1542. Find Longest Awesome Substring

Hard · Bit Manipulation · Hash Table · String · Leetcode Link

Problem Description

The problem involves finding the longest "awesome" substring within the given string `s`. An "awesome" substring is defined as a substring that can be transformed into a palindrome through a series of character swaps. In other words, you can take some characters in the substring and swap them with others to form a palindrome.

A palindrome is a string that reads the same forwards and backwards, like "racecar" or "level". Therefore, for a substring to be awesome and potentially become a palindrome, it must have at most one character that appears an odd number of times. All other characters must appear an even number of times, so they can be mirrored around the central character.

The goal is to find the length of the longest possible awesome substring.

Intuition

To solve this problem, one key observation must be made: a palindrome has a symmetric structure, which means if you split it in the middle, one side is the reverse of the other. In a palindrome, all characters occur an even number of times, except for potentially one character (in the middle of an odd-length palindrome), which occurs an odd number of times.

The solution approach utilizes bitmasks to represent the count of characters modulo 2 (even or odd). The state transitions occur as we iterate through the string and toggle bits for the respective numeric values of characters.

Here's the intuition breakdown for the solution:

- Create a bitmask `st` (of 10 bits, one for each digit 0-9) that represents which numbers have occurred an odd number of times as we iterate over the string.
- Use a dictionary `d` to store the first index where each bitmask state occurs. Initializing the dictionary with `{0: -1}` handles the case where an "awesome" substring starts at the beginning of `s`.
- The solution iterates through `s`, and at each iteration, it toggles the bit that corresponds to the current character using XOR operation (`st ^= 1 << v`).
- If the current state of `st` exists in the dictionary `d`, it indicates a palindrome from the index `d[st]` to the current index `i`.
- The solution keeps track of the maximum length of an "awesome" substring found so far.
- Additionally, it checks if changing the current state of `st` by toggling each bit (from 0-9) leads to a state present in the dictionary which means a single character swap might form an awesome substring. If this check finds a longer substring, it updates the maximum length accordingly.
- After completing the loop, the maximum length found is returned as the length of the longest "awesome" substring.

This approach allows us to solve the problem with an O(n) time complexity, since it iterates through the string once and does a constant amount of work for each character.

Solution Approach

The implementation of the solution makes use of bitwise operations, hash tables (dictionaries in Python), and the understanding of palindrome properties.

Here's a step-by-step explanation of the algorithm by referring to the code above:

1. Initialize `st` to `0`. This is a bitmask that will keep track of the parity (even or odd) of the counts of digits in the current substring. For example, if the third bit in `st` is `1`, it means the digit `2` has appeared an odd number of times so far.

2. Initialize a dictionary `d` with a single key-value pair `{0: -1}`. The dictionary will map the state of `st` to the earliest index where this state was seen. The value `-1` is used to handle cases where a palindrome starts from index 0.

3. Initialize `ans` to `1`, since the minimum length of an awesome substring is 1 (a single digit is always a palindrome).

4. Iterate over the given string `s`, using `enumerate` to have both index `i` and character `c` in the loop. Convert the current character to an integer `v`.

5. Toggle the bit in the `st` bitmask corresponding to `v`. This is done with the XOR operator `^` and the bitwise left shift operator `<<`.

6. Check if this new state of `st` has been seen before. If it has, calculate the length of the substring from the first occurrence of this state to the current index `i`. This is a potential palindrome, so update `ans` if this length is larger than the current `ans`.

7. Additionally, loop through all digit positions from `0` to `9`. Toggle each bit in the `st` bitmask to simulate having one character with an odd count (potential middle character in a palindrome). Check if this modified state has been seen before. If it has, it means that there exists a substring ending at the current index that could form an awesome substring with one swap. Update `ans` if a longer substring is found.

8. After the loop ends, return `ans`. This represents the length of the longest awesome substring found.

This solution cleverly utilizes bitmasks to track the parity of digit occurrences and a dictionary to remember first occurrences of bitmask states. The power of bitwise operations allows us to efficiently simulate all possible single-digit changes that may lead to a palindrome. The algorithm runs in O(n) time complexity with O(1) space complexity, as there are at most 2^10 possible bitmask states, which is a constant.

Example Walkthrough

Let us consider an example string `s = "3242415"` to illustrate the solution approach.

1. Initialize the bitmask `st` to `0`. The bits in `st` will eventually correspond to the parity of the counts of each digit in the current substring.

2. Initialize the dictionary `d` with `{0: -1}`. It maps the parity state to the index where it was first encountered. The `-1` handles cases where a palindrome starts at the first character of the string.

