2268. Minimum Number of Keypresses

Medium Greedy Array String Counting Sorting Leetcode Link

Problem Description

In this problem, you are given the task of simulating typing on a custom keypad that is mapped to the entire set of 26 lowercase English letters. This keypad has 9 buttons, numbered from 1 to 9. The challenge lies in mapping the 26 letters onto these buttons with the following stipulations:

- All letters must be mapped to a button.
- Each letter is mapped to exactly one button.
- Each button can map to at most 3 letters.

Letters are typed by pressing a button multiple times: once for the first letter, twice for the second letter, and thrice for the third letter associated with that button. You are given a string `s`, and your task is to determine the minimum number of keypresses required to type out `s` using your keypad configuration.

For example, if the string `s` is "abcabc" and your button mapping leads to 'a', 'b', and 'c' all being on the first button, the number of keypresses would be 6: one for each letter since they are the first letter on that button.

The problem emphasizes that the specific mapping and the order of the letters on each button is fixed and cannot be changed throughout the typing process.

Intuition

The key to this problem is frequency analysis and achieving an optimal configuration where frequently used letters require fewer keypresses. Since each button can map at most 3 letters, the idea is to assign the most frequently occurring letters in the string `s` to the first slot on each button, the next set of frequent letters to the second slot, and so on. Here's the intuition:

1. Count the frequency of each letter in the string `s`.
2. Sort these frequencies in descending order to know which letters are most commonly used.
3. Assign the most frequent letters to the first slot of each button (meaning they'll only require one keypress), the next set to the second slots (two keypresses), and the least frequent to the third slots (three keypresses).
4. Calculate the total number of keypresses required based on this configuration.

Through this approach, we minimize the number of presses for frequent letters, which overall leads to a reduced total number of keypresses for typing the entire string. The provided solution uses Python's Counter class to count the frequency of each character and a sorting process to implement this idea.

Solution Approach

The solution approach involves the following steps:

1. The `Counter` class from Python's `collections` module is used to create a frequency map (`cnt`) of each character in the input string `s`. This map keeps track of how many times each character appears.

2. We initialize two variables, `ans` to keep a count of the total keypresses and `i` to keep a track of the number of buttons already assigned, and `j` to track the number of keypresses required for a given slot.

3. We sort the frequency map in reverse order, ensuring that the most frequently occurring characters are considered first.

4. We iterate through the sorted frequencies, incrementally assigning characters to the next available slot on the buttons. For each character, we multiply the character's frequency (`v`) with the current required number of keypresses (`j`) and add this to `ans`.

5. Every button can hold up to three letters, so we keep a counter `i` that increments with each character assignment. When `i % 9` is zero, it means we have filled out a complete set of three slots for all nine buttons, and it is time to move to the next set of slots (which requires an additional keypress). That's when we increment `j`.

6. After the loop is done, `ans` holds the minimum number of keypresses needed to type the string `s` using the optimal keypad configuration as per our frequency analysis.

The pattern used here is Greedy. By allocating the most frequent characters to the key positions that require fewer keypresses, we ensure that the overall cost (total number of keypresses) is minimized, which is a typical hallmark of greedy algorithms.

Below is the code snippet, illustrating how this is implemented:

```python
class Solution:
    def minimumKeypresses(self, s: str) -> int:
        cnt = Counter(s)  # Step 1: Create frequency map of characters
        ans = 0
        i, j = 0, 1
        for v in sorted(cnt.values(), reverse=True):  # Step 2 & 3: Iterate over frequencies
            i += 1
            ans += j * v  # Step 4: Calculate cumulative keypresses
            if i % 9 == 0:  # Step 5: Move to next slot and increment keypress count if needed
                j += 1
        return ans  # Step 6: Return the total keypresses
```

This solution ensures that we achieve an optimal assignment of characters to keypresses based on their frequency of occurrence in the input string `s`.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Assume our input string `s` is "hello".

Following the steps outlined:

1. We first use the `Counter` class to create a frequency map of the characters in "hello". This results in the following frequency map: `{'h': 1, 'e': 1, 'l': 2, 'o': 1}`.

