514. Freedom Trail String **Dynamic Programming Breadth-First Search** Hard **Depth-First Search Leetcode Link**

Problem Description

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

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

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).

(the ring is circular).

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 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

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

can be built up by combining solutions to smaller subproblems.

is efficiently built up from the solutions to smaller subproblems.

counting the minimum number of steps required to do so.

The ring "abcde" has 5 characters, and the key "dae" has 3 characters.

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

Now, we consider the second character in key, which is 'a'.

We can rotate 4 steps clockwise or 1 step anticlockwise.

def findRotateSteps(self, ring: str, key: str) -> int:

Default dictionary to keep track of positions for each character

Considering the circular nature of the ring

int keyLength = key.length(); // Length of the key (sequence to spell)

// Fill the positions lists with the indexes of each character in the ring

int index = ring.charAt(i) - 'a'; // Convert char to an index 0-25

// Dynamic programming matrix where f[i][j] represents the minimum steps

// Initialize the matrix with high values as we are looking for minimum

// Create an array of lists to hold the positions of each character 'a'-'z' in the ring

pos[index].add(i); // Add this character's ring position to the corresponding list

Arrays.fill(row, Integer.MAX_VALUE / 2); // Use Integer.MAX_VALUE / 2 to avoid overflow

// possible previous state plus the distance between k and j plus one for the button press

dp[i][j], dp[i-1][k] + Math.min(Math.abs(j-k), ringLength-Math.abs(j-k)) + 1);

// required to spell the key up to i-th character, ending at j-th position on the ring

// Initialize the first row of the dp matrix based on the first character of the key

// The current state is the minimum of the current state and

Fill char_positions with indices for each character in ring

Initialize lengths for ring and key

char_positions = defaultdict(list)

for index, char in enumerate(ring):

for i in range(1, len_key):

char_positions[char].append(index)

dp = [[inf] * len_ring for _ in range(len_key)]

for k in char_positions[key[i - 1]]:

Return the minimum steps to spell the key last character

Initialize the DP array with infinity

for j in char_positions[key[i]]:

public int findRotateSteps(String ring, String key) {

Arrays.setAll(pos, k -> new ArrayList<>());

int[][] dp = new int[keyLength][ringLength];

for (int j : pos[key.charAt(0) - 'a']) {

for (int i = 1; i < keyLength; ++i) {</pre>

int answer = Integer.MAX_VALUE / 2;

dp[0][j] = Math.min(j, ringLength - j) + 1;

for (int j : pos[key.charAt(i) - 'a']) {

dp[i][j] = Math.min(

1 function findRotateSteps(ring: string, key: string): number {

for (let i = 0; i < ringLength; ++i) {</pre>

// Fill in the remainder of the dp table

for (let i = 1; i < keyLength; ++i) {</pre>

let minSteps = Number.MAX_SAFE_INTEGER;

const keyLength: number = key.length; // Length of the key

position[ring.charCodeAt(i) - 'a'.charCodeAt(0)].push(i);

// Base case: fill in the first row of the dynamic programming table

dpTable[0][index] = Math.min(index, ringLength - index) + 1;

for (const index of position[key.charCodeAt(0) - 'a'.charCodeAt(0)]) {

for (const j of position[key.charCodeAt(i) - 'a'.charCodeAt(0)]) {

// Find the minimum steps needed to spell the last character of the key

minSteps = Math.min(minSteps, dpTable[keyLength - 1][index]);

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

contribution for the dynamic programming table is O(m * n).

for (const k of position[key.charCodeAt(i - 1) - 'a'.charCodeAt(0)]) {

for (const index of position[key.charCodeAt(keyLength - 1) - 'a'.charCodeAt(0)]) {

dpTable[i][j] = Math.min(dpTable[i][j], dpTable[i - 1][k] + stepDiff);

// Initialize the dynamic programming table with high values

const ringLength: number = ring.length; // Length of the ring

// Populate the 'position' array with the indices of each character in the ring

// Populate the matrix using previously calculated entries

for (int k : pos[key.charAt(i - 1) - 'a']) {

// Initialize answer to a high value to find the minimum

List<Integer>[] pos = new List[26];

for (var row : dp) {

// Initialize each list in the array

for (int i = 0; i < ringLength; ++i) {</pre>

int ringLength = ring.length(); // Length of the ring

len_key, len_ring = len(key), len(ring)

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

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

first character of the key to the 12:00 direction. This takes into account the minimum of a clockwise or anticlockwise rotation,

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: 1 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: 1 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

ring = "abcde"

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

Step-by-Step Walkthrough: 1. Initialization

We create a 2D array f with dimensions 3×5, initialized with infinity, since our key has 3 characters and our ring has 5

We create a dictionary pos to map each character in the ring to the indices where they appear. For this example:

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

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).

characters.

