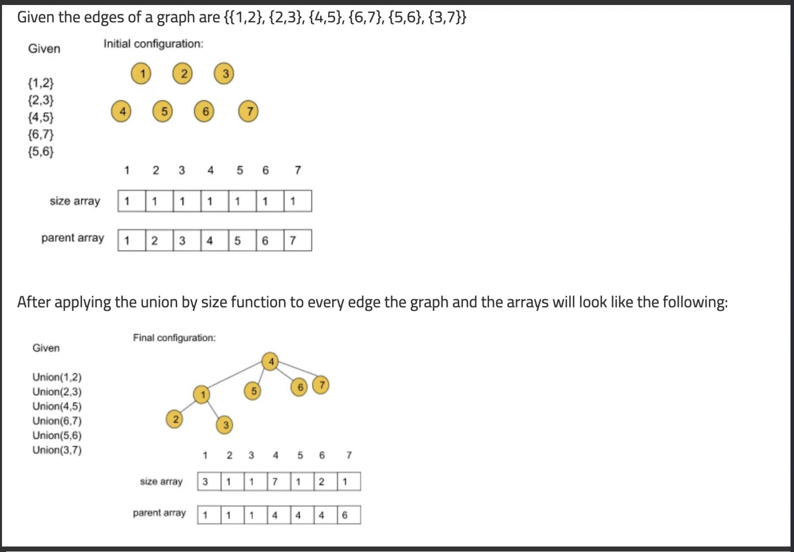
size array: This array is initialized with one.

parent array: The array is initialized with the value of nodes i.e. parent[i] = i.

The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(***let's say u and v***) as arguments. Then we will find the ultimate parent (using the findPar() function discussed earlier) of u and v. Let's consider the ultimate parent of u is ***pu*** and the ultimate parent of v is ***pv***.
2. After that, we will find the size of ***pu*** and ***pv*** i.e. size[pu] and size[pv].
3. Finally, we will connect the ultimate parent with a smaller size to the other ultimate parent with a larger size. But if the size of the two is equal, we can connect any parent to the other parent.
   While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

Let's understand it further using the below example.

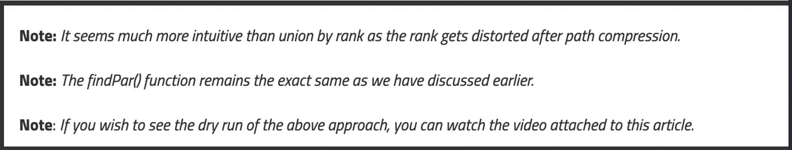


Given the edges of a graph are {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}, {3,7}}

	1	2	3	4	5	6	7
size array	1	1	1	1	1	1	1
parent array	1	2	3	4	5	6	7

After applying the union by size function to every edge the graph and the arrays will look like the following:

	1	2	3	4	5	6	7
size array	3	1	1	7	1	2	1
parent array	1	1	1	4	4	4	6



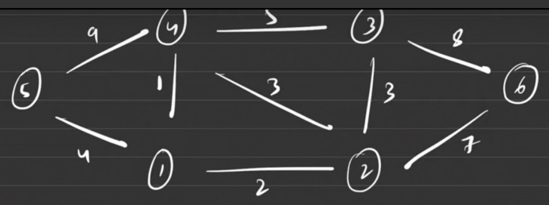
Note: *It seems much more intuitive than union by rank as the rank gets distorted after path compression.*

Note: *The findPar() function remains the exact same as we have discussed earlier.*

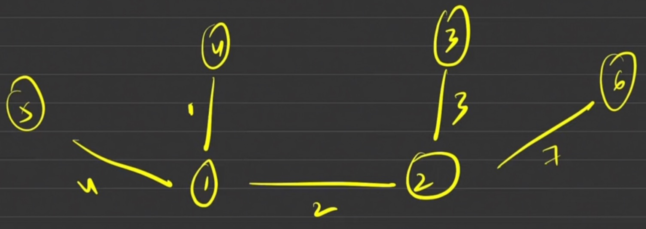
Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

```
class DisjointSet {
    constructor(n) {
        this.rank = new Array(n + 1).fill(0);
        this.parent = new Array(n + 1).fill(0).map((_, i) => i);
        this.size = new Array(n + 1).fill(1);
    }

    findUPar(node) {
        if (node === this.parent[node]) {
            return node;
        }
        return (this.parent[node] = this.findUPar(this.parent[node]));
    }

    unionByRank(u, v) {
        const ulp_u = this.findUPar(u);
        const ulp_v = this.findUPar(v);
        if (ulp_u === ulp_v) return;
        if (this.rank[ulp_u] < this.rank[ulp_v]) {
            this.parent[ulp_u] = ulp_v;
        } else if (this.rank[ulp_v] < this.rank[ulp_u]) {
            this.parent[ulp_v] = ulp_u;
        } else {
            this.parent[ulp_v] = ulp_u;
            this.rank[ulp_u]++;
        }
    }

    unionBySize(u, v) {
        const ulp_u = this.findUPar(u);
        const ulp_v = this.findUPar(v);
        if (ulp_u === ulp_v) return;
        if (this.size[ulp_u] < this.size[ulp_v]) {
            this.parent[ulp_u] = ulp_v;
            this.size[ulp_v] += this.size[ulp_u];
        } else {
            this.parent[ulp_v] = ulp_u;
            this.size[ulp_u] += this.size[ulp_v];
        }
    }
}

const main = () => {
    const ds = new DisjointSet(7);
    ds.unionBySize(1, 2);
    ds.unionBySize(2, 3);
    ds.unionBySize(4, 5);
    ds.unionBySize(6, 7);
    ds.unionBySize(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) === ds.findUPar(7)) {
        console.log("Same");
    } else {
        console.log("Not same");
    }

    ds.unionBySize(3, 7);

    if (ds.findUPar(3) === ds.findUPar(7)) {
        console.log("Same");
    } else {
        console.log("Not same");
    }
};

main();
// Not same
// Same
```

lec 47 → Kruskal's Algorithm – minimum spanning tree

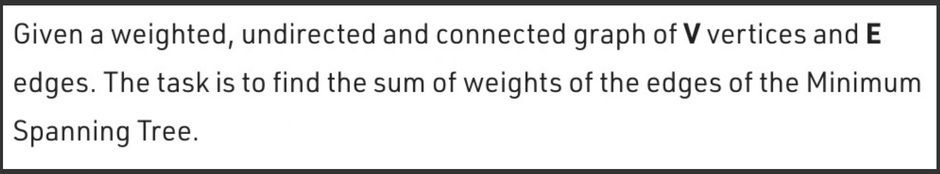


(•) sort all the edges acc to wt.
(wt, u, v)
✓→ 1, 1, 4
✓→ 2, 1, 2
✓→ 3, 2, 3
✗ 3, 2, 4
✓ 4, 1, 5
✗ 5, 3, 4
✓ 7, 2, 6
✗ 8, 3, 6
✗ 9, 4, 5

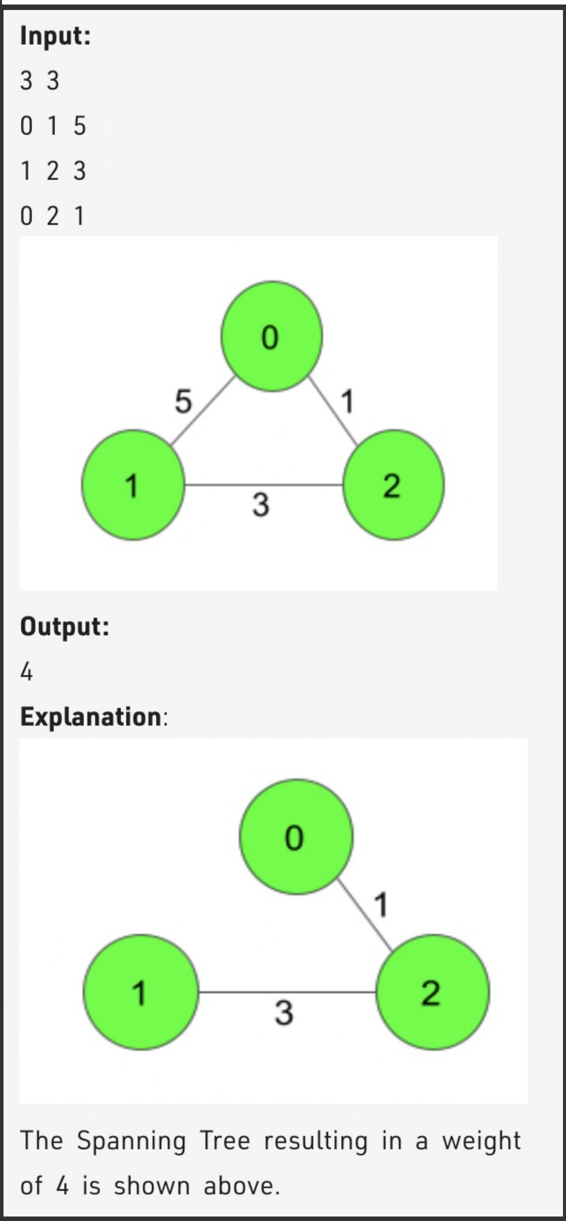


wt = 1+2+3+4+7
= 17

note → if a node is already part of MST, do not add it again



Given a weighted, undirected and connected graph of **V** vertices and **E** edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.



Input:

3 3
0 1 5
1 2 3
0 2 1

Output:

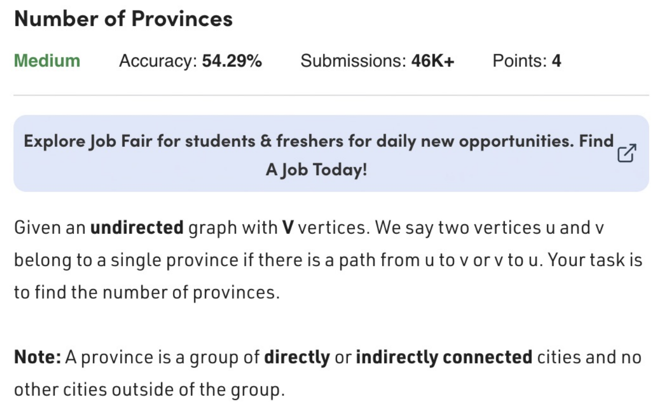
4

Explanation:

The Spanning Tree resulting in a weight of 4 is shown above.

```
class DisjointSet {
  constructor(n) {
    this.rank = new Array(n + 1).fill(0);
    this.parent = new Array(n + 1).fill(0).map((_, i) => i);
    this.size = new Array(n + 1).fill(1);
  }

  findUPar(node) {
    if (node === this.parent[node]) {
      return node;
    }
    return (this.parent[node] = this.findUPar(this.parent[node]));
  }

  unionByRank(u, v) {
    const ulp_u = this.findUPar(u);
    const ulp_v = this.findUPar(v);
    if (ulp_u === ulp_v) return;
    if (this.rank[ulp_u] < this.rank[ulp_v]) {
      this.parent[ulp_u] = ulp_v;
    } else if (this.rank[ulp_v] < this.rank[ulp_u]) {
      this.parent[ulp_v] = ulp_u;
    } else {
      this.parent[ulp_v] = ulp_u;
      this.rank[ulp_u]++;
    }
  }

  unionBySize(u, v) {
    const ulp_u = this.findUPar(u);
    const ulp_v = this.findUPar(v);
    if (ulp_u === ulp_v) return;
    if (this.size[ulp_u] < this.size[ulp_v]) {
      this.parent[ulp_u] = ulp_v;
      this.size[ulp_v] += this.size[ulp_u];
    } else {
      this.parent[ulp_v] = ulp_u;
      this.size[ulp_u] += this.size[ulp_v];
    }
  }
}

class Solution {
  spanningTree(V, adj) {
    const edges = [];

    for (let i = 0; i < V; i++) {
      for (const it of adj[i]) {
        const adjNode = it[0];
        const wt = it[1];
        const node = i;

        edges.push({ wt, nodePair: { node, adjNode } });
      }
    }

    const ds = new DisjointSet(V);
    edges.sort((a, b) => a.wt - b.wt);
    let mstWt = 0;

    for (const it of edges) {
      const { wt, nodePair: { node, adjNode } } = it;

      if (ds.findUPar(node) !== ds.findUPar(adjNode)) {
        mstWt += wt;
        ds.unionBySize(node, adjNode);
      }
    }

    return mstWt;
  }
}

const main = () => {
    const V = 5;
    const edges = [
        [0, 1, 2],
        [0, 2, 1],
        [1, 2, 1],
        [2, 3, 2],
        [3, 4, 1],
        [4, 2, 2],
    ];
    const adj = new Array(V).fill(0).map(() => []);

    for (const it of edges) {
        const tmp = [it[1], it[2]];
        adj[it[0]].push(tmp);

        const tmp2 = [it[0], it[2]];
        adj[it[1]].push(tmp2);
    }

    const obj = new Solution();
    const mstWt = obj.spanningTree(V, adj);
    console.log("The sum of all the edge weights:", mstWt);
};

main();
// The sum of all the edge weights: 5
```

lec48. number of provinces → disjoint

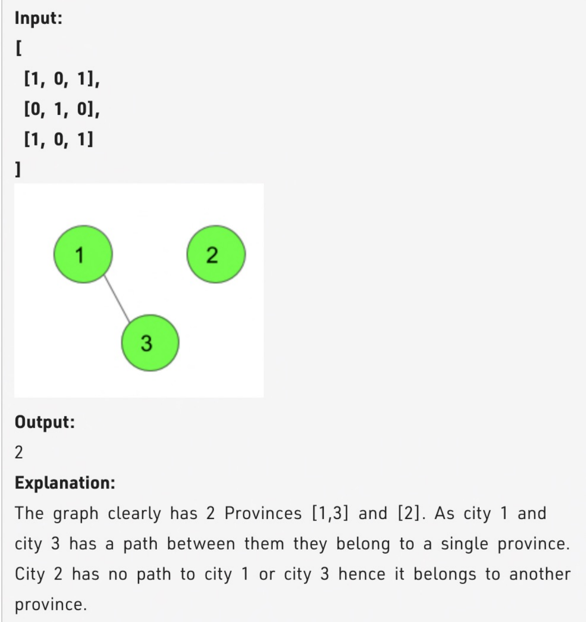


Number of Provinces

Medium Accuracy: **54.29%** Submissions: **46K+** Points: **4**

Given an **undirected** graph with **V** vertices. We say two vertices u and v belong to a single province if there is a path from u to v or v to u. Your task is to find the number of provinces.

Note: A province is a group of **directly** or **indirectly connected** cities and no other cities outside of the group.



