Disjoint Set Data Structures

Consider a situation with a number of persons and following tasks to be performed on them.

1. Add a new friendship relation, i.e., a person x becomes friend of another person y.
2. Find whether individual x is a friend of individual y (direct or indirect friend)

Example:

```
We are given 10 individuals say,
a, b, c, d, e, f, g, h, i, j

Following are relationships to be added.
a <-> b
b <-> d
c <-> f
c <-> i
j <-> e
g <-> j

And given queries like whether a is a friend of d
or not.

We basically need to create following 4 groups
and maintain a quickly accessible connection
among group items:
G1 = {a, b, d}
G2 = {c, f, i}
G3 = {e, g, j}
G4 = {h}
```

Problem : To find whether x and y belong to the same group or not, i.e., to find if x and y are direct/indirect friends.

Solution : Partitioning the individuals into different sets according to the groups in which they fall. This method is known as disjoint set data structure which maintains a collection of disjoint sets and each set is represented by its representative which is one of its members.

Approach:

- **How to Resolve sets ?** Initially all elements belong to different sets. After working on the given relations, we select a member as representative. There can by many ways to select a representative, a simple one is to select with the biggest index.
- **Check if 2 persons are in the same group ?** If representatives of two individuals are same, then they'll become friends.

Data Structures used:

Array : An array of integers, called parent[]. If we are dealing with n items, i'th element of the array represents the i'th item. More precisely, the i'th element of the array is the parent of the i'th item. These relationships create one, or more virtual trees.

Tree : It is a disjoint set. If two elements are in the same tree, then they are in the same disjoint set. The root node (or the topmost node) of each tree is called the representative of the set. There is always a single unique representative of each set. A simple rule to identify representative is, if i is the representative of a set, then parent[i] = i. If i is not the representative of his set, then it can be found by traveling up the tree until we find the representative.

Operations :

Find : Can be implemented by recursively traversing the parent array until we hit a node who is parent of itself.

```
// Finds the representative of the set
// that i is an element of
int find(int i) {
    // If i is the parent of itself
    if (parent[i] == i) {
        // Then i is the representative of
        // this set
        return i;
    } else {
        // Else if i is not the parent of
        // itself, then i is not the
        // representative of his set. So we
        // recursively call Find on its parent
        return find(parent[i]);
    }
}
```

Union: It takes, as input, two elements. And finds the representatives of their sets using the find operation, and finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees and the sets.

```
// Unites the set that includes i
// and the set that includes j
void union(int i, int j) {
    // Find the representatives
    // (or the root nodes) for the set
    // that includes i

    int irep = this.Find(i),

    // And do the same for the set
    // that includes j
    int jrep = this.Find(j);

    // Make the parent of i’s representative
    // be j’s  representative effectively
    // moving all of i’s set into j’s set)
    this.Parent[irep] = jrep;
}
```

Improvements (Union by Rank and Path Compression)

The efficiency depends heavily on the height of the tree. We need to minimize the height of tree in order improve the efficiency. We can use Path Compression and Union by rank methods to do so.

Path Compression (Modifications to find()) : It speeds up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into the Find operation. Take a look at the code for more details:

```
// Finds the representative of the set that i
// is an element of.
int find(int i) {
    // If i is the parent of itself
    if (Parent[i] == i) {
        // Then i is the representative
        return i;
    } else {
        // Recursively find the representative.
        int result = find(Parent[i]);

        // We cache the result by moving i’s node
        // directly under the representative of this
        // set
        Parent[i] = result;

        // And then we return the result
        return result;
     }
}
```

Union by Rank: First of all, we need a new array of integers called rank[]. Size of this array is same as the parent array. If i is a representative of a set, rank[i] is the height of the tree representing the set.

Now recall that, in the Union operation, it doesn’t matter which of the two trees is moved under the other (see last two image examples above). Now what we want to do is minimize the height of the resulting tree. If we are uniting two trees (or sets), let’s call them left and right, then it all depends on the rank of left and the rank of right.

- If the rank of left is less than the rank of right, then it’s best to move left under right, because that won’t change the rank of right (while moving right under left would increase the height). In the same way, if the rank of right is less than the rank of left, then we should move right under left.

