<https://leetcode.com/problems/minimum-cost-to-make-at-least-one-valid-path-in-a-grid/description/>

Given an m x n grid. Each cell of the grid has a sign pointing to the next cell you should visit if you are currently in this cell. The sign of grid[i][j] can be:

1 which means go to the cell to the right. (i.e go from grid[i][j] to grid[i][j + 1])

2 which means go to the cell to the left. (i.e go from grid[i][j] to grid[i][j - 1])

3 which means go to the lower cell. (i.e go from grid[i][j] to grid[i + 1][j])

4 which means go to the upper cell. (i.e go from grid[i][j] to grid[i - 1][j])

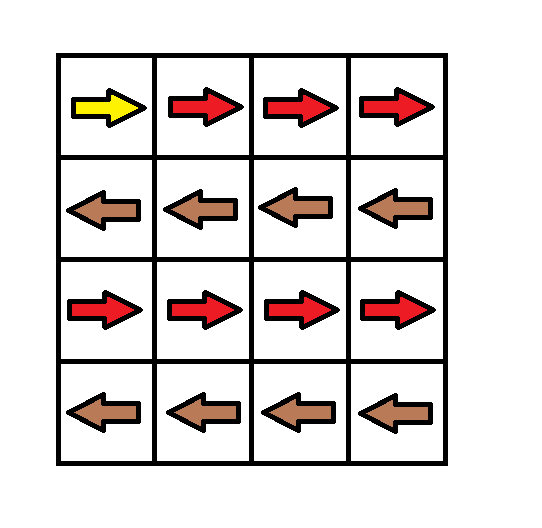
Notice that there could be some signs on the cells of the grid that point outside the grid.

You will initially start at the upper left cell (0, 0). A valid path in the grid is a path that starts from the upper left cell (0, 0) and ends at the bottom-right cell (m - 1, n - 1) following the signs on the grid. The valid path does not have to be the shortest.

You can modify the sign on a cell with cost = 1. You can modify the sign on a cell **one time only**.

Return the minimum cost to make the grid have at least one valid path.

**Example 1:**



**Input:** grid = [[1,1,1,1],[2,2,2,2],[1,1,1,1],[2,2,2,2]]

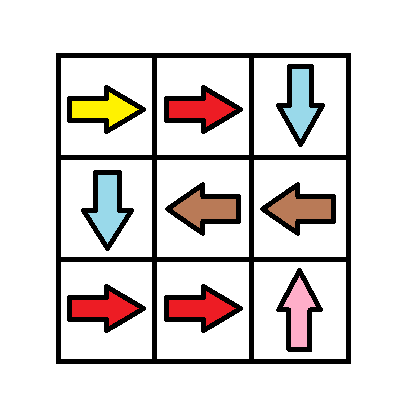
**Output:** 3

**Explanation:** You will start at point (0, 0).

The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) --> (0, 3) change the arrow to down with cost = 1 --> (1, 3) --> (1, 2) --> (1, 1) --> (1, 0) change the arrow to down with cost = 1 --> (2, 0) --> (2, 1) --> (2, 2) --> (2, 3) change the arrow to down with cost = 1 --> (3, 3)

The total cost = 3.

**Example 2:**

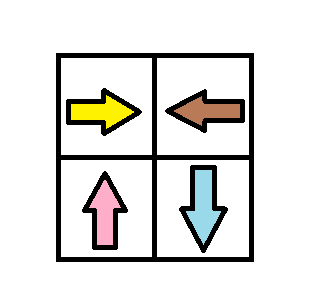


**Input:** grid = [[1,1,3],[3,2,2],[1,1,4]]

**Output:** 0

**Explanation:** You can follow the path from (0, 0) to (2, 2).

**Example 3:**



**Input:** grid = [[1,2],[4,3]]

**Output:** 1

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 100

1 <= grid[i][j] <= 4

**Attempt 1: 2024-12-16**

**Solution 1: 0-1 BFS (360 min)**

class Solution {

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

        int m = grid.length;

        int n = grid[0].length;

        boolean[][] visited = new boolean[m][n];

