1915. Number of Wonderful Substrings

Medium | Bit Manipulation | Hash Table | String | Prefix Sum | Leetcode Link

Problem Description

The problem presents a concept of a 'wonderful' string which is defined as a string where at most one letter appears an odd number of times. We are tasked with counting the number of wonderful non-empty substrings within a given string `word`. This string `word` contains only the first ten lowercase English letters (`'a'` through `'j'`). It is important to note that we must consider each occurrence of a wonderful substring separately even if it appears multiple times in `word`. A substring is defined as a contiguous sequence of characters within a string.

Intuition

The intuition behind solving this problem lies in recognizing patterns and efficiently tracking the frequency of characters as we consider different substrings. A naive approach would be to check all possible substrings and count how many meet the criteria, but this would be too slow for larger strings.

The solution uses bitwise operations to keep track of the frequency of each letter in a space-efficient manner. We can represent the frequency of each of the 10 letters by a single bit in an integer (`st`). If a letter appears an even number of times, its corresponding bit is set to 0, and if it appears an odd number of times, its bit is set to 1. The wonderful property is then satisfied if at most one bit is set to 1 in this integer.

We initialize a counter to keep track of the occurrences of each bit pattern as we iterate through `word`. As we consider each new character, we update our bit pattern state `st`. For each new character, we add to our answer the count of the current state `st` to capture the same pattern we have already seen (which automatically represents an even number of each letter in the substring). Additionally, we iterate through each bit position to flip it, checking if there is a pattern that had all bits even except for the current one. This would represent substrings with all even counts of characters except one, meeting the 'wonderful' criteria.

The `Counter` structure is used to keep track of how many times we've encountered each bit pattern. This allows us to quickly calculate the new number of wonderful substrings each time we process a new character in the string. By adding the counter for the current state and the counter for states with one bit flipped, we can account for all the substrings that end at the current character and are wonderful.

Using this method, we efficiently compute the number of wonderful substrings in the given word without checking every possible substring individually.

Solution Approach

The solution implemented in the reference code uses a combination of bit manipulation, hashing (via a `Counter` dictionary), and comprehension of the property of wonderful substrings.

1. **Bit Manipulation**: To track the odd or even frequency of each letter without storing the counts individually, the solution uses bit manipulation, where each bit in a state integer `st` represents the count of a particular letter from `'a'` to `'j'`. The least significant bit corresponds to `a`, the second least significant bit to `b`, and so on up to the tenth bit for `j`. If a letter has appeared an odd number of times, the corresponding bit is set to 1, and if it has appeared an even number of times, it's set to 0.

2. **Counter Dictionary**: This is used to hash the number of times each bit pattern appeared. It's implemented using Python's `Counter` class.

3. **State Transition**: When a new character from the string `word` is processed, we calculate the new state `st` by using the XOR operation `st ^= 1 << (ord(c) - ord("a"))`. The XOR operation with `1` shifted left by `(ord(c) - ord("a"))` positions flips the bit corresponding to the new character. This changes the bit from 0 to 1 if the character count was even (representing an odd count now) and from 1 to 0 if the count was odd (representing an even count now).

4. **Counting Wonderful Substrings**: The number of wonderful substrings that can be formed ending with the current character is the sum of:

   - The number of times the current state `st` has been seen before, which is `ans += cnt[st]`, because if we've seen this pattern before, it means there is a substring ending here that maintains the evenness of all previously processed characters.
   - The number of times each bit-flipped state has occurred, which is computed with `ans += cnt[st ^ (1 << i)] for i in range(10)`. Flipping each bit simulates having all the other letters appear an even number of times and only one letter an odd number of times, which is still a 'wonderful' state.

5. **Updating the Counter**: After accounting for the new wonderful substrings, `cnt[st] += 1` updates the count for the current state.

The solution goes through each character only once, and for each character, it checks 10 possible states—those with no letters appearing an odd number of times and those with exactly one. This results in a linear `O(n*10)` solution, where `n` is the length of `word`.

In terms of space complexity, the counter keeps track of at most `2^10` (1024) different bit patterns, which corresponds to a bitwise representation of letter counts. So the space complexity is `O(2^k)` where `k` is the number of unique letters, which is 10 in this case. However, it only keeps track of bit patterns that have been seen, so the actual space used could be much less.

