2658. Maximum Number of Fish in a Grid

You are given a **0-indexed** 2D matrix `grid` of size `m x n`, where `(r, c)` represents:

- A **land** cell if `grid[r][c] = 0`, or
- A **water** cell containing `grid[r][c]` fish, if `grid[r][c] > 0`.

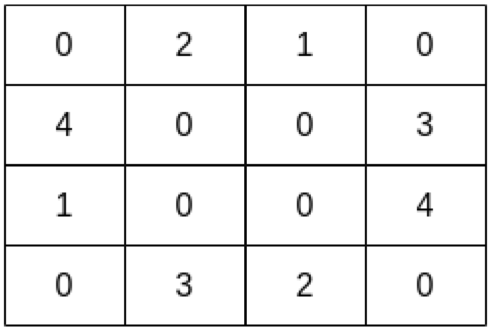
A fisher can start at any **water** cell `(r, c)` and can do the following operations any number of times:

- Catch all the fish at cell `(r, c)`, or
- Move to any adjacent **water** cell.

Return *the **maximum** number of fish the fisher can catch if he chooses his starting cell optimally, or* `0` if no water cell exists.

An **adjacent** cell of the cell `(r, c)`, is one of the cells `(r, c + 1)`, `(r, c - 1)`, `(r + 1, c)` or `(r - 1, c)` if it exists.

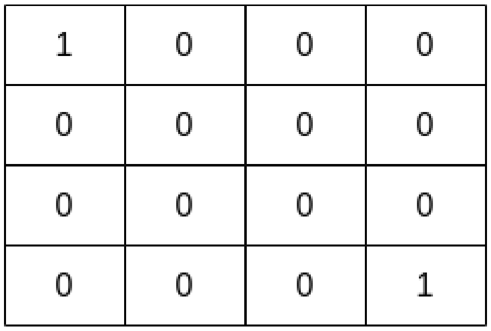
Example 1:



0	2	1	0
4	0	0	3
1	0	0	4
0	3	2	0

```
Input: grid = [[0,2,1,0],[4,0,0,3],
[1,0,0,4],[0,3,2,0]]
Output: 7
Explanation: The fisher can start at cell
(1,3) and collect 3 fish, then move to
cell (2,3) and collect 4 fish.
```

Example 2:



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

```
Input: grid = [[1,0,0,0],[0,0,0,0],
[0,0,0,0],[0,0,0,1]]
Output: 1
Explanation: The fisher can start at
cells (0,0) or (3,3) and collect a single
fish.
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 10`
- `0 <= grid[i][j] <= 10`

2658. Maximum Number of Fish in a Grid

Overview

We are given a `grid` of size `m x n`, where each cell `(r, c)` can either be land or water. The grid is represented by an integer matrix where:

- A land cell is denoted by `0`.
- A water cell contains a number of fish, indicated by a value greater than `0`.

We need to find the largest number of fish that a fisher can collect by starting at an optimal water cell and moving to connected water cells. The fisher can collect fish from any water cell they start from, and then they can move to any adjacent water cell to continue collecting more fish. The fisher can repeat this operation as many times as needed, moving between connected water cells to collect fish.

This problem is closely related to the "***695.Max Area of Island***" problem, which also deals with connected regions in a grid. However, the key difference here is that in this problem, the value in each water cell is not simply `1`, but rather the number of fish in that cell, which adds an extra layer of complexity.

Approach 1: Depth-First Search

Intuition

We can think of the grid as a map of a graph, where each water cell is a node connected to other water cells around it, either up, down, left, or right. The water cells are grouped together, forming distinct regions that are separated by land cells. The goal is to find the largest group of connected water cells, which represents the region with the most fish.

To solve this, we can use a Depth-First Search (DFS). DFS works by exploring every connected node (in this case, the water cells) starting from a given cell. When we find a water cell, we start a DFS from that cell. The DFS will look at all neighboring water cells (in all four directions), marking them as visited to ensure we don't count them again.

As we traverse each connected water region, we also keep a running total of the number of fish in that region. This means that for every new DFS call, we add up all the fish in that group of connected cells.

After exploring all the water cells in one region, we move on to the next unvisited water cell and repeat the process. While doing this, we always track the greatest number of fish encountered in any of the regions. By the time we finish going through the whole grid, we will have found the region with the most fish and that will be our result.

2658. Maximum Number of Fish in a Grid

```
/*
    DFS
    Time complexity: O(mn)
    Space complexity: O(mn)
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution {
    public:
        int findMaxFish(vvi& grid){
            int m=grid.size();
            int n=grid[0].size();

            vvi visited(m,vi(n,0));

            auto dfs=[&](int row,int col,auto& self)->int{
                if(row<0 || col<0 || row>m-1 || col>n-1) return 0;
                if(visited[row][col]) return 0;
                if(grid[row][col]==0) return 0;

                visited[row][col]=1;

                return grid[row][col]
                    +self(row,col+1,self)
                    +self(row,col-1,self)
                    +self(row-1,col,self)
                    +self(row+1,col,self);
            };

