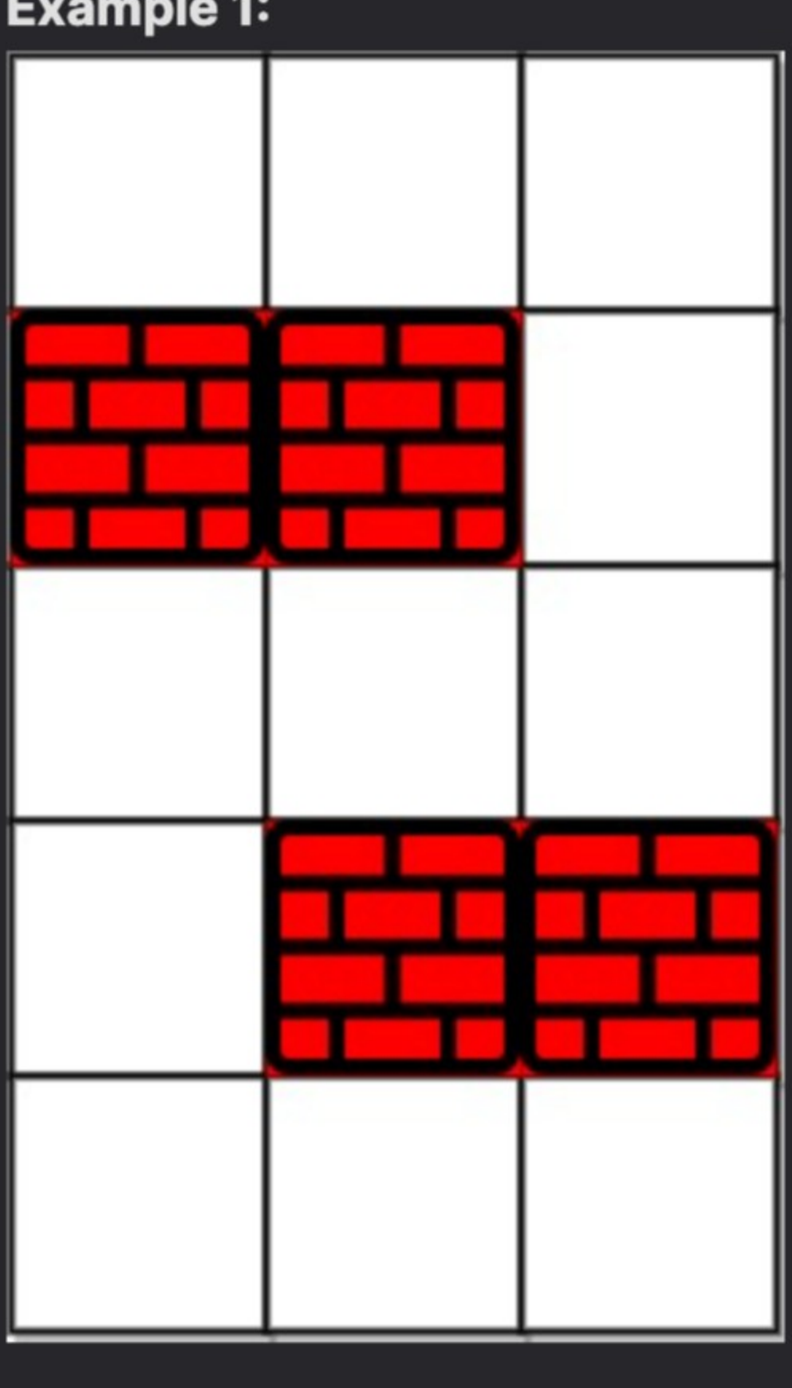
1293. Shortest Path in a Grid with Obstacles Elimination

Leetcode Link

You are given an $m \times n$ integer matrix `grid` where each cell is either 0 (empty) or 1 (obstacle). In one step, you can move up, down, left, or right to an empty cell.

