<https://leetcode.com/problems/shortest-bridge/description/>

You are given an n x n binary matrix grid where 1 represents land and 0 represents water.

An island is a 4-directionally connected group of 1's not connected to any other 1's. There are exactly two islands in grid.

You may change 0's to 1's to connect the two islands to form one island.

Return the smallest number of 0's you must flip to connect the two islands.

**Example 1:**

**Input:** grid = [[0,1],[1,0]]

**Output:** 1

**Example 2:**

**Input:** grid = [[0,1,0],[0,0,0],[0,0,1]]

**Output:** 2

**Example 3:**

**Input:** grid = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

**Output:** 1

**Constraints:**

n == grid.length == grid[i].length

2 <= n <= 100

grid[i][j] is either 0 or 1.

There are exactly two islands in grid.

**Attempt 1: 2024-12-17**

**Solution 1: DFS + BFS (60 min)**

class Solution {

    int[] dx = new int[]{0, 0, 1, -1};

    int[] dy = new int[]{1, -1, 0, 0};

    public int shortestBridge(int[][] grid) {

        int n = grid.length;

        Queue<int[]> queue = new LinkedList<>();

        // First, we find one of the islands using a DFS algorithm to look

        // for the first land cell (marked by a 1, and be aware, we only

        // need to find the first land cell, then have to block the further

        // check of land cells, because based on definition, the island is

        // connected 1's, so first 1 detect means first island found, the

        // DFS can start from this cell, the further check for land cells

        // are redundant for DFS). Once we find it, we perform a DFS starting

        // from that cell to mark all connected land cells as part of the

        // same island. For clarity, we mark these cells with a 2. At the

        // end of this step, we will have isolated one island, and the other

        // will remain marked with 1s.

        boolean foundFirstIsland = false;

        for(int i = 0; i < n && !foundFirstIsland; i++) {

            for(int j = 0; j < n; j++) {

                if(grid[i][j] == 1) {

                    foundFirstIsland = true;

                    dfs(grid, i, j, n, queue);

                    break;

                }

            }

        }

        // After marking one island, we use a BFS algorithm to expand outward

        // from the marked island. We look at all the cells at the current

        // 'frontier', and for each, we inspect their neighboring cells.

        // If we encounter a cell with a 0, we convert it into a 2 and add

        // it to the queue for the next layer of expansion. This expansion

        // simulates the process of 'building a bridge' to the other island.

        // If we encounter a 1 during this expansion, it means we have reached

        // the second island, and the current distance (or the number of 0s

        // converted to 2s) represents the minimal bridge length required to

        // connect the islands. The BFS finishes once the connection is made.

        int distance = 0;

        while(!queue.isEmpty()) {

            int size = queue.size();

            for(int i = 0; i < size; i++) {

                int[] cur = queue.poll();

                for(int k = 0; k < 4; k++) {

                    int new\_x = cur[0] + dx[k];

                    int new\_y = cur[1] + dy[k];

                        if(grid[new\_x][new\_y] == 1) {

                            return distance;

                        // Update from 0 to 2 which means expand first island

                        // boundary which also avoid using visited to check

                        } else if(grid[new\_x][new\_y] == 0) {

                            grid[new\_x][new\_y] = 2;

                            queue.offer(new int[]{new\_x, new\_y});

                        }

                    }

                }

            }

            distance++;

        }

        return distance;

    }

    private void dfs(int[][] grid, int x, int y, int n, Queue<int[]> queue) {

            // Mark the visited cell as 2

            grid[x][y] = 2;

            // Put all '2' into queue as candidate initial

            // start points for later BFS scan

            queue.offer(new int[]{x, y});

            for(int k = 0; k < 4; k++) {

                dfs(grid, x + dx[k], y + dy[k], n, queue);

            }

        }

    }

}

**Refer to**

<https://algo.monster/liteproblems/934>

**Problem Description**

The problem presents us with an n x n binary matrix grid, which contains 1s and 0s where 1 represents land and 0 represents water. There are exactly two separate areas (islands) in this grid, which are represented by connected groups of 1s. An important detail is that these connections are only in the four cardinal directions (up, down, left, right). The challenge is to determine the minimum number of 0s (water) that we must convert into 1s (land) to connect the two islands into a single landmass. To summarize, the goal is to transform the smallest possible number of 0s into 1s such that the two islands become directly connected.