        // To match the sequence of go direction:

        // go right, go left, go lower, go upper

        // the additional pair {0, 0} ahead used

        // as a padding to match sign 1 to 4

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

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

        // Deque for performing 0-1 BFS

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

        // Starting by adding the [0, 0] cell with 0 cost

        deque.offer(new int[]{0, 0, 0});

        while(!deque.isEmpty()) {

            int[] cur = deque.poll();

            int x = cur[0];

            int y = cur[1];

            int cost = cur[2];

            // If we've reached the bottom-right cell, return the cost

            if(x == m - 1 && y == n - 1) {

                return cost;

            }

            // If this cell is already visited, skip it

            if(visited[x][y]) {

                continue;

            }

            visited[x][y] = true;

            // Since dx, dy has a padding {0, 0}, we can directly use dx[k] and dy[k]

            for(int k = 1; k <= 4; k++) {

                int new\_x = x + dx[k];

                int new\_y = y + dy[k];

                    // If the current direction is the same as the arrow in

                    // this cell (no cost to move here), put at head of deque

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

                        deque.offerFirst(new int[]{new\_x, new\_y, cost});

                    // Otherwise, add the new cell at the tail of the queue

                    // and increase the cost by 1

                    } else {

                        deque.offer(new int[]{new\_x, new\_y, cost + 1});

                    }

                }

            }

        }

        // If the deque is empty and we didn't reach the bottom-right cell,

        // return -1 as it's not possible

        return -1;

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Refer to**

<https://algo.monster/liteproblems/1368>

**Problem Description**

The task is to find the minimum cost to establish at least one valid path in an m x n grid. This path must start at the top left cell (0, 0) and end at the bottom right cell (m - 1, n - 1). Each cell in the grid contains a sign which directs you to the next cell to visit. The possible signs are:

1: Move right to the cell (i, j + 1)

2: Move left to the cell (i, j - 1)

3: Move down to the cell (i + 1, j)

4: Move up to the cell (i - 1, j)

Some signs might point outside of the grid boundaries, which are to be considered invalid directions for the purpose of a valid path. The cost to change a sign in any cell is 1, and each sign can be changed only once.

The objective is to determine the minimum cost to alter the signs in such a way that at least one valid path from the top left to the bottom right cell exists.

**Flowchart Walkthrough**

To determine the appropriate algorithm using the [Flowchart](https://algo.monster/flowchart) for solving LeetCode 1368. Minimum Cost to Make at Least One Valid Path in a Grid, we follow these steps:

Is it a graph?

Yes: The grid can be interpreted as a graph where each cell is a node and edges connect to the adjacent cells depending on the direction stored in each cell.

Is it a tree?

No: The grid can contain cycles depending on the cell directions; it's not necessarily a hierarchical structure.

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

No: Although the directions create a directed graph, it's not acyclic because one can potentially return to a previous cell based on the directions and changes made.

Is the problem related to shortest paths?

Yes: The problem specifically asks about the "minimum cost" to ensure at least one valid path is available, framing it as a shortest path problem in terms of cost minimization.

Is the graph weighted?

Yes: The cost to change directions in a cell acts like a weight in the graph.

Using the flowchart, here's how you can deduce the use of the Breadth-First Search pattern:

Now we have to answer **why this problem can use both Djikstra and 0-1 BFS ?**

**Given that we identified the problem as a shortest path issue in a weighted graph, the next node in the decision tree is to choose between Dijkstra's Algorithm and BFS. Typically, BFS is applied to unweighted graphs, but since Dijkstra's algorithm can degrade to a form similar to BFS when dealing with uniform costs (where cells can either have a cost of 0 or 1 as moving to some adjacent cells may or may not incur a cost), BFS can be optimized to handle such scenarios using a deque to manage nodes based on their costs. This queue method is optimized for grids or graphs where transitions might be costless (weight of 0) in some cases (moving in the original cell direction) and have a cost (weight of 1) in other cases (changing direction).**

Conclusion: Based on the flowchart, BFS can be applied in an optimized manner using a priority approach (like 0-1 BFS) for the weighted shortest path problem in this non-standard grid configuration.

