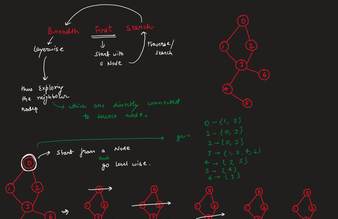
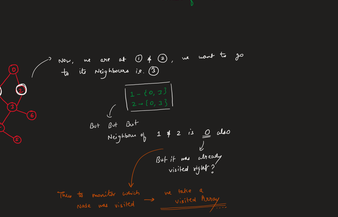
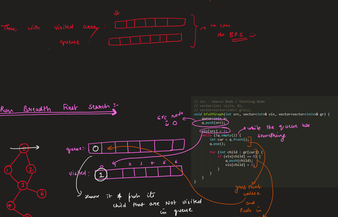
G-05 Breadth First Search (BFS) - Graph Traversal Technique - C++/Java

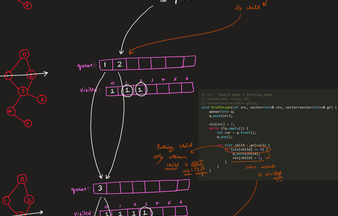


↳ Now, if we are standing at the start/root node.
we want to go to neighbours.





Dry Run Breadth First Search:



C++

```
#include<bits/stdc++.h>
using namespace std;

// vis - Source Node / Starting Node
// vector<int> vis(n, 0);
// vector<vector<int>> adj;
void bfsOfGraph(int sz, vector<int> vis, vector<vector<int>> adj) {
    queue<int> q;
    q.push(sz);
    vis[sz] = 1;
    while(!q.empty()) {
        int cur = q.front();
        q.pop();
        for (int child : adj[cur]) {
            if (!vis[child] == 0) {
                q.push(child);
                vis[child] = 1;
            }
        }
    }
}

int main()
{
    int n, m;
    cin >> n >> m;
    int in a, b;
    vector<vector<int>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> vis(n, 0);
    int sz = 0;
    cin >> sz;
    // 0 based indexing
    vis[sz] = 1;
    for (int i = 0; i < n; i++) {
        if (!vis[i] == 0) {
            bfsOfGraph(i, vis, adj);
        }
    }
}
```

→ number of edges

→ Initially undirected graph.

→ Source node

→ Initially visited array.

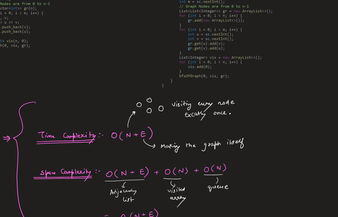
→ bfs function call for all.

Java

```
import java.util.*;

public class Main {
    // vis - Source Node / Starting Node
    // adj - Adjacency List
    // vector<int> vis(n, 0);
    public static void bfsOfGraph(int sz, List<Integer> vis, List<List<Integer>> adj) {
        Queue<Integer> q = new LinkedList();
        q.add(sz);
        vis.set(sz, 1);
        while(!q.isEmpty()) {
            int cur = q.poll();
            for (int child : adj.get(cur)) {
                if (!vis.get(child) == 0) {
                    q.add(child);
                    vis.set(child, 1);
                }
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        // Graph Nodes are from 0 to n-1
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adj.add(new ArrayList<>());
        }
        for (int i = 0; i < m; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            adj.get(u).add(v);
            adj.get(v).add(u);
        }
        List<Integer> vis = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            vis.add(0);
        }
        int sz = 0;
        sz = sc.nextInt();
        // 0 based indexing
        vis.set(sz, 1);
        for (int i = 0; i < n; i++) {
            if (!vis.get(i) == 0) {
                bfsOfGraph(i, vis, adj);
            }
        }
    }
}
```