            int ans=0;
            for(int i=0;i<m;++i){
                for(int j=0;j<n;++j){
                    if(grid[i][j]==0 || visited[i][j]) continue;
                    ans=std::max(ans,dfs(i,j,dfs));
                }
            }
            return ans;
        }
};
```

Complexity Analysis
Let $\boxed{m}$ be the number of rows and $\boxed{n}$ be the number of columns in the `grid`.

- Time Complexity: $\boxed{O(m \cdot n)}$

  In the worst case, where the `grid` is completely filled with water cells, the algorithm iterates through all `m x n` cells. For each cell, it performs a depth-first search (DFS) to calculate the total fish in the connected region. Therefore, the overall time complexity is $\boxed{O(m \cdot n)}$.

- Space Complexity: $\boxed{O(m \cdot n)}$

  The algorithm uses a `visited` matrix of size `m x n` to track visited cells. Additionally, the depth-first search (DFS) can recurse to explore all connected cells, contributing to the space complexity. Hence, the overall space complexity is $\boxed{O(m \cdot n)}$.

2658. Maximum Number of Fish in a Grid

Approach 2: Breadth-First Search

Intuition

Similar to Depth-First Search (DFS), we can also use a Breadth-First Search (BFS) to explore the grid and find the connected water regions. BFS works by exploring all neighboring cells at the present depth level before moving on to cells at the next level. This means that BFS explores level by level, starting from a water cell and expanding outward to its neighboring water cells.

We start by iterating through the grid and whenever we encounter a water cell that hasn't been visited yet, we initiate a BFS. From that cell, we explore its four neighboring cells (up, down, left, right), checking if they are also water cells and marking them as visited. This continues until all water cells in the current region have been explored.

While performing the BFS, we accumulate the number of fish in the connected region by adding up the values of all the visited water cells. This ensures that we get the total number of fish in that region.

After exploring all neighboring water cells in the current region, we move on to the next unvisited water cell and repeat the BFS process. Throughout the BFS traversal, we keep track of the largest fish count encountered. By the end of the grid traversal, we will have identified the connected water region with the most fish and return that as our result.

2658. Maximum Number of Fish in a Grid

```
/*
    BFS
    Time complexity: O(mn)
    Space complexity: O(mn)
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
typedef std::pair<int,int> ii;
class Solution {
    public:
        int findMaxFish(vvi& grid){
            int m=grid.size();
            int n=grid[0].size();
            vvi directions={{0,1},{0,-1},{-1,0},{1,0}};
            vvi visited(m,vi(n,0));
            auto bfs=[&](int row,int col)->int{
                int fishes=0;
                std::queue<ii> q;
                q.push({row,col});
                visited[row][col]=1;
                while(!q.empty()){
                    auto [cur_row,cur_col]=q.front();
                    q.pop();
                    fishes+=grid[cur_row][cur_col];
                    for(auto& dir: directions){
                        int r=cur_row+dir[0];
                        int c=cur_col+dir[1];
                        if(r<0 || c<0 || r>m-1 || c>n-1) continue;
                        if(grid[r][c]==0) continue;
                        if(visited[r][c]) continue;
                        visited[r][c]=1;
                        q.push({r,c});
                    }
                }
                return fishes;
            };
            int ans=0;
            for(int i=0;i<m;++i){
                for(int j=0;j<n;++j){
                    if(grid[i][j]==0 || visited[i][j]) continue;
                    ans=std::max(ans,bfs(i,j));
                }
            }