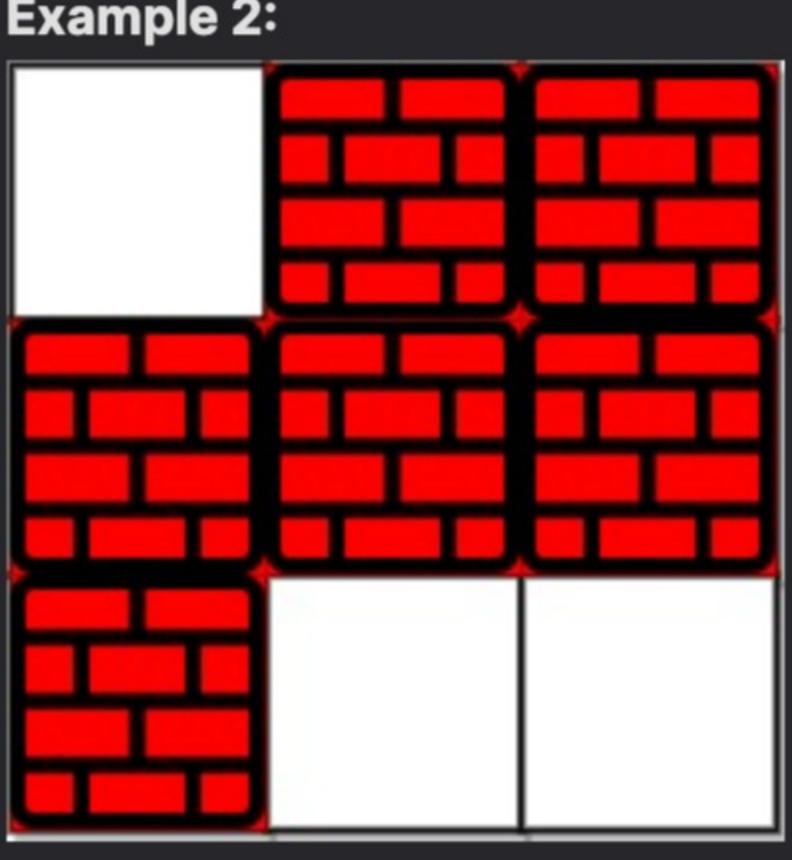
Return *the minimum number of **steps** to walk from the upper left corner* `(0, 0)` *to the lower right corner* `(m - 1, n - 1)` *given that you can eliminate **at most** k obstacles*. If no such walks exist, return -1.

Example 1:



Input: `grid = [[0,0,0],[1,1,0],[0,0,0],[0,1,1],[0,0,0]], k = 1`
Output: `6`
Explanation:
The shortest path without eliminating any obstacle is `10`.
The shortest path with one obstacle elimination at position `(3,2)` is `6`. Such path is `(0,0)` → `(0,1)` → `(0,2)` → `(1,2)` → `(2,2)` → `(3,2)` → `(4,2)`.

Example 2:



Input: `grid = [[0,1,1],[1,1,1],[1,0,0]], k = 1`
Output: `-1`
Explanation: We need to eliminate at least two obstacles to find such a walk.

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 40`
- `1 <= k <= m * n`
- `grid[i][j] is either 0 or 1.`
- `grid[0][0] == grid[m - 1][n - 1] == 0`

Solution

Brute Force

First, we might think to try all possible eliminations of **at most** k obstacles.

Then, on every possible elimination, we can run a BFS/flood fill algorithm on the new grid to find the length of the shortest path. Our final answer will be the minimum of all of these lengths.