- If the ranks are equal, it doesn’t matter which tree goes under the other, but the rank of the result will always be one greater than the rank of the trees.

```
// Unites the set that includes i and the set
// that includes j
void union(int i, int j) {
    // Find the representatives (or the root nodes)
    // for the set that includes i
    int irep = this.find(i);

    // And do the same for the set that includes j
    int jrep = this.Find(j);

    // Elements are in same set, no need to
    // unite anything.
    if (irep == jrep)
        return;

    // Get the rank of i’s tree
    irank = Rank[irep],

    // Get the rank of j’s tree
    jrank = Rank[jrep];

    // If i’s rank is less than j’s rank
    if (irank < jrank) {
        // Then move i under j
        this.parent[irep] = jrep;
    }
    // Else if j’s rank is less than i’s rank
    else if (jrank < irank) {
        // Then move j under i
        this.Parent[jrep] = irep;
    }
    // Else if their ranks are the same
    else {

        // Then move i under j (doesn’t matter which one goes where)
        this.Parent[irep] = jrep;

        // And increment the the result tree’s
        // rank by 1
        Rank[jrep]++;
    }
}
```

C++

```
// C++ implementation of disjoint set
#include <iostream>
using namespace std;
class DisjSet {
      int *rank, *parent, n;

public:
      // Constructor to create and
      // initialize sets of n items
      DisjSet(int n) {
            rank = new int[n];
            parent = new int[n];
            this->n = n;
            makeSet();
      }

      // Creates n single item sets
      void makeSet() {
            for (int i = 0; i < n; i++) {
                  parent[i] = i;
            }
      }

      // Finds set of given item x
      int find(int x) {
            // Finds the representative of the set
            // that x is an element of
            if (parent[x] != x) {

                  // if x is not the parent of itself
                  // Then x is not the representative of
                  // his set,
                  parent[x] = find(parent[x]);

                  // so we recursively call Find on its parent
                  // and move i's node directly under the
                  // representative of this set
            }
            return parent[x];
      }

      // Do union of two sets represented
      // by x and y.
      void Union(int x, int y) {
            // Find current sets of x and y
            int xset = find(x);
            int yset = find(y);
            // If they are already in same set
            if (xset == yset)
                  return;

            // Put smaller ranked item under
            // bigger ranked item if ranks are
            // different
            if (rank[xset] < rank[yset]) {
                  parent[xset] = yset;
            }
            else if (rank[xset] > rank[yset]) {
                  parent[yset] = xset;
            }
            // If ranks are same, then increment
            // rank.
            else {
                  parent[yset] = xset;
                  rank[xset] = rank[xset] + 1;
            }
      }
};

int main()
{
      DisjSet obj(5);
      obj.Union(0, 2);
      obj.Union(4, 2);
      obj.Union(3, 1);
      if (obj.find(4) == obj.find(0))
            cout << "Yes\n";
      else
            cout << "No\n";
      if (obj.find(1) == obj.find(0))
            cout << "Yes\n";
      else
            cout << "No\n";

      return 0;
}
```

Output:

```
Yes
No
```

Find the number of Islands | Set 1 (Using Disjoint Set)

Given a boolean 2D matrix, find the number of islands.

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

A cell in the 2D matrix can be connected to 8 neighbours.

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”. We have discussed a DFS based solution in below set 1.

Find the number of islands

We can also solve the question using disjoint set data structure explained here. The idea is to consider all 1 values as individual sets. Traverse the matrix and do a union of all adjacent 1 vertices. Below are detailed steps.

Approach:

1) Initialize result (count of islands) as 0

2) Traverse each index of the 2D matrix.

3) If the value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take the union of the index and its neighbour.

4) Now define an array of size row*column to store frequencies of all sets.

5) Now traverse the matrix again.

6) If the value at index is 1, find its set.

7) If the frequency of the set in the above array is 0, increment the result be 1.

