2405. Optimal Partition of String

Medium Greedy Hash Table String

Leetcode Link

Problem Description

This problem asks to partition a given string `s` into the minimum number of substrings where each substring contains unique characters; no character is repeated within a single substring. All characters in the original string must be used exactly once in some substring, and no character can be in more than one substring. The goal is to find and return the smallest possible number of such unique-character substrings.

Intuition

The solution approach is to iterate over the characters of the string while keeping track of characters we have seen in the current substring. There are different strategies to keep track of the unique characters. For example, we could use a set, an array, or, as in the provided solution, a bit vector (which is an integer that represents a set of characters using bits).

Here's the rundown of the intuition:

- We initialize a counter (`ans`) for the number of substrings to 1 because we need at least one substring.
- We also initialize a bit vector (`v`) to represent the characters we've seen in the current substring.
- We iterate through the string character by character.
- For each character, we convert it to a unique integer representing its position in the alphabet (`i`), where 'a' corresponds to 0, 'b' to 1, and so on. This is simply the ASCII value of the character minus the ASCII value of 'a'.
- We use bitwise operations to check if the bit at position `i` in the bit vector `v` is already set, which would imply that the character has already appeared in the current substring.
- If the character is already in the substring (i.e., the bit is set), we must start a new substring. So, we reset the bit vector to 0, and increment our substring counter (`ans`) by 1.
- Regardless of whether we've started a new substring, we now add the current character to the current substring by setting the bit at the corresponding position `i` in the bit vector `v` (using the bitwise OR operation).
- After processing all characters, the counter `ans` holds the minimum number of substrings needed, which we return.

Using a bit vector is particularly efficient in terms of space since it replaces a potentially large set or array with a single integer and makes use of fast bitwise operations to manage the set of characters.

Solution Approach

The implementation of the solution involves a bit manipulation strategy to keep track of which characters have been seen in the current substring. Let's delve into the details of the algorithm and the data structures used:

- **Bit Vector:** A bit vector is the primary data structure used here. It is simple, yet powerful for tracking the presence of characters. A 32-bit integer can be used as a bit vector because the lower 26 bits can represent each letter of the English alphabet.
- **Iterating Over the String:** The solution iterates over each character in the given string `s`. For each character, the following steps are performed:
  - Convert the character to an index:

    ```
    i = ord(c) - ord('a')
    ```

    This uses the `ord()` function to get the ASCII value of `c` and then subtracts the ASCII value of `'a'` to get a zero-based index `i`, where `0` would represent `'a'` and `25` would represent `'z'`.
  - Check for Character's Presence:

    ```
    if (v >> i) & 1:
        v = 0
        ans += 1
    ```

    This checks if the bit at position `i` of the bit vector `v` is already set, which signifies that the character has been seen previously in the current substring. The solution uses bitwise right shift (`>>`) to move the bit of interest to the least significant bit position, and bitwise AND (`&`) with `1` to isolate that bit. If this results in a value of `1`, indicating the character is a duplicate in the current substring, the counter `ans` is incremented by 1, and `v` is reset to 0, signaling the start of a new substring.
  - Update the Bit Vector:

    ```
    v |= 1 << i
    ```

    After potentially starting a new substring, the bit at position `i` is set in `v` using a bitwise OR (`|`). The expression `1 << i` creates an integer with only the `i`'th bit set, and then `v` is updated to include that bit, adding the character to the current substring's character set.
- **Returning the Final Answer:** At the end of the loop, once all characters in `s` have been processed, the `ans` variable holds the number of substrings formed and is returned as the final answer.

This approach effectively utilizes bit manipulation to compactly track characters and manage substrings, making the solution both space-efficient and fast due to the nature of bitwise operations being low-level and quick on modern computer architectures.

Example Walkthrough

To illustrate the solution approach, let's walk through a small example. Consider the string `s = "abac"`. We want to partition this string into the minimum number of substrings with unique characters.

1. **Initialize Counters**: Let's initialize our substring counter `ans` to 1 and our bit vector `v` to 0.
2. **Process First Character**: Starting with the first character `'a'`:
   - The index `i` for `'a'` is `0` (`ord('a') - ord('a')`).
   - We check if bit `0` in `v` is already set by checking `(v >> i) & 1`. Since `v` is `0`, the bit is not set.
   - We then set bit `0` in `v` to mark `'a'` as seen (`v` becomes `1` after `v |= 1 << i`).
3. **Process Second Character**: Next is `'b'`:
   - The index `i` for `'b'` is `1` (`ord('b') - ord('a')`).
   - We check if bit `1` in `v` is set, which it's not.
   - We set bit `1` in `v` (`v` becomes `3` after `v |= 1 << i`).
4. **Process Third Character**: Now we encounter `'a'` again:
   - The index for `'a'` is still `0`.
   - Checking `(v >> i) & 1`, we find that the bit is set since `v` is `3` (binary `11`), which means `'a'` was already included in the current substring.
   - Since `'a'` is a repeating character, we increment `ans` to `2` and reset `v` to `0`.
   - We then set bit `0` in `v` again (`v` becomes `1`).
5. **Process Fourth Character**: Lastly, we have `'c'`:
   - The index `i` for `'c'` is `2` (`ord('c') - ord('a')`).
   - We check if bit `2` in `v` is set, which it's not.
   - We set bit `2` in `v` (`v` becomes `5` after `v |= 1 << i`).