**Intuition**

To solve this problem, we use a [breadth-first search](https://algo.monster/problems/bfs_intro) (BFS) approach, but with a slight tweak. This problem can be thought of as traversing a [graph](https://algo.monster/problems/graph_intro) where each cell represents a node, and the signs are the directed edges to the neighboring nodes. The challenge, however, lies in the fact that these directed edges can be altered at a cost.

The modified BFS algorithm (0 - 1 BFS) uses a double-ended queue (deque) to keep track of the cells to be visited. This is crucial as the queue can have elements added to both its front and back, which helps in maintaining the order of traversal based on the cost associated with moving to a particular cell.

Here's the intuition behind the BFS traversal:

We begin at the starting cell (0, 0) with an initial cost of 0.

As we visit a cell, we look at all possible directions we could move from that cell.

If the direction aligns with the arrow currently in the cell, we can move to that cell at no additional cost. In this case, we add this cell at the front of the deque to prioritize it.

If the direction doesn't align with the arrow, we would need to change the sign with a cost of 1. For these cells, we add them to the back of the deque as they represent potential paths but at a higher cost.

Each cell is visited only once to ensure minimal traversal cost, thus, we maintain a visited set.

This process continues until we reach the destination cell (m - 1, n - 1) or there are no more cells left to visit in the deque.

The cost associated with the first visit to the destination cell is the minimum cost needed to make at least one valid path, which is what we return.

By always choosing to traverse in the indicated direction without any cost, we ensure that we are taking advantage of the free moves as much as possible before incurring any additional costs. The visited set prevents us from revisiting cells and possibly entering a loop.

**Solution Approach**

The provided reference code implements a BFS strategy using a deque. Here's a step-by-step breakdown of the solution's implementation:

Initialize the dimensions m (rows) and n (columns) of the grid.

Define an array dirs to represent the possible directions of travel based on the grid's signs: [[0, 0], [0, 1], [0, -1], [1, 0], [-1, 0]]. Each sub-array corresponds to the deltas for the row and column indices when moving in each of the four directions.

Create a double-ended queue q, initialized with a tuple containing the row index 0, column index 0, and the initial cost 0.

Create a set vis to keep track of visited cells and prevent revisiting them.

Start a loop to continue processing until the deque q is empty. Each iteration will handle one cell's visit:

Dequeue an element (i, j, d) from the front of q, where i and j are the current cell's indices, and d is the cost to reach this cell.

If the cell (i, j) has already been visited, skip processing it to avoid loops and unnecessary cost increments.

Mark the current cell (i, j) as visited by adding it to vis.

If the cell (i, j) is the destination (m-1, n-1), return the cost d because a valid path has been found.

For each possible direction k (1 to 4), calculate the indices of the adjacent cell (x, y) by adding the directional deltas to the current indices i and j.

Check if the new cell (x, y) is within the bounds of the grid:

**If the sign at the current cell grid[i][j] indicates the direction k, meaning no sign change is needed, add (x, y, d) to the front of q, not increasing the cost, as it's a free move in the desired direction.**

**If the sign does not match, add (x, y, d + 1) to the back of q, increasing the cost by 1 to account for the change of sign.**

If no valid path is found by the end of the traversal, return -1.

**The use of a deque allows for efficient addition of cells to either the front or back, depending on whether a cost is incurred. By prioritizing the moves that don't require sign changes (zero cost), the algorithm ensures the minimum cost path is found first.** The set vis prevents revisiting and recalculating paths for positions that have already been evaluated, which optimizes the search. This approach effectively treats the grid as a [graph](https://algo.monster/problems/graph_intro) and considers the costs of edges dynamically, resulting in a clever application of the BFS algorithm adapted for weighted pathfinding.

**Example Walkthrough**

Let's assume we have a 3x3 grid, where the signs in the cells are arranged as follows:

1 3 4

4 2 1

3 1 3

Now let's walk through the solution approach with this grid:

We start at the top-left cell (0, 0) with an initial cost of 0.

