2290. Minimum Obstacle Removal to Reach Corner

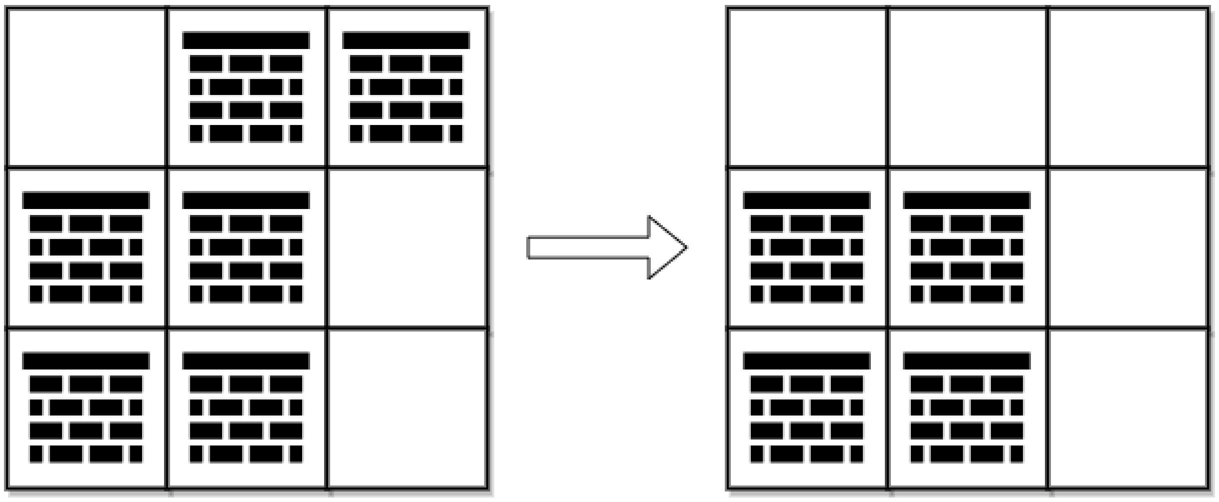
You are given a **0-indexed** 2D integer array `grid` of size `m x n`. Each cell has one of two values:

- `0` represents an **empty** cell,
- `1` represents an **obstacle** that may be removed.

You can move up, down, left, or right from and to an empty cell.

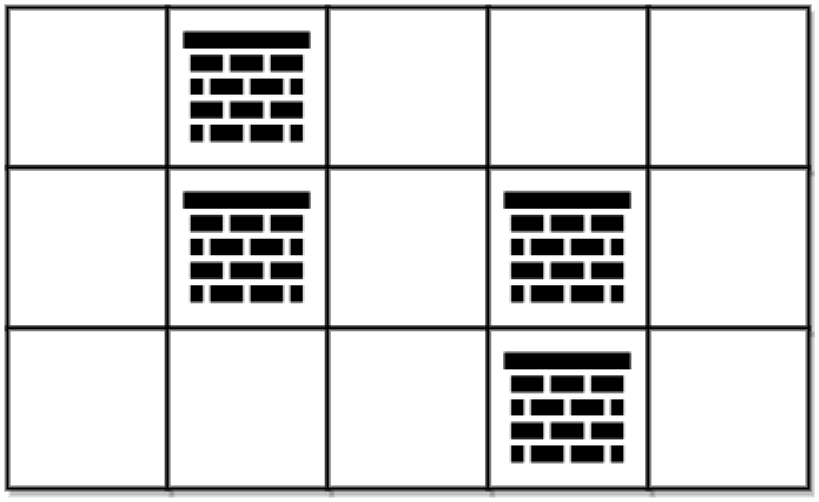
Return *the* ***minimum*** *number of* ***obstacles*** *to* ***remove*** *so you can move from the upper left corner* `(0, 0)` *to the lower right corner* `(m - 1, n - 1)`.

Example 1:



```
Input: grid = [[0,1,1],[1,1,0],[1,1,0]]
Output: 2
Explanation: We can remove the obstacles at (0, 1) and (0, 2) to create a path from
(0, 0) to (2, 2).
It can be shown that we need to remove at least 2 obstacles, so we return 2.
Note that there may be other ways to remove 2 obstacles to create a path.
```

Example 2:



Input: grid = [[0,1,0,0,0],[0,1,0,1,0],[0,0,0,1,0]]
Output: 0
Explanation: We can move from (0, 0) to (2, 4) without removing any obstacles, so we return 0.

