1663. Smallest String With A Given Numeric Value

Problem Description

The problem presents us with a unique challenge of constructing the lexicographically smallest string of a specified length, `n`, where each lowercase letter of the English alphabet carries a numeric value equivalent to its positional order (1-indexed). We need to ensure that the sum of the numeric values of the characters in the string equals a given integer, `k`. This translates to building the smallest possible string in dictionary order, which means if you were to list all possible strings that meet the criteria, this string would be the first one.

To illustrate, let's take an example: if given `n = 3` and `k = 27`, and considering that the numeric values are `{'a': 1, 'b': 2, 'c': 3, ..., 'z': 26}`, one way to achieve this is by constructing the string "aaa" which sums up to 3 (1+1+1). However, this does not sum up to 27. We would need to adjust the string, maybe to "zab" which sums up to 27 (26+1+1), but it's not the lexicographically smallest. The smallest string that sums up to 27 would be "aaz" (1+1+26).

Intuition

The solution hinges on the fact that we want the lexicographically smallest string. This means we should use 'a's as much as possible, as 'a' comes first in the alphabet. We start by initializing the result string with `n` 'a's, since 'a' has the smallest possible value (1). However, this string would only sum up to `n`, so we need to increase the sum to match `k`.

We start from the end of this string and substitute 'a's with 'z's or other letter accordingly, to increase the total value of the string to `k`. We use 'z's because they have the highest possible value (26), and replacing from the end ensures we maintain the lexicographic order—it's always better to have higher-valued characters towards the end. We subtract the value of 'z' (or the necessary character) from the remaining total required (`k - n` initially, as we already have a sum of `n` using 'a's) until it's no longer possible to place a 'z' (when the remaining value is less than 26).

When we reach a point where we can't place a 'z' (because the remaining value is less than 26), we place the highest possible character that we can at that point by converting the remaining value to a character (using the `chr` function and adding the ordinal value of 'a' minus 1), and place it in the correct position. After this step, since we have met the 'k' sum condition, we have our answer. The while loop ensures we are placing 'z's until we cannot, and the final adjustment is done once outside the loop.

In summary, the problem is a matter of allocating values in a way that keeps the string as 'small' as possible, by starting with the smallest value ('a') and only using higher value characters ('z' and others) when necessary, and doing so as far down the string as possible.

Solution Approach

The implementation of the solution provided can be broken down into the following steps:

1. **Initialization**: We initialize an array `ans` with `n` 'a's, because 'a' has the numeric value of 1, and hence, this gives us the base case where the string is filled with the smallest possible character lexicographically. The total initial numeric value of this string would be `n`.
2. **Deficit Calculation**: We calculate the difference `d` between the numeric value `k` our string must sum up to, and the numeric value our string currently has (`n` 'a's, thus `n`). `d` represents the total additional value we need to distribute across the string.
3. **String Construction**: We then iterate backwards through the string (starting at the last character) and make adjustments by replacing 'a's with 'z's where possible. Each 'z' is worth 26, so for each 'z' we add, we subtract 25 from `d` (since we are replacing an 'a' that's already been counted as 1) to reflect this addition.
4. **Optimal Character Placement**: We keep adding 'z's until we reach a point where `d` is no longer greater than 25 (meaning, we can't add a 'z' without exceeding the required sum `k`). At that point, we need to determine the precise character that will bring us to exactly `k`. This is done by taking the current character 'a' (which is already included in the sum as 1), and adding `d` to its numeric value (`ord('a') - 1`, because 'a' is 1 and not 0). We then convert this numeric value back to a character using `chr()` and place it in the string.
5. **Result Construction**: After completing the iteration through the string and replacing the necessary 'a's with 'z's and the final optimal character, we consolidate the result by joining the array into a string with `''.join(ans)`, which gives us the lexicographically smallest string with a numeric value equal to `k`.

This solution utilizes a **greedy algorithm**, a typical strategy for optimization problems, where the best solution is built piece by piece in a series of steps that choose the locally optimal choice at each step with the hope this will lead to a globally optimal solution.

