1062. Longest Repeating Substring

Medium String Binary Search Dynamic Programming Suffix Array Hash Function Rolling Hash

Problem Description

The problem asks for the length of the longest repeating substring within a given string `s`. A repeating substring is defined as a sequence of characters that appears more than once in the original string. To clarify, if the string was "abab", the longest repeating substring would be "ab", which has a length of 2. If no such repeating substring exists, such as in a string with all unique characters like "abc", the function should return 0.

Intuition

The solution to this problem involves dynamic programming, a method for solving complex problems by breaking them down into simpler subproblems. The intuition behind using dynamic programming for this problem is the idea of comparing characters at different positions and building a table that records the lengths of repeating substrings ended at those positions.

This is accomplished by initializing a square matrix `dp` with the dimensions of the string's length, where `dp[i][j]` will be used to store the length of the longest repeating substring that ends with the characters `s[i]` and `s[j]`. We can then iterate over each position `i` in the string, and for each `i`, iterate again through each position `j` from `i+1` to the end of the string. If the characters at positions `i` and `j` are the same, it means there's a repeating character, and we can build upon the length of previously found repeating substrings.

Specifically, if `s[i]` equals `s[j]`, then `dp[i][j]` is the length of the repeating substring that ended just before `i` and `j` (which is `dp[i - 1][j - 1]`), plus 1 for the current matching characters. If `i` is 0, there's no previous character, so `dp[i][j]` is just set to 1. Along the way, we keep track of the maximum length found in variable `ans`.

Every time we find a matching pair of characters, we update `ans` to hold the maximum length of any repeating substring found so far. Through this process, we arrive at the longest repeating substring's length without having to check each possible substring individually.

Solution Approach

The implementation of the solution uses dynamic programming, explicitly utilizing a 2D array (referred to as a matrix) for storing the lengths of repeating substrings. Here's how the code works:

1. Initialize a 2D list (matrix) `dp`, where each element `dp[i][j]` represents the length of the longest repeating substring that ends with `s[i]` and `s[j]`. Set the entire matrix to 0 initially because we haven't found any repeating substrings yet.
2. Set a variable `ans` to 0. This variable will keep track of the length of the longest repeating substring found so far.
3. Iterate over each character in the string using two nested loops:
   a. The outer loop goes through each character in the string with index `i`.
   b. The inner loop starts from `i+1` and goes to the end of the string with index `j`.
   c. For each pair of indices `i` and `j`, check if `s[i]` is equal to `s[j]`. If they are:
   i. If `i` is greater than 0, update `dp[i][j]` to be `dp[i - 1][j - 1] + 1`. This is because a matching pair of c
   ii. If `i` is equal to 0, set `dp[i][j]` to 1 since there are no previous characters to consider, and thus we ha
   d. Update the `ans` variable with the larger of its current value or `dp[i][j]`, ensuring `ans` always holds the length of the longest repeating substring found.
4. After completing the iterations, return the value stored in `ans`, which now represents the length of the longest repeating substring in `s`.

The algorithm uses a dynamic programming matrix to avoid redundant work, building up the solution based on previously computed values. By checking only pairs of indices where `j` is greater than `i`, the algorithm ensures that only substrings are considered, not individual characters or the entire string. This approach efficiently solves the problem with `O(n^2)` time complexity and `O(n^2)` space complexity, where `n` is the length of the string `s`.

Example Walkthrough

Let's take a small example string `s = "aabba"`. According to the given solution approach, we want to find the length of the longest repeating substring in `s`. Here is how the approach works step-by-step:

1. We initialize a matrix `dp` of 5×5 dimensions (since the string `s` has a length of 5) with all values set to 0. The matrix looks like this:

```
dp = [
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0]
]
```

2. We set `ans` to 0.
3. We start iterating over each character in `s` using two nested loops:
   - Outer loop with index `i` from 0 to 4.
   - Inner loop with index `j` from `i+1` to 4.
4. Now we work through the loops:
   - On the first iteration (`i=0`, `j=1`), we find that `s[0]` (which is 'a') is equal to `s[1]` (also 'a'). Since `i` is 0, we set `dp[0][1]` to 1. `ans` is updated to 1.

   ```
   dp = [
     [0, 1, 0, 0, 0],
     [0, 0, 0, 0, 0],
     ...
   ]
   ```

   - On iteration (`i=1`, `j=2`), `s[1]` is 'a' but `s[2]` is 'b', so we do nothing as they don't match.
   - On iteration (`i=3`, `j=4`), we find a match as both `s[3]` and `s[4]` are 'a'. Since `i > 0`, we set `dp[3][4]` to `dp[2][3] + 1` which is 1 (since 'b'[3] and 'b'[4] don't match, `dp[2][3]` is 0). Now, `ans` is updated to 1.

   ```
   dp = [
     ...
     [0, 0, 0, 0, 1]
   ]
   ```

5. None of the remaining iterations of the loops yield any greater repeating substring, so `ans` remains 1.
6. After iterating through the string and updating the `dp` matrix, we find that the value of `ans` remains 1.

Thus, the length of the longest repeating substring in the example string "aabba" is 1, and that repeating substring is "a".

