471. Encode String with Shortest Length

Hard String Dynamic Programming Leetcode Link

Problem Description

The goal of this problem is to encode a given string `s` in such a way that the encoded version has the shortest possible length. The encoding rule must follow the format `k[encoded_string]`, where `encoded_string` represents a sequence of the string that is repeated `k` times, and `k` is a positive integer that represents the number of times `encoded_string` occurs consecutively within `s`.

For example, if the string is `"aaaabaaaab"`, it can be encoded as `2[a4[ab]]`, which means that `a4[ab]` (which represents `"aaaab"`) is repeated twice.

The string should only be encoded if doing so reduces its length. For strings that would not be shortened by encoding, the original string should be returned. In cases where multiple encoding methods result in the shortest length, any of those methods are considered a valid solution.

Intuition

The intuition behind the solution is to apply dynamic programming, which is a method often used in optimization problems. Dynamic programming involves breaking down a complex problem into simpler subproblems and solving each of those subproblems just once, storing their solutions—often in a two-dimensional array.

The main idea of the solution is to:

1. Iterate over all possible substrings of `s` and determine the shortest encoding for each substring.
2. For each substring, check if it has a repetitive pattern that can be encoded. Patterns are identified by checking if the substring `t` can be found within the concatenation of `t` with itself, but not at the very beginning (this implies a repetition).
3. If such a pattern exists and it helps reduce the length of the current substring, encode it using the format `k[encoded_substring]`. If not, keep the substring unencoded.
4. For longer substrings that do not have a repetitive pattern that can be encoded, or such encoding does not reduce the length, explore breaking the substring into two smaller encoded subproblems (substrings) whose total encoded length is minimal.

By solving the subproblems from shortest to longest substrings and building up solutions, we ensure that when we determine whether to encode a longer substring, we are making the decision based on the optimal (shortest) encodings of all possible subparts of that substring.

Essentially, the dynamic programming array `f[i][j]` holds the shortest encoded version of the substring starting from index `i` to index `j` in the original string `s`. By the end of the iterations, `f[0][n-1]` will give us the shortest encoded version of the entire string.

Solution Approach

The solution approach implements dynamic programming to solve the problem, where a two-dimensional array `f` is used to store the shortest encoded strings for all the substrates from `i` to `j`. The implementation goes as follows:

1. Function `g(i, j)` takes the start index `i` and the end index `j` as inputs and returns the optimal encoded string for the substring `s[i:j+1]`. If the substring is shorter than 5 characters, it's not worth encoding because the encoded format would not be shorter than the original substring.

2. In function `g(i, j)`, `t` is the current substring. In order to find repetitive patterns within `t`, `(t + t).index(t, 1)` is used. This expression checks whether `t` repeats in a concatenation of itself after the first character. If `k` is the index where `t` is found, it means `t` is repeating every `k` characters within `t`. If `k` is less than the length of `t`, we have found a repeating pattern, and we encode it as `"{cnt}[{f[i][i + k - 1]}]"`, where `cnt` is the count of repetition.

3. For every substring, we store our finding in the `f` array. The elements `f[i][j]` will represent the shortest encoded form of the substring `s[i:j+1]`.

4. A nested loop is used to fill the table `f` in a bottom-up manner. The outer loop decrements `i` from `n-1` to `0` to ensure we solve for all smaller substrates first. The inner loop increments `j` from `i` to `n-1` to cover all substrings starting from `i`.

5. For every pair of `i` and `j`, we first check whether it's worth encoding the current substring using function `g(i, j)`. If the substring length from `i` to `j` is more than 4 (since we don't want to encode shorter substrings), we then check for potential splits within this substring that could lead to an overall shorter length.

6. To check for splits, we iterate through all possible split positions `k` from `i` to `j` and calculate the combined length of encoded substrings `f[i][k]` and `f[k+1][j]`. If this combined length is shorter than the currently stored encoded string for `s[i:j+1]`, we update `f[i][j]` with this shorter version.

7. After filling the array `f`, the shortest encoded version of the entire string `s` will be stored in `f[0][n-1]`, which is the solution to the original problem.

The algorithm capitalizes on the property that to get the shortest encoding of a longer substring, you need to know the shortest encodings of all its subparts. This leads to an optimization problem where dynamic programming excels. By storing solutions of small problems and using them to solve larger problems, the algorithm works efficiently without recalculating the encoded strings for the same substrates multiple times.

Example Walkthrough

Let's take the string `"abbbabbb"` to illustrate the solution approach:

1. Initialize a table `f` where `f[i][j]` will eventually contain the shortest encoded version of the substring `s[i:j+1]`.

2. Consider substrings `s[i:j+1]` of length less than 5, which are not worth encoding. So `f[i][j]` would just be `s[i:j+1]`. For instance, `f[0][3]` for substring `"abbb"` remains `"abbb"` because its length is less than 5.

3. For substrings of length 5 or more, check if they have a repeatable pattern that can be encoded. For example, the substring `"abbbabbb"` has a repetitive substring which is `"abbb"` that repeats twice.

