1165. Single-Row Keyboard

Easy Hash Table String

Problem Description

In this problem, we're given a string `keyboard` which represents the layout of a special keyboard with all the keys in a single row. The layout is a string of length `26`, containing each letter of the English alphabet exactly once. The goal is to type a given string `word` using this keyboard, knowing that our finger starts at the position of the 'a' character or index `0`.

We need to calculate the total time it takes to type the entire word, where the "time" is defined as the number of steps our finger moves from one character to the next. The time to move from one character at index `i` to another at index `j` is given by the absolute difference in their positions, `|i - j|`. The function we write must return the sum of these movements as we type out the word character by character.

Intuition

To solve this problem, we identify that the key operation is to calculate the distance between consecutive characters in the `word`. We need a fast way to look up the position of each character on the `keyboard`. A suitable data structure for this kind of look-up is a map or dictionary, where each key is a character from the keyboard and the corresponding value is its index.

Here's the step-by-step reasoning to create our solution:

1. Create a dictionary that maps each character (key) to its respective index (value) on the keyboard. This will give us O(1) access time to any character's index when we're typing the word.
2. Initialize a variable to keep track of the total time (or steps) spent moving between characters.
3. Start from the beginning position, index `0`, and iterate over each character in the given `word`.
4. For each character in the word, calculate the distance from the current index to the character's index using the absolute difference between indices.
5. Add the distance to the total time and update the current index to the character's index.
6. Return the total time after iterating through all characters in `word`.

Solution Approach

The solution uses a dictionary to store the keyboard layout mapping, efficient iteration over the input string, and simple arithmetic to calculate the total time taken to type the word. Here's how the implementation unfolds:

1. A dictionary (hash map) is created with a comprehension `pos = {c: i for i, c in enumerate(keyboard)}`. This maps each character `c` on the keyboard to its index `i`. The `enumerate()` function provides a convenient way of getting both the character and its index.
2. We use two variables in our solution: `ans` to track the total time taken, and `i` to keep track of the index of our finger's current position on the keyboard. Initially, `ans` is set to `0` and `i` to `0`, as our starting position is at the index `0`.
3. The solution then iterates over each character `c` in the `word` using a for loop: `for c in word:`. For each iteration, it performs two main operations:
   - First, it calculates the absolute difference between the current finger position and the target character's position using `abs(pos[c] - i)`. This gives us the distance or the number of steps needed to move the finger from the current position to the position of the character `c` on the keyboard.
   - Second, it adds this distance to `ans`, which accumulates the total time taken so far. After adding the distance, it updates the current position `i` to the position of the character `c`, i.e., `i = pos[c]`.
4. After iterating through all characters, the total time `ans` is returned, which gives us the answer to the problem.

There are no complex algorithms used in this solution; it primarily hinges on the efficient access of elements in a dictionary, and the computation of absolute differences between integers, which is constant-time operation. This solution is particularly efficient because each letter in `word` is looked up exactly once, resulting in O(n) time complexity, where n is the length of the `word`. The space complexity is O(1) since the dictionary holds a constant 26 key-value pairs, corresponding to the number of letters in the English alphabet.

Example Walkthrough

Let's go through the solution approach with a small example to illustrate how it works in action. Assume we have the following inputs:

- `keyboard = "pqrstuvwxyzabcdefghijklmno"`
- `word = "code"`

According to the problem, a dictionary is to be created first that will map each character of the `keyboard` to its index. Let's create this dictionary:

```
keyboard = "pqrstuvwxyzabcdefghijklmno"
# The dictionary mapping each character to its index.
pos = {c: i for i, c in enumerate(keyboard)}
# The dictionary will look something like this:
# {'p': 0, 'q': 1, 'r': 2, ..., 'n': 24, 'o': 25}
```

Now we need to type out the `word` "code". We start at index `0`, which is the position of the 'a' character according to the problem statement. But since our `keyboard` starts with 'p', 'a' is at index `15` in our mapping. Initially, both the `ans` variable (total time) and `i` variable (current position) will be set to `0`.

Let's simulate the typing:

1. To type "c", our finger moves from index `0` to index `16`. So `ans += abs(pos['c'] - i)`. This is `ans += abs(16 - 0)`, which is `16`.
   - Update `i` to `16` (position of 'c').
2. Next is "o", with `i` at `16` and 'o' at index `25`, so `ans += abs(25 - 16)`, which is `9`.
   - Update `i` to `25`.
3. Then "d", with `i` at `25` and 'd' is at index `13`, so `ans += abs(13 - 25)`, which is `12`.
   - Update `i` to `13`.
4. Lastly, we type "e", with `i` at `13` and 'e' at index `14`, so `ans += abs(14 - 13)`, which is `1`.
   - Update `i` to `14`.

Adding up all the movements, `ans = 16 + 9 + 12 + 1`, which equals `38`. Therefore, the total time taken to type the word "code" on this special keyboard would be `38` steps.

The entire walk-through we just did represents exactly what the algorithm does behind the scenes, using efficient mappings and arithmetic operations to find the total time.