            return ans;
        }
};
```

Complexity Analysis
Let $\boxed{m}$ be the number of rows and $\boxed{n}$ be the number of columns in the `grid`.

- Time Complexity: $\boxed{O(m \cdot n)}$

  In the worst case, where the `grid` is completely filled with water cells, the algorithm iterates through all `m \cdot n` cells. For each cell, it performs a Breadth-first search (BFS) to calculate the total fish in the connected region. Therefore, the overall time complexity is $\boxed{O(m \cdot n)}$.

- Space Complexity: $\boxed{O(m \cdot n)}$

  The algorithm uses a `visited` matrix of size `m \cdot n` to track visited cells. Hence, the overall space complexity is $\boxed{O(m \cdot n)}$.

2658. Maximum Number of Fish in a Grid

Approach 3: Union Find Algorithm

Intuition

Another approach to solving problems based on graph connectivity is the union-find data structure.

A disjoint-set data structure also called a union-find data structure or merge-find set, is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. More specifically, it allows us to perform two main operations:

1. **Find**: This operation helps us determine which set a particular element belongs to. In our case, it will help us check if two water cells are part of the same connected region.
2. **Union**: This operation merges two sets into one. It allows us to combine two connected water cells into the same region.

For this problem, we can think of each water cell as an individual set, and the goal is to merge them into larger sets based on their connectivity. As we perform the "Union" operation, we also need to keep track of the total number of fish in each connected component (group of connected water cells).

Our task, as with the previous approaches, is to count the maximum sum of fishes among all the connected components formed in the graph with water cells acting as nodes and an edge between directly connected cells.

First, we treat each water cell as its own separate component, initializing a structure to store the number of fish in each component. Initially, each water cell holds its own fish count.

We then iterate over all the cells in the grid. For each water cell, we check its four neighbors (up, down, left, right). If a neighboring cell is also water, we perform a "Union" operation to merge their components, effectively connecting the two cells. As we do this, we update the fish count for the newly merged component by adding the fish counts from both cells.

After merging the cells, we keep track of the maximum fish count encountered in any connected component. This can be done by maintaining a separate array (let's call it `fishes`) where each entry corresponds to the total fish count of a particular connected component.

At the end of this process, the largest value in the `fishes` array will give us the largest sum of fish in any connected component.

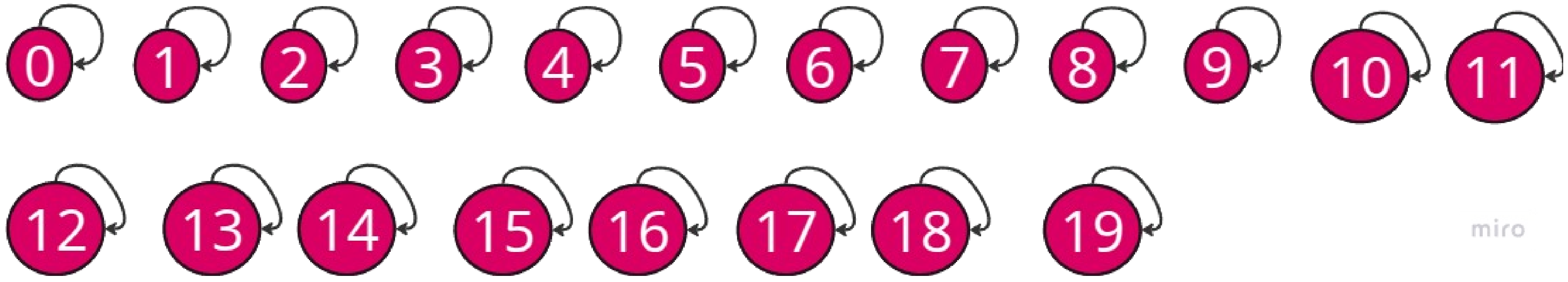
Initial state:

New table

0_{0}	2_{1}	3_{2}	0_{3}	0_{4}