Input:

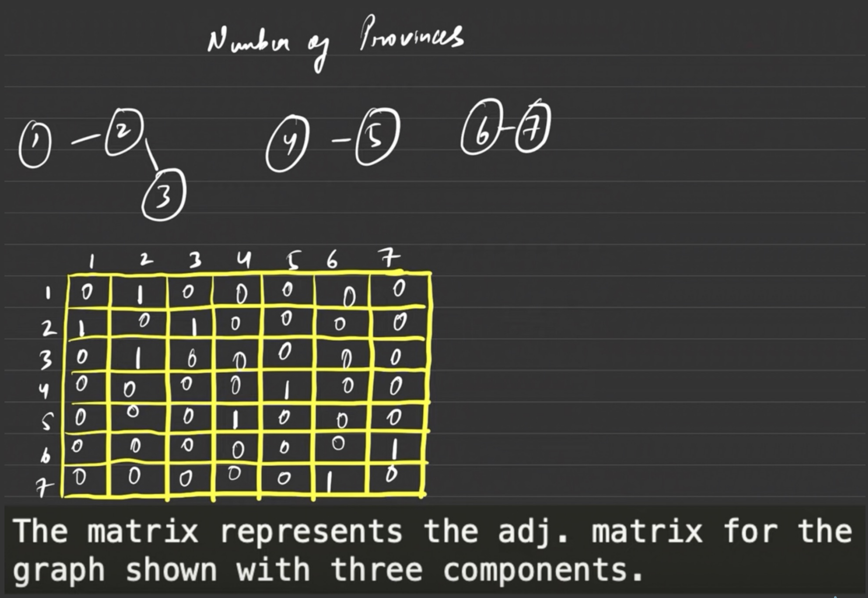
```
[
 [1, 0, 1],
 [0, 1, 0],
 [1, 0, 1]
]
```

Output:

2

Explanation:

The graph clearly has 2 Provinces [1,3] and [2]. As city 1 and city 3 has a path between them they belong to a single province. City 2 has no path to city 1 or city 3 hence it belongs to another province.



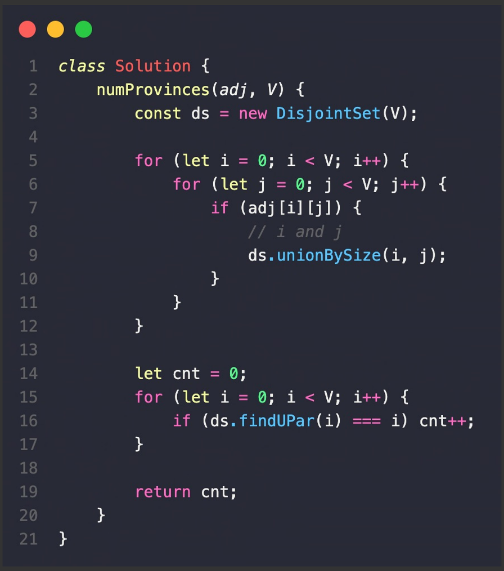
Number of Provinces

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	0	0	0	0	0
4	0	0	0	0	1	0	0
5	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1
7	0	0	0	0	0	1	0

The matrix represents the adj. matrix for the graph shown with three components.

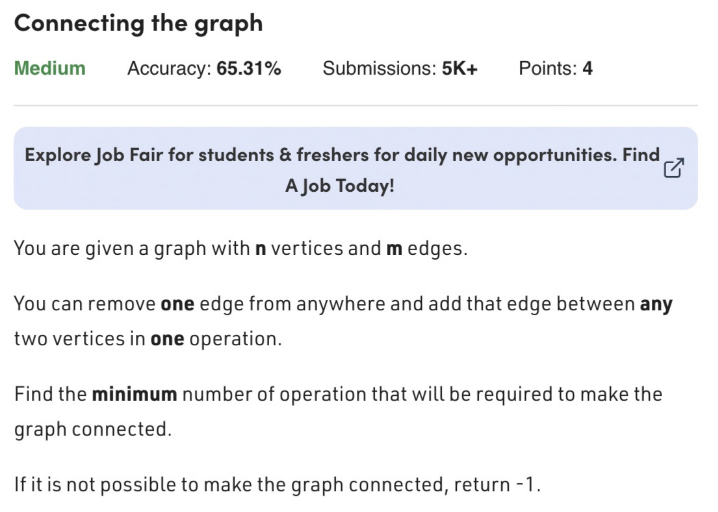
note → count the number of unique ultimate provinces.

```
class DisjointSet {
  constructor(n) {
    this.rank = new Array(n + 1).fill(0);
    this.parent = new Array(n + 1).fill(0).map((_, i) => i);
    this.size = new Array(n + 1).fill(1);
  }

  findUPar(node) {
    if (node === this.parent[node]) {
      return node;
    }
    return (this.parent[node] = this.findUPar(this.parent[node]));
  }

  unionByRank(u, v) {
    const ulp_u = this.findUPar(u);
    const ulp_v = this.findUPar(v);
    if (ulp_u === ulp_v) return;
    if (this.rank[ulp_u] < this.rank[ulp_v]) {
      this.parent[ulp_u] = ulp_v;
    } else if (this.rank[ulp_v] < this.rank[ulp_u]) {
      this.parent[ulp_v] = ulp_u;
    } else {
      this.parent[ulp_v] = ulp_u;
      this.rank[ulp_u]++;
    }
  }

  unionBySize(u, v) {
    const ulp_u = this.findUPar(u);
    const ulp_v = this.findUPar(v);
    if (ulp_u === ulp_v) return;
    if (this.size[ulp_u] < this.size[ulp_v]) {
      this.parent[ulp_u] = ulp_v;
      this.size[ulp_v] += this.size[ulp_u];
    } else {
      this.parent[ulp_v] = ulp_u;
      this.size[ulp_u] += this.size[ulp_v];
    }
  }
}
```



```
class Solution {
    numProvinces(adj, V) {
        const ds = new DisjointSet(V);

        for (let i = 0; i < V; i++) {
            for (let j = 0; j < V; j++) {
                if (adj[i][j]) {
                    // i and j
                    ds.unionBySize(i, j);
                }
            }
        }

        let cnt = 0;
        for (let i = 0; i < V; i++) {
            if (ds.findUPar(i) === i) cnt++;
        }

        return cnt;
    }
}
```

lec 49 number of operations to make network connected



Connecting the graph

Medium Accuracy: **65.31%** Submissions: **5K+** Points: **4**

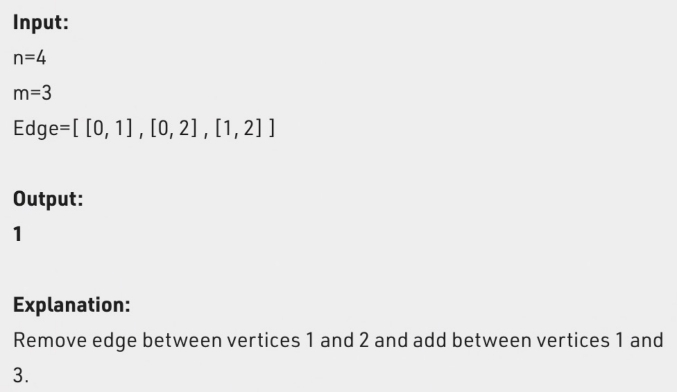
Explore Job Fair for students & freshers for daily new opportunities. Find A Job Today!

You are given a graph with **n** vertices and **m** edges.

You can remove **one** edge from anywhere and add that edge between **any** two vertices in **one** operation.

Find the **minimum** number of operation that will be required to make the graph connected.

If it is not possible to make the graph connected, return -1.



Input:

n=4

m=3

Edge=[[0, 1] , [0, 2] , [1, 2]]

Output:

1

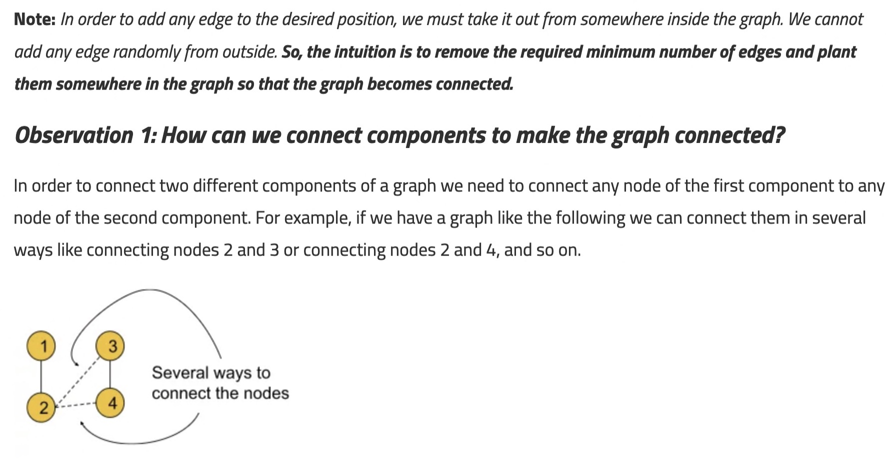
Explanation:

Remove edge between vertices 1 and 2 and add between vertices 1 and 3.

remove existing edge to connect/make new edge?

num of connected component → nC

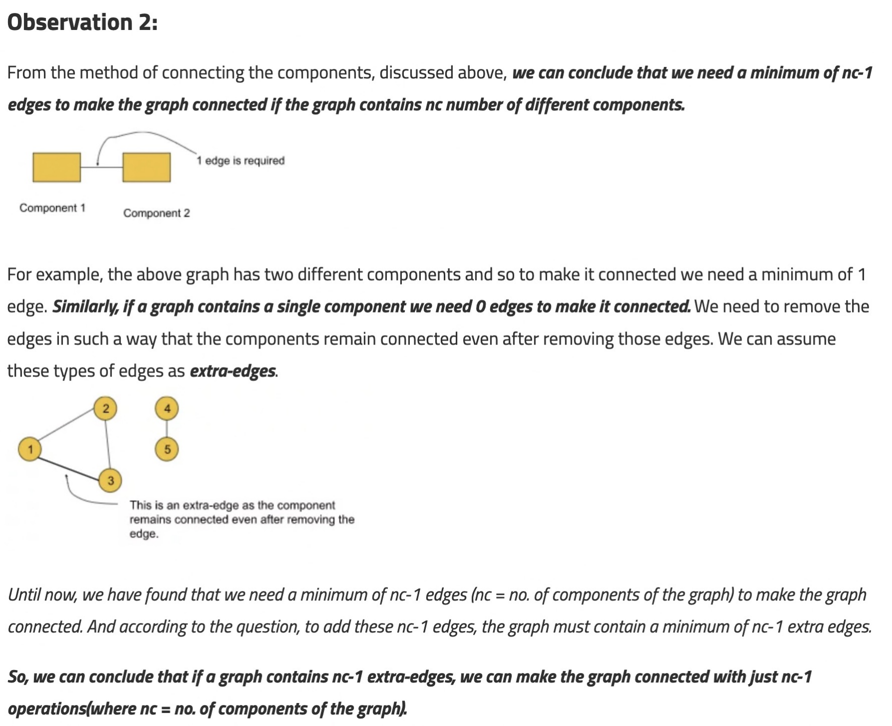
ans → nC-1



Note: *In order to add any edge to the desired position, we must take it out from somewhere inside the graph. We cannot add any edge randomly from outside.* ***So, the intuition is to remove the required minimum number of edges and plant them somewhere in the graph so that the graph becomes connected.***

Observation 1: How can we connect components to make the graph connected?

In order to connect two different components of a graph we need to connect any node of the first component to any node of the second component. For example, if we have a graph like the following we can connect them in several ways like connecting nodes 2 and 3 or connecting nodes 2 and 4, and so on.



Observation 2:

From the method of connecting the components, discussed above, ***we can conclude that we need a minimum of nc-1 edges to make the graph connected if the graph contains nc number of different components.***

For example, the above graph has two different components and so to make it connected we need a minimum of 1 edge. ***Similarly, if a graph contains a single component we need 0 edges to make it connected.*** We need to remove the edges in such a way that the components remain connected even after removing those edges. We can assume these types of edges as ***extra-edges***.

Until now, we have found that we need a minimum of nc-1 edges (nc = no. of components of the graph) to make the graph connected. And according to the question, to add these nc-1 edges, the graph must contain a minimum of nc-1 extra edges.

So, we can conclude that if a graph contains nc-1 extra-edges, we can make the graph connected with just nc-1 operations(where nc = no. of components of the graph).

Approach:

In order to solve this question we will first find out the number of extra-edges and then we will find out the number of components of the graph. We will be using the Disjoint Set data structure to do so.

The algorithm steps are the following:

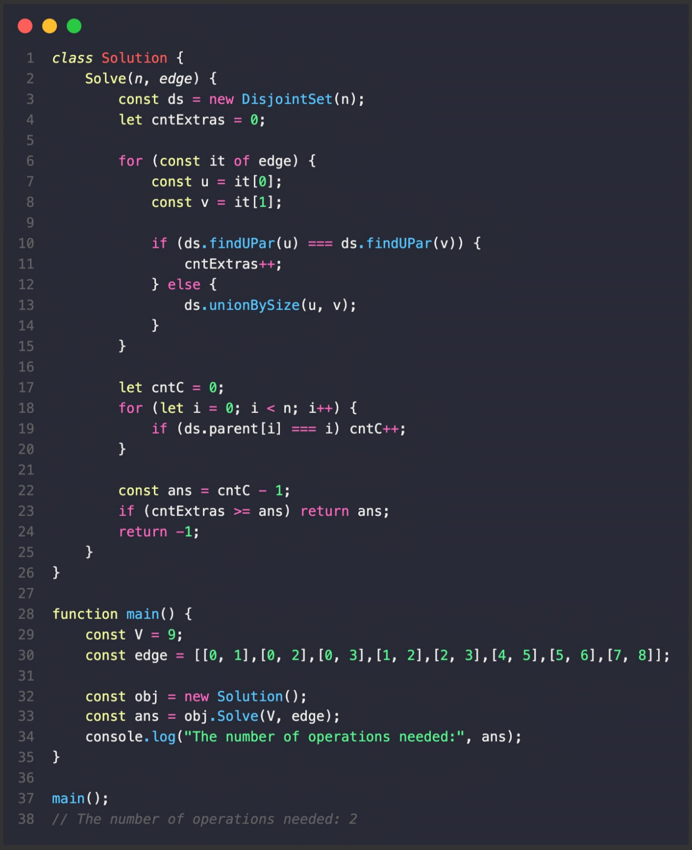
1. First we need to extract all the edge information (***If not already given***) in the form of the pair (u, v) where u = starting node and v = ending node. We should store all the edge information in an array.
2. Then we will iterate through the array selecting every pair and checking the following:
    1. **If the ultimate parent of u and v(checked using the findPar() method of the Disjoint set) becomes the same**, we should increase the count of extra-edges by 1.
       Because the same ultimate parent means the nodes are already connected and so we can consider the current edge as an extra edge.
    2. But if the ultimate parents are different, then we should apply the union(either unionBySize() or unionByRank()) method on those two nodes.
3. Thus we will get the count of the extra edges. Now it's time to count the number of components. In order to do so, we will just count the number of the nodes that are the ultimate parent of themselves.
4. We will iterate over all the nodes and for each node, we will check the following:
    1. If the node is the ultimate parent of itself, we will increase the count of components by 1.
    2. Otherwise, we will continue to the next node.

