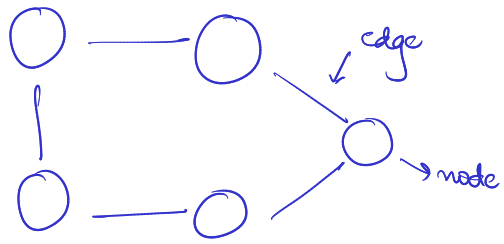
#Graphs

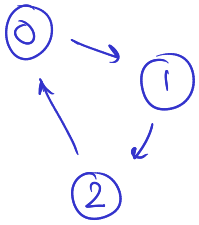


- → Directed
- → Undirected
- → Weighted
- → Non-weighted
- → Cyclic
- → Acyclic
- → Disconnected

Representation of Graph:-

- → Adjacency matrix
- → Adjacency list

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



0 → 1, 4
1 → 0, 2, 3
2 → 1, 3
3 → 4, 1, 2
4 → 3, 0

Adj. List

↓

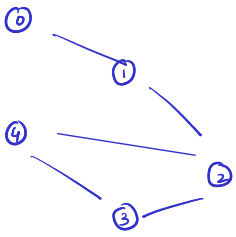
Implementation

↓

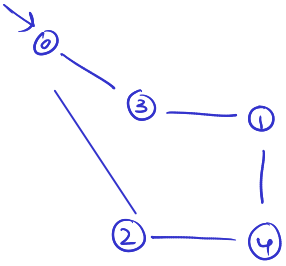
map<int, list<int>>

```cpp
#include <iostream>
#include <map>
#include <list>
using namespace std;

class graph{
public:
    unordered_map<int, list<int>> adj;

    void addEdge(int u, int v, bool direction) {
        // direction = 0 → undirected
        // direction = 1 → directed

        // create an edge from u to v
        adj[u].push_back(v);
        if(direction==0) adj[v].push_back(u);
    }
    void printAdjList() {
        for(auto i:adj) {
            cout<<i.first<<"→ ";
            for(auto j:i.second) {
                cout<<j<<", ";
            }
            cout<<endl;
        }
    }
};

int main() {
    int n;
    cout<<"Enter the number of nodes"<<endl;
    cin>>n;
    int m;
    cout<<"Enter the number of edges"<<endl;
    cin>>m;
    graph g;
    for(int i = 0; i < m; i++) {
        int u, v;
        cin>>u>>v;
        //creating undirected graph
        g.addEdge(u,v,0);
    }
    g.printAdjList();
    return 0;
}
```

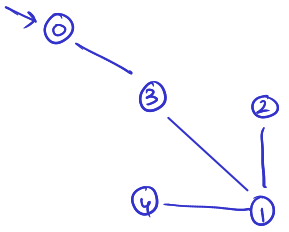
BFS



Print → 0, 1, 2, 3, 4



0, 2, 3, 1, 4



visited node

unordered_map
<node, bool>

0, 3, 1, 2, 4

Ans: ④ 0 3 1 2 4

adj list

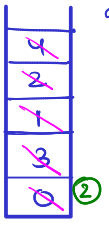
0 → 3
1 → 2, 3, 4
2 → 1
3 → 0, 1
4 → 1

visited

0 → ~~F~~ T
1 → ~~F~~ T
2 → ~~F~~ T
3 → ~~F~~ T ③
4 → ~~F~~ T

⑤

→ Now, after zero is added, push all the adjacent / neighbour nodes to queue.



queue

① frontNode = 0

frontNode = 3

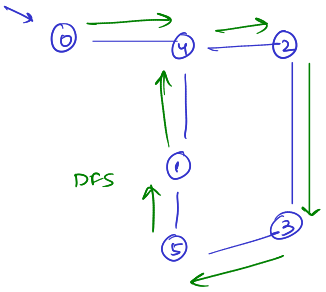
frontNode = 1

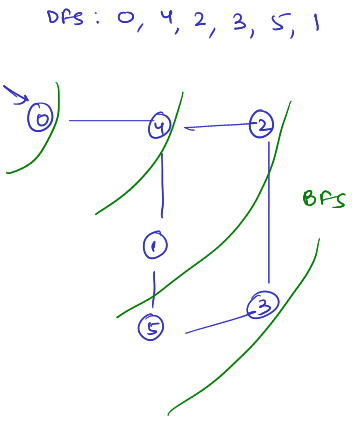
frontNode = 2

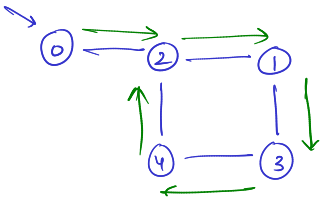
frontNode = 4

```cpp
#include <unordered_map>
#include<list>
#include<set>
#include<queue>

void prepareAdjList(unordered_map<int, list<int>> &adjList, vector<pair<int, int>> edges) {
    for(int i = 0; i < edges.size(); i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        adjList.push_back(v);
        adjList.push_back(u);
    }
}

void bfs(unordered_map<int, list<int>> adjList, unordered_map<int, bool> visited, vector<int> &ans, int node) {
    queue<int> q;
    q.push(node);
    visited[node] = 1;
    while(!q.empty()) {
        int frontNode = q.front();
        q.pop();

        //store frontNode in ans;
        ans.push_back(frontNode);

        // traverse all neighbours of frontNode
        for(auto i: adjList[frontNode]) {
            if(!visited[i]) {
                q.push(i);
                visited[i] = 1;
            }
        }
    }
}

vector<int> BFS(int vertex, vector<pair<int, int>> edges) {
    unordered_map<int, list<int>> adjList;
    vector<int> ans;
    unordered_map<int, bool> visited;

    prepareAdjList(adjList, edges);

    // traverse all component of unconnected graph
    for(int i = 0; i < vertex; i++) {
        if(!visited[i]) bfs(adjList, visited, ans, i);
    }
    return ans;
}
```

