Problem 1926: Nearest Exit from Entrance in Maze

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given an

m x n

matrix

maze

(

0-indexed

) with empty cells (represented as

'.'

) and walls (represented as

'+'

). You are also given the

entrance

of the maze, where

entrance = [entrance

row

, entrance

col

]

denotes the row and column of the cell you are initially standing at.

In one step, you can move one cell

up

,

down

,

left

, or

right

. You cannot step into a cell with a wall, and you cannot step outside the maze. Your goal is to find the

nearest exit

from the

entrance

. An

exit

is defined as an

empty cell

that is at the

border

of the

maze

. The

entrance

does not count

as an exit.

Return

the

number of steps

in the shortest path from the

entrance

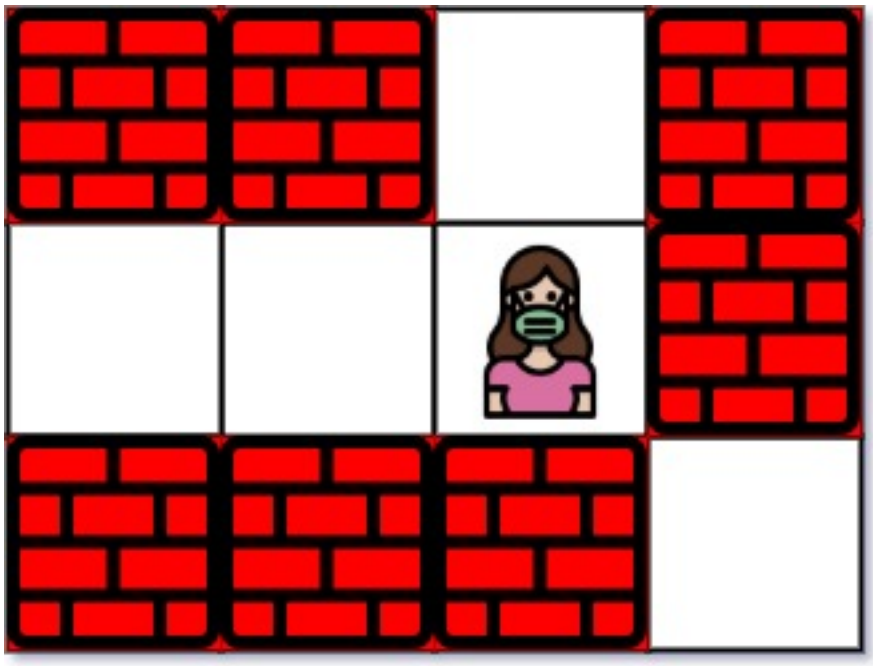
to the nearest exit, or

-1

if no such path exists

.

Example 1:



Input:

maze = [["+","+",".","+"],[".",".",".","+"],["+","+","+","."]], entrance = [1,2]

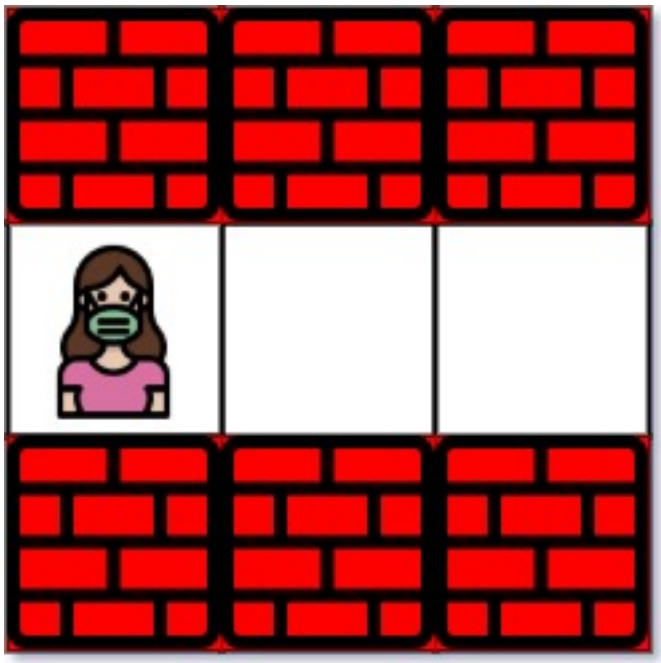
Output:

1

Explanation:

There are 3 exits in this maze at [1,0], [0,2], and [2,3]. Initially, you are at the entrance cell [1,2]. - You can reach [1,0] by moving 2 steps left. - You can reach [0,2] by moving 1 step up. It is impossible to reach [2,3] from the entrance. Thus, the nearest exit is [0,2], which is 1 step away.

Example 2:



Input:

maze = [["+","+","+"],[".",".","."],["+","+","+"]], entrance = [1,0]

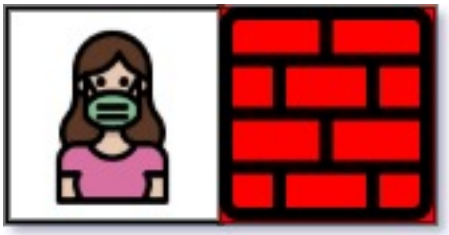
Output:

2

Explanation:

There is 1 exit in this maze at [1,2]. [1,0] does not count as an exit since it is the entrance cell. Initially, you are at the entrance cell [1,0]. - You can reach [1,2] by moving 2 steps right. Thus, the nearest exit is [1,2], which is 2 steps away.

