2430. Maximum Deletions on a String

Problem Description

In this problem, you are given a string `s` that consists only of lowercase English letters. Your task is to determine the maximum number of operations you can perform to delete the entire string. There are two types of operations you can perform:

1. Delete the entire string at once.
2. Delete the first `i` letters of the string if and only if the first `i` letters are exactly the same as the next `i` letters. This can be done for any `i` satisfying `1 <= i <= s.length / 2`.

For example, let's assume you have the string `s = "ababc"`. In one possible operation, you can delete the first two letters (`"ab"`) because the first two letters (`"ab"`) and the two following letters (`"ab"`) are equal. This will leave you with the string `"abc"`.

The goal is to return the *maximum number* of operations that can be applied to delete all of `s`.

Intuition

To solve this problem, one approach is to use dynamic programming to break down the problem into smaller, more manageable subproblems. With dynamic programming, you can determine the optimal sequence of operations for each substring. The recursive function `dfs(i)` is defined to determine the maximum operations that can be performed starting from the `i`-th character of the string up to the end.

The key intuition for the solution is as follows:

- If we are at the end of the string (index `i` is equal to `n`, the length of the string), there are no operations left to perform, hence the base case of the recursion is `dfs(i) = 0`.
- For any position `i` in the string, we try to find any index `i + j` such that `s[i : i + j]` is equal to `s[i + j : i + 2 * j]`, which means we can perform an operation to delete `s[i : i + j]`.
- For each valid `j` where a deletion can be performed, we recursively solve the problem for the remainder of the string starting from index `i + j`.
- We use memoization (cache) to remember the results of subproblems we've already solved to avoid redundant calculations and improve efficiency.
- The final answer is the maximum number of operations we can perform, which is the best result out of all possible deletions from index `i`.

Solution Approach

The solution makes use of dynamic programming and recursion to solve the problem. Dynamic programming is an optimization technique that solves problems by breaking them down into simpler subproblems and storing the results (usually in an array or hash table) to avoid redundant computations.

Here's how the implementation works:

1. We define a recursive function `dfs(i)` which computes the maximum number of operations that can be performed starting from index `i`. This helps us in breaking down the larger problem into smaller ones.
2. The base case of our recursion occurs when `i` reaches `n`, the length of `s`, meaning that there is no more string left to delete. In this case, `dfs(i)` returns `0` because no operations can be performed.
3. For any given index `i`, we iterate through all possible values of `j` such that `1 <= j <= (n - i) / 2`. These values represent potential points where we can split the string and perform a delete operation if the substring can be matched with the following string of the same length.
4. During the iteration, we check if the substring `s[i : i + j]` is equal to `s[i + j : i + 2 * j]`. If they are equal, we can delete `s[i : i + j]`. We then call `dfs(i + j)` to find out how many more operations can be performed starting from index `i + j`.
5. We use the `max` function to keep track of the maximum number of operations that we can perform. The variable `ans` is updated with `1 + dfs(i + j)` if a match is found since we have performed one operation plus however many more we can perform from the new starting point.
6. To ensure the solution is efficient, we use the `@cache` decorator from Python's `functools` module for memoization. This stores the result of `dfs(i)` the first time it is computed for any index `i`, and subsequent calls with the same `i` will use the cached result instead of recomputing it.
7. At the end, the function `dfs(0)` is called to kick off the recursion from the beginning of the string, and the maximum number of operations for the entire string is returned.

By caching intermediate results, the implementation ensures that each subproblem is solved only once, leading to a significant performance improvement, particularly for larger strings. This is a common optimization in dynamic programming solutions to reduce the time complexity from exponential to polynomial.

Example Walkthrough

Let's use the string `s = "aabbccc"` to illustrate the solution approach.