4. Use `(t + t).index(t, 1)` to find the repeating pattern within `t`. For our example, `t = "abbb"`, and by searching in `(t + t) = "abbbabbb"`, `t` starts repeating at index 4, which is equal to the length of `t`, indicating that it repeats every 4 characters.

5. Since the pattern `t` repeats `2` times, encode the substring as `"2[abbb]"` and store it in table `f` at `f[0][7]`, replacing `"abbbabbb"`.

6. If we were to consider a longer string and `f` is partially filled with shorter substrings' encodings, check for optimal splits by trying all possible split points. For every possible split at position `k` for the substring `s[i:j+1]`, calculate the total length of `f[i][k] + f[k+1][j]`. Choose the split that offers the shortest encoded length.

7. After checking for all possible splits and patterns, the final encoded strings are ready in `f`. For our example, since the substring `"abbbabbb"` has already been optimally encoded as `"2[abbb]"`, this would be the value of `f[0][7]`, and thus the output for the entire string.

By systematically applying these steps to increasingly longer substrates of the original string and using the dynamic programming array to store intermediate solutions, the algorithm ensures efficiency and avoids redundant calculations, leading to an optimal solution.

Python Solution

```python
class Solution:
    def encode(self, s: str) -> str:
        # Helper method to compute the encoded string for a substring s[i:j+1]
        def compute_encoded_substring(start: int, end: int) -> str:
            substring = s[start:end + 1]
            # If the substring is too short, encoding doesn't make sense, return as is
            if len(substring) < 5:
                return substring

            # Try to find a repeated pattern in the substring
            duplicate_index = (substring + substring).find(substring, 1)

            # If a repeated pattern is found that's shorter than the original string
            if duplicate_index < len(substring):
                count_of_repetition = len(substring) // duplicate_index
                # Encoded format: count[encoded substring for the repeated pattern]
                return f"{count_of_repetition}[{dynamic_table[start][start + duplicate_index - 1]}]"
            # No repeated pattern was found, return the original substring
            return substring

        # Main logic begins here
        length_of_s = len(s)
        # Initialize dynamic programming table with None
        dynamic_table = [[None] * length_of_s for _ in range(length_of_s)]

        # Build the table bottom-up
        for i in range(length_of_s - 1, -1, -1):
            for j in range(i, length_of_s):
                # Compute encoded substring for current i, j
                dynamic_table[i][j] = compute_encoded_substring(i, j)

                # If current substring can potentially be shortened
                if j - i + 1 > 4:
                    # Try to break the substring into two parts and see if this gives
                    # a shorter encoding by checking all possible split positions
                    for k in range(i, j):
                        # Combine encoded substrings for both parts
                        trial_encoded = dynamic_table[i][k] + dynamic_table[k + 1][j]
                        # If this combination gives us a shorter string, update the table
                        if len(dynamic_table[i][j]) > len(trial_encoded):
                            dynamic_table[i][j] = trial_encoded

        # Result is in the top-right cell of the dynamic programming table
        return dynamic_table[0][-1]

```

Java Solution

```java
class Solution {
    private String originalString;
    private String[][] encodedSubStrings;

    public String encode(String s) {
        originalString = s;
        int n = s.length();
        encodedSubStrings = new String[n][n];

        // Iterate over all possible substrings in reverse order
        for (int start = n - 1; start >= 0; --start) {
            for (int end = start; end < n; ++end) {
                // Attempt to find the encoded version of the current substring
                encodedSubStrings[start][end] = encodeSubstring(start, end);

                // Avoid unnecessary work for substrings shorter than 5 characters,
                // as encoding them would not be efficient.
                if (end - start + 1 > 4) {
                    for (int split = start; split < end; ++split) {
                        // Encode the current substring by combining the encoded versions
                        // of its two halves and check if this is shorter than the current encoding.
                        String combinedEncoding = encodedSubStrings[start][split] + encodedSubStrings[split + 1][end];
                        if (encodedSubStrings[start][end].length() > combinedEncoding.length()) {
                            encodedSubStrings[start][end] = combinedEncoding;
                        }
                    }
                }
            }
        }

        // The encoded string of the entire input is located at the top-left corner of the matrix.
        return encodedSubStrings[0][n - 1];
    }

    private String encodeSubstring(int start, int end) {
        // Extract the substring to be encoded
        String substring = originalString.substring(start, end + 1);

        // Substrings shorter than 5 characters shouldn't be encoded.
        if (substring.length() < 5) {
            return substring;
        }

        // Search for repeated patterns by concatenating the substring with itself
        // and looking for the index of the second occurrence of the substring.
        int repeatIndex = (substring + substring).indexOf(substring, 1);

        // If the repeated pattern exists within the length of the original substring,
        // encode the pattern.
        if (repeatIndex < substring.length()) {
            int repeatCount = substring.length() / repeatIndex;
            String pattern = encodedSubStrings[start][start + repeatIndex - 1];

            // Generate the encoded string with repetition count and pattern.
            return String.format("%d[%s]", repeatCount, pattern);
        }

        // If no repeated pattern was found, return the original substring.
        return substring;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string encode(string s) {
        int n = s.size(); // The length of the input string
        vector<vector<string>> dp(n, vector<string>(n));
        // dp[i][j] holds the shortest encoded string for s[i..j]

        auto encodeSubstring = [&](int i, int j) -> string {
            string t = s.substr(i, j - i + 1); // Substring from s[i] to s[j]
            // If the length of the substring is less than 5, do not encode it as it wouldn't be beneficial
            if (t.size() < 5) {
                return t;
            }
            // Check if the substring can be collapsed; i.e. it is a repeat of a smaller string
            int k = (t + t).find(t, 1);
            if (k < t.size()) {
                int cnt = t.size() / k; // Count the number of repeats
                return to_string(cnt) + "[" + dp[i][i + k - 1] + "]"; // Encode as cnt[sub_encoded_string]
            }
            return t; // If not collapsible, just return the original substring
        };

        // Build the dp array from the bottom up, from shorter to longer substrings
        for (int i = n - 1; i >= 0; --i) { // Start from the end of the string
            for (int j = i; j < n; ++j) { // From the current position to the end
                dp[i][j] = encodeSubstring(i, j); // Encode substring s[i..j]
                // If the length of the substring is more than 4 characters, try to split it and encode separately
                if (j - i + 1 > 4) {
                    for (int k = i; k < j; ++k) {
                        string t = dp[i][k] + dp[k + 1][j]; // The string got by encoding s[i..k] and s[k+1..j]
                        if (t.size() < dp[i][j].size()) { // Check if this new string is shorter than the current encoded one
                            dp[i][j] = t; // If yes, then update the dp array with this new shorter string
                        }
                    }
                }
            }
        }
        // Finally, return the encoded string of the entire string s
        return dp[0][n - 1];
    }
};

