1371. Find the Longest Substring Containing Vowels in Even Counts

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

Problem Description

In this problem, you're given a string `s`. Your task is to find the length of the longest substring of `s` where each of the vowels - 'a', 'e', 'i', 'o', 'u' - appears an even number of times. Substrings are continuous parts of the original string, and in this case, any substring that satisfies the vowel condition could potentially be the longest. The goal is to figure out the maximum length possible while adhering to the even occurrence condition for vowels.

Intuition

The intuition behind the solution is to use a bitwise representation to track the count of vowels in the substring efficiently. Because we only care about even or odd occurrences, we don't need to keep a count of each vowel; we only need to know if a vowel has appeared an even or odd number of times. Thus, each vowel can be represented with 1 bit, resulting in a 5-bit integer for all five vowels (where `0` means even and `1` means odd number of occurrences).

The key insight is that if at two different points in the string, the 5-bit integer (or state) is the same, it means that we have a substring (between these two points) where all vowels have appeared an even number of times.

The solution iterates through the string and toggles the respective bit in the state when a vowel is found. If the state has been seen before, we calculate the length of the new substring ending at the current character and update the answer if it's longer than the previous maximum. The `pos` array keeps track of the first occurrence of each state to calculate lengths of valid substrings. We start by initializing the `pos` array with `inf` to denote that those states have not been seen yet, except for the state `0`, which is initialized to `-1` to handle the edge case where a valid substring starts at the beginning of the input string.

By using this bitwise state and the `pos` array, we can efficiently find the longest substring where all vowels appear an even number of times.

Solution Approach

The solution utilizes a bitwise approach combined with a hash map (implemented as a list called `pos` in the code) to keep track of the indices at which each state occurs for the first time.

Key Components of the Solution:

- **Vowels Bitmasking**: Each vowel has a corresponding bit in a 5-bit integer, where the order is 'a', 'e', 'i', 'o', 'u'. For example, if only 'a' and 'i' have been seen an odd number of times, the state would be `10100` in binary, which is `20` in decimal.
- **Tracking States**: The `state` variable is a 5-bit integer representing the current state of vowels' counts (even/odd). The initial state is `0` since we start with an even count (zero occurrences) for all vowels.
- **Index Memory (`pos`)**: An array called `pos` of size 32 (since there are 2^5 possible states due to 5 vowels) is used to remember the earliest occurrence of every state. This array is initialized with `inf`, except for `pos[0]` which is set to `-1`.
- **Iterating Through the String**: As we iterate through the string, we check each character. If it's a vowel, we flip the corresponding bit in the `state` using the exclusive OR (XOR) operation: `state ^= 1 << j`, where `j` is the index of the vowel in the string 'aeiou'.
- **Updating the Answer**: With each new character processed, we check if the current state has been encountered before:
  - If it's been seen, we calculate the length of the substring from the first occurrence of this state to the current position.
  - If the calculated length is greater than the `ans` (which keeps track of the maximum length seen so far), we update `ans`.
- **First Occurrence**: We also check if the current state's first occurrence needs to be updated in the `pos` array. If the current index is less than the stored value in `pos[state]`, we update `pos[state]` with the current index.

By the end of the string traversal, `ans` will hold the length of the longest substring where all vowels appear an even number of times, and that's what we return as the solution.

Example Walkthrough:

Let's walk through a quick example:

```
s = "eleetminicoworoep"
```

- We start at `state = 0` and `pos[0] = -1` because we have an even count (zero) for all vowels at the start.
- Iterating over `s`, whenever we encounter a vowel, we update `state`. Suppose `state` becomes `3` after some operations; this means 'a' and 'e' have been seen an odd number of times so far.
- If `state` is `3` again at a later point, we know that between these two indices, 'a' and 'e' must have appeared an even number of times. Therefore, we calculate the length of this substring and check if it's the maximum.
- We continue this process until the end of the string, constantly updating `ans` with longer valid substrings as we find them.

The implementation is efficient with a time complexity of `O(n)`, where `n` is the length of the string, because we only need a single pass through the string to compute the result, and each operation within that pass is of constant time complexity.

