2745. Construct the Longest New String

Medium Greedy Brainteaser Math

Problem Description

You are tasked with creating the longest string possible by concatenating strings of types "AA", "BB", and "AB". You have `x` strings of type "AA", `y` strings of type "BB", and `z` strings of type "AB". While concatenating these strings to make the longest one possible, there is a constraint that the final string cannot contain the substrings "AAA" or "BBB". A substring is a sequence of characters that are adjacent in the string and not interrupted by others.

The goal is to figure out the combination and order of concatenation of the given strings so that the resulting string is as long as it can be without breaking the rules.

Intuition

The solution seems to calculate the maximum length of the string based on the values of `x`, `y`, and `z`. We need to avoid having three consecutive 'A's or 'B's, which means we have to alternate between "AA" and "BB" as much as possible. "AB" strings can help with the alternation, but they are not as useful as "AA" or "BB" because they only provide one 'A' or 'B', not two.

Here's the logic behind the solution:

- If `x < y`, it means that there are extra "BB" strings. Since we want to avoid three consecutive 'B's, we can only use a certain number of "BB" strings equal to the number of "AA" strings plus the number of "AB" strings to alternate effectively. This way, we place "AA" and "AB" strings between each "BB" to avoid the "BBB" sequence. So, `(x * 2 + z + 1) * 2` represents the maximum length considering this alternation plus one "B" from an extra "BB" that can be placed at the end or the beginning.
- Similarly, if `x > y`, then we have more "AA" strings. The logic is the same as above by switching the roles of "AA" and "BB". We use all "BB" and as many "AA" as we can without violating the rule concerning "AAA".
- If `x == y`, the situation is balanced. We can alternate "AA" and "BB" strings freely until we run out of them. The "AB" strings are used to perfectly intersperse between "AA" and "BB". The maximum length is then the sum of all strings times two because each "AA" and "BB" contributes two 'A's or 'B's.

In all these scenarios, we maximize the length of the string by strategically placing the "AB" strings to ensure that we don't create a forbidden sequence. The given solution could be incorrect and might not work for all test cases because it does not handle the conditions correctly; it is performing the same calculation for all three conditions when they should be treated differently based on the relative values of `x`, `y`, and `z`.

Solution Approach

The solution provided seems to follow direct mathematical calculations to determine the maximum length of the string that can be created. However, the implementation does not seem to align correctly with a valid approach for this specific problem. To accurately explain the algorithm, we need to consider a valid solution approach that would take into account the relationship between `x`, `y`, and `z` to properly place "AA", "BB", and "AB" strings without creating forbidden sequences such as "AAA" or "BBB".

A correct approach would involve:

1. Priority placement of "AA" and "BB" strings to ensure the alternation required to avoid three consecutive identical characters.
2. Utilization of "AB" strings, as they can serve as a buffer between "AA" and "BB" strings when the counts of "AA" and "BB" are almost equal or when a strict alternation is required.
3. A careful count to ensure that once either type of string runs out ("AA" or "BB"), the remaining types are placed in a way that they don't cause a forbidden sequence to appear.

While the intention of the provided code seems to attempt to address the above rules, the actual return statements seem incorrect. For example, `return (x * 2 + z + 1) * 2` asserts that simply doubling the sum of twice the count of "AA" strings, plus the "AB" strings, plus an extra 1 (for assumed additional placement of a "B"), will yield the correct maximum length, which doesn't take into account the actual alternation and placement strategy.

Similarly, the other return statements don't align with a correct implementation of these rules.

For a valid algorithm, one would typically perform the following steps:

1. Determine the possible pairs of "AA" and "BB" that can be alternatively placed (min(x, y)).
2. Place all available "AB" strings between "AA" and "BB" strings as needed.
3. Intersperse remaining "AA" or "BB" strings while respecting the substring constraint.
4. Calculate the total length based on actually placed strings.

Data structures such as queues could potentially be used to track available strings and easily manage their placement. Patterns such as greedy algorithms, where we first use up the strings that are in surplus, could also be central in a correct implementation of the algorithm.

Since the code provided as the "Reference Solution Approach" is empty, it is not possible to walk through the provided solution with the guarantees that it is correct. Instead, the above approach outlines a more reasoned and theoretically correct algorithm to tackle the problem described.

Example Walkthrough

Let's consider an example where `x = 3` (so we have "AA", "AA", "AA"), `y = 2` (thus "BB", "BB"), and `z = 1` (meaning one "AB"). Remember, we have to maximize the length of the concatenated string while avoiding "AAA" or "BBB".

Following the correct solution approach:

1. **Placement of "AA" and "BB" strings**: There are fewer "BB" strings than "AA" strings, so we start by placing "BB" strings and alternating them with "AA" strings. We have min(x, y) pairs we can alternate, which is the smaller of `x` or `y`. Here, it is 2.

   So we start with: "AABB" "AABB"

2. **Utilize the "AB" string**: We have one "AB" string, which can be placed either between "AA" and "BB" or at the beginning or end of the string. We choose to place it between "AA" and "BB" to maximize the alternation:

   "AABABBAABB"

3. **Placement of Remaining "AA" strings**: After placing "AB", we still have one "AA" left. We can't place it directly after a "AA", so it should be placed either at the beginning or end of the string. We decide to place it at the end:

   "AABABBAABBAA"

4. **Calculate the Total Length**: We sum up the lengths of the individual strings: each "AA" and "BB" contribute two characters, and "AB" contributes two as well, giving us a total length.

