1297. Maximum Number of Occurrences of a Substring

Medium Hash Table String Sliding Window

Problem Description

The problem is to find the substring that appears the most times in a given string `s`. The catch is that these substrings have to comply with two rules:

1. The quantity of unique characters in each substring must be less than or equal to `maxLetters`.
2. The length of the substring must be at least `minSize` and at most `maxSize`.

A key thing to note is that though substrings can have lengths anywhere from `minSize` to `maxSize`, we are interested in finding the maximum number of occurrences, and typically substrings with smaller lengths will have higher chances of repeated occurrence. The solution should return the maximum frequency of any such substring found in `s` that follows the above rules.

Intuition

When dissecting the problem, we focus on the constraints - unique characters within the substring and the substring's size. The fact that we're also looking for maximum occurrence leads to a strategy where smaller substrings are more likely to be repeated; hence, we prioritize evaluating substrings of `minSize`.

The intuition for the solution is to slide a window of size `minSize` across the string `s` and use a set to track the unique characters and a counter (hashmap) to count occurrences of each substring that fits the rules. For each window, we check the following:

1. If the set size is larger than `maxLetters`, we ignore this substring as it doesn't meet the first rule.
2. If the set size is within the limits, we add the substring to the counter and check if it's the most frequent one we've seen so far.

This way we iterate over the string once (`O(n)` time complexity, where `n` is the length of the string), and only consider substrings of length `minSize` because regardless of `maxSize`, any repeated instances of a longer substring will always contain a repeated `minSize` substring within it. Thus, `minSize` is the key to finding the maximum occurrence.

Solution Approach

The solution uses a sliding window algorithm and two data structures—a set and a counter from Python's `collections` module. The set is used to keep track of the unique characters within the current `minSize` window of the string, while the `Counter` is a hashmap to keep track of the frequency of each unique substring that fits the criteria.

Here are the steps in the implementation:

1. Initialize `cnt` as a `Counter` object to keep track of the frequency of each qualifying substring.
2. Initialize `ans` as an integer to keep track of the maximum frequency found.
3. Iterate over the string with a variable `i` ranging from `0` to `len(s) - minSize + 1`. This is your sliding window that will only consider substrings of size `minSize`.
4. At each iteration, create a substring `t` which is a slice of `s` starting from index `i` to `i + minSize`.
5. For the substring `t`, create a set `ss` to determine the number of unique characters it contains.
6. A crucial optimization here is that if `len(ss)` is greater than the `maxLetters`, the substring does not satisfy the required constraints, and no further action is taken for that window.
7. If `len(ss)` is less than or equal to `maxLetters`, add/subtract one to the `cnt[t]` for that substring sequence, which increases the count of how many times `t` has been seen.
8. Update `ans` which keeps track of the highest frequency seen so far by comparing it with the frequency count of `t` in `cnt`.
9. After the loop finishes, `ans` will hold the highest frequency of occurrence of any valid substring and is returned.

The algorithm sweeps through the string only once, making it efficient with time complexity of `O(n)` (for the loop) times `O(k)` (for the set creation, where `k` is at most `minSize`), and it only considers substrings of length `minSize` due to the provided insight that longer substrings will naturally contain these smaller, frequently occurring substrings if they are to be frequent themselves. The space complexity mainly depends on the number of unique substrings of length `minSize` that can be formed from `s`, which in worst cases is `O(n)`.

Overall, the algorithm efficiently finds the most frequent substring of size within the given bounds that also contains no more than `maxLetters` unique characters.

Example Walkthrough

Let's consider a simple example with the following parameters:

- `s`: "ababab"
- `maxLetters`: 2
- `minSize`: 2
- `maxSize`: 3

Following the solution approach:

1. We initialize `cnt` as an empty `Counter` object and `ans` as 0.
2. We then iterate over `s` using a window size of `minSize`. In this example, `minSize` is 2, so we will be sliding through the string two characters at a time.
3. In the first iteration, `i` is 0, and our substring `t` is "ab" (from index 0 to 1). We create a set `ss` containing unique characters in "ab", which are `{'a', 'b'}`.
4. Since the size of `ss` is 2 and does not exceed `maxLetters`, which is also 2, we proceed to increment the count of this substring in `cnt`: `cnt["ab"] += 1`.
5. Now we slide the window by one and repeat. Our next substring is "ba" (from index 1 to 2), and `ss` will again be `{'b', 'a'}`. We increment `cnt["ba"]`.
6. This process is repeated for the entire string: we then check "ab" again, and "ba" again. Each time, we are incrementing the count in `cnt` for the respective substring.
7. After we've processed each substring, we observe that "ab" and "ba" each have a count of 3 in `cnt`.
8. `ans` is then updated to be the maximum of the values in `cnt`, which in this case is 3.