2. We initialize `ans = 0` to count total keypresses, `i = 0` to track the button assignment, and `j = 1` to count the keypresses required for each letter.

3. We then sort the frequency map in descending order of frequency, which gives us `{'l': 2, 'h': 1, 'e': 1, 'o': 1}`.

4. Iterating through the sorted frequencies, we start by assigning 'l' to the first button (since 'l' has the highest frequency, 2). The total keypresses `ans` now become `j * 2 = 1 * 2 = 2`.

5. We assign the next characters 'h', 'e', and 'o' to the first slots on the next buttons. After each assignment, we increment `i` by 1. Since none make `i % 9 == 0`, there is no need to increment `j`. The keypresses for each are `j * 1 = 1 * 1 = 1`, so we add 1 for each character, making `ans = 2 + 1 + 1 + 1 = 5`.

6. At the end, `ans` is 5, which is the minimum number of keypresses needed to type "hello" using an optimal keypad configuration.

The resulting keypad map for the example could be as follows (considering only the assignments we made):

Button 1: l Button 2: h Button 3: e Button 4: o

And the total number of keypresses to type "hello" using this map is 5.

Python Solution

```python
from collections import Counter

class Solution:
    def minimumKeypresses(self, s: str) -> int:
        # Compute the frequency of each character in the string
        character_frequency = Counter(s)

        # Initialize the total number of key presses
        total_key_presses = 0

        # Initialize the keys already allocated and the current multiplier
        allocated_keys, current_multiplier = 0, 1

        # Loop through the character frequencies in descending order
        for frequency in sorted(character_frequency.values(), reverse=True):
            allocated_keys += 1  # Increment keys allocated to count this character
            total_key_presses += current_multiplier * frequency  # Add the key presses for this character

            # Every 9th character requires an additional key press (since you can only fit 9 on one screen)
            if allocated_keys % 9 == 0:
                current_multiplier += 1  # Increase the multiplier after filling a screen

        # Return the total number of key presses
        return total_key_presses

```

Java Solution

```java
class Solution {

    public int minimumKeypresses(String s) {
        // Initialize a frequency array to store occurrences of each letter
        int[] frequency = new int[26];

        // Fill the frequency array with counts of each character in the input string
        for (char character : s.toCharArray()) {
            frequency[character - 'a']++;
        }

        // Sort the frequency array in ascending order
        Arrays.sort(frequency);

        // Initialize variable to store the total number of keypresses
        int totalKeypresses = 0;

        // Initialize a variable to determine the number of keypresses per character
        int keypressesPerChar = 1;

        // Loop through the frequency array from the most frequent to the least frequent character
        for (int i = 1; i <= 26; i++) {
            // Add to the total keypress count: keypressesPerChar times the frequency of the character
            totalKeypresses += keypressesPerChar * frequency[26 - i];

            // Every 9th character will require an additional keypress
            if (i % 9 == 0) {
                keypressesPerChar++;
            }
        }

        // Return the total minimum number of keypresses needed
        return totalKeypresses;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Solution {
public:
    int minimumKeypresses(string s) {
        // Create a frequency vector to count the occurrences of each character.
        vector<int> frequencyCounter(26, 0);

        // Increment the frequency count for each character in the string.
        for (char& character : s) {
            ++frequencyCounter[character - 'a'];
        }

        // Sort the frequency vector in non-increasing order.
        sort(frequencyCounter.begin(), frequencyCounter.end(), greater<int>());

        // Initialize the answer to 0, which will hold the minimum keypresses required.
        int minimumKeyPresses = 0;

        // The number of keystrokes needed to type a character is determined by its position
        // in the sorted frequency list. The most frequent characters take 1 keystroke, the
        // next 9 take 2 keystrokes, and so on.
        int keystrokes = 1;  // Start with 1 keystroke for the most frequent characters.

        // Loop through the frequency vector to calculate the total number of keypresses.
        // The frequency array is sorted in non-increasing order, so we start from the most
        // frequent characters.
        for (int i = 0; i < 26; ++i) {
            // Calculate the keypresses required for current character frequency
            // and add it to minimumKeyPresses.
            minimumKeyPresses += keystrokes * frequencyCounter[i];

            // Every 9 characters, the number of keypresses increases by 1, since
            // we are basing our calculation off a 9-key keyboard layout.
            if ((i + 1) % 9 == 0) {
                ++keystrokes;
            }
        }

        // Return the final count of minimum keypresses needed.
        return minimumKeyPresses;
    }
};