From here, the sign 1 tells us to move right to cell (0, 1). This is a valid move with no cost so we add (0, 1) to the front of our deque.

Now we consider the cell (0, 1) with the sign 3, which tells us to move down. The target cell (1, 1) has not been visited, so we add (1, 1) to the front of the deque again without any additional cost.

At cell (1, 1), the sign 2 indicates moving left to (1, 0). This cell has not been visited, so we add (1, 0) to the front of the deque.

Next, we visit cell (1, 0) which contains the sign 4, telling us to move up to (0, 0). However, this has been visited, so we don't add anything to our deque.

Our next cell would be (0, 1) again, but since it's visited, we ignore it.

Continuing this process, we encounter cell (1, 1) again, which we've visited, so we move on.

Now, we get to cell (1, 0) and from here, the direction 4 (up) is not valid since it would take us outside the grid or into visited cells. At this point, we need to consider changing the direction, so we will add the neighboring cells (1, 1) to the right and (2, 0) down to the back of the deque with an added cost of 1.

Cell (1, 1) won't be processed as it's already visited, so we look at cell (2, 0) with the sign 3 which leads us down to (2, 1). There's no cost for moving down since the sign matches. We add (2, 1) to the front of the deque.

From cell (2, 1), we move right to (2, 2) as indicated by the 1 sign. As this is a free move, it is added to the front of the deque.

Finally, we arrive at the bottom right cell (2, 2) and our destination is reached without needing further sign changes. The cost at this point is 1.

The minimum cost required to establish at least one valid path in this grid is 1, which entails changing one sign. By strategically using a deque and a visited set, we conducted a breadth-first search that prioritized no-cost moves over those requiring a sign change.

**Solution Implementation**

class Solution {

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

// m holds the number of rows in the grid.

int numRows = grid.length;

// n holds the number of columns in the grid.

int numCols = grid[0].length;

// vis holds information whether a cell has been visited.

boolean[][] visited = new boolean[numRows][numCols];

// Queue for performing BFS with modifications for 0-cost moves.

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

// Starting by adding the top-left cell with 0 cost.

queue.offer(new int[] {0, 0, 0});

// dirs are used to navigate throughout the grid. (right, left, down, up)

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

// BFS starts here

while (!queue.isEmpty()) {

// Dequeue a cell info from the queue.

int[] position = queue.poll();

// i and j hold the current cell row and column, d holds the current cost.

int i = position[0], j = position[1], cost = position[2];

// If we've reached the bottom-right cell, return the cost.

if (i == numRows - 1 && j == numCols - 1) {

return cost;

}

// If this cell is already visited, skip it.

if (visited[i][j]) {

continue;

}

// Mark the cell as visited.

visited[i][j] = true;

// Explore all possible directions from the current cell.

for (int k = 1; k <= 4; ++k) {

int newX = i + directions[k][0], newY = j + directions[k][1];

// Check the validity of the new cell coordinates.

// If the current direction is the same as the arrow in this cell (no cost to move here).

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

// Add the new cell at the front of the queue to explore it sooner (as it's no cost).

queue.offerFirst(new int[] {newX, newY, cost});

} else {

// Otherwise, add the new cell at the end of the queue and increase the cost by 1.

queue.offer(new int[] {newX, newY, cost + 1});

}

}

}

}

// If the queue is empty and we didn't reach the bottom-right cell, return -1 as it's not possible.

return -1;

}

}

**Time and Space Complexity**

The provided code solves the problem using a Breadth First Search (BFS) algorithm with a slight optimization. The algorithm has two modes of queue operations: normally numbers are appended to the end of the queue, but when moving in the direction the grid arrow points to, it's added to the front. This has the potential to reduce the number of steps needed to reach the end.

**Time Complexity:**

The time complexity is O(m \* n), where m is the number of rows and n is the number of columns in the grid. This is because in the worst-case scenario, each cell is visited only once due to the use of the visited (vis) set. Even though there is a nested loop to iterate over directions, they only add constant work at each node, so it does not affect the overall linear complexity with respect to the number of cells.

