2514. Count Anagrams

Hard · Hash Table · Math · String · Combinatorics · Counting — Leetcode Link

Problem Description

The problem gives you a string `s` which consists of one or more words separated by single spaces. A string `t` is considered an anagram of string `s` if for every word in `t`, there is a corresponding word in `s` that has exactly the same letters, but possibly in a different order. The goal is to calculate the number of distinct anagrams of `s`. Since the result could be a very large number, you're asked to return this number modulo `10^9 + 7`, which is often used in programming contests to avoid dealing with exceedingly large numbers that could cause overflow errors.

Intuition

To solve this problem, we need to think about what an anagram actually is. An anagram of a word is a permutation of its letters. If we consider each word separately, the number of different permutations (and thus anagrams) for a word is given by the factorial of the length of the word.

However, there's a catch: if a word has duplicate letters, we must divide the total count by the factorials of the counts of each letter. That's because permuting the duplicates among themselves doesn't create unique anagrams.

For example, for the word "aabb", it has 4 letters, so naïvely we might think there are 4! (which is 24) anagrams. However, both 'a' and 'b' are repeated twice. We need to divide 4! by the number of permutations of 'a' (which is 2!) and by the number of permutations of 'b' (which is 2!) to get the right answer: 4! / (2! * 2!), which equals 6.

The solution code first builds a list `f` of precomputed factorials modulo `10^9 + 7` to avoid recalculating factorials for each word, thereby saving time.

Then, for each word in the given string, we calculate the factorial of the length of the word, multiply this by the resulting `ans` so far, and then modify `ans` by multiplying it by the modular inverses of the factorials of the counts of each letter in the word. The function `pow(f[v], -1, mod)` is used to compute the modular inverse of `f[v]` under the modulo `10^9 + 7`.

Finally, we return `ans`, which—because we've taken the modulo at each step—will be the correct count of distinct anagrams of `s` modulo `10^9 + 7`.

Solution Approach

The provided solution code implements the following approach:

1. **Precompute Factorials**: Initially, the code precomputes factorials of numbers up to `10^5` and stores them in a list `f`. This list will be used to look up the factorial of any number during the computation without having to recalculate it. This precomputation is done modulo `10^9 + 7` to prevent integer overflow and to comply with the requirement of returning the answer modulo `10^9 + 7`.
2. **Iterate Over Words**: The core logic of the solution iterates over each word within the input string `s`. We use the `split()` method to break the string into a list of words.
3. **Count Letter Frequencies**: For each word, a `Counter` (from Python's collections module) is used to calculate the frequencies of each letter within the word. The counter creates a dictionary that maps each letter to its frequency.
4. **Calculate Anagrams**: For every word, we first multiply `ans` with the factorial of the length of the word. This represents the total number of permutations of the letters without considering duplicate letters.
5. **Adjust for Duplicate Letters**: Since we need to account for repeated letters, we loop over the values of our letter frequency counter and get the factorial of each letter count. Using the precomputed factorials in `f`, we calculate the modular inverse of these factorials using the `pow` function, which allows us to compute inverse modulo operations. We then multiply `ans` by this inverse to correctly account for the permutations of the duplicate letters.
6. **Keep Result within Modulo**: Throughout these operations, we ensure that `ans` is kept within the modulo `10^9 + 7` by taking the modulo operation after each multiplication. This step is crucial because it ensures that the value of `ans` never exceeds the limit where it might cause overflow errors.
7. **Return the Result**: After iterating through all the words and updating `ans` accordingly, we return `ans`, which is the total number of distinct anagrams of the original string `s` modulo `10^9 + 7`.

It's worth noting that the `pow(x, -1, mod)` computation is a way of finding the modular multiplicative inverse of `x` mod `mod` when `x` and `mod` are coprime (which they are in this case because the mod is prime and the factorial is not zero).

Using this approach, the solution code efficiently calculates the number of distinct anagrams of the given string `s` by breaking the problem down into calculations involving factorials and considering the impact of duplicate letters.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the string `s = "abc cab"`.

Step 1: Precompute Factorials

Factorials up to the maximum length `10^5` are precomputed and stored in a list `f` modulo `10^9 + 7`. For this example, we would have `f[0] = 1, f[1] = 1, f[2] = 2, f[3] = 6, ...` and so on.