Practice for Backlog :-

BFS problems

- Flood Fill: https://leetcode.com/problems/flood-fill/
- 01 Matrix: https://leetcode.com/problems/01-matrix/
- As Far from Land as Possible: https://leetcode.com/problems/as-far-from-land-as-possible/
- Rotting Oranges: https://leetcode.com/problems/rotting-oranges/
- Shortest Path in Binary Matrix: https://leetcode.com/problems/shortest-path-in-binary-matrix/
- Number of Islands: https://leetcode.com/problems/number-of-islands/
- Word Ladder I: https://leetcode.com/problems/word-ladder/
- Word Ladder II: https://leetcode.com/problems/word-ladder-ii/
- Evaluate Division: https://leetcode.com/problems/evaluate-division/
- Get Watched Videos by Your Friends: https://leetcode.com/problems/get-watched-videos-by-your-friends/
- Cut Off Trees for Golf Event: https://leetcode.com/problems/cut-off-trees-for-golf-event/

G 06 BFS Problem Type-1 II Simple BFS II Snakes on Ladders

1971. Find If Path Exists in Graph

Easy | 2k Lists | 79% Acc | 0

https://leetcode.com/problems/find-if-path-exists-in-graph/

909. Snakes and Ladders

Medium | 40 Lists | 39% Acc | 0

https://leetcode.com/problems/snakes-and-ladders/

909. Snakes and Ladders

Medium | 2.3k | 2.8k | 0

Companies

You are given an $n \times n$ integer matrix board where the cells are labeled from 1 to n^2 in a Boustrophedon style starting from the bottom left of the board (i.e. board[n - 1][0]) and alternating direction each row.

You start on square 1 of the board. In each move, starting from square curr, do the following:

- Choose a destination square next with a label in the range $[curr + 1, \min(curr + 6, n^2)]$.
  - This choice simulates the result of a standard **6-sided die roll**: i.e. there are always at most 6 destinations, regardless of the size of the board.
- If next has a snake or ladder, you **must** move to the destination of that snake or ladder. Otherwise, you move to next.
- The game ends when you reach the square n^2.

A board square on row r and column c about board[r][c] if it and board[r][c] > -1. The destination of that snake or ladder is board[r][c]. Squares 1 and n^2 do not have a snake or ladder.

Note that you only take a snake or ladder at most once per move. If the destination to a snake or ladder is the start of another snake or ladder, you do not follow the subsequent snake or ladder.

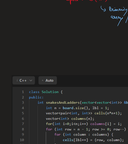
- For example, suppose the board is [[-1, 4], [-1, 3]], and on the first move, your destination square is 2. You follow the ladder to square 3, but do **not** follow the subsequent ladder to 4.

Return the least number of moves required to reach the square n^2. If it is not possible to reach the square, return -1.

Example 1:

Example 2:

Example 3:



```
C++ | Auto
class Solution {
public:
    int snakesAndLadders(vector<vector<int>> &board) {
        int n = board.size(), ble = 1;
        vector<pair<int, int>> vallocf*ort;
        vector<bool> visited;
        int ind, lc, rc, ncolumn, nrow;
        for(int i=0;i<n;i++) {
            for (int row = n - 1; row >= 0; row--) {
                for (int column = 0; column < n; column++) {
                    cells[ble++] = pair(row, column);
                }
                reverse(cells.begin(), cells.end());
            }
        }

        vector<int> dist(n*n+1, -1);
        dist[1] = 0;
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                int row, column = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};

class Solution {
public:
    int snakesAndLadders(vector<vector<int>> &board) {
        int n = board.size(), ble = 1;
        vector<pair<int, int>> cells(n*n+1);
        int ind, lc, rc, ncolumn, nrow;
        for(int i=0;i<n;i++) {
            for (int row = n - 1; row >= 0; row--) {
                for (int column = 0; column < n; column++) {
                    cells[ble++] = pair(row, column);
                }
                reverse(cells.begin(), cells.end());
            }
        }

        vector<int> dist(n*n+1, -1);
        dist[1] = 0;
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                int row, column = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};

Java | Auto
import java.util.*;

class Solution {
    int snakesAndLadders(vector<vector<int>> board) {
        int n = board.length, ble = 1;
        vector<pair<int, int>> cells(n*n+1);
        int ind, lc, rc, ncolumn, nrow;
        for(int i=0;i<n;i++) {
            for (int row = n - 1; row >= 0; row--) {
                for (int column = 0; column < n; column++) {
                    cells[ble++] = pair(row, column);
                }
                reverse(cells.begin(), cells.end());
            }
        }

        vector<int> dist(n*n+1, -1);
        dist[1] = 0;
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                int row, column = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};

import java.util.*;
class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length, ble = 1;
        List<Pair<Integer, Integer>> cells = new ArrayList<>();
        int ind, lc, rc, ncolumn, nrow;
        for (int i = 0; i < n; i++) {
            int row, column;
            for (int j = 0; j < n; j++) {
                row = (n - 1 - i);
                column = (i % 2 == 0) ? j : (n - 1 - j);
                cells.add(new Pair<>(row, column));
            }
        }

        int[] dist = new int[n * n + 1];
        Arrays.fill(dist, -1);
        dist[1] = 0;
        Queue<Integer> q = new LinkedList<>();
        q.add(1);
        while (!q.isEmpty()) {
            int curr = q.poll();
            for (int next = curr + 1; next <= Math.min(curr + 6, n * n); next++) {
                int[] rowColumn = cells.get(next);
                int row = rowColumn[0];
                int column = rowColumn[1];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.add(destination);
                }
            }
        }

        return dist[n * n];
    }
};
```

