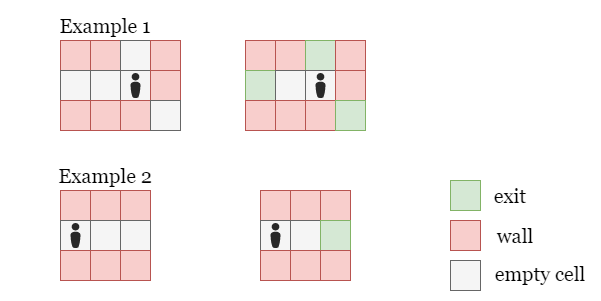
**Solution**

**Overview**

In this problem, we are given a matrix maze that represents a maze of walls and empty cells. We start from the starting point entrance and want to reach the exit of this maze.

An exit is an empty cell located at the border of maze.



As shown in the picture above, the cells colored in green are **exits** because they are empty and at the border of maze. Note that the cell (1, 0) in example 2 is not an exit because it is an **entrance**: entrance does not count as an exit.

Here our task is to find out the number of steps to the nearest exit in the given maze.

(In the first example above, we move up by 1 step and reach an exit, so the number of steps is 111, while in the second example we need to move two cells right to reach the only exit, thus the number of steps is 222.)

**Approach 1: Breadth First Search (BFS)**

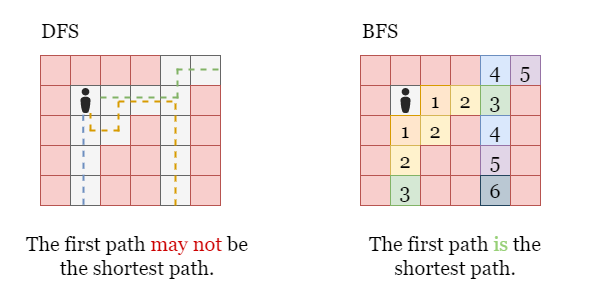
**Intuition**

This problem is about finding the shortest path in a matrix, thus Breadth First Search (BFS) is a promising method.

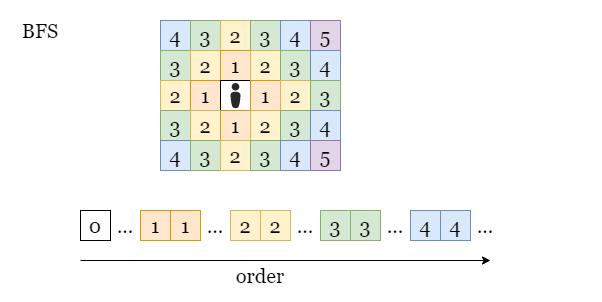
Why BFS over Depth First Search (DFS) for this problem?

The reason is that DFS is not guaranteed to find the shortest path, as it will explore the matrix as much as possible before moving on to another branch. As shown in the picture below, we may explore the matrix along the green or orange paths first, but these are not the shortest path.

In BFS, however, we explore cells by the order of their distance from the starting position, so whenever we reach an exit cell, we are guaranteed that it is the closest exit!



In BFS, we explore cells in the order of their distance from the starting position. We will first visit the cell with a distance of 000, then move on to all the cells with a distance of 111, then move on to all the cells with a distance of 222, and so forth.



We use a queue as the container to store all the cells to be visited. Since the operation on a queue is done in First In, First Out (FIFO) order, it allows us to explore all the cells with distance d which we previously stored, before moving on to cells with larger distance d + 1!

How do we prevent revisiting the same cells?

Upon finding an unvisited neighbor cell, we mark it as visited before adding it to the queue, and we skip these visited cells during further searches. Thus, each empty cell will be added to the queue at most once. (Since the input matrix maze use different characters to separate empty cells (.) and walls (+), we can take advantage of this by marking cells to be visited as +.)

Let's take a look at the following slides as an example:

1 / 14

**Algorithm**

1. Initialize an empty queue queue to store all the nodes to be visited.
2. Add entrance and its distance 000 to queue and mark entrance as visited.
3. While we don't reach an exit and queue still has cells, pop the first cell from queue. Suppose its distance from entrance is curr\_distance. We check its neighboring cells in all four directions, if it has an unvisited neighbor cell:
   * If this neighbor cell is an exit, return its distance from the starting position, curr\_distance + 1, as the nearest distance.
   * Otherwise, we mark it as visited, and add it to queue along with its distance curr\_distance + 1.
4. If we finish the iteration and no exit is found, return -1.

**Implementation**

class Solution {

public:

    int nearestExit(vector<vector<char>>& maze, vector<int>& entrance) {

        int rows = int(maze.size()), cols = int(maze[0].size());

        vector<pair<int, int>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

        // Mark the entrance as visited since its not a exit.

        int startRow = entrance[0], startCol = entrance[1];

        maze[startRow][startCol] = '+';

        // Start BFS from the entrance, and use a queue `queue` to store all

        // the cells to be visited.

        queue<array<int, 3>> queue;

        queue.push({startRow, startCol, 0});

        while (!queue.empty()) {

            auto [currRow, currCol, currDistance] = queue.front();

            queue.pop();

            // For the current cell, check its four neighbor cells.

            for (auto dir : dirs) {

                int nextRow = currRow + dir.first, nextCol = currCol + dir.second;

                // If there exists an unvisited empty neighbor:

                if (0 <= nextRow && nextRow < rows && 0 <= nextCol && nextCol < cols \

                   && maze[nextRow][nextCol] == '.') {

                    // If this empty cell is an exit, return distance + 1.

                    if (nextRow == 0 || nextRow == rows - 1 || nextCol == 0 || nextCol == cols - 1)

                        return currDistance + 1;

                    // Otherwise, add this cell to 'queue' and mark it as visited.

                    maze[nextRow][nextCol] = '+';

                    queue.push({nextRow, nextCol, currDistance + 1});

                }

            }

        }

        // If we finish iterating without finding an exit, return -1.

        return -1;

    }

};

**Complexity Analysis**

Let m,nm, n*m*,*n* be the size of the input matrix maze.

* Time complexity: O(m⋅n)O(m \cdot n)*O*(*m*⋅*n*)
  + For each visited cell, we add it to queue and pop it from queue once, which takes constant time as the operation on queue requires O(1)O(1)*O*(1) time.
  + For each cell in queue, we mark it as visited in maze, and check if it has any unvisited neighbors in all four directions. This also takes constant time.
  + In the worst-case scenario, we may have to visit O(m⋅n)O(m \cdot n)*O*(*m*⋅*n*) cells before the iteration stops.
  + To sum up, the overall time complexity is O(m⋅n)O(m \cdot n)*O*(*m*⋅*n*).
* Space complexity: O(max(m,n))O(max(m, n))*O*(*max*(*m*,*n*))
  + We modify the input matrix maze in-place to mark each visited cell, it requires constant space.
  + We use a queue queue to store the cells to be visited. In the worst-case scenario, there may be O(m+n)O(m + n)*O*(*m*+*n*) cells stored in queue.
  + The space complexity is O(m+n)+O(max(m,n))O(m + n) + O(max(m, n))*O*(*m*+*n*)+*O*(*max*(*m*,*n*)).