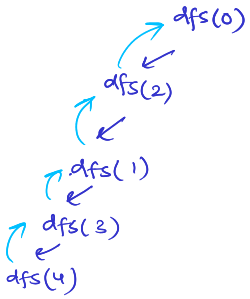
Dfs







0 2 1 3 4

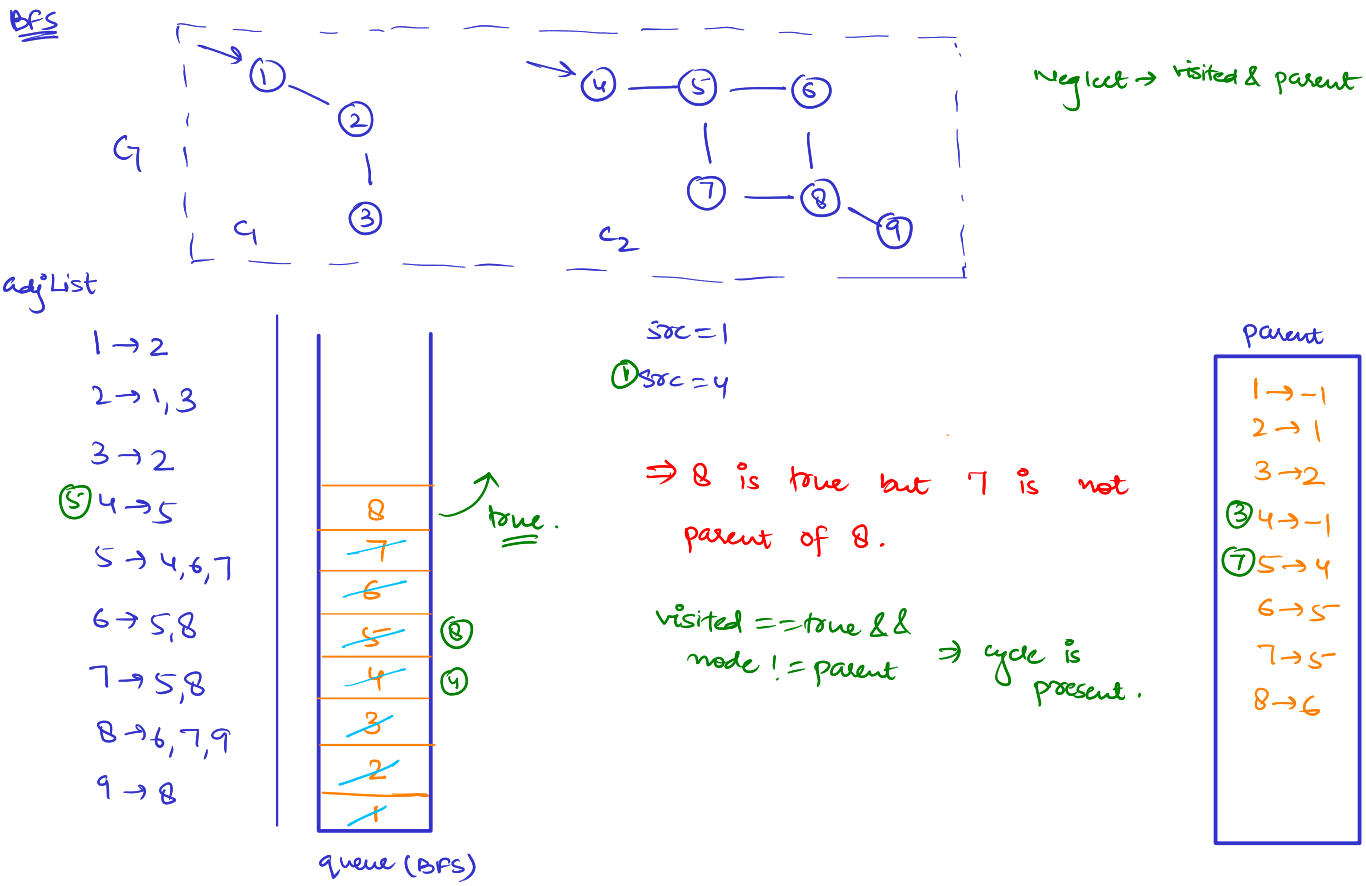



```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(int start, vector<int> adj[], vector<int> &innerAns, int vis[]) {
  vis[start] = 1;
  innerAns.push_back(start);
  for (auto it : adj[start]) {
    if (!vis[it]) {
      dfs(it, adj, innerAns, vis);
    }
  }
}

vector<vector<int>> depthFirstSearch(int V, int E, vector<vector<int>> &edges) {
  // Creating adjacency list
  vector<int> adj[V];
  for (int it = 0; it < edges.size(); it++) {
    int u = edges[it][0];
    int v = edges[it][1];
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  // Array for visited mark initially all are 0
  int vis[V] = {0};
  // Answer vetor to store the final ans
  vector<vector<int>> ans;

  // loop through out the vertices and look for unvisited one and then go dipper
  // and push ans for that
  for (int it = 0; it < V; it++) {
    if (!vis[it]) {
      vector<int> innerAns;
      // deeply visit the adjacency one by DFS
      dfs(it, adj, innerAns, vis);
      ans.push_back(innerAns);
    }
  }
  return ans;
}
```

→ If adjacency list is already given.

```
class Solution {
  public:
    void dfs(int i, vector<int> adj[], vector<int> &ans, int vis[]) {
        vis[i] = 1;
        ans.push_back(i);
        for(auto it:adj[i]) {
            if(!vis[it]) dfs(it, adj, ans, vis);
        }
    }

    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vector<int> ans;

        for(int i = 0; i < V; i++) {
            if(!vis[i]) {
                dfs(i, adj, ans, vis);
            }
        }
        return ans;
    }
};
```

