Solution: Last Day Where You Can Still Cross

Let's solve the Last Day Where You Can Still Cross problem using the Union Find pattern.

We'll cover the following

Statement

You are given two integers, `rows` and `cols`, which represent the number of rows and columns in a 1-based binary matrix. In this matrix, each 0 represents land, and each 1 represents water.

Initially, on day 0, the whole matrix will just be all 0s, that is, all land. With each passing day, one of the cells of this matrix will get flooded and, therefore, will change to water, that is, from 0 to 1. This continues until the entire matrix is flooded. You are given a 1-based array, `waterCells`, that records which cell will be flooded on each day. Each element $waterCells[i] = [r_i, c_i]$ indicates the cell present at the ${r_i}^{th}$ row and ${c_i}^{th}$ column of the matrix that will change from land to water on the i^{th} day.

We can cross any cell of the matrix as long as it's land. Once it changes to water, we can't cross it. To cross any cell, we can only move in one of the four cardinal directions. Given the number of rows and columns of a 1-based binary matrix and a 1-based array, `waterCells`, you are required to find the last day where you can still cross the matrix, from top to bottom, by walking over the land cells only.

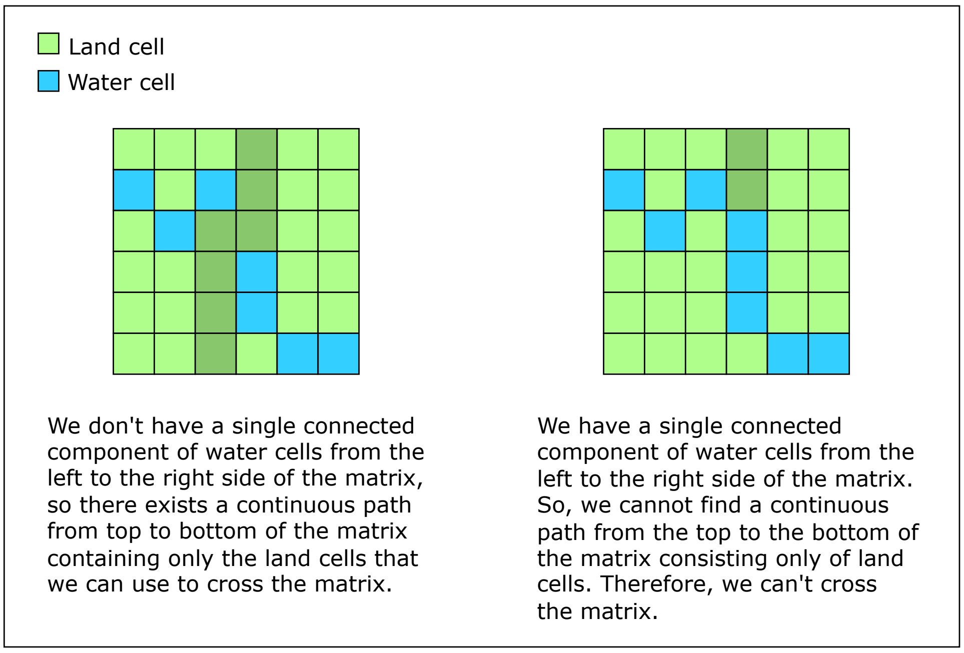
> **Note:** You can start from any cell in the top row, and you need to be able to reach just one cell in the bottom row for it to count as a crossing.

Constraints:

- $2 \leq$ `rows`, `cols` $\leq 2 \times 10^4$
- $4 \leq$ `rows` $\times$ `cols` $\leq 2 \times 10^4$
- `waterCells.length` $==$ `rows` $\times$ `cols`
- $1 \leq r_i \leq$ `rows`
- $1 \leq c_i \leq$ `cols`
- All values of `waterCells` are unique.

Solution

Our aim is to find the maximum number of days where we can cross a 1-based binary matrix, let's say, `matrix`, while it is being flooded one cell at a time. We need to cross `matrix` from top to bottom using a continuous path consisting only of land cells. If, at any time, we encounter a series of connected water cells from the leftmost side of the `matrix` to its rightmost side that blocks our continuous path of land cells, we won't be able to cross the matrix.



