664. Strange Printer

Hard String Dynamic Programming

Leetcode Link

Problem Description

Imagine you have a printer, but it's quite peculiar because of two constraints:

1. It can only print a string of identical characters in one go.
2. You can print over already printed characters.

Now, with this printer, you're given a specific string `s`. The question asks us to find out the smallest number of turns or print operations to get the entire string `s` printed, considering the printer's constraints.

A "turn" or "print operation" is defined by the printer printing a sequence of the same character. So each time, you can choose a segment of your string and print a character across that segment, potentially covering up different characters that were there from previous turns.

For example, if your string "s" is "aaabbb", a possible print strategy could be:

- In the first turn, print "aaa" from position 1 to 3.
- In the second turn, print "bbb" from position 4 to 6. So you would need 2 turns to print "aaabbb".

The crux of the problem is to plan your print operations strategically to minimize the number of turns needed.

Intuition

To solve this problem, a dynamic programming (DP) approach is used. The intuition behind DP is to solve small problems and use their solutions to solve larger problems.

Firstly, we start by understanding that if the first and last characters to be printed are the same, we can deal with them in the same turn. If they aren't the same, we should consider splitting the problem into smaller segments and solving for those.

To implement this, we define a 2D array `f`, where `f[i][j]` represents the minimum number of operations needed to print the substring s[i..j].

The solution relies on a few key insights:

1. If s[i] == s[j] for some substring s[i..j], printing s[i] can include s[j] for free, reducing the number of operations.
2. When s[i] != s[j], we need to consider all possible places we can split the substring into two parts - s[i..k] and s[k+1..j]. We're looking for the best place to split where the sum of operations for both substrings is minimized.
3. Since our solution to each substring depends on solutions to smaller substrings, we need to iterate from the end of the string to the beginning. This way, we ensure that the solution to any used substring is already computed.

So through a nested loop iterating over all possible substrings, and a third loop to handle splits within these substrings, we apply these insights and fill out the array `f` using the dynamic relationship. The final answer will be in `f[0][n-1]`, reflecting the minimum operations needed for the whole string `s`.

Solution Approach

The implementation of the solution can be broken down as follows, based on the provided Reference Solution Approach:

1. **Data Structures**: We use a 2D array `f` of size `n x n`, where `n` is the length of the input string `s`. This array holds the sub-results for the number of operations required to print substrings of `s`. The default value for all positions in `f` is set to infinity (`inf`) because initially, we do not know the minimum operations to print any substring.

2. **Initializing the 2D Array**: We loop through the array `f` and initialize `f[i][i]` to `1` for all `i`, because printing a single character always requires exactly one operation.

3. **Dynamic Programming**: The core idea of dynamic programming is utilized here, where we solve for small substrings and use these solutions to solve for larger substrings. We iterate over the starting index `i` of the substring in reverse (from `n-1` to `0`) and the ending index `j` in the normal order (from `i+1` to `n-1`).

4. **Transitions and Recurrence Relation**: The transition equations, as per the dynamic programming approach, are the cornerstone of this implementation.

   - If `s[i]` equals `s[j]`, we leverage the insight that we can print the last character `s[j]` for 'free' when we print `s[i]`, effectively ignoring `s[j]`. Therefore, `f[i][j] = f[i][j - 1]`.
   - If `s[i]` is not equal to `s[j]`, we consider all possibilities for splitting the substring into two parts. We iterate over every possible split point `k` between `i` and `j-1`. Our goal is to minimize the total print operations for the two substrings, so we calculate `f[i][j]` as the minimum of its current value and `f[i][k] + f[k+1][j]`.

5. **Answer Retrieval**: After all iterations and updates to the 2D array `f`, the answer to the problem, which is the minimum number of operations needed to print the entire string, is stored in `f[0][n-1]`.

In summary, the approach translates the problem into finding the minimal steps for progressively larger substrings using principles of dynamic programming. The solution carefully considers the two properties of the printer, especially the fact that a sequence of the same character can be used advantageously to reduce operation counts.

Example Walkthrough

Let's use the string "abba" to illustrate the solution approach.

