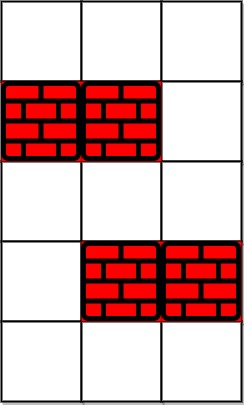
<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/>

You are given an m x n integer matrix grid where each cell is either 0 (empty) or 1 (obstacle). You can move up, down, left, or right from and to an empty cell in one step.

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 it is not possible to find such walk return -1.

**Example 1:**

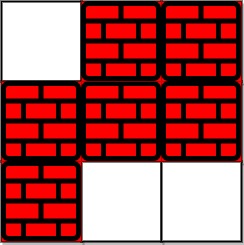


**Input:** grid = [[0,0,0],[1,1,0],[0,0,0],[0,1,1],[0,0,0]], k = 1

**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

**Attempt 1: 2024-06-02**

**Solution 1: BFS**

**Style 1: Additional 2D array to store status for deserving 're-visiting' cell (60 min)**

class Solution {

    public int shortestPath(int[][] grid, int k) {

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

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

        int m = grid.length;

        int n = grid[0].length;

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

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

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

        // Initially have k chances to eliminate obstacles

        eliminateChanceRemains[0][0] = k;

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

        visited[0][0] = true;

        int distance = 0;

        while(!q.isEmpty()) {

            int size = q.size();

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

                int[] cur = q.poll();

                int x = cur[0];

                int y = cur[1];

                int currPathRemainChance = cur[2];

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

                    return distance;

                }

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

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

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

                    // Two conditions able to add into queue for next move:

                    // 1. Not visited before

                    // 2. Visited before on other path but encounter more obstacles

                    //    than current path, which means the remains chances less

                    //    than current path, if that happened, we will replace

                    //    previous path with current path, also update the 2D array

                    //    used to record

                            || eliminateChanceRemains[new\_x][new\_y] < currPathRemainChance)) {

                        if(grid[new\_x][new\_y] == 0) {

                            // Since current cell value is 0 not consuming chance

                            eliminateChanceRemains[new\_x][new\_y] = currPathRemainChance;

                            q.offer(new int[] {new\_x, new\_y, eliminateChanceRemains[new\_x][new\_y]});

                        } else if(grid[new\_x][new\_y] == 1 && currPathRemainChance > 0) {

                            // Since current cell value is 1 consuming 1 chance

                            eliminateChanceRemains[new\_x][new\_y] = currPathRemainChance - 1;

                            q.offer(new int[] {new\_x, new\_y, eliminateChanceRemains[new\_x][new\_y]});

                        }

                        visited[new\_x][new\_y] = true;

                    }

                }

            }

            distance++;

        }

        return -1;

    }

}

**Refer to**

<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/discuss/452184/Clean-Java-BFS-with-comments.>

At every index (x,y) of the matrix along with the visited information, we also have to maintain the maximum amount of obstacles that can be eliminated(obstacleCount) and choose/update that path with highest obstacleCount.

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

public int shortestPath(int[][] grid, int k) {

int m = grid.length;

if(m==0) {

return 0;

}

int n = grid[0].length;

int[][] obstacleThatCanBeEliminated = new int[m][n]; // Number of obstacles that can be eliminated.

boolean[][] visited = new boolean[m][n];// Check if the index has been visited.

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

queue.add(new int[]{0,0,k});// x,y,obstacleCount

obstacleThatCanBeEliminated[0][0]=k; // Initially we can eliminate k obstacles.

int step=1;

while(!queue.isEmpty()) {

int size=queue.size();

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

int[] poll = queue.poll();

if(poll[0]==m-1 && poll[1]==n-1) {

return step-1;

}

int currentObstacleCount = poll[2];

for(int[] d : dir) {

int nextX = poll[0]+d[0];

int nextY = poll[1]+d[1];

/\*Add the next element to the queue if it has not been visited yet or it has

been visited but the number of obstacles encountered are greater than the current path,

hence we can replace it with the current path.\*/

(obstacleThatCanBeEliminated[nextX][nextY]<currentObstacleCount || !visited[nextX][nextY])

&& (grid[nextX][nextY]==0 || (grid[nextX][nextY]==1 && currentObstacleCount>0))

/\*currentObstacleCount>0 it means the current obstacle can also be eliminated\*/) {

if(grid[nextX][nextY]==1) {

queue.add(new int[]{nextX,nextY,currentObstacleCount-1});

obstacleThatCanBeEliminated[nextX][nextY]=currentObstacleCount-1;

} else {

queue.add(new int[]{nextX,nextY,currentObstacleCount});

obstacleThatCanBeEliminated[nextX][nextY]=currentObstacleCount;

}

visited[nextX][nextY]=true;

}

}

}

step++;

}

return -1;

}

**Refer to**

**O(m\*n\*k) BFS Solution with Explanation**

<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/solutions/451787/python-o-m-n-k-bfs-solution-with-explanation/>

**Solution Explanation**

Because we are trying to find the shortest path, use BFS here to exit immediately when a path reaches the bottom right most cell.