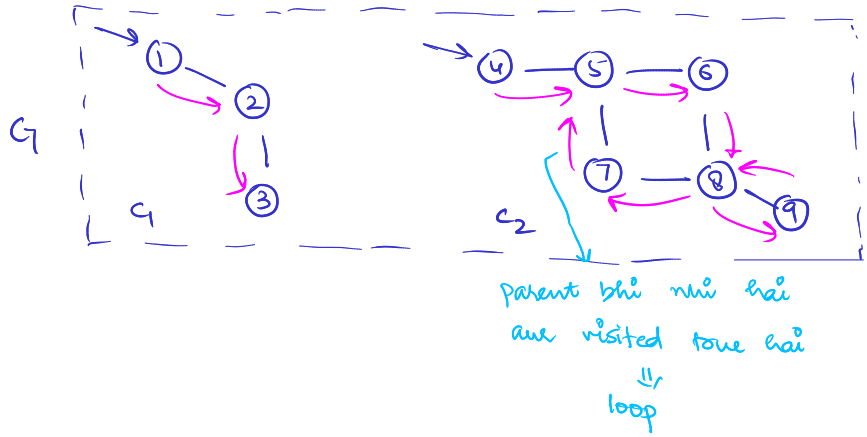
Cycle Detection in Undirected Graph




```cpp
#include <unordered_map>
#include <list>
#include <vector>
#include <queue>
#include <string>
using namespace std;

bool isCyclicBFS(int node, unordered_map<int, bool> &visited,  unordered_map<int, list<int>> &adj) {
  unordered_map<int, int> parent;

  parent[node] = -1;
  visited[node] = true;

  queue<int> q;
  q.push(node);

  while(!q.empty()) {
    int frontNode = q.front();
    q.pop();

    for(auto neighbour: adj[frontNode]) {
      if(visited[neighbour] == true && neighbour ≠ parent[frontNode]) {
        return true;
      }
      else if(!visited[neighbour]) {
        q.push(neighbour);
        parent[neighbour] = frontNode;
        visited[neighbour] = 1;
      }
    }
  }
  return false;
}
```

```cpp
string cycleDetection(vector<vector<int>>& edges, int n, int m)
{
    // Step 1: Create adjacency list
    unordered_map<int, list<int>> adj;
    for(int i = 0; i < m; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
        adj[edges[i][1]].push_back(edges[i][0]); // Add the reverse direction edge
    }

    // Step 2: Initialize visited map
    unordered_map<int, bool> visited;

    // Step 3: Search through the vertices, if not visited then go deeper
    for(int i = 1; i ⩽ n; i++) {
        if(!visited[i]) {
            bool ans = isCyclicBFS(i, visited, adj);
            if(ans) return "Yes";
        }
    }
    return "No";
}
```

DFS

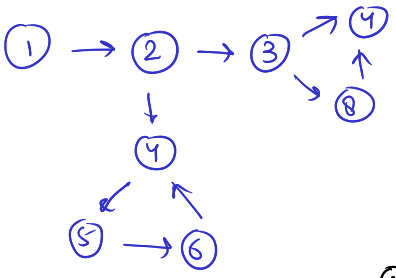


→ Here also we will use parent & visited.

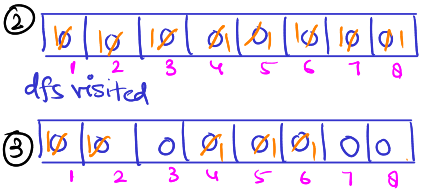
```cpp
#include <unordered_map>
#include <list>
#include <vector>
#include <string>
using namespace std;

bool isCyclicDFS(int node, int parent, unordered_map<int, bool> &visited, unordered_map<int, list<int>> &adj) {
    visited[node] = true;

    for(auto neighbour: adj[node]) {
        if(!visited[neighbour]) {
            bool cycleDetected = isCyclicDFS(neighbour, node, visited, adj);
            if(cycleDetected) return true;
        } else if(neighbour != parent) return true;
    }
    return false;
}

string cycleDetection(vector<vector<int>>& edges, int n, int m)
{
    // Step 1: Create adjacency list
    unordered_map<int, list<int>> adj;
    for(int i = 0; i < m; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
        adj[edges[i][1]].push_back(edges[i][0]); // Add the reverse direction edge
    }

    // Step 2: Initialize visited map
    unordered_map<int, bool> visited;

    // Step 3: Search through the vertices, if not visited then go deeper
    for(int i = 1; i <= n; i++) {
        if(!visited[i]) {
            bool ans = isCyclicDFS(i, -1, visited, adj);
            if(ans) return "Yes";
        }
    }
    return "No";
}
```

Cycle Detection in directed graph

DFS

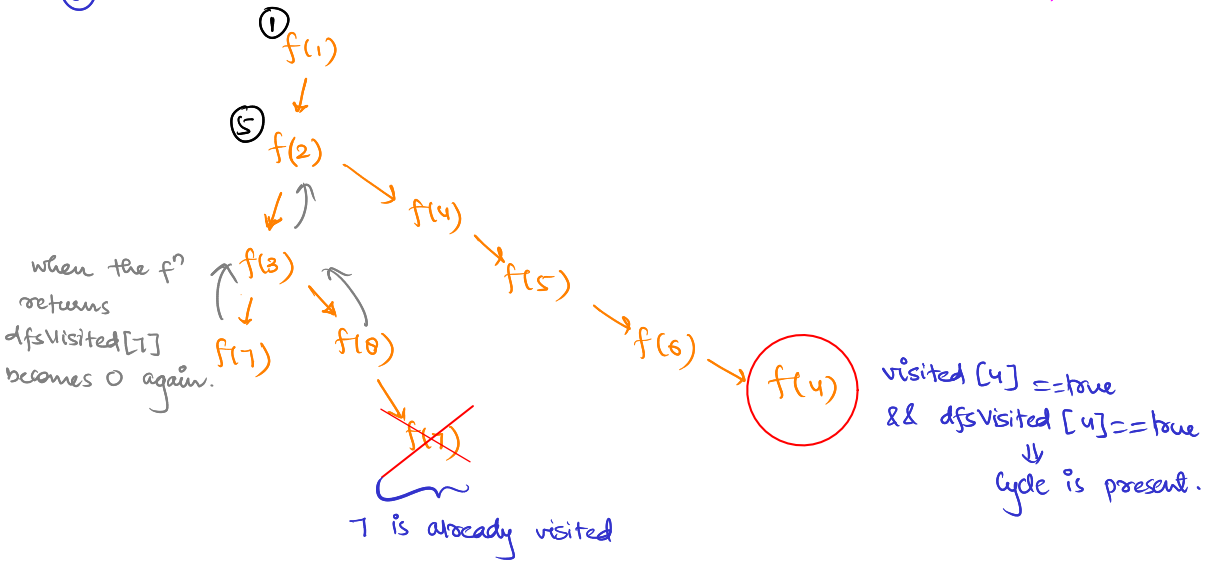


visited (yeh toh purana wala hi hai)