```

Typescript Solution

```typescript
// Import statements are not needed as we are not using modules or external libraries.

// Function to calculate the minimum number of keypresses required to type a string.
function minimumKeypresses(s: string): number {
    // Create an array to count the occurrences of each character.
    const frequencyCounter: number[] = new Array(26).fill(0);

    // Increment the frequency count for each character in the string.
    for (const character of s) {
        frequencyCounter[character.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Sort the frequency array in non-increasing order.
    frequencyCounter.sort((a, b) => b - a);

    // Initialize the answer to 0, which will hold the minimum keypresses required.
    let minimumKeyPresses: number = 0;

    // The number of keystrokes needed to type a character is determined by its
    // position in the sorted frequency array. The most frequent characters
    // take 1 keystroke, the next 9 take 2 keystrokes, and so on.
    let keystrokes: number = 1; // Start with 1 keystroke for the most frequent characters.

    // Loop through the frequency array to calculate the total number of keypresses.
    for (let i: number = 0; i < 26; ++i) {
        // Calculate the keypresses required for the current character frequency
        // and add it to the minimumKeyPresses total.
        minimumKeyPresses += keystrokes * frequencyCounter[i];

        // Every 9 characters, the number of keypresses increases by 1, since
        // we are basing our calculation on a 9-key keyboard layout.
        if ((i + 1) % 9 === 0) {
            keystrokes++;
        }
    }

    // Return the final count of minimum keypresses needed.
    return minimumKeyPresses;
}

// Example of using the function:
const inputString: string = "exampleusage";
const result: number = minimumKeypresses(inputString);
console.log(result); // Logs the minimum keypresses required to type the inputString.

```

Time and Space Complexity

The given Python code aims to compute the minimum number of keypresses required to type a string `s`, where characters in the string are sorted by frequency, and each successive 9 characters require one additional keypress.

Time Complexity:

1. `cnt = Counter(s)`: Creating a counter for the string `s` has a time complexity of `O(n)`, where `n` is the length of string `s`, as we have to count the frequency of each character in the string.

2. `sorted(cnt.values(), reverse=True)`: Sorting the values of the counter has a time complexity of `O(k log k)`, where `k` is the number of distinct characters in the string `s`. In the worst case (all characters are distinct), `k` can be at most 26 for lowercase English letters, resulting in `O(26 log 26)`, which is effectively constant time, but in general, sorting is `O(k log k)`.

3. The for loop iterates over the sorted frequencies, which in the worst case is `k`. The operations inside the loop are constant time, so the loop contributes `O(k)` to the total time complexity.

The overall time complexity is therefore `O(n + k log k + k)`. Since `k` is much smaller than `n` and has an upper limit, we often consider it a constant, leading to a simplified time complexity of `O(n)`.

Space Complexity:

1. `cnt = Counter(s)`: The space complexity of storing the counter is `O(k)`, where `k` is the number of distinct characters present in `s`. As with time complexity, `k` has an upper bound of 26 for English letters, so this is effectively `O(1)` constant space.

2. The space required for the sorted list of frequencies is also `O(k)`. As before, due to the constant limit on `k`, we consider this `O(1)`.

3. The variables `ans`, `i`, and `j` occupy constant space, contributing `O(1)` to the space complexity.

Therefore, the total space complexity of the algorithm is `O(k)`, which simplifies to `O(1)` due to the constant upper bound on `k`.

Overall, the given code has a time complexity of `O(n)` and a constant space complexity of `O(1)`.