2746. Decremental String Concatenation

Medium Array String Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given an array called `words`, which contains 'n' strings, each indexed from `0` to `n-1`. We need to combine these strings using a special '**join**' operation which merges two strings together. But there's a twist: if the last character of the first string is the same as the first character of the second string, we have to remove one of these matching characters while joining them. The goal is to perform `n - 1` join operations to generate a single string in such a way that the final string's length is as small as possible.

Here's what we are allowed to do during each join operation:

- We can append the next string in the array `words` to the current string `str_i`.
- Alternatively, we can prepend the next string in the array `words` to the current string `str_i`.

Intuition

To arrive at the solution approach, consider that at every step of joining, we have two choices: either append or prepend the next word. The impact of this choice on the final string length can vary depending on the matching characters at the edges of the strings being joined.

We aim to minimize the overall length each time we make a join. To do this for `n - 1` operations, an intuition may lead us to dynamic programming or recursion to check every possibility, remembering the results of sub-problems to avoid redundant calculations.

Since the problem is about making choices at each step and optimizing for the 'smallest length', and considering that each choice might impact the subsequent ones, we can visualize the decision-making process as a tree. This is a common characteristic of problems where backtracking or dynamic programming is a fit.

The solution uses recursion with memoization (notice the use of the `@cache` decorator) to explore all possible concatenations and record the results for sub-problems, thus avoiding the recalculation of the same state. The `dfs` function takes into account the current position `i`, as well as the potential characters at the beginning and end of the current concatenated string (`a` and `b`) to make decisions leading towards the minimum length of the final string.

Each call to the `dfs` function considers the effect of performing a join operation via appending or prepending the current word to the string being built so far and tracks which operation gives the shorter result. The use of a recursive function with memoization allows the efficient solving of this optimization problem by making the complexity feasible for larger input sizes.

Solution Approach

The implementation relies on **Depth-First Search (DFS)** recursion with **memoization** to efficiently explore all possible concatenation sequences. The `@cache` decorator is used to memoize the results. Let's examine how the solution works by breaking down the key components of the implementation:

- The function `dfs` is the core of this solution. It takes three parameters: the current index `i` in the `words` array, a character `a` representing the beginning character of the string up to this point, and a character `b` representing the ending character of the string up to this point.
- The base case for the recursion is when `i` exceeds the number of words we have. At that point, there are no more words to join, and the function returns `0` because there is no extra length to add from additional words.
- When inside `dfs`, we consider the next word `s` to be joined with the current string. We have two choices resembling a binary tree's branches:
  - **Append**: Where we consider adding `s` to the end of the current string, so we call `dfs(i + 1, a, s[-1])`, passing `s[-1]` as the new ending character since we're appending. We subtract `1` from the length if the current ending character `b` matches the starting character of `s`.
  - **Prepend**: Where we consider adding `s` at the beginning, so we call `dfs(i + 1, s[0], b)`, passing `s[0]` as the new starting character. We subtract `1` from the length if the ending character `s[-1]` matches the current beginning character `a`.
- The recursive calls return the sum of the length of the current word `s` and the minimum length from the two choices (append or prepend). The subtraction of `1` accounts for the removed character when there is a match.
- The final return statement `return len(words[0]) + dfs(1, words[0][0], words[0][-1])` initiates the process with the first word, taking its length and appending the result of the recursive call starting from the second word with the first word's beginning and ending characters.

The `@cache` decorator on `dfs` ensures that for each unique set of inputs to the function (combinations of `i`, `a`, and `b`), the result is stored. If the same inputs are ever called again, the stored result is returned immediately, which significantly reduces the number of recursive calls, especially for large arrays with many possible combinations.

Overall, this solution showcases a classic recursive problem-solving strategy enhanced with memoization, a powerful way to optimize recursive depth-first search algorithms.