Land cell
Water cell
We don't have a single connected component of water cells from the left to the right side of the matrix, so there exists a continuous path from top to bottom of the matrix containing only the land cells that we can use to cross the matrix.
We have a single connected component of water cells from the left to the right side of the matrix. So, we cannot find a continuous path from the top to the bottom of the matrix consisting only of land cells. Therefore, we can't cross the matrix.

In our solution, we'll look for a single component of connected water cells from the left to the right of `matrix`, on each day. Once encountered, we'll return this day as the last day where we can still cross `matrix`.

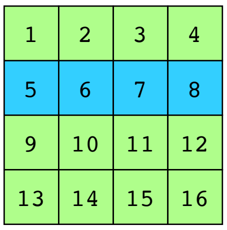
> **Note:** We are not subtracting 1 from our final answer, because, in the context of this problem, we are starting from day 0, so we don't need to adjust the value of days.

Since we are required to find a series of connected water cells, we can consider the cells in the matrix to be vertexes in a graph—each vertex connected to its neighbors—and then try to identify connected components consisting only of water cells using union find. The **union find** pattern is designed to group elements into sets based on a specified property. The pattern uses a disjoint set data structure, such as an array, to keep track of which set each element belongs to. In our case, there are two properties that define whether any two cells should be grouped together:

1. They both need to be water cells.
2. There must be a path between them consisting only of water cells, such that the movement from one water cell to its neighboring water cell is along one of eight directions: up, down, left, right, up-left, up-right, down-left, and down-right.

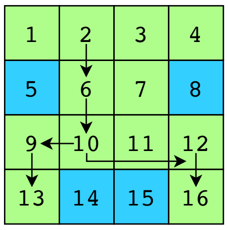
> **Note:** As per the problem statement, there is a restriction on the connectivity of land cells. A valid path of land cells will only contain land cells connected to one of its four neighboring land cells, not eight. However, when checking whether two water cells are connected, we need to take into account eight directions, as explained below.

The most obvious way to block all paths from top to bottom is for all the cells in any row to be flooded. For example, in the matrix on the right, flooding the second row does the job.

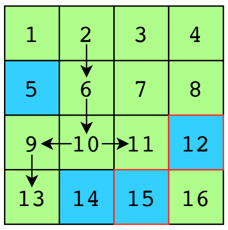


However, this isn't the only way to stop all the paths from the top row to the bottom.

Focusing on each column individually, we might think that flooding at least one cell in each column would be enough, yet as we can see in the matrix on the right, this doesn't quite work. There are two paths to the bottom row from cell 10: via cell 12 or via cell 9.



We have blocked the path to cell 16 by flooding cell 12, and we can similarly flood cell 9 to block the path to cell 13. Here, the water cells 12 and 15 are neighbors, not along a cardinal direction, but along an ordinal or intercardinal direction. The same applies to cells 9 and 14.



This illustrates why, when checking whether two water cells are connected, we need to check in eight directions: up, down, left, right, up-left, up-right, down-left, and down-right.

Now that we have understood why the connectivity of water cells needs to be checked along eight directions, we can use the disjoint set data structure to track which water cells are connected, i.e., which water cells belong to the same connected component.

To use the union find pattern, we create and use the `UnionFind` class that has two primary methods:

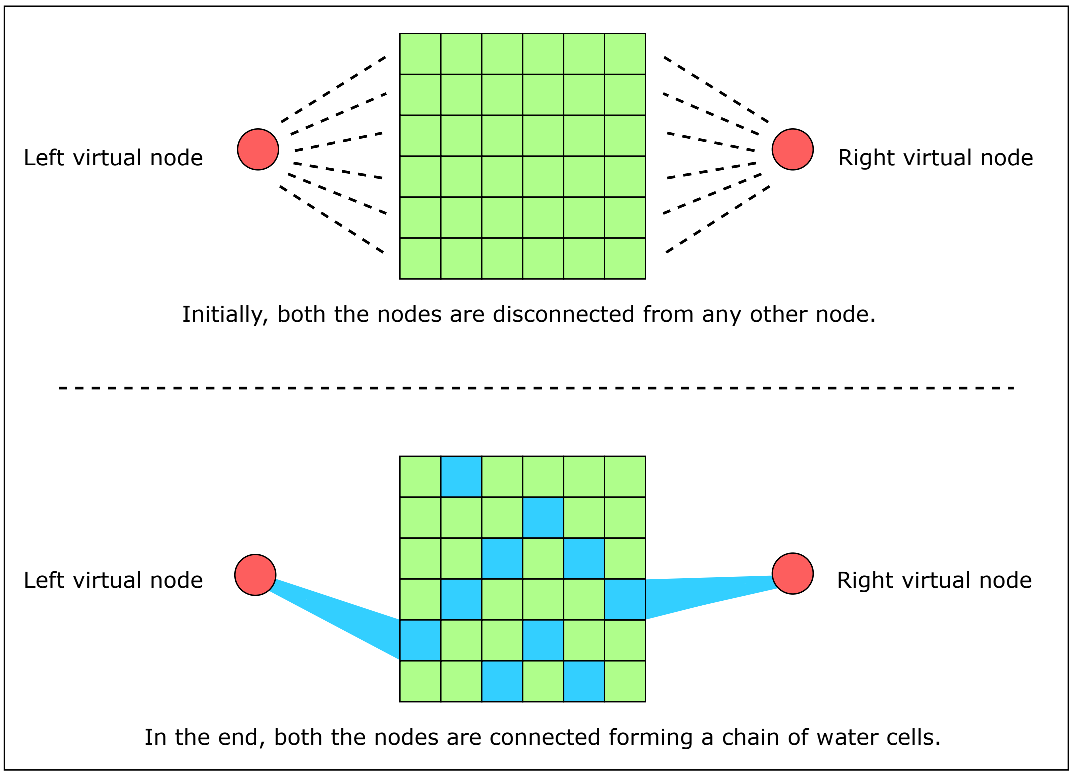
- `union(node1, node2)`: This merges the sets of `node1` and `node2` into one by changing the representative of `node1` to the representative of `node2`.
- `find(node)`: This finds and returns the representative of the set that contains the given node. It uses path compression, which speeds up the subsequent find operations by changing the parent of the visited node to the representative of the subset. This reduces the length of the path of that node to the representative, ensuring we don't have to traverse all the intermediate nodes on future find operations.

> **Note:** In terms of the tree representation of disjoint set union (DSU) data structure, a **representative** is an element in each set that resides at the root of the tree and represents that specific set.

Our solution will first initialize a variable, `day`, to 0, since we are starting from day 0. Next, we'll create a `rows` $\times$ `cols` grid, called `matrix`, with all cells initialized to 0s (since none of the cells have been flooded on day 0). Then, proceeding from day 1, we'll start filling the `matrix` with water cells as per the given `waterCells` array. The value in each cell in `matrix`, identified by the current entry in `waterCells`, is changed from 0 to 1. For example, if we are on day 2 and the value of `waterCells[2]` is $[1, 2]$, we will set the value of `matrix[1][2]` to 1 , indicating that it is now flooded with water.

Since we are solving this problem using the union find pattern, we initialize a `UnionFind` object, which results in the creation of a custom disjoint set data structure: an array, `reps`, of size `(rows * cols) + 2`. The array `reps` is essentially a 1D array representation of `matrix`. The two additional elements in `reps` represent two extra virtual nodes—one at the leftmost side of the matrix and the other at the rightmost side of the matrix. To operate this mapping, we use a helper function, `findIndex`, which converts the coordinates of a cell (i, j) in `matrix` to its corresponding unique index in `reps`. Initially, every element is its own representative, so the value at every index in `reps` is the index itself. During the execution of the `union` operation, when two elements are united, their representatives are set to the same value.

We use the virtual nodes to efficiently check whether the first and the last columns of `matrix` are connected. In simpler words, these nodes help identify whether we have a series of connected water cells from the left to the right side of `matrix`. If a water cell is present in the first column of `matrix`, we use the `union` operation to set the left virtual node as its representative. Similarly, if a water cell is in the last column, we use the `union` operation to set the right virtual node as its representative.