   Total Length = (3 "AA" * 2) + (2 "BB" * 2) + (1 "AB" * 2) = 6 + 8 + 2 = 16.

In the final string "AABABBAABBAA", we see that we have used all the strings we have, and there are no occurrences of "AAA" or "BBB". The final string has a length of 16 characters, which is the maximum length that can be achieved with the given "AA", "BB", and "AB" without breaking the rules.

Solution Implementation

Python

```python
class Solution:
    def longest_string(self, a: int, b: int, c: int) -> int:
        # If 'a' is less than 'b', calculate the longest string length based on 'a':
        # The resulting string will be of the pattern 'ab' repeated 'a' times, followed by 'c' single characters,
        # and possibly an additional 'a' character, double the sum to account for symmetry.
        if a < b:
            return (a * 2 + c + 1) * 2

        # If 'a' is greater than 'b', calculate the longest string length based on 'b':
        # The resulting string will be of the pattern 'ab' repeated 'b' times, followed by 'c' single characters,
        # and possibly an additional 'b' character, double the sum to account for symmetry.
        if a > b:
            return (b * 2 + c + 1) * 2

        # If 'a' equals 'b', use their sum together with 'c' and double it for the entire string length.
        # This will be the maximum length of the string with equal numbers of 'a's and 'b's, with 'c'
        # characters potentially interspersed within.
        return (a + b + c) * 2

# Using the class
# Instantiate the Solution class
solution = Solution()

# Example usage:
result = solution.longest_string(1, 2, 3)
print(result)  # This should output the longest string length based on the inputs (1, 2, 3)
```

Java

```java
class Solution {
    // Method to calculate the longest string length
    public int longestString(int x, int y, int z) {
        // If x is less than y
        if (x < y) {
            // Multiply x by 2, add z and 1 to it, and then multiply the whole by 2
            return (x * 2 + z + 1) * 2;
        }
        // If x is greater than y
        if (x > y) {
            // Multiply y by 2, add z and 1 to it, and then multiply the whole by 2
            return (y * 2 + z + 1) * 2;
        }
        // If x is equal to y
        // Add x, y, and z together and then multiply the whole by 2
        return (x + y + z) * 2;
    }
}
```

C++

```cpp
#include <algorithm> // Included for std::min and std::max

class Solution {
public:
    // Function to calculate the longest possible string length
    // given three integers x, y, and z.
    int longestString(int x, int y, int z) {
        // If x is strictly less than y, we perform a specific calculation
        if (x < y) {
            return (x * 2 + z + 1) * 2;
        }

        // If x is strictly greater than y, another calculation is performed
        if (x > y) {
            return (y * 2 + z + 1) * 2;
        }

        // In the case where x equals y, we use all of the components
        // x, y, and z to calculate the string length
        return (x + y + z) * 2;
    }
};
```

TypeScript

```typescript
/**
 * Calculate the length of the longest string based on input conditions.
 *
 * @param x - The first number value
 * @param y - The second number value
 * @param z - The third number value
 * @returns The calculated number representing the longest string length
 */
function longestString(x: number, y: number, z: number): number {
    // If x is strictly less than y
    if (x < y) {
        // Return the lenath based on the formula ((x * 2) + z + 1) * 2
        return (x * 2 + z + 1) * 2;
    }
    // If x is strictly greater than y
    if (x > y) {
        // Return the lenath based on the formula ((y * 2) + z + 1) * 2
        return (y * 2 + z + 1) * 2;
    }
    // If x is equal to y, return the length based on summing all and multiplying by 2
    return (x + y + z) * 2;
}


class Solution:
    def longest_string(self, a: int, b: int, c: int) -> int:
        # If 'a' is less than 'b', calculate the longest string length based on 'a':
        # The resulting string will be of the pattern 'ab' repeated 'a' times, followed by 'c' single characters,
        # and possibly an additional 'a' character, double the sum to account for symmetry.
        if a < b:
            return (a * 2 + c + 1) * 2

        # If 'a' is greater than 'b', calculate the longest string length based on 'b':
        # The resulting string will be of the pattern 'ab' repeated 'b' times, followed by 'c' single characters,
        # and possibly an additional 'b' character, double the sum to account for symmetry.
        if a > b:
            return (b * 2 + c + 1) * 2

        # If 'a' equals 'b', use their sum together with 'c' and double it for the entire string length.
        # This will be the maximum length of the string with equal numbers of 'a's and 'b's, with 'c'
        # characters potentially interspersed within.
        return (a + b + c) * 2

# Using the class
# Instantiate the Solution class
solution = Solution()

# Example usage:
result = solution.longest_string(1, 2, 3)
print(result)  # This should output the longest string length based on the inputs (1, 2, 3)
```

Time and Space Complexity

Time Complexity

The time complexity of the function `longestString` is `O(1)`. This is because the function consists only of simple arithmetic operations (multiplication, addition, and comparisons), which are executed a constant number of times regardless of the input values of `x`, `y`, and `z`. Therefore, the execution time is independent of the input sizes and is considered constant time complexity.

Space Complexity

The space complexity of the function `longestString` is also `O(1)`. The function uses a fixed amount of space for the input parameters `x`, `y`, and `z`, as well as a few additional variables for the arithmetic operations and the return value. As the amount of space used does not scale with the size of the inputs, the space complexity is constant.