Following is Java implementation of the above steps.

```
// C++ program to fnd number of islands
// using Disjoint Set data structure.
#include <bits/stdc++.h>
using namespace std;

// Class to represent Disjoint Set Data structure
class DisjointUnionSets {

      vector<int> rank, parent;
      int n;

      public:
```

```
	DisjointUnionSets(int n) {
		rank.resize(n);
		parent.resize(n);
		this->n = n;
		makeSet();
	}

	void makeSet() {
		// Initially, all elements
		// are in their own set.
		for (int i = 0; i < n; i++)
			parent[i] = i;
	}

	// Finds the representative of the set that x is an element of
	int find(int x) {
		if (parent[x] != x) {
			// if x is not the parent of itself,
			// then x is not the representative of
			// its set.
			// so we recursively call Find on its parent
			// and move i's node directly under the
			// representative of this set
			return find(parent[x]);
		}
		return x;
	}

	// Unites the set that includes x and the set that includes y
	void Union(int x, int y) {
		// Find the representatives(or the root nodes) for x an y
		int xRoot = find(x);
		int yRoot = find(y);

		// Elements are in the same set, no need to unite anything.
		if (xRoot == yRoot)
			return;

		// If x's rank is less than y's rank then move x under y so that
		// depth of tree remains less
		if (rank[xRoot] < rank[yRoot])
			parent[xRoot] = yRoot;

		// Else if y's rank is less than x's rank
		// Then move y under x so that depth of tree remains less
		else if (rank[yRoot] < rank[xRoot])
			parent[yRoot] = xRoot;

		else // Else if their ranks are the same
		{
			// Then move y under x (doesn't matter which one goes where)
			parent[yRoot] = xRoot;
```

```
                // And increment the the result tree's rank by 1
                rank[xRoot] = rank[xRoot] + 1;
            }
        }
};

// Returns number of islands in a[][]
int countIslands(vector<vector<int>>a) {
        int n = a.size();
        int m = a[0].size();

        DisjointUnionSets *dus = new DisjointUnionSets(n * m);

        /* The following loop checks for its neighbours
        and unites the indexes if both are 1. */
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < m; k++) {
                // If cell is 0, nothing to do
                if (a[j][k] == 0)
                    continue;

                // Check all 8 neighbours and do a Union
                // with neighbour's set if neighbour is
                // also 1
                if (j + 1 < n && a[j + 1][k] == 1)
                    dus->Union(j * (m) + k, (j + 1) * (m) + k);
                if (j - 1 >= 0 && a[j - 1][k] == 1)
                    dus->Union(j * (m) + k, (j - 1) * (m) + k);
                if (k + 1 < m && a[j][k + 1] == 1)
                    dus->Union(j * (m) + k, (j) * (m) + k + 1);
                if (k - 1 >= 0 && a[j][k - 1] == 1)
                    dus->Union(j * (m) + k, (j) * (m) + k - 1);
                if (j + 1 < n && k + 1 < m && a[j + 1][k + 1] == 1)
                    dus->Union(j * (m) + k, (j + 1) * (m) + k + 1);
                if (j + 1 < n && k - 1 >= 0 && a[j + 1][k - 1] == 1)
                    dus->Union(j * m + k, (j + 1) * (m) + k - 1);
                if (j - 1 >= 0 && k + 1 < m && a[j - 1][k + 1] == 1)
                    dus->Union(j * m + k, (j - 1) * m + k + 1);
                if (j - 1 >= 0 && k - 1 >= 0 && a[j - 1][k - 1] == 1)
                    dus->Union(j * m + k, (j - 1) * m + k - 1);
            }
        }

        // Array to note down frequency of each set
        int *c = new int[n * m];
        int numberOfIslands = 0;
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < m; k++) {
                if (a[j][k] == 1) {
                    int x = dus->find(j * m + k);
```

```
                        // If frequency of set is 0,
                        // increment numberOfIslands
                        if (c[x] == 0) {
                              numberOfIslands++;
                              c[x]++;
                        } else
                              c[x]++;
                  }
            }
      }
      return numberOfIslands;
}

// Driver Code
int main(void) {
      vector<vector<int>>a = {{1, 1, 0, 0, 0},
                              {0, 1, 0, 0, 1},
                              {1, 0, 0, 1, 1},
                              {0, 0, 0, 0, 0},
                              {1, 0, 1, 0, 1}};
      cout << "Number of Islands is: "
            << countIslands(a) << endl;
}
```