Example Walkthrough

Let's illustrate the solution approach using the string:

```
s = "aeiobcdf"
```

1. Initialize the `pos` array with length `32` to represent all possible states of vowel occurrences (2^5 for 5 vowels). Set all values to `inf`, except `pos[0]` to `-1`.
2. We start with `state = 0` (since no vowels have been seen yet) and `ans = 0` (since no substrings have been found yet).
3. As we iteratively check each character in `s`, we use the following steps:
   - When at index 0, `s[0] = 'a'`: We see our first vowel 'a'. The bit for 'a' in the `state` is toggled from `0` to `1`. Now, `state = 00001`.
   - At index 1, `s[1] = 'e'`: We toggle the bit for 'e'. Now, `state = 00011`.
   - At index 2, `s[2] = 'i'`: Toggling the 'i' bit: `state = 00111`.
   - At index 3, `s[3] = 'o'`: Toggling 'o': `state = 01111`.
   - At index 4, `s[4] = 'b'`: 'b' is not a vowel; state remains `01111`.
   - At index 5, `s[5] = 'c'`: 'c' is not a vowel; state remains `01111`.
   - At index 6, `s[6] = 'd'`: 'd' is not a vowel; state remains `01111`.
   - At index 7, `s[7] = 'f'`: 'f' is not a vowel; state remains `01111`.
4. We've reached the end of the string, and throughout, each vowel has appeared exactly once, so their counts are odd. Our final state is `01111`.
5. Throughout our iteration, the state has changed from `00000` to `01111`. After toggling the bits whenever we encountered a vowel, we checked `pos` to see if the current state had been recorded before. In this instance, since no state repeated other than the initial state `0`, the length of the longest valid substring is zero because we have not encountered a state twice where the state would be `00000` again (meaning that all vowels have an even count).

So, the `ans` remains `0` in this example; there are no substrings where each vowel occurs an even number of times.

The key takeaway is the tracking of vowel occurrences through bit manipulation, using XOR to toggle between even and odd counts, and using the `pos` array to store the first occurrence of a state. This approach allows for quick lookups and updates while maintaining an efficient assessment of the longest valid substring. However, in our example, there were no repeated states to determine a valid substring. In a longer string with repeated vowel occurrences, the `ans` would likely be greater than zero.

Python Solution

```python
from math import inf

class Solution:
    def findTheLongestSubstring(self, s: str) -> int:
        # Initialize a list to store the first position we encounter a given state
        # State is represented as a bitmask integer of size 32 (for 5 vowels, each can be on/off)
        positions = [inf] * 32

        # The starting state (all vowels have even counts) is at index 0
        positions[0] = -1

        # List of vowels for reference
        vowels = 'aeiou'

        # `state` to keep track of the count of vowels encountered (even/odd as a bitmask)
        # `max_length` to keep track of the length of the longest valid substring so far
        state = max_length = 0

        # Iterate over the string characters with their index
        for index, char in enumerate(s):
            # Check if the current character is a vowel
            for j, vowel in enumerate(vowels):
                # Toggle the corresponding bit if we encounter a vowel
                if char == vowel:
                    state ^= 1 << j

            # Calculate max length using current index and first index where current state was seen
            max_length = max(max_length, index - positions[state])
            # Update the position for this state if it's the first time we're seeing this state
            positions[state] = min(positions[state], index)

        return max_length
```

