3014. Minimum Number of Pushes to Type Word I

Easy Greedy Math String

Problem Description

In this problem, you are given a string `word` that has distinct lowercase English letters. Imagine you have a telephone keypad where each key can be used to type certain letters. Normally, keys 2 to 9 are associated with letters such as key 2 with 'a', 'b', 'c', key 3 with 'd', 'e', 'f', and so on. To type a letter, you need to press a key certain times. For example, to type 'b', you press key 2 twice.

However, in this scenario, you have the ability to remap these keys, meaning you can assign any collection of letters to any key, given that each letter is mapped to exactly one key. Your objective is to find a mapping that allows you to type the given `word` with the fewest number of key presses.

You need to return the minimum number of key presses required to type the string `word` after optimally remapping the keys.

Intuition

To solve this problem, we can use a greedy algorithm. Since all letters in the `word` are distinct and we can remap keys in any way we want, the most efficient approach is to distribute the letters across the keys as evenly as possible. Why? Because this ensures that we use the minimum number of presses per key.

With 8 keys available (keys 2 to 9), the first key will be pressed once for its first letter, twice for its second letter, and so on. By distributing letters evenly, we're minimizing the maximum number of presses for any key.

If the `word` is shorter than or equal to 8 letters, it can be mapped in such a way that each letter is assigned to a different key, and each key is pressed only once. If the word is longer, we put 8 letters on the first 8 keys; for the next 8 letters, we would need to press each key twice, and so forth. This way, the minimum number of total presses is achieved.

The provided solution calculates the number of presses by adding 8 presses for each complete set of 8 letters and then adding the remaining letters multiplied by the number of letter sets plus one.

Solution Approach

The solution provided does not require any complex data structures or advanced patterns due to the simplicity of its greedy algorithm. It focuses on the fact that all letters in the `word` are unique and can hence be distributed evenly across the 8 keys available.

The algorithm iterates through the string `word` and calculates the number of times keys need to be pressed in groups of 8 letters. To explain in greater detail:

1. The variable `ans` starts at `0` and is used to accumulate the total number of key presses.
2. The variable `k` represents the sequence index within the keys; that is, for the first set of up to 8 letters, `k` is `1`, as each key only needs to be pressed once. For the next set of 8 letters (if there are more than 8 unique letters), `k` would be `2`, because we need to press each key twice to access the second letter in each key, and so on.
3. The for-loop runs for the count of full groups of 8 letters (`n // 8`) in the word. For each group, it adds to `ans` the product of `k` (the current group index) and `8` (the number of letters in the group). Then it increments `k` to represent the next sequence for the next group of letters.
4. After processing all full groups of 8, there might be some leftover letters (less than 8). The algorithm adds to the `ans` the number of leftovers (`n % 8`) multiplied by the current `k` value.
5. Finally, the calculated `ans` value, which represents the total number of key presses, is returned.

To illustrate with an example, if `word` is `abcdefghijklmnop`, the first 8 letters (`abcdefgh`) would require one press each (total of 8 presses). The next 8 letters (`ijklmnop`) are the second set and would require two presses each (total of 2 * 8 = 16 presses). So the solution would be `8 + 16 = 24` presses in total.

This completes the explanation of the solution's implementation. It's a straightforward calculation that relies on the rules of evenly distributing unique letters over a fixed number of keys and optimizing for the fewest total presses.

Example Walkthrough

Let's go through a sample problem to illustrate the solution approach using the word: `program`.

In this scenario, the word `program` has 7 unique letters. According to the rules of the telephone keypad, we want to remap these letters to minimize the number of key presses.

Since we have 8 keys (2 to 9) and only 7 unique letters, we can assign each letter to a different key because the number of letters is less than or equal to the number of keys. Here is how we can remap the letters:

- Key 2: 'p' (1 press)
- Key 3: 'r' (1 press)
- Key 4: 'o' (1 press)
- Key 5: 'g' (1 press)
- Key 6: 'r' (1 press)
- Key 7: 'a' (1 press)
- Key 8: 'm' (1 press)

Since each letter is assigned to its own key, we only need to press each key once to type the word `program`. This means we only need a total of 7 presses - one press for each letter.

To summarize the steps in the algorithm:

1. We initialize `ans` to 0.
2. We note that `k = 1` because we have not yet exceeded 8 unique letters.
3. There are no full groups of 8, so the for-loop does not add anything to `ans`.
4. We then calculate the number of leftover letters, which is `7 % 8 = 7`, multiply by `k` (which is 1), and add the result to `ans`.
5. Finally, we have `ans = 0 + 7*1 = 7`. Therefore, the minimum number of key presses required is 7.