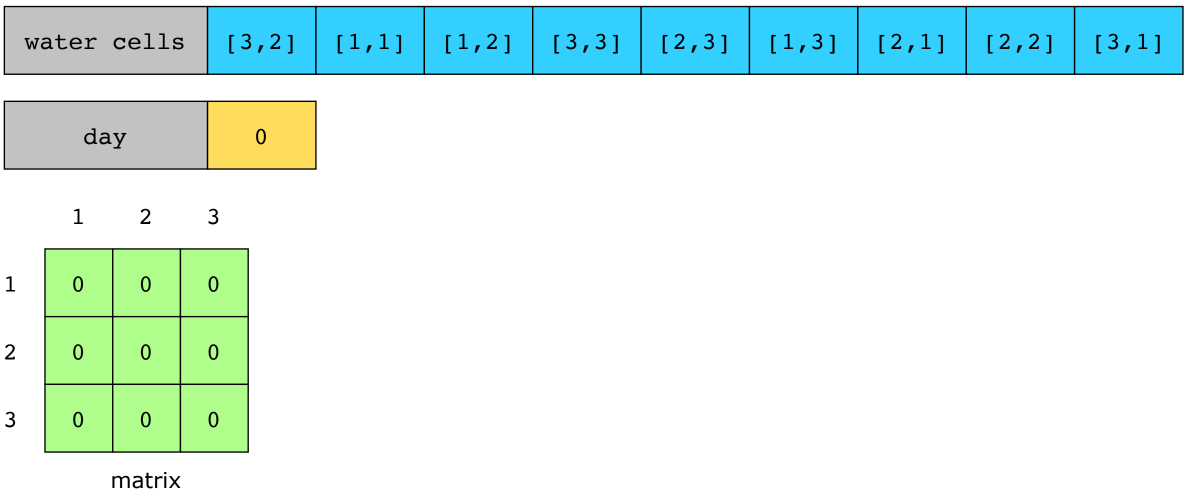

Left virtual node
Right virtual node
Initially, both the nodes are disconnected from any other node.
Left virtual node
Right virtual node
In the end, both the nodes are connected forming a chain of water cells.

Every day, we flood another cell in the matrix, changing its value to 1. At this point, we check whether any of its eight neighboring cells is also a water cell. If it is, we use the `union` operation to connect it with the recently flooded cell. The `union` operation will connect both the cells by making their representatives the same in the array, `reps`. As mentioned above, if the recently flooded cell is at the left or right boundary of `matrix`, it will connect with the corresponding virtual node.

Next, we check to see if the flooding of this cell has created a continuous component connecting the leftmost side of `matrix` with its rightmost side. For this purpose, we use the `find` operation to check whether the left and right virtual nodes are connected, that is, whether or not they have the same representative.

If the two sides are connected, we can no longer cross the matrix from top to bottom, and the current value of `day` is our answer. Otherwise, we increment the value of `day` and repeat the procedure with the next cell in `waterCells`.

The slides below help to understand the solution in a better way.



Let's find the last day where we can still cross the matrix while it's being flooded as per the values in water cells.

1 of 19

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

Java

main.java

UnionFind.java

```java
        int[][][] allWaterCells = {
            {{1, 2}, {2, 1}, {3, 3}, {2, 2}, {1, 1}, {1, 3}, {2, 3}, {3, 2},
            {{1, 1}, {2, 1}, {1, 2}, {2, 2}},
            {{1, 1}, {1, 2}, {1, 3}, {2, 1}, {3, 1}, {2, 2}, {3, 2}, {2, 3},
            {{1, 5}, {2, 5}, {2, 4}, {2, 3}, {2, 2}, {3, 2}, {4, 2}, {4, 1},
             {1, 1}, {1, 2}, {1, 3}, {1, 4}, {3, 3}, {3, 5}, {3, 4}, {4, 5},
            {{1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {2, 5}, {2, 6},
             {3, 2}, {3, 3}, {3, 7}, {4, 7}, {4, 5}, {4, 4}, {4, 3}, {4, 2},
             {5, 5}, {5, 7}, {6, 7}, {7, 7}, {7, 6}, {7, 5}, {7, 4}, {7, 3},
             {6, 4}, {6, 5}, {6, 6}, {5, 6}, {4, 6}, {3, 6}, {3, 5}, {3, 4},
            {{3, 2}, {1, 1}, {1, 2}, {3, 3}, {2, 3}, {1, 3}, {2, 1}, {2, 2},
        };

        int[] allRows = {3, 2, 3, 4, 7, 3};
        int[] allCols = {3, 2, 3, 5, 7, 3};

        for (int i = 0; i < allWaterCells.length; i++) {
            System.out.println(i + 1 + ". \tNumber of rows: " + allRows[i]);
            System.out.println("\tNumber of columns: " + allCols[i]);
            System.out.println("\n\tCells to be flooded: "+ Arrays.deepToStri
            int lastDay = lastDayToCross(allRows[i], allCols[i], allWaterCell

            System.out.println("\n\tLast day where you can still cross: " + l
            System.out.println(PrintHyphens.repeat("-", 100));
        }
    }
}
```

