2707. Extra Characters in a String

Medium Trie Array Hash Table String Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given a string `s` which is indexed starting at 0. We are also provided with a `dictionary` containing several words. Our goal is to split the string `s` into one or more non-overlapping substrings, with the condition that each substring must be a word that exists in the `dictionary`. It's possible that there are some characters in `s` that do not belong to any of the words in the dictionary, in which case they are considered as extra characters.

The challenge is to break `s` into substrings in such a way that the number of these extra characters is minimized. The task requires finding an optimal way to perform the breakup of `s` to achieve the least number of characters that cannot be associated with any dictionary words. The output of the problem is the minimum number of extra characters resulting from the optimal breakup of `s`.

Intuition

The approach to solving this problem revolves around dynamic programming—a method for solving a complex problem by breaking it down into a collection of simpler subproblems. The idea is to compute the best solution for every prefix of the string `s`, keeping track of the minimum number of extra characters at each step.

Let's create an array `f`, where `f[i]` represents the minimum number of extra characters if we consider breaking the string up to index `i - 1` (since our string is 0-indexed). We initialize this array in such a way that `f[0]` is 0 because no extra characters are present when no substring is considered.

We iterate over the length of the string `s`, and for each index `i`, we initially set `f[i]` to `f[i - 1] + 1`, which implies that the default action is not to match the character at `s[i - 1]` with any word in the dictionary (hence it's counted as an extra character).

Next, we need to check for every substring of `s` that ends at index `i` if it matches with any word in our dictionary. We do this by iterating from `j = 0` to `j = i`. For each `j`, we're effectively trying to match the substring `s[j:i]`. If a match is found (`s[j:i]` is in our dictionary), we update `f[i]` to be the minimum of its current value and `f[j]`, since using this word in our split would mean adding no extra characters from this substring.

The dynamic programming recurrence here is leveraging the idea that the optimal breakup of the string at position `i` depends on the optimal breakups of all the prior smaller substrings ending before `i`. By continually updating our `f` array, we build up the solution for the entire string. After the iteration completes, `f[n]` will contain the minimum number of extra characters for the entire string `s`.

This solution is efficient because we're reusing the results of subproblems rather than recalculating from scratch, hence displaying optimal substructure and overlapping subproblems—key characteristics of dynamic programming.

Solution Approach

In the solution code provided, we follow a dynamic programming approach, as this is an optimization problem where we aim to reduce the number of extra characters by breaking the string into valid dictionary words. The idea is to use a list `f` with the size `n + 1` where `n` is the length of the string `s`, to store the computed minimum number of extra characters at each position.

Let's walk through the key elements of the implementation:

1. **Initial Setup**: We start by converting the `dictionary` into a set called `ss` for constant-time lookup operations, which is faster than looking for words in a list.

2. **Array Initialization**: We initialize an array `f` with `n + 1` elements, where `n` is the length of `s`. We'll use this array to store the minimum number of extra characters at each index `i` of the string. The list is initialized with zeros.

3. **Dynamic Programming Iteration**:

   - We iterate `i` from `1` to `n` (inclusive) to consider all prefix subproblems of `s`.
   - At the start of each iteration, we set `f[i] = f[i - 1] + 1`, assuming the current character `s[i - 1]` will be an extra character if we can't match it with a dictionary word ending at this position.
   - We then check for all possible endings for words that end at index `i`. For this, we iterate over all `j` from `0` to `i` and check if the substring `s[j:i]` exists in our set `ss`.
   - If `s[j:i]` is a dictionary word, we compare and update `f[i]` with `f[j]`, if `f[j]` is smaller, as that means we can create a valid sentence ending at `i` without adding any extra character for this particular substring.
4. **Avoiding Recalculation**: Instead of computing whether each possible substring is in the dictionary, by traversing the entire dictionary, we use the fact that the dictionary entries have been stored in a set `ss`, which allows us to check the presence of a substring in constant time.

5. **Returning the Result**: After the loop completes, `f[n]` contains the minimum number of extra characters left over if we break up `s` optimally.

The solution leverages dynamic programming with the base case that no extra characters are needed when no string is processed (`f[0] = 0`). It uses iterative computations to build and improve upon the solutions to subproblems, leading to an optimal overall solution. The efficient data structure (set) is used to optimize the lookup time for checking words in the dictionary, and the iterative approach incrementally builds the solution without unnecessary recalculations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we are given the string `s = "leetcode"` and the `dictionary` with the words `["leet", "code"]`.

