1368. Minimum Cost to Make at Least One Valid Path in a Grid

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]`)

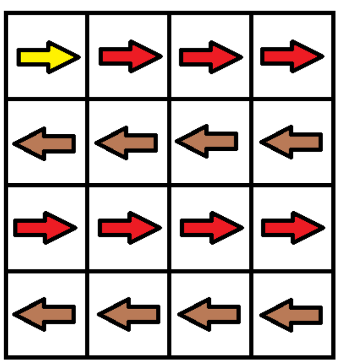
Notice that there could be some signs on the cells of the grid that point outside the grid.

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**.

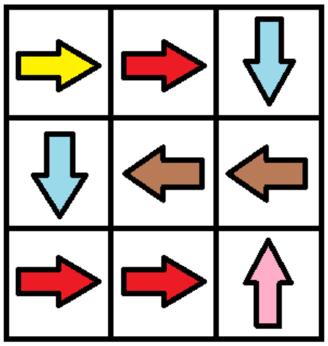
Return *the minimum cost to make the grid have at least one valid path*.

Example 1:



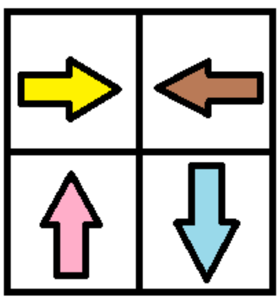
```
Input: grid = [[1,1,1,1],[2,2,2,2],[1,1,1,1],[2,2,2,2]]
Output: 3
Explanation: You will start at point (0, 0).
The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) --> (0, 3) change
the arrow to down with cost = 1 --> (1, 3) --> (1, 2) --> (1, 1) --> (1, 0) change
the arrow to down with 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`

1368. Minimum Cost to Make at Least One Valid Path in a Grid

```
/*
   Dijkstra
   Time complexity: O(mnlog(mn))
   Space complexity: O(2mn)
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
typedef std::pair<int,int> ii;
typedef std::pair<int,ii> iii;
typedef std::vector<iii> viii;

class Solution {
   private:
      // Directions for movement: right, left, down, up
      vvi directions= {{0,1},{0,-1},{1,0},{-1,0}};
public:
   int minCost(vvi& grid){
      int m=grid.size(),n=grid[0].size();

      vvi min_costs(m,vi(n,INT_MAX));
      min_costs[0][0]=0;

      std::priority_queue<iii,viii,std::greater<iii>> min_heap;
      min_heap.push({0,{0,0}});

      while(!min_heap.empty()){
         // Pick the cell with minimum cost
         auto[cur_cost,cur_cell]=min_heap.top();
         min_heap.pop();

         int cur_row=cur_cell.first;
         int cur_col=cur_cell.second;

         if(cur_row==m-1 && cur_col==n-1) return min_costs[m-1][n-1];
```

```
        for(int i=0;i<4;++i){
            vi dir=directions[i];
            int new_row=cur_row+dir[0];
            int new_col=cur_col+dir[1];

            if(new_row<0 || new_col<0 || new_row>m-1 || new_col>n-1) continue;

            // Update the current cost of the current cell
            // If we need to change directions add1 to the current cost
            int new_cost=cur_cost+(grid[cur_row][cur_col]!=(i+1));