G 07 BFS Problem Type-2 II BFS on Matrix(Implicit Graph)

1091. Shortest Path in Binary Matrix

Medium 1.6K 37 Hint

https://leetcode.com/problems/shortest-path-in-binary-matrix/description/

994. Rotting Oranges

Medium 11.1K 1.4K Hint

https://leetcode.com/problems/rotting-oranges/

1091. Shortest Path in Binary Matrix

Medium 1.6K 37 Hint

Companies

Given an $n \times n$ binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

A clear path in a binary matrix is a path from the top-left cell (i.e., $(0, 0)$) to the bottom-right cell (i.e., $(n - 1, n - 1)$) such that:

- All the visited cells of the path are 0.
- All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

The length of a clear path is the number of visited cells of the path.

→ topleft

→ clear path

→ bottomright

→ 8 directional

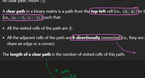
→ 1 step

→ 1 step

→ 1 step

→ shortest path length.

Example 2:



```
Input: grid = [[0,0,0],[1,1,0],[1,1,0]]
Output: 4
```

Example 3:

```
Input: grid = [[1,0,0],[1,1,0],[1,1,0]]
Output: -1
```

→ At every unit of time, I can apply to SPREAD to near to find cell [n-1][n-1]

→ Spreading is nothing but pop on queue use

→ thus, we can use B.F.S.

→ 1 step

G-08 BFS Problem Type-3 II Multi-Source BFS

1162. As Far from Land as Possible
Medium · 3.8K · 78 · 59 · 4.4

https://leetcode.com/problems/as-far-from-land-as-possible/description/

1765. Map of Highest Peak
Medium · 679 · 44

https://leetcode.com/problems/map-of-highest-peak/

Walls And Gates
Last Updated: 14 Jan, 2021
1.0M Avg Time

https://www.codingninjas.com/codestudio/problem-details/walls-and-gates_1092887 (Premium on Leetcode)

994. Rotting Oranges
Medium · 10.1K · 140 · 3.0

https://leetcode.com/problems/rotting-oranges/

G-08 BFS Problem Type-3 II Multi-Source BFS || As Far from Land as Possible || Leetcode - 1162

1162. As Far from Land as Possible Hint

Medium · 3.8K · 78

Companies

Given an $n \times n$ grid containing only values 0 and 1, where 0 represents water and 1 represents land, find a water cell such that its distance to the nearest land cell is maximized, and return the distance. If no land or water exists in the grid, return -1.

The distance used in this problem is the Manhattan distance: the distance between two cells (x_0, y_0) and (x_1, y_1) is $|x_0 - x_1| + |y_0 - y_1|$.

