Solution: Number of Islands

Let's solve the Number of Islands problem using the Union Find pattern.

We'll cover the following

Statement

Let's consider a scenario with an $(m \times n)$ 2D grid containing binary numbers, where `'0'` represents water and `'1'` represents land. If any `'1'` cells are connected to each other horizontally or vertically (not diagonally), they form an island. Your task is to return the total number of islands in the grid.

Constraints:

- $1 \leq$ `grid.length` ≤ 300
- $1 \leq$ `grid[i].length` ≤ 300
- `grid[i][j]` is either `'0'` or `'1'`.

Solution

The key approach is to traverse the 2D grid and join adjacent cell 1s horizontally or vertically. In the end, we return the number of connected components.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Let's start by counting the number of cell 1s in the grid. This can be done by declaring an instance of the **Union Find** class.

Here's how the **Union Find** class initializes itself:

- We initialize a variable, `count`, as 0.
- We initialize a list, `parent`, by traversing each element `grid[i][j]` as follows:
  - If we encounter a cell 1, we append the current index, `i * n + j`, of the grid according to the row major order and increment `count` by 1 (see **lines 18–20** in the **Union Find** class).

  - If a cell 0 is encountered, we append -1 to the array (see **line 22** in the **Union Find** class) because cell 0s in the grid will be ignored anyway.

At the end of this traversal, `count` will contain the number of occurrences of cell 1 in the grid (see **line 27** in `main.java`). Right now, all cell 1s are disconnected from each other. Let's see how we can connect them to form an island and count the total number of islands in the grid.

Java

main.java

UnionFind.java

```java
    public void union(int v1, int v2) {
        int p1 = find(v1);
        int p2 = find(v2);
        if (p1 != p2) {
            // The absolute value of the root node represents the size of thi
            // Make the one with larger size be the new parent
            if (this.rank.get(p1) > this.rank.get(p2)) {
                this.parent.set(p2, p1);
            } else if (this.rank.get(p1) < this.rank.get(p2)) {
                this.parent.set(p1, p2);
            } else {
                this.parent.set(p2, p1);
                this.rank.set(p1, this.rank.get(p1) + 1);
            }
            count--;
        }
    }

    // Function to find the root parent of a node
    public int find(int v) {
        if (this.parent.get(v) != v) {
            this.parent.set(v, this.find(this.parent.get(v)));
        }
        return this.parent.get(v);
    }
}

```

Storing the indexes of grid value 1s in a parent list

For now, we have the count of the total number of cell 1s in the grid stored in `count`. However, we aim to connect all the neighboring cell 1s on the top, bottom, left, and right into components. This will reduce our count from the total number of cell 1s to the total number of connected components of cell 1s.

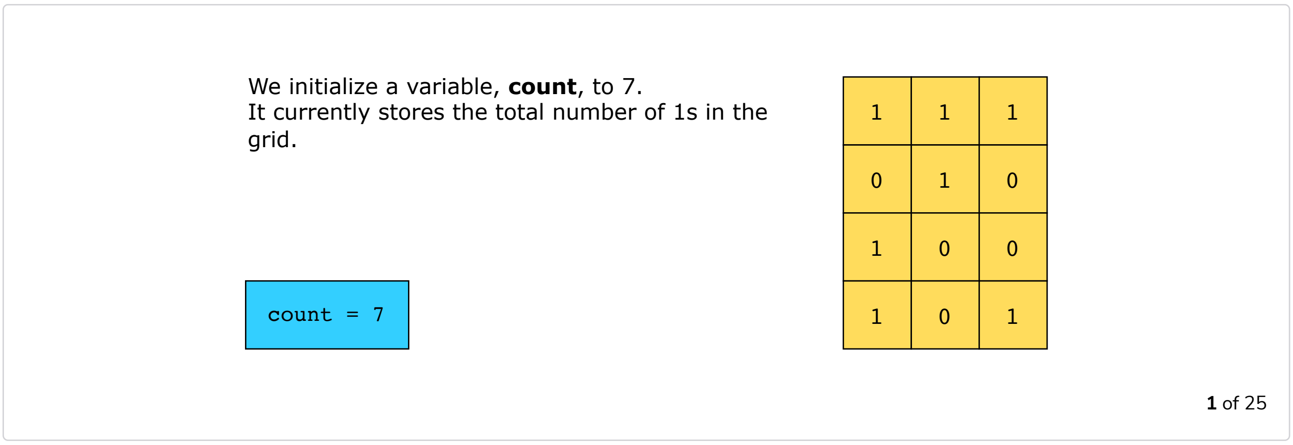
We can achieve this by using the union find algorithm to connect neighboring cell 1s, decrementing the `count` whenever a connection is made.