```

Typescript Solution

```typescript
function encode(s: string): string {
    // Initialize the length of the string
    const stringLength = s.length;

    // Create a 2D array to store the results of subproblems
    const dp: string[][] = new Array(stringLength).fill(0).map(() => new Array(stringLength).fill(''));

    // Helper function to find the encoded string for a substring
    const getEncodedSubstring = (start: number, end: number): string => {
        // Get the current substring to be encoded
        const substring = s.slice(start, end + 1);

        // Base case: for short substrings, encoding is not needed
        if (substring.length < 5) {
            return substring;
        }

        // Check if the substring is a repeated pattern by concatenating
        // it with itself and checking for the pattern occurrence
        const repeatIndex = substring.repeat(2).indexOf(substring, 1);

        // If repetition is found, encode as a repeated count and pattern
        if (repeatIndex < substring.length) {
            const repeatCount = Math.floor(substring.length / repeatIndex);
            return repeatCount + '[' + dp[start][start + repeatIndex - 1] + ']';
        }

        // If no repetition pattern is found, return the original substring
        return substring;
    };

    // Bottom-up approach to fill the 2D array with encoded substrings
    for (let i = stringLength - 1; i >= 0; --i) {
        for (let j = i; j < stringLength; ++j) {
            dp[i][j] = getEncodedSubstring(i, j);

            // Check for encoding possibilities by splitting the substring
            if (j - i + 1 > 4) {
                for (let k = i; k < j; ++k) {
                    const combined = dp[i][k] + dp[k + 1][j];
                    // Update the encoded string if a shorter encoding is possible
                    if (combined.length < dp[i][j].length) {
                        dp[i][j] = combined;
                    }
                }
            }
        }
    }

    // Return the encoded string for the entire input
    return dp[0][stringLength - 1];
}

```

Time and Space Complexity

The provided Python code is a dynamic programming solution for the problem of encoding the minimum length of a string where the string can be encoded by the number of repetitions and a pattern inside the brackets. To analyze the time and space complexity, we need to consider the operations performed inside the nested loops and the recursive calls.

Time Complexity

The time complexity of the code is determined by the three nested loops and the string operations inside the helper function `g`.

- The outer loop runs from `n-1` down to `0`, thus running `n` times.
- The second loop, for variable `j`, will run at most `n` times for each `i`.
- The third loop is used to find the optimal partition of the string for encoding and runs at most `n` times for each pair of `i` and `j`.

Additionally, the helper function `g` includes a string pattern check using `(t + t).index(t, 1)` which can be considered O(n) in the worst case because it could potentially scan the entire doubled string to find the pattern start. Thus, this operation could contribute significantly to the execution time.

Considering all these factors, the worst-case time complexity is O(n^3) for the loops multiplied by O(n) for the string pattern checking, leading to an overall time complexity of `O(n^4)`.

Space Complexity

The space complexity includes the space required for the dynamic programming table `f` and the stack space used by the helper function `g`.

- The dynamic programming table `f` is a 2D array of size `n x n`, contributing `O(n^2)` space complexity.
- The helper function `g` uses a temporary string `t`, but this does not increase the asymptotic space complexity since it requires space proportional to the input string `s`.

So the overall space complexity of the code is `O(n^2)` due to the 2D array `f`.