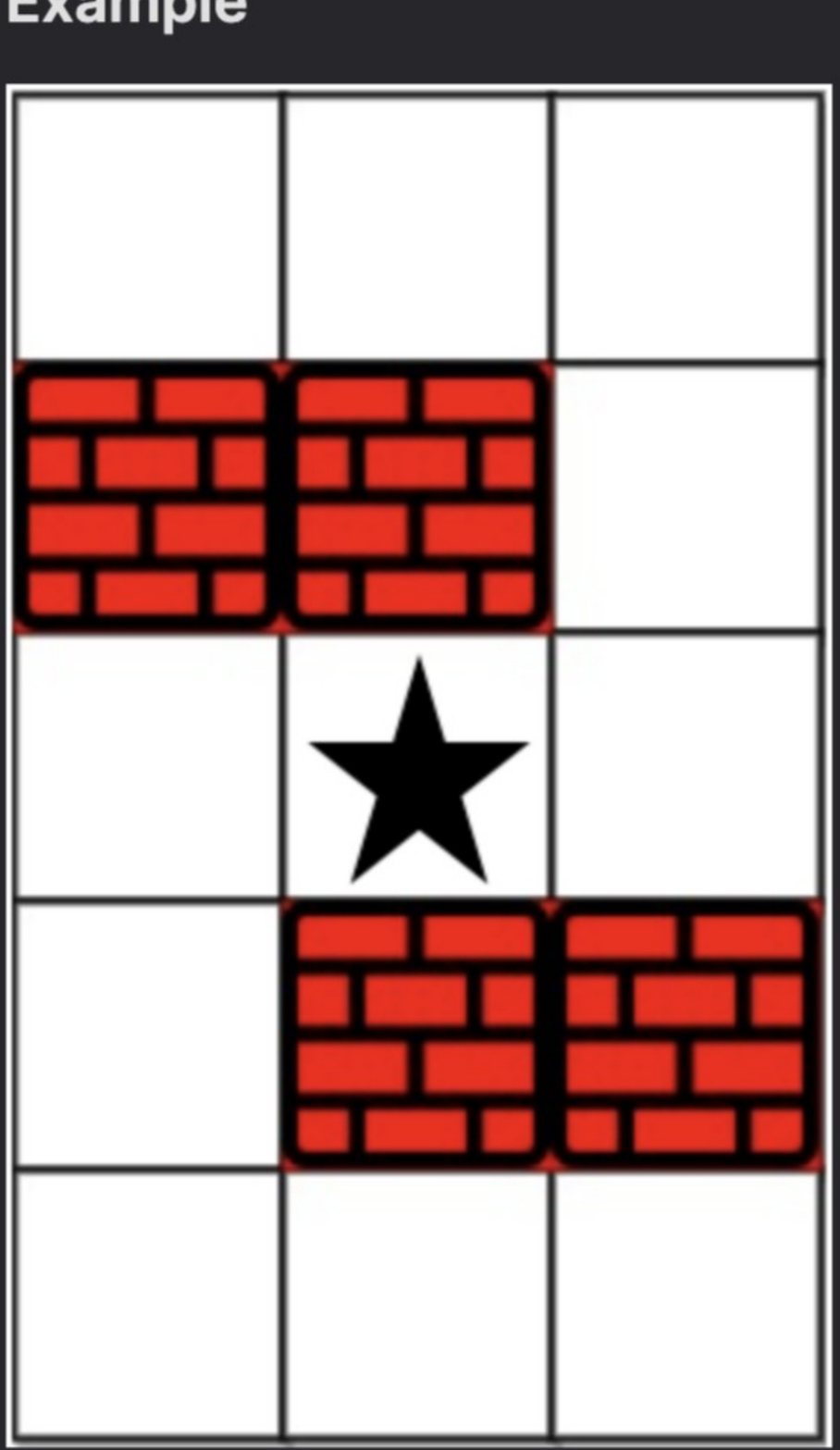
However, this is way too inefficient and complicated.

Full Solution

Instead of thinking of first removing obstacles and then finding the shortest path, we can find the shortest path and remove obstacles along the way when necessary.

To accomplish this, we'll introduce an additional state by adding a counter for the number of obstacles we removed so far in our path. For any path, we can extend that path by moving up, left, down, or right. If the new cell we move into is blocked, we will remove that obstacle and add 1 to our counter. However, since we can remove no more than K obstacles, we can't let our counter exceed K.

Example



Let's look at our destination options if we started in the cell `grid[2][1]` with the obstacle counter at 0.

Cell	Change In Row	Change In Column	Change In Obstacle Counter
`grid[1][1]`	-1	+0	+1
`grid[2][2]`	+0	+1	+0
`grid[3][1]`	+1	+0	+1
`grid[2][0]`	+0	-1	+0

We can also make the observation that each position/state (row, column, obstacle counter) can act as a node in a graph and each destination option can act as a directed edge in a graph.

In this specific example with the node `(2,1,0)`, we have 4 directed edges with the destinations being `(1,1,1)`, `(2,2,0)`, `(3,1,1)`, and `(2,0,0)`.

Since each edge has length 1, we can run a BFS/flood fill algorithm on the graph to find the answer. Using BFS to find the shortest path will work in this case as the graph is unweighted. While running the algorithm, we'll look for the first instance we traverse through a node u which is located in the bottom right corner (i.e. `row = m - 1` and `column = n - 1`). Since BFS/flood fill traverses through nodes in **non-decreasing** order by the distance from the start node `(0,0,0)`, our answer will be the distance from the start node `(0,0,0)` to node u. This is always true no matter what the obstacle counter is for that node (assuming it doesn't exceed k). If no such node is traversed, then our answer is -1.