1. **Initial Setup**: We first convert the `dictionary` into a set `ss`. So `ss = {"leet", "code"}` for quick lookups.

2. **Array Initialization**: Since `s` has a length `n = 8`, we initialize an array `f` with `n + 1 = 9` elements, all set to zero: `f = [0, 0, 0, 0, 0, 0, 0, 0, 0]`.

3. **Dynamic Programming Iteration**:

   - Iterating through `i` from `1` to `8`, we consider all combinations of substrings.
   - At `i = 1`, we set `f[1] = f[0] + 1 = 1`, treating the first character 'l' as an extra character.
   - As we proceed, we check substrings of `s` ending at each `i`.
   - When we reach `i = 4`, we find that the substring `s[0:4] = "leet"` is in `ss`. So we update `f[4]` to `min(f[4], f[0]) = min(4, 0) = 0`.
   - Continuing, each character 'c', 'o', 'd', 'e' initially increments the extra count.
   - At `i = 8`, checking the substring `s[4:8] = "code"`, which is also in `ss`, we update `f[8]` to `min(f[8], f[4]) = min(4, 0) = 0`.
4. **Avoiding Recalculation**: During this process, we avoid recalculating by using the set `ss` for checking the dictionary presence.

5. **Returning the Result**: After completing the iteration, `f[8]` indicates there are `0` extra characters needed, showing the string `s` can be perfectly split into words from the `dictionary`.

This process demonstrates the power of resourcefully applying dynamic programming to reduce the problem into subproblems while using optimal previous decisions at each step. The choice of using a set for the dictionary makes the lookup operations much more efficient, and as we build out `f`, we're effectively making the best decision at each step based on the previously computed states.

Python Solution

```python
# Define the Solution class
class Solution:
    def minExtraChar(self, s: str, dictionary: List[str]) -> int:
        # Convert the list of words in the dictionary to a set for faster lookup
        word_set = set(dictionary)
        # Get the length of the string s
        n = len(s)
        # Initialize an array to store the minimum number of extra characters
        # needed to form substrings present in the dictionary
        # 'f[i]' represents the minimum extra characters from the substring s[:i]
        min_extras = [0] * (n + 1)

        # Iterate through the string s
        for i in range(1, n + 1):
            # Assume that adding one more character increases the count of extra characters
            min_extras[i] = min_extras[i - 1] + 1
            # Check all possible substrings ending at index 'i'
            for j in range(i):
                # If the substring s[j:i] is in the dictionary and
                # using it reduces the count of extra characters,
                # update the minimum extra characters accordingly
                if s[j:i] in word_set and min_extras[j] < min_extras[i]:
                    min_extras[i] = min_extras[j]

        # Return the minimum number of extra characters needed for the whole string
        return min_extras[n]

# Note: The 'List' should be imported from 'typing', and the overwritten method
# names have not been changed as per the instructions.
```

Remember to include the import statement for `List` from the `typing` module at the beginning of your code file if it's not already present, as the function signature uses `List` for type hinting.

```python
from typing import List

```

Java Solution

```java
class Solution {
    public int minExtraChar(String s, String[] dictionary) {
        // Create a set from the dictionary for efficient lookup of words.
        Set<String> wordSet = new HashSet<>();
        for (String word : dictionary) {
            wordSet.add(word);
        }

        int n = s.length(); // Get the length of the string s

        // Create an array to store the minimum number of extra characters needed.
        // f[i] will be the minimum count for substring s[0..i)
        int[] minExtraChars = new int[n + 1];
        minExtraChars[0] = 0; // Base case: no extra characters needed for an empty string

        // Iterate over each character in the string
        for (int i = 1; i <= n; ++i) {
            // By default, assume one more extra char than the minExtraChars of the previous substring
            minExtraChars[i] = minExtraChars[i - 1] + 1;

            // Check each possible substring ending at current character i
            for (int j = 0; j < i; ++j) {
                // If the substring from index j to i is in the dictionary,
                // update minExtraChars[i] if a smaller value is found
                if (wordSet.contains(s.substring(j, i))) {
                    minExtraChars[i] = Math.min(minExtraChars[i], minExtraChars[j]);
                }
            }
        }

        // Return the minimum extra characters for the entire string
        return minExtraChars[n];
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the minimum number of extra characters
    // needed to construct the string 's' using the words from the 'dictionary'.
    int minExtraChar(std::string s, std::vector<std::string>& dictionary) {
        // Transform the dictionary into an unordered set for constant-time look-ups.
        std::unordered_set<std::string> wordSet(dictionary.begin(), dictionary.end());

        int stringLength = s.size();
        std::vector<int> dp(stringLength + 1);
        dp[0] = 0; // Base case: no extra character is needed for an empty substring.

        // Calculate the minimum number of extra characters for each substring ending at position 'i'.
        for (int i = 1; i <= stringLength; ++i) {
            // Initialize dp[i] assuming all previous characters are from the dictionary and only s[i-1] is extra.
            dp[i] = dp[i - 1] + 1;

            // Check all possible previous positions 'j' to see if s[j...i-1] is a word in the dictionary.
            for (int j = 0; j < i; ++j) {
                if (wordSet.count(s.substr(j, i - j))) {
                    // If s[j...i-1] is a word, update dp[i] to be the minimum of its current value
                    // and dp[j], which represents the minimum number of extra characters needed to form substring s[0...j-1].
                    dp[i] = std::min(dp[i], dp[j]);
                }
            }
        }

        // dp[stringLength] holds the minimum number of extra characters needed for the entire string.
        return dp[stringLength];
    }
};