Adj. List

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




```cpp
#include <list>
#include <unordered_map>

class Solution {
public:
    bool cyclicDFS(int i, unordered_map<int, bool> &visited, unordered_map<int, bool> &dfsVisited, unordered_map<int, list<int>> adjList) {
        visited[i] = true;
        dfsVisited[i] = true;

        for(auto elem: adjList[i]) {
            if(!visited[elem]) {
                bool isFoundCyclic = cyclicDFS(elem, visited, dfsVisited, adjList);
                if(isFoundCyclic) return true;
            } else if(dfsVisited[elem]) return true;
        }

        dfsVisited[i] = false;
        return false;

    }
    bool isCyclic(int V, vector<int> adj[]) {

        // Step 1: Create Adjacancy list
        unordered_map<int, list<int>> adjList;

        for(int i = 0; i < V; i++) {
            for(int j = 0; j < adj[i].size(); j++) {
                adjList[i].push_back(adj[i][j]);
            }
        }

        // Step 2: Check if visited, if not then go deeper
        unordered_map<int, bool> visited;
        unordered_map<int, bool> dfsVisited;

        for(int i = 0 ; i < V; i++) {
            if(!visited[i]) {
                bool isFoundCyclic = cyclicDFS(i, visited, dfsVisited, adjList);
                if(isFoundCyclic) return true;
            }
        }

        return false;
    }
};
```

TLE

Find the no. of province

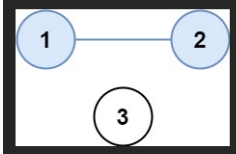
There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

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

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

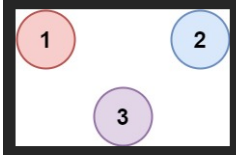
Return *the total number of* ***provinces***.

Example 1:




```
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2
```

Example 2:




```
Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]
Output: 3
```

Constraints:

- `1 <= n <= 200`
- `n == isConnected.length`
- `n == isConnected[i].length`
- `isConnected[i][j]` is `1` or `0`.

→ Approach 1 :-

↳ We just need to calculate the no. of disconnected graphs in the given domain.

```
class Solution {
public:
    void dfs(int i, unordered_map<int, bool> &visited, const vector<vector<int>> &isConnected) {
        visited[i] = true;
        for(int j = 0; j < isConnected[i].size(); j++) {
            if(isConnected[i][j] && !visited[j]) {
                dfs(j, visited, isConnected);
            }
        }
    }
    int findCircleNum(vector<vector<int>>& isConnected) {
        unordered_map<int, bool> visited;
        int V = isConnected.size();
        int provinces = 0;
        for(int i = 0; i < V; i++) {
            if(!visited[i]) {
                provinces++;
                dfs(i, visited, isConnected);
            }

        }
        return provinces;
    }
};
```

Time Complexity : $O(N) + O(V+2E)$

N is the outer loop & inner loop runs in total a single DFS over entire graph, & we know DFS takes a time of $O(V+2E)$.

Space Complexity: $O(N) + O(N)$

Space for recursion space stack & visited array.

4 Rotten Oranges

→ Approach 1 :-

Problem statement Send feedback

You have been given a grid containing some oranges. Each cell of this grid has one of the three integers values:
Value 0 - representing an empty cell.
Value 1 - representing a fresh orange.
Value 2 - representing a rotten orange.

Every second, any fresh orange that is adjacent(4-directionally) to a rotten orange becomes rotten.
Your task is to find out the minimum time after which no cell has a fresh orange.
If it's impossible to rot all the fresh oranges then print -1.
Note:

1. The grid has 0-based indexing.
2. A rotten orange can affect the adjacent oranges 4 directionally i.e. Up, Down, Left, Right.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= N <= 500
1 <= M <= 500
0 <= grid[i][j] <= 2

Time Limit: 1 sec

Sample Input 1:

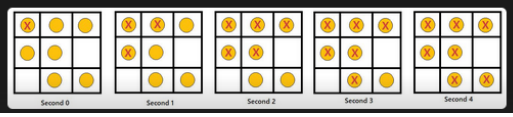
3 3
2 1 1
1 1 0
0 1 1

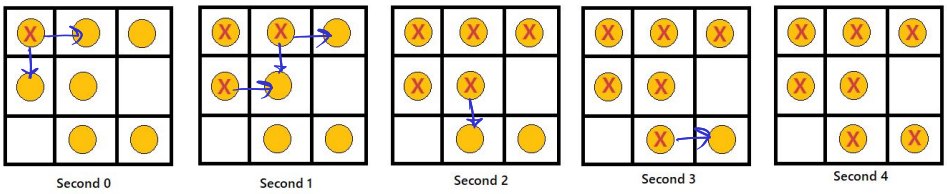
Sample Output 1:

4

Explanation of Sample Input 1:

Minimum 4 seconds are required to rot all the oranges in the grid as shown below.





If observed carefully you will see that, the cell with a rotten orange spreads its effect to the next one in just the next second ⇒ BFS.

```
#include <queue>

int minTimeToRot(vector<vector<int>> &grid, int m, int n) {
  if (grid.empty())
    return 0;
  int days = 0, tot = 0, cnt = 0;
  queue<pair<int, int>> rotten;
  for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
      if (grid[i][j] != 0)
        tot++;
      if (grid[i][j] == 2)
        rotten.push({i, j});
    }
  }

  int dx[4] = {0, 0, 1, -1};
  int dy[4] = {1, -1, 0, 0};

  while (!rotten.empty()) {
    int k = rotten.size();
    cnt += k;
    while (k--) {
      int x = rotten.front().first, y = rotten.front().second;
      rotten.pop();
      for (int i = 0; i < 4; ++i) {
        int nx = x + dx[i], ny = y + dy[i];
        if (nx < 0 || ny < 0 || nx >= m || ny >= n || grid[nx][ny] != 1
          continue;
        grid[nx][ny] = 2;
        rotten.push({nx, ny});
      }
    }
    if (!rotten.empty())
      days++;
  }

  return tot == cnt ? days : -1;
}
```

Time Complexity : O(n x n)**4

Space Complexity : O(n x n)

Flood fill

An ***'IMAGE'*** is represented by the 2-D array of positive integers, where each element of 2-D represents the pixel value of the image.

The given 'IMAGE' has ***'N'*** rows and ***'M'*** columns. You are given the location of the pixel in the image as ***('X', 'Y')***(0-based indexing) and a pixel value as ***'P'***.

Your task is to perform a "flood fill" operation on the given coordinate (X, Y) with pixel value 'P'.