0_{5}	0_{6}	1_{7}	1_{8}	0_{9}
0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

New table

fishes:																				
	0	2	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



Process (0,0):

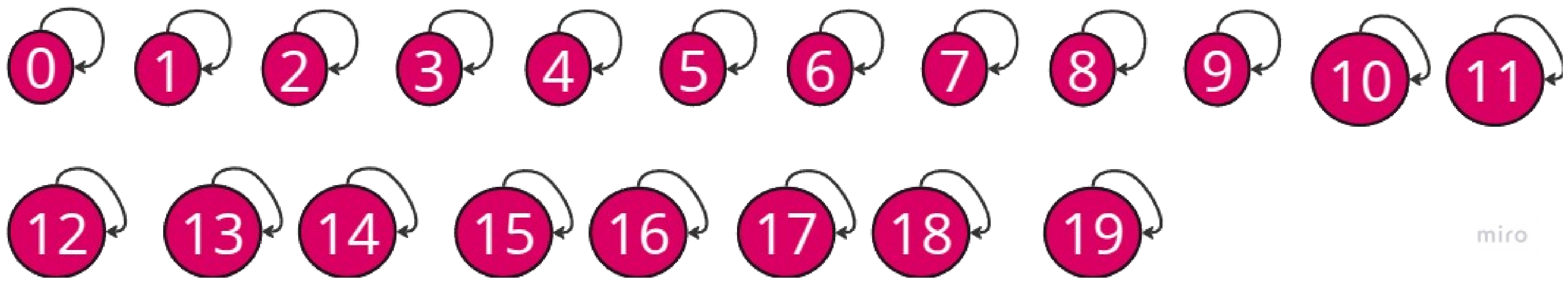
grid[0][0]=0, go next

New table

	0	1	2	3	4
0	0 0	2 1	3 2	0 3	0 4
1	0 5	0 6	1 7	1 8	0 9
2	0 10	0 11	2 12	3 13	0 14
3	1 15	2 16	0 17	0 18	0 19

New table

fishes:	0	2	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



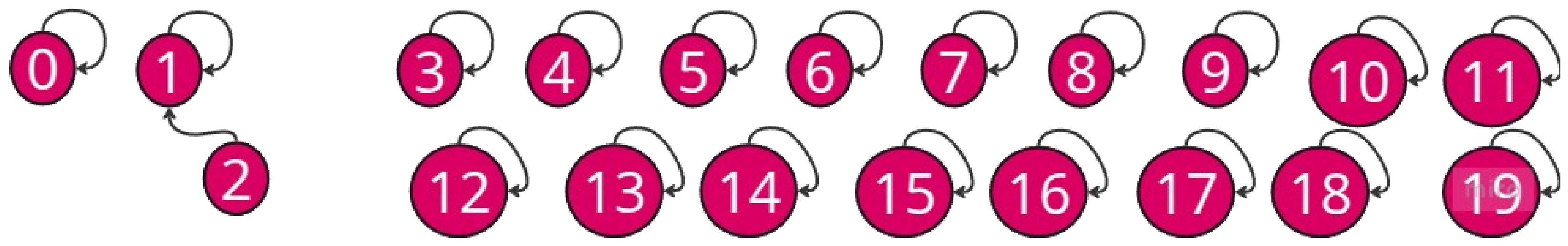
Process (0,2):

grid[0][2]=3, go four directions:

the only cell with a non zero value is (0,1), unify(2,0), already unified

	0	1	2	3	4
0	0_{0}	2_{1}	🟩 3_{2}	0_{3}	0_{4}
1	0_{5}	0_{6}	1_{7}	1_{8}	0_{9}
2	0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
3	1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

fishes:	0	5	🟩 3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

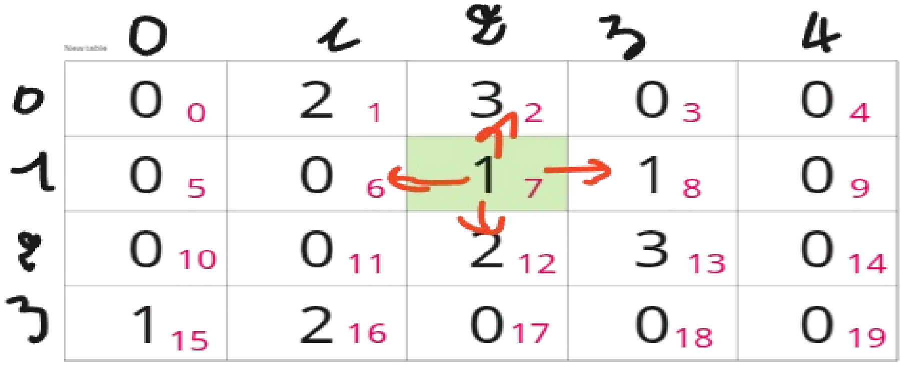


Process (0,3), (0,4),(1,0),(1,1): their values is 0, no need to process them

Process (1,2):

grid[1][2]=1, go four directions:

unify(7,2):
fishes[parent1 of 1]+=fishes[parent1 of 7]
fishes[1]+=fishes[7] =>fishes[1]=5+1

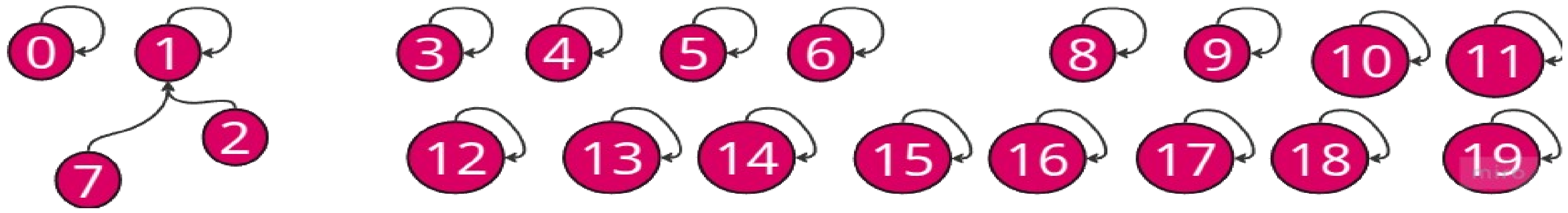


	0	1	2	3	4
0	0 0	2 1	3 2	0 3	0 4
1	0 5	0 6	1 7	1 8	0 9
2	0 10	0 11	2 12	3 13	0 14
3	1 15	2 16	0 17	0 18	0 19

5+1

fishes:

0	6	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19



unify(7,8):

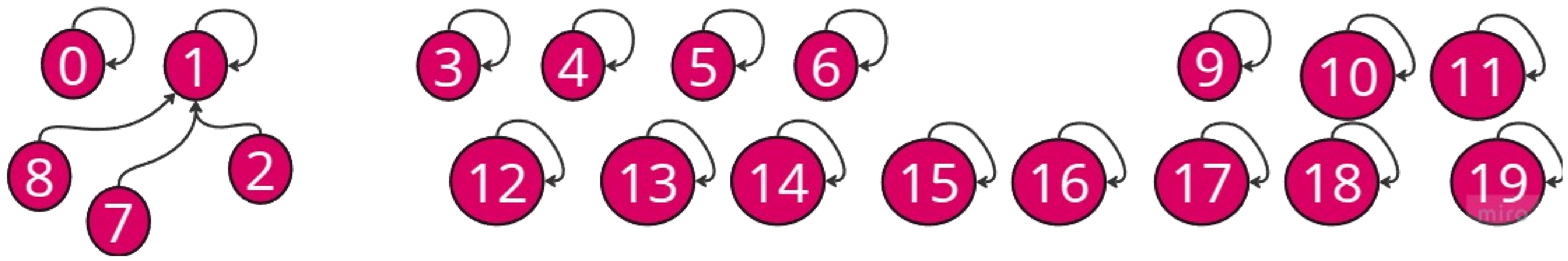
fishes[parent1 of 1]+=fishes[parent1 of 8]