**Space Complexity:**

The space complexity is also O(m \* n) because of the visited set (vis) storing up to m \* n unique cell positions to ensure cells are not revisited. Additionally, the queue (q) in the worst case may also contain elements from all the cells in the grid when they are being processed sequentially. Hence, the overall space complexity remains O(m \* n).

**Solution 2: Djikstra (60 min, refer to** [L743.Network Delay Time (Ref.L505,L1368)](note://17E4C21F015F4605AAD4D6878F0D0A67)**)**

class Solution {

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

        int m = grid.length;

        int n = grid[0].length;

        // To match the sequence of go direction:

        // go right, go left, go lower, go upper

        // the additional pair {0, 0} ahead used

        // as a padding to match sign 1 to 4

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

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

        // Dijkstra algorithm requires minimum heap, sort by cost

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> (a[2] - b[2]));

        // Dijkstra algorithm initialize with all cells distance as max value

        // except the start cell [0, 0] as 0

        int[][] distances = new int[m][n];

        for(int i = 0; i < m; i++) {

            Arrays.fill(distances[i], Integer.MAX\_VALUE);

        }

        distances[0][0] = 0;

        minPQ.offer(new int[]{0, 0, 0});

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int cost = cur[2];

            // If we've reached the bottom-right cell, return the cost

            if(x == m - 1 && y == n - 1) {

                return cost;

            }

            // Add below statement improve Elapsed Time from 21ms to 20ms

            // Skip if encounter same cell again and cell's distance is outdated:

            // Once a cell is processed earlier, no shorter distance can be found

            // for it due to the PriorityQueue. The distances[][] array acts as a

            // safeguard: it prevents processing outdated or longer paths,

            // eliminating the need for a visited array. If a cell is encountered

            // again in the PriorityQueue, its distance will not be processed

            // since this cell's guaranteed smallest distance result (based on

            // Minimum PriorityQueue natruality) stored at distances[x][y]

            // earlier, ensured by the condition below:

            // "if (cost > distances[x][y]) continue;"

            // Additionally, if not add this statement, the Dijkstra algorithm

            // still works, only won't terminate early when encountering same cell

            // again, and same cell will be added onto PriorityQueue again and

            // all further process blocked later with condition below:

            // "if (newCost < distances[new\_x][new\_y]) {...}"

            // since the newCost is surely >= distances[new\_x][new\_y]

            // at a certain moment

            if(cost > distances[x][y]) {

                continue;

            }

            // Since dx, dy has a padding {0, 0}, we can directly use dx[k] and dy[k]

            for(int k = 1; k <= 4; k++) {

                int new\_x = x + dx[k];

                int new\_y = y + dy[k];

                    // Change direction lead to 1 more cost

                    int new\_cost = cost + (grid[x][y] == k ? 0 : 1);

                    // Dijkstra algorithm only update the distance if

                    // a shorter path is found till current cell

                    if(new\_cost < distances[new\_x][new\_y]) {

                        distances[new\_x][new\_y] = new\_cost;

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

                    }

                }

            }

        }

        // If the Minimum PriorityQueue is empty and we didn't reach

        // the bottom-right cell, return -1 as it's not possible

        return -1;

    }

}

Time Complexity:

Each cell is processed once in the priority queue.

For each cell, up to 4 neighbors are checked.

Total complexity: O(m \* n \* log(m \* n)), where m and n are the grid dimensions.

Space Complexity:

Priority queue space: O(m \* n).

Distance array: O(m \* n).

