1189. Maximum Number of Balloons

Easy Hash Table String Counting

Leetcode Link

Problem Description

The problem provides us with a string named `text` and expects us to find out how many instances of the word "balloon" can be formed using the characters from `text`. Each character in `text` can be used only once, therefore it controls how many times the word "balloon" can be assembled. One thing to note is that different characters are needed in different quantities - specifically, 'b', 'a', and 'n' appear once in "balloon", whereas 'l' and 'o' appear twice. The goal is to find the maximum number of complete "balloon" words that can be created without reusing characters.

Intuition

To determine how many times the word "balloon" can be formed, we need to count the occurrences of each character that is required to form the word within `text`. Bearing in mind that 'l' and 'o' are needed twice in each instance of "balloon", we must count how many times 'b', 'a', 'l', 'o', and 'n' appear in `text`. The approach is straightforward:

1. Count the frequency of each character in the input `text`.
2. Since 'l' and 'o' are required twice, we divide the counted occurrences of 'l' and 'o' by 2. This adjusts their counts to reflect how many full "balloon" instances they can contribute to.
3. The number of possible "balloon" instances will then be limited by the character with the least number of usable occurrences after adjustments, as every "balloon" needs at least one of each required character.
4. We find the minimum occurrence among the character counts for 'b', 'a', 'l', 'o', and 'n' — this gives us the maximum number of "balloon" instances that can be formed.

Using a `Counter` to tally the characters and performing integer division (with the `>>` operator which is equivalent to dividing by 2 and flooring the result) on counts of 'l' and 'o' makes the solution efficient and allows us to simply use the `min` function to get our answer.

Solution Approach

The implementation of the solution can be understood step by step as follows:

1. **Using `Counter` from `collections` module:** Python's `Counter` is a container that stores elements as dictionary keys, and their counts are stored as dictionary values. Instantiating a `Counter` object with the string `text` gives us a dictionary-like object that contains all characters as keys with their respective counts as values. This is beneficial as `Counter` efficiently counts the occurrences of each character in the input string `text`.

2. **Adjusting counts for 'l' and 'o':** Since the word "balloon" contains two 'l' and two 'o' characters, we need to know how many pairs of 'l' and 'o' we have. This is done by right-shifting the count of these characters by 1 position with `cnt['o'] >>= 1` and `cnt['l'] >>= 1`. The right shift `>>` operator is equivalent to performing integer division by 2 and then flooring the result, which is exactly what we need to get the pair counts for 'l' and 'o'.

3. **Finding the limiting character count:** The `min` function is used to iterate over the counts of 'b', 'a', 'l', 'o', and 'n' in the `Counter` object, and to find the smallest count among them. This is because we can only form as many instances of "balloon" as the least common necessary character allows. For example, if we only have one 'b', we cannot form more than one "balloon", regardless of how many 'l's or 'o's we have.

4. **Returning the result:** Finally, the minimum count found is returned which represents the maximum number of times the word "balloon" can be formed from the given `text`.

The algorithm is efficient primarily due to the `Counter` data structure, which allows O(n) tallying of characters in `text`. Further operations (bitwise shifts and minimum value calculation) are O(1) with respect to the distinct characters involved since the word "balloon" has a constant number of unique characters.

Overall, this implementation is concise and leverages Python's built-in features to solve the problem with minimal lines of code and in efficient time complexity.

Example Walkthrough

Let's say our input string `text` is "loonbalxballpoon".

1. **Count characters in `text`:**
   - We instantiate a `Counter` with our `text`:
     - Counter will be: `{'l': 3, 'o': 4, 'n': 2, 'b': 2, 'a': 2, 'x': 1, 'p': 1}`
   - In "balloon", 'b' appears 1 time, 'a' appears 1 time, 'l' appears 2 times, 'o' appears 2 times, and 'n' appears 1 time.
2. **Adjust counts for 'l' and 'o':**
   - We right-shift the counts for 'l' and 'o' by 1 (integer division by 2):
     - After adjustment, 'l' count is 3 >> 1 = 1 and 'o' count is 4 >> 1 = 2 (>> is bit shift to the right which is equivalent to dividing by 2 and flooring the result).
3. **Find the limiting character count:**
   - Using the counts we have for 'b', 'a', 'l', 'o', and 'n':
     - 'b' count is 2
     - 'a' count is 2
     - 'l' adjusted count is 1 (from the step above)
     - 'o' adjusted count is 2 (from the step above)
     - 'n' count is 2
   - The character that limits the number of possible "balloon" words is 'l', with an adjusted count of 1.
4. **Final result:**
   - Since the adjusted 'l' count is 1, we can only form the word "balloon" one time from the input `text`.

This example helps to illustrate how the use of `Counter`, bit shifts, and finding the minimum count among necessary characters enables us to efficiently solve the problem and determine the maximum number of times the word "balloon" can be created from a given string of text.

