1239. Maximum Length of a Concatenated String with Unique Characters

Medium Bit Manipulation Array String Backtracking Leetcode Link

Problem Description

In this problem, you're given an array of strings named `arr`. Your task is to create the longest possible string `s` by concatenating some or all strings from `arr`. However, there's an important rule: the string `s` must consist of unique characters only.

In other words, you need to pick a subsequence of strings from `arr` such that when these strings are combined, no character appears more than once. The length of the resulting string `s` is your main objective, and you must return the maximum possible length. Remember that a subsequence is formed from the original array by potentially removing some elements, but the order of the remaining elements must be preserved.

Intuition

To find the optimal solution for the maximum length of `s`, we use a bit manipulation technique known as "state compression". This technique involves using a 32-bit integer to represent the presence of each letter where each bit corresponds to a letter in the alphabet.

The intuition behind this approach is that it allows us to efficiently check whether two strings contain any common characters. We can do this by performing an AND operation (`&`) on their respective compressed states (masks). If the result is zero, it means there are no common characters.

Here's a step-by-step breakdown of how we arrive at the solution:

1. Initialize a variable `ans` to zero. This will hold the maximum length found.
2. Create a list `masks` with a single element, zero, to keep track of all unique character combinations we've seen so far.
3. Iterate through each string `s` in the array `arr`.
   - For each string, create a bit mask `mask` that represents the unique characters in the string.
   - If the string has duplicate characters, we set `mask` to zero and skip it since such a string cannot contribute to a valid `s`.
4. Iterate through the current masks in `masks`.
   - We use the AND operation to check if the current string's mask has any overlap with `mask`. If it doesn't (i.e., the result is zero), it means we can concatenate this string without repeating any character.
   - In such a case, we combine (OR operation `|`) the current mask with `mask` and add the new mask to our `masks` list.
   - Update `ans` with the count of set bits in the new mask, which represents the length of the unique character string formed up to this point. This is done using the built-in `bit_count()` method on the new mask.
5. Finally, return the maximum length `ans` found during the process.

The use of state compression and bit manipulation makes the solution efficient, as it simplifies the process of tracking which characters are present in the strings and eliminates the need for complex data structures or string operations.

Solution Approach

The solution uses two significant concepts: **bit manipulation** for state compression and **backtracking** to explore all combinations. The algorithm follows these steps:

1. Initialize an integer `ans` with the value `0`, which will track the maximum length of a string with unique characters found during the process.

2. We begin with an array `masks` that starts with a single element, `0`, to record the baseline state (the empty string).

3. Loop through each string `s` in the input array `arr`.

   - For each string `s`, we create an integer `mask` that serves as its unique character identifier. The binary representation of `mask` will have a `1` in the position corresponding to a letter (where `a` is the least significant bit, and `z` is the most significant).

   - As we iterate over each character `c` in the string `s`, we calculate the difference between the ASCII value of `c` and that of `'a'` to find the corresponding bit position.

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

   - We then check if the bit at that position is already set. If so, it means the character has appeared before, and we break the loop, setting `mask` to `0`, as this string cannot be part of the result due to the duplicate character.

```python
if mask >> i & 1:
    mask = 0
    break
```

   - Otherwise, we set the bit corresponding to the character in the `mask`.

```python
mask |= 1 << i
```

   - If the current string `s` has any duplicate characters, we disregard this `mask` and move on to the next string in `arr`.

4. Next, for each `mask` computed from the current string, we examine the masks collected so far.

   - For each existing `m` in `masks`, we ensure there's no overlap between `m` and the current `mask` using the bitwise AND operation.

```python
if m & mask == 0:
```

   - If there's no overlap, it means that adding the current string's unique characters to the characters represented by `m` would still result in a string with all unique characters.

   - In that case, we combine `m` and `mask` using the bitwise OR operation. The result is a new mask representing a unique combination of characters from both masks. We add this new mask to `masks`.

```python
masks.append(m | mask)
```

   - We then calculate the total number of unique characters we have so far with the new mask by counting the number of set bits. In Python, this can be done with the `.bit_count()` method. We update our answer `ans` if this count is greater than the previous maximum.

```python
ans = max(ans, (m | mask).bit_count())
```

5. Once we've checked all strings and recorded all possible unique character combinations, we return the value of `ans`, which represents the maximum length of a string with all unique characters we can create by concatenating a subsequence of `arr`.

The use of bit masks elegantly handles the uniqueness constraint, and the iterative approach ensures that all potential combinations are considered without duplication, leading to an efficient solution.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider the input array of strings `arr` as `["un","iq","ue"]`.

