3016. Minimum Number of Pushes to Type Word II

Medium Greedy Hash Table String Counting Sorting

Problem Description

In this problem, we are given a string `word` that is comprised of lowercase English letters. The task revolves around the concept that each number key on a telephone keypad can be mapped to a group of distinct lowercase English letters, similar to old feature phones where you would press a number key multiple times to type a letter.

However, the problem allows us to 'remap' the keys numbered `2` through `9` to whatever groups of letters we want. A key point is that any letter can only map to a single key, but a key can have any number of letters assigned to it, and these collections of letters assigned to each key must be distinct from each other.

The ultimate goal is to find the most efficient way to type the given word by remapping these keys in such a way that the total number of key presses is minimized.

Let's consider an example. If the word is `"abcf"`, and if we map `"a"`, `"b"`, and `"c"` all to the number key `2`, then we'd press `2` one time to type `"a"`, two times to type `"b"`, and three times to type `"c"`, resulting in a total of `1 + 2 + 3 = 6` presses. The problem asks us to minimize such presses across the whole word by an intelligent mapping of letters to keys.

Intuition

To solve this problem, we follow a greedy algorithm approach combined with sorting. The main intuition behind the solution is that we want to minimize the number of times we press the keys to type the most frequent letters in the word. To achieve this, we analyze the frequency of each letter in the word since the more frequently a letter occurs, the more we can save by minimizing its required key presses.

Firstly, we count the occurrences of each letter using a `Counter` from the `collections` module in Python. This gives us the frequency of every distinct letter in the word.

We then sort these frequencies in descending order because we want to assign the highest frequencies to the least number of presses. We map the most frequent letters to single press keys, the next set of most frequent letters to two press keys, and so on. Since there are `8` number keys (`2` to `9`) available to map, we group every `8` letters together, assigning each group consecutively higher number of key presses.

To calculate the number of presses for the i-th most frequent letter, we divide its rank `i` (starting from `0`) by `8` because there are eight keys that the letters could be assigned to. We then take the integer division result (which represents the group number), add `1` to get the actual number of presses for the group, and then multiply it by the frequency count of the letter. Summing this value across all the letters gives us the total number of key presses required after an optimal remapping of keys.

By following this strategy, we ensure that the most frequent letters in the word are the quickest to type, thus minimizing the total number of presses.

Solution Approach

The implementation of the provided solution uses a combination of a greedy algorithm, sorting, and a counting mechanism to solve the problem efficiently.

Here's a step-by-step breakdown of the implementation process:

1. **Counting the Letter Frequencies**:
   - The solution begins by using a `Counter` from Python's `collections` module to count the frequency of each letter within the `word`.
   - The `Counter` data structure automatically creates a hash table (dictionary-like structure) where the keys are the distinct letters, and the values are the counts of how many times each letter appears in the `word`.
2. **Sorting the Frequencies in Descending Order**:
   - This next step is to sort the counted letter frequencies in descending order.
   - This is done because we want to allocate the most frequently occurring letters to the least number of key presses, following the greedy strategy.
   - Sorting is typically achieved through comparison-based algorithms like quicksort or mergesort, which ensure an efficient sorting process—most Python implementations use a variant of Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort.
3. **Assigning Letters to Keys**:
   - Once sorted, the letters are virtually grouped into sets of 8 because we have 8 keys (`2` to `9`) that can be remapped to any set of letters.
   - Starting from the most frequent letter (at the beginning of the sorted count values), each letter is assigned a number of key presses that corresponds to the key it would be mapped to.
   - The calculation for the number of presses required for the `i`-th letter (starting index at `0`) is:
     - Divide the letter's rank `i` by `8` since we have 8 keys, take the integer quotient which represents the group number.
     - Add `1` to translate the group number to the number of presses (since group `0` would be a single press).
     - Multiply this result by the frequency of the letter, to find out the total presses needed for that letter.
   - This is repeated for each letter using a loop.
4. **Calculating the Total Minimum Presses**:
   - As the last step, each individual letter's presses are added together to yield the total minimum number of key presses required to type the entire `word` after the optimal remapping.