Example Walkthrough

Let's take an example array of `words`: `["abc", "cda", "dac"]`. We will walk through the process to understand how we might join these strings to minimize the overall length.

Firstly, starting with the first word "abc", we initiate the recursive process. There are no characters on either side, so `a` and `b` are 'a' and 'c' respectively, which represent the first and last characters of the current string.

Now, we look at the next word "cda". We have two choices:

1. **Append**: If we append "cda" to "abc", the resulting string would be "abccda" because the last character of "abc" is the same as the first of "cda", we can combine them by removing one 'c'. So, the string becomes "abcdac". We then call `dfs(2, 'a', 'a')` since we have moved on to the next word and the end character has been updated to 'a' from "cda".
2. **Prepend**: If we prepend "cda" to "abc", the resulting string would be "cdabc". There is no matching character in this case, so no character is removed. We then call `dfs(2, 'c', 'c')`, updating the start character to 'c'.

The return value for each of these choices would be the length of "cda" (which is 3) plus the minimum of the lengths obtained from appending or prepending the next word "dac".

Continuing with the recursion, we now have the third word "dac" and the choices described above for the second word:

1. If we chose to append during the previous operation, we now have "abcdac":
   - **Append**: Append "dac" leading to "abcdacdac", and after removing the duplicate 'a', we have "abcdacdc". The recursive call would be `dfs(3, 'a', 'c')`.
   - **Prepend**: Prepend "dac" leading to "dacabcdac", and after removing the duplicate 'd', it remains the same as prepending will not benefit us this time. The recursive call would be `dfs(3, 'd', 'c')`.
2. If we chose to prepend in the previous operation, our string would be "cdabc":
   - **Append**: Append "dac" leading to "cdabcdac", with no character removed. The recursive call would be `dfs(3, 'c', 'c')`.
   - **Prepend**: Prepend "dac" to "cdabc" leading to "daccdabc" and after removing the duplicate 'c', we have "dacdabc". The recursive call would be `dfs(3, 'd', 'c')`.

For each of these calls, we consider the removal of a duplicate character if there's a match at any end.

When `i` becomes equal to `n`, the recursion has considered all words, and we return `0` since there are no more words to add length from.

The memoization stores the results of each recursive call like `dfs(2, 'a', 'a')`, `dfs(2, 'c', 'c')`, `dfs(3, 'a', 'c')`, etc., so if the function with the same set of parameters is called again, it doesn't recalculate but retrieves the value from the cache, saving time.

In our example, assuming optimal choices were made, the solution might deduce that calling `dfs(3, 'a', 'c')` results in the smallest final string length. Ending with an efficient concatenated string "abcdacdc" or "dacdabc", depending on the sequence of operations chosen by the DFS algorithm.

Python Solution

```python
class Solution:
    def minimize_concatenated_length(self, words: List[str]) -> int:
        # Introduce caching to avoid recomputing overlapping subproblems
        @functools.lru_cache(None)
        def search(i: int, first_char: str, last_char: str) -> int:
            # Base case: all words have been considered
            if i >= len(words):
                return 0  # No extra length is added

            current_word = words[i]  # Current word at index i
            # Recursive case for appending the current word to the first string
            # Subtract 1 if the first character of current word matches the last character of the second string
            append_to_first = search(i + 1, first_char, current_word[-1]) - int(current_word[0] == last_char)
            # Recursive case for appending the current word to the second string
            # Subtract 1 if the last character of current word matches the first character of the first string
            append_to_second = search(i + 1, current_word[0], last_char) - int(current_word[-1] == first_char)

            # Add the length of the current word to the minimum of both scenarios
            return len(current_word) + min(append_to_first, append_to_second)

        # Start the search with the first word and its first and last characters
        return len(words[0]) + search(1, words[0][0], words[0][-1])

# Example usage:
# solution = Solution()
# result = solution.minimize_concatenated_length(["abc", "bcd", "cde"])
```