Solution Implementation

Python

```python
class Solution:
    def longestRepeatingSubstring(self, s: str) -> int:
        # Length of the input string
        n = len(s)

        # Initialize a 2D array(dynamically programming table)
        # to store lengths of repeating substrings
        dp_table = [[0] * n for _ in range(n)]

        # Variable to store the length of the largest repeating substring
        max_length = 0

        # Iterate through each character in the string
        for i in range(n):
            # Compare with other characters in the string to the right
            for j in range(i + 1, n):
                # Check if the characters s[i] and s[j] match
                if s[i] == s[j]:
                    # Extend the length of the repeating substring
                    # If i is not 0, get the previous length from the dp table, else just start with length 1.
                    dp_table[i][j] = dp_table[i - 1][j - 1] + 1 if i else 1
                    # Update the max_length if the current repeating substring is longer
                    max_length = max(max_length, dp_table[i][j])

        # Return the length of the longest repeating substring
        return max_length
```

Java

```java
class Solution {
    public int longestRepeatingSubstring(String s) {
        int length = s.length(); // Length of the string 's'
        int longestLength = 0; // Variable to store the length of the longest repeating substring
        int[][] dp = new int[length][length]; // dp[i][j] will hold the length of the longest repeating substring ending at i and

        // Iterate through the string 's'
        for (int i = 0; i < length; ++i) {
            for (int j = i + 1; j < length; ++j) {
                // Check if characters at positions i and j are the same.
                if (s.charAt(i) == s.charAt(j)) {
                    // If they match and are not at the first character,
                    // increment the length of the repeating substring by 1.
                    // Otherwise, set the repeating substring length to 1 as it's the start of a possible repeating pattern.
                    dp[i][j] = i > 0 ? dp[i - 1][j - 1] + 1 : 1;
                    // Update the longestLength if the current repeating substring is the longest found so far.
                    longestLength = Math.max(longestLength, dp[i][j]);
                }
                // If they don't match, the default value of dp[i][j] remains 0, indicating no repetition.
            }
        }
        return longestLength; // Return the length of the longest repeating substring.
    }
}
```

C++

```cpp
class Solution {
public:
    int longestRepeatingSubstring(string s) {
        int length = s.size(); // Get the size of the string
        // Create a 2D DP table with 'length' rows and 'length' columns initialized to 0
        vector<vector<int>> dp(length, vector<int>(length, 0));
        int longest = 0; // Variable to store the length of the longest repeating substring

        // Iterate over the string with two pointers
        for (int i = 0; i < length; ++i) {
            for (int j = i + 1; j < length; ++j) {
                // If characters at the current position in both pointers match
                if (s[i] == s[j]) {
                    // Check if it is not the first character, then add 1 to the length of substring
                    // found till the previous characters else start with 1
                    dp[i][j] = (i > 0) ? dp[i - 1][j - 1] + 1 : 1;
                    // Update the longest substring found so far
                    longest = max(longest, dp[i][j]);
                }
                // If characters don't match, dp[i][j] remains 0 as initialized
            }
        }
        // Return the length of longest repeating substring
        return longest;
    }
};
```

TypeScript

```typescript
function longestRepeatingSubstring(s: string): number {
    const length = s.length; // Get the length of the string
    // Create a 2D DP array with 'length' rows and 'length' columns initialized to 0
    const dp: number[][] = Array.from({ length }, () => Array(length).fill(0));
    let longest = 0; // Variable to store the length of the longest repeating substring

    // Iterate over the string with two pointers
    for (let i = 0; i < length; ++i) {
        for (let j = i + 1; j < length; ++j) {
            // If characters at the current position in both pointers match
            if (s[i] === s[j]) {
                // Check if it is not the first character, then add 1 to the length of the substring
                // found till the previous characters; otherwise, start with 1
                dp[i][j] = (i > 0) ? dp[i - 1][j - 1] + 1 : 1;
                // Update the longest substring found so far
                longest = Math.max(longest, dp[i][j]);
            }
            // If characters don't match, dp[i][j] remains 0 as initialized
        }
    }
    // Return the length of the longest repeating substring
    return longest;
}
```

```python
class Solution:
    def longestRepeatingSubstring(self, s: str) -> int:
        # Length of the input string
        n = len(s)

        # Initialize a 2D array(dynamically programming table)
        # to store lengths of repeating substrings
        dp_table = [[0] * n for _ in range(n)]

        # Variable to store the length of the largest repeating substring
        max_length = 0

        # Iterate through each character in the string
        for i in range(n):
            # Compare with other characters in the string to the right
            for j in range(i + 1, n):
                # Check if the characters s[i] and s[j] match
                if s[i] == s[j]:
                    # Extend the length of the repeating substring
                    # If i is not 0, get the previous length from the dp table, else just start with length 1.
                    dp_table[i][j] = dp_table[i - 1][j - 1] + 1 if i else 1
                    # Update the max_length if the current repeating substring is longer
                    max_length = max(max_length, dp_table[i][j])

        # Return the length of the longest repeating substring
        return max_length
```

Time and Space Complexity

Time Complexity

The provided code uses a double for-loop to iterate over all possible starting positions of substrings in the string `s`, which has the length `n`. Inside these loops, it checks if two characters in `s` are equal and if so, updates the dynamic programming (DP) table.

The outer loop runs `n` times, and the inner loop can run up to `n` times as well, which could suggest an `O(n^2)` complexity. However, since the inner loop starts at `i + 1`, the number of iterations in the inner loop decreases as `i` increases. The total number of iterations could then be roughly calculated as the sum of the first `n` natural numbers, minus the diagonal elements (which are not accessed), which is `(n * (n - 1)) / 2`, which is still in the order of `O(n^2)`.

Therefore, the time complexity of the code is `O(n^2)`.

Space Complexity

The space complexity is dominated by the space used to store the dynamic programming (DP) table called `dp`, which is a 2D array of size `n x n`. Since the DP table is filled with 0 values at the beginning, and no other data structures scale with the size of `n` grow, the space complexity is `O(n^2)`.