(x, y)

distance = $|x_0 - x_1| + |y_0 - y_1|$

Example 1:

1	0	1
0	0	0
1	0	1

Inputs: grid = [[1,0,1],[0,0,0],[1,0,1]]
Output: 2
Explanation: The cell (1, 1) is as far as possible from all the land with distance 2.

⇒ Easy to the Question: a water cell such that its distance to the nearest land cell is maximized.

distance = 1 · distance = 1 · distance = 2 · distance = 1 · distance = 1

Maximum distance = 2

⇒ [Easy to every 0 cell] = [Easy = BFS to reach to nearest land cell]

$n^2 \times n^2$

$= O(n^4)$

⇒ Optimize it :- why not start with all the land cells itself and do a BFS to reach to water cells in Nearest time.

distance = 1 → distance = 2

As we water cells are visited & to visit land water cells, it took 2 steps thus distance = 2

Example 2:

1	0	0
0	0	0
0	0	0

Inputs: grid = [[1,0,0],[0,0,0],[0,0,0]]
Output: 4
Explanation: The cell (2, 2) is as far as possible from all the land with distance 4.

distance = 1 ⇒ distance = 2 ⇒ distance = 3

water cell that is visited in the end

the maximum land distance

distance = 4

Approach

time: $O(n^2)$ → In worst case, we visit all the cells of the grid

space: $O(n^2)$ → Need to store all the cells in the queue — x — And also visited array cells.

```
class Solution {
public:
    int maxDistance(vector<vector<int>>& grid) {
        int n = grid.size();
        queue<pair<int, int>> q;
        vector<vector<int>> visited = grid;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (visited[i][j] == 1) {
                    q.push({i, j});
                }
            }
        }

        if (q.empty() || q.size() == n * n)
            return -1;
        int distance = 0;
        vector<pair<int, int>> dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
        while (!q.empty()) {
            int size = q.size();
            while (size--) {
                auto [x, y] = q.front();
                q.pop();
                for (auto [dx, dy] : dirs) {
                    int i = x + dx, j = y + dy;
                    if (i >= 0 && i < n && j >= 0 && j < n && visited[i][j] == 0) {
                        visited[i][j] = 1;
                        q.push({i, j});
                    }
                }
            }
            distance++;
        }
        return distance-1;
    }
};
```

I store the coordinates (i, j) = cell.

visited the only visit a cell once (we can use dummy grid - To check visit condition)

Rule: In all land cells in the queue & start BFS from there.

Size (Horizontally traversing)

BFS (Till in the 4-DIR step)

Easy on the water cell location

Not visited already

directions of movement

visited already

visit it if pushed in queue.

If every cell is reachable, no cell exists, no land exists.