**Flowchart Walkthrough**

Let's analyze Leetcode 934. Shortest Bridge using the algorithm flowchart as a guide. Here's a step-by-step walkthrough based on the Flowchart found [here](https://algo.monster/flowchart):

Is it a graph?

Yes: The 2D grid in this problem can be represented as a graph where each cell is a node, and adjacent cells (horizontally or vertically) are connected if they are land cells.

Is it a tree?

No: Although it's based on a grid structure, the multiple connections mean it's not a tree.

Is the problem related to directed acyclic graphs (DAGs)?

No: The problem involves finding the shortest bridge (minimum distance) between two islands, not about traversing directed acyclic graphs.

Is the problem related to shortest paths?

Yes: The task requires finding the shortest path between two distinct islands.

Does the problem involve connectivity?

Yes: Connectivity is key as we are trying to connect two already existing islands with the shortest path.

Is the graph weighted?

No: The distances involved in connecting two islands in the grid are uniform (simply counting steps), implying an unweighted scenario.

Conclusion: **While the flowchart suggests using BFS for this unweighted connectivity problem (as both BFS and DFS can be suited for finding shortest paths in unweighted graphs), using DFS to initially identify the two distinct islands followed by BFS to calculate the shortest bridge makes efficient use of the Depth-First Search pattern in the initial enumeration phase. DFS is optimal here for identifying and marking the two separate islands comprehensively before applying BFS for distance calculation.**

**Intuition**

The solution to this problem uses both [Depth-First Search](https://algo.monster/problems/dfs_intro) (DFS) and [Breadth-First Search](https://algo.monster/problems/bfs_intro) (BFS) algorithms. The intuition behind the solution is to treat the problem as a two-step process:

**Identify and Separate the Two Islands:** First, we find one of the islands using a DFS algorithm to look for the **first** **land** **cell** (marked by a 1, and be aware, **we only need to find the first land cell, then have to block the further check of land cells**, because based on definition, the island is connected 1's, so first 1 detect means first island found, the DFS can start from this cell, the further check for land cells are redundant for DFS). **Once we find it, we perform a DFS starting from that cell to mark all connected land cells as part of the same island. For clarity, we mark these cells with a 2.** At the end of this step, we will have isolated one island, and the other will remain marked with 1s.

**Find the Shortest Path to the Other Island:** After marking one island, we use a BFS algorithm to expand outward from the marked island. We look at all the cells at the current 'frontier', and for each, we inspect their neighboring cells. If we encounter a cell with a 0, we convert it into a 2 and add it to the queue for the next layer of expansion. This expansion simulates the process of 'building a bridge' to the other island. If we encounter a 1 during this expansion, it means we have reached the second island, and the current distance (or the number of 0s converted to 2s) represents the minimal bridge length required to connect the islands. The BFS finishes once the connection is made.

The key aspect of this approach is to expand uniformly from the entire frontier of the marked island instead of from a single point. This ensures that the shortest path will be found, as BFS guarantees the shortest path in an unweighted graph, which in this case is represented by the grid with 0s and 1s.

**Solution Approach**

The implementation of the solution can be broken down into two primary components: identifying one of the islands (isolation step) using DFS, and then finding the shortest path to the other island (connection step) using BFS.

Isolation Step (DFS): We start by initializing a variable dirs to represent the four cardinal directions for traversal. We then find a starting point, which is the first 1 encountered in the grid, indicating a part of the first island. From this initial cell, we perform a DFS. In the DFS function, we mark the current cell with 2 to distinguish it from the other unvisited land cells (1) and water (0). For every cell marked as 2, we continue the DFS for each of its unvisited land neighbors by recursively calling the DFS function. We also add the cells to a queue q, which will be used later for the BFS expansion.