Example 3:



Input:

maze = [["."."+"]], entrance = [0,0]

Output:

-1

Explanation:

There are no exits in this maze.

Constraints:

maze.length == m

maze[i].length == n

1 <= m, n <= 100

maze[i][j]

is either

'.'

or

'+'

.

entrance.length == 2

0 <= entrance

row

< m

0 <= entrance

col

< n

entrance

will always be an empty cell.

Code Snippets

C++:

```
class Solution {
public:
int nearestExit(vector<vector<char>>& maze, vector<int>& entrance) {

}
};
```

Java:

```
class Solution {
public int nearestExit(char[][] maze, int[] entrance) {

}
}
```

Python3:

```
class Solution:
def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:
```

Python:

```
class Solution(object):
def nearestExit(self, maze, entrance):
"""
:type maze: List[List[str]]
:type entrance: List[int]
:rtype: int
```

```
"""
```

JavaScript:

```
/**
* @param {character[][]} maze
* @param {number[]} entrance
* @return {number}
*/
var nearestExit = function(maze, entrance) {

};
```

TypeScript:

```
function nearestExit(maze: string[][], entrance: number[]): number {

};
```

C#:

```
public class Solution {
public int NearestExit(char[][] maze, int[] entrance) {

}
}
```

C:

```
int nearestExit(char** maze, int mazeSize, int* mazeColSize, int* entrance,
int entranceSize) {

}
```

Go:

```
func nearestExit(maze [][]byte, entrance []int) int {

}
```

Kotlin:

```
class Solution {
fun nearestExit(maze: Array<CharArray>, entrance: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func nearestExit(_ maze: [[Character]], _ entrance: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn nearest_exit(maze: Vec<Vec<char>>, entrance: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Character[][]} maze
# @param {Integer[]} entrance
# @return {Integer}
def nearest_exit(maze, entrance)

end
```

PHP:

```
class Solution {

/**
* @param String[][] $maze
* @param Integer[] $entrance
* @return Integer
*/
function nearestExit($maze, $entrance) {

}
```

```
}
```

Dart:

```
class Solution {
int nearestExit(List<List<String>> maze, List<int> entrance) {

}
}
```

Scala:

```
object Solution {
def nearestExit(maze: Array[Array[Char]], entrance: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec nearest_exit(maze :: [[char]], entrance :: [integer]) :: integer
def nearest_exit(maze, entrance) do

end
end
```

Erlang:

```
-spec nearest_exit(Maze :: [[char()]], Entrance :: [integer()]) -> integer().
nearest_exit(Maze, Entrance) ->
.
```

Racket:

```
(define/contract (nearest-exit maze entrance)
(-> (listof (listof char?)) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int nearestExit(vector<vector<char>>& maze, vector<int>& entrance) {

}
};
```

Java Solution:

```
/**
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int nearestExit(char[][] maze, int[] entrance) {

}
}
```

Python3 Solution:

```
"""
Problem: Nearest Exit from Entrance in Maze
Difficulty: Medium
Tags: array, graph, search
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def nearestExit(self, maze, entrance):
"""
:type maze: List[List[str]]
:type entrance: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {character[][]} maze
* @param {number[]} entrance
* @return {number}
*/
var nearestExit = function(maze, entrance) {

};
```

TypeScript Solution:

```
/**
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function nearestExit(maze: string[][], entrance: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int NearestExit(char[][] maze, int[] entrance) {

}
}
```

C Solution:

```
/*
* Problem: Nearest Exit from Entrance in Maze
* Difficulty: Medium
* Tags: array, graph, search
*
* Approach: Use two pointers or sliding window technique
```

```
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int nearestExit(char** maze, int mazeSize, int* mazeColSize, int* entrance,
int entranceSize) {

}
```

Go Solution:

```
// Problem: Nearest Exit from Entrance in Maze
// Difficulty: Medium
// Tags: array, graph, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func nearestExit(maze [][]byte, entrance []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun nearestExit(maze: Array<CharArray>, entrance: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func nearestExit(_ maze: [[Character]], _ entrance: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Nearest Exit from Entrance in Maze
// Difficulty: Medium
// Tags: array, graph, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn nearest_exit(maze: Vec<Vec<char>>, entrance: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Character[][]} maze
# @param {Integer[]} entrance
# @return {Integer}
def nearest_exit(maze, entrance)

end
```

PHP Solution:

```
class Solution {

/**
* @param String[][] $maze
* @param Integer[] $entrance
* @return Integer
*/
function nearestExit($maze, $entrance) {

}
}
```

Dart Solution:

```
class Solution {
int nearestExit(List<List<String>> maze, List<int> entrance) {
```

```
}
}
```

Scala Solution:

```
object Solution {
def nearestExit(maze: Array[Array[Char]], entrance: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec nearest_exit(maze :: [[char]], entrance :: [integer]) :: integer
def nearest_exit(maze, entrance) do

end
end
```

Erlang Solution:

```
-spec nearest_exit(Maze :: [[char()]], Entrance :: [integer()]) -> integer().
nearest_exit(Maze, Entrance) ->
.
```

Racket Solution:

```
(define/contract (nearest-exit maze entrance)
(-> (listof (listof char?)) (listof exact-integer?) exact-integer?)
)
```