514. Freedom Trail

Hard | Depth-First Search | Breadth-First Search | String | Dynamic Programming | Leetcode Link

Problem Description

In this LeetCode problem, inspired by a quest in the video game Fallout 4, you are given two strings: one represents a dial (`ring`) and the other represents a keyword (`key`). The ring represents a circular dial with characters engraved on its outer edge. The initial position of the dial is such that the first character starts at the 12:00 position. Your task is to rotate the dial in either a clockwise or anticlockwise direction to spell the keyword by aligning each character of the keyword at the 12:00 position and then pressing a button to confirm each character. The goal is to find the minimum number of steps required to complete this task, where each rotation (either clockwise or anticlockwise) of one place and each press of the button count as a separate step.

When rotating the ring, you may take as many steps as necessary to bring a character to the 12:00 position, and you can choose the direction that minimizes the number of steps. Each alignment followed by a press on the center button to confirm counts as a sequence necessary to spell one character of the keyword. After spelling all characters in the key, the process completes.

Intuition

The intuition behind the solution is to use dynamic programming to keep track of the minimum steps required to align each character of the `key` at the 12:00 position by the time we reach that character in the sequence. Our state will depend on which character from the `key` we're currently looking to match and which index in the `ring` is currently at the top (at the 12:00 position).

To implement this, we initialize a 2D array where each element `f[i][j]` represents the minimum number of steps taken to spell the first `i+1` characters of the `key` with the `j`-th character of `ring` aligned at the top.

For the base case, we look at the first character in `key` and compute the minimum steps necessary to get each occurrence of this character to the 12:00 position. This step accounts for both directions of possible rotation.

For each subsequent character in `key`, we consider each position in `ring` that matches the current character we would like to align and calculate the minimum steps required to reach that position from every position that matches the previous character (which we already stored in our DP array). We account for the minimum path by considering the direct distance and the wrap-around distance (the ring is circular).

The number of steps includes the actual rotations plus one step to press the button once the correct character is aligned. We repeat this process for every character in the `key`.

Finally, the answer is the minimum value in the last row of our DP array since it represents the minimum steps required to align the last character of the `key`.

Solution Approach

The solution involves using dynamic programming (DP), a common technique for solving optimization problems where the solution can be built up by combining solutions to smaller subproblems.

1. **Data Structures**: We use a 2D array `f` for our DP table to keep track of the minimum steps needed, and a dictionary `pos` to store the indices of each character as it appears in the `ring`.
2. **Initializing the DP Table**: The DP table `f` is initialized with `inf` (infinity) to indicate that we have not yet determined the minimum steps for those positions. We have `m` rows in this table for each character in the `key`, and `n` columns for each character in the `ring`.
3. **Base Case**: We populate the first row of the DP table by calculating the minimum steps required to rotate each occurrence of the first character of the `key` to the 12:00 direction. This takes into account the minimum of a clockwise or anticlockwise rotation, which is either `j` steps (clockwise) or `n - j` steps (anticlockwise) where `j` is the index of the current character in the ring. We add 1 for pressing the button.
4. **DP Iteration**: For each subsequent character in the `key`, we iterate through DP states from the second character to the last character. For each index `j` corresponding to the current character in `key` and each index `k` corresponding to the previous character, we update `f[i][j]`, which represents the minimum steps to reach the `j-th` position of the ring to spell the `i-th` character. We minimize over two possible distances: the direct step count `abs(j - k)` and the wrap-around step count `n - abs(j - k)`. The update equation is:

   ```
   f[i][j] = min(f[i][j], f[i - 1][k] + min(abs(j - k), n - abs(j - k)) + 1)
   ```

5. **Final Answer**: After filling in the DP table, the final answer is the minimum number of steps among all the positions that can be used to spell the last character of the `key`, which is:

   ```
   min(f[-1][j] for j in pos[key[-1]])
   ```

   This represents taking the minimum steps across all possible indices where the last character of the `key` can be aligned.

The solution leverages the circular nature of the problem by using modular arithmetic and DP's overlapping subproblems property by storing interim solutions to avoid redundant calculations. By populating the DP table iteratively, an optimal solution to the full problem is efficiently built up from the solutions to smaller subproblems.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following ring and key:

```
ring = "abcde"
key = "dae"
```

In this scenario, our goal is to align each character of `key` at the 12:00 position starting with 'd', followed by 'a', and finally 'e', while counting the minimum number of steps required to do so.

