1763. Longest Nice Substring

Easy | Bit Manipulation | Hash Table | String | Divide and Conquer | Sliding Window | Leetcode Link

Problem Description

The challenge in this problem is to find the longest substring of the given string, `s`, where a "nice" substring is defined as one which contains every letter in both uppercase and lowercase forms. For example, a string "aA" is nice because it contains both 'a' and 'A'. If there are several such nice substrings, we are to return the one that occurs first. If no nice substrings exist, we should return an empty string.

Intuition

To solve this problem, we iterate through the string character by character, starting from each character in turn. We use two bitmask integers, `lower` and `upper`, to represent the presence of lowercase and uppercase letters, respectively. For each character in 'a' to 'z', a bit is set to 1 in `lower` if the lowercase letter is present, and correspondingly in `upper` if the uppercase letter is present.

For instance, if the lowercase letter 'b' is encountered, the second bit (assuming 0-indexing) of `lower` is set to 1. Similarly, if the uppercase letter 'B' is spotted, the second bit of `upper` is turned on.

We move through the string with a nested loop, checking consecutive characters starting from each index pointed by the outer loop and ending at the length of the string. While doing this, we keep updating our `lower` and `upper` bitmasks for each character we see. A substring is nice if the `lower` and `upper` bitmasks are equal at some point, meaning every lowercase character observed so far has a corresponding uppercase version present, and vice versa.

When we find such a substring, we check if it's longer than any previous "nice" substring found (initially none, as denoted by an empty string `ans`). If it's longer, we update our answer with the current substring.

This brute force approach checks all possible substrings starting at each point in the string, and while it's easy to understand and implement, its time complexity is not optimal for very long strings. However, it is perfectly viable for strings of moderate length and guarantees that the longest "nice" substring will be found.

Solution Approach

To implement the solution, we use a simple brute force approach which may not be the most efficient in terms of time complexity but is straightforward to understand and guarantees to find the correct answer. Here's a step-by-step breakdown:

1. Initialize an empty string `ans` to keep track of the longest nice substring found.
2. Iterate through the string `s` using a nested loop. The outer loop starts from each character indexed by `i` and the inner loop checks every subsequent character up to the end of the string, indexed by `j`.
3. For each character, two bitmasks `lower` and `upper` are maintained. If the character is lowercase (checked using `s[j].islower()`), we set the corresponding bit in `lower`. This is done by shifting 1 to the left by `ord(s[j]) - ord('a')` places (since 'a' is the ASCII starting point for lowercase letters, the difference gives us the correct bit position).
4. Similarly, if the character is uppercase, the corresponding bit in `upper` is set by shifting 1 to the left by `ord(s[j]) - ord('A')` places (as 'A' is the ASCII starting point for uppercase letters).
5. We compare `lower` and `upper` to check if the current substring is nice, which would be true if `lower` equals `upper`. This comparison realizes if for every lowercase letter there's a matching uppercase letter and vice versa.
6. If a nice substring is found and its length is greater than the length of the current `ans`, the `ans` string is updated to this substring. We achieve substring extraction using Python's slice notation `s[i : j + 1]`.
7. We repeat this process, expanding our current substring check from all possible starting points (`i`) to include all following characters (`j`). Since the answer should be of the earliest occurrence of the longest nice substring, by iterating from left to right we naturally prioritize earlier substrings over later ones of the same length.
8. Finally, after all iterations, `ans` holds the longest nice substring, and it is returned.

Throughout this implementation, we rely on basic bitwise operations, simple string manipulation, and nested loops. The algorithm's complexity is O(n^2) where n is the number of characters in the string. Each nested loop iteration checks one substring, and there are O(n^2) substrings in total.

Although not efficient for very large strings, for smaller strings, this simple and direct method effectively locates the desired nice substring without additional data structures or complex algorithms.

Example Walkthrough

Let's walk through an example to illustrate how the described solution approach is applied to the problem.

Consider the string `s = "aAbBcC"`. We are looking for the longest "nice" substring, that is, a substring where each letter exists in both uppercase and lowercase form.

