2577. Minimum Time to Visit a Cell In a Grid

You are given a `m x n` matrix `grid` consisting of **non-negative** integers where `grid[row][col]` represents the **minimum** time required to be able to visit the cell `(row, col)`, which means you can visit the cell `(row, col)` only when the time you visit it is greater than or equal to `grid[row][col]`.

You are standing in the **top-left** cell of the matrix in the `0th` second, and you must move to **any** adjacent cell in the four directions: up, down, left, and right. Each move you make takes 1 second.

Return *the* ***minimum*** *time required in which you can visit the bottom-right cell of the matrix*. If you cannot visit the bottom-right cell, then return `-1`.

Example 1:

0	1	3	2
5	1	2	5
4	3	8	6

```
Input: grid = [[0,1,3,2],[5,1,2,5],[4,3,8,6]]
Output: 7
Explanation: One of the paths that we can take is the following:
- at t = 0, we are on the cell (0,0).
- at t = 1, we move to the cell (0,1). It is possible because grid[0][1] <= 1.
- at t = 2, we move to the cell (1,1). It is possible because grid[1][1] <= 2.
- at t = 3, we move to the cell (1,2). It is possible because grid[1][2] <= 3.
- at t = 4, we move to the cell (1,1). It is possible because grid[1][1] <= 4.
- at t = 5, we move to the cell (1,2). It is possible because grid[1][2] <= 5.
- at t = 6, we move to the cell (1,3). It is possible because grid[1][3] <= 6.
- at t = 7, we move to the cell (2,3). It is possible because grid[2][3] <= 7.
The final time is 7. It can be shown that it is the minimum time possible.
```

Example 2:

0	2	4
3	2	1
1	0	4

Input: grid = [[0,2,4],[3,2,1],[1,0,4]]
Output: -1
Explanation: There is no path from the top left to the bottom-right cell.

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `2 <= m, n <= 1000`
- `4 <= m * n <= 105`
- `0 <= grid[i][j] <= 105`
- `grid[0][0] == 0`

2577. Minimum Time to Visit a Cell In a Grid

```
/*
   Modified Dijkstra
   Time complexity: O(m.n.log(m.n))
   Space complexity: 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));

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

      while(!min_heap.empty()){
          auto [cur_time,cur_cell]=min_heap.top();
          min_heap.pop();

          int i=cur_cell.first;
          int j=cur_cell.second;

          if(i==m-1 && j==n-1) return cur_time;


          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 needed_time; // Needed time to enter cell(x,y)

              // If we are able to enter next cell(x,y)
              if(cur_time>=grid[x][y]) needed_time=cur_time+1;

              // Otherwise, (cur_time<grid[x][y], we need to wait until, we be
              // able to reach cell(x,y)
              // Move back and forth between cell(i,j) and cell(x,y)
              else{
                  // Time we need to "waste" to move to next cell (x,y)
                  int wasted_time=(grid[x][y]-cur_time)%2==0?1:0;

                  needed_time=grid[x][y]+wasted_time;
              }
              min_heap.push({needed_time,{x,y}});
          }
      }
      return -1; // Never reached
  }
```

```
    int minimumTime(vvi& grid){
        // If from (0,0) we can't move to any cell
        // because current time=1, and values of grid[0][1]>1
        // and grid[1][0]>1
        if(grid[0][1]>1 && grid[1][0]>1) return -1;

        // Otherwise, we have an answer
        m=grid.size();
        n=grid[0].size();

        return dijkstra(grid);
    }
};
```