Using this approach, the `Solution` class's `minimumPushes` method ensures we arrive at the minimum number of pushes needed to type the word after remapping the keys. The algorithm is efficient because it leverages frequency counts and sorting to minimize the total number of presses in a very direct way, well-suited to the constraints of the problem.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose our given `word` is `"aaaaaaaaaabbbccc"`.

Step 1: Counting the Letter Frequencies: We use the `Counter` to count the frequency of each letter.

- `a`: 11 times
- `b`: 3 times
- `c`: 3 times

Step 2: Sorting the Frequencies in Descending Order: We sort the letter frequencies:

- `a`: 11 times
- `b`: 3 times
- `c`: 3 times

Since `a` is the most frequent, followed by `b` and `c` (which have equal frequency), we keep this order.

Step 3: Assigning Letters to Keys: We have 8 keys, so let's distribute the letters.

- The letter `a` gets the first key (Let's assume `2`), which needs just 1 press per `a`. Since `a` appears 11 times, we have a total of `11 * 1 = 11` presses for `a`.
- The letters `b` and `c` are next in frequency, and they get the second key (Let's assume `3`). `b` requires 2 presses, and we have 3 `b`s making `3 * 2 = 6` presses for `b`. The same calculation applies to `c`, resulting in another `6` presses.

Step 4: Calculating the Total Minimum Presses: We add up the total number of presses:

- `a` contribution: 11 presses
- `b` contribution: 6 presses
- `c` contribution: 6 presses

So the total minimum number of key presses is `11 + 6 + 6 = 23` presses.

Therefore, by remapping the keys as described, it takes a minimum of 23 presses to type `"aaaaaaaaaabbbccc"` using a telephone keyboard.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def minimumPushes(self, word: str) -> int:
        # Count the occurrences of each character in the string
        char_count = Counter(word)

        # Initialize the variable to store the minimum pushes
        minimum_pushes = 0

        # Sort the character counts in descending order
        sorted_counts = sorted(char_count.values(), reverse=True)

        # Calculate the minimum pushes needed by iterating over the sorted counts
        for index, count in enumerate(sorted_counts):
            # Every 8 characters can be grouped and pushed together.
            # so we divide the index by 8 and add 1 to find the number of pushes.
            # Multiply by the count of each character to find total pushes.
            minimum_pushes += ((index // 8) + 1) * count

        # Return the total minimum pushes calculated
        return minimum_pushes

# Example usage:
solution = Solution()
result = solution.minimumPushes("examplewrd")
print(result)  # It will print the minimum number of pushes for "examplewrd"
```

Java

```java
import java.util.Arrays; // Required for using Arrays.sort()

class Solution {
    public int minimumPushes(String word) {
        // Create an array to hold the frequency of each letter.
        int[] letterCounts = new int[26];

        // Count the frequency of each letter in the word.
        for (int i = 0; i < word.length(); ++i) {
            // Increment the count for the current letter.
            ++letterCounts[word.charAt(i) - 'a'];
        }

        // Sort the frequencies in ascending order.
        Arrays.sort(letterCounts);

        // Initialize the number of pushes to 0.
        int pushesRequired = 0;

        // Calculate the minimum pushes required.
        // Iterate over the sorted counts in reverse order (highest to lowest frequency).
        for (int i = 0; i < 26; ++i) {
            // The formula determines the number of pushes for each letter
            // taking into account the increasing number of button push multiplicities
            // as you move to different sets of 8 buttons.
            // The index for the counts is adjusted to start from the highest frequency.
            pushesRequired += (i / 8 + 1) * letterCounts[26 - i - 1];
        }

        // Return the total number of pushes required to type the word.
        return pushesRequired;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum number of pushes required.
    int minimumPushes(string word) {
        // Create a vector to store frequency of each letter in the 'word'.
        vector<int> frequency(26, 0);
        // Increment the frequency count for each letter in the 'word'.
        for (char& c : word) {
            ++frequency[c - 'a'];
        }