Java Solution

```java
class Solution {

    // Function to find the length of the longest substring containing vowels in even counts
    public int findTheLongestSubstring(String s) {
        // pos array to keep track of the earliest index of each state
        int[] earliestPos = new int[32]; // 32 possible states for 5 vowels (2^5)
        // Initialize all positions to max value except for state 0
        Arrays.fill(earliestPos, Integer.MAX_VALUE);
        // State 0 (no vowels seen or all seen even times) starts at index -1
        earliestPos[0] = -1;

        // String of vowels to check against
        String vowels = "aeiou";

        // 'state' will represent the binary value of the vowels seen odd number of times
        int state = 0;
        // 'maxLength' stores the length of the longest substring found so far
        int maxLength = 0;

        // Loop through characters of input string
        for (int i = 0; i < s.length(); i++) {
            char currentChar = s.charAt(i);
            // Check and flip the corresponding bit for the vowel
            for (int j = 0; j < 5; j++) {
                if (currentChar == vowels.charAt(j)) {
                    state ^= (1 << j);
                }
            }

            // If state has been seen before, update the maxLength
            maxLength = Math.max(maxLength, i - earliestPos[state]);
            // If state has not been seen before, set it to the current index
            if (earliestPos[state] == Integer.MAX_VALUE) {
                earliestPos[state] = i;
            }
        }

        // Return the length of the longest substring
        return maxLength;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int findTheLongestSubstring(string s) {
        // Initialize a vector to keep track of the first occurrence of all states.
        vector<int> firstOccurrence(32, INT_MAX);
        // Setting the base condition that for state '0', the first occurrence is at -1
        firstOccurrence[0] = -1;

        // Define a string of vowels for easy indexing.
        string vowels = "aeiou";

        // This will keep track of the current state of vowels encountered.
        int currentState = 0;

        // The result variable to store the length of the longest substring.
        int longestSubstringLength = 0;

        // Iterate over the characters of the string.
        for (int i = 0; i < s.size(); ++i) {
            // Check if the current character is a vowel and update the state accordingly.
            for (int j = 0; j < 5; ++j) { // There are 5 vowels.
                if (s[i] == vowels[j]) {
                    // Toggle the j-th bit of the state to represent the occurrence of vowel.
                    currentState ^= (1 << j);
                }
            }
            // Calculate the length if the current state has occurred before
            // The length of the valid string would be difference of indices.
            longestSubstringLength = max(longestSubstringLength, i - firstOccurrence[currentState]);
            // Update the occurrence of current state if it's the first time.
            firstOccurrence[currentState] = min(firstOccurrence[currentState], i);
        }

        // Return the length of the longest valid substring found.
        return longestSubstringLength;
    }
};
```

Typescript Solution

```typescript
// Define a string of vowels for easy indexing.
const vowels: string = "aeiou";

// This function calculates and returns the length of the longest substring
// where the count of each vowel is even.
function findTheLongestSubstring(s: string): number {
    // Initialize an array to keep track of the first occurrence of all states.
    let firstOccurrence: number[] = new Array(32).fill(Number.MAX_SAFE_INTEGER);
    // Setting the base condition that for state '0', the first occurrence is at -1
    firstOccurrence[0] = -1;

    // This will keep track of the current state of vowels encountered.
    let currentState: number = 0;

    // The variable to store the length of the longest substring.
    let longestSubstringLength: number = 0;

    // Iterate over the characters of the string.
    for (let i = 0; i < s.length; i++) {
        // Check if the current character is a vowel and update the state accordingly.
        for (let j = 0; j < vowels.length; j++) {
            if (s[i] === vowels[j]) {
                // Toggle the j-th bit of the state to represent the occurrence of the vowel.
                currentState ^= (1 << j);
            }
        }
        // If the current state hasn't occurred before, update the occurrence of the current state.
        if (firstOccurrence[currentState] === Number.MAX_SAFE_INTEGER) {
            firstOccurrence[currentState] = i;
        }

        // Calculate the length if the current state has occurred before
        // The length of the valid substring would be the difference of indices.
        longestSubstringLength = Math.max(longestSubstringLength, i - firstOccurrence[currentState]);
    }

    // Return the length of the longest valid substring found.
    return longestSubstringLength;
}
```

Time and Space Complexity

Time Complexity

The given code iterates over each character of the string `s`, which has a length `n`. It performs a constant amount of work for each character: checking if the character is a vowel, possibly flipping a bit in the `state` variable, and updating `ans` and `pos[state]`. Because these operations all have a constant time complexity, the overall time complexity of the function is `O(n)`.

Space Complexity

The space complexity is determined by the fixed-size array `pos`, which has 32 elements (one for each possible state of the 5 vowel bits), and a few additional integer variables (`state`, `ans`, and the loop variables). Therefore, the space complexity of the algorithm is `O(1)` since the space required does not grow with the size of the input string `s`.