Connection Step (BFS): After isolating one island and marking it with 2s, we use BFS to 'build the bridge' to the second island. We initialize a variable ans to keep track of the number of flips required. During the BFS, we dequeue cells from q and explore their neighbors in all four directions. We have three cases for each neighbor:

If the cell is water (0), we flip it to (2) to indicate it is now part of the bridge and add it to the queue for further exploration.

If the cell is a part of the unvisited island (1), we have reached the second island, and we return the ans variable because we found the minimal number of flips to connect the two islands.

If the cell is already included in the bridge or is a part of the isolated island (2), we ignore it and continue.

The BFS continues in this manner, layer by layer, in a while loop. Each loop iteration corresponds to expanding the 'frontier' by one more step. This ensures that when we do encounter a cell belonging to the second island, we have taken the minimal number of steps to get there. When encounter occurs, the value of ans represents the smallest number of 0s that we have flipped, and this value is returned as the final answer.

In summary, the DFS is used to isolate and mark one island, and the BFS is used to calculate the minimal distance to the other island by simulating the process of building a bridge one cell at a time.

**Example Walkthrough**

Let's consider a small 5x5 binary matrix grid as an example to walk through the solution approach.

Grid:

0 1 0 0 0

0 1 0 0 0

0 0 0 1 1

0 0 0 0 0

0 0 0 0 1

Let's apply both DFS and BFS algorithms to solve it following the steps described above.

**Step 1: Isolation Step (DFS)**

Begin the DFS by scanning for "1" from the top-left corner. The first "1" is encountered at grid[0][1].

Initiate DFS from this cell, marking all connected "1s" as "2s", so that cell's part of the first island are differentiated.

Updated grid after DFS:

0 2 0 0 0

0 2 0 0 0

0 0 0 1 1

0 0 0 0 0

0 0 0 0 1

Now, all the cells from the first island are marked as 2, and we create a queue q with these cells, which will be (0,1) and (1,1).

**Step 2: Connection Step (BFS)**

Begin a BFS starting from the queue of the first island's perimeter—the 2s.

Start expanding layer by layer. Initially, ans is set to 0, representing the number of flips (or bridge's length) required so far.

First iteration of BFS (adding surrounding 0s of starting island to the queue):

0 2 2 0 0

0 2 2 0 0

0 0 0 1 1

0 0 0 0 0

0 0 0 0 1

ans = 1 (since we transformed one '0' to '2')

Second iteration of BFS (continuing to expand):

0 2 2 2 0

0 2 2 2 0

0 0 0 1 1

0 0 0 0 0

0 0 0 0 1

ans = 2

Third iteration of BFS (the frontier now touches the second island):

0 2 2 2 0

0 2 2 2 0

0 0 0 1 1

0 0 0 0 0

0 0 0 0 1

ans = 2

The BFS stops here as the expanding cells meet the second island, which is still marked with 1s. The minimum number of flips is 2, as that's the BFS depth when the connection with the second island was made.

So the minimum number of 0s that must be converted to 1s to connect the two islands in this example is 2.

**Solution Implementation**

class Solution {

// Movement directions (up, right, down, left)

private int[] directions = {-1, 0, 1, 0, -1};

// Queue to keep track of the expansion of the island

private Deque<int[]> queue = new ArrayDeque<>();

// The input grid representation of islands

private int[][] grid;

// Size of the grid

private int size;

public int shortestBridge(int[][] grid) {

this.grid = grid;

size = grid.length;

// Find the first island and perform DFS to mark it

boolean foundFirstIsland = false;

for (int i = 0; i < size && !foundFirstIsland; ++i) {

for (int j = 0; j < size; ++j) {

if (grid[i][j] == 1) {

performDFS(i, j);

foundFirstIsland = true;

break; // Break out of the loops after finding the first island

}

}

}

// BFS to find the shortest path to the second island

int steps = 0;

while (true) {

// Iterate over all points added in the last expansion

for (int i = queue.size(); i > 0; --i) {

int[] point = queue.pollFirst();

// Explore all possible directions from current point

for (int k = 0; k < 4; ++k) {

int x = point[0] + directions[k];

int y = point[1] + directions[k + 1];

// Check if the next point is within the grid boundaries

// If we reach the second island, return the number of steps

if (grid[x][y] == 1) {

return steps;