Python Solution

```python
from collections import Counter

class Solution:
    def maxNumberOfBalloons(self, text: str) -> int:
        # Create a counter from the characters in the text
        character_count = Counter(text)

        # The letters 'o' and 'l' appear twice in "balloon", so divide their counts by 2
        character_count['o'] //= 2
        character_count['l'] //= 2

        # Return the minimum count among the letters in the word "balloon"
        # Since 'balloon' has the characters 'b', 'a', 'l' (now halved), 'o' (halved), and 'n'
        return min(character_count[char] for char in 'balon')
```

Java Solution

```java
class Solution {
    public int maxNumberOfBalloons(String text) {
        // Initialize an array to count the frequency of each character in the given text
        int[] charFrequency = new int[26];

        // Iterate through each character in the text and increment the count in the array
        for (int i = 0; i < text.length(); ++i) {
            ++charFrequency[text.charAt(i) - 'a'];
        }

        // Since 'l' and 'o' appear twice in "balloon", divide their counts by 2
        charFrequency['l' - 'a'] >>= 1; // Equivalent to dividing by 2 using bitwise operator
        charFrequency['o' - 'a'] >>= 1; // Equivalent to dividing by 2 using bitwise operator

        // Initialize the answer with a large number
        int maxBalloons = Integer.MAX_VALUE;

        // Iterate over the characters of the word "balloon"
        for (char c : "balon".toCharArray()) {
            // Find the minimum frequency among the characters 'b', 'a', 'l', 'o', 'n'
            maxBalloons = Math.min(maxBalloons, charFrequency[c - 'a']);
        }

        // The minimum frequency determines the maximum number of "balloon" strings that can be formed
        return maxBalloons;
    }
}
```

C++ Solution

```cpp
#include <algorithm>  // For using the std::min function
#include <string>
using namespace std;

class Solution {
public:
    int maxNumberOfBalloons(string text) {
        int charCounts[26] = {}; // Initialize an array to store the counts of each letter

        // Count the frequency of each character in the text
        for (char c : text) {
            ++charCounts[c - 'a'];
        }

        // Since 'l' and 'o' are counted twice in "balloon", we need to halve their counts
        charCounts['o' - 'a'] /= 2;
        charCounts['l' - 'a'] /= 2;

        // Set the initial answer to a very high value
        int minBalloons = INT_MAX;

        // "balon" represents the unique characters in "balloon"
        string balloon = "balon";
        for (char c : balloon) {
            // Find the minimum count of the letters in "balon" to determine the answer
            minBalloons = min(minBalloons, charCounts[c - 'a']);
        }

        return minBalloons; // Return the minimum number of "balloon" strings
    }
};
```

Typescript Solution

```typescript
function maxNumberOfBalloons(text: string): number {
    // Create an array to keep the count of each letter in the English alphabet initialized with zeroes.
    const charCount = new Array(26).fill(0);

    // Iterate through the given text and increment the count of each letter.
    for (const char of text) {
        charCount[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Calculate the maximum number of times the word "balloon" can be formed.
    // For the letters 'l' and 'o', we should divide the count by 2 as they appear twice in the word "balloon".
    // 'b' is at index 1, 'a' is at index 0, 'l' is at index 11, 'o' is at index 14, and 'n' is at index 13.
    const maxBalloons = Math.min(
        charCount['b'.charCodeAt(0) - 'a'.charCodeAt(0)],  // Count of 'b'
        charCount['a'.charCodeAt(0) - 'a'.charCodeAt(0)],  // Count of 'a'
        charCount['l'.charCodeAt(0) - 'a'.charCodeAt(0)] >> 1, // Count of 'l' divided by 2
        charCount['o'.charCodeAt(0) - 'a'.charCodeAt(0)] >> 1,  // Count of 'o' divided by 2
        charCount['n'.charCodeAt(0) - 'a'.charCodeAt(0)]    // Count of 'n'
    );

    // Return the maximum number of "balloon" strings that can be formed.
    return maxBalloons;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the string `text`. This is because the code iterates over each character in the string once to construct the `Counter` object, which is an operation that takes `O(n)` time. After that, the operations that halve the counts for 'o' and 'l' and the minimum finding operation take constant time, since they don't depend on the size of the input string; they only iterate over the fixed set of characters in the string 'balon'.

The space complexity of the code is also `O(n)`. Although there are a fixed number of keys ('b', 'a', 'l', 'o', 'n') in the counter that could suggest constant space, in the worst case, the `Counter` might store every unique character in the input string if `text` does not contain any characters from "balloon". Therefore, the space used by the `Counter` object scales with the size of the `text`.

Note that these analyses assume typical implementations of Python's `Counter` and looping constructs.