Step-by-Step Walkthrough:

1. **Initialization**
   - The `ring` "abcde" has 5 characters, and the `key` "dae" has 3 characters.
   - We create a 2D array `f` with dimensions 3×5, initialized with `infinity`, since our `key` has 3 characters and our `ring` has 5 characters.
   - We create a dictionary `pos` to map each character in the `ring` to the indices where they appear. For this example:

     ```
     pos = {'a': [0], 'b': [1], 'c': [2], 'd': [3], 'e': [4]}
     ```

2. **Base Case**
   - We fill in the first row of `f` based on the first character of `key`, which is 'd'.
   - The character 'd' is at the index 3 in `ring`. To align 'd' at the top, you can rotate clockwise 3 steps, or anticlockwise 2 steps (since the ring is circular).
   - Since the anticlockwise rotation is shorter, we choose that path and set `f[0][3]` to 2 (for the rotations) + 1 (for pressing the button), thus `f[0][3] = 3`.
3. **DP Iteration**
   - Now, we consider the second character in `key`, which is 'a'.
   - The character 'a' is at the index 0 in `ring`, and we can reach it from the position of 'd' (index 3).
   - We must consider both paths: rotating clockwise 3 steps or anticlockwise 2 steps from 'd' to 'a'.
   - Since we're already at 'd', the distance is 0 in this case, so we update `f[1][0]` with the value from `f[0][3] (which is 3) + 0 (for the rotation) + 1 (for pressing the button)`, thus `f[1][0] = 4`.
4. **Continuing with DP Iteration**
   - For the last character 'e' in `key`, positioned at index 4 in `ring`, we consider the path from 'a' at index 0.
   - We can rotate 4 steps clockwise or 1 step anticlockwise.
   - The anticlockwise path is shorter, so we update `f[2][4]` with `f[1][0] + 1 (for the rotation) + 1 (for pressing the button)`, thus `f[2][4] = 6`.
5. **Final Answer**
   - The last row of array `f` now contains [inf, inf, inf, inf, 6], which represents the minimum steps required to align each position of the ring to spell the last character of `key`.
   - The final answer is the minimum of these values, which is 6, at index 4.

Using the above steps, we navigate the dial optimally to confirm the keyword "dae" with the minimum number of steps, 6 in this case.

Python Solution

```python
from collections import defaultdict
from math import inf

class Solution:

    def findRotateSteps(self, ring: str, key: str) -> int:
        # Initialize lengths for ring and key
        len_key, len_ring = len(key), len(ring)

        # Default dictionary to keep track of positions for each character
        char_positions = defaultdict(list)

        # Fill char_positions with indices for each character in ring
        for index, char in enumerate(ring):
            char_positions[char].append(index)

        # Initialize the DP array with infinity
        dp = [[inf] * len_ring for _ in range(len_key)]

        # Base case for the first character in key
        for j in char_positions[key[0]]:
            dp[0][j] = min(j, len_ring - j) + 1

        # Iterate through remaining characters in key
        for i in range(1, len_key):
            for j in char_positions[key[i]]:
                for k in char_positions[key[i - 1]]:
                    # Calculate the minimum steps to rotate from character key[i-1] to key[i]
                    # Considering the circular nature of the ring
                    dp[i][j] = min(dp[i][j], dp[i - 1][k] + min(abs(j - k), len_ring - abs(j - k)) + 1)

        # Return the minimum steps to spell the key last character
        return min(dp[-1][j] for j in char_positions[key[-1]])

# The Solution class can now be used with the findRotateSteps method to solve the problem.

```

Java Solution

```java
class Solution {
    public int findRotateSteps(String ring, String key) {
        int keyLength = key.length(); // Length of the key (sequence to spell)
        int ringLength = ring.length(); // Length of the ring
        // Create an array of lists to hold the positions of each character 'a'-'z' in the ring
        List<Integer>[] pos = new List[26];
        // Initialize each list in the array
        Arrays.setAll(pos, k -> new ArrayList<>());
        // Fill the positions lists with the indexes of each character in the ring
        for (int i = 0; i < ringLength; ++i) {
            int index = ring.charAt(i) - 'a'; // Convert char to an index 0-25
            pos[index].add(i); // Add this character's ring position to the corresponding list
        }

        // Dynamic programming matrix where f[i][j] represents the minimum steps
        // required to spell the key up to i-th character, ending at j-th position on the ring
        int[][] dp = new int[keyLength][ringLength];
        // Initialize the matrix with high values as we are looking for minimum
        for (var row : dp) {
            Arrays.fill(row, Integer.MAX_VALUE / 2); // Use Integer.MAX_VALUE / 2 to avoid overflow
        }
        // Initialize the first row of the dp matrix based on the first character of the key
        for (int j : pos[key.charAt(0) - 'a']) {
            dp[0][j] = Math.min(j, ringLength - j) + 1;
        }
        // Populate the matrix using previously calculated entries
        for (int i = 1; i < keyLength; ++i) {
            for (int j : pos[key.charAt(i) - 'a']) {
                for (int k : pos[key.charAt(i - 1) - 'a']) {
                    // The current state is the minimum of the current state and
                    // possible previous state plus the distance between k and j plus one for the button press
                    dp[i][j] = Math.min(
                        dp[i][j], dp[i - 1][k] + Math.min(Math.abs(j - k), ringLength - Math.abs(j - k)) + 1);
                }
            }
        }
        // Initialize answer to a high value to find the minimum
        int answer = Integer.MAX_VALUE / 2;
        // Iterate through the final characters positions to find the minimum steps required
        for (int j : pos[key.charAt(keyLength - 1) - 'a']) {
            answer = Math.min(answer, dp[keyLength - 1][j]);
        }
        // Return the minimum steps found
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <cstring>
#include <algorithm>

using namespace std;

class Solution {
public:
    int findRotateSteps(string ring, string key) {
        int keyLength = key.size();                 // Length of the key
        int ringLength = ring.size();               // Length of the ring
        vector<int> position[26];                   // Array of vectors to hold the positions of each character

        // Populate the position array with the indices of each character in the ring
        for (int i = 0; i < ringLength; ++i) {
            position[ring[i] - 'a'].push_back(i);
        }

        // Initialize the dynamic programming table, f, with infinity
        int dpTable[keyLength][ringLength];
        memset(dpTable, 0x3f, sizeof(dpTable));

        // Base case: fill in the first row of the dynamic programming table
        for (int index : position[key[0] - 'a']) {
            dpTable[0][index] = min(index, ringLength - index) + 1;
        }

        // Fill in the remainder of the dp table
        for (int i = 1; i < keyLength; ++i) {
            for (int j : position[key[i] - 'a']) {
                for (int k : position[key[i - 1] - 'a']) {
                    int stepDiff = min(abs(j - k), ringLength - abs(j - k)) + 1; // Calculate the minimum steps
                    dpTable[i][j] = min(dpTable[i][j], dpTable[i - 1][k] + stepDiff);
                }
            }
        }

        // Find the minimum steps needed to spell the last character of the key
        int minSteps = INT_MAX;
        for (int index : position[key[keyLength - 1] - 'a']) {
            minSteps = min(minSteps, dpTable[keyLength - 1][index]);
        }

        // Return the minimum steps to spell all characters in the key
        return minSteps;
    }
};