We'll update the code by traversing each element of the grid as follows:

- If a cell 1 is encountered, we do the following:
  - We update the value of the grid from 1 to 0.
  - We check the value of each neighbor on the top, bottom, left, and right of the current element. If the value is 1, we perform the union operation between the current element and its neighbor, which consists of 1.

    - Inside the union operation, we check if the parent of the current element and its neighbor is different.
        - If so, we connect these two cells and decrement `count` by 1.
        - Otherwise, if their parent is the same, they are already part of a connected component, so we do not decrement `count`.
- At the end of the traversal, `count` will contain the total number of islands.

The slides below illustrate how we would like the algorithm to run:

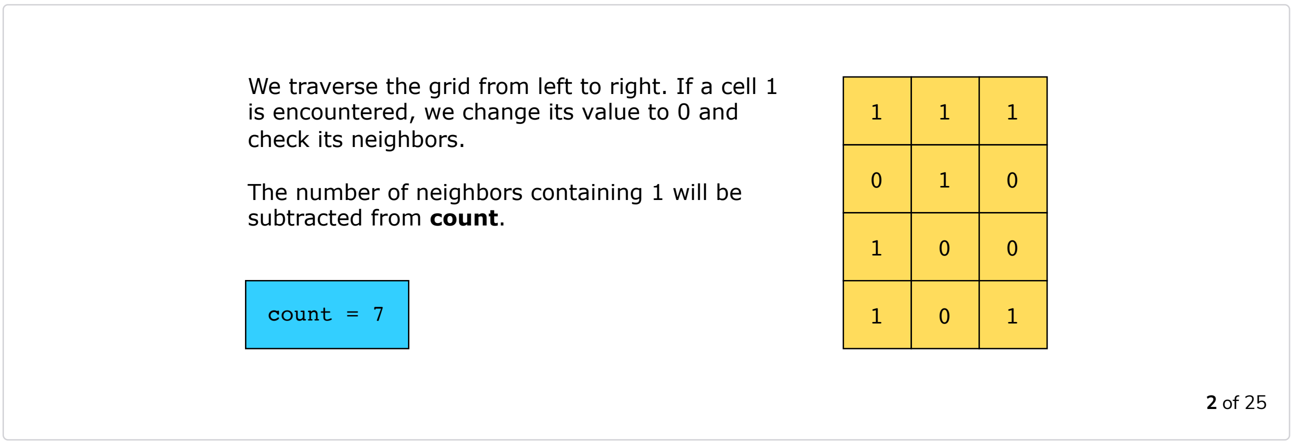


We initialize a variable, **count**, to 7.
It currently stores the total number of 1s in the grid.

```
count = 7
```