**Refer to**

<https://leetcode.com/problems/minimum-cost-to-make-at-least-one-valid-path-in-a-grid/solutions/524820/java-2-different-solutions-clean-code/>

**✔️ Solution 1: Dijkstra's algorithm ~ 20ms**

For the problem, we can create a graph with 4mn edges and mn nodes. By using the Dijkstra algorithm, we can guarantee to achieve it in O(ElogV) ~ O(mn \* log(mn)).

class Solution {

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

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

int m = grid.length, n = grid[0].length;

PriorityQueue<int[]> q = new PriorityQueue<>((o1, o2) -> o1[0] - o2[0]); // minHeap by cost

q.offer(new int[]{0, 0, 0});

int[][] dist = new int[m][n];

for (int i = 0; i < m; i++) Arrays.fill(dist[i], Integer.MAX\_VALUE);

dist[0][0] = 0;

while (!q.isEmpty()) {

int[] top = q.poll();

int cost = top[0], r = top[1], c = top[2];

if (dist[r][c] != cost) continue; // avoid outdated (dist[r,c], r, c) to traverse neighbors again!

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

int nr = r + DIR[i][0], nc = c + DIR[i][1];

int ncost = cost;

if (i != (grid[r][c] - 1)) ncost += 1; // change direction -> ncost = cost + 1

if (dist[nr][nc] > ncost) {

dist[nr][nc] = ncost;

q.offer(new int[]{ncost, nr, nc});

}

}

}

}

return dist[m - 1][n - 1];

}

}

Complexity

Time: O(ElogV) ~ O(mn \* log(mn)), E = 4mn, V = mn

Space: O(m\*n)

Similar problems:

Swim in Rising Water

Path With Maximum Minimum Value

**✔️ Solution 2: BFS + DFS ~ 8ms**

Inspired from @lee215 by this post: [BFS + DFS](https://leetcode.com/problems/minimum-cost-to-make-at-least-one-valid-path-in-a-grid/discuss/524886/Python-BFS-and-DFS)

class Solution {

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

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

int m = grid.length, n = grid[0].length, cost = 0;

int[][] dp = new int[m][n];

for (int i = 0; i < m; i++) Arrays.fill(dp[i], Integer.MAX\_VALUE);

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

dfs(grid, 0, 0, dp, cost, bfs);

while (!bfs.isEmpty()) {

cost++;

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

int[] top = bfs.poll();

int r = top[0], c = top[1];

for (int i = 0; i < 4; i++) dfs(grid, r + DIR[i][0], c + DIR[i][1], dp, cost, bfs);

}

}

return dp[m - 1][n - 1];

}

void dfs(int[][] grid, int r, int c, int[][] dp, int cost, Queue<int[]> bfs) {

int m = grid.length, n = grid[0].length;

if (r < 0 || r >= m || c < 0 || c >= n || dp[r][c] != Integer.MAX\_VALUE) return;

dp[r][c] = cost;

bfs.offer(new int[]{r, c}); // add to try to change direction later

int nextDir = grid[r][c] - 1;

dfs(grid, r + DIR[nextDir][0], c + DIR[nextDir][1], dp, cost, bfs);

}

}

Complexity

Time & Space: O(m\*n)

**Refer to**

[L743.Network Delay Time (Ref.L505,L1368)](note://17E4C21F015F4605AAD4D6878F0D0A67)

[L778.Swim in Rising Water (Ref.L1368)](note://WEB2176826d0a2ce65cdc47c7317edd5a19)

[L934.Shortest Bridge (Ref.L1368)](note://WEBc4385dc955b9c6b7f57e72f45e3ef5f7)

[L1102.Path With Maximum Minimum Value (Ref.L1368)](note://WEBf4a6c0c7ead8e64acb71fe2f9814cb8f)

[L2556.Disconnect Path in a Binary Matrix by at Most One Flip](note://WEB3322914f94be5e34556afdcf5b35b500)

[L2203.Minimum Weighted Subgraph With the Required Paths](note://WEBa77cc0b31467a4612ff75d4806548614)

[L2290.Minimum Obstacle Removal to Reach Corner (Ref.L1293)](note://WEBcd918985c5097f42e558acf5ab25def8)

[0-1 BFS](note://WEB28136176b9a074a7d4b4ae45d6b6bc7a)

[Dijkstra Shortest Path Algorithm - A Detailed and Visual Introduction](note://80857119213E49EC840091BB3F7E4356)