            // Path relaxation: Update the cost if we found a better path
            if(min_costs[new_row][new_col]>new_cost){
                min_costs[new_row][new_col]=new_cost;
                min_heap.push({new_cost,{new_row,new_col}});
            }
        }
    }
    return min_costs[m-1][n-1];
  }
};
```

1368. Minimum Cost to Make at Least One Valid Path in a Grid

Approach#2: 0-1 Breadth-First Search

Dijkstra's algorithm works well for finding the shortest path, but our problem has a unique feature: the path costs are either 0 or 1. This is key because ***any path with only 0-cost edges, no matter how long, will always be better than one that uses even a single 1-cost edge***. ***Therefore, it makes sense to prioritize exploring 0-cost edges first***. Only after all 0-cost edges have been explored, should we move on to the 1-cost edges. ***This insight leads us to a modification of the Breadth-First Search (BFS) algorithm, known as 0-1 BFS***.

In 0-1 BFS, we adjust the traditional BFS by using a deque (double-ended queue) instead of a regular queue. The deque allows us to prioritize 0-cost edges more efficiently. Each element of the deque will store the row and column indices of a cell, and we will maintain a min_cost grid to track the minimum cost to reach each cell.

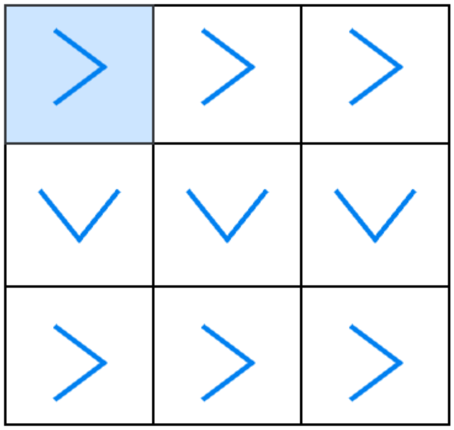
As we visit each cell, we evaluate its four neighboring cells. If moving to a neighbor doesn't require a sign change (i.e., the move is a 0-cost move), we add that neighbor to the front of the deque because we want to explore it immediately. On the other hand, if a sign change is required (making it a 1-cost move), we add the neighbor to the back of the deque, ensuring it gets explored later, after all the 0-cost moves.

For each neighbor we explore, we calculate the cost to reach it and compare it to the current value in the min_cost grid. If the calculated cost is lower, we update min_cost with the new, cheaper value.

Once the BFS traversal completes and all cells have been processed, the minimum cost to reach the bottom-right corner will be stored in min_cost. We return this value as the solution to the problem.

The below slideshow demonstrates the algorithm in action:

Initial State



Current Grid

0	inf	inf
inf	inf	inf
inf	inf	inf

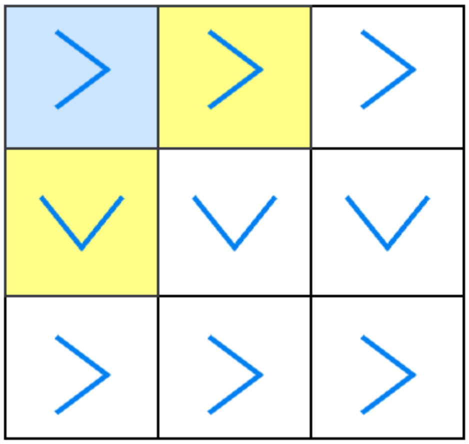
minCost Grid

(0,0)

deque

Start at cell (0,0). Add (0,0) to front of deque

Process (0,0)



Current Grid

0	0	inf
1	inf	inf
inf	inf	inf

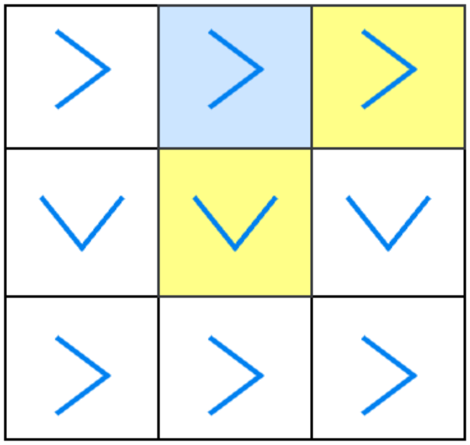
minCost Grid

(0,1) (1,0)

deque

Check neighbours. Fill minCost array.

Process (0,1)



Current Grid

0	0	0
1	1	inf
inf	inf	inf

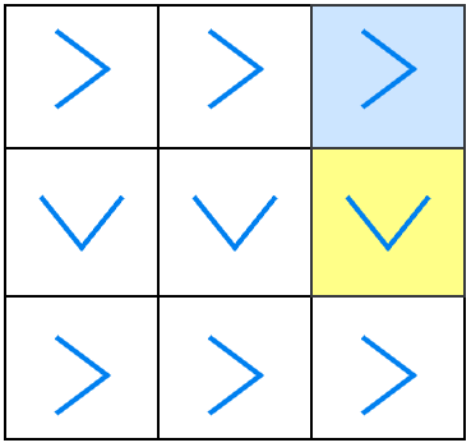
minCost Grid

(0,2) (1,0) (1,1)

deque

Check neighbours. Fill minCost array.

Process (0,2)



Current Grid

0	0	0
1	1	1
inf	inf	inf

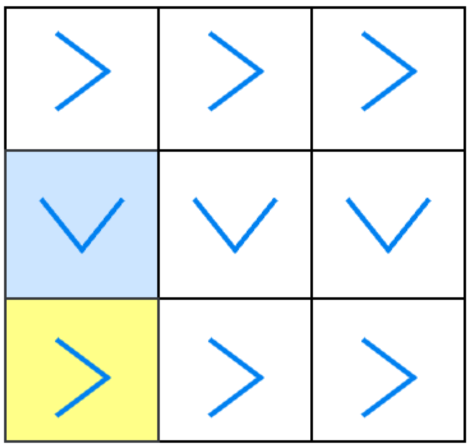
minCost Grid

(1,0) (1,1) (1,2)

deque

Check neighbours. Fill minCost array.

Process (1,0)



Current Grid

0	0	0
1	1	1
1	inf	inf

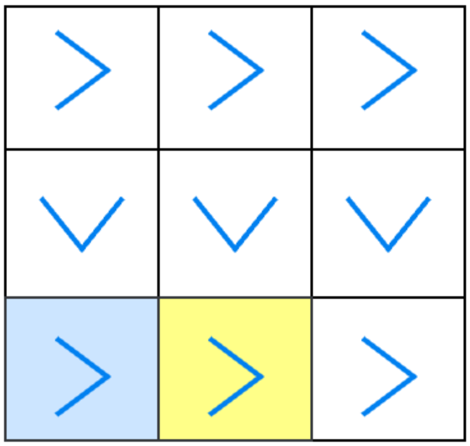
minCost Grid

(2,0) (1,1) (1,2)

deque

Check neighbours. Fill minCost array.

Process (2,0)



Current Grid

0	0	0
1	1	1
1	1	inf

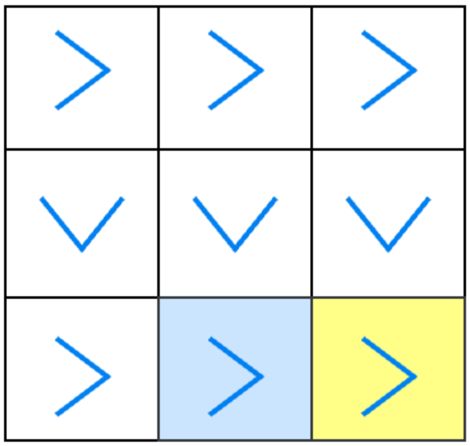
minCost Grid

(2,1)	(1,1)	(1,2)

deque

Check neighbours. Fill minCost array.

Process (2,1)



Current Grid

0	0	0
1	1	1
1	1	1

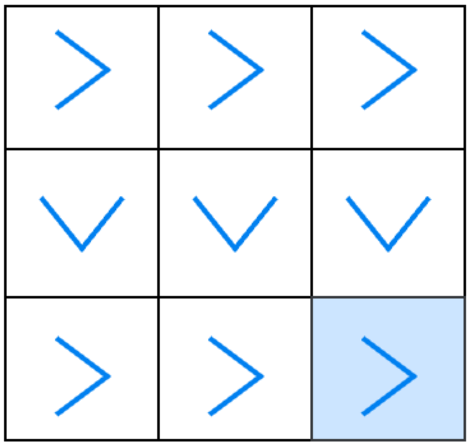
minCost Grid

(2,2) (1,1) (1,2)

deque

Check neighbours. Fill minCost array.

Process (2,2)



Current Grid

0	0	0
1	1	1
1	1	1

minCost Grid

(1,1) (1,2)

deque

Reached destination. Needed 1 modification

1368. Minimum Cost to Make at Least One Valid Path in a Grid