        // Sort the frequency vector in descending order to prioritize letters with a higher frequency.
        sort(frequency.rbegin(), frequency.rend());

        // Initialize a variable to store the total number of pushes.
        int totalPushes = 0;
        // Loop through the frequency vector to calculate the pushes for each letter.
        for (int i = 0; i < 26; ++i) {
            // The number of times a button needs to be pushed is determined by its position in the frequency vector.
            // (i / 8 + 1) gives the tier of the button (1st-8th, 9th-16th, etc.),
            // Multiply the tier by the frequency of the letter to get the pushes for that letter.
            totalPushes += (i / 8 + 1) * frequency[i];
        }

        // Return the total number of pushes required.
        return totalPushes;
    }
};
```

TypeScript

```typescript
function minimumPushes(word: string): number {
    // Initialize a count array of size 26 to count frequency of each alphabet letter
    const count: number[] = Array(26).fill(0);

    // Loop through each character in the word to populate the frequency in the count array
    for (const char of word) {
        ++count[char.charCodeAt(0) - 'a'.charCodeAt(0)];
    }

    // Sort the count array in descending order
    count.sort((a, b) => b - a);

    // Initialize an accumulator for the minimum number of pushes
    let minimumPushes = 0;

    // Iterate over the count array to calculate the total pushes
    for (let i = 0; i < 26; ++i) {
        // The pushes required is determined by the position of the letter
        // and its frequency in the word. The position is affected by which
        // group of 8 letters it is in, which is calculated by i/8 and rounded down.
        // The first group (i/8 = 0) incurs a multiplier of 1, the second a multiplier of 2, etc.
        minimumPushes += (((i / 8) | 0) + 1) * count[i];
    }

    // Return the calculated minimum number of pushes
    return minimumPushes;
}
```

```python
from collections import Counter

class Solution:
    def minimumPushes(self, word: str) -> int:
        # Count the occurrences of each character in the string
        char_count = Counter(word)

        # Initialize the variable to store the minimum pushes
        minimum_pushes = 0

        # Sort the character counts in descending order
        sorted_counts = sorted(char_count.values(), reverse=True)

        # Calculate the minimum pushes needed by iterating over the sorted counts
        for index, count in enumerate(sorted_counts):
            # Every 8 characters can be grouped and pushed together.
            # so we divide the index by 8 and add 1 to find the number of pushes.
            # Multiply by the count of each character to find total pushes.
            minimum_pushes += ((index // 8) + 1) * count

        # Return the total minimum pushes calculated
        return minimum_pushes

# Example usage:
solution = Solution()
result = solution.minimumPushes("examplewrd")
print(result)  # It will print the minimum number of pushes for "examplewrd"
```

Time and Space Complexity

The provided code snippet aims to calculate the minimum number of pushes required for a given word. The time and space complexity analysis is as follows:

Time Complexity

The time complexity of the code consists of two main operations:

1. **Counter Construction**: Constructing the counter object `cnt` from the string `word`. This operation has a time complexity of $O(n)$ where n is the length of the input string `word`. This is because each character in the string must be visited once to build the frequency count.
2. **Sorting and Iteration**: The second operation is to sort the values obtained from the counter and then to iteratively calculate the total number of pushes. The sorting operation has a time complexity of $O(|\Sigma| * \log|\Sigma|)$, where $|\Sigma|$ represents the size of the set of unique characters in the string. The iteration over the sorted counts has a complexity of $O(|\Sigma|)$ since each element is visited once.

Therefore, the combined time complexity is $O(n + |\Sigma| * \log|\Sigma|)$.

Space Complexity

The space complexity of the code is governed by the additional space used by the data structures within the function:

1. **Counter Object**: The counter object `cnt` stores the frequency of each character in the string which requires $O(|\Sigma|)$ space, where $|\Sigma|$ is the number of unique characters in `word`.

Summing up the space complexities, the total space complexity is $O(|\Sigma|)$.