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 105`
- `2 <= m * n <= 105`
- `grid[i][j]` is either `0` **or** `1`.
- `grid[0][0] == grid[m - 1][n - 1] == 0`

2290. Minimum Obstacle Removal to Reach Corner

```
/*
  Dijkstra
  Time complexity: O(2m.n+m.n log(m.n)) = O(m.n log(m.n))
  Space complexity: O(3mn))=O(m.n)
*/
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:
    int m,n;
  public:
    int dijkstra(vvi& grid){
      vvi directions={{-1,0},{1,0},{0,1},{0,-1}};
      vvi visited(m,vi(n,false));
      vvi distances(m,vi(n,INT_MAX));
      distances[0][0]=0;
      std::priority_queue<iii,viii,std::greater<iii>> min_heap;
      min_heap.push({0,{0,0}});
      while(!min_heap.empty()){
        auto[d,cell]=min_heap.top();
        min_heap.pop();
        int i=cell.first;
        int j=cell.second;
        if(i==m-1 && j==n-1) return distances[m-1][n-1];
        if(visited[i][j]) continue;
        visited[i][j]=true;
        for(auto& dir: directions){
          int x=i+dir[0];
          int y=j+dir[1];
          if(x<0 || x>m-1 || y<0 || y>n-1) continue;
          int w=grid[x][y]==0?0:1;
          if(distances[x][y]>distances[i][j]+w){
            distances[x][y]=distances[i][j]+w;
            min_heap.push({distances[x][y],{x,y}});
          }
        }
      }
      return distances[m-1][n-1];
    }
```

```
        int minimumObstacles(vvi& grid){
            m=grid.size();
            n=grid[0].size();
            return dijkstra(grid);
        }
};
```

2290. Minimum Obstacle Removal to Reach Corner

```
/*
   A*
   Time complexity: O(m.n+m.n log(m.n))
   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:
        int m,n;
        // Create class cell
        class Cell{
            public:
                int row,col;
                int cost; // Cost the reach the cell at (row,col)
            public:
                Cell(int row,int col,int cost):
                    row(row),
                    col(col),
                    cost(cost) {}
        };
```

```
public:
  // Using A* to prioritize the cell with minimum cost.
  // The cost of a cell is the number of removed walls to reach that cell.
  int a_star(vvi& grid,Cell* cell){
     vvi directions={{-1,0},{1,0},{0,1},{0,-1}};
     vvi visited(m,vi(n,false));
     std::priority_queue<std::pair<int,Cell*>,
                      std::vector<std::pair<int,Cell*>>,
                      std::greater<std::pair<int,Cell*>>> min_heap;
     min_heap.push({0,cell});
     while(!min_heap.empty()){
        // Pick the cell with minimum cost
        Cell* cell=min_heap.top().second;
        min_heap.pop();
        // If we reach the destination, return its cost
        // which is the number of removed walls to reach it
        if(cell->row==m-1 && cell->col==n-1) return cell->cost;
        if(visited[cell->row][cell->col]) continue;
        visited[cell->row][cell->col]=true;
        for(auto& dir: directions){
           int x=cell->row+dir[0];
           int y=cell->col+dir[1];
           if(x<0 || x>m-1 || y<0 || y>n-1) continue;
           if(!visited[x][y]){
              // New cost = f+h
              // f: cost of current cell
              // h: 0, if no wall to remove
              //1, if there is a wall to remove
              int f=cell->cost;
              int h=grid[x][y];
              int cost=f+h;
              // Add the new cell to the min heap
              Cell* next_cell=new Cell(x,y,cost);
              min_heap.push({h,next_cell});
           }
        }
     }
     return -1;
  }
```

```
        int minimumObstacles(vvi& grid){
            m=grid.size();
            n=grid[0].size();
            Cell* cell=new Cell(0,0,0);
            return a_star(grid,cell);
        }
};
```

Approach 2: 0-1 Breadth-First Search (BFS)