1. Start by initializing `ans` to `0` and `masks` to `[0]`.

2. The first string is `un`.

   - For `u`, `ord('u') - ord('a')` gives us `20`. So, `mask` becomes `0 | (1 << 20)`.
   - For `n`, `ord('n') - ord('a')` gives us `13`. So, `mask` becomes `mask | (1 << 13)`.

   After processing `un`, our `mask` is `100001000000000000000` in binary, which is `1048576` decimal.

3. There are no duplicates in `un`, so we move on and create a new combination by OR-ing this `mask` with `0` from `masks`. We update our `masks` to be `[0, 1048576]`.

4. The next string is `iq`.

   - For `i`, `ord('i') - ord('a')` gives us `8`. So, `mask` becomes `0 | (1 << 8)`.
   - For `q`, `ord('q') - ord('a')` gives us `16`. So, `mask` becomes `mask | (1 << 16)`.

   After processing `iq`, our `mask` is `1000000010000` in binary, which is `65792` decimal.

5. Check this `mask` against all in `masks`. There's no common bit set with `1048576` (previous `mask`). Therefore, we can combine them to form a new mask: `1048576 | 65792` which in binary is `1000010010000`, and in decimal is `1111368`.

6. Update `ans` with the bit count of the new mask. It has 4 bits set, representing 4 unique characters. `ans` becomes `4`.

7. The masks array becomes `[0, 1048576, 65792, 1111368]`.

8. Lastly, we have the string `ue`.

   - For `u`, there's already a bit set in the previous mask, so we skip the creation of a new mask.

9. There are no more strings to process. We return the `ans`, which is `4`. This is the maximum length of a string with all unique characters we can create by concatenating strings from `arr`.

This process of using bit masks allows us to efficiently manage and combine the unique characters from various strings without having to deal with actual string concatenation and character counting.

Python Solution

```python
from typing import List

class Solution:
    def maxLength(self, arr: List[str]) -> int:
        max_length = 0  # Variable to store the maximum length of unique characters
        masks = [0]  # List to store the unique character sets as bit masks

        # Iterate through each string in the input list
        for string in arr:
            mask = 0  # Initialize mask for current string

            # Check each character in the string
            for char in string:
                char_index = ord(char) - ord('a')  # Map 'a'-'z' to 0-25

                # If the character is already in the mask, reset mask and break
                if mask >> char_index & 1:
                    mask = 0
                    break
                # Add the character to the mask
                mask |= 1 << char_index

            # If mask is not zero, it means the string had unique characters
            if mask != 0:
                # Check the new mask with existing masks for no overlap of characters
                for existing_mask in masks.copy():
                    if existing_mask & mask == 0:
                        new_mask = existing_mask | mask  # Combine the masks
                        masks.append(new_mask)  # Append the new mask to masks
                        max_length = max(max_length, bin(new_mask).count('1'))  # Update max length

        return max_length  # return the maximum length found

```

Java Solution

```java
class Solution {

    public int maxLength(List<String> arr) {
        int maxLen = 0; // This will hold the maximum length of unique characters.
        List<Integer> bitMasks = new ArrayList<>(); // This list will store unique character combinations using bit masking.
        bitMasks.add(0); // Initialize the list with a bitmask of 0 (no characters).

        // Iterate over each string in the list.
        for (String s : arr) {
            int bitMask = 0;
            // Iterate over the characters in the string to create a bitmask.
            for (int i = 0; i < s.length(); ++i) {
                int bitIndex = s.charAt(i) - 'a'; // Convert character to a bitmask index (0-25).
                // If the character is already in the bitmask (duplicate character),
                // set the bitmask to 0 and break out of the loop.
                if (((bitMask >> bitIndex) & 1) == 1) {
                    bitMask = 0;
                    break;
                }
                bitMask |= 1 << bitIndex; // Add the character into the bitmask.
            }

            // If bitmask is 0, the string contains duplicates and should be ignored.
            if (bitMask == 0) {
                continue;
            }

            int currentSize = bitMasks.size();
            // Iterate over existing masks and combine them with the new mask if possible.
            for (int i = 0; i < currentSize; ++i) {
                int combinedMask = bitMasks.get(i);
                // If there is no collision between the current mask and the new mask,
                // we can combine them into a new mask.
                if ((combinedMask & bitMask) == 0) {
                    bitMasks.add(combinedMask | bitMask); // Combine masks by OR operation.
                    maxLen = Math.max(maxLen, Integer.bitCount(combinedMask | bitMask)); // Update maxLen if necessary.
                }
            }
        }

        return maxLen; // Return the maximum length found.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    int maxLength(std::vector<std::string>& arr) {
        int max_length = 0; // to store the max length of unique characters
        // masks initially contains only one element: "0", which represents an empty string
        std::vector<int> masks = {0};

        // Iterate over all strings in the input vector
        for (std::string& str : arr) {

            int mask = 0; // Bitmask to represent the current string

            // Iterate over each character in the current string
            for (char& ch : str) {
                int bitIndex = ch - 'a';

                // Check if this character has already appeared in the string (mask)
                if ((mask >> bitIndex) & 1) {
                    // If the character repeats, discard this string by setting the mask to 0 and break
                    mask = 0;
                    break;
                }

                // Add the current character to the mask
                mask |= 1 << bitIndex;
            }

            // Only proceed if mask is not zero (valid string without any repeating character)
            if (mask == 0) continue;

            int currentMasksCount = masks.size();
            // Iterate over existing combinations of strings represented by masks
            for (int i = 0; i < currentMasksCount; ++i) {
                int combinedMask = masks[i];

                // Check if current mask and combined mask have no characters in common
                if ((combinedMask & mask) == 0) {
                    // Combine current string with the string represented by combinedMask
                    masks.push_back(combinedMask | mask);
                    // Update max_length using the number of unique characters in the new combination
                    max_length = std::max(max_length, __builtin_popcount(combinedMask | mask));
                }
            }
        }

        // Return the maximum length of a string with all unique characters
        return max_length;
    }
};