```
/*
   0/1 BFS
   Time complexity: O(mn)
   Space complexity: O(2mn)
*/

typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
typedef std::pair<int,int> ii;
class Solution {
   private:
      // Directions for movement: right, left, down, up
      vvi directions= {{0,1},{0,-1},{1,0},{-1,0}};
public:
   int minCost(vvi& grid){
      int m=grid.size(),n=grid[0].size();

      vvi min_costs(m,vi(n,INT_MAX));
      min_costs[0][0]=0;


      std::deque<ii> deq; // {row,col}
      deq.push_back({0,0});
```

```
    while (!deq.empty()){
        auto [cur_row,cur_col]=deq.front();
        deq.pop_front();

        if(cur_row==m-1 && cur_col==n-1) return min_costs[m-1][n-1];

        for (int i=0;i<4;++i){
            vi dir=directions[i];
            int new_row=cur_row+dir[0];
            int new_col=cur_col+dir[1];

            if(new_row<0 || new_col<0 || new_row>m-1 || new_col>n-1) continue;

            // Compute the cost of the current cell
            // 1: if we need to change direction
            // 0: otherwise
            int cost=grid[cur_row][cur_col]!=(i+1);

            // If the cost at the new cell is greater than the cost of the current
            // cell + the cost of the current cell, then update the cost of the new cell
            if(min_costs[new_row][new_col]>min_costs[cur_row][cur_col]+cost){
                min_costs[new_row][new_col]=min_costs[cur_row][cur_col]+cost;

                // If the we need to change direction from the current cell to
                // reach the new cell, push the new cell to the back to deal with it later
                if(cost==1) deq.push_back({new_row,new_col});

                // Otherwise, push it to the front to deal with it as soon as possible
                else deq.push_front({new_row,new_col});
            }
        }
    }

    return min_costs[m-1][n-1];
  }
};
```