1. **Data Structures**: We create a 2D array `f` of size `4 x 4` since the string "abba" has 4 characters. We initialize all values to infinity except for `f[i][i]`, which we set to `1` since a single character can always be printed in one operation.

   Initial `f` array:

   ```
   1  f[0][0] = 1    f[0][1] = inf    f[0][2] = inf    f[0][3] = inf
   2  f[1][0] = ∞    f[1][1] = 1      f[1][2] = inf    f[1][3] = inf
   3  f[2][0] = ∞    f[2][1] = ∞      f[2][2] = 1      f[2][3] = inf
   4  f[3][0] = ∞    f[3][1] = ∞      f[3][2] = ∞      f[3][3] = 1
   ```

2. **Initializing the 2D Array**: We've already initialized `f[i][i]` to `1`.

3. **Dynamic Programming**: We consider all substrings starting from length 2 to the full string length. For "abba", we consider "ab", "bb", and "ba" for lengths 2, and "abb" and "bba" for length 3, and finally "abba" for length 4.

4. **Transitions and Recurrence Relation**:

   a. For "ab" (`s[0] != s[1]`), we split at every point (though there's only one in this case), and find that `f[0][1]` should be `f[0][0] + f[1][1]` which evaluates to `2`.

   b. For "bb" (`s[1] == s[2]`), the characters are the same, so `f[1][2]` is just `f[1][1]` which is `1`.

   c. For "ba" (`s[1] != s[2]`), similar to "ab", `f[2][3]` should be `f[2][2] + f[3][3]` giving us `2`.

   d. Now we look at "abb" and "bba":

   - For "abb" (`s[0] != s[2]`), we check both splits "a | bb" and "ab | b" and we find the minimum of `f[0][0] + f[1][2]` and `f[0][1] + f[2][2]`, which will be the minimum of `2` and `3`, so `f[0][2]` becomes `2`.
   - For "bba" (`s[1] != s[3]`), we do a similar check and find `f[1][3]` to also be `2`.

   e. Finally, for "abba" (`s[0] == s[3]`), we either consider it as "a | bba" or "abb | a", or we take advantage of the fact that the first and last characters are the same. So we just need the result of "abb", which is `2`.

   Updated `f` array:

   ```
   1  f[0][0] = 1    f[0][1] = 2    f[0][2] = 2    f[0][3] = 2
   2  f[1][0] = ∞    f[1][1] = 1    f[1][2] = 1    f[1][3] = 2
   3  f[2][0] = ∞    f[2][1] = ∞    f[2][2] = 1    f[2][3] = 2
   4  f[3][0] = ∞    f[3][1] = ∞    f[3][2] = ∞    f[3][3] = 1
   ```

5. **Answer Retrieval**: The minimum number of operations needed to print the entire string "abba" is in `f[0][3]`, which is `2`.

As the example shows, the DP approach systematically reduces the complexity by utilizing the relationship between substrings, allowing us to find the minimum number of print operations needed.

Python Solution

```python
class Solution:
    def strangePrinter(self, string: str) -> int:
        # Length of the input string
        length = len(string)
        # Initialize the operations matrix with infinity, representing the minimum number of turns
        operations = [[float("inf") * length for _ in range(length)]

        # Start from the end of the string and move towards the beginning
        for start in range(length - 1, -1, -1):
            # A single character takes one turn to print
            operations[start][start] = 1

            # Fill the operations matrix for substrings [start:end+1]
            for end in range(start + 1, length):
                # If the characters at the start and end are the same,
                # it takes the same number of turns as [start:end-1]
                if string[start] == string[end]:
                    operations[start][end] = operations[start][end - 1]
                else:
                    # If they are different, find the minimum number of turns needed
                    # to print the substring by splitting it at different points (k)
                    for split_point in range(start, end):
                        # The number of turns is the sum of printing both substrings separately
                        operations[start][end] = min(
                            operations[start][end],
                            operations[start][split_point] + operations[split_point + 1][end]
                        )

        # Return the minimum number of turns to print the whole string
        return operations[0][-1]
```