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

1 of 25



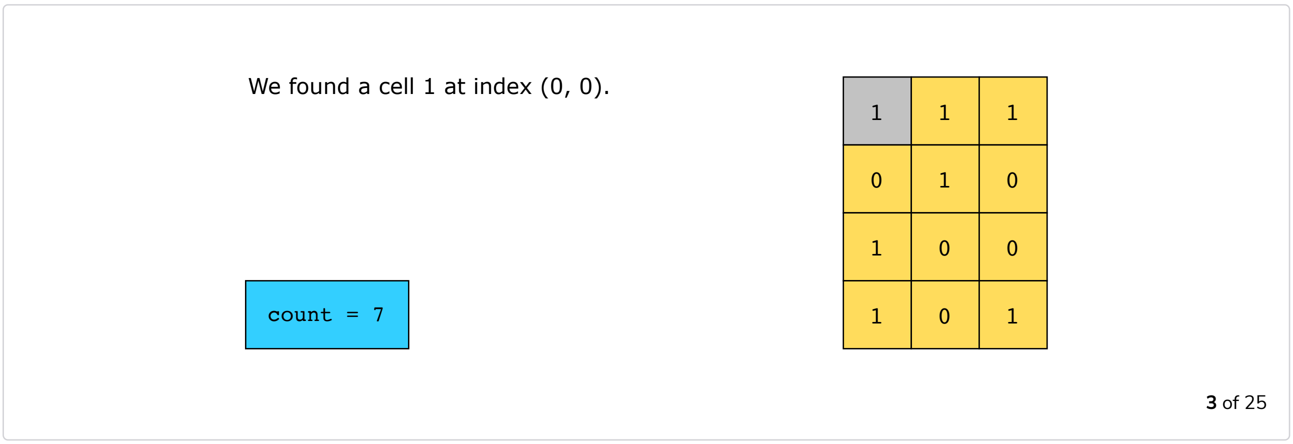
We traverse the grid from left to right. If a cell 1 is encountered, we change its value to 0 and check its neighbors.

The number of neighbors containing 1 will be subtracted from **count**.

```
count = 7
```

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

2 of 25

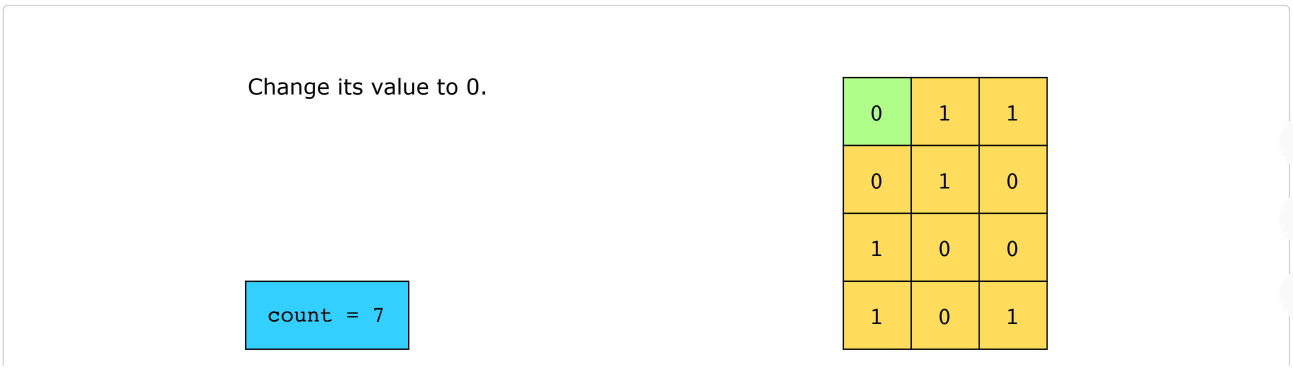


We found a cell 1 at index (0, 0).

```
count = 7
```

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

3 of 25



Change its value to 0.

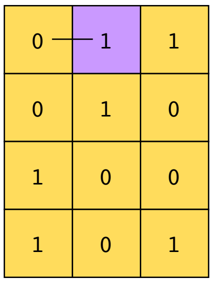
```
count = 7
```

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

4 of 25

This cell has one neighbor containing 1, so we connect the two by taking the union between them and subtract 1 from **count**.

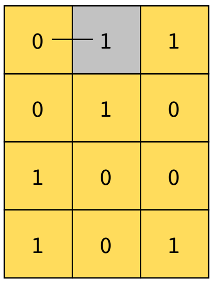
`count = 7`



5 of 25

We found a 1 at index (0, 1).