}

// If water is found, mark it as visited and add to the queue for further exploration

if (grid[x][y] == 0) {

grid[x][y] = 2;

queue.offer(new int[] {x, y});

}

}

}

}

++steps; // Increment steps after each level of BFS expansion

}

// Should not reach here

return -1;

}

// DFS to mark all the squares of the first island

private void performDFS(int i, int j) {

grid[i][j] = 2; // Mark as visited

queue.offer(new int[] {i, j}); // Add to queue for BFS later

// Explore all directions from the current square

for (int k = 0; k < 4; ++k) {

int x = i + directions[k];

int y = j + directions[k + 1];

// If the next square is part of the island, continue DFS

performDFS(x, y);

}

}

}

}

**Time and Space Complexity**

The time complexity of the given code can be considered in two major parts: the Depth First Search (DFS) to find the first island and paint it to a different color (in this case 2), and then performing a Breadth First Search (BFS) to find the shortest path to the second island.

DFS Complexity: The DFS runs at most once for every cell in the grid (n \* n where n is the length of a side of the grid). In the worst case, the entire grid is one island and we visit every cell once. Therefore, the time complexity of the DFS is O(n^2).

BFS Complexity: For BFS, in the worst case, we might have to go through all cells in the grid again to find the shortest path to the second island. The BFS visits each cell at most once, so this is also O(n^2).

pairwise Function: The pairwise function on dirs can be seen as O(1) since dirs has a constant size of 5 elements regardless of the input size n. Thus, the loops utilizing pairwise(dirs) do not significantly affect the asymptotic time complexity.

Thus, combining DFS and BFS gives us a total time complexity of O(n^2) as both operations are bound by the size of the grid and do not exceed it.

For space complexity:

The queue q might hold all cells in the worst case (if they all become part of the BFS queue), hence it requires up to O(n^2) space.

The recursive stack for DFS could, in the worst-case scenario (one large island), also take up to O(n^2) space depending on the implementation of the system stack (since we might have to go as deep as the number of cells in the grid if they are all connected in a straight line).

Therefore, the overall space complexity of the algorithm is also O(n^2) due to the usage of the BFS queue and the system stack for recursion in DFS.

**Refer to chatGPT**

**Key Steps in the Solution**

Identify the Two Islands:

Use DFS to find all cells belonging to one island and mark them with a unique identifier.

Breadth-First Search (BFS):

Use BFS starting from the first island to find the shortest distance to the second island. As BFS expands layer by layer, the first time we encounter a cell in the second island gives the shortest path.

Optimization:

Instead of marking the second island entirely, we stop BFS when we reach any cell belonging to the second island.

class Solution {

public int shortestBridge(int[][] grid) {

int n = grid.length;

Queue<int[]> queue = new LinkedList<>();

boolean found = false;

// Step 1: Use DFS to find the first island and mark it

for (int i = 0; i < n; i++) {

if (found) break;

for (int j = 0; j < n; j++) {

if (grid[i][j] == 1) {

dfs(grid, i, j, queue);

found = true;

break;

}

}

}

// Step 2: Use BFS to expand from the first island and find the shortest path to the second island

int steps = 0;

int[] dx = {0, 0, 1, -1};

int[] dy = {1, -1, 0, 0};

while (!queue.isEmpty()) {

int size = queue.size();

for (int i = 0; i < size; i++) {

int[] cell = queue.poll();

int x = cell[0], y = cell[1];

for (int k = 0; k < 4; k++) {

int nx = x + dx[k];

int ny = y + dy[k];

// Check boundaries

if (grid[nx][ny] == 1) {

return steps; // Reached the second island

} else if (grid[nx][ny] == 0) {

grid[nx][ny] = -1; // Mark as visited

queue.offer(new int[]{nx, ny});

}

}

}

}

steps++;

}

return -1; // Should not reach here

}

// Helper method for DFS to mark the first island

private void dfs(int[][] grid, int x, int y, Queue<int[]> queue) {

int n = grid.length;

if (x < 0 || x >= n || y < 0 || y >= n || grid[x][y] != 1) {

return;

}

grid[x][y] = -1; // Mark the cell as visited

queue.offer(new int[]{x, y}); // Add the cell to the BFS queue

// Explore in all 4 directions

dfs(grid, x + 1, y, queue);

dfs(grid, x - 1, y, queue);

dfs(grid, x, y + 1, queue);

dfs(grid, x, y - 1, queue);

}

}

**Explanation of the Code**

DFS:

Finds the first island and marks all its cells as -1.