fishes[1]+=fishes[8] =>fishes[1]=6+1

	0	1	2	3	4
0	0 0	2 1	3 2	0 3	0 4
1	0 5	0 6	1 7	1 8	0 9
2	0 10	0 11	2 12	3 13	0 14
3	1 15	2 16	0 17	0 18	0 19

6+1

fishes:	0	7	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



unify(7,12):

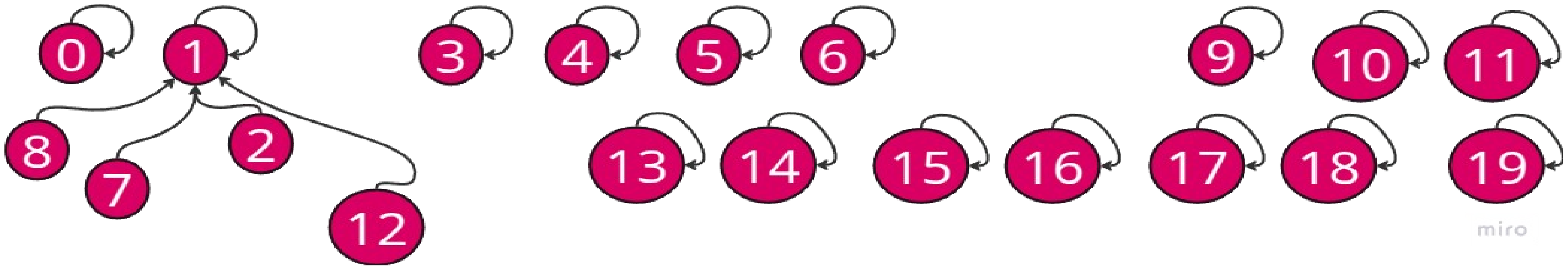
fishes[parent1 of 1]+=fishes[parent1 of 12]

fishes[1]+=fishes[12] =>fishes[1]=7+2

	0	1	2	3	4
0	0_{0}	2_{1}	3_{2}	0_{3}	0_{4}
1	0_{5}	0_{6}	🟩 1_{7}	1_{8}	0_{9}
2	0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
3	1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

7+2

fishes:	0	9	3	0	0	0	0	🟩 1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	🟩 *7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



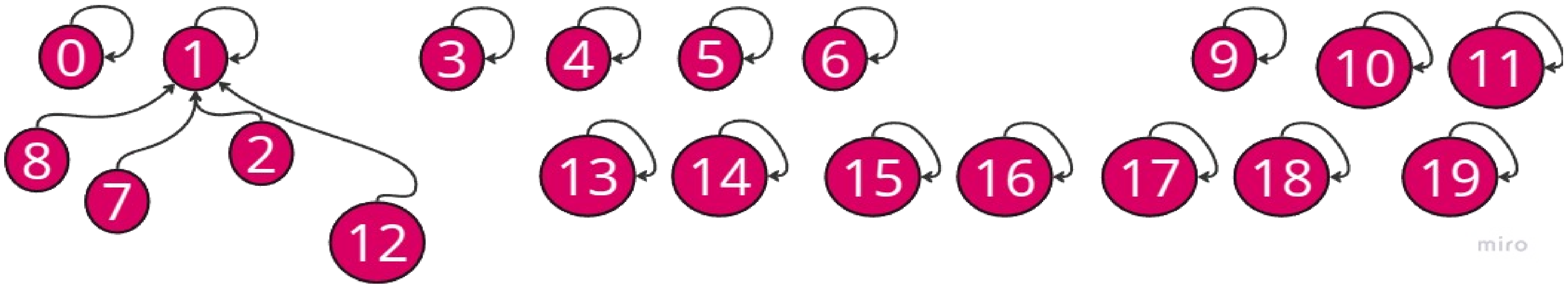
Process (1,3):

grid[1][3]=1, go four directions:

unify(8,7): already unified

	0	1	2	3	4
0	0 0	2 1	3 2	0 3	0 4
1	0 5	0 6	1 7	1 8	0 9
2	0 10	0 11	2 12	3 13	0 14
3	1 15	2 16	0 17	0 18	0 19

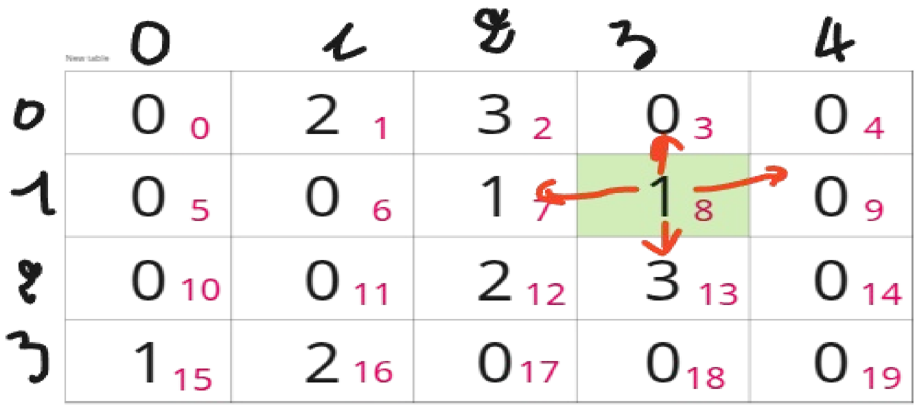
fishes:	0	9	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



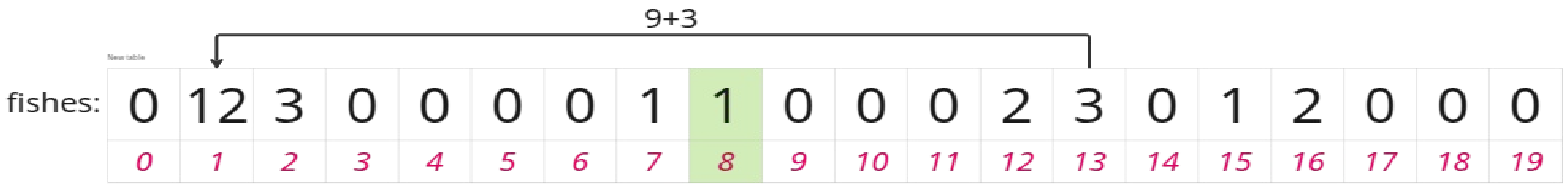
unify(8,13):

fishes[parent1 of 1]+=fishes[parent1 of 13]

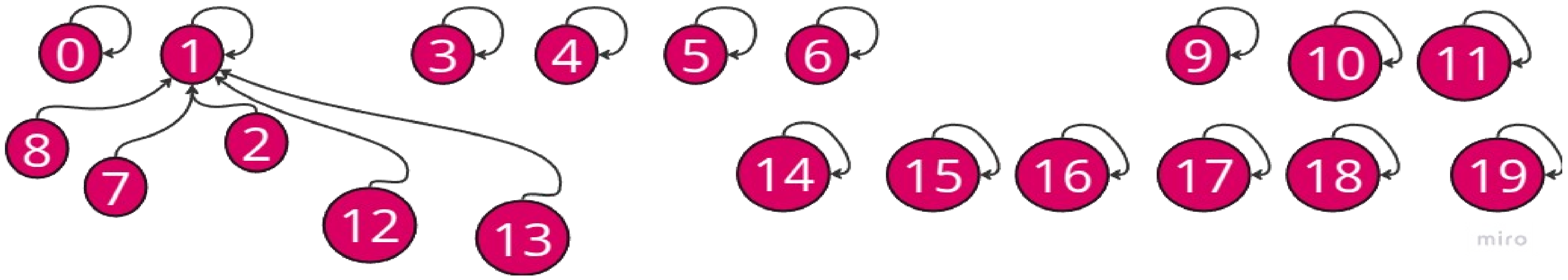
fishes[1]+=fishes[12] =>fishes[1]=7+2



	0	1	2	3	4
0	0 0	2 1	3 2	0 3	0 4
1	0 5	0 6	1 7	1 8	0 9
2	0 10	0 11	2 12	3 13	0 14
3	1 15	2 16	0 17	0 18	0 19



fishes:	0	12	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19



Process (1,4), (2,0),(2,1): their values is 0, no need to process them

✓ Process (2,2):