Example Walkthrough

Let's consider the string `word = "aba"`. We will walk through the solution approach step by step.

1. **Bit Manipulation**: We are using bit manipulation to track the frequency of each letter using the state integer `st`. Initially, `st = 0`, which means all bits are set to 0, indicating even counts of all letters.

2. **Counter Dictionary**: We initialize a `Counter` with `cnt = {0:1}`, which implies we have already encountered a state with all even counts once (an empty substring).

3. **State Transition**:

   - Process the first character `'a'`: The state transition is `st ^= 1 << (ord('a') - ord('a'))`, hence `st = 0b0000000001`. The `Counter` now recognizes one odd appearance of `'a'`.

   - Process the second character `'b'`: The state changes with `st ^= 1 << (ord('b') - ord('a'))`, so `st` becomes `0b0000000010`. Now we have both `'a'` and `'b'` with odd appearances (but not in the same substring).

4. **Counting Wonderful Substrings**:

   - For `st = 0b0000000001` after processing the first `'a'`:

     - `ans += cnt[st]` (the previous same state), so `ans = 0+1 = 1` because an all-even state was seen before, and the current state is the wonderful substring "a".
     - Flip each bit in `st` and add those counts: There are no previous `1`-bit-flipped states, so no addition here.
   - Considering the state `0b0000000010` after processing `'b'`:

     - `ans += cnt[st]` (the previous same state), so `ans = 1+0 = 1` because there's yet no state with only 'b' with an odd count.
     - Flipping each bit of `st` (checking for `0b0000000001` and `0b0000000010`), we find `ans += cnt[0b0000000001] = 1`, hence `ans = 2`. This represents the wonderful substring "b".
   - Now, process the third character, another `'a'`:

     - Update `st` using `st ^= 1 << (ord('a') - ord('a'))`, `st` is now `0b0000000010` again.
     - `ans += cnt[st]` would result in `ans = 2+1 = 3` because we have seen this state before, representing the substrings "b" and "aba".
     - By flipping each bit in `st` and adding those counts, we find `ans += cnt[0b0000000001] = 1`; therefore, `ans = 4`. This represents the substring "ab".

5. **Updating the Counter**: After each character is processed, we update the counter for the current state:

   - After the first 'a', `cnt[0b0000000001] = 1` as we've seen the state of 'a' once.
   - After 'b', `cnt[0b0000000010] = 1` as we've seen the state of 'b' once.
   - After the second 'a', `cnt[0b0000000010] = 2` since we've returned to this state.

In the end, `ans = 4`. We have the wonderful substrings: "a", "b", "ab", "aba". Each of these substrings meets the criteria of at most one letter having an odd count within the substring.

Python Solution

```python
from collections import Counter

class Solution:
    def wonderfulSubstrings(self, word: str) -> int:
        # Initialize a counter for the mask state, starting with the 0 state seen once
        mask_count = Counter({0: 1})

        # Initialize answer and mask state
        wonderful_count = 0
        current_mask = 0

        # Iterate over the characters in the word
        for char in word:
            # Toggle the bit for the current character in the mask state
            current_mask ^= 1 << (ord(char) - ord('a'))

            # Add to the wonderful string count the number of times this mask state has been seen
            # This covers the case where the substring has an even count of all characters
            wonderful_count += mask_count[current_mask]

            # Check all masks that differ by one bit, which correspond to having
            # exactly one character with an odd count
            for i in range(10):
                # Toggle the i-th bit to check for a previous state that would
                # complement the current state to make a wonderful substring
                wonderful_count += mask_count[current_mask ^ (1 << i)]

            # Increment the count for the current mask state
            mask_count[current_mask] += 1

        # Return the total count of wonderful substrings
        return wonderful_count