Example Walkthrough

button), thus f[0][3] = 3.

3. DP Iteration

2. Base Case

(for the rotation) + 1 (for pressing the button), thus f[1][0] = 4. 4. Continuing with DP Iteration

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

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'.

For the last character 'e' in key, positioned at index 4 in ring, we consider the path from 'a' at index 0.

Since the anticlockwise rotation is shorter, we choose that path and set f[0][3] to 2 (for the rotations) + 1 (for pressing the

- 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, 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

from collections import defaultdict from math import inf

Python Solution

class Solution:

case.

6

9

10

11

12

13

14

15

16

17

18

21

25

26

27 28

29

30

31

32

3

4

5

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

45

46

47

Java Solution

class Solution {

19 20 # Base case for the first character in key for j in char_positions[key[0]]: 22 $dp[0][j] = min(j, len_ring - j) + 1$ 23 24 # Iterate through remaining characters in key

33 return min(dp[-1][j] for j in char_positions[key[-1]]) 34 # The Solution class can now be used with the findRotateSteps method to solve the problem. 36

 $dp[i][j] = min(dp[i][j], dp[i - 1][k] + min(abs(j - k), len_ring - abs(j - k)) + 1)$

Calculate the minimum steps to rotate from character key[i-1] to key[i]

```
39
           // Iterate through the final characters positions to find the minimum steps required
           for (int j : pos[key.charAt(keyLength - 1) - 'a']) {
40
               answer = Math.min(answer, dp[keyLength - 1][j]);
41
42
43
           // Return the minimum steps found
44
           return answer;
```

```
C++ Solution
   #include <vector>
  2 #include <string>
    #include <cstring>
     #include <algorithm>
    using namespace std;
    class Solution {
     public:
         int findRotateSteps(string ring, string key) {
 10
             int keyLength = key.size();
 11
                                                       // Length of the key
                                                       // Length of the ring
             int ringLength = ring.size();
 12
 13
             vector<int> position[26];
                                                       // Array of vectors to hold the positions of each character
 14
 15
             // Populate the position array with the indices of each character in the ring
 16
             for (int i = 0; i < ringLength; ++i) {</pre>
 17
                 position[ring[i] - 'a'].push_back(i);
 18
 19
 20
             // Initialize the dynamic programming table, f, with infinity
 21
             int dpTable[keyLength][ringLength];
 22
             memset(dpTable, 0x3f, sizeof(dpTable));
 23
 24
             // Base case: fill in the first row of the dynamic programming table
 25
             for (int index : position[key[0] - 'a']) {
 26
                 dpTable[0][index] = min(index, ringLength - index) + 1;
 27
 28
             // Fill in the remainder of the dp table
 29
             for (int i = 1; i < keyLength; ++i) {</pre>
 30
                 for (int j : position[key[i] - 'a']) {
 31
 32
                     for (int k : position[key[i - 1] - 'a']) {
 33
                         int stepDiff = min(abs(j - k), ringLength - abs(j - k)) + 1; // Calculate the minimum steps
 34
                         dpTable[i][j] = min(dpTable[i][j], dpTable[i - 1][k] + stepDiff);
 35
 36
 37
 38
 39
             // Find the minimum steps needed to spell the last character of the key
 40
             int minSteps = INT_MAX;
 41
             for (int index : position[key[keyLength - 1] - 'a']) {
 42
                 minSteps = min(minSteps, dpTable[keyLength - 1][index]);
 43
 44
 45
             // Return the minimum steps to spell all characters in the key
 46
             return minSteps;
 47
 48 };
 49
Typescript Solution
```

const position: number[][] = Array.from({length: 26}, () => []); // Array of arrays to hold positions of each character

const stepDiff = Math.min(Math.abs(j - k), ringLength - Math.abs(j - k)) + 1; // Calculate the minimum steps

const dpTable = Array.from({length: keyLength}, () => Array(ringLength).fill(Number.MAX_SAFE_INTEGER));

33 34 35 // Return the minimum steps to spell all characters in the key 36 return minSteps; 37 }

Time Complexity

complexity.

Time and Space Complexity

3

4

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

38

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).
- Space Complexity

2. The post dictionary can hold at most n positions for each unique character in the ring. In the worst case, where all characters are

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

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

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 post 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).