827. Making A Large Island

You are given an `n x n` binary matrix `grid`. You are allowed to change **at most one** `0` to be `1`.

Return *the size of the largest* ***island*** *in* `grid` *after applying this operation*.

An **island** is a 4-directionally connected group of `1`s.

Example 1:

```
Input: grid = [[1,0],[0,1]]
Output: 3
Explanation: Change one 0 to 1 and connect two 1s, then we get an island with area
= 3.
```

Example 2:

```
Input: grid = [[1,1],[1,0]]
Output: 4
Explanation: Change the 0 to 1 and make the island bigger, only one island with
area = 4.
```

Example 3:

```
Input: grid = [[1,1],[1,1]]
Output: 4
Explanation: Can't change any 0 to 1, only one island with area = 4.
```

Constraints:

- `n == grid.length`
- `n == grid[i].length`
- `1 <= n <= 500`
- `grid[i][j]` is either `0` or `1`.

Overview

We are given a binary matrix where each cell is either `0` (representing water) or `1` (representing land) and the ability to flip at most one `0` to `1`. Our task is to find the largest island in the matrix, or in other words, the largest group of `1`s connected with each other either up, down, left, or right (4-directionally) after the flip operation.

Brute force

At first, we might think of flipping each `0` to `1` and then calculating the size of the largest island in the modified matrix. However, this brute-force approach is inefficient, especially for larger grids, as it involves multiple recalculations for each flip, which would lead to Time Limit Exceeded (TLE) error.

827. Making A Large Island

/*

Brute force

Time complexity: $O(n^2 * n^2) = O(n^4)$ => TLE

Space complexity: $O(n^2 + n^4)$ => MLE

n: #size of the grid

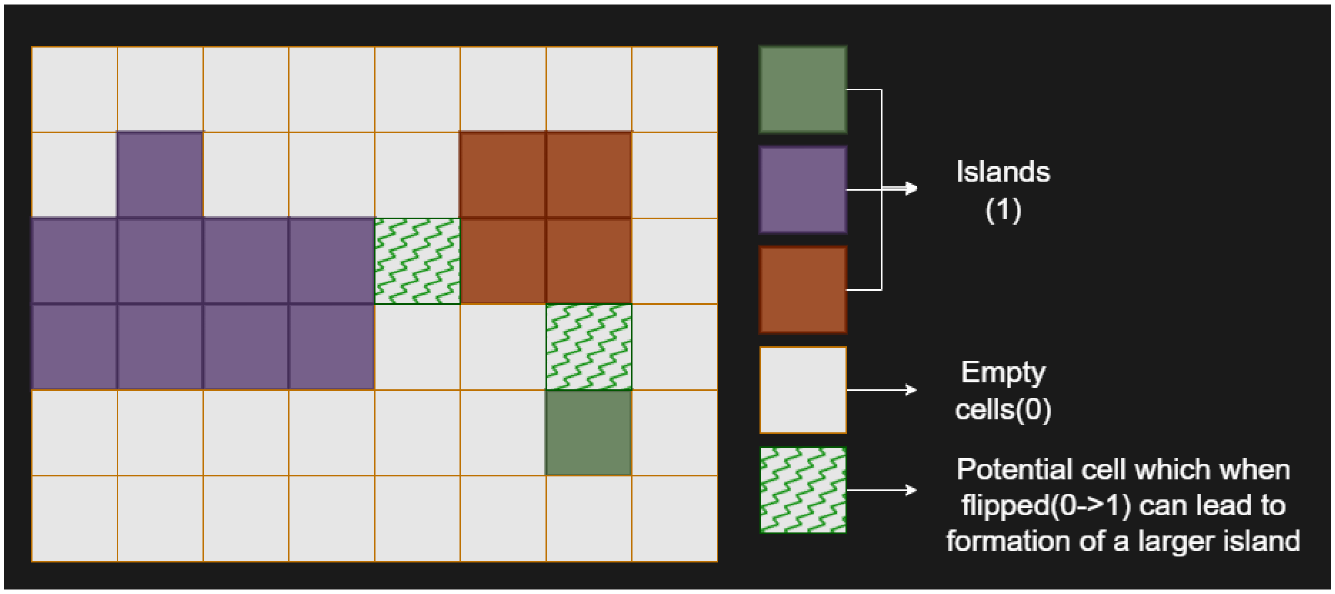
*/

```
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution {
  public:
    int largestIsland(vvi& grid) {
      int n=grid.size();
      vvi directions={{0,1},{0,-1},{-1,0},{1,0}};

      // DFS to compute the size of the island starting from the current cell
      auto dfs=[&](int cur_row,int cur_col,vvi& visited,auto& self)->int{
        if(visited[cur_row][cur_col]) return 0;
        visited[cur_row][cur_col]=1;
        int size=1;
        for(auto& dir: directions){
          int row=cur_row+dir[0];
          int col=cur_col+dir[1];
          if(row<0 || col<0 || row>n-1 || col>n-1) continue;
          if(grid[row][col]==0) continue;
          size+=self(row,col,visited,self);
        }
        return size;
      };
      // For each cell:
      int ans=0;
      for(int row=0;row<n;++row){
        for(int col=0;col<n;++col){

          // Save its original value
          int it_was=grid[row][col];

          // Make it an island
          grid[row][col]=1;

          // Explore the grid using DFS
          vvi visited(n,vi(n,0));
          ans=std::max(ans,dfs(row,col,visited,dfs));

          // Restore its original value
          grid[row][col]=it_was;

        }
      }
      return ans;
    }
};
```

Preprocess islands sizes

Instead of recalculating island sizes for every flip, we can take advantage of the fact that flipping a single `0` only affects the islands adjacent to it. Specifically, flipping a `0` merges neighboring islands into one larger island. This insight allows us to efficiently compute the largest island after flipping by precomputing the sizes of all islands first.