Step 2: Iterate Over Words

We split the string `s` into words using the `.split()` method, resulting in the list `["abc", "cab"]`.

Step 3: Count Letter Frequencies

For each word, we count letter frequencies. For `abc`, it's `{'a': 1, 'b': 1, 'c': 1}`. For `cab`, it's the same since `cab` is a permutation of `abc`.

Step 4: Calculate Anagrams

We start with `ans = 1`. For each word:

- For the word `abc`, the length is 3, so we multiply `ans` by `f[3] which is 6. Now,` ans=ans * 6 % (10^9 + 7)->ans` = 6. We don't need to adjust for duplicates because all letters are distinct.
- For the word `cab`, we repeat the process. Multiplying `ans` by `f[3]` which is 6, we get `ans = 36` modulo `10^9 + 7`.

Step 5: Adjust for Duplicate Letters

Both words in our example do not have duplicate letters, so we don't need to adjust for duplicates. If there were duplicates, we would multiply `ans` by the modular inverses of the factorials of the letter counts.

Step 6: Keep Result within Modulo

This step ensures that `ans` remains within the range allowed by modulo `10^9 + 7`. In our case, since `ans` is 36 after processing both words, there is no issue with overflow, and we don't need further modulo operations here.

Step 7: Return the Result

We have completed the iteration over all words, and the final `ans` represents the total number of distinct anagrams of the string `s` modulo `10^9 + 7`. So we would return `ans = 36`.

That number, 36, says that there are 36 different permutations of `s` considering each word and its anagrams, but this is only for the length and uniqueness of the chosen words "abc" and "cab". In a more complex example with longer words and repeating characters, steps 4 and 5 would show more adjustment for duplicates.

Python Solution

```python
from collections import Counter

MOD = 10**9 + 7

# Pre-calculate factorial values modulo MOD
factorials = [1]
for i in range(1, 10**5 + 1):
    new_value = factorials[-1] * i % MOD
    factorials.append(new_value)

class Solution:
    def countAnagrams(self, s: str) -> int:
        # Initialize the answer as 1
        answer = 1

        # Split the string into words
        for word in s.split():
            # Count the frequency of each character in the word
            character_counts = Counter(word)
            # Multiply the answer by the factorial of the length of the word
            answer *= factorials[len(word)]
            answer %= MOD

            # For each character, update the answer by multiplying with
            # the multiplicative inverse of the factorial of the character's count
            for count in character_counts.values():
                answer *= pow(factorials[count], -1, MOD)
                answer %= MOD

        return answer

# Example usage:
# sol = Solution()
# result = sol.countAnagrams("the quick brown fox")
# print(result)

```

Java Solution

```java
import java.math.BigInteger;

class Solution {
    // Define the modulus constant for taking the mod of large numbers to prevent overflow
    private static final int MOD = (int) 1e9 + 7;

    // Method to count the number of anagrams for groups of words separated by spaces
    public int countAnagrams(String s) {
        int length = s.length();
        // Factorial array to store precomputed factorials under MOD
        long[] factorials = new long[length + 1];
        factorials[0] = 1; // 0! is 1

        // Calculate factorial for each number up to length, with mod
        for (int i = 1; i <= length; ++i) {
            factorials[i] = (factorials[i - 1] * i) % MOD;
        }

        long product = 1; // Initialize the product to 1
        // Split the given string by spaces and process each group of characters
        for (String word : s.split(" ")) {
            int[] characterCount = new int[26];

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

            // Multiply the product by the factorial of the length of the word
            product = (product * factorials[word.length()]) % MOD;

            // Iterating over the character frequencies,
            // taking the modInverse of their factorial, and updating the product
            for (int count : characterCount) {
                if (count > 0) {
                    BigInteger factorial = BigInteger.valueOf(factorials[count]);
                    BigInteger modInverse = factorial.modInverse(BigInteger.valueOf(MOD));
                    product = (product * modInverse.intValue()) % MOD;
                }
            }
        }
        // Finally, return the product casted to an integer
        return (int) product;
    }
}