1. We start by defining the recursive function `dfs(i)` to find the maximum number of operations starting from index `i`.
2. We call `dfs(0)` because we want to start from the beginning of the string.
3. At index `i`, we have the whole string `s = "aabbccc"` to work with. We look for `j`s where `1 <= j <= (n - i) / 2` which translates to `1 <= j <= 3` for our string.
4. For `j = 1`, we check if `s[0 : 1]` (`"a"`) is the same as `s[1 : 2]` (`"a"`). They match, so we can perform a delete operation. We also call `dfs(1)` to find the maximum number of operations from `s = "abbccc"`.
5. Now, inside `dfs(1)`, we repeat the process. We find that there are no `j` values that allow us to perform a delete operation, so `dfs(1)` returns `0`.
6. Since `dfs(1)` returns `0`, the maximum operations for `j = 1` at `i = 0` is `1 + dfs(1)` which equals `1`.
7. Next, we try `j = 2`. We find that `s[0 : 2]` (`"aa"`) is the same as `s[2 : 4]` (`"bb"`), which do not match, so we cannot perform a delete operation for this `j`.
8. Finally, we try `j = 3`, and find that `s[0 : 3]` (`"aab"`) is not equal to `s[3 : 6]` (`"bcc"`) so we cannot perform a delete operation here as well.
9. Since only `j == 1` allowed us to perform an operation, the answer from `dfs(0)` is `1`, which we've previously calculated.
10. No more operations can be performed, so the `max` number of operations for the entire string is `1`.
11. Using memoization with the `@cache` decorator, if `dfs(1)` was called again, it would immediately return `0` without recomputation.

This recursion and memoization process allows us to efficiently calculate the maximum number of operations that can be performed to delete the string `s = "aabbccc"`, with the final answer being `1`.

Solution Implementation

Python

```python
from functools import lru_cache

class Solution:
    def deleteString(self, string: str) -> int:
        # Use lru cache to memoize previously computed results
        @lru_cache(None)
        def dfs(index: int) -> int:
            # If we've reached the end of the string, there's no more to delete, so return 0
            if index == length:
                return 0

            # Initialize answer at 1, as there's at least the possibility of deleting the current character
            answer = 1

            # Try to find a duplicated substring starting at the current index
            for i in range(1, (length - index) // 2 + 1):
                # If a duplicate is found, recursively call dfs from the end of this duplicate substring
                if string[index : index + i] == string[index + i : index + 2 * i]:
                    # Take the max of the current answer and 1 (for this deletion) + the result of dfs from the next index
                    answer = max(answer, 1 + dfs(index + i))

            # Return the maximum number of deletions that can be made from this index
            return answer

        # Get the length of the string to avoid recalculating it
        length = len(string)

        # Begin the dfs from the start of the string
        return dfs(0)
```

Java

```java
class Solution {

    public int deleteString(String s) {
        // Length of the string
        int n = s.length();
        // dp[i:] will hold the length of the longest prefix of substring starting at i
        // which is also a prefix of substring starting at j
        int[][] longestPrefix = new int[n + 1][n + 1];

        // Calculate the longest common prefix for all possible substrings
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) {
                if (s.charAt(i) == s.charAt(j)) {
                    longestPrefix[i][j] = longestPrefix[i + 1][j + 1] + 1;
                }
            }
        }

        // dp[i] will hold the maximum number of ways to delete the substring starting at i
        int[] maxDeleteWays = new int[n];

        // Calculate the maximum number of ways to delete from each position
        for (int i = n - 1; i >= 0; --i) {
            // Initially, you can delete at least once
            maxDeleteWays[i] = 1;
            // Try to delete every possible substring length starting at i
            for (int len = 1; len <= (n - i) / 2; ++i) {
                // If the current substring can be deleted (found in its continuation)
                if (longestPrefix[i][i + len] >= len) {
                    // Update dp[i] if deleting substring leads to more delete operations
                    maxDeleteWays[i] = Math.max(maxDeleteWays[i], maxDeleteWays[i + len] + 1);
                }
            }
        }

        // Result is the maximum number of deletions starting from first character
        return maxDeleteWays[0];
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <cstring>
using namespace std;

class Solution {
public:
    // Function to determine the maximum number of times we can delete a non-empty prefix from the string.
    int deleteString(string s) {
        int length = s.size();
        // Define a matrix to store the longest common prefix information.
        vector<vector<int>> longestCommonPrefix(length + 1, vector<int>(length + 1, 0));

        // Calculate the longest common prefix for all the substrings.
        for (int i = length - 1; i >= 0; --i) {
            for (int j = i + 1; j < length; ++j) {
                if (s[i] == s[j]) {
                    longestCommonPrefix[i][j] = longestCommonPrefix[i + 1][j + 1] + 1;
                }
            }
        }

        // A vector to store the maximum number of deletions starting from each index.
        vector<int> maxDeletions(length, 1);

        // Calculate the maximum number of deletions for every prefix.
        for (int i = length - 1; i >= 0; --i) {
            // Check all the possible next parts of the string to delete.
            for (int k = 1; i + k <= length - i / 2; ++i) {
                // Check if we have a matching prefix of at least length j.
                if (longestCommonPrefix[i][i + k] >= k) {
                    // If so, update the maximum deletions at this index.
                    maxDeletions[i] = max(maxDeletions[i], maxDeletions[i + k] + 1);
                }
            }
        }

        // The maximum number of deletions starting from the beginning is the answer.
        return maxDeletions[0];
    }
};
```