Intuition

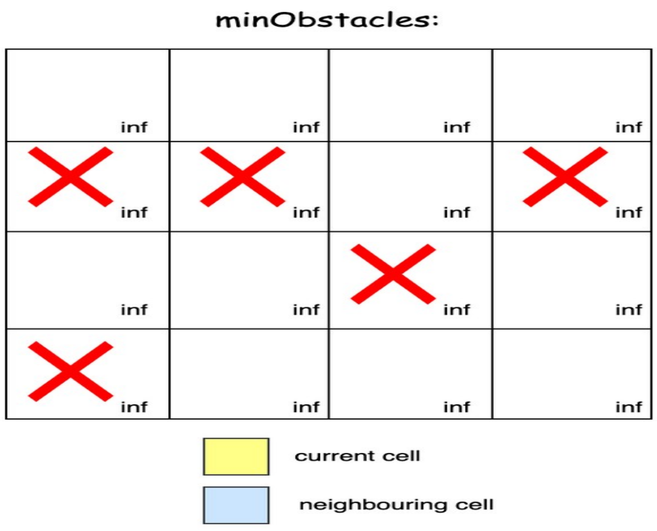
As stated earlier, moving through cells without obstacles has no cost. Therefore, we prioritize exploring neighboring empty cells first, only moving to cells with obstacles when no free cells are left.

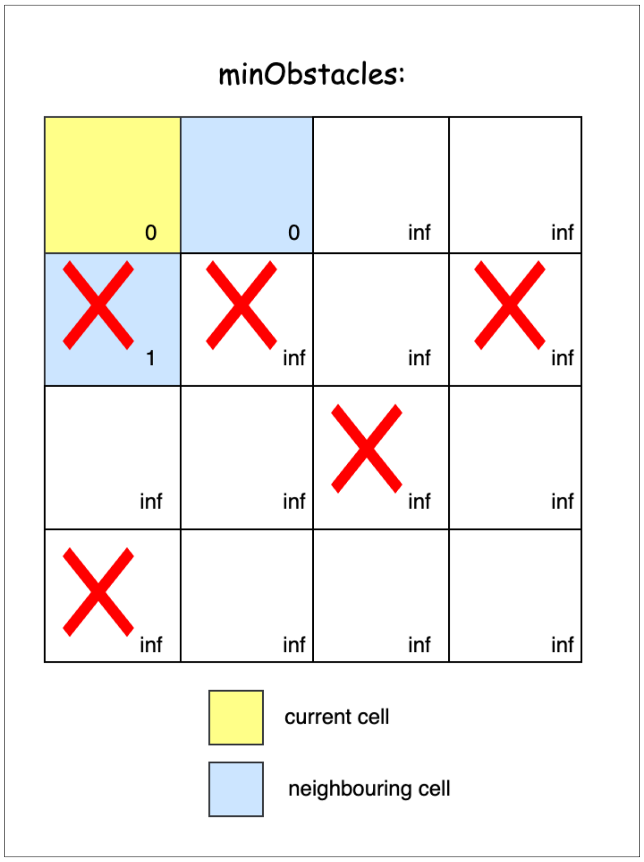
We perform a BFS using a deque to manage the queue. When exploring neighboring cells, we add empty cells to the front of the deque for immediate exploration, and cells with obstacles to the back, delaying their exploration.

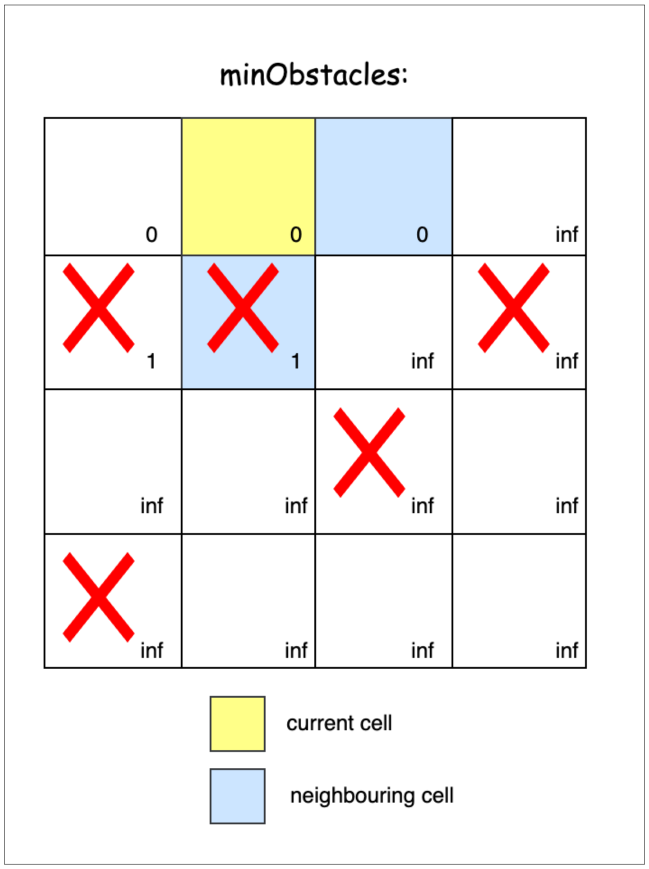
We maintain a result grid, `minObstacles`, initialized to infinity (indicating they are unvisited), to track the minimum obstacles encountered at each cell. We'll add the top left cell to the deque and begin our exploration. At each step, we'll pop the top cell in the deque and explore its neighbors. All empty neighbors go to the front of the deque, while others go to the bottom with their obstacle count increased by 1. Simultaneously, we'll update the `minObstacles` value for each neighboring position.