Java Solution

```java
class Solution {
    public int strangePrinter(String s) {
        final int INFINITY = Integer.MAX_VALUE / 2; // value representing infinity, halved to avoid overflow when adding
        int length = s.length(); // length of the input string
        int[][] dp = new int[length][length]; // initialize DP table, dp[i][j] represents the minimal turns to print s[i..j]

        for (int[] row : dp) {
            Arrays.fill(row, INFINITY); // fill the DP table with 'infinity' as initial value
        }

        for (int i = length - 1; i >= 0; --i) { // iterate from the end to the beginning of the string
            dp[i][i] = 1; // it takes 1 turn to print a single character

            for (int j = i + 1; j < length; ++j) { // iterate over the rest of the string
                if (s.charAt(i) == s.charAt(j)) { // if characters match, it costs the same as without the last character
                    dp[i][j] = dp[i][j - 1];
                } else {
                    for (int k = i; k < j; ++k) { // split the range into two parts and sum their costs
                        dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k + 1][j]); // find the minimum cost
                    }
                }
            }
        }

        return dp[0][length - 1]; // answer is the minimal turns to print the entire string s[0..length-1]
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int strangePrinter(string s) {
        int length = s.size();
        // Create a 2D array to store the minimum number of turns needed
        // Initialize all entries with a high value (0x3f3f3f3f) to signify uncalculated/infinity
        int dp[length][length];
        memset(dp, 0x3f, sizeof(dp));

        // Loop backwards through the string (start from the end)
        for (int start = length - 1; start >= 0; --start) {
            // Always takes one turn to print a single character
            dp[start][start] = 1;

            // Iterate over the substring from the current start to the end of the string
            for (int end = start + 1; end < length; ++end) {
                // If the characters are the same, it takes the same number of turns
                // as it took to print the substring without the current character
                // because it can be printed in the same turn as the last character
                if (s[start] == s[end]) {
                    dp[start][end] = dp[start][end - 1];
                } else {
                    // If characters are different, find the minimum number of turns
                    // needed by splitting the substring into two parts at every possible position
                    for (int split = start; split < end; ++split) {
                        dp[start][end] = min(dp[start][end], dp[start][split] + dp[split + 1][end]);
                    }
                }
            }
        }

        // The answer is the minimum number of turns to print the whole string
        // which is stored in dp[0][length - 1]
        return dp[0][length - 1];
    }
};
```

Typescript Solution

```typescript
function strangePrinter(s: string): number {
    const length = s.length;
    // Initialize a 2D array to store the minimum number of turns for s[i..j] substring.
    const dp: number[][] = new Array(length).fill(0).map(() => new Array(length).fill(Infinity));

    // Traverse the string in reverse to deal with substrings in a bottom-up manner.
    for (let i = length - 1; i >= 0; --i) {
        // A single character needs one turn to be printed.
        dp[i][i] = 1;

        // Compute the number of turns needed for substrings [i..j]
        for (let j = i + 1; j < length; ++j) {
            // If the characters at position i and j are the same,
            // no extra turn is needed beyond what is required for [i..j-1]
            if (s[i] === s[j]) {
                dp[i][j] = dp[i][j - 1];
            } else {
                // If characters are different, find the split point 'k'
                // that minimizes the sum of turns for [i..k] and [k+1..j] substrings.
                for (let k = i; k < j; ++k) {
                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k + 1][j]);
                }
            }
        }
    }

    // The result for the entire string is then stored in dp[0][length - 1].
    return dp[0][length - 1];
}
```

Time and Space Complexity

The time complexity of the code is `O(n^3)` due to the triple nested loop structure where `n` is the length of string `s`. The outer loop runs `n` times in reverse, the middle loop runs a maximum of `n` times for each value of `i`, and the inner loop, also running up to `n` times for each combination of `i` and `j`, represents the main source of the cubic time complexity. Inside the inner loop, the code checks every substring from `i` to `j` and calculates the minimum operations required by comparing the current value of `f[i][j]` with `f[i][k] + f[k + 1][j]`, where `k` is between `i` and `j-1`.

The space complexity of the code is `O(n^2)` which is caused by the storage requirements of the 2D array `f` that has a size of `n * n`, necessary for dynamic programming and storing the minimum print operations for every substring from `i` to `j`.