This checking will be done using the parent array inside the Disjoint set.

5. Finally, we will check the count of extra edges and the number of components. If the count of extra-edges is greater or the same, we will return the answer that is (number of components – 1), and otherwise, we will return -1.

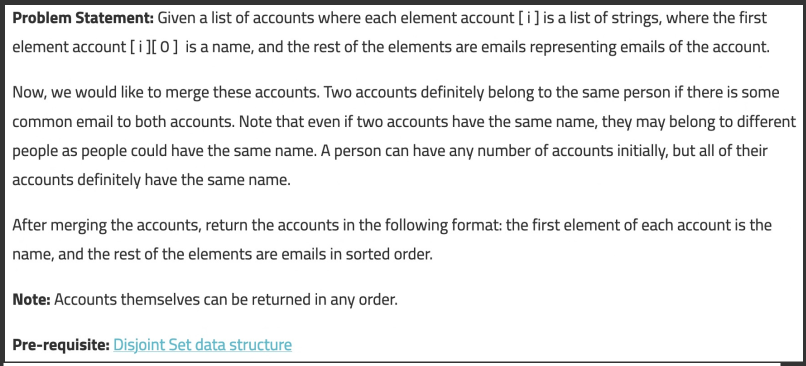
Time Complexity: O(E*4α)+O(N*4α) where E = no. of edges and N = no. of nodes. The first term is to calculate the number of extra edges and the second term is to count the number of components. 4α is for the disjoint set operation we have used and this term is so small that it can be considered constant.

Space Complexity: O(2N) where N = no. of nodes. 2N for the two arrays(parent and size) of size N we have used inside the disjoint set.



```
class Solution {
    Solve(n, edge) {
        const ds = new DisjointSet(n);
        let cntExtras = 0;

        for (const it of edge) {
            const u = it[0];
            const v = it[1];

            if (ds.findUPar(u) === ds.findUPar(v)) {
                cntExtras++;
            } else {
                ds.unionBySize(u, v);
            }
        }

        let cntC = 0;
        for (let i = 0; i < n; i++) {
            if (ds.parent[i] === i) cntC++;
        }

        const ans = cntC - 1;
        if (cntExtras >= ans) return ans;
        return -1;
    }
}

function main() {
    const V = 9;
    const edge = [[0, 1],[0, 2],[0, 3],[1, 2],[2, 3],[4, 5],[5, 6],[7, 8]];

    const obj = new Solution();
    const ans = obj.Solve(V, edge);
    console.log("The number of operations needed:", ans);
}

main();
// The number of operations needed: 2
```

lec50 → Accounts merge- DSU



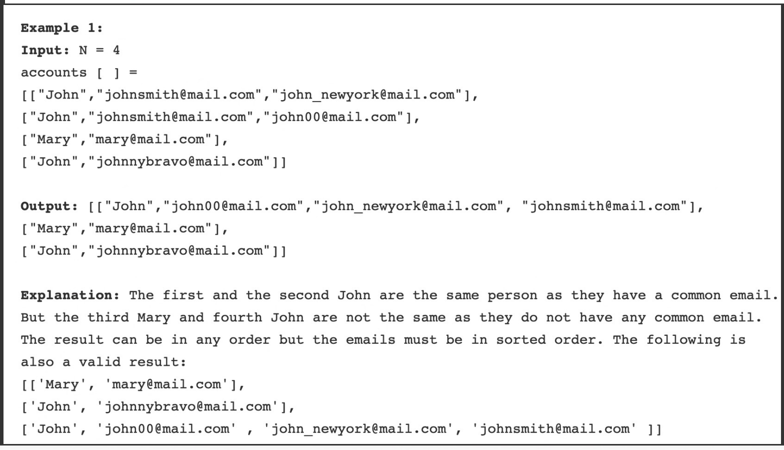
Problem Statement: Given a list of accounts where each element account [i] is a list of strings, where the first element account [i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

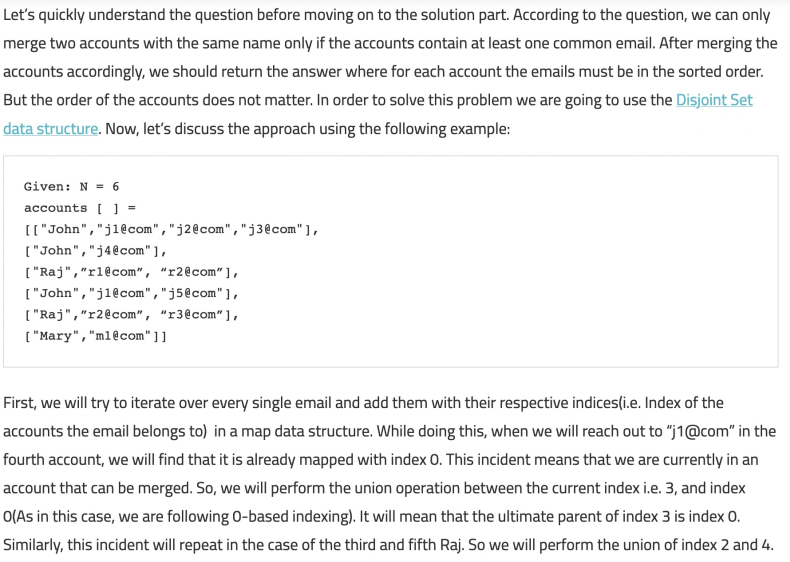
After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order.

Note: Accounts themselves can be returned in any order.

Pre-requisite: Disjoint Set data structure



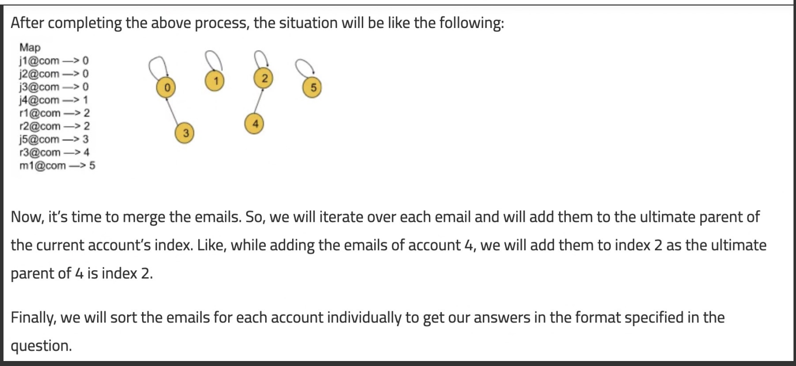
```
Example 1:
Input: N = 4
accounts [ ] =
[["John","johnsmith@mail.com","john_newyork@mail.com"],
["John","johnsmith@mail.com","john00@mail.com"],
["Mary","mary@mail.com"],
["John","johnnybravo@mail.com"]]

Output: [["John","john00@mail.com","john_newyork@mail.com", "johnsmith@mail.com"],
["Mary","mary@mail.com"],
["John","johnnybravo@mail.com"]]

Explanation: The first and the second John are the same person as they have a common email.
But the third Mary and fourth John are not the same as they do not have any common email.
The result can be in any order but the emails must be in sorted order. The following is
also a valid result:
[['Mary', 'mary@mail.com'],
['John', 'johnnybravo@mail.com'],
['John', 'john00@mail.com' , 'john_newyork@mail.com', 'johnsmith@mail.com' ]]
```



Let's quickly understand the question before moving on to the solution part. According to the question, we can only merge two accounts with the same name only if the accounts contain at least one common email. After merging the accounts accordingly, we should return the answer where for each account the emails must be in the sorted order. But the order of the accounts does not matter. In order to solve this problem we are going to use the Disjoint Set data structure. Now, let's discuss the approach using the following example:

```
Given: N = 6
accounts [ ] =
[["John","j1@com","j2@com","j3@com"],
["John","j4@com"],
["Raj",”r1@com”, “r2@com”],
["John","j1@com","j5@com"],
["Raj",”r2@com”, “r3@com”],
["Mary","m1@com"]]
```

First, we will try to iterate over every single email and add them with their respective indices(i.e. Index of the accounts the email belongs to) in a map data structure. While doing this, when we will reach out to "j1@com" in the fourth account, we will find that it is already mapped with index 0. This incident means that we are currently in an account that can be merged. So, we will perform the union operation between the current index i.e. 3, and index 0(As in this case, we are following 0-based indexing). It will mean that the ultimate parent of index 3 is index 0. Similarly, this incident will repeat in the case of the third and fifth Raj. So we will perform the union of index 2 and 4.



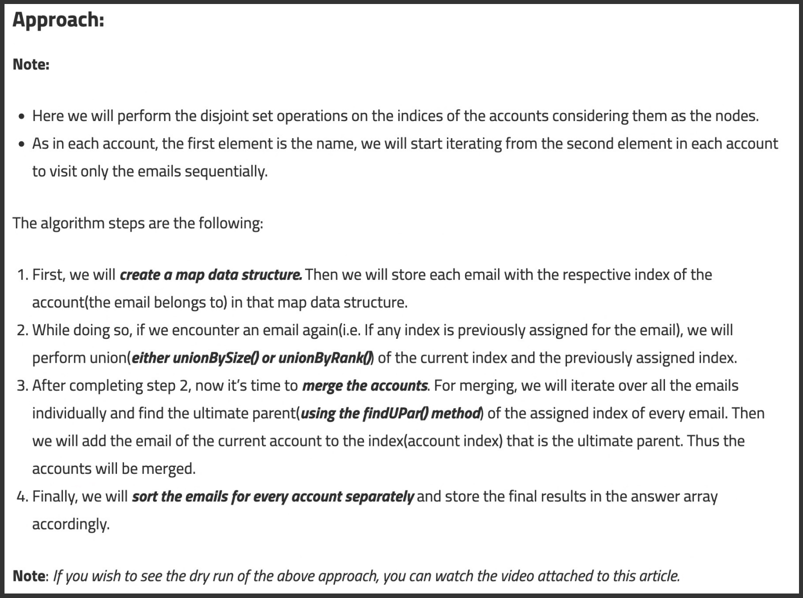
After completing the above process, the situation will be like the following:

Map
j1@com —> 0
j2@com —> 0
j3@com —> 0
j4@com —> 1
r1@com —> 2
r2@com —> 2
j5@com —> 3
r3@com —> 4
m1@com —> 5

Now, it's time to merge the emails. So, we will iterate over each email and will add them to the ultimate parent of the current account's index. Like, while adding the emails of account 4, we will add them to index 2 as the ultimate parent of 4 is index 2.

Finally, we will sort the emails for each account individually to get our answers in the format specified in the question.

```javascript
class Solution {
    accountsMerge(details) {
        const n = details.length;
        const ds = new DisjointSet(n);
        details.sort();

        const mapMailNode = new Map();
        for (let i = 0; i < n; i++) {
            for (let j = 1; j < details[i].length; j++) {
                const mail = details[i][j];
                if (!mapMailNode.has(mail)) {
                    mapMailNode.set(mail, i);
                } else {
                    ds.unionBySize(i, mapMailNode.get(mail));
                }
            }
        }

        const mergedMail = Array.from({ length: n }, () => []);
        for (const [mail, nodeIndex] of mapMailNode.entries()) {
            const node = ds.findUPar(nodeIndex);
            mergedMail[node].push(mail);
        }

        const ans = [];

        for (let i = 0; i < n; i++) {
            if (mergedMail[i].length === 0) continue;
            mergedMail[i].sort();

            const temp = [details[i][0], ...mergedMail[i]];
            ans.push(temp);
        }
        ans.sort();

        return ans;
    }
}

function main() {
    const accounts = [
        ["John", "j1@com", "j2@com", "j3@com"],
        ["John", "j4@com"],
        ["Raj", "r1@com", "r2@com"],
        ["John", "j1@com", "j5@com"],
        ["Raj", "r2@com", "r3@com"],
        ["Mary", "m1@com"],
    ];

    const obj = new Solution();
    const ans = obj.accountsMerge(accounts);
    for (const acc of ans) {
        console.log(`${acc[0]}: ${acc.slice(1).join(" ")}`);
    }
}

main();
/*
John: j1@com j2@com j3@com j5@com
John: j4@com
Mary: m1@com
Raj: r1@com r2@com r3@com
*/
```



Approach:

Note:

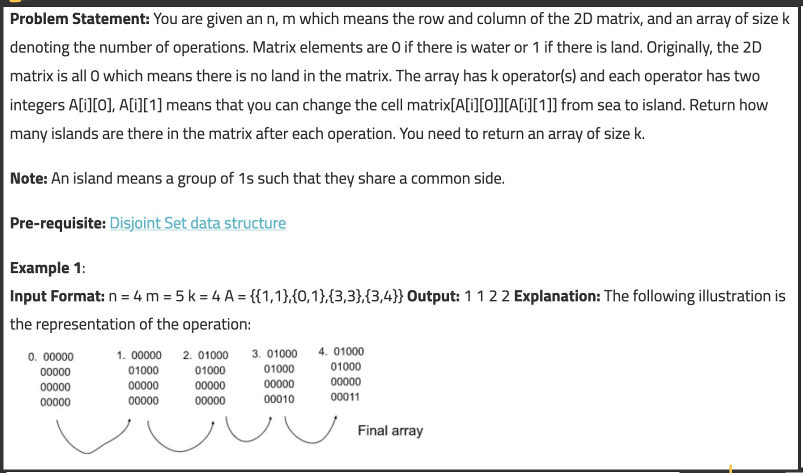
- Here we will perform the disjoint set operations on the indices of the accounts considering them as the nodes.
- As in each account, the first element is the name, we will start iterating from the second element in each account to visit only the emails sequentially.