3. Initialize the answer `ans` to `1`. Any single digit is trivially a palindrome.

4. Begin iterating over each character in the string `s`:

   - At index `0`, the character is `3`. Convert `3` to an integer and toggle the 3rd bit of `st`, making it `001000`. Since this state has not been seen, add `d[001000] = 0`.
   - At index `1`, the character is `2`. Toggle the 2nd bit of `st`, making it `001100`. Add `d[001100] = 1` as this state is new.
   - At index `2`, the character is `4`. Toggle the 4th bit of `st`, making it `011100`. This state has not been seen before, so add `d[011100] = 2`.
   - At index `3`, the character is `2`. Toggle the 2nd bit of `st`, reverting it to `010000`. This state is new, add `d[010000] = 3`.
   - At index `4`, the character is `4`. Toggle the 4th bit of `st`, reverting it back to `000000`. This is the first time encountering a state of all even counts, but it indicates a substring `32424` that is a palindrome and can be mirrored. The length is `5`, so we update `ans` to `5`.
   - At index `5`, the character is `1`. Toggle the 1st bit of `st`, making it `000001`. Add `d[000001] = 5`, as this is a new state.
   - At index `6`, the character is `5`. Toggle the 5th bit of `st`, making it `000101`. This state is new, so add `d[000101] = 6`.

5. Now, suppose we're at index `6`. For each digit from 0-9, we consider toggling each bit of the current `st` (000101) one by one and look it up in the dictionary `d`. The only interesting toggles are `001101` and `000001` because these states have been encountered earlier, which means there's a palindrome `324241` and `5` when `5` or `1` is the middle character. The lengths are `6` and `1`, respectively. The longest length is the one for `324241`, which does not surpass our current `ans` of `5`.

6. Continue this process until the end of the string, always updating `ans` to the maximum length found.

7. After completing the loop, the maximum value of `ans` remains `5`, which is the length of the longest-awesome substring in `s`, and that substring is `32424`.

By following the detailed solution approach using bitwise representation and a hash table, we efficiently found the longest awesome substring in our example.

Python Solution

```python
class Solution:
    def longestAwesome(self, s: str) -> int:
        # Initialize the current state of the palindrome (bit mask representation of character counts)
        current_state = 0
        # Dictionary to store the earliest index of a particular state
        state_index_map = {0: -1}
        # Variable to store the length of the longest awesome substring
        max_length = 1

        # Iterate over each character in the string along with its index
        for index, char in enumerate(s):
            digit = int(char)
            # Flip the corresponding bit for the current digit
            # This keeps track of odd/even counts of digits in the substring
            current_state ^= 1 << digit

            # Check if this state occurred before to find a palindrome without a middle character
            if current_state in state_index_map:
                max_length = max(max_length, index - state_index_map[current_state])
            else:
                # Store the first occurrence of this state
                state_index_map[current_state] = index

            # Check all possible states differing by 1 bit
            # (represents palindromes with one middle character)
            for offset in range(10):
                potential_state = current_state ^ (1 << offset)
                if potential_state in state_index_map:
                    max_length = max(max_length, index - state_index_map[potential_state])

        # Return the length of the longest awesome substring found
        return max_length