Adds all marked cells to the BFS queue for further processing.

BFS:

Expands layer by layer from the first island.

Stops and returns the current step count as soon as a cell from the second island is reached.

**Complexity Analysis**

Time Complexity:

DFS: O(N²), where N is the grid size. This marks all cells of the first island.

BFS: O(N²), as we potentially explore all cells in the grid to reach the second island.

Overall: **O(N²)**.

Space Complexity:

BFS queue: O(N²) in the worst case.

Overall: **O(N²)**.

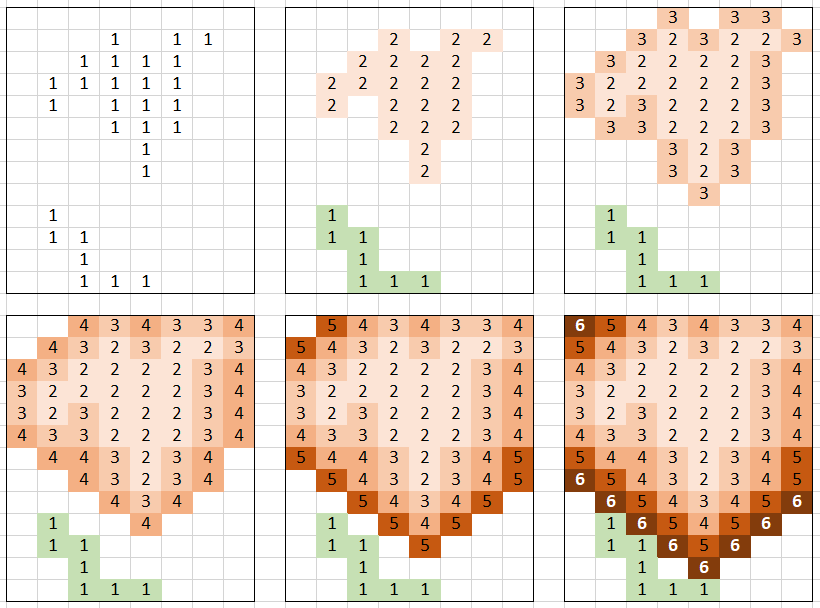
**Refer to**

<https://leetcode.com/problems/shortest-bridge/solutions/189293/c-bfs-island-expansion-uf-bonus/>

We first paint one of the islands using DFS with color 2, so we can easily identify island #1 and island #2. Thanks [@davidluoyes](https://leetcode.com/davidluoyes/) for pointing out that we only need to paint one island.

Then we start expanding island #2 by paining connected empty area. Each round, we increase the color (3, 4, and so on) so we can keep track of the newly painted area. This ends when we "bump" into the first island.

The fact that we are increasing the color is also useful for the backtracking, if we need to return the coordinates of the bridge.**C++**



int paint(vector<vector<int>>& A, int i, int j) {

if (min(i, j) < 0 || max(i, j) == A.size() || A[i][j] != 1)

return 0;

A[i][j] = 2;

return 1 + paint(A, i + 1, j) + paint(A, i - 1, j) + paint(A, i, j + 1) + paint(A, i, j - 1);

}

bool expand(vector<vector<int>>& A, int i, int j, int cl) {

if (min(i, j) < 0 || max(i, j) == A.size())

return false;

A[i][j] = A[i][j] == 0 ? cl + 1 : A[i][j];

return A[i][j] == 1;