The algorithm steps are the following:

1. First, we will ***create a map data structure.*** Then we will store each email with the respective index of the account(the email belongs to) in that map data structure.
2. While doing so, if we encounter an email again(i.e. If any index is previously assigned for the email), we will perform union(***either unionBySize() or unionByRank()***) of the current index and the previously assigned index.
3. After completing step 2, now it's time to ***merge the accounts***. For merging, we will iterate over all the emails individually and find the ultimate parent(***using the findUPar() method***) of the assigned index of every email. Then we will add the email of the current account to the index(account index) that is the ultimate parent. Thus the accounts will be merged.
4. Finally, we will ***sort the emails for every account separately*** and store the final results in the answer array accordingly.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

lec 51 => numbers of islands2 - online queries - DSU



Problem Statement: You are given an n, m which means the row and column of the 2D matrix, and an array of size k denoting the number of operations. Matrix elements are 0 if there is water or 1 if there is land. Originally, the 2D matrix is all 0 which means there is no land in the matrix. The array has k operator(s) and each operator has two integers A[i][0], A[i][1] means that you can change the cell matrix[A[i][0]][A[i][1]] from sea to island. Return how many islands are there in the matrix after each operation. You need to return an array of size k.

Note: An island means a group of 1s such that they share a common side.

Pre-requisite: Disjoint Set data structure

Example 1:
Input Format: n = 4 m = 5 k = 4 A = {{1,1},{0,1},{3,3},{3,4}} **Output:** 1 1 2 2 **Explanation:** The following illustration is the representation of the operation:

0. 00000
00000
00000
00000

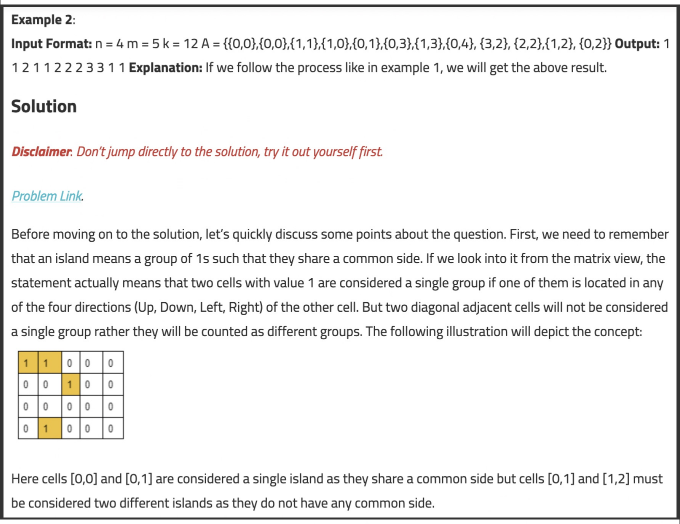
1. 00000
01000
00000
00000

2. 01000
01000
00000
00000

3. 01000
01000
00000
00010

4. 01000
01000
00000
00011

Final array



Example 2:
Input Format: n = 4 m = 5 k = 12 A = {{0,0},{0,0},{1,1},{1,0},{0,1},{0,3},{1,3},{0,4}, {3,2}, {2,2},{1,2}, {0,2}} **Output:** 1 2 1 1 2 2 2 3 3 1 1 **Explanation:** If we follow the process like in example 1, we will get the above result.

Solution

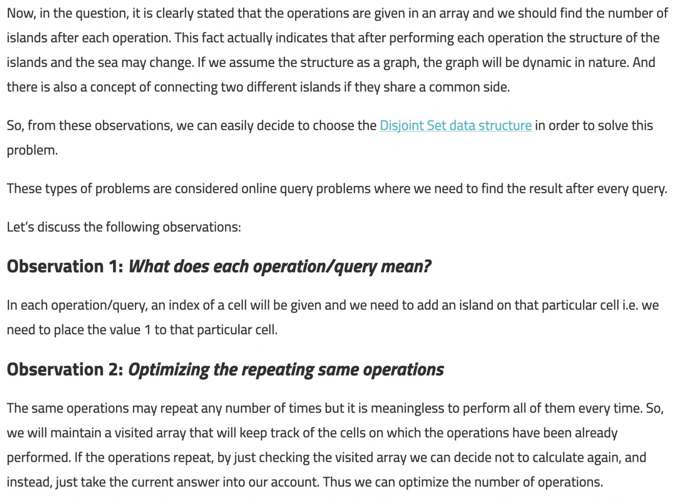
Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link.

Before moving on to the solution, let's quickly discuss some points about the question. First, we need to remember that an island means a group of 1s such that they share a common side. If we look into it from the matrix view, the statement actually means that two cells with value 1 are considered a single group if one of them is located in any of the four directions (Up, Down, Left, Right) of the other cell. But two diagonal adjacent cells will not be considered a single group rather they will be counted as different groups. The following illustration will depict the concept:

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

Here cells [0,0] and [0,1] are considered a single island as they share a common side but cells [0,1] and [1,2] must be considered two different islands as they do not have any common side.



Now, in the question, it is clearly stated that the operations are given in an array and we should find the number of islands after each operation. This fact actually indicates that after performing each operation the structure of the islands and the sea may change. If we assume the structure as a graph, the graph will be dynamic in nature. And there is also a concept of connecting two different islands if they share a common side.

So, from these observations, we can easily decide to choose the Disjoint Set data structure in order to solve this problem.

These types of problems are considered online query problems where we need to find the result after every query.

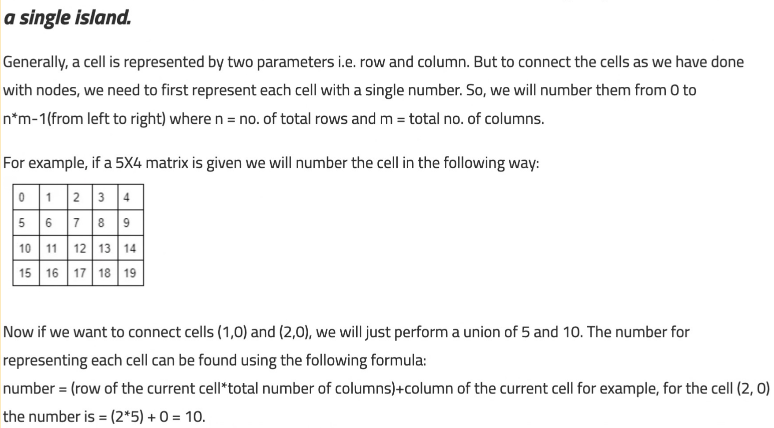
Let's discuss the following observations:

Observation 1: *What does each operation/query mean?*

In each operation/query, an index of a cell will be given and we need to add an island on that particular cell i.e. we need to place the value 1 to that particular cell.

Observation 2: *Optimizing the repeating same operations*

The same operations may repeat any number of times but it is meaningless to perform all of them every time. So, we will maintain a visited array that will keep track of the cells on which the operations have been already performed. If the operations repeat, by just checking the visited array we can decide not to calculate again, and instead, just take the current answer into our account. Thus we can optimize the number of operations.



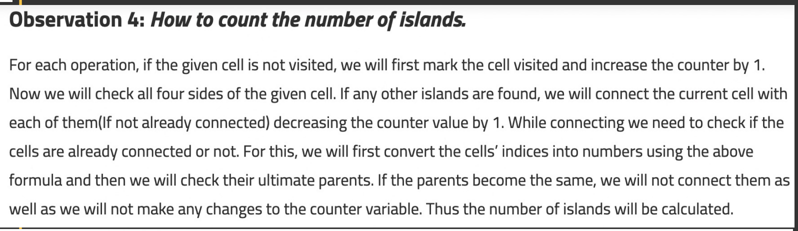
Observation 3: *How to connect cells to include them in the same group or consider them a single island.*

Generally, a cell is represented by two parameters i.e. row and column. But to connect the cells as we have done with nodes, we need to first represent each cell with a single number. So, we will number them from 0 to n*m-1(from left to right) where n = no. of total rows and m = total no. of columns.

For example, if a 5X4 matrix is given we will number the cell in the following way:

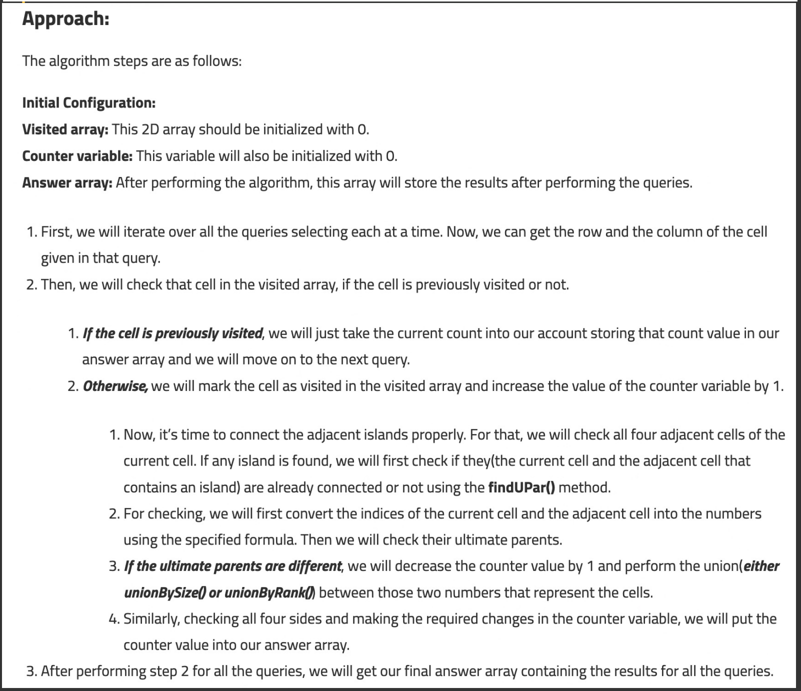
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula:
number = (row of the current cell*total number of columns)+column of the current cell for example, for the cell (2, 0) the number is = (2*5) + 0 = 10.



Observation 4: *How to count the number of islands.*

For each operation, if the given cell is not visited, we will first mark the cell visited and increase the counter by 1. Now we will check all four sides of the given cell. If any other islands are found, we will connect the current cell with each of them(If not already connected) decreasing the counter value by 1. While connecting we need to check if the cells are already connected or not. For this, we will first convert the cells' indices into numbers using the above formula and then we will check their ultimate parents. If the parents become the same, we will not connect them as well as we will not make any changes to the counter variable. Thus the number of islands will be calculated.



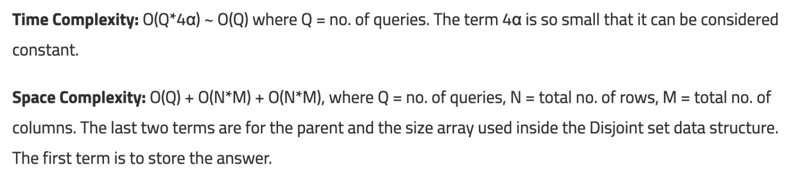
Approach:

The algorithm steps are as follows:

Initial Configuration:
Visited array: This 2D array should be initialized with 0.
Counter variable: This variable will also be initialized with 0.
Answer array: After performing the algorithm, this array will store the results after performing the queries.

1. First, we will iterate over all the queries selecting each at a time. Now, we can get the row and the column of the cell given in that query.
2. Then, we will check that cell in the visited array, if the cell is previously visited or not.
    1. ***If the cell is previously visited***, we will just take the current count into our account storing that count value in our answer array and we will move on to the next query.
    2. ***Otherwise,*** we will mark the cell as visited in the visited array and increase the value of the counter variable by 1.
        1. Now, it's time to connect the adjacent islands properly. For that, we will check all four adjacent cells of the current cell. If any island is found, we will first check if they(the current cell and the adjacent cell that contains an island) are already connected or not using the **findUPar()** method.
        2. For checking, we will first convert the indices of the current cell and the adjacent cell into the numbers using the specified formula. Then we will check their ultimate parents.
        3. ***If the ultimate parents are different***, we will decrease the counter value by 1 and perform the union(***either unionBySize() or unionByRank()***) between those two numbers that represent the cells.
        4. Similarly, checking all four sides and making the required changes in the counter variable, we will put the counter value into our answer array.
3. After performing step 2 for all the queries, we will get our final answer array containing the results for all the queries.

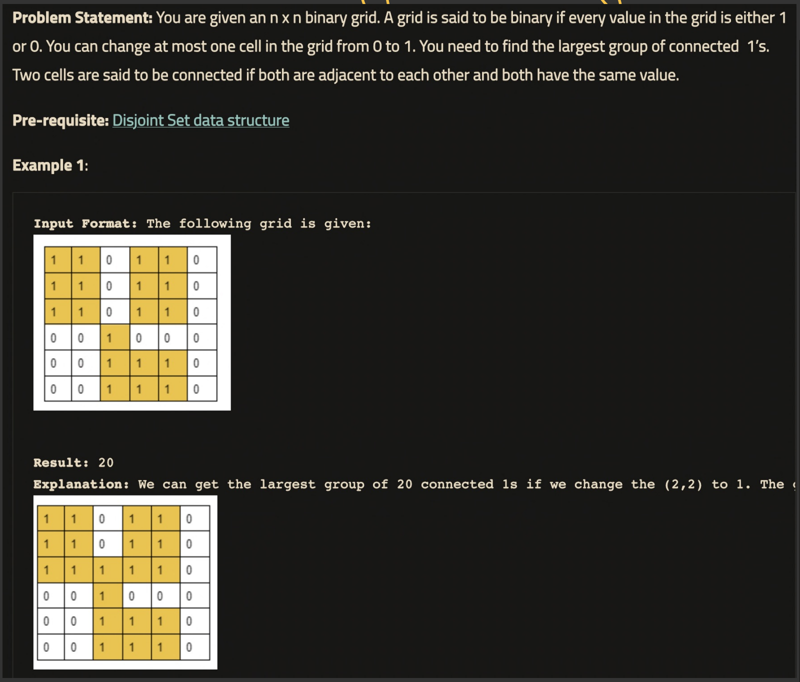
```
class Solution {
    isValid(adjr, adjc, n, m) {
        return adjr >= 0 && adjr < n && adjc >= 0 && adjc < m;
    }

    numOfIslands(n, m, operators) {
        const ds = new DisjointSet(n * m);
        const vis = Array.from({ length: n }, () => Array(m).fill(0));
        let cnt = 0;
        const ans = [];

        for (const it of operators) {
            const row = it[0];
            const col = it[1];
            if (vis[row][col] === 1) {
                ans.push(cnt);
                continue;
            }
            vis[row][col] = 1;
            cnt++;

            const dr = [-1, 0, 1, 0];
            const dc = [0, 1, 0, -1];

            for (let ind = 0; ind < 4; ind++) {
                const adjr = row + dr[ind];
                const adjc = col + dc[ind];

                if (this.isValid(adjr, adjc, n, m)) {
                    if (vis[adjr][adjc] === 1) {
                        const nodeNo = row * m + col;
                        const adjNodeNo = adjr * m + adjc;
                        if (ds.findUPar(nodeNo) !== ds.findUPar(adjNodeNo)) {
                            cnt--;
                            ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.push(cnt);
        }
        return ans;
    }
}

function main() {
    const n = 4;
    const m = 5;
    const operators = [
        [0, 0],
        [0, 0],
        [1, 1],
        [1, 0],
        [0, 1],
        [0, 3],
        [1, 3],
        [0, 4],
        [3, 2],
        [2, 2],
        [1, 2],
        [0, 2],
    ];

    const obj = new Solution();
    const ans = obj.numOfIslands(n, m, operators);
    console.log(ans.join(" "));
    // 1 1 2 1 1 2 2 2 3 3 1 1
}

main();
```



Time Complexity: O(Q*4α) ~ O(Q) where Q = no. of queries. The term 4α is so small that it can be considered constant.