Output:

```
Number of Islands is: 5
```

```
# Python3 program to find the number of islands using Disjoint Set data structure.

# Class to represent Disjoint Set Data structure
class DisjointUnionSets:
      def __init__(self, n):
            self.rank = [0] * n
            self.parent = [0] * n
            self.n = n
            self.makeSet()

      def makeSet(self):
            # Initially, all elements are in their own set.
            for i in range(self.n):
                  self.parent[i] = i

      # Finds the representative of the set that x is an element of
      def find(self, x):
            if (self.parent[x] != x):

                  # if x is not the parent of itself, then x is not the representative
                  # of its set. so we recursively call Find on its parent and move
                  # i's node directly under the representative of this set
                  return self.find(self.parent[x])
            return x

      # Unites the set that includes x and the set that includes y
      def Union(self, x, y):
            # Find the representatives(or the root nodes) for x an y
            xRoot = self.find(x)
            yRoot = self.find(y)

            # Elements are in the same set, no need to unite anything.
            if xRoot == yRoot:
                  return

            # If x's rank is less than y's rank Then move x under y so that
            # depth of tree remains less
            if self.rank[xRoot] < self.rank[yRoot]:
                  parent[xRoot] = yRoot

            # Else if y's rank is less than x's rank
            # Then move y under x so that depth of tree remains less
            elif self.rank[yRoot] < self.rank[xRoot]:
                  self.parent[yRoot] = xRoot

            else:

                  # Else if their ranks are the same
                  # Then move y under x (doesn't matter which one goes where)
                  self.parent[yRoot] = xRoot

                  # And increment the the result tree's rank by 1
                  self.rank[xRoot] = self.rank[xRoot] + 1

# Returns number of islands in a[][]
def countIslands(a):
      n = len(a)
      m = len(a[0])

      dus = DisjointUnionSets(n * m)

      # The following loop checks for its neighbours
      # and unites the indexes if both are 1.
      for j in range(0, n):
            for k in range(0, m):

                  # If cell is 0, nothing to do
                  if a[j][k] == 0:
                        continue

                  # Check all 8 neighbours and do a Union
                  # with neighbour's set if neighbour is also 1
                  if j + 1 < n and a[j + 1][k] == 1:
                        dus.Union(j * (m) + k, (j + 1) * (m) + k)
                  if j - 1 >= 0 and a[j - 1][k] == 1:
                        dus.Union(j * (m) + k, (j - 1) * (m) + k)
                  if k + 1 < m and a[j][k + 1] == 1:
                        dus.Union(j * (m) + k, (j) * (m) + k + 1)
                  if k - 1 >= 0 and a[j][k - 1] == 1:
                        dus.Union(j * (m) + k, (j) * (m) + k - 1)
                  if (j + 1 < n and k + 1 < m and a[j + 1][k + 1] == 1):
                        dus.Union(j * (m) + k, (j + 1) * (m) + k + 1)
                  if (j + 1 < n and k - 1 >= 0 and a[j + 1][k - 1] == 1):
                        dus.Union(j * m + k, (j + 1) * (m) + k - 1)
                  if (j - 1 >= 0 and k + 1 < m and a[j - 1][k + 1] == 1):
                        dus.Union(j * m + k, (j - 1) * m + k + 1)
                  if (j - 1 >= 0 and k - 1 >= 0 and a[j - 1][k - 1] == 1):
                        dus.Union(j * m + k, (j - 1) * m + k - 1)

      # Array to note down frequency of each set
      c = [0] * (n * m)
      numberOfIslands = 0
      for j in range(n):
            for k in range(n):
                  if a[j][k] == 1:
                        x = dus.find(j * m + k)

                        # If frequency of set is 0, increment numberOfIslands
                        if c[x] == 0:
                              numberOfIslands += 1
                              c[x] += 1
                        else:
                              c[x] += 1
      return numberOfIslands

# Driver Code
a = [[1, 1, 0, 0, 0],
      [0, 1, 0, 0, 1],
      [1, 0, 0, 1, 1],
      [0, 0, 0, 0, 0],
      [1, 0, 1, 0, 1]]
print("Number of Islands is:", countIslands(a))
```

Find the number of islands | Set 2 (Using DFS)

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

Example:

```
Input : mat[][] = {{1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1}
Output : 5
```

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”.

Before we go to the problem, let us understand what is a connected component. A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.