}

int shortestBridge(vector<vector<int>>& A) {

for (int i = 0, found = 0; !found && i < A.size(); ++i)

for (int j = 0; !found && j < A[0].size(); ++j)

found = paint(A, i, j);

for (int cl = 2; ; ++cl)

for (int i = 0; i < A.size(); ++i)

for (int j = 0; j < A.size(); ++j)

if (A[i][j] == cl && ((expand(A, i - 1, j, cl) || expand(A, i, j - 1, cl) ||

expand(A, i + 1, j, cl) || expand(A, i, j + 1, cl))))

return cl - 2;

}

Same idea, but using a queue instead of scaning the grid.

int dir[5] = {0, 1, 0, -1, 0};

if (min(i, j) >= 0 && max(i, j) < A.size() && A[i][j] == 1) {

A[i][j] = 2;

q.push\_back({i, j});

for (int d = 0; d < 4; ++d)

paint(A, i + dir[d], j + dir[d + 1], q);

}

}

int shortestBridge(vector<vector<int>>& A) {

vector<pair<int, int>> q;

for (int i = 0; q.size() == 0 && i < A.size(); ++i)

for (int j = 0; q.size() == 0 && j < A[0].size(); ++j)

paint(A, i, j, q);

while (!q.empty()) {

vector<pair<int, int>> q1;

for (auto [i, j] : q) {

for (int d = 0; d < 4; ++d) {

int x = i + dir[d], y = j + dir[d + 1];

if (min(x, y) >= 0 && max(x, y) < A.size()) {

if (A[x][y] == 1)

return A[i][j] - 2;

if (A[x][y] == 0) {

A[x][y] = A[i][j] + 1;

q1.push\_back({x, y});

}

}

}

}

swap(q, q1);

}

return 0;

}

As a bonus, below is the union-find based solution. We convert the map into the graph, and join connected "land" cells into sub-graphs (island #1 and island #2). Then, we expand both of these islands by adding directly connected "water" cells to the sub-graphs. Finally, we stop when we detect that two sub-graphs are about to merge into one.

Interestingly, the runtime of these two very different solutions are similar (24 - 35 ms), and the UF solution is much more complex. Probably, the UF solution can be further optimized by only processing direct connections...

int uf\_find(int i, vector<int>& nodes) {

if (nodes[i] <= 0) return i;

else return nodes[i] = uf\_find(nodes[i], nodes);

}

int uf\_union(int i, int j, vector<int>& nodes) {

auto pi = uf\_find(i, nodes), pj = uf\_find(j, nodes);

if (pi == pj) return 0;

if (nodes[pi] > nodes[pj]) swap(pi, pj);

nodes[pi] += min(-1, nodes[pj]);

nodes[pj] = pi;

return -nodes[pi];

}

int shortestBridge(vector<vector<int>> &A) {

int sz = A.size();

vector<int> nodes(sz \* sz + 1);

list<pair<int, int>> edges;

for (auto i = 0; i < sz; ++i)

for (auto j = 0; j < sz; ++j) {

auto idx = i \* sz + j + 1;

if (A[i][j]) nodes[idx] = -1;

if (j > 0) {

if (A[i][j] && A[i][j - 1]) uf\_union(idx - 1, idx, nodes);

else edges.push\_back({ idx - 1, idx });

}

if (i > 0) {

if (A[i][j] && A[i - 1][j]) uf\_union(idx - sz, idx, nodes);

else edges.push\_back({ idx - sz, idx });

}

}

for (auto step = 1; ; ++step) {

vector<pair<int, int>> merge\_list;

for (auto it = edges.begin(); it != edges.end(); ) {

if (nodes[it->first] == 0 && nodes[it->second] == 0) ++it;

else {

if (nodes[it->first] != 0 && nodes[it->second] != 0) {

if (uf\_find(it->first, nodes) != uf\_find(it->second, nodes)) return (step - 1) \* 2;

}

merge\_list.push\_back({ it->first, it->second });

edges.erase(it++);

}

}

for (auto p : merge\_list) {

if (nodes[p.first] != 0 && nodes[p.second] != 0) {

if (uf\_find(p.first, nodes) != uf\_find(p.second, nodes)) return step \* 2 - 1;

}

uf\_union(p.first, p.second, nodes);

}

}

}

**Refer to**

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid (Ref.L743,L2290)](note://WEB4b3aa9003dbf5915537540decf09b434)