distance = 3

```
Code | Auto
class Solution {
public:
    int maxDistance(vector<vector<int>>& grid) {
        int n = grid.size();
        queue<pair<int, int>> q;
        vector<vector<int>> visited = grid;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (visited[i][j] == 1) {
                    q.push({i, j});
                }
            }
        }

        if (q.empty() || q.size() == n * n)
            return -1;
        int distance = 0;
        vector<pair<int, int>> dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
        while (!q.empty()) {
            int size = q.size();
            while (size--) {
                auto [x, y] = q.front();
                q.pop();
                for (auto [dx, dy] : dirs) {
                    int i = x + dx, j = y + dy;
                    if (i >= 0 && i < n && j >= 0 && j < n && visited[i][j] == 0) {
                        visited[i][j] = 1;
                        q.push({i, j});
                    }
                }
            }
            distance++;
        }
        return distance-1;
    }
};
```

```
class Solution {
public:
    int maxDistance(vector<vector<int>>& grid) {
        int n = grid.size();
        queue<pair<int, int>> q;
        vector<vector<int>> visited = grid;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (visited[i][j] == 1) {
                    q.push({i, j});
                }
            }
        }

        if (q.empty() || q.size() == n * n)
            return -1;
        int distance = 0;
        vector<pair<int, int>> dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
        while (q.size()) {
            int size = q.size();
            while (size--) {
                auto [x, y] = q.front();
                q.pop();
                for (auto [dx, dy] : dirs) {
                    int i = x + dx, j = y + dy;
                    if (i >= 0 && i < n && j >= 0 && j < n && visited[i][j] == 0) {
                        visited[i][j] = 1;
                        q.push({i, j});
                    }
                }
            }
            distance++;
        }
        return distance-1;
    }
};
```

```
Java | Auto
class Solution {
    public int maxDistance(int[][] grid) {
        int n = grid.length;
        Queue<int[]> q = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) q.offer(new int[]{i, j});
            }
        }

        if (q.isEmpty() || q.size() == n * n) return -1;
        int distance = 0;
        int[][] dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
        while (q.size()) {
            int size = q.size();
            while (size-- > 0) {
                int[] cell = q.poll();
                int x = cell[0], y = cell[1];
                for (int[] dir : dirs) {
                    int i = x + dir[0], j = y + dir[1];
                    if (i >= 0 && i < n && j >= 0 && j < n && grid[i][j] == 0) {
                        grid[i][j] = 1;
                        q.offer(new int[]{i, j});
                    }
                }
            }
            distance++;
        }
        return distance-1;
    }
}
```

```
class Solution {
    public int maxDistance(int[][] grid) {
        int n = grid.length;
        Queue<int[]> q = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) q.offer(new int[]{i, j});
            }
        }

        if (q.isEmpty() || q.size() == n * n) return -1;
        int distance = 0;
        int[][] dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
        while (q.isEmpty()) {
            int size = q.size();
            while (size-- > 0) {
                int[] cell = q.poll();
                int x = cell[0], y = cell[1];
                for (int[] dir : dirs) {
                    int i = x + dir[0], j = y + dir[1];
                    if (i >= 0 && i < n && j >= 0 && j < n && grid[i][j] == 0) {
                        grid[i][j] = 1;
                        q.offer(new int[]{i, j});
                    }
                }
            }
            distance++;
        }
        return distance-1;
    }
}
```

G-09 BFS Problem Type-4 II BFS Visit More than Once

[Node can be visited more than Once]

1129. Shortest Path with Alternating Colors

Medium 2.1k 115 98 57

https://leetcode.com/problems/shortest-path-with-alternating-colors/

1293. Shortest Path in a Grid with Obstacles Elimination

Hard 4.9k 115 5

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

864. Shortest Path to Get All Keys

Hard 5.6k 271 47

https://leetcode.com/problems/shortest-path-to-get-all-keys/

1293. Shortest Path in a Grid with Obstacles Elimination

Hard 4.9k 115 5

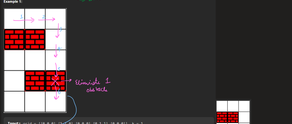
Companies

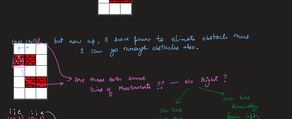
You are given an $m \times n$ integer matrix `grid`, where each cell is either 0 (empty) or 1 (obstacle). You can move up, down, left, or right from and to an empty cell in one step.

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 (3,1) is 6. Such path is (0,0) → (0,1) → (0,2) → (1,2) → (2,2) → (3,2) → (4,2).
```





```
class Solution {
public:
    int shortestPath(vector<vector<int>>& grid, int k) {
        int m = grid.size(), n = grid[0].size();
        vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(n, vector<bool>(k + 1)));
        queue<tuple<int, int, int>> q;
        int steps = 0;
        q.push({0, 0, k});
        vector<vector<int>> dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        while (!q.empty()) {
            int size = q.size();
            while (size--) {
                vector<int> cur = q.front();
                q.pop();
                if (cur[0] == m - 1 && cur[1] == n - 1) return steps;

                for (vector<int> d : dirs) {
                    int x = cur[0] + d[0];
                    int y = cur[1] + d[1];
                    int obs = cur[2];

                    if (x >= 0 && x < m && y >= 0 && y < n) {
                        if (grid[x][y] == 0 && !visited[x][y][obs]) {
                            q.push({x, y, obs});
                            visited[x][y][obs] = true;
                        } else if (grid[x][y] == 1 && obs > 0 && !visited[x][y][obs-1]) {
                            q.push({x, y, obs - 1});
                            visited[x][y][obs - 1] = true;
                        }
                    }
                }
            }
            steps++;
        }
        return -1;
};
```