Check out the diagram below, where we can see that we can merge two islands into one by flipping a zero in between.


Islands
(1)
Empty
cells(0)
Potential cell which when
flipped(0->1) can lead to
formation of a larger island

We start by traversing the grid and identifying all the islands using Depth-First Search (DFS). During this traversal, we give each island a unique identifier (like a color). At the same time, we also calculate and store the size of each island in a map, where the key is the island's unique identifier and the value is its size. This precomputation allows us to avoid recalculating island sizes later.

After labeling the islands and knowing their sizes, we then look at each `0` in the grid. Flipping a `0` to `1` might connect neighboring islands, creating a larger island. For each `0`, we examine the islands around it and collect their unique identifiers using a set (to avoid counting the same island more than once). We then sum up the sizes of these islands to calculate the size of the new island that would be formed if this `0` were flipped to `1`.

As we evaluate each potential flip, we compare the size of the island that would be formed with the largest island we've seen so far. This ensures that we find the largest possible island we can form by flipping a single `0`. We will handle special edge cases (e.g., the grid is full with `1`s or `0`s) separately.

This strategy is efficient because the grid is only traversed twice:

1. To label the islands and compute their sizes.
2. To evaluate the potential island size for each `0` flip.

827. Making A Large Island

/*

Preprocess all islands+DFS

Time complexity: $O\left(n^2\right)$

Space complexity: $O\left(n^2+k\right)$

n: #size of the grid

k: #number of islands

*/

```
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution {
    private:
        vvi directions={{0,1},{0,-1},{-1,0},{1,0}};

    public:
```

```
void get_islands_size(vvi& grid,vi& islands_size){
    int n=grid.size();
    islands_size.resize(2,0);



    auto dfs=[&](int cur_row,int cur_col,int color,auto& self)->int{
        int visited=cur_row<0 || cur_col<0 || cur_row>n-1 || cur_col>n-1?0:grid[cur_row][cur_col];
        if(visited!=1) return 0;

        grid[cur_row][cur_col]=color;

        int size=1;
        for(auto& dir: directions){
            int row=cur_row+dir[0];
            int col=cur_col+dir[1];
            if(row<0 || col<0 || row>n-1 || col>n-1) continue;
            if(grid[row][col]==0) continue;
            size+=self(row,col,color,self);
        }
        return size;

    };

    for(int row=0;row<n;++row){
        for(int col=0;col<n;++col){
            if(grid[row][col]!=1) continue;
            islands_size.push_back(dfs(row,col,islands_size.size(),dfs));
        }
    }
}
```

```
int largestIsland(vvi& grid) {
        int n=grid.size();
        // Phase#1:
        //Iterate over every cell in the grid to identify and mark islands using a Depth - First
Search(DFS) approach.
        // During this process, each cell is visited at most once, ensuring that the DFS traversal
        // contributes O(n^2) to the time complexity.vi islands_size;
        get_islands_size(grid,islands_size);

        // Phase#2:  O(n^2).
        int ans=0;
        int m=islands_size.size();

        // Iterate over every cell again to explore the possibility of converting each 0 to 1 and
calculating the potential island size.
        for(int row=0;row<n;++row){
           for(int col=0;col<n;++col){
              // For each 0, we check its four neighboring cells, which is a constant-time operation.
              if(grid[row][col]!=0) continue;

              // The use of an unordered set ensures that neighboring islands are counted uniquely.
              std::unordered_set<int> ne;
              for(auto& dir: directions){
                 int r=row+dir[0];
                 int c=col+dir[1];
                 if(r<0 || c<0 || r>n-1 || c>n-1) continue;
                 ne.insert(grid[r][c]);
              }
              int new_size=1;
              for(auto& v: ne){
                 new_size+=islands_size[v];
              }
              ans=std::max(ans,new_size);
           }
        }
        // If we have an answer (ans!=0), return it
        // Otherwise, means all the grid cells values are != 0, return the whole number of cell
        return ans?ans:n*n;
     }};
```

Complexity Analysis
Let n be the number of rows in the grid, m be the number of columns in the grid.

- Time complexity: $O(n \times m)$

  The algorithm consists of two main phases. In the first phase, we iterate over every cell in the grid to identify and mark islands using a Depth-First Search (DFS) approach. During this process, each cell is visited at most once, ensuring that the DFS traversal contributes $O(n \times m)$ to the time complexity.

  In the second phase, we iterate over every cell again to explore the possibility of converting each `0` to `1` and calculating the potential island size. For each `0`, we check its four neighboring cells, which is a constant-time operation. The use of an unordered set ensures that neighboring islands are counted uniquely, and the total work done in this phase is also $O(n \times m)$.

  Thus, the overall time complexity is dominated by the grid traversal and DFS, resulting in $O(n \times m)$.

- Space complexity: $O(n \times m)$

  The space complexity is primarily determined by the recursion stack used during the DFS traversal and the storage required for the unordered map that keeps track of island sizes. In the worst case, the recursion depth of the DFS can be $O(n \times m)$ if the entire grid forms a single large island. The unordered map stores the sizes of all islands, and in the worst case, the number of islands can be proportional to the number of cells, contributing $O(n \times m)$ to the space complexity.

  Furthermore, the unordered set used to store neighboring islands for each `0` cell has a maximum size of 4, as there are only four possible neighboring cells. This does not significantly impact the overall space complexity.

  Therefore, the dominant factors are the recursion stack and the unordered map, resulting in an overall space complexity of $O(n \times m)$.