Let the current pixel value of ('X', 'Y') be equal to C. To perform the flood fill operation on the coordinate (X, Y) with pixel value 'P' you need to do the following operations in order:

1. Replace the pixel value of ('X', 'Y') with 'P'.

2. Visit all non-diagonal neighbours of ('X', 'Y') having pixel values equal to C and replace their pixel value with 'P'.

3. Non – diagonal neighbours are Up('X' - 1, 'Y'), Down('X' + 1, 'Y'), Left('X', 'Y' - 1), right('X', 'Y' + 1). Also, you cannot go out of bounds.

4. Visit all non-diagonals neighbours of coordinates visited in step 2 having pixel value equal to C and replace their pixel value with 'P'.

5. Repeat step 2, until you have visited all your neighbours

For Example:

For 'N' = 5 , 'M' = 4 , 'X' = 2 , 'Y' = 2 and 'P' = 5
Given 'IMAGE' is shown below:

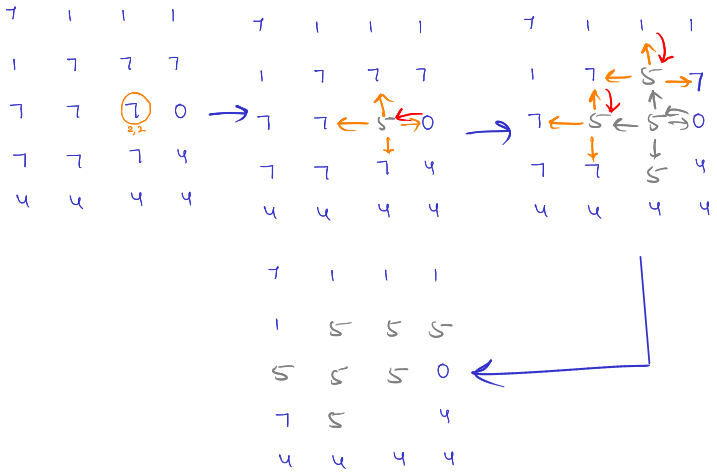
[7, 1, 1, 1]
[1, 7, 7, 7]
[7, 7, 7, 0]
[7, 7, 7, 4]
[4, 4, 4, 4]

After the flood fill operation, we will replace all neighbour's 7s with 5.

So our 'IMAGE' will become:

[7, 1, 1, 1]
[1, 5, 5, 5]
[5, 5, 5, 0]
[5, 5, 5, 4]
[4, 4, 4, 4]

→ Approach 1:-



N=5, M=4, X=2, Y=2, P=5

This is the basic code flow.

Time Complexity : O(N * M)

Space Complexity: O(N*M)

```
int direction[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
bool checkOutOfBound(int rowSize, int columnSize, int x, int y) {
        return ((x >= 0) and (x < rowSize) and (y >= 0) and (y < columnSize));
}

void dfs(vector<vector<int>> &image, int n, int m, int currentX, int currentY, int p, int c) {
        if (checkOutOfBound(n, m, currentX, currentY) == false or image[currentX][currentY] != c) {
                return;
        }

        image[currentX][currentY] = p;

        for (int i = 0; i < 4; i++) {
                dfs(image, n, m, currentX + direction[i][0], currentY + direction[i][1], p, c);
        }
}

vector<vector<int>> floodFill(vector<vector<int>> image, int n, int m, int x, int y, int p) {
        if (image[x][y] == p) {
                return image;
        }
        dfs(image, n, m, x, y, p, image[x][y]);

        return image;
}
```

Distance of nearest cell having one in a Binary Matrix

Problem statement Send feedback

You have been given a binary matrix 'MAT' containing only 0's and 1's of size N x M. You need to find the distance of the nearest cell having 1 in the matrix for each cell.

The distance is calculated as |i1 – i2| + |j1 – j2|, where i1, j1 are the coordinates of the current cell and i2, j2 are the coordinates of the nearest cell having value 1.

Note :

You can only move in four directions which are : Up, Down, Left and Right.

For example :

If N = 3, M = 4

and mat[ ][ ] = { 0, 0, 0, 1,
0, 0, 1, 1,
0, 1, 1, 0 }

then the output matrix will be

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

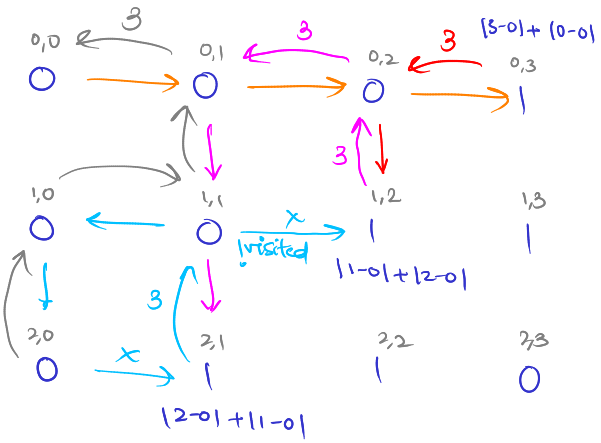
Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= T <= 5
1 <= N <= 2*10^2
1 <= M <= 2*10^2

Where 'T' is the number of test cases, 'N' is the number of rows in the matrix and 'M' is the number of columns in the matrix.

→ Approach :-



→ This was a single loop for finding the nearest 1 for {0,0} which comes out to be 3.

→ We will repeat the same steps for all of the cells which has value of 0.

```
#include <bits/stdc++.h>

void dfs(int i, int j, int x, int y, vector<vector<int>> &visited,
         vector<vector<int>> &distance,
         const vector<vector<int>> &mat, int &ans) {
            int n = mat.size(), m = mat[0].size();
  visited[x][y] = true;
    if (mat[x][y] == 1) {
      int d = abs(i - x) + abs(j - y);
      ans = min(d, ans);
      return;
    }
  int dx[4] = {-1, 1, 0, 0};
  int dy[4] = {0, 0, -1, 1};
  for (int index = 0; index < 4; index++) {
    int a = x + dx[index];
    int b = y + dy[index];
    if (a >= 0 && a < n && b >= 0 && b < m && !visited[a][b])
      dfs(i, j, a, b, visited, distance, mat, ans);
  }
}

vector<vector<int>> nearest(vector<vector<int>> &mat, int n, int m) {
  vector<vector<int>> distance(n, vector<int>(m, 0));
  vector<vector<int>> visited(n, vector<int>(m, 0));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      int ans = INT_MAX;
      if (mat[i][j] == 1) {
        distance[i][j] = 0;
        continue;
      }
      dfs(i, j, i, j, visited, distance, mat, ans);
      distance[i][j] = ans;
      for (auto &vec : visited) {
        for (auto &element : vec) {
          element = 0;
        }
      }
    }
  }
  return distance;
}
```

Replace 'O' with 'X'

Given a 2D array grid G of 'O's and 'X's. The task is to replace all 'O' with 'X' contained in each island. Where, an island is a set of 'O's connected horizontally or vertically and surrounded by 'X' from all of it's boundaries. (Boundary means top, bottom, left and right)

Example:

{{X, X, O, X, X, X},
{X, X, O, X, O, X},
{X, X, X, O, O, X},
{X, O, X, X, X, X},
{X, O, X, O, X, X},
{X, X, X, X, O, X}}

If I enter from the boundaries with cell value O, all the neighbour cells with value O can't be surrounded.
Rest the cells with values O are surrounded.