```

Typescript Solution

```typescript
function findRotateSteps(ring: string, key: string): number {
    const keyLength: number = key.length;          // Length of the key
    const ringLength: number = ring.length;        // Length of the ring
    const position: number[][] = Array.from({length: 26}, () => []); // Array of arrays to hold positions of each character

    // Populate the 'position' array with the indices of each character in the ring
    for (let i = 0; i < ringLength; ++i) {
        position[ring.charCodeAt(i) - 'a'.charCodeAt(0)].push(i);
    }

    // Initialize the dynamic programming table with high values
    const dpTable = Array.from({length: keyLength}, () => Array(ringLength).fill(Number.MAX_SAFE_INTEGER));

    // Base case: fill in the first row of the dynamic programming table
    for (const index of position[key.charCodeAt(0) - 'a'.charCodeAt(0)]) {
        dpTable[0][index] = Math.min(index, ringLength - index) + 1;
    }

    // Fill in the remainder of the dp table
    for (let i = 1; i < keyLength; ++i) {
        for (const j of position[key.charCodeAt(i) - 'a'.charCodeAt(0)]) {
            for (const k of position[key.charCodeAt(i - 1) - 'a'.charCodeAt(0)]) {
                const stepDiff = Math.min(Math.abs(j - k), ringLength - Math.abs(j - k)) + 1; // Calculate the minimum steps
                dpTable[i][j] = Math.min(dpTable[i][j], dpTable[i - 1][k] + stepDiff);
            }
        }
    }

    // Find the minimum steps needed to spell the last character of the key
    let minSteps = Number.MAX_SAFE_INTEGER;
    for (const index of position[key.charCodeAt(keyLength - 1) - 'a'.charCodeAt(0)]) {
        minSteps = Math.min(minSteps, dpTable[keyLength - 1][index]);
    }

    // Return the minimum steps to spell all characters in the key
    return minSteps;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this dynamic programming solution can be analyzed based on the nested loops present in the algorithm:

1. The first loop is iterating over the characters in the key string, which has a length of `m`. Therefore, it contributes O(m) to the time complexity.
2. The second and third loops are iterating over the positions of characters in the `ring` that match the current and previous characters of the `key`. In the worst case, it's possible that each character in the ring matches the key characters, hence both these loops could iterate up to `n` times, where `n` is the length of the `ring`.
3. The innermost statement that executes within the nested loops does constant work (calculating minimums and arithmetic operations), therefore each execution contributes O(1).

By multiplying these together, the overall worst-case time complexity is O(m * n^2).

Space Complexity

The space complexity is determined by the space required to store the dynamic programming table `f` and the position map `pos`:

1. The DP table `f` is an `m` by `n` matrix where `m` is the length of the `key` and `n` is the length of the `ring`. Thus, the space complexity contribution for the dynamic programming table is O(m * n).
2. The `pos` dictionary can hold at most `n` positions for each unique character in the `ring`. In the worst case, where all characters are unique, it could store `n` positions for `n` different characters. Thus, its contribution is O(n^2) in this scenario.

However, since the ring consists of characters, and even considering an extended character set, the number of unique characters would not realistically scale with `n`. Therefore, many consider the space complexity for the `pos` dictionary to be O(n) because the number of unique keys (characters) in the dictionary is a constant factor not dependent on the input size.

Taking the larger space complexity between these two, the overall space complexity is O(m * n).