```

Java Solution

```java
class Solution {
    /**
     * Returns the number of 'wonderful' substrings in a given word.
     * A 'wonderful' substring is defined as a substring that has at most one character appear
     * an odd number of times.
     *
     * @param word The input string for which we want to find the number of 'wonderful' substrings.
     * @return The count of 'wonderful' substrings.
     */
    public long wonderfulSubstrings(String word) {
        // Initialize an array to count the state occurrences.
        int[] stateCount = new int[1 << 10]; // 1 << 10 because there are 10 possible characters (a-j).
        stateCount[0] = 1; // Empty string is a valid starting state.
        long totalCount = 0; // This will hold the total number of 'wonderful' substrings.
        int state = 0; // This represents the bitmask state of characters a-j. Each bit represents if a character has an odd or even

        // Loop over each character in the string
        for (char c : word.toCharArray()) {
            // Toggle the bit corresponding to the character c.
            state ^= 1 << (c - 'a');

            // Add the count of the current state to answer.
            totalCount += stateCount[state];

            // Try toggling each bit to account for at most one character that can appear an odd number of times.
            for (int i = 0; i < 10; ++i) {
                totalCount += stateCount[state ^ (1 << i)];
            }

            // Increment the count of the current state.
            ++stateCount[state];
        }

        // The total number of wonderful substrings is now calculated in totalCount.
        return totalCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long wonderfulSubstrings(string word) {
        // Array to count the number of times each bit mask appears
        int count[1024] = {1}; // Initialize with 1 at index 0 to handle the empty substring scenario
        long long totalSubstrings = 0; // Total count of wonderful substrings
        int state = 0; // Current state of bit mask representing character frequency parity

        // Iterate over each character in the string
        for (char ch : word) {
            // Update the state: Flip the bit corresponding to the current character
            state ^= 1 << (ch - 'a');

            // Add the count of the current state to the total substrings count
            totalSubstrings += count[state];

            // Check for states that differ by exactly one bit from the current state
            for (int i = 0; i < 10; ++i) {
                totalSubstrings += count[state ^ (1 << i)];
            }

            // Increment the count for the current state
            ++count[state];
        }

        // Return the total count of wonderful substrings
        return totalSubstrings;
    }
};

```

Typescript Solution

```typescript
function wonderfulSubstrings(word: string): number {
    const count: number[] = new Array(1 << 10).fill(0); // An array to store the count of all character state occurrences
    count[0] = 1; // Initial state with 0 odd characters
    let totalWonderfulSubstrings = 0; // Variable to count the number of wonderful substrings
    let charState = 0; // Bitmask state representing the parity of character counts

    // Iterate through each character of the word
    for (const char of word) {
        // XOR the current state with the bit representing the current character's position
        charState ^= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0));

        // Add the count of the current state to the total count of wonderful substrings
        totalWonderfulSubstrings += count[charState];

        // Check every possible character state with one bit flipped
        for (let i = 0; i < 10; ++i) {
            totalWonderfulSubstrings += count[charState ^ (1 << i)];
        }

        // Increment the count of the current character state
        count[charState]++;
    }

    // Return the total number of wonderful substrings
    return totalWonderfulSubstrings;
}

```

Time and Space Complexity

Time Complexity:

Analyzing the code, the main part contributing to time complexity is the two nested loops: the outer loop iterating over each character of the string once, and the inner loop iterating 10 times for each character (since an alphabet in lowercase has 26 letters).

- The outer loop runs `n` times where `n` is the length of the input string `word`.
- The inner loop runs up to `10` times for each iteration of the outer loop, irrespective of the input string.

Thus, the overall time complexity can be computed as `O(10n)` or `O(n)` when we ignore the constant factor.

The operations within the loops (bitwise XOR, dictionary access/update) are constant time.

Hence, the time complexity of the code is: `O(n)`.

Space Complexity:

For space complexity, the code maintains a counter `cnt` dictionary that at most contains the number of different states that a bitset of size `10` (size corresponding to the first `10` alphabets) can have, plus 1 for the initial {0:1} state. The `st` variable holds the current state and changes in-place.

The states are bitsets that correspond to the parity (even or odd count) of each letter, and there can be `2^10` such states.

So, the space complexity is determined by the number of different states that can be held in counter `cnt`, which gives us `2^10`.

Therefore, the space complexity of the code is: `O(2^10)` or `O(1)` because `2^10` is a constant.