<https://leetcode.com/problems/number-of-provinces/description/>

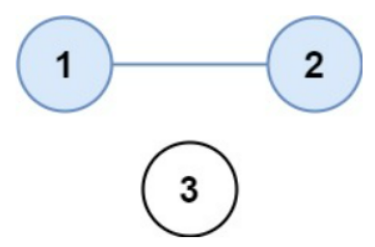
There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

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 ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.

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

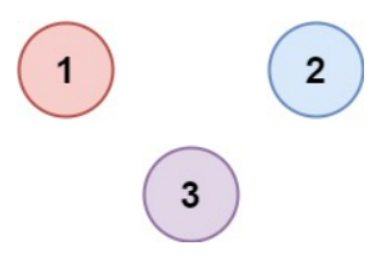
**Example 1:**



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.
* isConnected[i][i] == 1
* isConnected[i][j] == isConnected[j][i]

<https://leetcode.com/problems/friend-circles/>

There are N students in a class. Some of them are friends, while some are not.

Their friendship is transitive in nature. For example, if A is a direct friend of B,  and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N\*N matrix M representing the friend relationship between students in the class.

If M[i][j] = 1, then the ith and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

**Example 1:**

Input:

[[1,1,0],

[1,1,0],

[0,0,1]]

Output: 2

Explanation: The 0th and 1st students are direct friends, so they are in a friend circle. The 2nd student himself is in a friend circle. So return 2.

**Example 2:**

Input:

[[1,1,0],

[1,1,1],

[0,1,1]]

Output: 1

Explanation: The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends, so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

**Note:**

N is in range [1,200].

M[i][i] = 1 for all students.

If M[i][j] = 1, then M[j][i] = 1.

**Attempt 1: 2022-12-16**

**Solution 1:  DFS (10 min)**

class Solution {

public int findCircleNum(int[][] isConnected) {

int n = isConnected.length;

boolean[] visited = new boolean[n];

int count = 0;

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

if(!visited[i]) {

helper(isConnected, visited, i);

count++;

}

}

return count;

}

private void helper(int[][] isConnected, boolean[] visited, int i) {

for(int j = 0; j < isConnected.length; j++) {

if(!visited[j] && isConnected[i][j] == 1) {

visited[j] = true;

helper(isConnected, visited, j);

}

}

}

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Solution 2:  Union Find using adjacent matrix (10 min)**

**Style 1: Simple Union Find**

class Solution {

public int findCircleNum(int[][] isConnected) {

int n = isConnected.length;

int[] parent = new int[n];

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

parent[i] = i;

}

int count = n;

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

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

if(isConnected[i][j] == 1) {

int rootA = find(i, parent);

int rootB = find(j, parent);

if(rootA != rootB) {

parent[rootA] = rootB;

count--;

}

}

}

}

return count;

}

private int find(int x, int[] parent) {

if(x == parent[x]) {

return x;

}

return parent[x] = find(parent[x], parent);

}

// Alternative find style

private int find2(int x, int[] parent) {

while(x != parent[x]) {

parent[x] = parent[parent[x]];

x = parent[x];

}

return x;

}

}

Time Complexity : O(N^2 \* logN)

Space Complexity : O(N)

**Style 2: Union Find with weighted union and path compression**

class Solution {

public int findCircleNum(int[][] isConnected) {

int n = isConnected.length;

int[] parent = new int[n];

int[] rank = new int[n];

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

parent[i] = i;

rank[i] = 1;

}

int count = n;

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

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

if(isConnected[i][j] == 1) {

int rootA = find(i, parent);

int rootB = find(j, parent);

// Weighted union

if(rootA != rootB) {

if(rank[rootA] > rank[rootB]) {

parent[rootB] = rootA;

rank[rootA] += rank[rootB];

} else {

parent[rootA] = rootB;

rank[rootB] += rank[rootA];

}

count--;

}

}

}

}

return count;

}

private int find(int x, int[] parent) {

if(x == parent[x]) {

return x;

}

return parent[x] = find(parent[x], parent);

}

private int find2(int x, int[] parent) {

while(x != parent[x]) {

parent[x] = parent[parent[x]];

x = parent[x];

}

return x;

}

}

Time Complexity : O(N^2 \* α(N)) ~ O(N^2)

Space Complexity : O(N)

**Refer to**

**Complete analysis and solutions for this question, DFS/BFS/UnionFind.**

<https://leetcode.com/problems/number-of-provinces/solutions/112286/complete-analysis-and-solutions-for-this-question-dfs-bfs-unionfind/>

# **Solution1: DFS or BFS**

We can reduce abstract this problem into finding **connected groups** in a undirected graph represented as an **adjacency matrix**.

Since we want to treat the input M as a adjacency matrix, we treated each row from 0 to n - 1 as n nodes. Hence we use a boolean[] to store the visited status.

Therefore, a normal graph traversal algorithms can be utilized to find the number of connected groups in this undirected graph.

## **DFS solution:**

Since the input matrix M is n\*n in size

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int findCircleNum(int[][] M) {

if (M == null || M.length == 0 || M[0].length == 0) return 0;

boolean[] visited = new boolean[M.length];

int count = 0;

for (int i = 0; i < M.length; i++) {

if (!visited[i]) {

count++;

dfs(M, i, visited);

}

}

return count;

}

private void dfs(int[][] M, int i, boolean[] visited) {

for (int j = 0; j < M[i].length; j++) {

if (M[i][j] == 1 && !visited[j]) {

visited[j] = true;

dfs(M, j, visited);

}

}

}

}

## **BFS solution:**

The same idea, but used a Queue to perform the BFS process.