Essentially, we'll create a graph with nodes having 3 states (row, column, obstacle counter) and run a BFS/flood fill on it to find the minimum distance between the start and end nodes.

Time Complexity

Let's think of how different nodes exist in our graph. There are $O(MN)$ cells in total, and in each cell, our current counter of obstacles ranges from 0 to K, inclusive. This gives us $O(K)$ options for our obstacle counter, yielding $O(MNK)$ nodes. From each node, we have a maximum of 4 other destinations we can visit (i.e. edges), which is equivalent to $O(1)$. From all $O(MNK)$ nodes, we also obtain $O(MNK)$ total edges.

Our graph has $O(MNK)$ nodes and $O(MNK)$ edges. A BFS with $O(MNK)$ nodes and $O(MNK)$ edges will have a final time complexity of $O(MNK)$.

Time Complexity: $O(MNK)$

Space Complexity

Our graph has $O(MNK)$ nodes so a BFS will have a space complexity of $O(MNK)$ as well.

Space Complexity: $O(MNK)$

C++ solution

```cpp
class Solution {
   public:
    int shortestPath(vector<vector<int>>& grid, int k) {
        int m = grid.size();
        int n = grid[0].size();  // dimensions of the grid
        vector<int> deltaX = {-1, 0, 1, 0};
        vector<int> deltaY = {0, 1, 0, -1};
        // nodes are in the form (row, column, obstacles removed so far)
        int dis[m][n][k + 1];   // keeps track of distance of nodes
        bool vis[m][n][k + 1];  // keeps track of whether or not we visited a node
        memset(dis, 0, sizeof(dis));
        memset(vis, false, sizeof(vis));
        queue<vector<int>> q;
        q.push({0, 0, 0});  // starting at upper left corner for BFS/floodfill
        vis[0][0][0] = true;
        while (!q.empty()) {
            vector<int> cur = q.front();
            q.pop();
            int curX = cur[0];  // current row
            int curY = cur[1];  // current column
            int curK = cur[2];  // current obstacles removed
            if (curX == m - 1 &&
                curY == n - 1) {  // check if node is in bottom right corner
                return dis[curX][curY][curK];
            }
            for (int i = 0; i < 4; i++) {
                int newX = curX + deltaX[i];  // row of destination
                int newY = curY + deltaY[i];  // column of destination
                if (newX < 0 || newX >= m || newY < 0 ||
                    newY >= n) {  // check if it's in boundary
                    continue;
                }
                int newK = curK;  // obstacle count of destination
                if (grid[newX][newY] == 1) newK++;
                if (newK > k) {  // surpassed obstacle removal limit
                    continue;
                }
                if (vis[newX][newY][newK]) {  // check if node has been visited before
                    continue;
                }
                dis[newX][newY][newK] = dis[curX][curY][curK] + 1;
                vis[newX][newY][newK] = true;
                q.push({newX, newY, newK});
                // process destination node
            }
        }
        return -1;  // no valid answer found
    }
};
```

Java solution

```java
class Solution {
    public int shortestPath(int[][] grid, int k) {
        int m = grid.length;
        int n = grid[0].length; // dimensions of the grid
        int[] deltaX = {-1, 0, 1, 0};
        int[] deltaY = {0, 1, 0, -1};
        // nodes are in the form (row, column, obstacles removed so far)
        int[][][] dis = new int[m][n][k + 1]; // keeps track of distance of nodes
        boolean[][][] vis =
            new boolean[m][n][k + 1]; // keeps track of whether or not we visited a node
        Queue<int[]> q = new LinkedList<int[]>();
        int[] start = {0, 0, 0};
        q.add(start); // starting at upper left corner for BFS/floodfill
        vis[0][0][0] = true;
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int curX = cur[0]; // current row
            int curY = cur[1]; // current column
            int curK = cur[2]; // current obstacles removed
            if (curX == m - 1
                && curY == n - 1) { // check if node is in bottom right corner
                return dis[curX][curY][curK];
            }
            for (int i = 0; i < 4; i++) {
                int newX = curX + deltaX[i]; // row of destination
                int newY = curY + deltaY[i]; // column of destination
                if (newX < 0 || newX >= m || newY < 0
                    || newY >= n) { // check if it's in boundary
                    continue;
                }
                int newK = curK; // obstacle count of destination
                if (grid[newX][newY] == 1)
                    newK++;
                if (newK > k) { // surpassed obstacle removal limit
                    continue;
                }
                if (vis[newX][newY][newK]) { // check if node has been visited before
                    continue;
                }
                dis[newX][newY][newK] = dis[curX][curY][curK] + 1;
                vis[newX][newY][newK] = true;
                int[] destination = {newX, newY, newK};
                q.add(destination);
                // process destination node
            }
        }
        return -1; // no valid answer found
    }
}
```