```

Typescript Solution

```typescript
// Import necessary functions from built-in modules
import { max } from 'lodash';

// Utility function to count the number of set bits (1s) in the binary representation of a number
const popCount = (n: number): number => {
  let count = 0;
  while (n) {
    count += n & 1;
    n >>= 1;
  }
  return count;
};

// Function to compute the max length of a concatenated string of unique characters
const maxLength = (arr: string[]): number => {
  let maxLength = 0; // To store the max length of unique characters
  // Masks initially contains only one element: 0, which represents an empty string
  let masks: number[] = [0];

  // Iterate over all strings in the input array
  arr.forEach(str => {
    let mask = 0; // Bitmask to represent the current string

    // Iterate over each character in the current string
    for (const ch of str) {
      let bitIndex = ch.charCodeAt(0) - 'a'.charCodeAt(0);

      // Check if this character has already appeared in the string (mask)
      if ((mask >> bitIndex) & 1) {
        // If the character repeats, discard this string by setting the mask to 0 and break
        mask = 0;
        break;
      }

      // Add the current character to the mask
      mask |= 1 << bitIndex;
    }

    // Only proceed if mask is not zero (valid string without any repeating character)
    if (mask === 0) return;

    // Create a copy of the current masks to iterate over
    const currentMasks = [...masks];

    // Iterate over existing combinations of strings represented by masks
    currentMasks.forEach(combinedMask => {
      // Check if the current mask and combined mask have no characters in common
      if ((combinedMask & mask) === 0) {
        // Combine current string with the string represented by combinedMask
        masks.push(combinedMask | mask);
        // Update maxLength using the number of unique characters in the new combination
        maxLength = max([maxLength, popCount(combinedMask | mask)])!;
      }
    });
  });

  // Return the maximum length of a string with all unique characters
  return maxLength;
};

// Example usage:
// console.log(maxLength(["un", "iq", "ue"])); // Should print 4

```

Time and Space Complexity

Time Complexity

The time complexity of the code primarily stems from two nested loops: the outer loop iterating over each string `s` in `arr`, and the inner loop iterating over the set of bitmasks `masks`. For every character `c` in each string `s`, we perform a constant-time operation to check if that character has already been seen by using a bitmask. If it has, we break early and the mask is set to `0`. The worst-case scenario for this operation is `O(k)` where `k` is the length of the longest string.

Considering the generation of new masks, for every `mask` variable generated from string `s`, the code iterates through the `masks` list to check if there's any overlap with existing masks (`m & mask == 0`). In the worst-case scenario, if all characters in `arr` are unique and `n` is the length of `arr`, there could be up to `2^n` combinations as every element in `arr` could be included or excluded from the combination.

The bit count operation (`.bit_count()`) is generally considered a constant-time operation, but since it's applied for each newly created mask, it doesn't dominate the time complexity.

Therefore, the overall time complexity is `O(n * 2^n * k)`.

Space Complexity

The space complexity is dictated by the `masks` list which stores the unique combinations of characters that have been seen. In the worst-case scenario, this could be as large as `2^n` where `n` is the length of `arr`. No other data structure in the code uses space that scales with the size of the input to the same degree.

Therefore, the space complexity of the algorithm is `O(2^n)`.