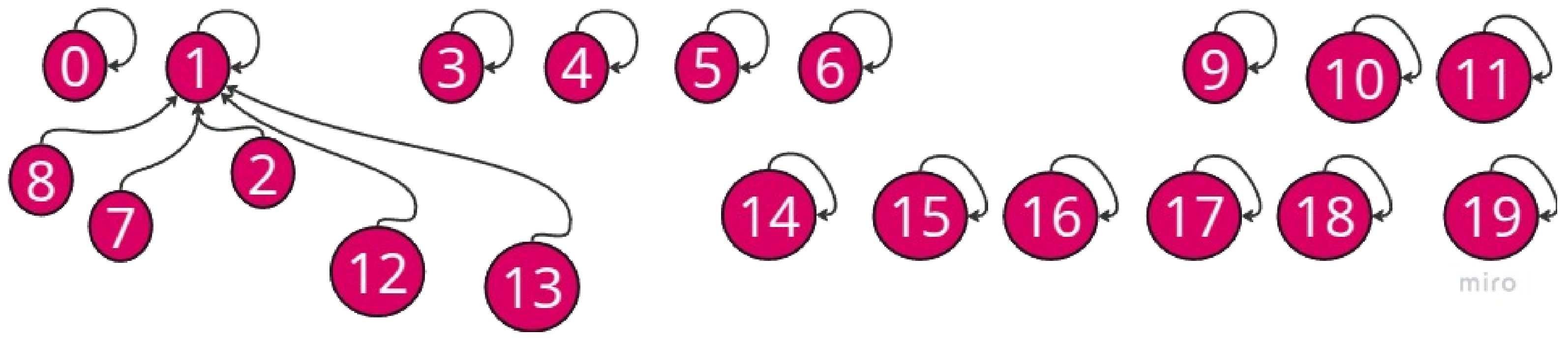
✓ Process (2,3):

12,13,7,8 are already unified

	0	1	2	3	4
0	0_{0}	2_{1}	3_{2}	0_{3}	0_{4}
1	0_{5}	0_{6}	1_{7}	1_{8}	0_{9}
2	0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
3	1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

9+3

fishes:	0	12	3	0	0	0	0	1	1	0	0	0	2	3	0	1	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



Process (3,0):

unify(15,16):

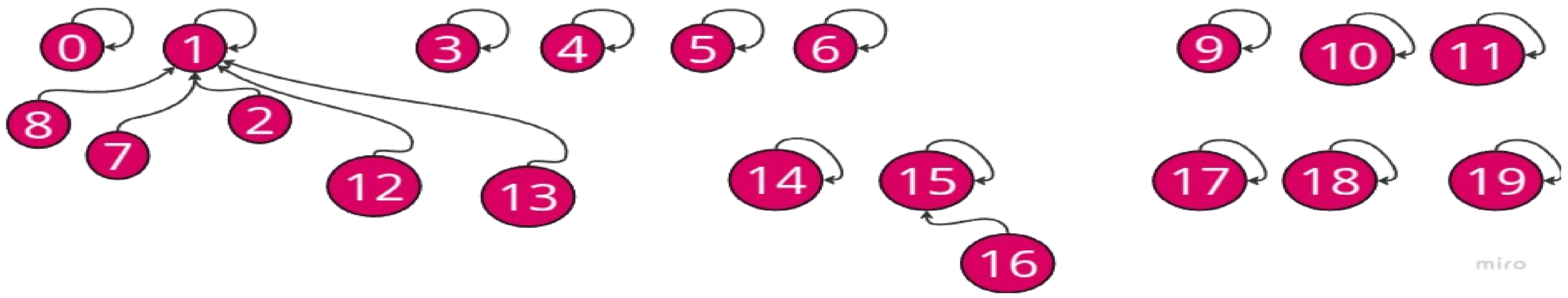
fishes[parent1 of 15]+=fishes[parent1 of 16]

fishes[15]+=fishes[16] =>fishes[15]=1+2

	0	1	2	3	4
0	0_{0}	2_{1}	3_{2}	0_{3}	0_{4}
1	0_{5}	0_{6}	1_{7}	1_{8}	0_{9}
2	0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
3	🟩 1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

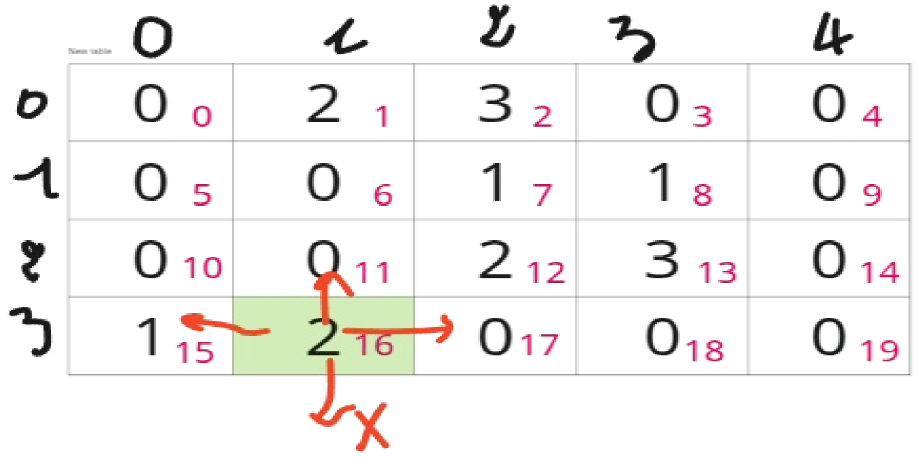
1+2

fishes:	0	12	3	0	0	0	0	1	1	0	0	0	2	3	0	3	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



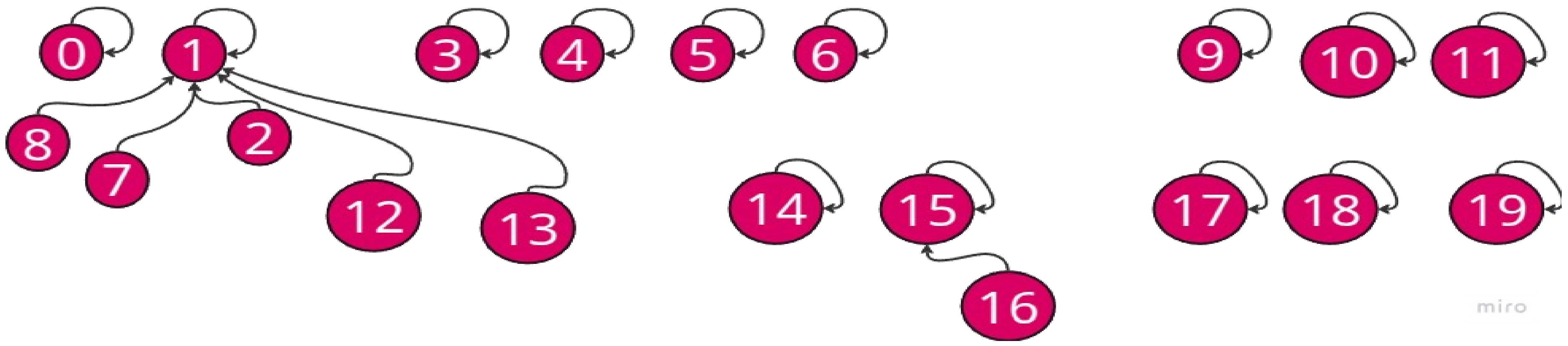
Process (3,1):

unify(16,15): already unified



	0	1	2	3	4
0	0_{0}	2_{1}	3_{2}	0_{3}	0_{4}
1	0_{5}	0_{6}	1_{7}	1_{8}	0_{9}
2	0_{10}	0_{11}	2_{12}	3_{13}	0_{14}
3	1_{15}	2_{16}	0_{17}	0_{18}	0_{19}

fishes:	0	12	3	0	0	0	0	1	1	0	0	0	2	3	0	3	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*



Process (3,2), (3,3),(3,4): their values is 0, no need to process them

answer

fishes:	0	12	3	0	0	0	0	1	1	0	0	0	2	3	0	3	2	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

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

2658. Maximum Number of Fish in a Grid