Python Solution

```python
class Solution(object):
    def shortestPath(self, grid, k):
        """
        :type grid: List[List[int]]
        :type k: int
        :rtype: int
        """
        m = len(grid)
        n = len(grid[0])
        # dimensions of the grid
        deltaX = [-1, 0, 1, 0]
        deltaY = [0, 1, 0, -1]
        # nodes are in the form (row, column, obstacles removed so far)
        dis = [
            [[0 for x in range(k + 1)] for y in range(n)] for z in range(m)
        ]  # keeps track of distance of nodes
        vis = [
            [[False for col in range(k + 1)] for col in range(n)] for row in range(m)
        ]  # keeps track of whether or not we visited a node
        q = []
        q.append((0, 0, 0))  # starting at upper left corner for BFS/floodfill
        vis[0][0][0] = True
        while q:
            (curX, curY, curK) = q.pop(0)
            # curX refers to current row
            # curY refers to current column
            # curK refers to current obstacles removed
            if (
                curX == m - 1 and curY == n - 1
            ):  # check if node is in bottom right corner
                return dis[curX][curY][curK]
            for i in range(4):
                newX = curX + deltaX[i]  # row of destination
                newY = curY + deltaY[i]  # column of destination
                if (
                    newX < 0 or newX >= m or newY < 0 or newY >= n
                ):  # check if it's in boundary
                    continue
                newK = curK  # obstacle count of destination
                if grid[newX][newY] == 1:
                    newK += 1
                if newK > k:  # surpassed obstacle removal limit
                    continue
                if vis[newX][newY][newK]:  # check if node has been visited before
                    continue
                dis[newX][newY][newK] = dis[curX][curY][curK] + 1
                vis[newX][newY][newK] = True
                q.append((newX, newY, newK))
                # process destination node
        return -1  # no valid answer found

```

Javascript Solution

```javascript
/**
 * @param {number[][]} grid
 * @param {number} k
 * @return {number}
 */
var shortestPath = function (grid, k) {
  const m = grid.length;
  const n = grid[0].length; // dimensions of the grid
  let deltaX = [-1, 0, 1, 0];
  let deltaY = [0, 1, 0, -1];
  // nodes are in the form (row, column, obstacles removed so far)
  let dis = new Array(m)
    .fill()
    .map((_) => new Array(n).fill().map((_) => new Array(k).fill(0)));
  // keeps track of distance of nodes
  let vis = new Array(m)
    .fill()
    .map((_) => new Array(n).fill().map((_) => new Array(k).fill(false)));
  // keeps track of whether or not we visited a node
  let q = [[0, 0, 0]]; // starting at upper left corner for BFS/floodfill
  vis[0][0][0] = true;
  while (q.length > 0) {
    let [curX, curY, curK] = q.shift();
    // curX refers to current row
    // curY refers to current column
    // curK refers to current obstacles removed
    if (curX == m - 1 && curY == n - 1) {
      // check if node is in bottom right corner
      return dis[curX][curY][curK];
    }
    for (let i = 0; i < 4; i++) {
      let newX = curX + deltaX[i]; // row of destination
      let newY = curY + deltaY[i]; // column of destination
      if (newX < 0 || newX >= m || newY < 0 || newY >= n) {
        // check if it's in boundary
        continue;
      }
      let newK = curK; // obstacle count of destination
      if (grid[newX][newY] === 1) {
        newK++;
      }
      if (newK > k) {
        // surpassed obstacle removal limit
        continue;
      }
      if (vis[newX][newY][newK]) {
        // check if node has been visited before
        continue;
      }
      dis[newX][newY][newK] = dis[curX][curY][curK] + 1;
      vis[newX][newY][newK] = true;
      q.push([newX, newY, newK]);
      // process destination node
    }
  }
  return -1; // no valid answer found
};
```

Got a question? Ask the Teaching Assistant anything you don't understand.