2309. Greatest English Letter in Upper and Lower Case

Easy Hash Table String Enumeration Leetcode Link

Problem Description

Given a string `s` consisting of English letters, the task is to find the largest letter (by the position in the alphabet) that exists both in lowercase and uppercase in the string `s`. The result should be the uppercase version of this letter. If there are no such letters that exist in both cases, return an empty string. The concept of being "larger" is determined by the standard order of letters in the English alphabet, meaning 'B' is greater than 'A', 'C' is greater than 'B', and so on up to 'Z'.

Intuition

For an efficient solution, one could use bit manipulation to keep track of which letters are present in the string `s` in both lowercase and uppercase forms. To do so, you can create two bit masks: one for lowercase letters (`mask1`) and one for uppercase letters (`mask2`). The idea is to use each bit in these masks to represent the presence of each letter in the alphabet— for example, the least significant bit would represent 'a' or 'A', the second least significant bit 'b' or 'B', and so on.

As you iterate through the string, if you encounter a lowercase letter, you calculate its positional index (by subtracting the ASCII value of 'a' from that of the letter) and then set the corresponding bit in `mask1`. Similarly, for an uppercase letter, you calculate the index (subtracting 'A') and set the corresponding bit in `mask2`.

After scanning all letters in the string and updating the bit masks, you compute the bitwise AND of the two masks (`mask1` & `mask2`). This will give you a new mask that has bits set only at positions where both a lowercase and uppercase instance of that letter exists in `s`.

The last part is to find the highest set bit in the combined mask, which represents the greatest letter present in both forms. The `bit_length` method is used to find the position of the highest set bit. Since you're looking for the actual letter, not the index, you have to add the ASCII value of 'A' to convert the index back to the character. In the event where no bits are set in the combined mask, it indicates there is no such letter that meets the criteria, and an empty string should be returned.

Solution Approach

The solution approach uses bit manipulation techniques to efficiently track whether each letter in the English alphabet appears as both a lowercase and uppercase in the string `s`.

Algorithm Steps:

1. Initialize two variables `mask1` and `mask2` to `0`. Each will serve as a bitmask to track the occurrence of lowercase and uppercase letters, respectively.
2. Loop through each character `c` in the string `s`.
3. For each character `c`, determine if it's lowercase or uppercase.
   - If it is lowercase, find the difference between the ASCII values of `c` and `'a'`. This difference gives the index (0 for 'a', 1 for 'b', ..., 25 for 'z').
   - Set the corresponding bit in `mask1` by shifting `1` left by the index calculated above. This is done with `mask1 |= 1 << (ord(c) - ord("a"))`.
4. If the character is uppercase, perform a similar operation as step 3 but on `mask2` by finding the difference between the ASCII values of `c` and `'A'`.
5. After completing the loop, use a bitwise AND operation between `mask1` and `mask2` to find common letters (bits set in both masks). `mask = mask1 & mask2`.
6. Find the greatest (most significant) bit set in `mask`. This is done using `mask.bit_length()`, which returns the number of bits necessary to represent `mask` in binary, which is also the position of the highest bit set plus one.
7. If `mask` is not zero, convert the index of the greatest bit set back to an uppercase letter using `chr(mask.bit_length() - 1 + ord("A"))`.
8. If `mask` is zero, which means no bit is set and no such common letter in both cases exists, return an empty string.

Data Structures:

- Two integer bit masks (variables `mask1` and `mask2`) to track the presence of each lowercase and uppercase letter, efficiently using bitwise operations.

Patterns Used:

- **Bit Manipulation**: To compactly store and compute the presence of letters and to find the greatest letter present in both forms.
- **ASCII Value Calculations**: To map characters to their respective positions in the bitmask and vice versa.

By following this approach, the algorithm uses constant space regardless of the size of the input string and linear time in terms of the length of the string `s`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the string `s = "aAbBdDxXyYzZcC"` and we want to find the largest letter that exists both in lowercase and uppercase in this string.

1. We initialize `mask1` and `mask2` to `0`. These will track lowercase and uppercase letters, respectively.
2. We start looping through the string `s`. The first character is `'a'`, which is lowercase.
3. We calculate the index for `'a'` by subtracting the ASCII value of `'a'` from it (which is 0) and set the corresponding bit in `mask1` (so `mask1` now has the least significant bit set).
4. We encounter `'A'`, which is uppercase. We calculate its index (also 0) and set the corresponding bit in `mask2`.
5. Continuing this process, `mask1` and `mask2` will have bits set based on the lowercase and uppercase letters, respectively.

   After processing all characters, suppose `mask1 = 0011001100010101` and `mask2 = 0011001100010101`.
6. We apply a bitwise AND operation on `mask1` and `mask2`, which gives us `mask = 0011001100010101`, showing that we have common letters.
7. We use `mask.bit_length()` to find the highest set bit, which in this case would return `14`. This means the letter we are looking for corresponds to the bit just before the 14th position in the mask (because `bit_length` returns the position of the highest set bit plus one).
8. We subtract 1 from this value to get the actual index of the letter in the mask and add the ASCII value of `'A'` to convert it back to the character, giving us `'Z'`, which is our answer. The calculation will look like `chr(14 - 1 + ord('A'))`.

If no common uppercase and lowercase letter was present in `s`, `mask` would be `0` and we would return an empty string. But in this example, we found `'Z'` as the largest letter that exists in both cases, so the final result is `'Z'`.