Space Complexity: O(Q) + O(N*M) + O(N*M), where Q = no. of queries, N = total no. of rows, M = total no. of columns. The last two terms are for the parent and the size array used inside the Disjoint set data structure. The first term is to store the answer.

lec 52. making a large island — DSU



Problem Statement: You are given an n x n binary grid. A grid is said to be binary if every value in the grid is either 1 or 0. You can change at most one cell in the grid from 0 to 1. You need to find the largest group of connected 1's. Two cells are said to be connected if both are adjacent to each other and both have the same value.

Pre-requisite: Disjoint Set data structure

Example 1:

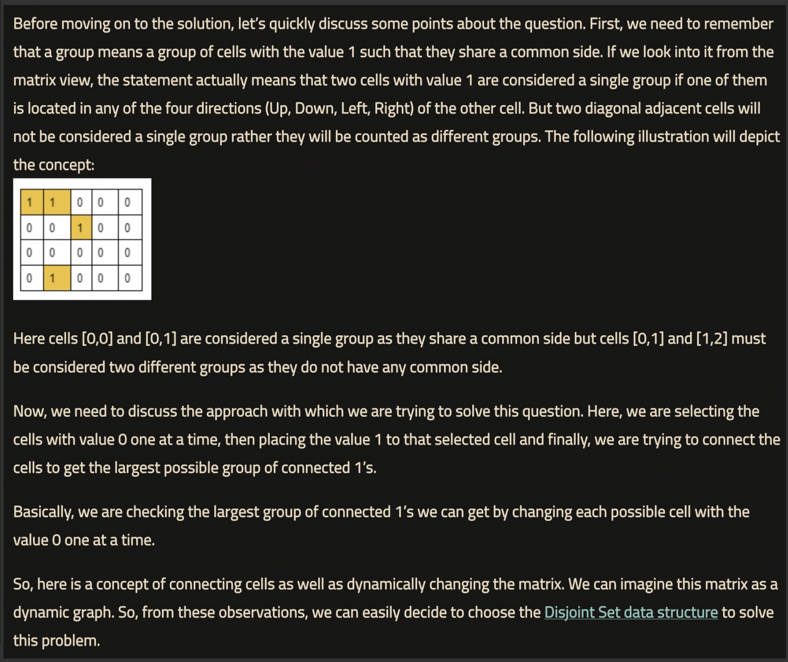
Input Format: The following grid is given:

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

Result: 20

Explanation: We can get the largest group of 20 connected 1s if we change the (2,2) to 1. The

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



Before moving on to the solution, let's quickly discuss some points about the question. First, we need to remember that a group means a group of cells with the value 1 such that they share a common side. If we look into it from the matrix view, the statement actually means that two cells with value 1 are considered a single group if one of them is located in any of the four directions (Up, Down, Left, Right) of the other cell. But two diagonal adjacent cells will not be considered a single group rather they will be counted as different groups. The following illustration will depict the concept:

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

Here cells [0,0] and [0,1] are considered a single group as they share a common side but cells [0,1] and [1,2] must be considered two different groups as they do not have any common side.

Now, we need to discuss the approach with which we are trying to solve this question. Here, we are selecting the cells with value 0 one at a time, then placing the value 1 to that selected cell and finally, we are trying to connect the cells to get the largest possible group of connected 1's.

Basically, we are checking the largest group of connected 1's we can get by changing each possible cell with the value 0 one at a time.

So, here is a concept of connecting cells as well as dynamically changing the matrix. We can imagine this matrix as a dynamic graph. So, from these observations, we can easily decide to choose the Disjoint Set data structure to solve this problem.

Observation 1: *How to connect cells to include them in the same group.*

Generally, a cell is represented by two parameters i.e. row and column. But to connect the cells as we have done with nodes, we need to first represent each cell with a single number. So, we will number them from 0 to n*m-1(from left to right) where n = no. of total rows and m = total no. of columns.

For example, if a 5X4 matrix is given we will number the cell in the following way:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula:
node number = (row of the current cell*total number of columns)+column of the current cell for example, for the cell (2, 0) the number is = (2*5) + 0 = 10.

Observation 2: *How to find the cell in which if we invert the value, we will get the largest possible group of connected 1s.*

In order to find the cell, we will follow the brute force approach. We will check for every possible cell with a value of 0 one by one and we will try to figure out the largest group we can get after inverting that particular cell to 1 in each case. Among all the answers we will find the cell that creates the largest possible group.

Now, with these two observations, the following is our first approach:

We will first invert a cell from the value 0 to 1 and will check all its four adjacent cells(Up, Down, Left, Right). If any component/group exists, we will just connect the current cell to that adjacent component and add the component's size to our answer. Finally, checking all four cells, we will add an extra 1 to our answer for the current cell being included in the group, and then we will get the total size of the newly created group.

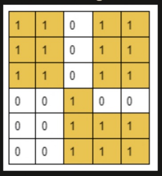
But How to get the size of an existing group/component of connected 1s:

In order to get the size of the existing groups, first, we need to create the existing group by connecting the cells with the value 1. To do so we will do a union of the two node numbers calculated using the above-specified formula if the cells contain 1 and they share a common side. Now after connecting all such cells we will get the different existing components.
Now ***to find the size of the components***, we will just find their ultimate parents and refer to the ultimate parent index of the size array inside the Disjoint Set data structure(size[ultimateParent]).

Thus we can calculate the size of the components/groups. But there exists an edge case in this approach.

Edge Case:

Here is the edge case. Let's understand it using the following example.

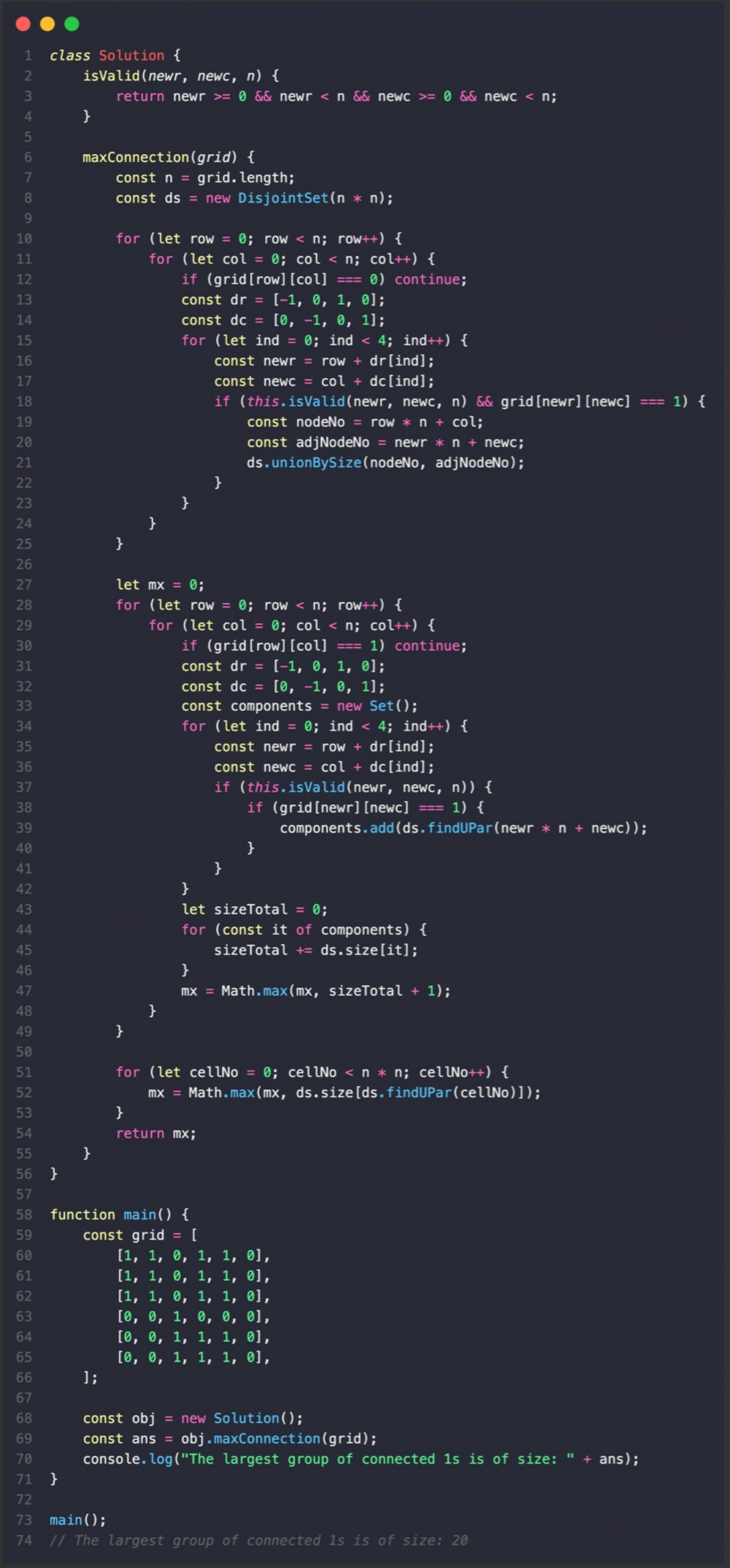


1	1	0	1	1
1	1	0	1	1
1	1	0	1	1
0	0	1	0	0
0	0	1	1	1
0	0	1	1	1

In this given grid, we will check for every cell with the value 0. When we come to cell (3,3), we will check all four adjacent cells to get the components' sizes. Now it will first add the component of size 7 in our answer while checking the left cell and will again add the same component while checking the downward cell. This is where the answer gets incorrect. ***So, to avoid this edge case, instead of adding the component sizes to our answer we will store the ultimate parents in a set data structure.*** This process will automatically discard the case of adding duplicate components. After that, to get the size of the ultimate parents we will just refer to the ultimate parent index of the size array inside the Disjoint Set data structure(size[ultimateParent]). Thus we will get the final answer.

The algorithm steps are as follows (***step 3 is very important***):

1. Our first objective is to connect all the nodes that have formed groups. In order to do so, we will visit each cell of the grid and check if it contains the value 1.
    1. If the value is 1, we will check all four adjacent cells of the current cell. If we find any adjacent cell with the same value 1, we will perform the union(***either unionBySize() or unionByRank()*** of the two node numbers that represent those two cells i.e. the current cell and the adjacent cell.
    2. Now, step 1 is completed.
2. Then, we will again visit each cell of the grid and check if it contains the value 0.
    1. If the value is 0, we will check all four adjacent cells of the current cell. If we found any cell with value 1, we will just insert the ultimate parent of that cell(using the ***findUPar()*** method) in the set data structure. This process will add the adjacent components to our answer.
    2. After doing so for all the adjacent cells containing 1, we will iterate through the set data structure and add the size of each ultimate parent(*referring to the size array inside the Disjoint Set data structure*) to our answer. Finally, we will add an extra 1 to our answer for the current cell being included in the group.
    3. Now, we will compare to get the maximum answer among all the previous answers we got for the previous cells with the value 0 and the current one.
3. But if the matrix does not contain any cell with 0, step 2 will not be executed. For that reason, we will just run a loop from node number 0 to n*n and for each node number, we will find the ultimate parent. After that, we will find the sizes of those ultimate parents and will take the size of the largest one.
4. Thus we will get the maximum size of the group of connected 1s stored in our answer.

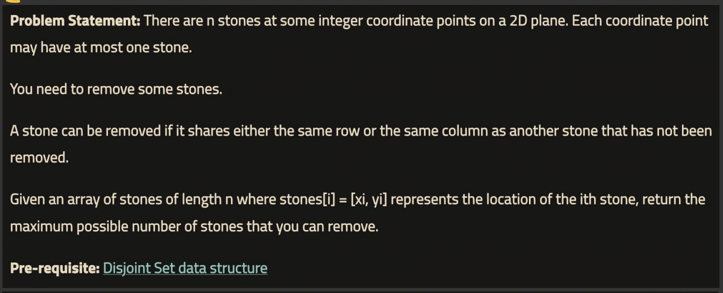


```
class Solution {
    isValid(newr, newc, n) {
        return newr >= 0 && newr < n && newc >= 0 && newc < n;
    }

    maxConnection(grid) {
        const n = grid.length;
        const ds = new DisjointSet(n * n);

        for (let row = 0; row < n; row++) {
            for (let col = 0; col < n; col++) {
                if (grid[row][col] === 0) continue;
                const dr = [-1, 0, 1, 0];
                const dc = [0, -1, 0, 1];
                for (let ind = 0; ind < 4; ind++) {
                    const newr = row + dr[ind];
                    const newc = col + dc[ind];
                    if (this.isValid(newr, newc, n) && grid[newr][newc] === 1) {
                        const nodeNo = row * n + col;
                        const adjNodeNo = newr * n + newc;
                        ds.unionBySize(nodeNo, adjNodeNo);
                    }
                }
            }
        }

        let mx = 0;
        for (let row = 0; row < n; row++) {
            for (let col = 0; col < n; col++) {
                if (grid[row][col] === 1) continue;
                const dr = [-1, 0, 1, 0];
                const dc = [0, -1, 0, 1];
                const components = new Set();
                for (let ind = 0; ind < 4; ind++) {
                    const newr = row + dr[ind];
                    const newc = col + dc[ind];
                    if (this.isValid(newr, newc, n)) {
                        if (grid[newr][newc] === 1) {
                            components.add(ds.findUPar(newr * n + newc));
                        }
                    }
                }
                let sizeTotal = 0;
                for (const it of components) {
                    sizeTotal += ds.size[it];
                }
                mx = Math.max(mx, sizeTotal + 1);
            }
        }

        for (let cellNo = 0; cellNo < n * n; cellNo++) {
            mx = Math.max(mx, ds.size[ds.findUPar(cellNo)]);
        }
        return mx;
    }
}

function main() {
    const grid = [
        [1, 1, 0, 1, 1, 0],
        [1, 1, 0, 1, 1, 0],
        [1, 1, 0, 1, 1, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 1, 1, 1, 0]
    ];

    const obj = new Solution();
    const ans = obj.maxConnection(grid);
    console.log("The largest group of connected 1s is of size: " + ans);
}

main();
// The largest group of connected 1s is of size: 20
```

Time Complexity: O(N^2)+O(N^2) ~ O(N^2) where N = total number of rows of the grid. Inside those nested loops, all the operations are taking apparently constant time. So, O(N^2) for the nested loop only, is the time complexity.

