2027. Minimum Moves to Convert String

Easy Greedy String

Leetcode Link

Problem Description

You are given a string `s` which is made up of characters that are either `'X'` or `'O'`. Your goal is to convert all the characters in the string to `'O'` using a specific operation. The operation allows you to select *three consecutive characters* in the string and change them all to `'O'`. If any of those characters are already `'O'`, they remain `'O'`. Your task is to find out the minimum number of such operations required to change every character in the string to `'O'`.

Intuition

The solution requires a greedy approach. Start scanning the string from the beginning, and whenever you find an `'X'`, that means you need to perform the operation because it's the leftmost `'X'` that can be converted along with the next two characters. After performing the operation on three characters, you can safely move three positions ahead because those positions are now guaranteed to be `'O'`.

This approach reliably finds the minimum number of moves because by always taking the earliest possible operation, you ensure that no move is wasted. If an `'O'` is encountered, simply move one position to the right since this position doesn't necessitate the operation. Continue the process until the end of the string. The number of moves made gives the desired result.

Solution Approach

The solution is implemented in a straightforward manner without requiring complex data structures or patterns. Here is a detailed walkthrough of the implementation:

- We initialize two variables, `ans` which will hold the number of moves required, initially set to `0` and `i` which is our pointer to iterate through the string, initially set to `0`.
- We use a `while` loop to iterate over the string until `i` reaches the end of the string (`i < len(s)`).
- Inside the loop, we check each character of the string. If the current character `s[i]` is `'X'`, then a move is required. We increment the `ans` variable by one to count this move.
- Since we can change three consecutive characters with each move, we increment `i` by 3 after a move is applied (`i += 3`). This allows us to skip the next two characters because they have been turned into `'O'` by the operation, or they were already `'O'`.
- If the current character `s[i]` is not `'X'` (it's `'O'`), then no move is required for these positions. Thus, we move the pointer `i` by one to check the next character (`i += 1`).
- The loop continues until all characters have been checked.
- Finally, the `ans` variable, which has been accumulating the number of moves required, is returned.

So the important points in the algorithm are:

- The use of a greedy approach, always taking the earliest opportunity to perform an operation.
- The increment of the pointer based on whether a move was made or not, moving by 3 if a move was performed or by 1 otherwise.

This method ensures that the solution is efficient, with a time complexity of O(n), where n is the length of the string, since each character in the string is considered at most once.

Example Walkthrough

Let's say we have the string `s = "XXOXOOXOXOOX"`.

1. Starting from the left, we encounter `'X'` at position 0. We can perform an operation and change `s[0:3]` from `'XXO'` to `'OOO'`. We increment our answer to `1` and skip 3 characters to index 3.
2. At position 3, the character is `'O'`, so no operation is needed. We increment the index by 1 to move to position 4.
3. Position 4 is `'X'`, so we perform an operation on `s[4:7]`, changing `'OOX'` to `'OOO'`. We increment our answer to `2` and skip 3 characters to index 7.
4. At position 7, we find another `'X'`. We perform the operation on `s[7:10]`, changing `'XOX'` to `'OOO'`. Increment our answer to `3` and skip 3 characters to index 10.
5. At position 10, there is an `'X'`. Since this is near the end and no more sets of three are available, we perform our final operation on `s[10:13]` (which is actually just up to `s[11]` since we've reached the end), changing `'OX'` to `'OO'`. We increment our answer to `4`.

After these steps, the string `s` is now `'OOOOOOOOOOOO'`, and we used a total of `4` operations. Hence, the minimum number of operations required to change every 'X' to 'O' in `s` is `4`.

Python Solution

```python
class Solution:
    def minimumMoves(self, grid: str) -> int:
        # Initialize the number of moves to 0
        moves_count = 0
        # Initialize the position index to 0
        index = 0

        # Loop through the grid until the end is reached
        while index < len(grid):
            # Check if the current character is an 'X'
            if grid[index] == "X":
                # If 'X' is found, increase the moves count by 1 since we can flip three consecutive cells
                moves_count += 1
                # Move the index 3 cells forward as these cells will be flipped
                index += 3
            else:
                # If the current character is not an 'X', just move to the next cell
                index += 1

        # Return the total number of moves needed
        return moves_count

```

Java Solution

```java
class Solution {
    public int minimumMoves(String s) {
        int moves = 0; // Initialize the count of moves to 0

        // Loop over the string, increment the loop counter accordingly
        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == 'X') {
                moves++;  // If the current character is 'X', increment the move count
                i += 2;   // Skip the next two characters since one move can change up to 3 consecutive characters
            }
        }

        return moves; // Return the total number of moves required
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the minimum number of moves required to convert the given string into 'OOO...OOO' (no 'X's)
    int minimumMoves(string s) {
        int moveCount = 0; // Variable to store the count of moves

        // Loop through each character in the string
        for (int i = 0; i < s.size(); ++i) {

            // If the current character is 'X', we need to perform a move
            if (s[i] == 'X') {
                ++moveCount;  // Increment the move count
                i += 2; // Skip the next two characters because a move changes three consecutive characters
            }
        }

        // Return the total number of moves required to change all 'X's to 'O's
        return moveCount;
    }
};

```

Typescript Solution

```typescript
/**
 * Determines the minimum number of moves required to convert the string
 * so that there are no 'X' characters. Each move can cover up to three consecutive characters.
 * @param {string} sequence - The string consisting of '.' and 'X' characters.
 * @return {number} - The minimum number of moves required.
 */
function minimumMoves(sequence: string): number {
    // The length of the string
    const sequenceLength = sequence.length;

    // Variable to store the minimum moves
    let minimumMoves = 0;

    // Index variable to iterate over the string
    let index = 0;

    // Loop through the string
    while (index < sequenceLength) {
        // If an 'X' is found, a move is required
        if (sequence[index] === 'X') {
            minimumMoves++; // Increment the move counter
            index += 3; // Skip the next two positions as they are also covered by the move
        } else {
            index++; // Move to the next character if current is not 'X'
        }
    }

    // Return the computed minimum moves
    return minimumMoves;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the string `s`. The while loop iterates through each character of the string at most once. In the worst case, the loop makes `n` iterations when there are no "X" characters found. In the best case, if "X" characters are found, it skips 3 characters at a time due to `i += 3`. However, regardless of the pattern of "X" characters within the string, every character is examined no more than once, thereby maintaining a linear time complexity relative to the string length.

Space Complexity

The space complexity of the code is `O(1)`. The function maintains a fixed number of variables (`ans` and `i`) that do not scale with the input size. The space used does not depend on the length of the string, therefore it is constant, and no additional data structures are used that would increase the memory usage. Thus, the additional memory required remains constant regardless of the input.