2351. First Letter to Appear Twice

Easy Hash Table String Counting

Problem Description

The problem provides us with a string `s` which is composed of lowercase English letters. Our task is to find the first letter that appears twice in the string. It's important to understand that by 'first', it means the letter whose second occurrence comes before the second occurrence of any other character that might also appear more than once. The string is guaranteed to have at least one character that meets this condition.

Intuition

To solve this problem efficiently, we consider each character in the string one by one and keep track of the ones we have already seen. The approach is to use a bitmask to keep track of the characters that have occurred once. Here's the intuition:

- We initialize an integer `mask` to 0 to use as our bitmask.
- For each character `c` in the string `s`, we convert it to an integer index `i` by subtracting the ASCII value of 'a' from the ASCII value of `c`. This maps 'a' to 0, 'b' to 1, and so on up to 'z'.
- We then check if the ith bit in the `mask` is already set (which would mean we've seen this character before). This is done by shifting 1 left by `i` positions, ANDing it with `mask`, and checking if the result is not zero.
- If we find that the `mask` already has the ith bit set, it means that this character `c` is the first character to appear twice, so we return it.
- If not, we set the ith bit in the `mask` to indicate that we have seen the current character `c` for the first time.

Solution Approach

The solution approach uses bitwise operators to implement an efficient algorithm to find the first repeating character in the string `s`. Let's dive into each part of the implementation:

- **Initializing the Bitmask**: A variable `mask` is initialized to 0. This `mask` will be used to keep track of characters that have been seen once in the string. The ith bit in `mask` will correspond to the ith character in the alphabet (where 'a' is 0, 'b' is 1, etc.).
- **Iterating through the String**: The code uses a `for` loop to iterate through each character `c` of the string `s`.
- **Mapping Character to Bit Position**: For each character `c`, we calculate an index `i` that represents its position in the alphabet (`i = ord(c) - ord('a')`). This index `i` is then used to check or set the corresponding bit in the `mask`.
- **Checking for Repeats**: To check if a character has appeared before, we shift 1 to the left by `i` positions (`1 << i`) and perform a bitwise AND with the `mask`. If the result is nonzero (`mask >> i & 1`), the character has been seen before, and it is the first character to appear twice.
- **Setting Bits for Unseen Characters**: If the character has not been seen before, we set its corresponding bit in the `mask` using a bitwise OR with `1 << i` (`mask |= 1 << i`).
- **Returning the Result**: Once we find a character whose bit is already set in the `mask`, we return that character since it is the first character to occur twice.

This algorithm efficiently solves the problem in `O(n)` time complexity, where `n` is the length of the string `s`. Because we only go through the string once and the operations for each character are constant time. Moreover, it does not require additional data structures like hash maps or arrays to store counts or indexes of characters, thereby offering `O(1)` space complexity, aside from the input string.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the string `s = "abccba"`:

1. **Initializing the Bitmask**: Start with a `mask` equal to 0. This mask will be used to track each letter we encounter in `s`.
2. **Iterating through the String**: We go through each letter in `s` one-by-one:

   a. **First Letter - 'a'**: For `c = 'a'`:
   - `i` is computed as `ord('a') - ord('a')` which is `0`.
   - Check if `mask & (1 << 0)` is non-zero. It's not; it's zero since `mask` is currently 0.
   - Set the 0th bit of `mask` to denote that we've seen 'a': `mask |= 1 << 0`. Now `mask = 1`.

   b. **Second Letter - 'b'**: For `c = 'b'`:
   - `i` is computed as `ord('b') - ord('a')` which is `1`.
   - Check if `mask & (1 << 1)` is non-zero. It's not; hence, 'b' hasn't appeared before.
   - Set the 1st bit of `mask`: `mask |= 1 << 1`. Now `mask = 3`.

   c. **Third Letter - 'c'**: For `c = 'c'`:
   - `i` is computed as `ord('c') - ord('a')` which is `2`.
   - Check if `mask & (1 << 2)` is non-zero. It's not; hence, 'c' hasn't appeared before.
   - Set the 2nd bit of `mask`: `mask |= 1 << 2`. Now `mask = 7`.

   d. **Fourth Letter - 'c'** (repeat): For `c = 'c'` again:
   - `i` is computed as `ord('c') - ord('a')` again, which is `2`.
   - Check if `mask & (1 << 2)` is non-zero. It is non-zero since the 2nd bit in `mask` is already set.
   - We've found the first letter that appears twice in the string, so we return 'c'.
3. **Returning the Result**: The first repeating character is `'c'`, as its second occurrence is before the second occurrence of any other letter. Therefore, `'c'` is the output.

Understand that every time we "set a bit" in the mask, we transform the mask to represent the letters we've seen. For 'a', the binary representation of `mask` becomes `000001`, for 'b' it becomes `000011`, and for 'c' we have `000111`. When the second 'c' is encountered, the corresponding bit is already set, and because bitwise AND of `mask` and `1 << 2` results in a non-zero value, we know that 'c' is a repeating character.