This simple example shows how the provided solution uses the rules of a remappable keypad and a greedy distribution of letters to keys to minimize the number of key presses required to type the word.

Solution Implementation

Python

```python
class Solution:
    def minimumPushes(self, word: str) -> int:
        # Calculate the length of the word
        word_length = len(word)
        # Initialize 'total_pushes' to store the total pushes required
        # Initialize 'pushes_per_batch' which represents the number of pushes for each batch of 8 characters
        total_pushes, pushes_per_batch = 0, 1

        # Process all complete batches of 8 characters
        for _ in range(word_length // 8):
            total_pushes += pushes_per_batch * 8  # Add the pushes required for a batch of 8 characters
            pushes_per_batch += 1  # Increment the number of pushes needed for each subsequent batch

        # Add the pushes required for the remaining characters, if any
        total_pushes += pushes_per_batch * (word_length % 8)

        # Return the total number of pushes required
        return total_pushes
```

Java

```java
class Solution {
    public int minimumPushes(String word) {
        int wordLength = word.length(); // Length of the string word
        int totalPushes = 0; // Total number of pushes required
        int pushMultiplier = 1; // Multiplier for each group of 8 characters

        // Loop through the groups of 8 characters in the word
        for (int i = 0; i < wordLength / 8; ++i) {
            totalPushes += pushMultiplier * 8; // Add pushes for this group
            ++pushMultiplier; // Increment the multiplier for the next group
        }

        // Add pushes for the remaining characters (if any)
        totalPushes += pushMultiplier * (wordLength % 8);

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

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum number of pushes needed
    int minimumPushes(string word) {
        int length = word.size();  // Get the length of the input string
        int answer = 0;            // Initialize the answer to zero
        int multiplier = 1;        // Initialize the multiplier to one

        // Process complete sets of 8 characters
        for (int i = 0; i < length / 8; ++i) {
            answer += multiplier * 8;  // Add 8 times the current multiplier to the answer
            ++multiplier;              // Increment the multiplier for the next set
        }

        // Add the remaining characters (less than 8 if any)
        answer += multiplier * (length % 8);

        // Return the total minimum pushes calculated
        return answer;
    }
};
```

TypeScript

```typescript
/**
 * Calculate the minimum number of pushes required based on the "word" string's length.
 * Each set of 8 characters requires an increasing number of pushes.
 *
 * @param {string} word - The string to be analyzed for push calculations.
 * @return {number} - The minimum number of pushes required.
 */
function minimumPushes(word: string): number {
    // Get the length of the input string.
    const wordLength = word.length;
    // Initialize the answer to be returned with 0.
    let totalPushes = 0;
    // Start with k = 1, which will be multiplied with the character count.
    let multiplier = 1;

    // Loop through the string 8 characters at a time.
    for (let i = 0; i < wordLength / 8) | 0; ++i) {
        totalPushes += multiplier * 8; // Add 8 times the current multiplier to the total.
        ++multiplier; // Increment the multiplier for the next set of characters.
    }

    // Add the remaining characters multiplied by the current multiplier
    // for the case where word length is not a multiple of 8.
    totalPushes += multiplier * (wordLength % 8);

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

```python
class Solution:
    def minimumPushes(self, word: str) -> int:
        # Calculate the length of the word
        word_length = len(word)
        # Initialize 'total_pushes' to store the total pushes required
        # Initialize 'pushes_per_batch' which represents the number of pushes for each batch of 8 characters
        total_pushes, pushes_per_batch = 0, 1

        # Process all complete batches of 8 characters
        for _ in range(word_length // 8):
            total_pushes += pushes_per_batch * 8  # Add the pushes required for a batch of 8 characters
            pushes_per_batch += 1  # Increment the number of pushes needed for each subsequent batch

        # Add the pushes required for the remaining characters, if any
        total_pushes += pushes_per_batch * (word_length % 8)

        # Return the total number of pushes required
        return total_pushes
```

Time and Space Complexity

The time complexity of the given code is `O(1)`, also described as constant time complexity. The reason for this is that the loop runs a fixed number of times that depends on the length of the string `word` divided by `8`. Since division is a constant-time operation, and the loop consequently runs a constant number of iterations (specifically, `n // 8` times), the complexity does not grow with the size of the input but remains capped. The modifications inside the loop are also constant-time operations, resulting in an overall constant time complexity.

The space complexity of the code is `O(1)`, meaning it requires a constant amount of additional space that does not grow with the size of the input. This is because the variables `n`, `ans`, `k`, and `_` each occupy a fixed amount of space, and no additional space that scales with the input size (`word`) is required or allocated during the execution of the code.