187. Repeated DNA Sequences

Medium | Bit Manipulation | Hash Table | String | Sliding Window | Hash Function | Rolling Hash | Leetcode Link

Problem Description

In the given LeetCode problem, we are tasked with analyzing a string that represents a DNA sequence. A DNA sequence consists of a series of nucleotides, which are denoted by the characters 'A', 'C', 'G', and 'T'. Our goal is to find all the unique substrings of length 10 that repeat more than once within this DNA sequence. We need to return these substrings in any order, without accounting for their frequency beyond the first occurrence.

Intuition

Encountering this problem, the first intuition is to track the frequency of every possible 10-letter-long substring within the larger DNA string. To efficiently look up these sequences, a **HashTable** (Python's `Counter` dictionary can be used here for convenience) is a natural choice because it allows insertions and lookups in constant time, assuming a good hash distribution.

The thinking process goes something like this:

1. We want to iterate over the sequence only once and keep track of every 10-letter sequence as we go. This is a sliding window problem.
2. Each time we encounter a new 10-letter sequence, we add it to the HashTable. If the sequence already exists in the table, we increment its count.
3. Instead of waiting until the end to find out which sequences have occurred more than once, we can check the count as we go. If a sequence's count reaches 2, we add it to the answer list. We only need to do this once per sequence because the problem asks for sequences that occur 'more than once,' not the specific number of times they occur.
4. When a sequence's count exceeds 2, we don't need to do anything more with it, so we can ignore it. This avoids adding duplicates to our results list and keeps the operation more efficient.
5. Finally, we return the list of repeating 10-letter sequences.

By using this approach, we optimize for both time and space by not keeping track of sequences that don't matter (those with counts less than 2 after processing) and by not duplicating entries in our results list.

Solution Approach

The implementation leverages a HashTable to track 10-letter sequences, which provides an efficient way to check for duplicate sequences as we iterate through the DNA string. Here's how the solution unfolds:

1. Define a variable `n` which represents the range we need to check within the string. Since we're looking for sequences of length 10, we set `n` to `len(s) - 10`. This ensures we don't go over the string length when checking for the substrings at the end of the DNA sequence.
2. Initialize a `Counter` (a special HashTable from Python's collections library) to keep track of the counts of all 10-letter sequences we find.
3. Define an empty list `ans`, which will store the sequences that repeat more than once.
4. Iterate over the DNA string using a range that goes from `0` to `n+1`. This range represents the starting index of each 10-letter sequence we want to examine.
5. Inside the loop, extract a substring `sub` of length 10 starting from the current index `i`.
6. Increment the count of this substring in the `Counter`. The `Counter` automatically handles the creation of new keys if the substring hasn't been seen before.
7. Check if the count of the current substring has reached 2. If it has, it means this sequence has been encountered once before and now is the second occurrence. It is now qualified as a repeating sequence and should be appended to the `ans` list.
8. Continue this process until we've checked all possible starting indices of 10-letter sequences in the DNA string.
9. Once the loop completes, return the `ans` list holding all the sequences that repeated more than once.

The application of this method ensures we only store and return unique sequences that fulfill the problem's conditions. The usage of the `Counter` allows for both efficient insertion and lookup operations, where the worst-case time complexity is `O(n * 10)` accounting for the substring operation in each iteration. The space complexity is `O(n)`, as we potentially store information about each 10-letter sequence present in the DNA string.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using a given DNA string: `s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"`.

1. We initialize `n` as `len(s) - 10`, which equates to `20` since the string length is `30`. This signifies that we will check for 10-letter sequences starting at indices `0` to `20`.
2. We then create a `Counter` to keep track of the sequences.
3. We initialize an empty list `ans` to store our answers.
4. We begin iterating from index `0` to `20` (inclusive) to check all possible 10-letter-long sequences.
   - At index `0`, we get the substring `sub = "AAAAACCCCC"`. We increment its count in the `Counter`. It's the first time we've seen it, so its count is now `1`.
   - Moving to index `1`, the substring is `sub = "AAAACCCCCA"`. We increment its count in the `Counter`.
   - This process continues with new substrings as the index increases.
   - At index `10`, we reach the substring `sub = "AAAAACCCCC"`, which was previously seen at index `0`. Its count in the `Counter` becomes `2`. Since the count is `2`, we add this sequence to the `ans` list.
   - At index `15`, the substring `sub = "CCCCCAAAAA"` appears. This is also the second time we've encountered it, so its count becomes `2`, and it is added to the `ans` list.
5. After iterating through all starting indices from `0` to `20`, we finish with `ans` containing the sequences that appeared more than once. In this case, `ans = ["AAAAACCCCC", "CCCCCAAAAA"]`.
6. Finally, we return the list `ans` as our answer, which contains the repeated sequences, satisfying the problem's conditions.