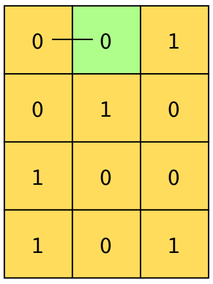
`count = 6`



6 of 25

Change its value to 0.

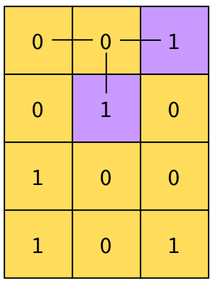
`count = 6`



7 of 25

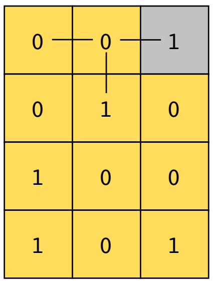
This cell has two neighbors containing 1, so we connect the two by taking the union between them and subtract 2 from **count**.

`count = 6`



8 of 25

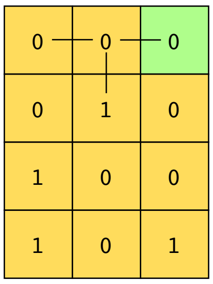
We found a 1 at index (0, 2).



`count = 4`

9 of 25

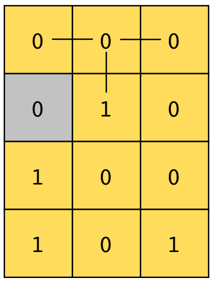
Change its value to 0.



`count = 4`

10 of 25

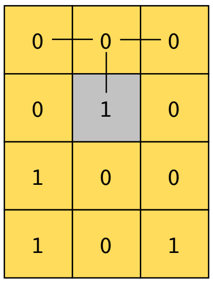
A 0 is found, so we move to the next element in the row.



`count = 4`

11 of 25

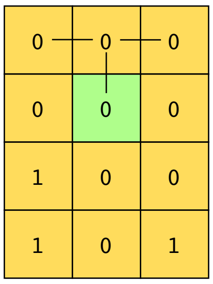
We found a 1 at index (1, 1).



`count = 4`

12 of 25

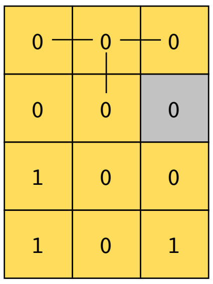
Change its value to 0.



`count = 4`

13 of 25

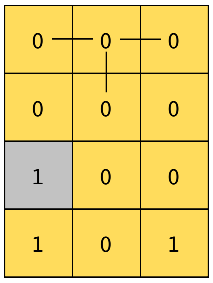
A 0 is found, so we move to the next element.



`count = 4`

14 of 25

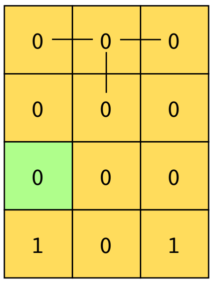
We found a 1 at index (2, 0).



`count = 4`

15 of 25

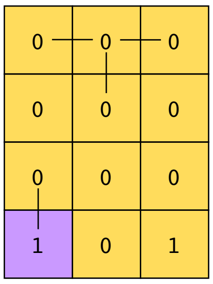
Change its value to 0.



`count = 4`

16 of 25

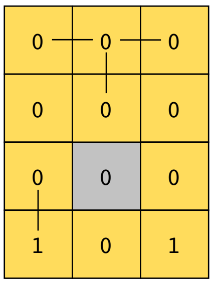
This cell has one neighbor containing 1, so we connect the two by taking the union between them and subtract 1 from **count**.



`count = 3`

17 of 25

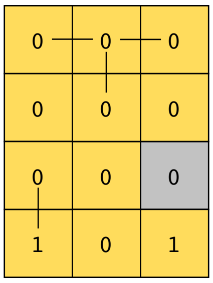
A 0 is found, so we move to the next element.