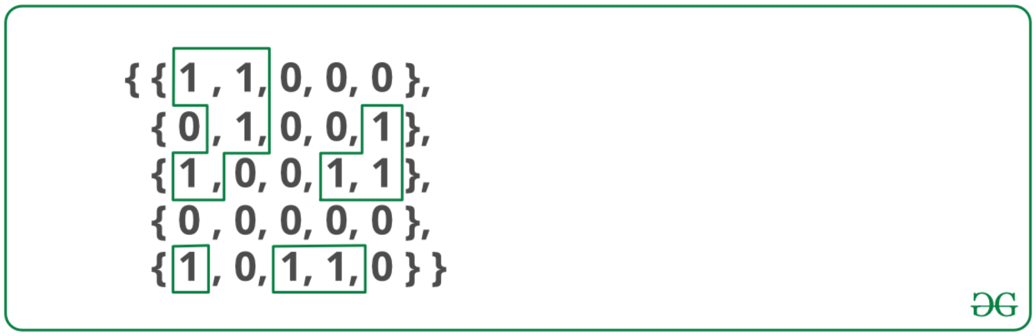
For example, the graph shown below has three connected components.

A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such a graph with only one connected component is called a Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 4 islands



A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

```
// C++ Program to count islands in boolean 2D matrix
#include <bits/stdc++.h>
using namespace std;

#define ROW 5
#define COL 5

// A function to check if a given cell (row, col) can be included in DFS
int isSafe(int M[][COL], int row, int col, bool visited[][COL]) {
      // row number is in range, column number is in range and value is 1
      // and not yet visited
      return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL) && (M[row][col] &&
!visited[row][col]);
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
// the 8 neighbours as adjacent vertices
void DFS(int M[][COL], int row, int col, bool visited[][COL]) {
      // These arrays are used to get
      // row and column numbers of 8
      // neighbours of a given cell
      static int rowNbr[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
      static int colNbr[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

      // Mark this cell as visited
      visited[row][col] = true;

      // Recur for all connected neighbours
      for (int k = 0; k < 8; ++k)
            if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited))
                  DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given boolean 2D matrix
int countIslands(int M[][COL]) {
      // Make a bool array to mark visited cells.
      // Initially all cells are unvisited
      bool visited[ROW][COL];
      memset(visited, 0, sizeof(visited));

      // Initialize count as 0 and
      // travese through the all cells of
      // given matrix
      int count = 0;
      for (int i = 0; i < ROW; ++i)
            for (int j = 0; j < COL; ++j)
                  // If a cell with value 1 is not
                  if (M[i][j] && !visited[i][j]) {
                        // visited yet, then new island found
                        // Visit all cells in this island.
                        DFS(M, i, j, visited);
                        // and increment island count
                        ++count;
                  }
      return count;
}

// Driver code
int main()
{
      int M[][COL] = { { 1, 1, 0, 0, 0 },
                       { 0, 1, 0, 0, 1 },
                       { 1, 0, 0, 1, 1 },
                       { 0, 0, 0, 0, 0 },
                       { 1, 0, 1, 0, 1 } };

      cout << "Number of islands is: " << countIslands(M);
      return 0;
}
```

Output:

```
Number of islands is: 5
```

Time complexity: O(ROW x COL)

```
# Program to count islands in boolean 2D matrix
class Graph:

      def __init__(self, row, col, g):
            self.ROW = row
            self.COL = col
            self.graph = g

      # A function to check if a given cell (row, col) can be included in DFS
      def isSafe(self, i, j, visited):
            # row number is in range, column number is in range and value is 1
            # and not yet visited
            return (i >= 0 and i < self.ROW and
                        j >= 0 and j < self.COL and
                        not visited[i][j] and self.graph[i][j])


      # A utility function to do DFS for a 2D boolean matrix. It only considers
      # the 8 neighbours as adjacent vertices
      def DFS(self, i, j, visited):

            # These arrays are used to get row and column numbers of 8 neighbours
            # of a given cell
            rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1]
            colNbr = [-1, 0, 1, -1, 1, -1, 0, 1]
            # Mark this cell as visited
            visited[i][j] = True

            # Recur for all connected neighbours
            for k in range(8):
                  if self.isSafe(i + rowNbr[k], j + colNbr[k], visited):
                        self.DFS(i + rowNbr[k], j + colNbr[k], visited)


      # The main function that returns count of islands in a given boolean
      # 2D matrix
      def countIslands(self):
            # Make a bool array to mark visited cells.
            # Initially all cells are unvisited
            visited = [[False for j in range(self.COL)]for i in range(self.ROW)]

            # Initialize count as 0 and traverse through the all cells of
            # given matrix
            count = 0
            for i in range(self.ROW):
                  for j in range(self.COL):
                        # If a cell with value 1 is not visited yet,
                        # then new island found
                        if visited[i][j] == False and self.graph[i][j] == 1:
                              # Visit all cells in this island
                              # and increment island count
                              self.DFS(i, j, visited)
                              count += 1

            return count


graph = [[1, 1, 0, 0, 0],
            [0, 1, 0, 0, 1],
            [1, 0, 0, 1, 1],
            [0, 0, 0, 0, 0],
            [1, 0, 1, 0, 1]]


row = len(graph)
col = len(graph[0])

g = Graph(row, col, graph)

print "Number of islands is:"
print g.countIslands()
```