Space Complexity: O($2*N^2$) where N = the total number of rows of the grid. This is for the two arrays i.e. parent array and size array of size N^2 inside the Disjoint set.

lec53 → most stones Removed with same row or column – DSU



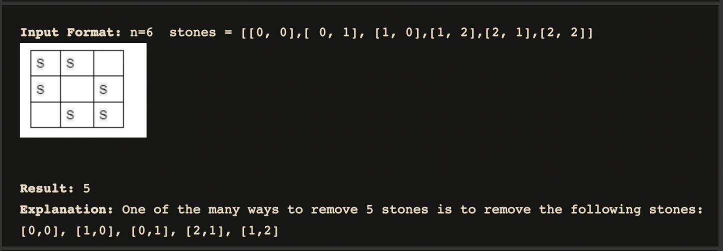
Problem Statement: There are n stones at some integer coordinate points on a 2D plane. Each coordinate point may have at most one stone.

You need to remove some stones.

A stone can be removed if it shares either the same row or the same column as another stone that has not been removed.

Given an array of stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return the maximum possible number of stones that you can remove.

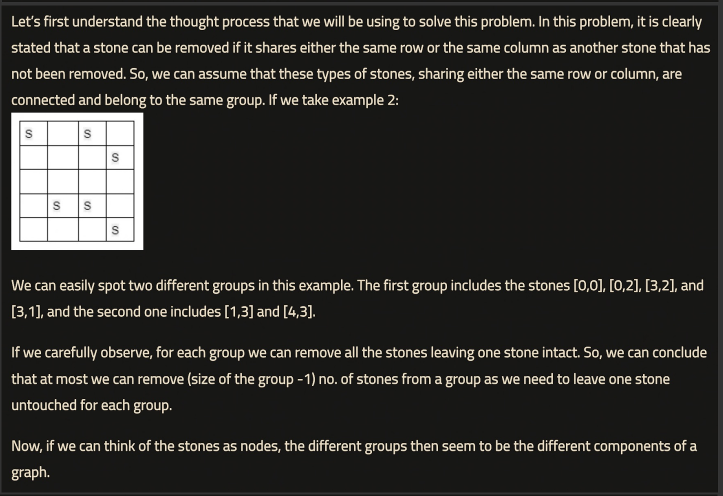
Pre-requisite: Disjoint Set data structure



```
Input Format: n=6  stones = [[0, 0],[ 0, 1], [1, 0],[1, 2],[2, 1],[2, 2]]
```

S	S	
S		S
	S	S

```
Result: 5
Explanation: One of the many ways to remove 5 stones is to remove the following stones:
[0,0], [1,0], [0,1], [2,1], [1,2]
```



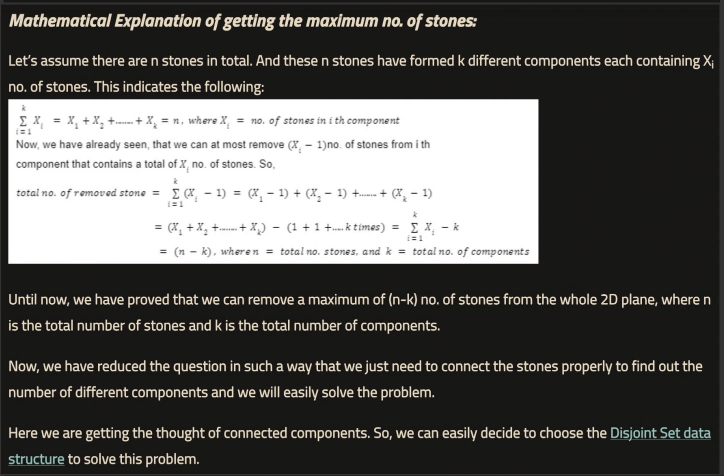
Let's first understand the thought process that we will be using to solve this problem. In this problem, it is clearly stated that a stone can be removed if it shares either the same row or the same column as another stone that has not been removed. So, we can assume that these types of stones, sharing either the same row or column, are connected and belong to the same group. If we take example 2:

S		S	
			S
	S	S	
			S

We can easily spot two different groups in this example. The first group includes the stones [0,0], [0,2], [3,2], and [3,1], and the second one includes [1,3] and [4,3].

If we carefully observe, for each group we can remove all the stones leaving one stone intact. So, we can conclude that at most we can remove (size of the group -1) no. of stones from a group as we need to leave one stone untouched for each group.

Now, if we can think of the stones as nodes, the different groups then seem to be the different components of a graph.



Mathematical Explanation of getting the maximum no. of stones:

Let's assume there are n stones in total. And these n stones have formed k different components each containing X_i no. of stones. This indicates the following:

$$\sum_{i=1}^{k} X_i = X_1 + X_2 + \ldots\ldots + X_k = n\text{, where } X_i = \text{no. of stones in i th component}$$

Now, we have already seen, that we can at most remove $(X_i - 1)$no. of stones from i th component that contains a total of X_i no. of stones. So,

$$\text{total no. of removed stone} = \sum_{i=1}^{k}(X_i - 1) = (X_1 - 1) + (X_2 - 1) + \ldots\ldots + (X_k - 1)$$

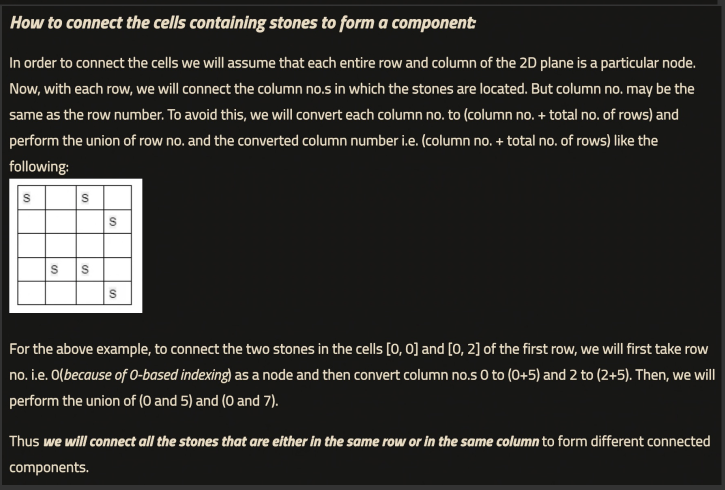
$$= (X_1 + X_2 + \ldots\ldots + X_k) - (1 + 1 + \ldots\text{.. k times}) = \sum_{i=1}^{k} X_i - k$$

$$= (n - k)\text{, where n} = \text{total no. stones, and } k = \text{total no. of components}$$

Until now, we have proved that we can remove a maximum of (n-k) no. of stones from the whole 2D plane, where n is the total number of stones and k is the total number of components.

Now, we have reduced the question in such a way that we just need to connect the stones properly to find out the number of different components and we will easily solve the problem.

Here we are getting the thought of connected components. So, we can easily decide to choose the Disjoint Set data structure to solve this problem.



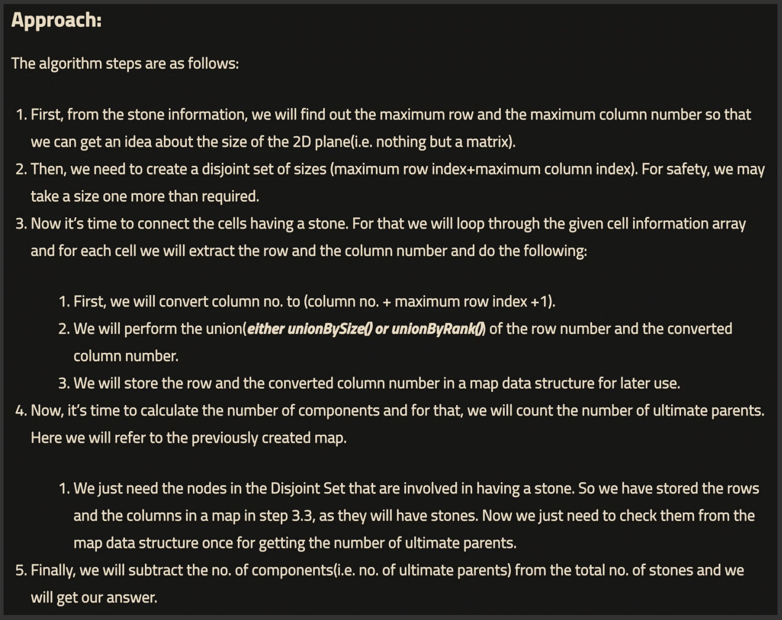
How to connect the cells containing stones to form a component:

In order to connect the cells we will assume that each entire row and column of the 2D plane is a particular node. Now, with each row, we will connect the column no.s in which the stones are located. But column no. may be the same as the row number. To avoid this, we will convert each column no. to (column no. + total no. of rows) and perform the union of row no. and the converted column number i.e. (column no. + total no. of rows) like the following:

S		S	
			S
	S	S	
			S

For the above example, to connect the two stones in the cells [0, 0] and [0, 2] of the first row, we will first take row no. i.e. 0(*because of 0-based indexing*) as a node and then convert column no.s 0 to (0+5) and 2 to (2+5). Then, we will perform the union of (0 and 5) and (0 and 7).

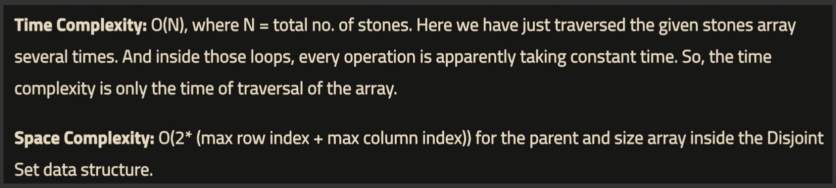
Thus ***we will connect all the stones that are either in the same row or in the same column*** to form different connected components.



Approach:

The algorithm steps are as follows:

1. First, from the stone information, we will find out the maximum row and the maximum column number so that we can get an idea about the size of the 2D plane(i.e. nothing but a matrix).
2. Then, we need to create a disjoint set of sizes (maximum row index+maximum column index). For safety, we may take a size one more than required.
3. Now it's time to connect the cells having a stone. For that we will loop through the given cell information array and for each cell we will extract the row and the column number and do the following:
    1. First, we will convert column no. to (column no. + maximum row index +1).
    2. We will perform the union(***either unionBySize() or unionByRank()***) of the row number and the converted column number.
    3. We will store the row and the converted column number in a map data structure for later use.
4. Now, it's time to calculate the number of components and for that, we will count the number of ultimate parents. Here we will refer to the previously created map.
    1. We just need the nodes in the Disjoint Set that are involved in having a stone. So we have stored the rows and the columns in a map in step 3.3, as they will have stones. Now we just need to check them from the map data structure once for getting the number of ultimate parents.
5. Finally, we will subtract the no. of components(i.e. no. of ultimate parents) from the total no. of stones and we will get our answer.



Time Complexity: O(N), where N = total no. of stones. Here we have just traversed the given stones array several times. And inside those loops, every operation is apparently taking constant time. So, the time complexity is only the time of traversal of the array.

Space Complexity: O(2* (max row index + max column index)) for the parent and size array inside the Disjoint Set data structure.

```
class Solution {
    maxRemove(stones, n) {
        let maxRow = 0;
        let maxCol = 0;
        for (const it of stones) {
            maxRow = Math.max(maxRow, it[0]);
            maxCol = Math.max(maxCol, it[1]);
        }
        const ds = new DisjointSet(maxRow + maxCol + 1);
        const stoneNodes = new Map();
        for (const it of stones) {
            const nodeRow = it[0];
            const nodeCol = it[1] + maxRow + 1;
            ds.unionBySize(nodeRow, nodeCol);
            stoneNodes.set(nodeRow, 1);
            stoneNodes.set(nodeCol, 1);
        }

        let cnt = 0;
        for (const [key, value] of stoneNodes) {
            if (ds.findUPar(key) === key) {
                cnt++;
            }
        }
        return n - cnt;
    }
}

function main() {
    const n = 6;
    const stones = [
        [0, 0],
        [0, 2],
        [1, 3],
        [3, 1],
        [3, 2],
        [4, 3],
    ];

    const obj = new Solution();
    const ans = obj.maxRemove(stones, n);
    console.log("The maximum number of stones we can remove is: " + ans);
}

main();
// The maximum number of stones we can remove is: 4
```

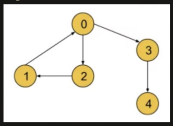
lec 54 :- strongly connected component — Kosaraju's Algo

Problem Statement: Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

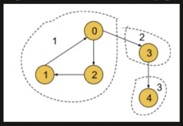
Pre-requisite: DFS algorithm

Example 1:

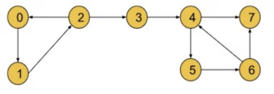
```
Input Format:
```



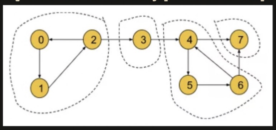
```
Result: 3
Explanation: Three strongly connected components are marked below:
```



```
Input Format:
```



```
Result: 4
Explanation: Four strongly connected components are marked below:
```



In this article, we are going to discuss strongly connected components(SCC) and Kosaraju's algorithm. In an interview, we can expect two types of questions from this topic:

- ***Find the number of strongly connected components of a given graph.***
- ***Print the strongly connected components of a given graph.***

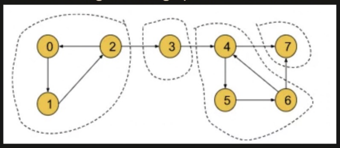
In this article, we are going to discuss the logic part in detail and once the logic part is clear, these two types of questions can be easily solved.

Strongly connected components(SCC) are only valid for directed graphs.

Strongly Connected Component(SCC):

A component is called a Strongly Connected Component(SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u.

In the following directed graph, the SCCs have been marked:



If we take 1st SCC in the above graph, we can observe that each node is reachable from any of the other nodes. For example, if take the pair (0, 1) from the 1st SCC, we can see that 0 is reachable from 1 and 1 is also reachable from 0. Similarly, this is true for all other pairs of nodes in the SCC like (0,2), and (1,2). But if we take node 3 with the component, we can notice that for pair (2,3) 3 is reachable from 3 but 2 is not reachable from 3. So, the first SCC only includes vertices 0, 1, and 2.