After processing all characters in `s`, we have an `ans` of `2`, indicating the given string `"abac"` can be partitioned into 2 substrings with unique characters, which are `"ab"` and `"ac"`.

This example demonstrates how the solution efficiently tracks characters using bit manipulation to ascertain when a new substring needs to be started, ensuring that each substring contains unique characters.

Python Solution

```python
class Solution:
    def partitionString(self, s: str) -> int:
        # Initialize 'partitions' as the counter for required partitions and
        # 'used_characters' to keep track of the characters used in the current partition.
        partitions = 1
        used_characters = 0

        # Iterate over each character in the string.
        for char in s:
            # Calculate the bit index corresponding to the character.
            char_index = ord(char) - ord('a')

            # Check if the character has already been used in the current partition.
            # (v >> i) & 1 checks if the ith bit in 'used_characters' is set to 1,
            # meaning that character was already used.
            if (used_characters >> char_index) & 1:
                # If so, we need a new partition. Reset 'used_characters' and
                # increment the 'partitions' counter.
                used_characters = 0
                partitions += 1

            # Set the bit at the character's index to 1 to mark it as used for the current partition.
            used_characters |= 1 << char_index

        # Return the number of partitions needed such that no two partitions have any characters in common.
        return partitions
```

Java Solution

```java
class Solution {
    public int partitionString(String s) {
        // 'bitMask' is used to keep track of the characters seen in the current partition
        int bitMask = 0;

        // 'partitionsCount' represents the number of partitions needed
        int partitionsCount = 1;

        // Iterate over each character in the string
        for (char character : s.toCharArray()) {
            // Calculate the bit number corresponding to 'character'
            int bitNumber = character - 'a';

            // 'bitMask >> bitNumber' shifts the bitMask 'bitNumber' times to the right
            // '& 1' checks if the bit at position 'bitNumber' is set to 1, i.e., if character is already seen
            if (((bitMask >> bitNumber) & 1) == 1) {
                // If the character has been seen, reset bitMask for a new partition
                bitMask = 0;

                // Increment partition count as we're starting a new partition
                ++partitionsCount;
            }

            // '|=' is the bitwise OR assignment operator. It sets the bit at position 'bitNumber' in bitMask to 1
            // '1 << bitNumber' creates a bitmask with only bitNumber'th bit set
            // This indicates that character is included in the current partition
            bitMask |= 1 << bitNumber;
        }

        // Return the total number of partitions needed
        return partitionsCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine the number of substrings without repeating characters
    int partitionString(string s) {
        int numPartitions = 1;  // Initialize the count of partitions to 1
        int charBitset = 0;     // Use a bitset to track the occurrence of characters

        // Iterate through each character in the string
        for (char& c : s) {
            int charIndex = c - 'a';  // Convert character to its corresponding bit index

            // Check if the character has already been seen in the current partition
            // by checking the corresponding bit in the bitset
            if (!!(charBitset >> charIndex) & 1) {
                // If the character is repeated, start a new partition
                charBitset = 0;  // Reset the bitset for the new partition
                ++numPartitions;  // Increment the count of partitions
            }

            // Mark the current character as seen in the bitset for the current partition
            charBitset |= 1 << charIndex;
        }

        // Return the total number of partitions found
        return numPartitions;
    }
};
```

Typescript Solution

```typescript
// Function that calculates the minimum number of partitions
// of a string such that each letter appears in at most one part
function partitionString(s: string): number {
    // Initialize a Set to store unique characters
    const uniqueChars = new Set();

    // Initialize variable to count partitions
    let partitionCount = 1;

    // Iterate over each character in the string
    for (const char of s) {
        // If the character is already in the set, increment the partition count
        // and clear the set to start a new partition with unique characters
        if (uniqueChars.has(char)) {
            partitionCount++;
            uniqueChars.clear();
        }

        // Add the current character to the set
        uniqueChars.add(char);
    }

    // Return the total number of partitions
    return partitionCount;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is primarily determined by the loop through the string `s`. Since each character in the string is checked exactly once, the time complexity is O(n), where `n` is the length of the string `s`.

During each iteration, the operations performed include basic bitwise operations (`>>`, `&`, `|`, and `<<`), which are O(1) operations, and thus do not affect the linear nature of the time complexity.

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

Space Complexity

The space complexity of the algorithm is due to the storage required for the variable `v`, which serves as a mask to track the unique characters.

Since `v` is a single integer and its size does not grow with the size of the input string `s`, the space complexity of storing this integer is constant O(1), regardless of the length of the string.

Additionally, since there are no other data structures used that scale with the input size, the overall space complexity remains constant.

Hence, the space complexity of the code is `O(1)`.