TypeScript

```typescript
function deleteString(s: string): number {
    // Get the length of the input string.
    const lengthOfString = s.length;
    // Initialize an array to store the maximum number of identical contiguous substrings from each position.
    const maxDeletes: number[] = new Array(lengthOfString).fill(1);

    // Iterate over the string in reverse.
    for (let i = lengthOfString - 1; i >= 0; --i) {
        // Try to match substrings of all possible lengths starting from the current position.
        for (let substringLength = 1; substringLength <= (lengthOfString - i) >> 1; ++substringLength) {
            // Check if two contiguous substrings of half the remaining string are identical.
            if (s.slice(i, i + substringLength) === s.slice(i + substringLength, i + 2 * substringLength)) {
                // If there are, update the maximum number of deletes for the current position.
                maxDeletes[i] = Math.max(maxDeletes[i], maxDeletes[i + substringLength] + 1);
            }
        }
    }

    // Return the maximum number of identical contiguous substrings that can be deleted from the entire string.
    return maxDeletes[0];
}
```

```python
from functools import lru_cache

class Solution:
    def deleteString(self, string: str) -> int:
        # Use lru cache to memoize previously computed results
        @lru_cache(None)
        def dfs(index: int) -> int:
            # If we've reached the end of the string, there's no more to delete, so return 0
            if index == length:
                return 0

            # Initialize answer at 1, as there's at least the possibility of deleting the current character
            answer = 1

            # Try to find a duplicated substring starting at the current index
            for i in range(1, (length - index) // 2 + 1):
                # If a duplicate is found, recursively call dfs from the end of this duplicate substring
                if string[index : index + i] == string[index + i : index + 2 * i]:
                    # Take the max of the current answer and 1 (for this deletion) + the result of dfs from the next index
                    answer = max(answer, 1 + dfs(index + i))

            # Return the maximum number of deletions that can be made from this index
            return answer

        # Get the length of the string to avoid recalculating it
        length = len(string)

        # Begin the dfs from the start of the string
        return dfs(0)
```

Time and Space Complexity

Time Complexity

The provided code uses a recursive function `dfs` that explores the possibilities of splitting the string into two equal parts and checking if they are the same, then recursively applying the same logic to the rest of the string.

Considering that `n` is the length of the string `s`, the recursion could run theoretically for each starting position `i` and try to match substrings of length `j`, which can range from `1` to `(n - i) / 2`. Therefore, in the worst-case scenario where the function checks all possibilities, it would make $O(n/2)$ comparisons for each `i`. And since this is done for each `i`, the overall time for comparisons is $O(n^2/2)$.

Moreover, due to the use of memoization (indicated by `@cache`), each state `dfs(i)` is only computed once, which reduces the number of recursive computations from what would be exponential in the recursive case to linear in the number of unique states. Given that the states correspond to the starting indices of the string `s`, there are `n` unique states.

Hence, the memoization does not change the number of comparisons per se, but it does ensure that each state calculation is only done once, rather than being recomputed multiple times.

Thus, the time complexity, which involves the nested iteration and memoization, is $O(n^3/2)$ because for each `i`, you could perform $O(n/2)$ comparisons and this is done for `n` different starting positions. Hence, the final time complexity is $O(n^3)$.

Space Complexity

The space complexity of the code is influenced by the maximum size of the recursion stack and the space used by the cache to store results of previous computations.

1. **Recursion Stack**: In the worst-case scenario, the recursion can go as deep as the length of the string `s`, which is `n`. Thus, the recursion stack could potentially take up $O(n)$ space.
2. **Cache**: Since the cache stores the results for each unique state (`i`), and there are `n` possible unique states, the space complexity for memoization is also $O(n)$.

Therefore, the total space complexity is the sum of the recursion stack and the cache, which gives us $O(n) + O(n)$. However, since we drop constants in Big O notation, the space complexity simplifies to $O(n)$.