```

C++ Solution

```cpp
#include <sstream>
#include <string>

class Solution {
public:
    // We declare 'mod' as a static constant because it's a property of the class and
    // should not change.
    static const int MOD = 1e9 + 7;

    // Calculate the number of anagrams for words in a string
    // Break down the string into words and compute the result
    int countAnagrams(std::string s) {
        std::stringstream ss(s);  // used to break the string into words
        std::string word;         // to store each word individually
        long ans = 1;             // to store the number of permutations
        long factorial = 1;       // to store the factorial part

        // Process each word in the string
        while (ss >> word) {
            int count[26] = {0};  // to count occurrences of each letter

            for (int i = 1; i <= word.size(); ++i) {
                // Calculate zero-based index for character in alphabet
                int charIndex = word[i - 1] - 'a';

                // Increment the count for this character
                ++count[charIndex];

                // Update the permutation count: ans = ans * i
                ans = (ans * i) % MOD;

                // Update the factorial part using the letter's occurrence
                factorial = (factorial * count[charIndex]) % MOD;
            }
        }

        // Use modular inverse to divide ans by the factorial part
        return static_cast<int>((ans * pow(factorial, MOD - 2)) % MOD);
    }

    // Calculate x raised to the power of n mod MOD using fast exponentiation
    long pow(long x, int n) {
        long result = 1L;  // start with result as 1

        // Continue looping until n becomes zero
        while (n > 0) {
            // If n is odd, multiply the result by x
            if (n % 2 == 1) {
                result = (result * x) % MOD;
            }

            // Square x and reduce n by half
            x = (x * x) % MOD;
            n /= 2;
        }

        return result;
    }
};

```

Typescript Solution

```typescript
// Define the modulo constant.
const MOD: number = 1e9 + 7;

// Calculate the number of anagrams for words in a string.
// Break down the string into words and compute the result.
function countAnagrams(s: string): number {
    const words: string[] = s.split(/\s+/); // Split the string into words
    let ans: number = 1; // To store the number of permutations
    let factorial: number = 1; // To store the factorial part

    // Process each word in the string
    for (let word of words) {
        let count: number[] = new Array(26).fill(0); // To count occurrences of each letter

        for (let i: number = 0; i < word.length; i++) {
            // Calculate zero-based index for character in alphabet
            let charIndex: number = word.charCodeAt(i) - 'a'.charCodeAt(0);

            // Increment the count for this character
            count[charIndex]++;

            // Update the permutation count: ans = ans * (i + 1)
            ans = (ans * (i + 1)) % MOD;

            // Update the factorial part using the letter's occurrence
            factorial = (factorial * count[charIndex]) % MOD;
        }
    }

    // Use modular inverse to divide ans by the factorial part
    return (ans * pow(factorial, MOD - 2)) % MOD;
}

// Calculate x raised to the power of n mod MOD using fast exponentiation.
function pow(x: number, n: number): number {
    let result: number = 1; // Start with result as 1

    // Continue looping until n becomes zero.
    while (n > 0) {
        // If n is odd, multiply the result by x.
        if (n % 2 === 1) {
            result = (result * x) % MOD;
        }

        // Square x and reduce n by half.
        x = (x * x) % MOD;
        n = Math.floor(n / 2);
    }

    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the precomputation part (where the factorials are computed up to `10**5`) is `O(N)` where `N` is `10**5`. This part is run only once, and each operation is a multiplication and a modulo operation, which are both `O(1)`. When analyzing the `countAnagrams` method, for each word in the input string `s`, the code counts the frequency of characters, which is `O(K)` where `K` is the length of the word. Multiplying by the factorial of the length of the word is `O(1)` since we have precomputed the factorials. The `for` loop for dividing by the factorial of the count of each character runs at most `26` times (since there are only 26 letters in the alphabet), and each iteration does a modular inverse and multiplication, which can both be considered `O(1)` under modular arithmetic. Therefore, for each word, the complexity is dominated by `O(K)`. Since this is done for every word in the string, and if there are `W` words in `s`, the overall time complexity of `countAnagrams` is `O(W*K)`.

Space Complexity

The space complexity for storing the precomputed factorials is `O(N)`, where `N` is `10**5`. Aside from that, the space used by the function `countAnagrams` depends on the size of the counter object, which is at most `26` for each word. Therefore, the space complexity for the function `countAnagrams` is `O(1)`, making the total space complexity of the program `O(N)`.