This example demonstrates the process of using the sliding window technique, combined with a `Counter`, to efficiently determine sequences that repeat more than once within a given DNA string. By checking and updating counts as we iterate, we're able to avoid unnecessary work and produce our answer without redundancies.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def findRepeatedDnaSequences(self, dna_sequence: str) -> List[str]:
        # Initialize a counter to keep track of the DNA sequence occurrences
        sequence_count = Counter()
        # Initialize an empty list to store the repeated sequences
        repeated_sequences = []
        # Calculate the number of possible substrings of length 10
        num_substrings = len(dna_sequence) - 10
        # Iterate over all the substrings of length 10 in the DNA sequence
        for i in range(num_substrings + 1):
            # Extract a substring of length 10
            subsequence = dna_sequence[i: i + 10]
            # Increment the count for this particular substring
            sequence_count[subsequence] += 1
            # If the count reaches 2, it means we've found a repeated sequence
            if sequence_count[subsequence] == 2:
                # Add it to the list of repeated sequences
                repeated_sequences.append(subsequence)
        # Return the list of repeated sequences
        return repeated_sequences

```

Java Solution

```java
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;

class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        // Calculate the number of 10-character substrings in 's'
        int substringLengthLimit = s.length() - 10;
        // Create a map to store counts of each DNA sequence
        Map<String, Integer> sequenceCounts = new HashMap<>();
        // List to store the DNA sequences that appear more than once
        List<String> repeatedSequences = new ArrayList<>();

        // Iterate through the string checking for 10-character long substrings
        for (int i = 0; i <= substringLengthLimit; ++i) {
            // Extract the 10-character substring starting at index 'i'
            String currentSubstring = s.substring(i, i + 10);
            // Increase the count of the current DNA sequence in the map
            sequenceCounts.put(currentSubstring, sequenceCounts.getOrDefault(currentSubstring, 0) + 1);
            // If this is the second occurrence of the DNA sequence, add it to the answer list
            if (sequenceCounts.get(currentSubstring) == 2) {
                repeatedSequences.add(currentSubstring);
            }
        }
        // Return the list of repeated DNA sequences
        return repeatedSequences;
    }
}

```

C++ Solution

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

class Solution {
public:
    std::vector<std::string> findRepeatedDnaSequences(std::string s) {
        // Use an unordered_map for efficient lookup and insertion
        std::unordered_map<std::string, int> sequenceCount;

        // Calculate the number of possible starting indices for 10-letter sequences
        int possibleStarts = s.size() - 9;

        // Prepare a vector to store the resulting repeated sequences
        std::vector<std::string> repeats;

        // Loop through the string to examine all possible 10-letter sequences
        for (int i = 0; i < possibleStarts; ++i) {
            // Extract a 10-letter substring starting at index i
            std::string subsequence = s.substr(i, 10);

            // Increment the count for this particular sequence
            // If the count becomes 2, it implies this is the second time we've seen it,
            // so we add it to the list of repeats
            if (++sequenceCount[subsequence] == 2) {
                repeats.push_back(subsequence);
            }
        }

        // Return the list of repeated sequences
        return repeats;
    }
};

```

Typescript Solution

```typescript
function findRepeatedDnaSequences(dnaSequence: string): string[] {
    // Define the length of the sequence.
    const sequenceLength = dnaSequence.length;
    // Create a map to keep track of all 10-letter sequences.
    const sequenceCountMap = new Map<string, number>();
    // Initialize the array to store the result.
    const repeatedSequences = [];
    // Iterate over the DNA sequence to the point where a 10-letter sequence can be extracted.
    for (let i = 0; i <= sequenceLength - 10; i++) {
        // Extract the current 10-letter sequence.
        const currentSequence = dnaSequence.slice(i, i + 10);
        // Check if the current sequence is already in the map.
        if (sequenceCountMap.has(currentSequence)) {
            // If it's in the map and the value is true, add it to the results.
            // The value is set to true when the sequence is seen for the second time.
            if (sequenceCountMap.get(currentSequence) === 1) {
                repeatedSequences.push(currentSequence);
                // Increment the count to indicate that the sequence has been added to the results.
                // This prevents adding the same sequence more than once.
                sequenceCountMap.set(currentSequence, 2);
            }
        } else {
            // If it's a new sequence, add it to the map with a count of 1,
            // indicating that it has been seen for the first time.
            sequenceCountMap.set(currentSequence, 1);
        }
    }
    // Return the array containing all repeated 10-letter sequences found.
    return repeatedSequences;
}

```

Time and Space Complexity

The provided code snippet finds all the 10-letter-long sequences in the string `s` that appear more than once. To analyze the time and space complexity of this Python code, we will review each significant portion of the code.

Time Complexity:

The code loops through the string `s`, starting from index `0` to index `n` (which is `len(s) - 10`). In each iteration, it creates a substring of length `10` and increases the corresponding counter for this substring. Since the loop runs for `(len(s) - 10) + 1` times and each operation within the loop (creating a substring and updating the counter) can be considered as O(1), the overall time complexity is `O(n - 10 + 1)` which simplifies to `O(n)` where `n` is the length of the input string `s`.

Space Complexity:

The space complexity is determined by the additional space required by the algorithm which is primarily dependent on the `cnt` and `ans` variables.

- The `Counter` object will at most store each unique 10-letter-long sequence present in `s`. In the worst case, this will be `n - 9` unique sequences if every sequence is unique. This gives O(n) as the space complexity contribution from `cnt`.
- The `ans` list stores each 10-letter sequence that appears exactly twice. In the worst case, if every possible 10-letter sequence appeared exactly twice, this would yield `O(n/10)` space complexity which simplifies to `O(n)`.

By combining the above considerations, the overall space complexity is `O(n)`.