1138. Alphabet Board Path

String

Since x is the row and y is the column, we get:

Problem Description

Hash Table

Medium

the board. The board is laid out such that the top left corner corresponds to "a" and letters continue in alphabetical order from left to right and then top to bottom, ending with "z" on its own row. The task is to navigate this board starting from the top left corner (0, 0) to spell out a given target word. We can move one step

In this problem, we are given a representation of an alphabet board as a list of strings, where each string corresponds to a row on

in the four cardinal directions: up (U), down (D), left (L), right (R), but only within the limits of the board. We append a letter to our output by reaching its position on the board and issuing an exclamation mark (!). We are required to find and return a sequence of moves that will result in the target word in the minimum number of moves

possible. Note that there may be multiple valid sequences that will result in the target word, and any such valid sequence is acceptable. Intuition

coordinates. The key insight is to map each character of the target to its coordinate on the board and then determine the series of moves to reach from one character to the next. The intuition behind the given solution is that for each character in the target string, we calculate its position (x, y) on the board.

To approach this problem, we should think about it as navigating a 2D grid, translating our desired string into a series of

• x as the quotient of the division of the character's index in the alphabet by the number of columns, y as the remainder of the same division.

- Move horizontally (L or R) first to get to the correct column, • Then, move vertically (U or D) to get to the correct row.
- This order of moves is important, especially when dealing with the character "z". Since "z" is located at the bottom of the board

and has no right neighbor, if we needed to go right after moving down to "z", it would be impossible. Moving horizontally first at

Once we have the target position for the current character, we execute a specific order of moves:

- other locations ensures that we never encounter a scenario where we cannot make the next move.
- this process for each character in the target string. The concatenation of all the instructions yields our result. **Solution Approach**

After moving to the correct position, we append an exclamation mark (!) to signify that we have 'typed' the character. We repeat

The solution uses a simple simulation approach with no fancy data structures or algorithms required. The key is to understand the direct correspondence between characters and their positions on the board and how to translate between characters and

1. Initialize your starting position as (0, 0), which corresponds to the top-left corner of the board, where 'a' is located.

positions.

2. For each character in the target string: Compute the character's row (x) and column (y) based on its ASCII value subtracted by the ASCII value of 'a'. Horizontal move: If the current position's column (j) is greater than the target character's column (y), add 'L' (left) moves until both columns match; else if it's less, add 'R' (right) moves. This is done to ensure we are at the correct column before adjusting the row, which is

Vertical move: Similarly, if the current position's row (i) is greater than the target character's row (x), add 'U' (up) moves until both rows

match; else if it's less, add 'D' (down) moves. This brings us to the correct row. Once at the correct position, append '!' to "type" the character.

move up

move down

append "!" to path

append "D" to path

We "type" d by appending !.

We "type" o by appending !.

We "type" g by appending !.

Solution Implementation

row, col = 0, 0

for char in target:

col -= 1

row -= 1

col += 1

answer.append("L")

answer.append("U")

answer.append("R")

path.append('D');

// Return the full path as a string

string alphabetBoardPath(string target) {

string path; // This will hold the final path sequence.

path.append("!");

return path.toString();

C++

public:

class Solution {

return path;

function alphabetBoardPath(target: string): string {

for (const character of target) {

// Move left if necessary

currentCol--;

path += 'L';

while (currentCol > targetCol) {

let path: string = ""; // This will hold the final path sequence.

let currentRow: number = 0; // Start position's row at the top-left corner of the board ('a').

const targetRow: number = Math.floor(targetPosition / 5); // Calculate the target row.

const targetCol: number = targetPosition % 5; // Calculate the target column.

let currentCol: number = 0; // Start position's column at the top-left corner of the board ('a').

const targetPosition: number = character.charCodeAt(0) - 'a'.charCodeAt(0); // Calculate the numeric position in the alp!

// Due to how 'z' is positioned, the 'L' and 'U' moves must be prioritized over 'R' and 'D' to avoid invalid moves

};

TypeScript

while row > target_row:

while col < target_col:</pre>

answer = []

class Solution:

important due to the last row having only 'z'.

3. Repeat this procedure for all characters in the target.

while (current row) > (target row):

while (current row) < (target row):</pre>

At target position, 'type' the character

The algorithm goes as follows:

The pseudocode for the part of the code that determines the movements is:

for c in target: v = ord(c) - ord('a') # The ASCII difference gives us the linear index x, y = v // 5, v % 5 # Translate linear index to 2D board coords (5 columns) # Ensure to move horizontally first to handle 'z' special case while (current col) > (target col): move left append "L" to path

append "U" to path while (current col) < (target col):</pre> move right append "R" to path

loop for each character, the answer list is joined into a string to provide the final sequence of moves.

• We need to move down ('D') 2 times to get to row 2, and then move right ('R') 1 time to get to column 4.

instructions to navigate to a 2D point from another 2D point within given constraints, ensuring that we do not get stuck in any edge cases, particularly with the isolated 'z'. **Example Walkthrough** Let's consider a target word "dog". We will walk through the sequence of moves to spell "dog" on the alphabet board. 1. The initial position is (0,0) for the character 'a'. 2. The target string is "dog":

∘ The first character is d, and its 2D board coordinates are (3 // 5, 3 % 5) = (0, 3) since 'd' is the 3rd letter ('a' being indexed at 0).