```
class Solution {
public:
    int shortestPath(vector<vector<int>>& grid, int k) {
        int m = grid.size(), n = grid[0].size();
        vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(n, vector<bool>(k + 1)));
        queue<tuple<int, int, int>> q;
        int steps = 0;
        q.push({0, 0, k});
        vector<vector<int>> dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        while (!q.empty()) {
            int size = q.size();
            while (size--) {
                vector<int> cur = q.front();
                q.pop();
                if (cur[0] == m - 1 && cur[1] == n - 1) return steps;

                for (vector<int> d : dirs) {
                    int x = cur[0] + d[0];
                    int y = cur[1] + d[1];
                    int obs = cur[2];

                    if (x >= 0 && x < m && y >= 0 && y < n) {
                        if (grid[x][y] == 0 && !visited[x][y][obs]) {
                            q.push({x, y, obs});
                            visited[x][y][obs] = true;
                        } else if (grid[x][y] == 1 && obs > 0 && !visited[x][y][obs-1]) {
                            q.push({x, y, obs - 1});
                            visited[x][y][obs - 1] = true;
                        }
                    }
                }
            }
            steps++;
        }
        return -1;
};
```

```
class Solution {
    public int shortestPath(int[][] grid, int k) {

        int m = grid.length, n = grid[0].length;
        int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        boolean[][][] v = new boolean[m][n][k+1];
        Queue<int[]> q = new LinkedQueue<>();
        int steps = 0;
        q.offer(new int[]{0,0,k});
        while (!q.isEmpty()) {
            int size = q.size();
            while (size--) {
                int[] cur = q.poll();
                if (cur[0] == m - 1 && cur[1] == n - 1) return steps;
                for (int[] d : dirs) {
                    int x = cur[0] + d[0];
                    int y = cur[1] + d[1];
                    int obs = cur[2];

                    if (x >= 0 && x < m && y >= 0 && y < n) {
                        if (grid[x][y] == 0 && !v[x][y][obs]) {
                            q.offer(new int[]{x, y, obs});
                            v[x][y][obs] = true;
                        } else if (grid[x][y] == 1 && obs > 0 && !v[x][y][obs-1]) {
                            q.offer(new int[]{x, y, obs - 1});
                            v[x][y][obs - 1] = true;
                        }
                    }
                }
            }
            steps++;
        }
        return -1;
    }
}
```

```
class Solution {
    public int shortestPath(int[][] grid, int k) {

        int m = grid.length, n = grid[0].length;
        int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        boolean[][][] v = new boolean[m][n][k+1];
        Queue<int[]> q = new LinkedQueue<>();
        int steps = 0;
        q.offer(new int[]{0,0,k});
        while (!q.isEmpty()) {
            int size = q.size();
            while (size--) {
                int[] cur = q.poll();
                if (cur[0] == m - 1 && cur[1] == n - 1) return steps;
                for (int[] d : dirs) {
                    int x = cur[0] + d[0];
                    int y = cur[1] + d[1];
                    int obs = cur[2];

                    if (x >= 0 && x < m && y >= 0 && y < n) {
                        if (grid[x][y] == 0 && !v[x][y][obs]) {
                            q.offer(new int[]{x, y, obs});
                            v[x][y][obs] = true;
                        } else if (grid[x][y] == 1 && obs > 0 && !v[x][y][obs-1]) {
                            q.offer(new int[]{x, y, obs - 1});
                            v[x][y][obs - 1] = true;
                        }
                    }
                }
            }
            steps++;
        }
        return -1;
    }
}
```