1. We start with an empty string `ans` to hold the longest nice substring we find.
2. We begin by iterating over the string `s` with index `i` from 0 to the length of `s`. The first character is 'a', and we start the inner loop from `i` to check subsequent characters.
3. For each character `s[j]` we encounter as we move through the inner loop:
   - If `s[j].islower()` is true, for example `s[j] = 'a'`, we modify `lower` bitmask. We do `lower |= (1 << (ord('a') - ord('a')))`, resulting in `lower = 1` as 'a' is the first lowercase letter.
   - If `s[j].isupper()` is true instead, for example `s[j] = 'A'`, we modify the `upper` bitmask in a similar fashion, so `upper = 1`.
4. Continuing this process of updating `lower` and `upper` for each character in the inner loop, we reach the end of the string. By now, `lower` and `upper` should both be `111111` in binary, which corresponds to `63` in decimal, standing for the presence of 'a', 'b', and 'c' in lowercase and uppercase.
5. We check if `lower` equals `upper` after each inner loop iteration and update `ans` only if the current substring (`s[i : j + 1]`) is longer than `ans`. In this case, after the first full iteration (i = 0 to j = the end of `s`), the `ans` will become "aAbBcC", which is the entire string, since `lower == upper` is true.
6. To continue our process, we would next start with `i = 1`, but given that we already found a nice substring that includes the entire string, no longer substring is possible.
7. Completing the loops without finding a longer nice substring, we would still have `ans = "aAbBcC"`, which is indeed the longest nice substring that meets the criteria.

At the end of the algorithm, the `ans` string, which equals "aAbBcC", is returned as the correct answer.

This example demonstrates the scenario where the entire string meets the conditions to be a nice string. Thus, the first and longest nice substring is found on the very first iteration of the outer loop.

Python Solution

```python
class Solution:
    def longestNiceSubstring(self, s: str) -> str:
        n = len(s)  # length of the input string
        longest_nice_substring = ''  # Initialize the longest nice substring

        # Iterate over the string with two pointers
        for i in range(n):
            lower_case_flags = 0  # Bit flags for lowercase letters
            upper_case_flags = 0  # Bit flags for uppercase letters

            # Explore the substring starting from index i
            for j in range(i, n):
                if s[j].islower():
                    # Set the bit corresponding to the lowercase letter
                    lower_case_flags |= 1 << (ord(s[j]) - ord('a'))
                else:
                    # Set the bit corresponding to the uppercase letter
                    upper_case_flags |= 1 << (ord(s[j]) - ord('A'))

                # Check if the current substring is nice (lowercase and uppercase bits match)
                if lower_case_flags == upper_case_flags and len(longest_nice_substring) < j - i + 1:
                    # Update the longest nice substring if the current one is longer
                    longest_nice_substring = s[i : j + 1]

        # Return the longest nice substring found
        return longest_nice_substring