In the above example, there are 3 islands. Considering Zero based indexing of rows and columns, in the following islands described here, (x,y) represents the element in xth row and yth column.
Island 1: Formed by three elements at (1, 4), (2, 3), (2, 4) positions.
Island 2: Formed by a single element at (3, 1) position.
Island 3: Formed by two elements at (4, 2), (4, 3) positions.
Note:

In the above example, elements at positions (0, 2) and (1,2) do not form an island as there is no 'X' surronding it from the top.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= 'N', 'M' <= 1000

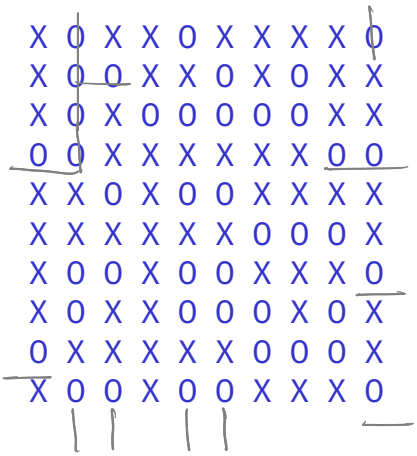
Time Limit: 1 sec

Sample Input 1:

5 4
X X O O
X O X X
X O O X
X O X X
X X X X

Sample Output 1:

X X O O
X X X X
X X X X
X X X X
X X X X



→ Rest all zeros will be converted to 'X'.

X O X X O X X X X O
X O O X X X X X X X
X O X X X X X X X X
O O X X X X X X O O
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X O
X X X X X X X X X X
O X X X X X X X X X
X O O X O O X X X O

```
#include <vector>

void dfs(int row, int col, vector<vector<int>> &vis, char **mat,
         int delrow[], int delcol[], int n, int m) {
  vis[row][col] = 1;

  // check for top, right, bottom, left
  for (int i = 0; i < 4; i++) {
    int nrow = row + delrow[i];
    int ncol = col + delcol[i];
    // check for valid coordinates and unvisited Os
    if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && !vis[nrow][ncol] &&
        mat[nrow][ncol] == 'O') {
      dfs(nrow, ncol, vis, mat, delrow, delcol, n, m);
    }
  }
}

void replaceOWithX(char **arr, int n, int m) {
  int delrow[] = {-1, 0, +1, 0};
  int delcol[] = {0, 1, 0, -1};
  vector<vector<int>> vis(n, vector<int>(m, 0));
  // traverse first row and last row
  for (int j = 0; j < m; j++) {
    // check for unvisited Os in the boundary rows
    // first row
    if (!vis[0][j] && arr[0][j] == 'O') {
      dfs(0, j, vis, arr, delrow, delcol, n, m);
    }

    // last row
    if (!vis[n - 1][j] && arr[n - 1][j] == 'O') {
      dfs(n - 1, j, vis, arr, delrow, delcol, n, m);
    }
  }

  for (int i = 0; i < n; i++) {
    // check for unvisited Os in the boundary columns
    // first column
    if (!vis[i][0] && arr[i][0] == 'O') {
      dfs(i, 0, vis, arr, delrow, delcol, n, m);
    }

    // last column
    if (!vis[i][m - 1] && arr[i][m - 1] == 'O') {
      dfs(i, m - 1, vis, arr, delrow, delcol, n, m);
    }
  }

  // if unvisited O then convert to X
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (!vis[i][j] && arr[i][j] == 'O')
        arr[i][j] = 'X';
    }
  }

}
```

Time Complexity : $O(N) + O(M) + O(N \times M \times 4)$

Space Complexity : $O(N \times M)$

Matrix Traps

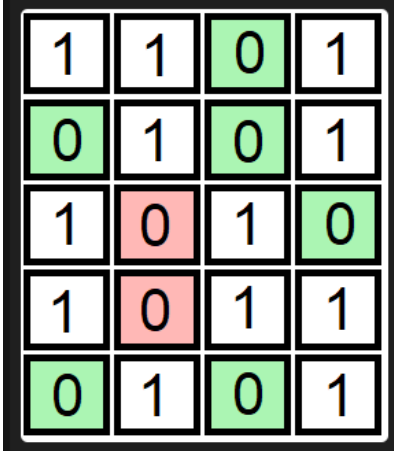
Given a *'N*M'* matrix. Each cell in the matrix can have a value of either 0 or 1. You can start from any zero-valued cell and move to adjacent cells (up, down, left, or right) as long as the destination cell also has a value of 0.

An edge of the matrix is defined as the first row, the first column, the last row, or the last column. A trap cell is a cell where you can start but cannot reach an edge.

Return the number of trap cells.

For Example:

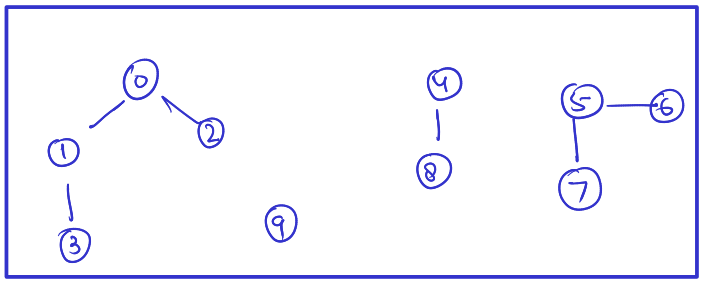
N = 5 and M = 4
Matrix = {{1,1,0,1}, {0,1,0,1}, {1,0,1,0}, {1,0,1,1}, {0,1,0,1}}
The count of trap cells is 2 which are (2,1) and (3,1).
Trap cells are coloured red.



