1312. Minimum Insertion Steps to Make a String Palindrome

Hard String Dynamic Programming Leetcode Link

Problem Description

The task is to make a given string `s` a palindrome by inserting any number of characters at any position in the string. The objective is to achieve this with the minimum number of insertions possible. A palindrome is a word, number, phrase, or other sequences of characters that reads the same forward and backward, ignoring spaces, punctuation, and capitalization.

Intuition

To solve this problem, we can use dynamic programming. The core idea is to build a solution using the answers to smaller subproblems. These subproblems involve finding the minimum number of insertions for all substrings of the given string and building upon those to arrive at the final answer.

We can define our subproblem as `f(i, j)`, which is the minimum number of insertions to make the substring `s[i...j]` a palindrome. Therefore, `f(0, n-1)` (where `n` is the length of the string) will eventually be our answer for the whole string `s`.

If the characters at the position `i` and `j` are the same, no insertion is needed, and `f(i, j)` will be the same as `f(i+1, j-1)` — the number of insertions needed for the substring without these two matching characters. However, if they do not match, we have to do an insertion either at the beginning or the end of the substring. This means we have two options: either insert a character matching `s[j]` before `i` or insert a character matching `s[i]` after `j`. Therefore, we'll take the minimum of `f(i+1, j)` and `f(i, j-1)` and add one (for the insertion we've made). We do this for all possible substrings starting from the end of the string and moving backward, which finally gives us the minimum number of insertions needed to make the entire string a palindrome.

The Python solution provided implements this dynamic programming approach. It utilizes a 2D array `f` where `f[i][j]` holds the minimum number of insertions needed for the substring `s[i...j]`. We iterate through the string in reverse, gradually building up the solution for the entire string and returning `f[0][-1]`, which represents the minimum number of insertions needed for the whole string `s`.

Solution Approach

The solution to this problem applies dynamic programming because a direct approach would involve checking every possible insertion, leading to an inefficient exploration of the problem space. Dynamic programming, however, allows us to solve the problem more efficiently by breaking it down into overlapping subproblems and building up the answer.

Here's a step-by-step explanation of the dynamic programming solution provided in the Reference Solution Approach:

1. We use a 2D array `f` with dimensions `n` by `n`, where `n` is the length of the string `s`. `f[i][j]` will represent the minimum number of insertions required to make the substring `s[i...j]` a palindrome.
2. We initialize our dp array `f` with zeros because if `i` equals `j`, the substring is already a palindrome, and no insertions are needed.
3. The main process occurs in a nested loop. We first iterate over `i` in reverse, starting from `n-2` down to `0`. The reason for starting at `n-2` is that the last character does not need any insertions to become a palindrome; it already is one on its own.
4. For each `i`, we then iterate over `j` from `i+1` to `n-1`. This loop considers all substrings that start at index `i` and end at index `j`.
5. Inside the nested loops, we check if the characters at index `i` and `j` are the same.
   - If `s[i]` is equal to `s[j]`, no additional insertions are needed to pair them up, so `f[i][j]` is set equal to `f[i+1][j-1]` (the minimum insertions needed for the inside substring).
   - If `s[i]` is not equal to `s[j]`, we must insert a character. We can choose to align `s[i]` with some character to the right or to align `s[j]` with a character to the left. Therefore, `f[i][j]` is the minimum of `f[i+1][j]` and `f[i][j-1]` (the solutions to subproblems considering one side extended), plus one for the current insertion.
6. At the completion of the loops, `f[0][-1]` contains the answer. It represents the minimum number of insertions needed for the whole string, because it refers to the subproblem considering the entire string `s[0...n-1]`.

By using dynamic programming, we avoid re-computing solutions to subproblems, which makes for an efficient solution. The Reference Solution Approach leverages this overlapping of subproblems and optimal substructure properties common in dynamic programming problems.