Python Solution

```python
class Solution:
    def greatestLetter(self, s: str) -> str:
        # Initialize bit masks for lowercase and uppercase letters.
        lowercase_mask = 0
        uppercase_mask = 0

        # Iterate through each character in the string.
        for char in s:
            # If the character is lowercase,
            # set the corresponding bit in the lowercase bit mask.
            if char.islower():
                lowercase_mask |= 1 << (ord(char) - ord('a'))
            # If the character is uppercase,
            # set the corresponding bit in the uppercase bit mask.
            else:
                uppercase_mask |= 1 << (ord(char) - ord('A'))

        # Calculate the intersection bit mask to find common letters in upper and lowercase.
        common_letters_mask = lowercase_mask & uppercase_mask

        # If there are common letters, find the greatest one.
        if common_letters_mask:
            # Calculate the most significant bit's position and convert it back to an uppercase letter.
            # The 'bit_length' method returns the position of the highest 1 bit,
            # corresponding to the largest letter available in both cases.
            greatest_letter_ascii = common_letters_mask.bit_length() - 1 + ord('A')
            return chr(greatest_letter_ascii)
        else:
            # If there are no common letters, return an empty string.
            return ""

```

Java Solution

```java
class Solution {
    public String greatestLetter(String s) {
        // Initialize two masks to keep track of lower and upper case letters.
        int lowerCaseMask = 0, upperCaseMask = 0;

        // Iterate through each character in the string.
        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i);

            // If the character is a lower case letter, update the lowerCaseMask.
            if (Character.isLowerCase(c)) {
                lowerCaseMask |= 1 << (c - 'a');
            }
            // If the character is an upper case letter, update the upperCaseMask.
            else {
                upperCaseMask |= 1 << (c - 'A');
            }
        }

        // Calculate the common mask to identify letters appearing in both cases.
        int commonMask = lowerCaseMask & upperCaseMask;

        // If there is at least one letter in commonMask, calculate and return the greatest letter.
        if (commonMask > 0) {
            // Find the largest index of the set bit which represents the greatest letter.
            // '31 - Integer.numberOfLeadingZeros(commonMask)' gives the last (greatest) index where the bit is set.
            // Adding 'A' converts it back to the ASCII character.
            return String.valueOf((char) (31 - Integer.numberOfLeadingZeros(commonMask) + 'A'));
        } else {
            // If commonMask is zero, it means no letter exists in both cases, return an empty string.
            return "";
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This method finds the greatest letter that appears both in lower and upper case in the input string.
    string greatestLetter(string s) {
        // Initialize two bitmasks to track the presence of 26 lowercase and 26 uppercase letters.
        int lowercaseMask = 0;
        int uppercaseMask = 0;

        // Iterate through each character of the string
        for (char &c : s) {
            // If the character is lowercase
            if (islower(c)) {
                // Set the corresponding bit in the lowercase mask
                lowercaseMask |= 1 << (c - 'a');
            }
            // If the character is uppercase
            else {
                // Set the corresponding bit in the uppercase mask
                uppercaseMask |= 1 << (c - 'A');
            }
        }

        // The 'mask' will only have bits set that are common in both uppercase and lowercase masks
        // It represents letters existing in both cases
        int mask = lowercaseMask & uppercaseMask;

        // If there is at least one common letter
        if (mask != 0) {
            // __builtin_clz returns the number of leading 0-bits, hence 31 - __builtin_clz(mask)
            // Gives the position of the highest-order bit that is set to 1
            // Then we add 'A' to get the ASCII value of the greatest letter
            // that appears in both lower and uppercase in the string 's'.
            return string(1, 'A' + 31 - __builtin_clz(mask));
        } else {
            // If there is no common letter, return an empty string
            return "";
        }
    }
};

```

Typescript Solution

```typescript
function greatestLetter(s: string): string {
    // Initialize an array to track the occurrence of characters in the string,
    // sized to fit the ASCII table for uppercase and lowercase English letters.
    const seenCharacters = new Array(128).fill(false);

    // Iterate through each character in the provided string,
    // and record its occurrence in the seenCharacters array
    // by setting the corresponding ASCII index to true.
    for (const char of s) {
        seenCharacters[char.charCodeAt(0)] = true;
    }

    // Iterate backwards from ASCII code 90 (Z) to 65 (A) to find the largest lexicographical
    // uppercase letter that has both uppercase and lowercase forms present in the string.
    for (let i = 90; i >= 65; --i) {
        // Check if current letter i is present in both its uppercase and lowercase forms.
        if (seenCharacters[i] && seenCharacters[i + 32]) {
            // If a valid letter is found, return the uppercase character as the result.
            return String.fromCharCode(i);
        }
    }

    // If no such letter is found, return an empty string.
    return '';
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is dominated by the loop running through every character in the input string, `s`. Each iteration performs a constant amount of work: bitwise operations and comparisons. As `s` can be of length `n`, the loop runs `n` times. Hence, the time complexity is `O(n)`.

Space Complexity

The space complexity of the code is determined by the two integer variables `mask1` and `mask2`. Regardless of the length of the input string, these variables occupy a constant amount of space, as they are not dependent on the input size. Thus, the space complexity is `O(1)` because the space used by the algorithm does not scale with the input size.