→ Approach:- Use the above approach.

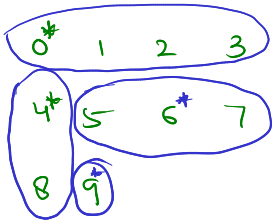
```
void findTraps(int n, int m, int i, int j, vector<vector<int>> &matrix, vector<vector<int>> &trapCells) {
    if(matrix[i][j] == 1) return;

    matrix[i][j] = 1;

    int dx[4] = { -1, 1, 0, 0 };
    int dy[4] = { 0, 0, -1, 1 };

    for(int index = 0; index < 4; index++)  {
        int a = i + dx[index];
        int b = j + dy[index];
        if(a >= 0 && a < n && b >= 0 && b < m) {
            findTraps(n, m, a, b, matrix, trapCells);
        }
    }
}

int matrixTraps(int n,int m,vector<vector<int>> &matrix)
{
    vector<vector<int>> trapCells(n, vector<int>(m, 0));

    // top boundary → matrix[0][j]
    // left boundary → matrix[i][0]
    // right boundary → matrix[i][m - 1]
    // bottom boundary → matrix[n - 1][j]
    int noOfTrapCells = 0;

    for(int j = 0; j < m; j++) {
        // top
        if(matrix[0][j] == 0) findTraps(n, m, 0, j, matrix, trapCells);

        // bottom
        if(matrix[n - 1][j] == 0) findTraps(n, m, n - 1, j, matrix, trapCells);
    }

    for(int i = 0; i < n; i++) {
        // left
        if(matrix[i][0] == 0) findTraps(n, m, i, 0, matrix, trapCells);

        // right
        if(matrix[i][m - 1] == 0) findTraps(n, m, i, m - 1, matrix, trapCells);
    }

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(matrix[i][j] == 0) noOfTrapCells++;
        }
    }
    return noOfTrapCells;
}
```

Disjoint Sets



→ Mainly used in dynamic graph
→ Are 1 & 5 connected?
→ DFS takes O(N+E) [Brute force]
→ Disjoint set gives it in O(1)
  → findParent
  → Union
    → By rank
    → By size

Initially:



Union

Set 1: (0,1) (0,2) (1,3)

Set 2: (4,8) (5,6) (5,7)

Check if vertices are connected

(0,3) (1,5) (7,8)

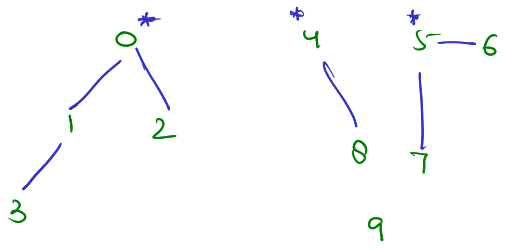
↓

they have the same head

⇒ they must be connected directly or indirectly.

→ Implementation
  → Quick find
  → Quick union

Initial State:



Union (Input)

(0,1) (0,2) (1,3)

(4,8) (5,6) (5,7)

Check if vertices connected

(0,3) ✓ (1,5) ✗ (7,8) ✗

Root Array

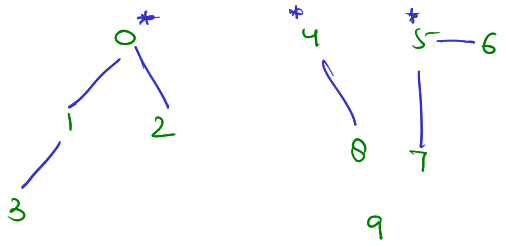
(Parent vertex) Array value	0	0	0	1	4	5	5	5	4	9
(Vertex) Array Index	0	1	2	3	4	5	6	7	8	9

} Initially the array value is same as array index.

→ If parent of a node is the node itself, then that node is the root node.

Root node of 0 & 3 is 0 ⇒ They are connected.

Initial State:



Union

(0,1) (0,2) (1,3)

(4,8) (5,6) (5,7)

Check if vertices connected

(0,3) ✓ (1,5) ✗ (7,8) ✗

For connecting 4 & 7, we just need to change the parent of 4 to 7.

(Parent vertex) Array value	0	1	2	3	4	5	6	7	8	9
(Vertex) Array Index	0	1	2	3	4	5	6	7	8	9

→ **find method** finds the root node of the element x.

→ **union fn** connects x & y together & changes the root nodes to be the same or chooses one of them to be the parent vertex.

→ Union by rank:-

1,2
2,3
4,5
6,7
5,6
3,7

rank

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

parent

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

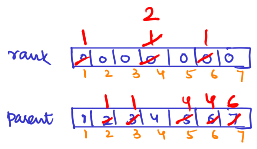
Here the meaning of rank 0 is that there is NO node beneath 1.
The meaning of rank will change eventually once we optimise our code.

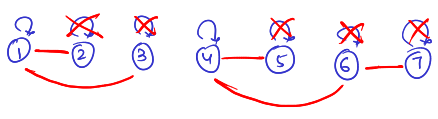
union (1,2)

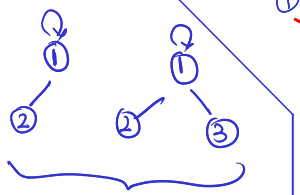
$P_u = 1$ $P_v = 2$
Rank = 0 = 0
} Connect anyone to the other.

union (u,v)
→ find ultimate parent of u & v ie P_u & P_v
→ find rank of P_u & P_v
→ connect smaller rank to larger always.

✓1,2
✓2,3
✓4,5
✓6,7
✓5,6
3,7



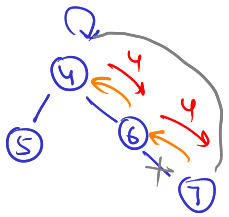


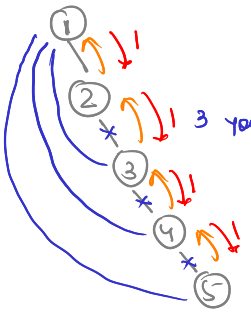


Rank of 1 = 1 in both cases

→ Does 1 & 7 belong to same component?
Get the ultimate parent, if same then they belong to same component
Getting the ultimate parent takes logN.
↓
Need more optimization.