This algorithm has a time complexity of `O(n^2)` due to the nested for loops iterating over all substrings, and a space complexity of `O(n^2)` as well for storing the dp array. While the space complexity could be a concern for very long strings, the quadratic time complexity is a significant improvement over any naive approach.

Example Walkthrough

Let's consider a short example with the string `s = "abca"`. We want to find the minimum number of insertions required to make `s` a palindrome.

1. Initialize a 2D array `f` with dimensions 4×4, with all values set to 0, since the length of the string `s` is 4.

	0	1	2	3
0	0			
1		0		
2			0	
3				0

2. Fill in the dp array `f` starting from `i = 2` down to `0`. We need to loop from `j = i+1` to `3`. We ignore cases where `i == j` because those are already palindromes.
3. Start with `i = 2` and `j = 3`: `s[2] = "c"`, `s[3] = "a"`. They do not match, so we need one insertion. We take the minimum from `f[2+1][3]` and `f[2][3-1]`, which are both `0` at the moment, so after adding 1, `f[2][3] = 1`.

	0	1	2	3
0	0			
1		0		
2			0	1
3				0

4. Move to `i = 1` and `j = 2`: `s[1] = "b"`, `s[2] = "c"`. They do not match, so we take the minimum of `f[1+1][2]` and `f[1][2-1]`, with an extra insertion. Since `f[2][2]` and `f[1][1]` are `0`, we set `f[1][2]` to `1`.
5. For `i = 1` and `j = 3`, compare `s[1]` to `s[3]`, which are "b" and "a". They're different, so `f[1][3]` is the minimum of `f[1+1][3]` and `f[1][3-1]`, which are `1` and `1` at this point, plus one for the insertion; we get `f[1][3] = 2`.
6. Now for `i = 0` and `j = 1`: `s[0] = "a"`, `s[1] = "b"`. They're different, so we set `f[0][1]` to the minimum of `f[0+1][1]` or `f[0][1-1]` plus 1, which equals `1`.
7. For `i = 0`, `j = 2`, we take the minimum of `f[1][2]` and `f[0][1]`. Both are `1` currently, so `f[0][2] = 2`.
8. Finally, for `i = 0`, `j = 3`, we compare `s[0]` with `s[3]`, which are the same. So, we set `f[0][3]` to `f[1][2]`, which is `1`.

The final dp array `f` looks like this:

```
|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 1 |
| 1 |   | 0 | 1 | 2 |
| 2 |   |   | 0 | 1 |
| 3 |   |   |   | 0 |
```

The top right cell `f[0][3]` gives us the minimum number of insertions needed, which is `1`. In this case, we can insert a "b" at the end of the string to get "abcba", which is a palindrome.

This example demonstrates the procedure described in the solution approach, illustrating the steps taken to fill the dp array and find the minimum number of insertions to make the string `s` a palindrome.

Python Solution

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

        # dp (Dynamic Programming) table where dp[i][j] will hold the
        # minimum number of insertions needed to make s[i...j] a palindrome
        dp = [[0] * length for _ in range(length)]

        # Loop backwards through the string so that we can solve
        # for all the smaller subproblems first
        for i in range(length - 2, -1, -1):
            for j in range(i + 1, length):
                # If the characters at position i and j are the same,
                # no more insertions are required here since it already
                # contributes to a palindrome
                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1]
                else:
                    # If the characters are different, we need one more
                    # insertion. We can insert either at the beginning
                    # or at the end of the substring. We choose the option
                    # that requires fewer insertions, hence the min function.
                    dp[i][j] = min(dp[i + 1][j], dp[i][j - 1]) + 1

        # The top-right corner of the dp table contains the answer
        # for the whole string, which is what we return.
        return dp[0][-1]

```

Java Solution

```java
class Solution {
    public int minInsertions(String s) {
        int length = s.length();
        int[][] dp = new int[length][length]; // Using dp array to store minimum insertion results