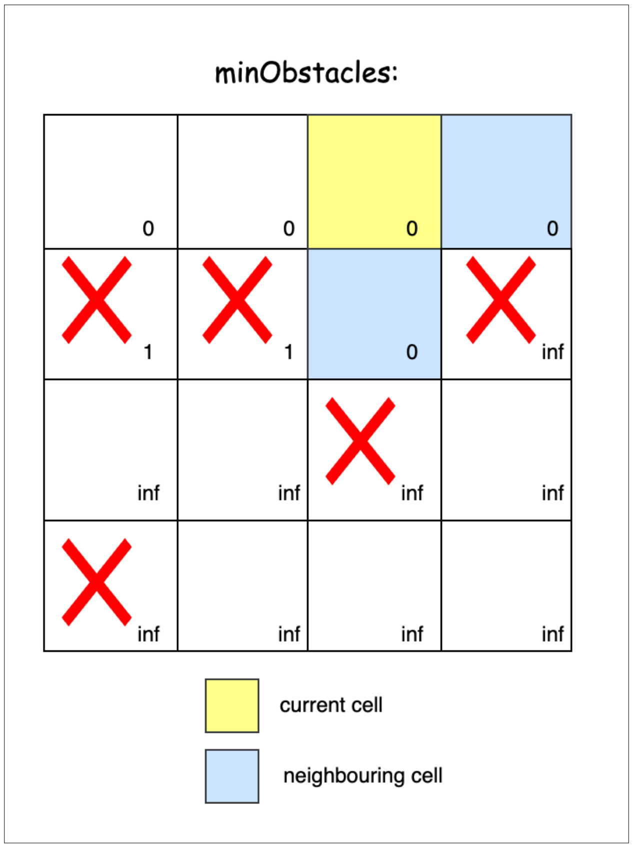
Once all cells are explored, the value at the bottom-right cell of `minObstacles` will give the minimum obstacles encountered on the shortest path.

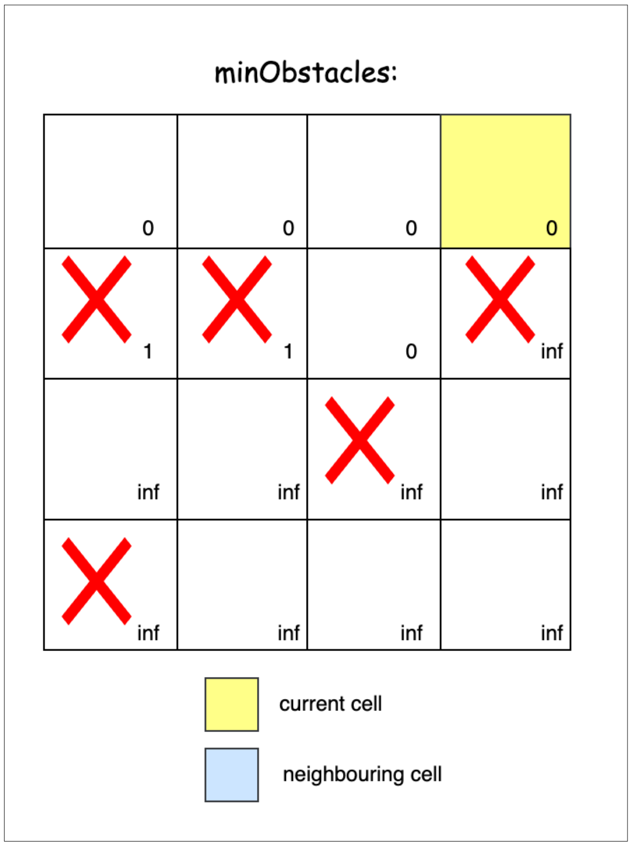
Here's a brief visualization of how the `minObstacles` matrix is filled up step by step:

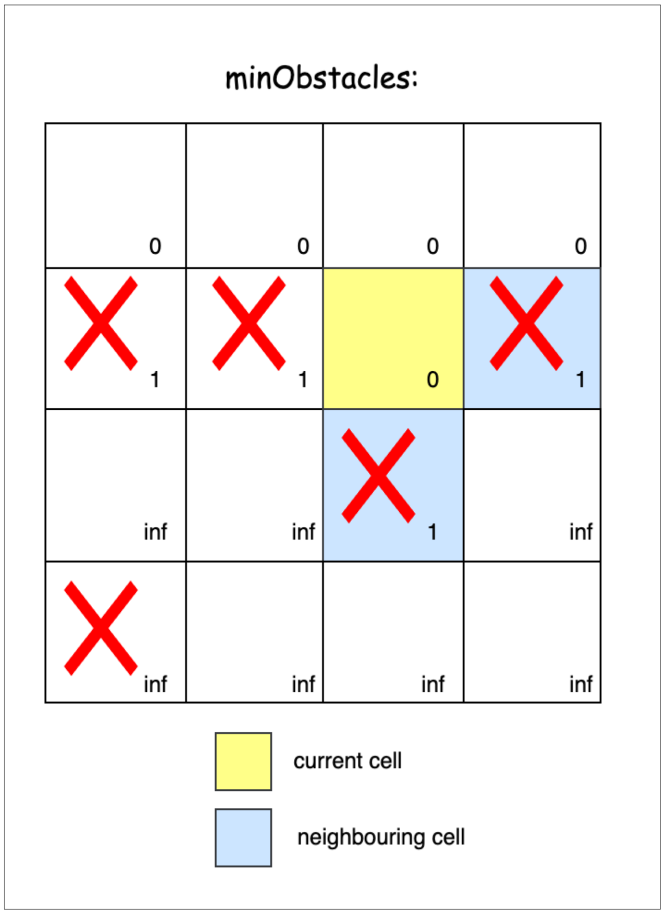


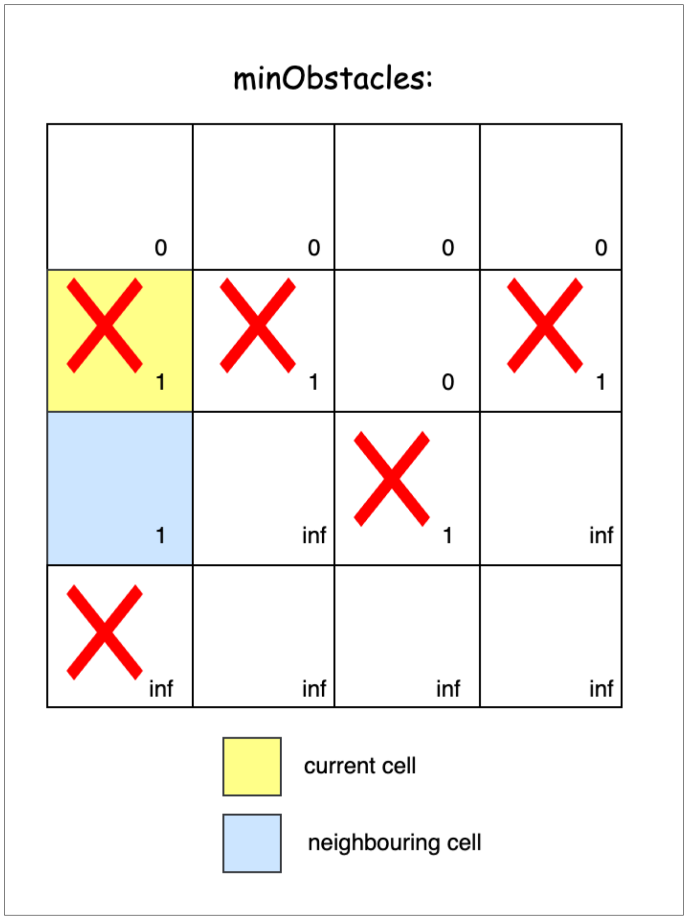

minObstacles:
0 0 inf inf
1 inf inf inf
inf inf inf inf
inf inf inf inf
current cell
neighbouring cell