Solution Implementation

Python

```python
class Solution:
    def repeatedCharacter(self, s: str) -> str:
        # Initialize a mask to keep track of characters encountered
        encountered_chars_mask = 0

        # Loop through each character in the string
        for char in s:
            # Calculate the position of the character in the alphabet
            # 'a' maps to 0, 'b' maps to 1, ..., 'z' maps to 25
            alphabet_pos = ord(char) - ord('a')

            # Check if the bit at the position of the current character is already set
            if encountered_chars_mask & (1 << alphabet_pos):
                # If the bit is set, the current character has been encountered before
                return char

            # Set the bit at the position of the current character in the mask,
            # indicating the character has now been encountered
            encountered_chars_mask |= 1 << alphabet_pos

        # If the function exits the loop without returning, no repeated character was found
        # (Although given the context of this function, it seems that an assumption is made
        # that there will always be at least one repeated character in the string.)
```

Java

```java
class Solution {
    // This method finds the first repeated character in a given string
    public char repeatedCharacter(String s) {
        // 'mask' is used to store the information about characters seen so far
        int mask = 0;
        // Loop through each character in the string
        for (int i = 0; i < s.length(); ++i) {
            // Get the current character
            char c = s.charAt(i);
            // Calculate the bit position for the current character
            int charPosition = c - 'a';
            // Check if the current character has already been seen
            if ((mask & (1 << charPosition)) != 0) {
                // If the character has been seen before, return it as the first repeated character
                return c;
            }
            // Set the bit for the current character in the 'mask' to mark it as seen
            mask |= 1 << charPosition;
        }
        // This line is never reached since the problem statement guarantees at least one repetition
        return '\0'; // Placeholder for compile-time checks, it represents a null character
    }
}
```

C++

```cpp
#include <string> // Include necessary header

class Solution {
public:
    // Method to find the first recurring character in a string
    char repeatedCharacter(string s) {
        int charBitmask = 0; // Initialize a bitmask to keep track of characters seen

        // Loop through the characters of the string
        for (int i = 0; i < s.length(); ++i) {
            int charIndex = s[i] - 'a'; // Calculate the index of the current character (0-25 for a-z)

            // Check if this character has been seen before by checking the bitmask.
            // If the corresponding bit is set, we've found a repeated character
            if ((charBitmask >> charIndex) & 1) {
                return s[i]; // Return the repeated character
            }

            // Set the bit corresponding to this character in the bitmask
            // to mark that it has now been seen
            charBitmask |= 1 << charIndex;
        }

        // If no repeated character is found by the end of the string
        // (which shouldn't happen as per the method's contract),
        // the loop would exit by an exception since the exit condition of the loop is never true.
        // To avoid the potential of reaching the end of the function without a return value,
        // you can return a placeholder here (e.g., 0) or handle the error according to specifications.
        throw std::logic_error("No repeated characters in input string.");
    }
};
```

TypeScript

```typescript
/**
 * Finds the first repeated character in a given string.
 *
 * @param {string} targetString - The string to search for repeated characters.
 * @returns {string} - The first repeated character if found; otherwise, a space character.
 */
function repeatedCharacter(targetString: string): string {
    // Initialize a bitmask to track the presence of characters.
    let charPresenceMask = 0;

    // Iterate through each character in the given string.
    for (const char of targetString) {
        // Calculate the bit position by getting ASCII value difference
        // from the target char to the 'a' character.
        const bitPosition = char.charCodeAt(0) - 'a'.charCodeAt(0);

        // Check if the bit at the calculated position is already set in the mask.
        // If it is, we have found a repeated character.
        if (charPresenceMask & (1 << bitPosition)) {
            return char; // Return the repeated character.
        }

        // Update the mask to include the current character.
        charPresenceMask |= 1 << bitPosition;
    }

    // If no repeated character was found, return a space character.
    return ' ';
}


class Solution:
    def repeatedCharacter(self, s: str) -> str:
        # Initialize a mask to keep track of characters encountered
        encountered_chars_mask = 0

        # Loop through each character in the string
        for char in s:
            # Calculate the position of the character in the alphabet
            # 'a' maps to 0, 'b' maps to 1, ..., 'z' maps to 25
            alphabet_pos = ord(char) - ord('a')

            # Check if the bit at the position of the current character is already set
            if encountered_chars_mask & (1 << alphabet_pos):
                # If the bit is set, the current character has been encountered before
                return char

            # Set the bit at the position of the current character in the mask,
            # indicating the character has now been encountered
            encountered_chars_mask |= 1 << alphabet_pos

        # If the function exits the loop without returning, no repeated character was found
        # (Although given the context of this function, it seems that an assumption is made
        # that there will always be at least one repeated character in the string.)
```

Time and Space Complexity

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the length of the input string `s`. This is because the code iterates over each character of the string exactly once. Within the loop, the operations—calculating the index `i`, shifting the bitmask, performing the bitwise AND and OR operations—are all constant time operations, i.e., `O(1)`.

Space Complexity:

The space complexity of the code is `O(1)`. This is because the space used by the variables `mask` and `i`, and any other temporary space, is constant and does not depend on the input size. The bitmask `mask` is an integer value that only requires a fixed amount of space, and regardless of the size of the string, it does not require additional space as the input size grows.