Use a set to keep track of already visited paths. We only need to keep track of the row, column, and the eliminate obstacle usage count. We don't need to keep track of the steps because remember we are using BFS for the shortest path. That means there is no value storing a 4th piece of the data, the current steps. This will reduce the amount of repeat work.

m = rows

n = columns

k = allowed elimination usages

**Time Complexity**

O(m**n**k) time complexity

This is because for every cell (m\*n), in the worst case we have to put that cell into the queue/bfs k times.

Runtime: 68 ms, faster than 33.33% of Python3 online submissions

**Space Complexity**

O(m**n**k) space complexity

This is because for every cell (m\*n), in the worst case we have to put that cell into the queue/bfs k times which means we need to worst case store all of those steps/paths in the visited set.

Memory Usage: 13.9 MB, less than 100.00% of Python3 online submissions

Code

from collections import deque

class Solution:

def shortestPath(self, grid: List[List[int]], k: int) -> int:

if len(grid) == 1 and len(grid[0]) == 1:

return 0

queue = deque([(0,0,k,0)])

visited = set([(0,0,k)])

if k > (len(grid)-1 + len(grid[0])-1):

return len(grid)-1 + len(grid[0])-1

while queue:

row, col, eliminate, steps = queue.popleft()

for new\_row, new\_col in [(row-1,col), (row,col+1), (row+1, col), (row, col-1)]:

if (new\_row >= 0 and

new\_row < len(grid) and

new\_col >= 0 and

new\_col < len(grid[0])):

if grid[new\_row][new\_col] == 1 and eliminate > 0 and (new\_row, new\_col, eliminate-1) not in visited:

visited.add((new\_row, new\_col, eliminate-1))

queue.append((new\_row, new\_col, eliminate-1, steps+1))

if grid[new\_row][new\_col] == 0 and (new\_row, new\_col, eliminate) not in visited:

if new\_row == len(grid)-1 and new\_col == len(grid[0])-1:

return steps+1

visited.add((new\_row, new\_col, eliminate))

queue.append((new\_row, new\_col, eliminate, steps+1))

return -1

**Some analysis of the problem:**

All the cells would be visited more than once as we could reach same cell with more distance but less obstacles which could be helpful later in traversal so we need to consider all the paths passing through same cell even with more distance.

Using PQ doesn't make sense here due to reason no 1. so we are better off with simple queue without any comparator.

Here we can also use 2D array for visited[m][n] = obstacles\_till\_here, then you will have to check if you find some cell with lesser no of obstacles reaching this point then consider that path. 3D array makes life easier in contest, but in interview its better to discuss about space constraints

**Style 2: 3D array without explicit declare new 'currPathRemainChance' (60 min)**

class Solution {

    public int shortestPath(int[][] grid, int k) {

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

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

        int m = grid.length;

        int n = grid[0].length;

        boolean[][][] visited = new boolean[m][n][k + 1];

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

        // Initially have k chances to eliminate obstacles

        q.offer(new int[] {0, 0, k});

        visited[0][0][k] = true;

        int distance = 0;

        while(!q.isEmpty()) {

            int size = q.size();

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

                int[] cur = q.poll();

                int x = cur[0];

                int y = cur[1];

                int currPathRemainChance = cur[2];

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

                    return distance;

                }

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

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

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

                        if(grid[new\_x][new\_y] == 1 && currPathRemainChance > 0

&& !visited[new\_x][new\_y][currPathRemainChance - 1]) {

                            visited[new\_x][new\_y][currPathRemainChance - 1] = true;

                            q.offer(new int[] {new\_x, new\_y, currPathRemainChance - 1});

                        }

                        if(grid[new\_x][new\_y] == 0 && !visited[new\_x][new\_y][currPathRemainChance]) {

                            visited[new\_x][new\_y][currPathRemainChance] = true;

                            q.offer(new int[] {new\_x, new\_y, currPathRemainChance});

                        }

                    }

                }

            }

            distance++;