Lines added include `functools.lru_cache(None)` to cache results of subproblems, explanatory comments, and better-named variables for readability. The `List` type also should be imported from `typing` at the beginning of the file, as follows:

```python
from typing import List
import functools

```

Java Solution

```java
class Solution {
    private Integer[][][] memoization; // 3D array for memoization to save calculated values
    private String[] words;            // array of words
    private int wordCount;             // number of words

    // The entry method that initiates the process
    public int minimizeConcatenatedLength(String[] words) {
        this.wordCount = words.length;  // store the number of words
        this.words = words;             // store the array of words
        memoization = new Integer[wordCount][26][26];  // initialize the 3D array for memoization
        // Start the recursion with the first word, and use the first and last characters to track the state
        return words[0].length() + dfs(1, words[0].charAt(0) - 'a',
                                       words[0].charAt(words[0].length() - 1) - 'a');
    }

    // Recursive method to find the minimum concatenated length of words
    private int dfs(int currentIndex, int firstCharIndex, int lastCharIndex) {
        // Base case: if we have processed all words, return 0
        if (currentIndex >= wordCount) {
            return 0;
        }
        // If this state has already been computed, return the stored value
        if (memoization[currentIndex][firstCharIndex][lastCharIndex] != null) {
            return memoization[currentIndex][firstCharIndex][lastCharIndex];
        }

        String currentWord = words[currentIndex];          // get the current word
        int currentWordLength = currentWord.length();      // length of the current word
        int newLastCharIndex = currentWord.charAt(currentWordLength - 1) - 'a';  // index of the last character

        // Choose the current word to be at the start of remaining part and compute the remaining minimum
        int choiceAtStart = dfs(currentIndex + 1, firstCharIndex, newLastCharIndex) -
                            (currentWord.charAt(0) - 'a' == lastCharIndex ? 1 : 0);

        // Choose the current word to be at the end of previously considered words and compute the remaining minimum
        int choiceAtEnd = dfs(currentIndex + 1, currentWord.charAt(0) - 'a', lastCharIndex) -
                           (newLastCharIndex == firstCharIndex ? 1 : 0);

        // Store the computed value in memoization table and return it,
        // The total minimum length for this state is the length of the current word plus
        // the minimum of choosing the word at the start or the end
        memoization[currentIndex][firstCharIndex][lastCharIndex]
            = currentWordLength + Math.min(choiceAtStart, choiceAtEnd);

        return memoization[currentIndex][firstCharIndex][lastCharIndex];
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    // Function to minimize concatenated length of the given words sequence
    int minimizeConcatenatedLength(vector<string>& words) {
        int numWords = words.size(); // Number of words in the vector
        int dp[numWords][26][26]; // Dynamic Programming (DP) table
        memset(dp, 0, sizeof(dp)); // Initialize the DP table with zeros

        // Recursive depth-first search function to find the
        // minimized concatenation length for the given words
        // starting from index i with a and b as the last characters
        function<int(int, int, int)> dfs = [&](int index, int lastCharOfPrev, int firstCharOfNext) {
            if (index >= numWords) { // Base case: reached the end of the words vector
                return 0;
            }

            if (dp[index][lastCharOfPrev][firstCharOfNext]) { // Already calculated this subproblem
                return dp[index][lastCharOfPrev][firstCharOfNext];
            }

            string currentWord = words[index]; // Current word to process
            int wordLength = currentWord.size(); // Length of the current word
            int option1 = dfs(index + 1, lastCharOfPrev, currentWord[wordLength - 1] - 'a') - (currentWord[0] - 'a' == firstCharOfN
            int option2 = dfs(index + 1, currentWord[0] - 'a', firstCharOfNext) - (currentWord[wordLength - 1] - 'a' == lastCharOfP

            // Store the result in the DP table for memoization
            // Adding the length of the current word and taking the minimum option
            return dp[index][lastCharOfPrev][firstCharOfNext] = wordLength + min(option1, option2);
        };

        // Start DFS with the first word and the last characters of the first word
        return words[0].size() + dfs(1, words[0].front() - 'a', words[0].back() - 'a');
    }
};