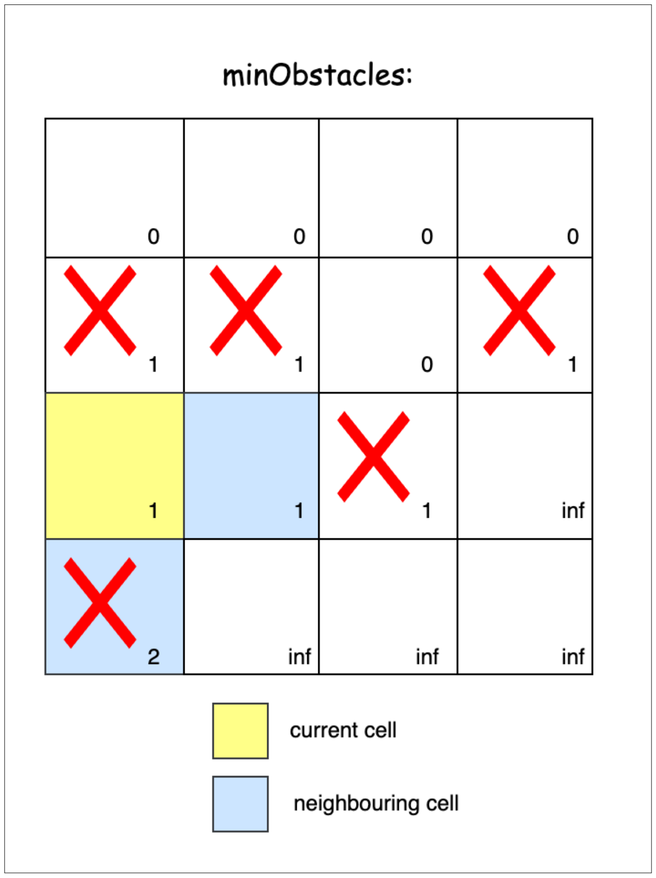

minObstacles:
0 0 0 inf
1 1 inf inf
inf inf inf inf
inf inf inf inf
current cell
neighbouring cell

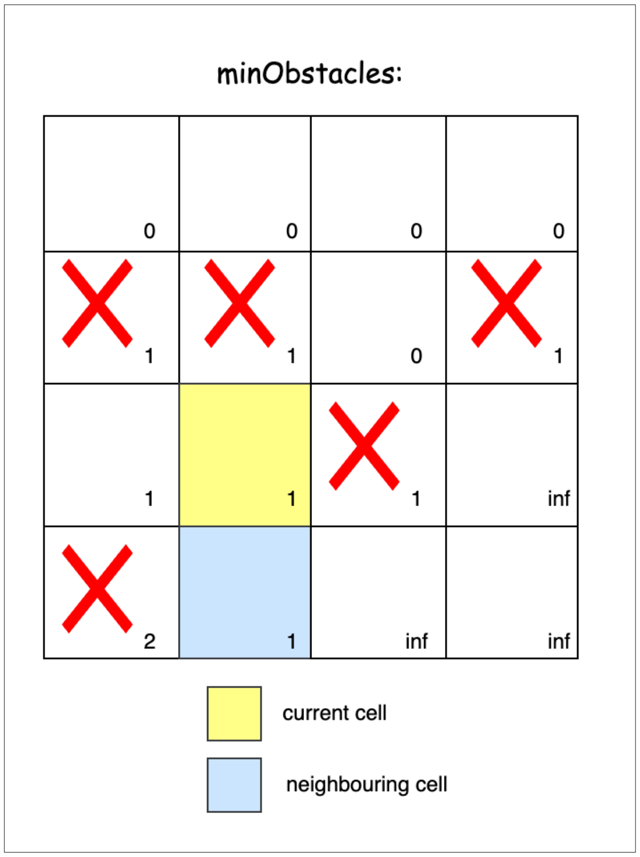

minObstacles:
0 0 0 0
1 1 0 inf
inf inf inf inf
inf inf inf inf
current cell
neighbouring cell