Find the number of islands | Set 3 (Using BFS)

```
// A BFS based solution to count number of
// islands in a graph.
#include <bits/stdc++.h>
using namespace std;

// R x C matrix
#define R 5
#define C 5

// A function to check if a given cell (u, v) can be included in DFS
bool isSafe(int mat[R][C], int i, int j, bool vis[R][C]) {
      return (i >= 0) && (i < R) && (j >= 0) && (j < C) && (mat[i][j] && !vis[i][j]);
}

void BFS(int mat[R][C], bool vis[R][C], int si, int sj) {
      // These arrays are used to get row and
      // column numbers of 8 neighbours of a given cell
      int row[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
      int col[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

      // Simple BFS first step, we enqueue source and mark it as visited
      queue<pair<int, int> > q;
      q.push(make_pair(si, sj));
      vis[si][sj] = true;

      // Next step of BFS. We take out items one by one from queue and
      // enqueue their univisited adjacent
      while (!q.empty()) {
            int i = q.front().first;
            int j = q.front().second;
            q.pop();

            // Go through all 8 adjacent
            for (int k = 0; k < 8; k++) {
                  if (isSafe(mat, i + row[k],
                              j + col[k], vis)) {
                  vis[i + row[k]][j + col[k]] = true;
                  q.push(make_pair(i + row[k], j + col[k]));
                  }
            }
      }
}

// This function returns number islands (connected components) in a graph. It simply
// works as BFS for disconnected graph and returns count of BFS calls.
int countIslands(int mat[R][C]) {
      // Mark all cells as not visited
      bool vis[R][C];
      memset(vis, 0, sizeof(vis));

      // Call BFS for every unvisited vertex Whenever we see an univisted vertex,
      // we increment res (number of islands) also.
      int res = 0;
      for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                  if (mat[i][j] && !vis[i][j]) {
                        BFS(mat, vis, i, j);
                        res++;
                  }
            }
      }
      return res;
}

int main() {
      int mat[][C] = { { 1, 1, 0, 0, 0 },
                              { 0, 1, 0, 0, 1 },
                              { 1, 0, 0, 1, 1 },
                              { 0, 0, 0, 0, 0 },
                              { 1, 0, 1, 0, 1 } };

      cout << countIslands(mat);
      return 0;
}
```

Output:

```
5
```

Time Complexity : O(V + E) where V is number of vertices and E is number of edges. Note that the given solution is simply works as BFS for disconnected graph.

Find the number of distinct islands in a 2D matrix

Given a boolean 2D matrix. The task is 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 equal to another (not rotated or reflected).

Examples:

Input: grid[][] =
{{1, 1, 0, 0, 0},
1, 1, 0, 0, 0},
0, 0, 0, 1, 1},
0, 0, 0, 1, 1}}
Output: 1
Island 1, 1 at the top left corner is same as island 1, 1 at the bottom right corner

Input: grid[][] =
{{1, 1, 0, 1, 1},
1, 0, 0, 0, 0},
0, 0, 0, 0, 1},
1, 1, 0, 1, 1}}
Output: 3
Distinct islands in the example above are: 1, 1 at the top left corner; 1, 1 at the top right corner and 1 at the bottom right corner. We ignore the island 1, 1 at the bottom left corner since 1, 1 it is identical to the top right corner.