```

Java Solution

```java
class Solution {
    public int longestAwesome(String s) {
        // This array will keep track of the first appearance of a binary representation of st
        int[] firstAppear = new int[1024];

        // Initialize array with -1 assuming that we haven't encountered any state yet
        Arrays.fill(firstAppear, -1);

        // This is our status tracker; it will keep track of the count of each digit in the prefix
        int currentState = 0;

        // Initialize the answer to be at least 1, as single digit is always awesome
        int longestAwesomeLength = 1;

        // Set the starting state to 0, which makes an empty string awesome since there are even counts
        firstAppear[0] = 0;

        // Iterate through each character of the string
        for (int i = 1; i <= s.length(); i++) {
            // Get the numeric value of the current character
            int digitVal = s.charAt(i - 1) - '0';

            // Update currentState by toggling the bit at the digitVal position
            currentState ^= 1 << digitVal;

            // Check if this state has occurred before
            if (firstAppear[currentState] >= 0) {
                // Calculate the length of the awesome substring and update the longestAwesomeLength
                longestAwesomeLength = Math.max(longestAwesomeLength, i - firstAppear[currentState]);
            } else {
                // If haven't met this state, set the current position as the first appearance
                firstAppear[currentState] = i;
            }

            // Check all the states that differ by one digit flip (this represents at most one odd count digit)
            for (int v = 0; v < 10; ++v) {
                // If a similar state has been encountered before, compare and update the longestAwesomeLength
                int analogousState = currentState ^ (1 << v);
                if (firstAppear[analogousState] >= 0) {
                    longestAwesomeLength = Math.max(longestAwesomeLength, i - firstAppear[analogousState]);
                }
            }
        }

        // Return the length of the longest awesome substring
        return longestAwesomeLength;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int longestAwesome(string s) {
        // Create a vector to store the first occurrence index of each state
        vector<int> firstOccurrenceIndex(1024, -1);
        // Initialize the first occurrence of state 0 to index 0
        firstOccurrenceIndex[0] = 0;
        // This will keep track of the current state of digit frequency parity
        int currentState = 0, maxLength = 1;
        // Iterate over the string characters, 1-indexed
        for (int i = 1; i <= s.size(); ++i) {
            // Convert current character to integer
            int digit = s[i - 1] - '0';
            // Toggle the bit corresponding to the current digit to update parity state
            currentState ^= 1 << digit;
            // Check if we have seen this state before
            if (firstOccurrenceIndex[currentState] != -1) {
                // Calculate max length if the same state has been encountered before
                maxLength = max(maxLength, i - firstOccurrenceIndex[currentState]);
            } else {
                // Record the first occurrence of this new state
                firstOccurrenceIndex[currentState] = i;
            }
            // Check states that differ by one digit (flip each digit's parity)
            for (digit = 0; digit < 10; ++digit) {
                int toggledState = currentState ^ (1 << digit);
                // Check if we have seen the toggled state before
                if (firstOccurrenceIndex[toggledState] != -1) {
                    // Calculate max length if the toggled state has been encountered before
                    maxLength = max(maxLength, i - firstOccurrenceIndex[toggledState]);
                }
            }
        }
        // Return the maximum length of awesome substring found
        return maxLength;
    }
};

```

Typescript Solution

```typescript
// This function calculates the longest awesome substring
function longestAwesome(s: string): number {
    // Create an array to store the first occurrence index of each state
    const firstOccurrenceIndex: number[] = new Array(1024).fill(-1);
    // Initialize the first occurrence of state 0 to index -1 to adjust for 0 indexing in the loop
    firstOccurrenceIndex[0] = -1;
    // This will keep track of the current state of digit frequency parity
    let currentState: number = 0;
    let maxLength: number = 1;

    // Iterate over the string characters, 0-indexed
    for (let i = 0; i < s.length; i++) {
        // Convert current character to integer
        const digit: number = parseInt(s[i], 10);
        // Toggle the bit corresponding to the current digit to update parity state
        currentState ^= 1 << digit;
        // If we haven't seen this state, set the index, otherwise calculate max length
        if (firstOccurrenceIndex[currentState] === -1) {
            firstOccurrenceIndex[currentState] = i + 1; // Store the index +1 to adjust for the initial value of state 0
        } else {
            maxLength = Math.max(maxLength, i + 1 - firstOccurrenceIndex[currentState]);
        }
        // Check states that differ by one digit (flip each digit's parity)
        for (let j = 0; j < 10; j++) {
            const toggledState: number = currentState ^ (1 << j);
            // If the toggled state has been seen, calculate max length
            if (firstOccurrenceIndex[toggledState] !== -1) {
                maxLength = Math.max(maxLength, i + 1 - firstOccurrenceIndex[toggledState]);
            }
        }
    }
    // Return the maximum length of the awesome substring found
    return maxLength;
}

// Example usage:
// const s: string = "3242415";
// const result: number = longestAwesome(s);
// console.log(result); // Outputs the length of the longest awesome substring

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the loops and the operations within them. There is a single loop running through the length of the string `s`. Inside the loop, the code performs a constant-time bitwise operation and checks for existence in a dictionary, which is generally considered to be O(1). Additionally, there is another loop within the main loop that iterates 10 times (constant) for each character in the string.

Thus, the overall time complexity is O(n) for the main loop multiplied by O(1) for operations within the loop and O(1) for dictionary lookup. Then, for every character, we loop a constant 10 times (assuming the cost of each iteration is constant), which does not change the linear time complexity.

Therefore, the total time complexity can be represented as `O(n)`.

Space Complexity

The space complexity is primarily due to the dictionary `d` which is used to store the previous encounter of a certain state of `st`. In the worst case, this dictionary can have an entry for each unique state `st` produces. Since `st` represents a bitmask of at most 10 digits (representing 10 different digits in the string), there can be at most `2^10` different states. Additionally, the integer `st` and variable `ans` are of constant size.

Hence, the space complexity is `O(1)` for the constant variables and `O(2^10)` for the dictionary, regardless of the size of the input string. Since `2^10` is a constant number (1024), this can also be considered constant space in the context of big O notation: `O(1)`.

Therefore, we can conclude that the space complexity is `O(1)`.