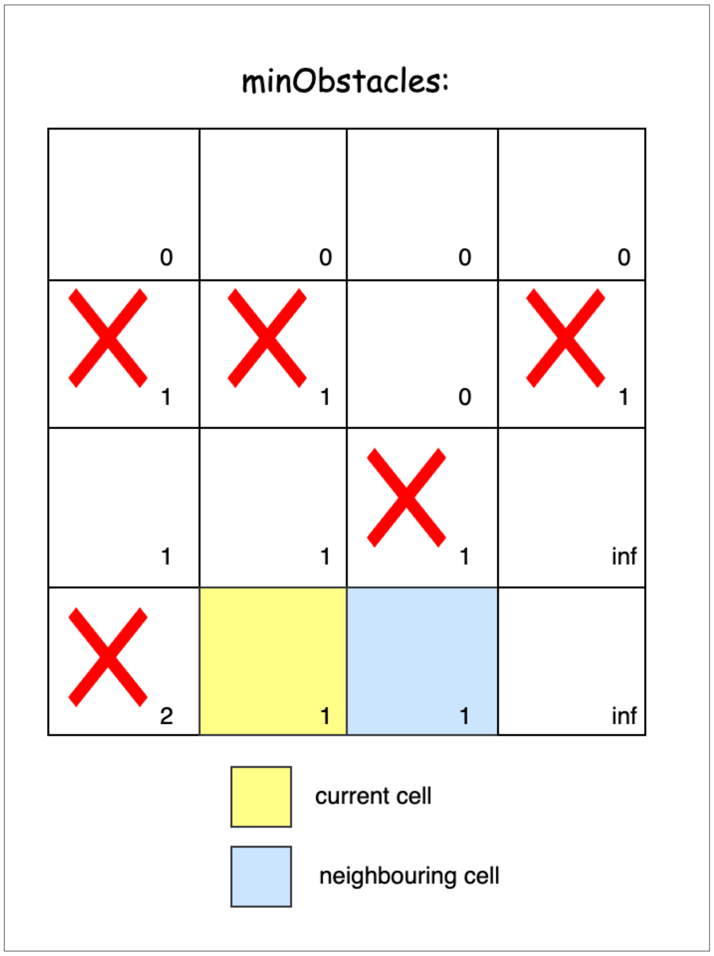

minObstacles:
0 0 0 0
1 1 0 inf
inf inf inf inf
inf inf inf inf
current cell
neighbouring cell

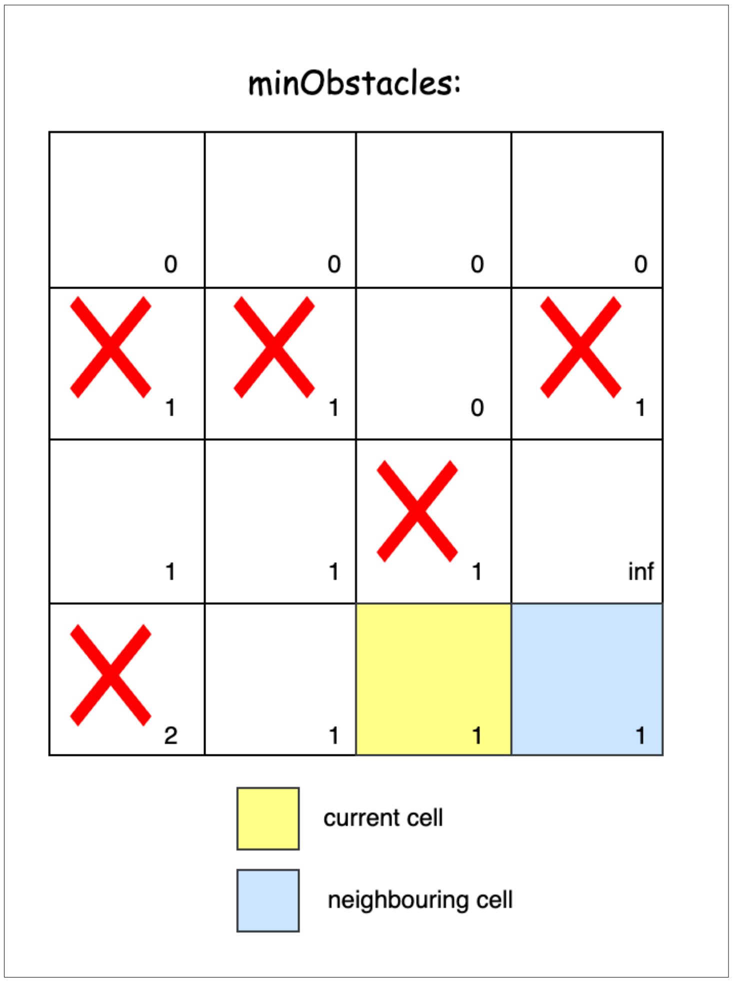

minObstacles:
0 0 0 0
1 1 0 1
inf inf 1 inf
inf inf inf inf
current cell
neighbouring cell