Approach: This problem is an extension of the problem Number of Islands.

The core of the question is to know if 2 islands are equal. The primary criteria is that the number of 1's should be same in both. But this cannot be the only criteria as we have seen in example 2 above. So how do we know? We could use the position/coordinates of the 1's.

If we take the first coordinates of any island as a base point and then compute the coordinates of other points from the base point, we can eliminate duplicates to get the distinct count of islands. So, using this approach, the coordinates for the 2 islands in example 1 above can be represented as: [(0, 0), (0, 1), (1, 0), (1, 1)].

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// 2D array for the storing the horizontal and vertical
// directions. (Up, left, down, right}
vector<vector<int> > dirs = { { 0, -1 }, { -1, 0 }, { 0, 1 }, { 1, 0 } };

// Function to perform dfs of the input grid
void dfs(vector<vector<int> >& grid, int x0, int y0,
            int i, int j, vector<pair<int, int> >& v) {
      int rows = grid.size(), cols = grid[0].size();

      if (i < 0 || i >= rows || i < 0 || j >= cols || grid[i][j] <= 0)
            return;
```

```cpp
	// marking the visited element as -1
	grid[i][j] *= -1;

	// computing coordinates with x0, y0 as base
	v.push_back({ i - x0, j - y0 });

	// repeat dfs for neighbors
	for (auto dir : dirs) {
		dfs(grid, x0, y0, i + dir[0], j + dir[1], v);
	}
}

// Main function that returns distinct count of islands in a given boolean 2D matrix
int countDistinctIslands(vector<vector<int> >& grid)
{
	int rows = grid.size();
	if (rows == 0)
		return 0;

	int cols = grid[0].size();
	if (cols == 0)
		return 0;

	set<vector<pair<int, int> > > coordinates;

	for (int i = 0; i < rows; ++i) {
		for (int j = 0; j < cols; ++j) {

			// If a cell is not 1
			// no need to dfs
			if (grid[i][j] != 1)
				continue;

			// vector to hold coordinates
			// of this island
			vector<pair<int, int> > v;
			dfs(grid, i, j, i, j, v);

			// insert the coordinates for
			// this island to set
			coordinates.insert(v);
		}
	}

	return coordinates.size();
}

// Driver code
int main()
{
	vector<vector<int> > grid = { { 1, 1, 0, 1, 1 },
                                                { 1, 0, 0, 0, 0 },
                                                { 0, 0, 0, 0, 1 },
                                                { 1, 1, 0, 1, 1 } };

      cout << "Number of distinct islands is "
            << countDistinctIslands(grid);

      return 0;
}
```

```
# Python implementation of above approach

# 2D array for the storing the horizontal and vertical
# directions. (Up, left, down, right)
dirs = [ [ 0, -1 ], [ -1, 0 ], [ 0, 1 ],  [ 1, 0 ] ]

# Function to perform dfs of the input grid
def dfs(grid, x0, y0, i, j, v):
      rows = len(grid)
      cols = len(grid[0])

      if i < 0 or i >= rows or i < 0 or j >= cols or grid[i][j] <= 0:
            return
      # marking the visited element as -1
      grid[i][j] *= -1

      # computing coordinates with x0, y0 as base
      v.append( [i - x0, j - y0] )

      # repeat dfs for neighbors
      for dir in dirs:
            dfs(grid, x0, y0, i + dir[0], j + dir[1], v)


# Main function that returns distinct count of islands in a given boolean 2D matrix
def countDistinctIslands(grid):
      rows = len(grid)
      if rows == 0:
            return 0

      cols = len(grid[0])
      if cols == 0:
            return 0

      coordinates = []

      for i in range(rows):
            for j in range(cols):
                  # If a cell is not 1 no need to dfs
                  if grid[i][j] != 1:
                        continue

                  # to hold coordinates of this island
                  v = []
                  dfs(grid, i, j, i, j, v)

                  # insert the coordinates for this island to set
                  coordinates.append(v)

      return len(coordinates)

# Driver code
grid = [[ 1, 1, 0, 1, 1 ],
[ 1, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 1 ],
[ 1, 1, 0, 1, 1 ] ]

print("Number of distinct islands is",countDistinctIslands(grid))
```