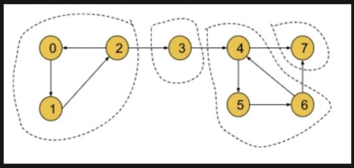
By definition, ***a component containing a single vertex is always a strongly connected component.*** For that vertex 3 in the above graph is itself a strongly connected component.

By applying this logic, we can conclude that the above graph contains 4 strongly connected components like (0,1,2), (3), (4,5,6), and (7).

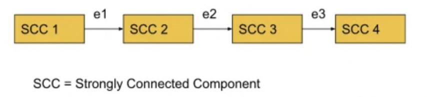
Kosaraju's Algorithm:

To find the strongly connected components of a given directed graph, we are going to use Kosaraju's Algorithm.

Before understanding the algorithm, we are going to discuss the thought process behind it. If we start DFS from node 0 for the following graph, we will end up visiting all the nodes. So, it is impossible to differentiate between different SCCs.

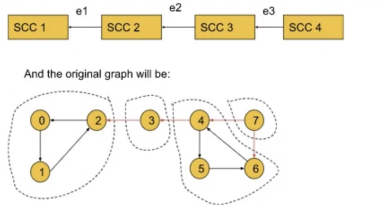


Now, we need to think in a different way. We can convert the above graph into the following illustration:



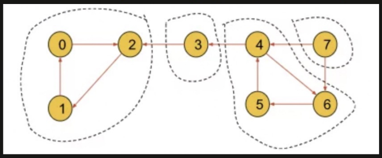
By definition, within each SCC, every node is reachable. So, if we start DFS from a node of SCC1 we can visit all the nodes in SCC1 and via edge e1 we can reach SCC2. Similarly, we can travel from SCC2 to SCC3 via e2 and SCC3 to SCC4 via e3. Thus all the nodes of the graph become reachable.

But if we reverse the edges e1, e2, and e3, the graph will look like the following:



Now in this graph, if we start DFS from node 0 it will visit only the nodes of SCC1. Similarly, if we start from node 3 it will visit only the nodes of SCC2. Thus, by reversing the SCC-connecting edges, the adjacent SCCs become unreachable. Now, the DFS will work in such a way, that in one DFS call we can only visit the nodes of a particular SCC. So, ***the number of DFS calls will represent the number of SCCs.***

Until now, we have successfully found out the process of getting the number of SCCs. But here, comes a new problem i.e. if we do not know the SCCs, how the edges will be reversed? To solve this problem, we will simply try to reverse all the edges of the graph like the following:



If we carefully observe, the nodes within an SCC are reachable from each one to everyone even if we reverse the edges of the SCC. So, the SCCs will have no effect on reversing the edges. Thus we can fulfill our intention of reversing the SCC-connecting edge without affecting the SCCs.

Now, the question might be like, if node 0 is located in SCC4 and we start DFS from node 0, again we will visit all the SCCs at once even after reversing the edges. This is where ***the starting time and the finishing time*** concept will come in.

Now, we have a clear intuition about reversing edges before we move on to the starting and the finishing time concept in the algorithm part.

The algorithm steps are as follows:

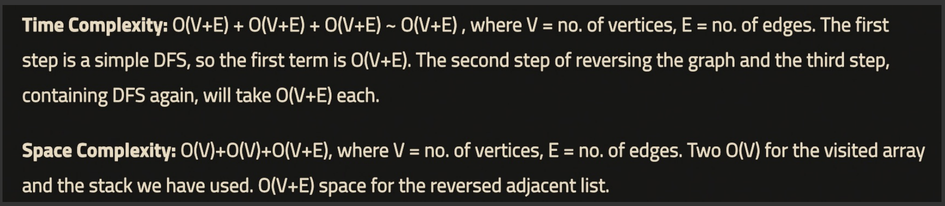
1. ***Sort all the nodes according to their finishing time:***
   To sort all the nodes according to their finishing time, we will start DFS from node 0 and while backtracking in the DFS call we will store the nodes in a stack data structure. The nodes in the last SCC will finish first and will be stored in the last of the stack. After the DFS gets completed for all the nodes, the stack will be storing all the nodes in the sorted order of their finishing time.
2. ***Reverse all the edges of the entire graph:***
   Now, we will create another adjacency list and store the information of the graph in a reversed manner.
3. ***Perform the DFS and count the no. of different DFS calls to get the no. of SCC:***
   Now, we will start DFS from the node which is on the top of the stack and continue until the stack becomes empty. For each individual DFS call, we will increment the counter variable by 1. We will get the number of SCCs by just counting the number of individual DFS calls as in each individual DFS call, all the nodes of a particular SCC get visited.
4. Finally, we will get the number of SCCs in the counter variable. If we want to store the SCCs as well, we need to store the nodes in some array during each individual DFS call in step 3.

Note:

- The first step is to know, from which node we should start the DFS call.
- The second step is to make adjacent SCCs unreachable and to limit the DFS traversal in such a way, that in each DFS call, all the nodes of a particular SCC get visited.
- The third step is to get the numbers of the SCCs. In this step, we can also store the nodes of each SCC if we want to do so.

Note: *The sorting of the nodes according to their finishing time is very important. By performing this step, we will get to know where we should start our DFS calls. The top-most element of the stack will finish last and it will surely belong to the SCC1. So, the sorting step is important for the algorithm.*

```
class Solution {
    dfs(node, vis, adj, st) {
        vis[node] = 1;
        for (const it of adj[node]) {
            if (!vis[it]) {
                this.dfs(it, vis, adj, st);
            }
        }
        st.push(node);
    }

    dfs3(node, vis, adjT) {
        vis[node] = 1;
        for (const it of adjT[node]) {
            if (!vis[it]) {
                this.dfs3(it, vis, adjT);
            }
        }
    }

    kosaraju(V, adj) {
        const vis = new Array(V).fill(0);
        const st = [];
        for (let i = 0; i < V; i++) {
            if (!vis[i]) {
                this.dfs(i, vis, adj, st);
            }
        }

        const adjT = new Array(V).fill(0).map(() => []);
        for (let i = 0; i < V; i++) {
            vis[i] = 0;
            for (const it of adj[i]) {
                adjT[it].push(i);
            }
        }

        let scc = 0;
        while (st.length > 0) {
            const node = st.pop();
            if (!vis[node]) {
                scc++;
                this.dfs3(node, vis, adjT);
            }
        }
        return scc;
    }
}

function main() {
    const n = 5;
    const edges = [[1, 0],[0, 2],[2, 1],[0, 3],[3, 4]];
    const adj = new Array(n).fill(0).map(() => []);
    for (let i = 0; i < n; i++) {
        adj[edges[i][0]].push(edges[i][1]);
    }

    const obj = new Solution();
    const ans = obj.kosaraju(n, adj);
    console.log("The number of strongly connected components is: " + ans);
}

main();
// The number of strongly connected components is: 3
```

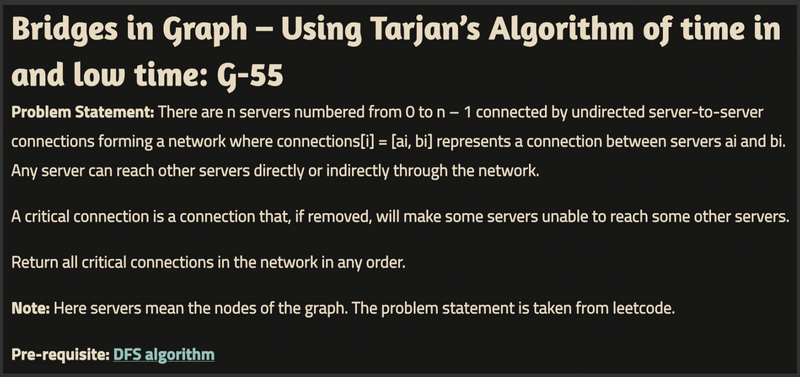


Time Complexity: O(V+E) + O(V+E) + O(V+E) ~ O(V+E) , where V = no. of vertices, E = no. of edges. The first step is a simple DFS, so the first term is O(V+E). The second step of reversing the graph and the third step, containing DFS again, will take O(V+E) each.

Space Complexity: O(V)+O(V)+O(V+E), where V = no. of vertices, E = no. of edges. Two O(V) for the visited array and the stack we have used. O(V+E) space for the reversed adjacent list.

---

lec 55 bridges in graph → using tarjan's algorithm of time in and low time



Bridges in Graph – Using Tarjan's Algorithm of time in and low time: G-55

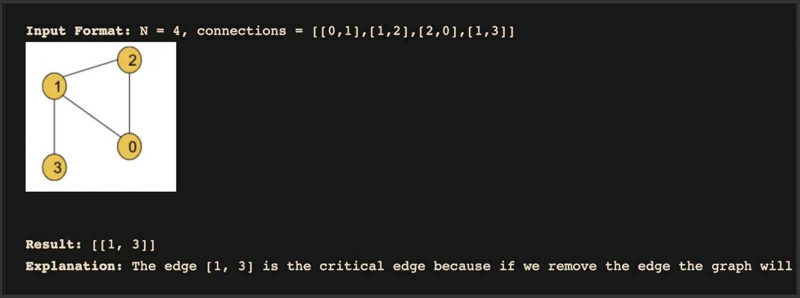
Problem Statement: There are n servers numbered from 0 to n – 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.

A critical connection is a connection that, if removed, will make some servers unable to reach some other servers.

Return all critical connections in the network in any order.

Note: Here servers mean the nodes of the graph. The problem statement is taken from leetcode.

Pre-requisite: DFS algorithm



Input Format: N = 4, connections = [[0,1],[1,2],[2,0],[1,3]]

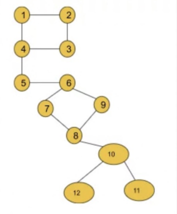
Result: [[1, 3]]

Explanation: The edge [1, 3] is the critical edge because if we remove the edge the graph will

Bridge:

Any edge in a component of a graph is called a bridge when the component is divided into 2 or more components if we remove that particular edge.

Example:



If in this graph, we remove the edge (5,6), the component gets divided into 2 components. So, it is a bridge. But if we remove the edge (2,3) the component remains connected. So, this is not a bridge. In this graph, we have a total of 3 bridges i.e. (4,5), (5,6), and (10, 8).

In order to find all the bridges of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let's discuss the algorithm in detail. Before that, we will discuss two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

- **Time of insertion:** Dring the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 1 it will visit node 1 first then node 2, node 3, node 4, and so on. So, the time of insertion for node 1 will be 1, node 2 will be 2, node 3 will be 3 and it will continue like this. ***To store the time of insertion for each node, we will use a time array.***
- **Lowest time of insertion:** In this case, the current node refers to all its adjacent nodes ***except the parent*** and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another 'low' array.

The logical modification of the DFS algorithm is discussed below.

After the DFS for any adjacent node gets completed, we will just check if the edge, whose starting point is the current node and ending point is that adjacent node, is a bridge. For that, we will just check if any other path from the current node to the adjacent node exists if we remove that particular edge. If any other alternative path exists, this edge is not a bridge. Otherwise, it can be considered a valid bridge.

1. First, we need to create the adjacency list for the given graph from the edge information(***If not already given***). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node.
2. Then we will start DFS from node 0(assuming the graph contains a single component otherwise, we will call DFS for every component) with parent -1.

    1. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. The timer may be initialized to 0 or 1.
    2. Now, it's time to visit the adjacent nodes.

        1. ***If the adjacent node is the parent itself***, we will just continue to the next node.
        2. ***If the adjacent node is not visited***, we will call DFS for the adjacent node with the current node as the parent.
        After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
        Now, we will check if the lowest time of insertion of the adjacent node is greater than the time of insertion of the current node.
        If it is, then we will store the adjacent node and the current node in our answer array as they are representing the bridge.
        3. ***If the adjacent node is already visited***, we will just compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
3. Finally, our answer array will store all the bridges.

Note: *We are not considering the parent's insertion time during calculating the lowest insertion time as we want to check if any other path from the node to the parent exists excluding the edge we intend to remove.*

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

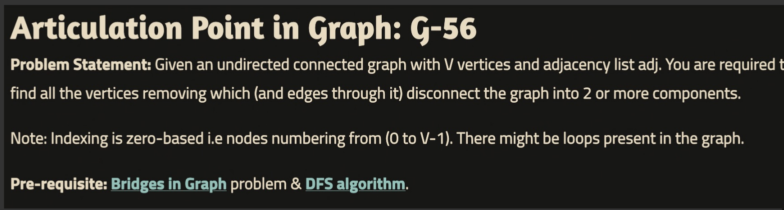
```
class Solution {
    constructor() {
        this.timer = 1;
    }

    dfs(node, parent, vis, adj, tin, low, bridges) {
        vis[node] = 1;
        tin[node] = low[node] = this.timer;
        this.timer++;

        for (let it of adj[node]) {
            if (it === parent) continue;
            if (vis[it] === 0) {
                this.dfs(it, node, vis, adj, tin, low, bridges);
                low[node] = Math.min(low[it], low[node]);
                if (low[it] > tin[node]) {
                    bridges.push([it, node]);
                }
            } else {
                low[node] = Math.min(low[node], low[it]);
            }
        }
    }

    criticalConnections(n, connections) {
        let adj = Array.from({ length: n }, () => []);
        for (let it of connections) {
            let u = it[0],
                v = it[1];
            adj[u].push(v);
            adj[v].push(u);
        }
        let vis = Array(n).fill(0);
        let tin = Array(n).fill(0);
        let low = Array(n).fill(0);
        let bridges = [];
        this.dfs(0, -1, vis, adj, tin, low, bridges);
        return bridges;
    }
}

let n = 4;
let connections = [[0, 1],[1, 2],[2, 0],[1, 3]];

let obj = new Solution();
let bridges = obj.criticalConnections(n, connections);
for (let it of bridges) {
    console.log(`[${it[0]}, ${it[1]}]`); // [3, 1]
}
console.log();
```

Output: [3, 1] (In example 1, [1, 3] and [3, 1] both are accepted.)

Time Complexity: O(V+2E), where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: O(V+2E) + O(3V), where V = no. of vertices, E = no. of edges. O(V+2E) to store the graph in an adjacency list and O(3V) for the three arrays i.e. tin, low, and vis, each of size V.

lec 56 :→ Articulation Point in Graph

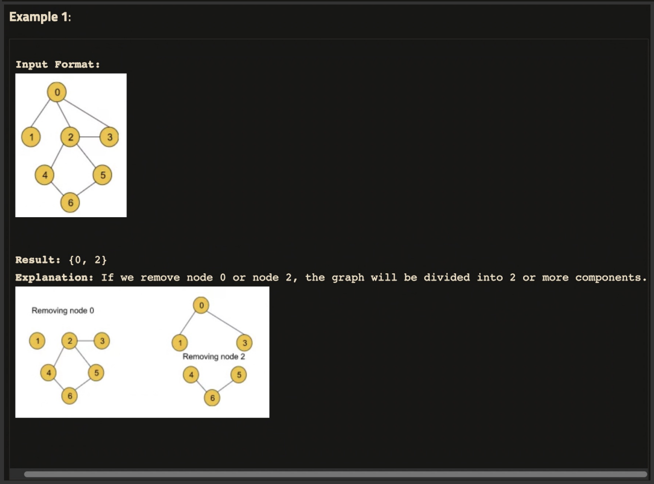


Articulation Point in Graph: G-56

Problem Statement: Given an undirected connected graph with V vertices and adjacency list adj. You are required to find all the vertices removing which (and edges through it) disconnect the graph into 2 or more components.