        }

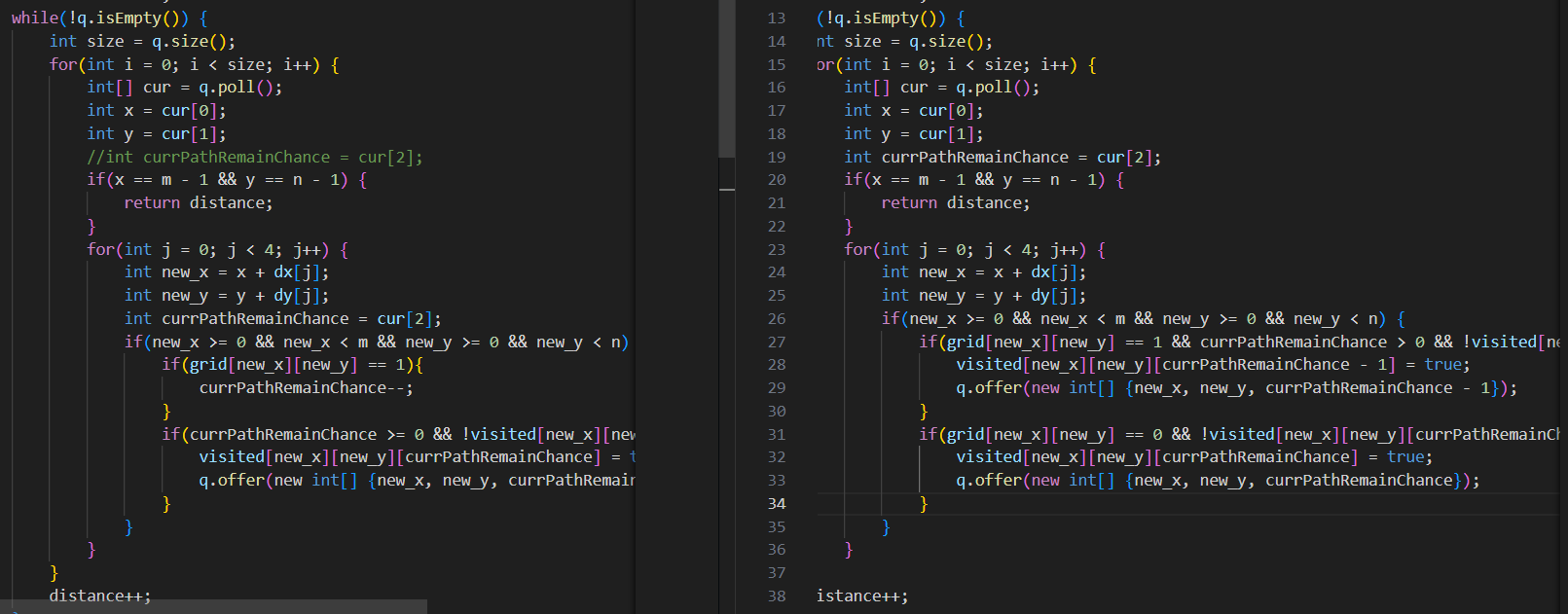
        return -1;

    }

}

**Style 3: 3D array with explicit declare new 'currPathRemainChance' (60 min)**

**与Style 2的重大不同来自于改变了 currPathRemainChance 定义的位置，必须在对4个方向开启扫描的时候分别定义1次（意味着总共初始化4次），而不是在开启扫描之前定义1次，而4次扫描中每一次 都是cur[2]，必须通过这个办法来保持每次扫描的 currPathRemainChance 不变**



class Solution {

    public int shortestPath(int[][] grid, int k) {

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

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

        int m = grid.length;

        int n = grid[0].length;

        boolean[][][] visited = new boolean[m][n][k + 1];

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

        // Initially have k chances to eliminate obstacles

        q.offer(new int[] {0, 0, k});

        visited[0][0][k] = true;

        int distance = 0;

        while(!q.isEmpty()) {

            int size = q.size();

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

                int[] cur = q.poll();

                int x = cur[0];

                int y = cur[1];

                //int currPathRemainChance = cur[2];

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

                    return distance;

                }

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

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

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

                    int currPathRemainChance = cur[2];

                        if(grid[new\_x][new\_y] == 1){

                            currPathRemainChance--;

                        }

                        if(currPathRemainChance >= 0 && !visited[new\_x][new\_y][currPathRemainChance]) {

                            visited[new\_x][new\_y][currPathRemainChance] = true;

                            q.offer(new int[] {new\_x, new\_y, currPathRemainChance});

                        }

                    }

                }

            }

            distance++;

        }

        return -1;

    }

}

**Refer to**

<https://algo.monster/liteproblems/1293>

**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.

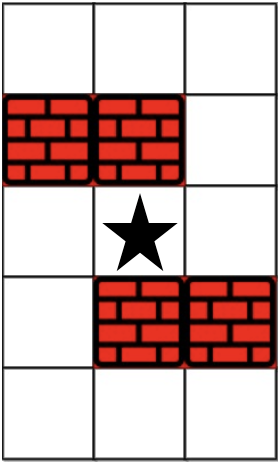
However, this is way too inefficient and complicated.

**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.

**Java solution**

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

}

}