```

Typescript Solution

```typescript
function minExtraChar(s: string, dictionary: string[]): number {
    // Convert the dictionary to a Set for faster lookup
    const wordSet = new Set(dictionary);

    // Get the length of the string
    const strLength = s.length;

    // Initialize an array to store the minimum number of extra characters needed
    // to reach each position in the string
    const minExtraChars = new Array(strLength + 1).fill(0);

    // Iterate over the string
    for (let endIndex = 1; endIndex <= strLength; ++endIndex) {
        // By default, assume we need one more extra character than the previous position
        minExtraChars[endIndex] = minExtraChars[endIndex - 1] + 1;

        // Check all possible substrings that end at the current position
        for (let startIndex = 0; startIndex < endIndex; ++startIndex) {
            // Extract the current substring
            const currentSubstring = s.substring(startIndex, endIndex);

            // If the current substring is in the dictionary, update the minimum extra chars needed
            if (wordSet.has(currentSubstring)) {
                minExtraChars[endIndex] = Math.min(minExtraChars[endIndex], minExtraChars[startIndex]);
            }
        }
    }

    // Return the minimum extra characters needed for the entire string
    return minExtraChars[strLength];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by two nested loops. The outer loop runs `n` times, where `n` is the length of the input string `s`. Inside the outer loop, there is an inner loop that also can run up to `n` times, since `j` ranges from `0` to `i-1`. During each iteration of the inner loop, we check if `s[j:i]` is in the set `ss`. Since `ss` is a set, the lookup operation takes `O(1)` on average.

However, considering the worst-case scenario of slicing the string `s[j:i]`, which can take `O(k)` time (where `k` is the length of the substring), the total time complexity of the nested loops can be `O(n^3)` if we multiply `n` (from the outer loop), by `n` (from the possible slices in an inner loop), by `k` (for the slicing operation, which in the worst case can be `n`). For large strings, this might not be efficient.

But since we know strings in Python are immutable and slicing a string actually creates a new string, the slicing operation should be considered when calculating the complexity. However, the slicing operation can sometimes be optimized by the interpreter, and the complexity may be less in practice, but to be rigorous we often consider the worst case.

So, breaking it down:

- Outer loop runs `n` times: `O(n)`
- Inner loop can run up to `n` times: `O(n)`
- Slicing string operation: `O(n)`

Multiplying these factors together, we get `O(n^3)` for the time complexity.

Space Complexity

For space complexity, there is an auxiliary array `f` of size `n + 1` being created. Additionally, we have the set `ss`, which can contain up to the number of words in the `dictionary`. However, since the problem statement doesn't provide the size of the dictionary relative to `n`, and typically, space complexity is more concerned with scalability regarding the input size `n`, we focus on `n`.

The space complexity for the array `f` is thus `O(n)`. The space required for the set `ss` depends on the size of the `dictionary`, but since it is not mentioned to scale with `n`, we can consider it a constant factor for the analysis of space complexity relative to `n`.

Summing it up:

- Array `f`: `O(n)`
- Set `ss`: Size of the dictionary (constant with respect to `n`)

Consequently, the overall space complexity of the code is `O(n)`.