Solution Implementation

Python

```
class Solution:
    def calculate_time(self, keyboard: str, word: str) -> int:
        # Create a dictionary to map each character to its index in the keyboard.
        char_to_index = {char: index for index, char in enumerate(keyboard)}

        # Initialize the total time and the starting index.
        total_time = current_index = 0

        # Iterate through each character in the word.
        for char in word:
            # Calculate the time to move from the current index to the index of the char.
            total_time += abs(char_to_index[char] - current_index)
            # Update the current index to the index of the char.
            current_index = char_to_index[char]

        # Return the total time to type the word.
        return total_time
```

Java

```
class Solution {
    public int calculateTime(String keyboard, String word) {
        // Create an array to store the index positions of each character in the keyboard
        int[] charPositions = new int[26];
        // Fill the array with the index positions
        for (int i = 0; i < 26; ++i) {
            charPositions[keyboard.charAt(i) - 'a'] = i;
        }

        // Initialize the total time taken to type the word to 0
        int totalTime = 0;
        // Set the initial position to the start of the keyboard (index 0)
        int currentPosition = 0;

        // Iterate over each character in the word
        for (int k = 0; k < word.length(); ++k) {
            // Find the index position of the current character
            int targetPosition = charPositions[word.charAt(k) - 'a'];
            // Add the distance to travel from the current position to the target position
            totalTime += Math.abs(currentPosition - targetPosition);
            // Update the current position to be the target position for the next iteration
            currentPosition = targetPosition;
        }
        // Return the total time taken to type the word
        return totalTime;
    }
}
```

C++

```
class Solution {
public:
    int calculateTime(string keyboard, string word) {
        // Create an array to hold the indices of each character in the keyboard.
        int charPositions[26];

        // Fill the array with each character's index from the 'keyboard' string.
        for (int i = 0; i < 26; ++i) {
            charPositions[keyboard[i] - 'a'] = i; // 'a' maps to 0, 'b' maps to 1, etc.
        }

        // Initialize 'totalTime' to record the total time to type the word.
        int totalTime = 0;
        // Initialize 'currentPosition' to track the current position of the finger on the keyboard, starting at index 0.
        int currentPosition = 0;

        // Loop over each character in the 'word' string to calculate the total time.
        for (char& currentChar : word) {
            // Determine the next position of the character on the keyboard.
            int nextPosition = charPositions[currentChar - 'a'];
            // Add the distance from the current position to the next position to 'totalTime'.
            totalTime += abs(currentPosition - nextPosition);
            // Update 'currentPosition' to be the next position for the following iteration.
            currentPosition = nextPosition;
        }

        // Return the calculated total time.
        return totalTime;
    }
};
```

TypeScript

```
function calculateTime(keyboard: string, word: string): number {
    // Create an array to store the position of each character in the keyboard.
    const keyPositions: number[] = new Array(26).fill(0);

    // Fill the keyPositions array with the correct positions of the characters.
    for (let index = 0; index < keyboard.length; ++index) {
        // Calculate the position based on character 'a' having charCode of 97.
        keyPositions[keyboard.charCodeAt(index) - 'a'.charCodeAt(0)] = index;
    }

    // Initialize the total time to 0.
    let totalTime = 0;

    // Initialize the current position to the starting point (0).
    let currentPosition = 0;

    // Iterate through each character in the word.
    for (const char of word) {
        // Find the target position for the current character.
        const targetPosition = keyPositions[char.charCodeAt(0) - 'a'.charCodeAt(0)];

        // Add to the total time the distance from the current position to the target position.
        totalTime += Math.abs(currentPosition - targetPosition);

        // Move the current position to the target position.
        currentPosition = targetPosition;
    }

    // Return the total time to type out the word.
    return totalTime;
}
```

```
class Solution:
    def calculate_time(self, keyboard: str, word: str) -> int:
        # Create a dictionary to map each character to its index in the keyboard.
        char_to_index = {char: index for index, char in enumerate(keyboard)}

        # Initialize the total time and the starting index.
        total_time = current_index = 0

        # Iterate through each character in the word.
        for char in word:
            # Calculate the time to move from the current index to the index of the char.
            total_time += abs(char_to_index[char] - current_index)
            # Update the current index to the index of the char.
            current_index = char_to_index[char]

        # Return the total time to type the word.
        return total_time
```

Time and Space Complexity

Time Complexity

The algorithm consists of two parts: building a dictionary with positions of characters in the `keyboard`, and then iterating over the `word` to calculate the total time.

1. Building the position dictionary has a time complexity of `O(n)`, where `n` is the length of the `keyboard` string. This is because each character in the `keyboard` string is visited once.
2. Calculating the time takes `O(m)`, where `m` is the length of the `word`. Each character in the `word` requires a constant time operation of addition and obtaining the value from the dictionary, which is in O(1).

Therefore, the overall time complexity of the `calculateTime` function is `O(n + m)`.

Space Complexity

The space complexity of the algorithm is also determined by two factors:

1. The position dictionary, which contains as many entries as there are characters in `keyboard`. This results in a space complexity of `O(n)`.
2. The variables `ans` and `i` use constant space, so they do not scale with the input size.

Hence, the total space complexity is `O(n)` due to the dictionary storing the positions of keyboard characters.