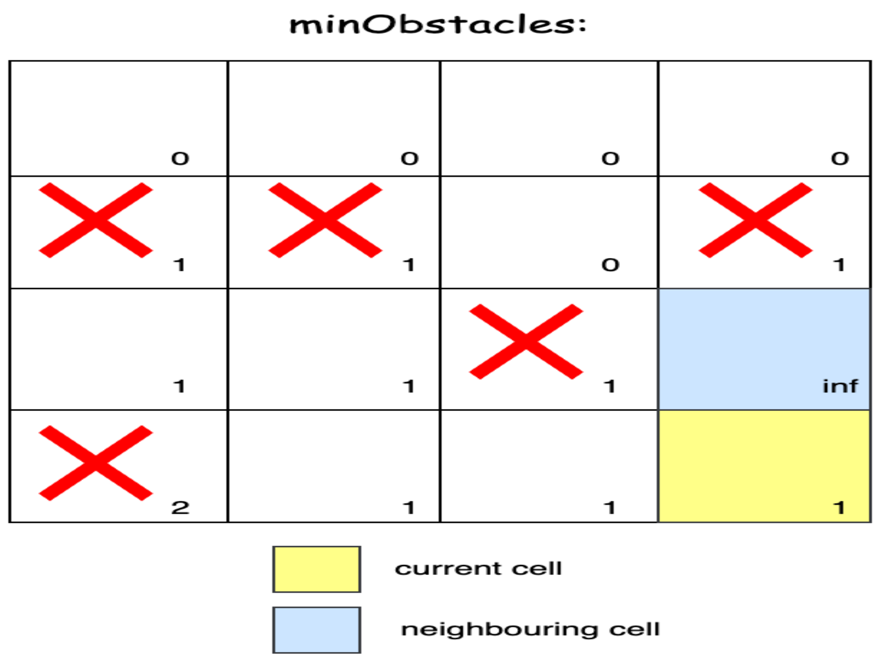

minObstacles:
0 0 0 0
1 1 0 1
1 inf 1 inf
inf inf inf inf
current cell
neighbouring cell


minObstacles:
0 0 0 0
1 1 0 1
1 1 1 inf
2 inf inf inf
current cell
neighbouring cell


minObstacles:
0 0 0 0
1 1 0 1
1 1 1 inf
2 1 inf inf
current cell
neighbouring cell


minObstacles:
0 0 0 0
1 1 0 1
1 1 1 inf
2 1 1 inf
current cell
neighbouring cell


minObstacles:
0 0 0 0
1 1 0 1
1 1 1 inf
2 1 1 1
current cell
neighbouring cell


minObstacles:
0 0 0 0
1 1 0 1
1 1 1 inf
2 1 1 1
current cell
neighbouring cell

```
class Solution {
private:
    // Directions for movement: right, left, down, up
    vector<vector<int>> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
```

```
public:
    int minimumObstacles(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        // Distance matrix to store the minimum obstacles removed to reach each
        // cell
        vector<vector<int>> minObstacles(m, vector<int>(n, INT_MAX));

        minObstacles[0][0] = 0;

        deque<array<int, 3>> deque;

        deque.push_back({0, 0, 0});  // {obstacles, row, col}
        while (!deque.empty()) {
            auto current = deque.front();
            deque.pop_front();

            int obstacles = current[0], row = current[1], col = current[2];

            // Explore all four possible directions from the current cell
            for (auto& dir : directions) {
                int newRow = row + dir[0], newCol = col + dir[1];

                if (isValid(grid, newRow, newCol) &&
                    minObstacles[newRow][newCol] == INT_MAX) {

                    if (grid[newRow][newCol] == 1) {
                        // If it's an obstacle, add 1 to obstacles and push to
                        // the back
                        minObstacles[newRow][newCol] = obstacles + 1;
                        deque.push_back({obstacles + 1, newRow, newCol});
                    } else {
                        // If it's an empty cell, keep the obstacle count and
                        // push to the front
                        minObstacles[newRow][newCol] = obstacles;
                        deque.push_front({obstacles, newRow, newCol});
                    }
                }
            }
        }
        return minObstacles[m - 1][n - 1];
    }
```

```
private:
    // Helper method to check if the cell is within the grid bounds
    bool isValid(vector<vector<int>>& grid, int row, int col) {
        return row >= 0 && col >= 0 && row < grid.size() &&
               col < grid[0].size();
    }
};
```

Complexity Analysis

Let m be the number of rows and n be the number of columns in the grid.

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

  Each of the $m \cdot n$ cells in the grid is visited exactly once because we only process unvisited cells. The deque operations are all $O(1)$.

  Thus, the total time complexity is $O(m \cdot n)$.

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

  The `minObstacles` array and the deque both take $O(m \cdot n)$ space. All other variables take constant space.

  Thus, the space complexity remains $O(m \cdot n)$.