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int findCircleNum(int[][] M) {

if (M == null || M.length == 0 || M[0].length == 0) return 0;

boolean[] visited = new boolean[M.length];

int count = 0;

for (int i = 0; i < M.length; i++) {

if (!visited[i]) {

bfs(M, i, visited);

count++;

}

}

return count;

}

private void bfs(int[][] M, int i, boolean[] visited) {

Queue<Integer> queue = new LinkedList<>();

queue.offer(i);

visited[i] = true;

while (!queue.isEmpty()) {

int curr = queue.poll();

for (int j = 0; j < M[curr].length; j++) {

if (M[curr][j] == 1 && !visited[j]) {

queue.offer(j);

visited[j] = true;

}

}

}

}

}

# **Solution2: Union-find**

Since we've already reduced the question into a connectivity problem, **union-find** algorithm seems to be appliable to this question, for it's suitable to be used for dynamic connectivity problem.

For this question, specifically, we still treat the input M as a **adjacency matrix**. And row index 0 to n-1 as n nodes. We check each edge (M[i][j]) between each node pairs, and union i and j. After we unioned each edge, we check the number of roots, i.e. where i == id[i], and return it as the number of connected components.

Note that we have 2 optimization for the union-find algorithm:

1. During the union() process, we check the size of each connected component and union the smaller one to the greater one. This is called **weighed union** and can flatten the depth of the connected component and improve the efficiency of the union-find algorithm.
2. During the findRoot() process, we used path compression to flatten the depth of the connected component, also improved the efficiency of the algorithm.

**By utilizing this 2 improvements, the time complexity of calling union() for M times is O(n + Mlg\*n), which can be viewed as O(n), because lg\*n can be viewed as a constant.**

class Solution {

// weighed quick union with path compression

public int findCircleNum(int[][] M) {

int[] size = new int[M.length];

int[] id = new int[M.length];

for (int i = 0; i < M.length; i++) {

id[i] = i;

size[i] = 1;

}

for (int i = 0; i < M.length; i++) {

for (int j = 0; j < M[i].length; j++) {

if (M[i][j] == 1) {

union(id, size, i, j);

}

}

}

int count = 0;

for (int i = 0; i < id.length; i++) {

if (i == id[i]) {

count++;

}

}

return count;

}

private void union(int[] id, int[] size, int i, int j) {

int rootI = findRoot(id, i);

int rootJ = findRoot(id, j);

// weighed quick union

if (size[rootI] >= size[rootJ]) {

id[rootJ] = rootI;

size[rootI] += size[rootJ];

} else {

id[rootI] = rootJ;

size[rootJ] += size[rootI];

}

}

private int findRoot(int[] id, int curr) {

while (curr != id[curr]) {

// path compression

id[curr] = id[id[curr]];

curr = id[curr];

}

return curr;

}

}

**Refer to**

**3 different Union Find Time & Space Complexity evolution**

<https://leetcode.com/problems/number-of-provinces/solutions/1461633/python-union-find-clean-concise/>

**Solution 1: Union Find (Naive)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, u):

if u != self.parent[u]:

u = self.find(self.parent[u])

return u

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

self.parent[pu] = pv

return True

class Solution:

def findCircleNum(self, isConnected: List[List[int]]) -> int:

n = len(isConnected)

component = n

uf = UnionFind(n)

for i in range(n):

for j in range(i+1, n):

if isConnected[i][j] == 1 and uf.union(i, j):

component -= 1

return component

Complexity:

* Time: O(N^3), where N <= 200 is number of nodes
* Space: O(N)

**Solution 2: Union Find (Path Compression)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, u):

if u != self.parent[u]:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

self.parent[pu] = pv

return True

class Solution:

def findCircleNum(self, isConnected: List[List[int]]) -> int:

n = len(isConnected)

component = n

uf = UnionFind(n)

for i in range(n):

for j in range(i+1, n):

if isConnected[i][j] == 1 and uf.union(i, j):

component -= 1

return component

Complexity:

* Time: O(N^2 \* logN), where N <= 200 is number of nodes
* Space: O(N)

**Solution 3: Union Find (Union by Size & Path Compression)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

self.size = [1] \* n

def find(self, u):

if u != self.parent[u]:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

if self.size[pu] < self.size[pv]: # Merge pu to pv

self.size[pv] += self.size[pu]

self.parent[pu] = pv

else:

self.size[pu] += self.size[pv]

self.parent[pv] = pu

return True

class Solution:

def findCircleNum(self, isConnected: List[List[int]]) -> int:

n = len(isConnected)

component = n

uf = UnionFind(n)

for i in range(n):

for j in range(i+1, n):

if isConnected[i][j] == 1 and uf.union(i, j):

component -= 1

return component

Complexity:

* Time: O(N^2 \* α(N)) ~ O(N^2), where N <= 200 is number of nodes

Explanation: Using both **path compression** and **union by size** ensures that the **amortized time** per **Union Find** operation is only α(n), which is optimal, where α(n) is the inverse Ackermann function. This function has a value α(n) < 5 for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.Reference: <https://en.wikipedia.org/wiki/Disjoint-set_data_structure> or <https://www.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534> for more information.

* Space: O(N)