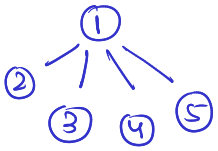
⇒ PATH COMPRESSION :-





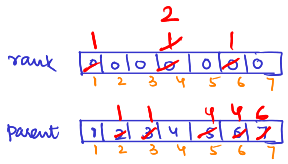
3 your parent is 1 so why are you connected to 2, go and connect to 1.

⇒

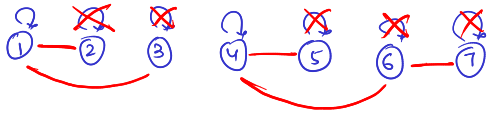


find parent now takes O(1).

→ But there's a twist

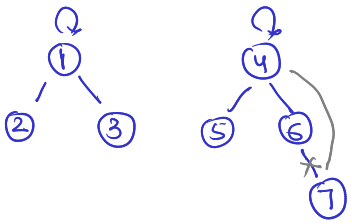


✓1,2
✓2,3
✓4,5
✓6,7
✓5,6
3,7



union of 3 & 7

$P_3 = 1$ for finding parent of 7 we will end up doing path compression.



But rank of 4 is 2

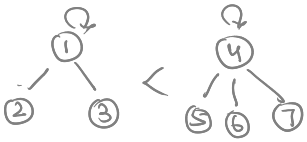
Path compression can't reduce rank.

Ex:

Rank is still 2.

We are not sure if rank decrease

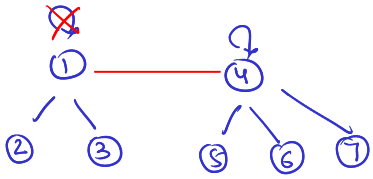
→ Meaning of rank is



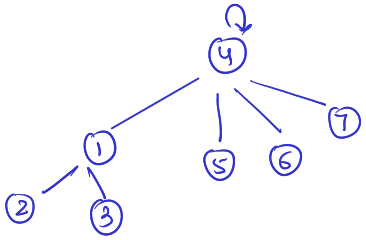
⇒ Now, $P_7 = 4$ $P_3 = 1$

$R = 2$ $R = 1$

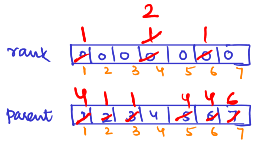
⇒



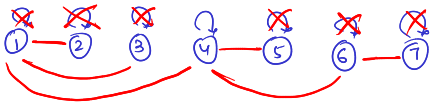
⇒



✓1,2
✓2,3
✓4,5
✓6,7
✓5,6
✓3,7



Rank of 4 won't change coz 4 has larger rank than 1.



Time Complexity for union: $O(4\alpha) \approx O(1)$ → close to 1

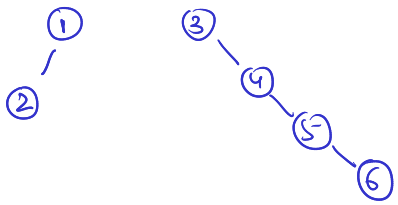
for parent: $O(4\alpha)$

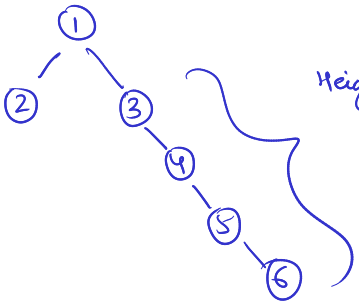
→ Implement path compression :-


```
findParent(u) {
    if (u == parent[u])
        return u;
    return parent[u] = findParent( parent[u]);
}
```

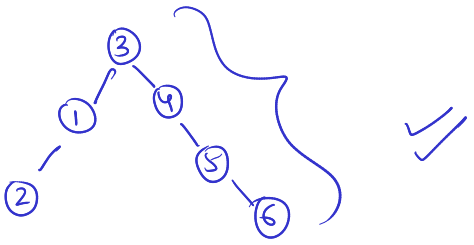
```cpp
#include <bits/stdc++.h>

using namespace std;

class DisjointSet {
  vector<int> rank, parent, size;

public:
  DisjointSet(int n) {
    rank.resize(n + 1, 0);
    parent.resize(n + 1);
    size.resize(n + 1);
    for (int i = 0; i <= n; i++) {
      parent[i] = i;
      size[i] = 1;
    }
  }

  int findUParent(int node) {
    if (node == parent[node])
      return node;
    return parent[node] = findUParent(parent[node]);
  }

  void unionByRank(int u, int v) {
    int ulp_u = findUParent(u);
    int ulp_v = findUParent(v);
    if (ulp_u == ulp_v)
      return;

    if (rank[ulp_u] < rank[ulp_v]) {
      parent[ulp_u] = ulp_v;
    } else if (rank[ulp_v] < rank[ulp_u]) {
      parent[ulp_v] = ulp_u;
    } else {
      parent[ulp_v] = ulp_u;
      rank[ulp_u]++;
    }
  }

  void unionBySize(int u, int v) {
    int ulp_u = findUParent(u);
    int ulp_v = findUParent(v);

    if (ulp_u == ulp_v)
      return;

    if (size[ulp_u] < size[ulp_v]) {
      parent[ulp_u] = ulp_v;
      size[ulp_v] += size[ulp_u];
    } else {
      parent[ulp_v] = ulp_u;
      size[ulp_u] += size[ulp_v];
    }
  }
};

int main() {

  DisjointSet ds(7);

  ds.unionBySize(1, 2);
  ds.unionBySize(2, 3);
  ds.unionBySize(4, 5);
  ds.unionBySize(6, 7);
  ds.unionBySize(5, 6);
  if (ds.findUParent(3) == ds.findUParent(7)) {
    cout << "Yes\n";
  } else {
    cout << "No\n";
  }
  ds.unionBySize(3, 7);
  if (ds.findUParent(3) == ds.findUParent(7)) {
    cout << "Yes\n";
  } else {
    cout << "No\n";
  }
  return 0;
}
```

→ Why connect larger to smaller?





Height increases we have to take longer distance to find the parent.



→ Why are we using Rank anyways becoz after path compression it doesn't gives us height.

⇓

Union by size.

→ Union by size :-

✓1,2
✓2,3
✓4,5
✓6,7
✓5,6
3,7