The final answer, held by `ans`, is 3, indicating that the most frequent substrings of length `minSize` are "ab" and "ba," each appearing 3 times in the string `s`. Our example is now complete, demonstrating how the sliding window and counter work together to find the most frequent qualifying substring.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def maxFreq(self, s: str, maxLetters: int, minSize: int, maxSize: int) -> int:
        # This function finds the maximum frequency of any substring that meets the criteria

        # Initialize the variable to store the maximum frequency found
        max_frequency = 0

        # Create a Counter object to keep track of the frequencies of the substrings
        substring_counter = Counter()

        # Loop through the string to check all possible substrings of length minSize
        for i in range(len(s) - minSize + 1):

            # Extract a substring of length minSize
            substring = s[i: i + minSize]

            # Create a set of unique characters in the substring
            unique_chars = set(substring)

            # If the number of unique characters is less than or equal to maxLetters
            if len(unique_chars) <= maxLetters:
                # Increment the frequency count for this substring
                substring_counter[substring] += 1

                # Update max_frequency with the maximum value between the current max
                # and the frequency of the current substring
                max_frequency = max(max_frequency, substring_counter[substring])

        # Return the maximum frequency
        return max_frequency
```

Java

```java
class Solution {
    public int maxFreq(String s, int maxLetters, int minSize, int maxSize) {
        // Initialize the answer variable to store the maximum frequency.
        int maxFrequency = 0;
        // Create a HashMap to store the frequency of substrings.
        Map<String, Integer> substringFrequency = new HashMap<>();
        // Loop through the string to find valid substrings of length minSize.
        for (int start = 0; start <= s.length() - minSize; ++start) {
            // Extract substring of size `minSize` from the original string.
            String substring = s.substring(start, start + minSize);
            // HashSet to track unique characters in the current substring.
            Set<Character> uniqueChars = new HashSet<>();
            // Calculate the number of unique characters in the substring.
            for (int j = 0; j < minSize; ++j) {
                // Add unique characters in the substring to the set.
                uniqueChars.add(substring.charAt(j));
            }
            // Check if the number of unique characters meets the requirement.
            if (uniqueChars.size() <= maxLetters) {
                // Increment the count of this valid substring in the map.
                substringFrequency.put(substring, substringFrequency.getOrDefault(substring, 0) + 1);
                // Update the maximum frequency with the highest count found so far.
                maxFrequency = Math.max(maxFrequency, substringFrequency.get(substring));
            }
        }
        // Return the maximum frequency of any valid substring.
        return maxFrequency;
    }
}
```

C++

```cpp
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>

class Solution {
public:
    int maxFreq(string s, int maxLetters, int minSize, int maxSize) {
        // Initialize the answer to be returned
        int maxFrequency = 0;

        // Make use of an unordered_map to count the occurrences of substrings
        unordered_map<string, int> substringCounts;

        // Loop through the string, only up to the point where minSize substrings can still be formed
        for (int i = 0; i <= s.size() - minSize; ++i) {

            // Extract the substring of length minSize starting at index i
            string t = s.substr(i, minSize);

            // Create a set of unique characters (ss) from the substring
            unordered_set<char> uniqueChars(t.begin(), t.end());

            // Check if the number of unique characters does not exceed maxLetters
            if (uniqueChars.size() <= maxLetters) {

                // If the conditions are met, increment the count for this substring
                // and update the maxFrequency with the maximum value
                maxFrequency = std::max(maxFrequency, ++substringCounts[t]);
            }
        }

        // Return the maximum frequency found for any substring that meets the criteria
        return maxFrequency;
    }
};
```

TypeScript

```typescript
// Import necessary elements from 'collections' for Map and Set
import { Map, Set } from 'collections';

// Function to calculate the maximum frequency of any substring meeting certain criteria
function maxFreq(s: string, maxLetters: number, minSize: number, maxSize: number): number {
    // Initialize the answer to be returned
    let maxFrequency: number = 0;

    // Make use of a Map to count the occurrences of substrings
    let substringCounts: Map<string, number> = new Map<string, number>();

    // Loop through the string, only up to the point where minSize substrings can still be formed
    for (let i = 0; i <= s.length - minSize; ++i) {

        // Extract the substring of length minSize starting at index i
        let t: string = s.substring(i, i + minSize);

        // Create a set of unique characters from the substring
        let uniqueChars: Set<string> = new Set<string>(t.split(''));

        // Check if the number of unique characters does not exceed maxLetters
        if (uniqueChars.size <= maxLetters) {

            // If the conditions are met, increment the count for this substring
            let count: number = substringCounts.get(t) || 0;
            substringCounts.set(t, count + 1);

            // Update the maxFrequency with the maximum value
            maxFrequency = Math.max(maxFrequency, count + 1);
        }
    }

    // Return the maximum frequency found for any substring that meets the criteria
    return maxFrequency;
}
```


```python
from collections import Counter

class Solution:
    def maxFreq(self, s: str, maxLetters: int, minSize: int, maxSize: int) -> int:
        # This function finds the maximum frequency of any substring that meets the criteria

        # Initialize the variable to store the maximum frequency found
        max_frequency = 0

        # Create a Counter object to keep track of the frequencies of the substrings
        substring_counter = Counter()

        # Loop through the string to check all possible substrings of length minSize
        for i in range(len(s) - minSize + 1):

            # Extract a substring of length minSize
            substring = s[i: i + minSize]

            # Create a set of unique characters in the substring
            unique_chars = set(substring)

            # If the number of unique characters is less than or equal to maxLetters
            if len(unique_chars) <= maxLetters:
                # Increment the frequency count for this substring
                substring_counter[substring] += 1

                # Update max_frequency with the maximum value between the current max
                # and the frequency of the current substring
                max_frequency = max(max_frequency, substring_counter[substring])

        # Return the maximum frequency
        return max_frequency
```


Time and Space Complexity

The time complexity of the code is `O(n * minSize)` where `n` is the length of the string `s`. This is because the code iterates over the string in slices of length `minSize` which takes `O(minSize)` time for each slice, and this is done for all starting points in the string, of which there are `n - minSize + 1`.

The space complexity of the code is `O(n)` in the worst case. This is due to the fact that the `Counter` could potentially store every unique substring of length `minSize` in the worst scenario where all possible substrings are unique. Since the length of each substring is bounded by `minSize`, this does not affect the order of space complexity.