```
/*
    DSU
    Time complexity: O(mnα(mn))
    Space complexity: O(mn)
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
typedef std::pair<int,int> ii;
class DSU{
    public:
        vi parent;
        vi group;
        vi fishes;
        int _N,n,m;
        vvi grid;
    public:
        DSU(int _N,int m,int n,vvi& grid){
            this->_N=_N;
            this->m=m;
            this->n=n;
            this->grid=grid;

            parent.resize(_N);
            group.resize(_N,1);
            fishes.resize(_N,0);

            for(int i=0;i<_N;++i) {
                parent[i]=i;
                fishes[i]=grid[i/n][i%n];
            }
        }

        int find(int p){
            if(p==parent[p]) return p;
            return parent[p]=find(parent[p]);
        }
```

```
  void unify(int p,int q){
          int parent_p=find(p);
          int parent_q=find(q);
          if(parent_p==parent_q) return;

          if(group[parent_p]<group[parent_q]){
              group[parent_q]+=group[parent_p];
              fishes[parent_q]+=fishes[parent_p];
              group[parent_p]=1;
              parent[parent_p]=parent_q;

          }
          else{
              group[parent_p]+=group[parent_q];
              fishes[parent_p]+=fishes[parent_q];
              group[parent_q]=1;
              parent[parent_q]=parent_p;
          }
      }
};
```

```
class Solution {
    public:
        int cell_id(int row,int col,int n){
            return row*n+col;
        }

         bool is_in_the_grid(int row, int col, int m, int n){
            return (row >=0 && row < m && col >= 0 && col < n);
        }


        int findMaxFish(vvi& grid){
            int m=grid.size();
            int n=grid[0].size();

            DSU dsu=DSU(m*n,m,n,grid);

            for(int i=0;i<m;++i){
                for(int j=0;j<n;++j){
                    if(grid[i][j]==0) continue;
                    int cur=cell_id(i,j,n);
                    int right=cell_id(i,j+1,n);
                    int left=cell_id(i,j-1,n);
                    int up=cell_id(i-1,j,n);
                    int down=cell_id(i+1,j,n);

                    if (is_in_the_grid(i,j+1,m,n) && grid[i][j+1]) dsu.unify(cur,right);
                    if (is_in_the_grid(i,j-1,m,n) && grid[i][j-1]) dsu.unify(cur,left);
                    if (is_in_the_grid(i-1,j,m,n) && grid[i-1][j]) dsu.unify(cur,up);
                    if (is_in_the_grid(i+1,j,m,n) && grid[i+1][j]) dsu.unify(cur,down);
                }
            }

            int ans=*std::max_element(dsu.fishes.begin(),dsu.fishes.end());

            return ans;
        }
};
```

Complexity Analysis
Let $\boxed{m}$ be the number of rows and $\boxed{n}$ be the number of columns in the `grid`.

- Time Complexity: $\boxed{O(n \times m \times \alpha(n \times m))}$

  The outer loop iterates over all cells in the grid, which takes $\boxed{O(n \times m)}$ time.

  For each cell, the algorithm checks its four neighbors (right, left, down, up), which is a constant $\boxed{O(4)}$ operation.

  The `findParent` and `unionComponents` operations are performed using the Union-Find data structure with path compression and union by size. These operations have an amortized time complexity of $\boxed{O(\alpha(n \times m))}$, where $\boxed{\alpha}$ is the inverse Ackermann function, which is very small and can be considered almost constant.

  Therefore, the overall time complexity is $\boxed{O(n \times m \times \alpha(n \times m))}$.

- Space Complexity: $\boxed{O(n \times m)}$

  The algorithm uses three auxiliary arrays: `parent`, `componentSize`, and `totalFish`, each of size $\boxed{n \times m}$.

  The space required for these arrays is $\boxed{O(n \times m)}$.

  Additionally, the recursion stack for the `findParent` function is bounded by the height of the Union-Find tree, which is $\boxed{O(\alpha(n \times m))}$ due to path compression. However, this is negligible compared to the space used by the arrays.

  Therefore, the overall space complexity is $\boxed{O(n \times m)}$.