```

Typescript Solution

```typescript
// Define the function to minimize the concatenated length of strings in an array
function minimizeConcatenatedLength(words: string[]): number {
    const numWords = words.length;

    // Create a 3D array to store intermediate results for memoization
    // f[i][a][b] represents the minimum length from words[i], ending
    // with char a and before words[i] starts with char b.
    const memo: number[][][] = Array.from({ length: numWords }, () =>
        Array.from({ length: 26 }, () => Array(26).fill(0))
    );

    // Define a helper method for depth-first search
    const dfs = (index: number, prevLastCharIndex: number, nextFirstCharIndex: number): number => {
        // Base case: when all words have been considered, return 0
        if (index >= numWords) {
            return 0;
        }
        // If the result is already computed, return it to avoid redundant calculations
        if (memo[index][prevLastCharIndex][nextFirstCharIndex] > 0) {
            return memo[index][prevLastCharIndex][nextFirstCharIndex];
        }
        // Get the current word and its length
        const currentWord = words[index];
        const currentWordLength = currentWord.length;

        // Calculate the next value recursively while reducing 1 if the first or last character matches
        // the adjacent character index from the previous or next word, respectively
        // The two options represent attaching the current word at the beginning or the end
        const option1 =
            dfs(index + 1, prevLastCharIndex, currentWord.charCodeAt(currentWordLength - 1) - 97) -
            (currentWord.charCodeAt(0) - 97 === nextFirstCharIndex ? 1 : 0);
        const option2 =
            dfs(index + 1, currentWord.charCodeAt(0) - 97, nextFirstCharIndex) -
            (currentWord.charCodeAt(currentWordLength - 1) - 97 === prevLastCharIndex ? 1 : 0);

        // Store the result (minimum of the two options) in the memo array and return it
        return (memo[index][prevLastCharIndex][nextFirstCharIndex] = Math.min(option1, option2) + currentWordLength);
    };

    // Call the helper function with the first word and initial indices for previous last character and next first character
    return (
        words[0].length +
        dfs(1, words[0].charCodeAt(0) - 97, words[0].charCodeAt(words[0].length - 1) - 97)
    );
}

```

Time and Space Complexity

The given Python code implements a solution to minimize the concatenated length of a list of words by exploiting memoization (using the `@cache` decorator, which caches the results of the `dfs` function calls). Analyzing the provided code, we can deduce the complexity as follows:

Time Complexity

The function `dfs(i: int, a: str, b: str)` is recursively called with different parameters, which correspond to the current index (`i`), the first character of the last added word to string `a` (`a`), and the last character of the last added word to string `b` (`b`). Since memoization is used, each state will be computed at most once. There are `n` possible indices (`n` being the size of `words`), and each word can provide 26 possible characters for `a` and `b` (assuming the input is lower-case English letters). Therefore, we have:

- Number of states: `n * 26 * 26`

For each state, operations are performed in constant time (`O(1)`) considering there is no loop within `dfs`, just two recursive calls and a few constant-time conditions and assignments:

- Operation per state: `O(1)`

Therefore, the total time complexity is the number of states times the operations per state, which is:

- Total Time Complexity: `O(n * 26^2)`

Space Complexity

The space complexity is mainly affected by the call stack during the recursive calls, and the space used for memoization. In the worst case, the recursion depth can go up to `n`:

- Recursion stack: `O(n)`

The memoization will store results for every possible state:

- Memoization storage: `O(n * 26^2)`

Hence, the total space complexity can be seen as the maximum of the above two concerns. Since `n * 26^2` is typically greater than just `n`:

- Total Space Complexity: `O(n * 26^2)`

Note: The exact constants for the number of characters would depend on the character set allowed in the input.