        // Iterating in reverse order from second last character to the beginning
        for (int i = length - 2; i >= 0; --i) {
            // Iterating from the character just after i up to the end of the string
            for (int j = i + 1; j < length; ++j) {
                // If the characters at i and j match, no insertion is needed; carry over the value from the previous subproblem
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i + 1][j - 1];
                } else {
                    // If the characters do not match, find the minimum insertion from the two adjacent subproblems and add 1
                    dp[i][j] = Math.min(dp[i + 1][j], dp[i][j - 1]) + 1;
                }
            }
        }

        // The top-right corner of the DP matrix contains the answer for the whole string
        return dp[0][length - 1];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minInsertions(string s) {
        int n = s.size();
        vector<vector<int>> dp(n, vector<int>(n, 0)); // Create a DP table with `n` rows and `n` columns initialized to 0

        // Fill the DP table starting from the bottom right and moving to the top left
        for (int startIdx = n - 2; startIdx >= 0; --startIdx) { // Start from second last character since last character doesn't neec
            for (int endIdx = startIdx + 1; endIdx < n; ++endIdx) { // End index ranges from the character after startIdx to the end
                if (s[startIdx] == s[endIdx]) {
                    // No insertion needed if characters at startIdx and endIdx are the same,
                    // we just take the value from the diagonal entry before the next iteration
                    dp[startIdx][endIdx] = dp[startIdx + 1][endIdx - 1];
                } else {
                    // If characters are not the same, we take the minimum insertions needed
                    // from the positions right after startIdx or right before endIdx and add one
                    dp[startIdx][endIdx] = min(dp[startIdx + 1][endIdx], dp[startIdx][endIdx - 1]) + 1;
                }
            }
        }

        // The minimum number of insertions needed to make the string `s` a palindrome
        // is stored in the top right corner of the DP table
        return dp[0][n - 1];
    }
};

```

Typescript Solution

```typescript
// Define a function to calculate the minimum number of insertions to make a string a palindrome
function minInsertions(s: string): number {
    const n: number = s.length;
    // Create a DP table with `n` rows and `n` columns initialized to 0
    const dp: number[][] = Array.from({ length: n }, () => Array(n).fill(0));

    // Loop to fill the DP table starting from the bottom right and moving to the top left
    for (let startIdx = n - 2; startIdx >= 0; --startIdx) { // Start from the second-last character.
        for (let endIdx = startIdx + 1; endIdx < n; ++endIdx) { // Loop over the remaining characters to the right.
            if (s[startIdx] === s[endIdx]) {
                // If characters at startIdx and endIdx are the same,
                // no insertions are needed so take value from the diagonal entry before the next iteration.
                dp[startIdx][endIdx] = dp[startIdx + 1][endIdx - 1];
            } else {
                // If characters are different, find the minimum of the insertions needed
                // after startIdx or before endIdx and increment by one.
                dp[startIdx][endIdx] = Math.min(dp[startIdx + 1][endIdx], dp[startIdx][endIdx - 1]) + 1;
            }
        }
    }

    // Return the minimum number of insertions needed to make the string `s` a palindrome,
    // which is stored in the top right corner of the DP table.
    return dp[0][n - 1];
}

```

Time and Space Complexity

The given Python code snippet is designed to find the minimum number of insertions needed to make the input string a palindrome. It uses dynamic programming to accomplish this task. The analysis of time and space complexity is as follows:

Time Complexity:

The time complexity of the code is `O(n^2)`, where `n` is the length of the input string `s`. This is because there are two nested loops:

- The outer loop runs backwards from `n-2` to `0`, which contributes to an `O(n)` complexity.
- The inner loop runs from `i + 1` to `n`, which also contributes to `O(n)` complexity when considered with the outer loop.

Since each element `f[i][j]` of the DP matrix `f` is filled once and the amount of work done for each element is constant, the overall time complexity is the product of the two `O(n)` complexities.

Space Complexity:

The space complexity of the code is also `O(n^2)`. This is because a two-dimensional array `f` of size `n * n` is created to store intermediate results of the dynamic programming algorithm.

So, both the space and time complexity are quadratic in terms of the length of the input string.