```

Java Solution

```java
class Solution {

    public String longestNiceSubstring(String inputString) {
        // Length of the input string
        int stringLength = inputString.length();

        // 'start' will keep the index at which the longest nice substring begins
        int start = -1;
        // 'maxLength' is the length of the longest nice substring found so far
        int maxLength = 0;

        // Iterate over each character in the string as the starting point
        for (int i = 0; i < stringLength; ++i) {
            // 'lowerCaseBitmask' and 'upperCaseBitmask' are bitmasks to keep track of
            // lowercase and uppercase characters encountered
            int lowerCaseBitmask = 0, upperCaseBitmask = 0;

            // Try extending the substring from the starting point 'i' to 'j'
            for (int j = i; j < stringLength; ++j) {
                // Get the current character
                char currentChar = inputString.charAt(j);

                // If it's lowercase, set the corresponding bit in the bitmask
                if (Character.isLowerCase(currentChar)) {
                    lowerCaseBitmask |= 1 << (currentChar - 'a');
                }
                // If it's uppercase, set the corresponding bit in the bitmask
                else {
                    upperCaseBitmask |= 1 << (currentChar - 'A');
                }

                // Check if the substring from 'i' to 'j' is a nice string
                // A nice string has the same set of lowercase and uppercase characters
                if (lowerCaseBitmask == upperCaseBitmask && maxLength < j - i + 1) {
                    // Update the maxLength and the starting index 'start'
                    maxLength = j - i + 1;
                    start = i;
                }
            }
        }

        // If 'start' was updated (meaning a nice substring was found), return it
        // Otherwise, if no nice substring exists, return an empty string
        return (start == -1) ? "" : inputString.substring(start, start + maxLength);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string longestNiceSubstring(string s) {
        // Initialize the size of the string.
        int strSize = s.size();
        // These will keep track of the start index of the longest nice substring
        // and its length.
        int startIndex = -1, maxLength = 0;

        // Iterate through each character as starting point of the nice substring.
        for (int i = 0; i < strSize; ++i) {
            // Bitmask to represent lowercase and uppercase letters encountered.
            int lowerBitmask = 0, upperBitmask = 0;

            // Explore substrings starting at index i.
            for (int j = i; j < strSize; ++j) {
                // Get the current character.
                char c = s[j];

                // Set the bit for this character in the appropriate bitmask.
                if (islower(c))
                    lowerBitmask |= 1 << (c - 'a');
                else
                    upperBitmask |= 1 << (c - 'A');

                // Check if the current substring is nice: it has both cases for each letter.
                // Also check if it's the longest so far.
                if (lowerBitmask == upperBitmask && maxLength < j - i + 1) {
                    maxLength = j - i + 1; // Update max length to the new longest nice substring.
                    startIndex = i; // Update start index to the starting index of the new longest nice substring.
                }
            }
        }

        // If no nice substring is found, return an empty string.
        // Otherwise, return the longest nice substring found.
        return startIndex == -1 ? "" : s.substr(startIndex, maxLength);
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the longest substring where each character appears in both lower and upper case.
 * @param {string} s - The input string to search for the nice substring.
 * @return {string} - The longest nice substring found in the input string.
 */
function longestNiceSubstring(s: string): string {
    const lengthOfString = s.length;
    let longestSubstring = '';

    // Iterate through the string to find all possible substrings
    for (let start = 0; start < lengthOfString; start++) {
        let lowerCaseMask = 0;  // Bitmap for tracking lowercase letters
        let upperCaseMask = 0;  // Bitmap for tracking uppercase letters

        // Explore the substrings starting from 'start' index
        for (let end = start; end < lengthOfString; end++) {
            const charCode = s.charCodeAt(end);

            // If the character is lowercase, update the lowerCaseMask
            if (charCode > 96) {
                lowerCaseMask |= 1 << (charCode - 97);
            }
            // If the character is uppercase, update the upperCaseMask
            else {
                upperCaseMask |= 1 << (charCode - 65);
            }

            // Check if the current substring is "nice" and if it is longer than the current longest
            if (lowerCaseMask === upperCaseMask && end - start + 1 > longestSubstring.length) {
                longestSubstring = s.substring(start, end + 1);
            }
        }
    }

    // Return the longest nice substring found
    return longestSubstring;
}

```

Time and Space Complexity

The given Python code snippet is designed to find the longest substring of a given string `s` such that the substring is "nice". A substring is considered "nice" if it contains both the uppercase and lowercase forms of the same letter.

Time Complexity

The time complexity of the code is determined by the nested for-loops. The outer loop runs `n` times where `n` is the length of string `s`. The inner loop runs at most `n` times for each iteration of the outer loop as it starts from the current position of `i` to the end of the string `s`. This gives us a quadratic number of iterations in the worst case.

The operations inside the inner loop are constant time operations, such as checking if a character is lower or uppercase and setting bits in an integer. Hence, they don't affect the time complexity's order.

Therefore, the overall time complexity of the code is `O(n^2)`.

Space Complexity

The space complexity includes the variables to store the bitmasks for lowercase (`lower`) and uppercase (`upper`) letters, a variable for the answer (`ans`), and two loop variables (`i` and `j`). The bitmasks `lower` and `upper` use fixed space since there are exactly 26 lowercase and 26 uppercase English letters, and they can be represented within a fixed-size integer type.

Since the algorithm's space usage does not scale with the size of the input string `s`, besides the output string `ans`, the space complexity is `O(1)` for the working variables. However, if we consider the space used by the output `ans`, it could be `O(n)` in the case when the whole string `s` is a "nice" substring itself. Therefore, the overall space complexity, including the space for the output, is `O(n)`.