Note: Indexing is zero-based i.e nodes numbering from (0 to V-1). There might be loops present in the graph.

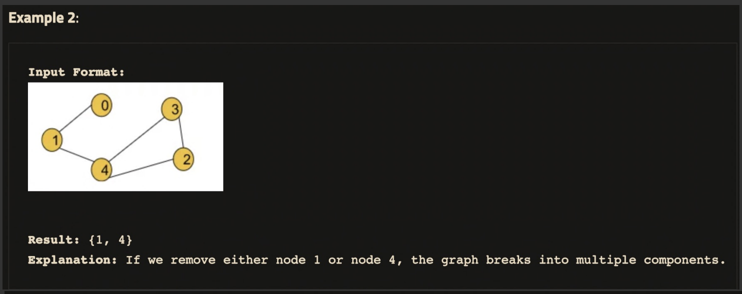
Pre-requisite: Bridges in Graph problem & DFS algorithm.



Example 1:

Input Format:

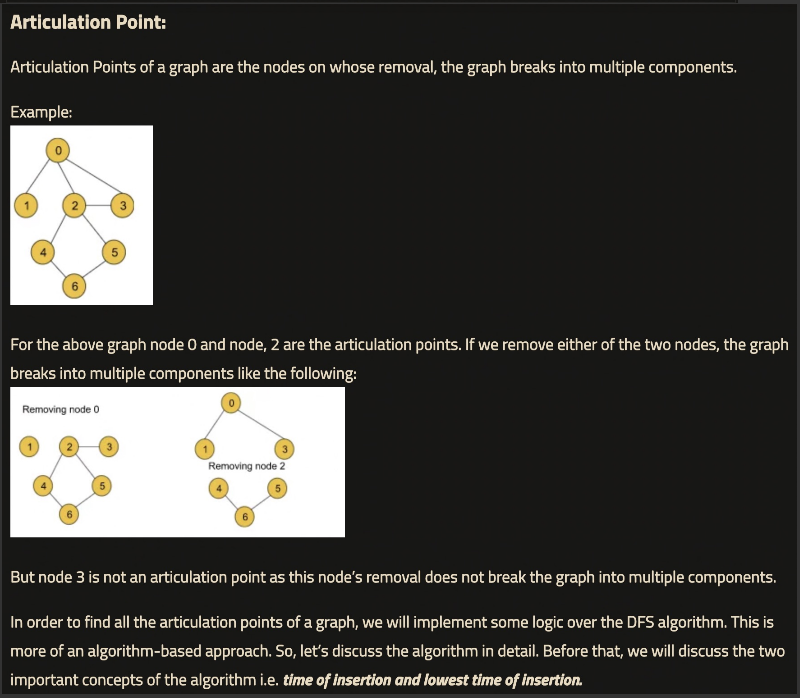
Result: {0, 2}
Explanation: If we remove node 0 or node 2, the graph will be divided into 2 or more components.



Example 2:

Input Format:

Result: {1, 4}
Explanation: If we remove either node 1 or node 4, the graph breaks into multiple components.



Articulation Point:

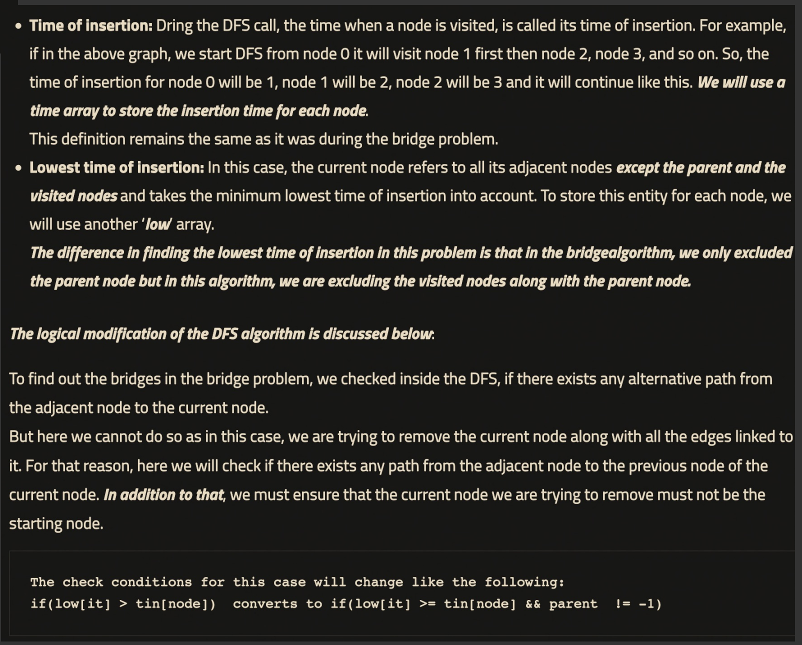
Articulation Points of a graph are the nodes on whose removal, the graph breaks into multiple components.

Example:

For the above graph node 0 and node, 2 are the articulation points. If we remove either of the two nodes, the graph breaks into multiple components like the following:

But node 3 is not an articulation point as this node's removal does not break the graph into multiple components.

In order to find all the articulation points of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let's discuss the algorithm in detail. Before that, we will discuss the two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

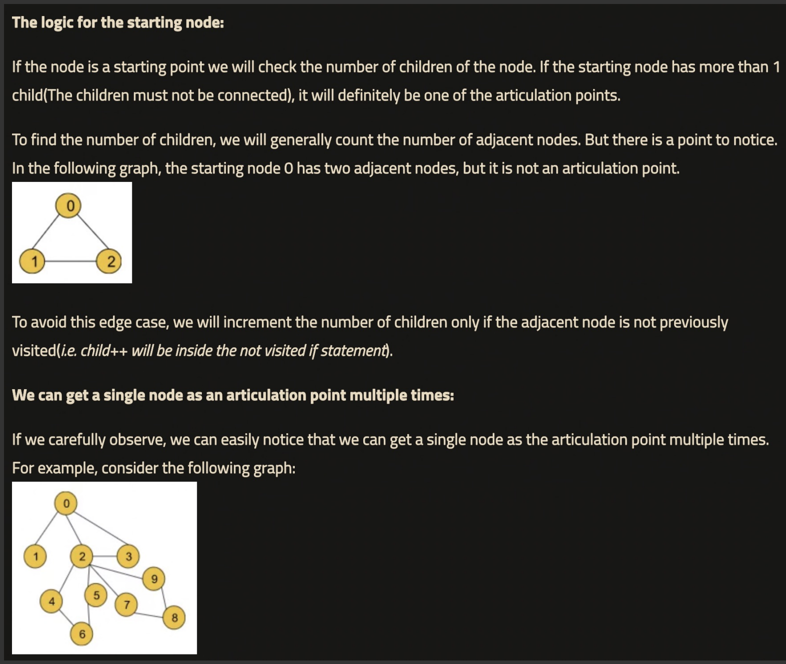


- **Time of insertion:** Dring the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 0 it will visit node 1 first then node 2, node 3, and so on. So, the time of insertion for node 0 will be 1, node 1 will be 2, node 2 will be 3 and it will continue like this. ***We will use a time array to store the insertion time for each node.***
  This definition remains the same as it was during the bridge problem.
- **Lowest time of insertion:** In this case, the current node refers to all its adjacent nodes ***except the parent and the visited nodes*** and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another '***low***' array.
  The difference in finding the lowest time of insertion in this problem is that in the bridgealgorithm, we only excluded the parent node but in this algorithm, we are excluding the visited nodes along with the parent node.

The logical modification of the DFS algorithm is discussed below:

To find out the bridges in the bridge problem, we checked inside the DFS, if there exists any alternative path from the adjacent node to the current node.
But here we cannot do so as in this case, we are trying to remove the current node along with all the edges linked to it. For that reason, here we will check if there exists any path from the adjacent node to the previous node of the current node. ***In addition to that***, we must ensure that the current node we are trying to remove must not be the starting node.

```
The check conditions for this case will change like the following:
if(low[it] > tin[node])  converts to if(low[it] >= tin[node] && parent  != -1)
```



The logic for the starting node:

If the node is a starting point we will check the number of children of the node. If the starting node has more than 1 child(The children must not be connected), it will definitely be one of the articulation points.

To find the number of children, we will generally count the number of adjacent nodes. But there is a point to notice. In the following graph, the starting node 0 has two adjacent nodes, but it is not an articulation point.

To avoid this edge case, we will increment the number of children only if the adjacent node is not previously visited(*i.e. child++ will be inside the not visited if statement*).

We can get a single node as an articulation point multiple times:

If we carefully observe, we can easily notice that we can get a single node as the articulation point multiple times. For example, consider the following graph:

While checking for node 2, we will get the node as the articulation point once for the first component that contains nodes 4, 5, and 6 and we will again get the same node 2 for the second component that includes the nodes 7, 8, and 9.

To avoid the storing of duplicate nodes, we will store the nodes in a hash array(i.e. mark array used in the code) instead of directly inserting them in a simple array.

Approach:

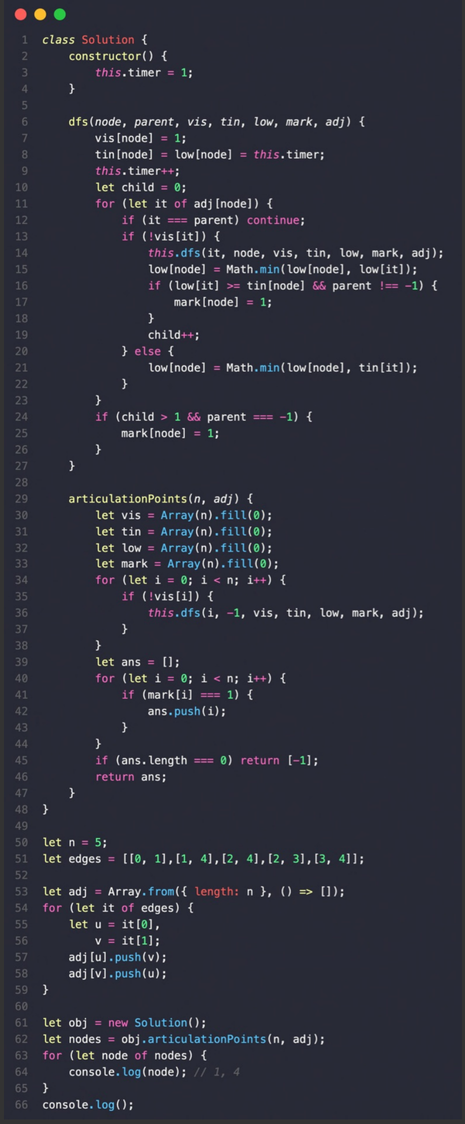
The algorithm steps are as follows:

1. First, we need to create the adjacency list for the given graph from the edge information(***If not already given***). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node. The timer may be initialized to 0 or 1 accordingly.
2. Then we will perform DFS for each component. For each component, the starting node will carry -1 as its parent.
    1. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. We will declare a child variable to implement the logic for starting node.
    2. Now, it's time to visit the adjacent nodes.
        1. ***If the adjacent node is the parent itself***, we will just continue to the next node.
        2. ***If the adjacent node is not visited***, we will call DFS for the adjacent node with the current node as the parent.
           After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum.
           Now, we will check if the lowest time of insertion of the adjacent node is greater or equal to the

Now, we will check if the lowest time of insertion of the adjacent node is greater or equal to the

           time of insertion of the current node and also ensure that the current node is not the starting node(checking parent not equal -1).
           If the condition matches, then we will mark the current node in our hash array as one of our answers as it is one of the articulation points of the graph.
           Then we will increment the child variable by 1.
        3. ***If the adjacent node is visited***, we will just compare the lowest time of insertion of the current node and the time of insertion of the adjacent node and take the minimum.
    3. Finally, we will check if the child value is greater than 1 and if the current node is the starting node. If it is then we will keep the starting node marked in our hash array as the starting node is also an articulation point in this case.
3. Finally, our answer array will store all the bridges.

Note: *We are not considering the parent and the visited nodes during calculating the lowest insertion time as they may be the articulation points of the graph which means they may be the nodes we intend to remove.*



```
class Solution {
    constructor() {
        this.timer = 1;
    }

    dfs(node, parent, vis, tin, low, mark, adj) {
        vis[node] = 1;
        tin[node] = low[node] = this.timer;
        this.timer++;
        let child = 0;
        for (let it of adj[node]) {
            if (it === parent) continue;
            if (!vis[it]) {
                this.dfs(it, node, vis, tin, low, mark, adj);
                low[node] = Math.min(low[node], low[it]);
                if (low[it] >= tin[node] && parent !== -1) {
                    mark[node] = 1;
                }
                child++;
            } else {
                low[node] = Math.min(low[node], tin[it]);
            }
        }
        if (child > 1 && parent === -1) {
            mark[node] = 1;
        }
    }

    articulationPoints(n, adj) {
        let vis = Array(n).fill(0);
        let tin = Array(n).fill(0);
        let low = Array(n).fill(0);
        let mark = Array(n).fill(0);
        for (let i = 0; i < n; i++) {
            if (!vis[i]) {
                this.dfs(i, -1, vis, tin, low, mark, adj);
            }
        }
        let ans = [];
        for (let i = 0; i < n; i++) {
            if (mark[i] === 1) {
                ans.push(i);
            }
        }
        if (ans.length === 0) return [-1];
        return ans;
    }
}

let n = 5;
let edges = [[0, 1],[1, 4],[2, 4],[2, 3],[3, 4]];

let adj = Array.from({ length: n }, () => []);
for (let it of edges) {
    let u = it[0],
        v = it[1];
    adj[u].push(v);
    adj[v].push(u);
}

let obj = new Solution();
let nodes = obj.articulationPoints(n, adj);
for (let node of nodes) {
    console.log(node); // 1, 4
}
console.log();
```

Output: 1 4 (Example 2)

Time Complexity: O(V+2E), where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: O(3V), where V = no. of vertices. O(3V) is for the three arrays i.e. tin, low, and vis, each of size V.