`count = 3`

18 of 25

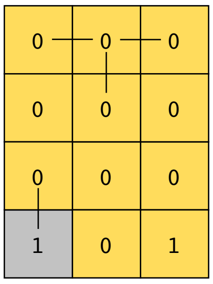
A 0 is found, so we move to the next element.



`count = 3`

19 of 25

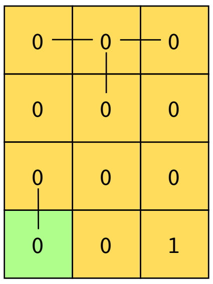
We found a 1 at index (3, 0).



`count = 3`

20 of 25

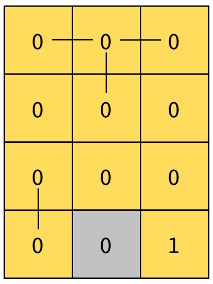
Change its value to 0.



`count = 3`

21 of 25

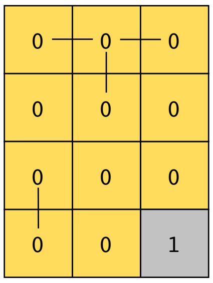
A 0 is found, so we move to the next element.



`count = 3`

22 of 25

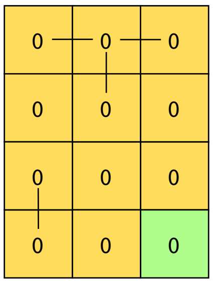
We found a 1 at index (3, 2).



`count = 3`

23 of 25

Change its value to 0.

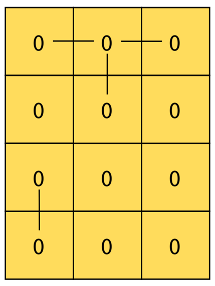


`count = 3`

24 of 25

We've iterated the whole grid. The total number of connected components comes out to be three. Therefore, the number of islands in the given input grid is 3.

count = 3



25 of 25

Let's look at the code for this solution below:

Java

main.java

UnionFind.java

```java
import java.util.*;

class NoOfIslands {
    public static void printGrid(ArrayList<ArrayList<Character>> grid) {
        for (int i = 0; i < grid.size(); i++) {
            System.out.print("\t\t[");
            for (int j = 0; j < grid.get(i).size() - 1; j++) {
                System.out.print("'" + grid.get(i).get(j) + "', ");
            }
            System.out.println("'" + grid.get(i).get(grid.get(i).size() - 1)
```

```java
    public static int numIslands(ArrayList<ArrayList<Character>> grid) {
        // If the grid is empty, then return 0
        if (grid.size() == 0)
            return 0;

        // Get the number of rows and columns in the grid
        int cols = grid.get(0).size();
        int rows = grid.size();

        // Create a Union Find object to represent the islands in the grid
        UnionFind unionFind = new UnionFind(grid);

        // Iterate over each cell in the grid
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
```

Number of Islands

Just the code

Here's the complete solution to this problem:

Java

main.java

UnionFind.java

```java
import java.util.*;

class NoOfIslands {
    public static int numIslands(ArrayList<ArrayList<Character>> grid) {
        if (grid.size() == 0)
            return 0;

        int cols = grid.get(0).size();
        int rows = grid.size();

        UnionFind unionFind = new UnionFind(grid);

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                if (grid.get(r).get(c) == '1') {
                    grid.get(r).set(c, '0');
                    if (r - 1 >= 0 && grid.get(r - 1).get(c) == '1')
                        unionFind.union(r * cols + c, (r - 1) * cols + c);
                    if (r + 1 < rows && grid.get(r + 1).get(c) == '1')
                        unionFind.union(r * cols + c, (r + 1) * cols + c);
                    if (c - 1 >= 0 && grid.get(r).get(c - 1) == '1')
                        unionFind.union(r * cols + c, r * cols + c - 1);
```