Last Day Where You Can Still Cross

Time complexity

The time complexity of this solution is $O((m \cdot n) \cdot \alpha(m \cdot n))$, where m represents the number of rows in the matrix, n represents the number of columns in the matrix, and α is the inverse Ackermann function that grows very slowly and whose maximum value for all practical purposes is 4.

The initialization of the `UnionFind` object takes $O(m \cdot n)$ time, since it involves creating an array of size $(m \cdot n) + 2$ and initializing every element to a unique index.

There may be, in the worst case, $O(m \cdot n)$ entries in `waterCells`. For each entry, there can be a maximum of eight `union` operations to merge the flooded cell with adjacent water cells, and each `union` operation takes $O(\alpha(m \cdot n))$ time on average.

Therefore, the overall time complexity of this solution is $O((m \cdot n) \cdot \alpha(m \cdot n))$.

Space complexity

The space complexity of the solution above is $O(m \cdot n)$.

Alternative solutions

Now, let’s see what are some other ways to solve this problem.

- **Union find on land cells:**

  This is similar to the solution presented above. Here, we start at the ending state of `matrix` with all the cells flooded. Then, moving from the end of the `waterCells` array towards its start, we start rolling back the flooding (one cell at a time), that is, changing that cell’s value from 1 to 0. After changing a cell from water to land, we check the land cells adjacent to it in all four cardinal directions, and if any are found, we connect them to the recently reverted cell. To check whether we got a single connected component of land cells from top to bottom of `matrix`, we keep two virtual nodes—one at the top and the other at the bottom. The moment we find these virtual nodes connected to each other, we'll return this day as the

  The time and space complexity of this solution is the same as the solution above.

- **Breadth-first search (BFS) with binary search:**

  In this solution, for each new flooded cell on the i^{th} day, we use BFS to figure out whether we have a path of land cells from top to bottom to cross `matrix`. There is a total of $(m \cdot n)$ cells that needs to be flooded as per `waterCells`, and finding a path from the top to the bottom row for each of them would be highly inefficient. Therefore, we use binary search to reduce this number. Our binary search method is guided by these two observations:

  - If we can cross on day i, we can cross on any day before that day.

- If we can’t cross on day i, we cannot cross on any day after that day.

For example, if there are 16 cells in `waterCells`, then, instead of searching for paths on each of the 16 days, we start from the middle cell, the 8^{th} cell. We flood all the cells mentioned in `waterCells` up to the 8^{th} day and then check if a path to cross `matrix` exists. If it does, then we flood all the cells mentioned up to the 12^{th} day, and check again. If no path exists, we roll back the flooding to the 10^{th} day and check again, and so on.

To find a continuous path from top to bottom to cross `matrix`, we start the BFS from the top row of `matrix` and insert all its land cells into a queue. Then, we start to dequeue the elements in the queue one by one until it’s empty. Whenever we dequeue a cell, we check all of its adjacent land cells, and if we have not visited them yet, we add them to the queue. After checking all of its adjacent land cells, we mark the cell as visited and process the next element in the queue. We repeat this until we reach a cell that is in the bottom row of `matrix`. At this point, we have found a path from the top row to the bottom row. Another possible terminating condition for the BFS is that we are unable to find a path from any of the cells in the top row that reaches a cell in the bottom row.

Overall, the binary search takes $log(m \cdot n)$, and the BFS takes $O(m \cdot n)$ time, so the time complexity of this solution is $O((m \cdot n) \cdot log(m \cdot n))$. The space complexity of this solution is $O(m \cdot n)$.

- **Depth-first search (DFS) with binary search:**

  This solution is similar to the one we just discussed, except it uses DFS instead of BFS. On each day, for each cell in the top row of `matrix`, we explore its unvisited neighboring land cells recursively. We continue doing this until we reach the bottom row, or until all reachable land cells in `matrix` have been