The key takeaway is that the algorithm effectively decouples the horizontal and vertical movements. It treats the problem as

The function uses a list ans to keep track of the path, appending directions as it figures out the moves required. At the end of the

o Since we cannot move directly left from z if we were there, and we're dealing with the general algorithm now, we should move up ('U') 1 time first, then move left ('L') 3 times to get to column 1 and down ('D') 1 time to get to row 1.

solution approach:

Putting it all together, the path to spell "dog" would be "RRR!DDDR!UULLD!". This is the series of moves following the described

• Start at a (initial position). • For d: move right 3 times (RRR), "type" (!).

From (0,0) we need to move right ('R') 3 times to get to (0,3).

The third character is g, with coordinates (6 // 5, 6 % 5) = (1, 1).

◦ The second character is o, with coordinates (14 // 5, 14 % 5) = (2, 4).

• For o: move down 2 times (DD), move right 1 time (R), "type" (!). • For g: move up 1 time (U), move left 3 times (LLL), move down 1 time (D), "type" (!).

the alphabet board and the isolated position of 'z'.

def alphabetBoardPath(self, target: str) -> str:

initial position on the alphabet board

target_value = ord(char) - ord('a')

Python

calculate the target's position on the 5x5 board

target_row, target_col = divmod(target_value, 5) # Since for 'z', the board needs to go all the way down before going right, # make sure to move left before moving down. while col > target_col:

This example illustrates how the algorithm navigates through each character in the target word, considering the special layout of

This part is for moving down to reach the target row, # which is placed after left and right moves to handle 'z' correctly. while row < target_row:</pre> row += 1 answer.append("D")

```
# append '!' after reaching the correct alphabet position
            answer_append("!")
       # join the list into a string to provide the path sequence
        return "".join(answer)
Java
class Solution {
    public String alphabetBoardPath(String target) {
       // StringBuilder to keep track of the path
       StringBuilder path = new StringBuilder();
       // Starting position on the board (top-left corner: 'a')
        int currentRow = 0, currentCol = 0;
       // Iterate through each character in the target string
        for (int k = 0; k < target.length(); ++k) {</pre>
            // Get the board position for the target character
            int targetPos = target.charAt(k) - 'a';
           // Calculate the row and column on the board
            int targetRow = targetPos / 5, targetCol = targetPos % 5;
            // Move left while the current column is to the right of the target column
            while (currentCol > targetCol) {
                --currentCol;
                path.append('L');
           // Move up while the current row is below the target row
           while (currentRow > targetRow) {
                --currentRow;
                path.append('U');
            // Move right while the current column is to the left of the target column
            while (currentCol < targetCol) {</pre>
                ++currentCol;
                path.append('R');
            // Move down while the current row is above the target row
           while (currentRow < targetRow) {</pre>
                ++currentRow;
```

```
for (const char &character : target) {
    int targetPosition = character - 'a'; // Calculate the numeric position in the alphabet.
    int targetRow = targetPosition / 5; // Calculate the target row.
    int targetCol = targetPosition % 5; // Calculate the target column.
    // Move left if necessary.
    while (currentCol > targetCol) {
        --currentCol;
        path += 'L';
    // Move up if necessary.
    while (currentRow > targetRow) {
        --currentRow;
        path += 'U';
    // Move right if necessary.
    while (currentCol < targetCol) {</pre>
        ++currentCol;
        path += 'R';
   // Move down if necessary.
    while (currentRow < targetRow) {</pre>
        ++currentRow;
        path += 'D';
    // Add an exclamation point to mark the arrival at the target character.
    path += '!';
// Return the completed path sequence.
```

// Add an exclamation point to indicate that the target letter is selected

int currentRow = 0, currentCol = 0; // Starting position at the top-left corner of the board ('a').

```
// Move up if necessary
          while (currentRow > targetRow) {
              currentRow--;
              path += 'U';
          // Move right if necessary
          while (currentCol < targetCol) {</pre>
              currentCol++;
              path += 'R';
          // Move down if necessary
          while (currentRow < targetRow) {</pre>
              currentRow++;
              path += 'D';
          // Add an exclamation point to mark the arrival at the target character
          path += '!';
      // Return the completed path sequence
      return path;
class Solution:
   def alphabetBoardPath(self, target: str) -> str:
        # initial position on the alphabet board
        row, col = 0, 0
        answer = []
        for char in target:
            # calculate the target's position on the 5x5 board
            target_value = ord(char) - ord('a')
            target_row, target_col = divmod(target_value, 5)
            # Since for 'z', the board needs to go all the way down before going right,
            # make sure to move left before moving down.
            while col > target_col:
                col -= 1
                answer.append("L")
            while row > target_row:
                row -= 1
                answer.append("U")
            while col < target_col:</pre>
                col += 1
                answer.append("R")
            # This part is for moving down to reach the target row,
            # which is placed after left and right moves to handle 'z' correctly.
            while row < target_row:</pre>
                row += 1
                answer.append("D")
            # append '!' after reaching the correct alphabet position
            answer_append("!")
```

The Solution provided above has a time complexity of O(n), where n is the length of the input string target. This is because the algorithm must iterate over each character in the target string once, and for each character, it performs a constant amount of

return "".join(answer)

Time and Space Complexity

join the list into a string to provide the path sequence

work: calculating x and y coordinates, then moving horizontally and vertically on the board. The space complexity of the code is O(n) as well, primarily due to the ans list that collects the sequence of moves. The length of

ans directly corresponds to the number of moves, which is proportional to the number of characters in the target string because for each character, the code appends several movements (up to 4 direction changes plus one "!" per character) to the ans list.