The python code provided translates directly from the logical steps described above and uses basic data structures like an array (represented as a list in Python) to build the string character by character.

```
class Solution:
    def getSmallestString(self, n: int, k: int) -> str:
        ans = ['a'] * n
        k, d = k - n, k - n
        while d > 25:
            ans[i] = 'z'
            d -= 25
            i -= 1
        ans[i] = chr(ord(ans[i]) + d)
        return ''.join(ans)
```

Each step in the solution is designed to use the minimum value character ('a') and maximize its occurrence. Only when it's necessary to reach the sum `k`, we use the value of other characters, and locating these as far down the string as possible ensures lexicographical ordering.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `n = 5` and `k = 73`. Our goal is to construct the lexicographically smallest string such that it is of length 5 and the sum of the numeric values of the characters equals 73.

Step 1: Initialization

We initialize our string with `n` 'a's. Here, `ans = ['a', 'a', 'a', 'a', 'a']`. The total value is currently 5 (`n`).

Step 2: Deficit Calculation

We calculate the deficit `d` by subtracting the current sum (5) from `k` (73). So `d = 73 - 5 = 68`.

Step 3: String Construction

We start from the end of the array and work backward, substituting 'a's with 'z's where we can without exceeding the sum of 73.

The first 'a' from the end can be replaced by 'z' because `d` is 68 and replacing 'a' with 'z' gives us a value of 25 more than the 'a', which would still be less than or equal to 68, `ans = ['a', 'a', 'a', 'a', 'z']`. We reduce `d` by 25, as we added 25 more to the sum. Now `d = 68 - 25 = 43`.

Once again, the second 'a' from the end can also be replaced by 'z', `ans = ['a', 'a', 'a', 'z', 'z']`. Again, reduce `d` by 25, `d = 43 - 25 = 18`.

We continue this process for another 'a', `ans = ['a', 'a', 'z', 'z', 'z']` and reduce `d` by 25, `d = 18 - 25 = -7`. However, since `d` has gone below 0, this means we overshot the sum of 73. So, we need to revert the last addition and figure out the correct character that gives us exactly the sum we need.

We're replacing the third 'z' with 'a' again and set `d` back to 18, because we had reduced it too much, `ans = ['a', 'a', 'a', 'z', 'z']`, `d = 18`.

Step 4: Optimal Character Placement

Since `d` is now less than 25, we look for the right character to replace the third 'a' from the end such that the sum is exactly 73. We add `d` to the value of 'a' (which equals 1). So, the character to place is `chr(ord('a') - 1 + 18) = chr(97 - 1 + 18) = chr(114) = 'r'`. Our string now looks like `ans = ['a', 'a', 'r', 'z', 'z']`.

Step 5: Result Construction

We combine the characters to form the final string. The resulted lexicographically smallest string is `''.join(['a', 'a', 'r', 'z', 'z'])`, which equals `"aarzz"`.

In this small example, the string `"aarzz"` is the lexicographically smallest string of length 5 where the sum of the numeric values of the characters equals 73. It demonstrates the process of filling up the string with 'a's and replacing them from the end forward with 'z's until we close in on the target sum `k`. When we can no longer add 'z's without overshooting `k`, we find the highest-valued character (`'r'` in this case), which brings us exactly to `k`. This example demonstrates the effectiveness of the greedy approach for this specific problem.

Python Solution

```
class Solution:
    def getSmallestString(self, length: int, numeric_value: int) -> str:
        # Initialize the string with 'a' repeated 'length' times
        smallest_string = ['a'] * length

        # Starting from the last position and calculating the remaining value by subtracting 'length' from 'numeric_value'
        index = length - 1
        remaining_value = numeric_value - length

        # As long as the remaining_value is greater than 25, place 'z' at the current index
        while remaining_value > 25:
            smallest_string[index] = 'z'
            remaining_value -= 25  # Decrease remaining_value by 25 for each 'z' added
            index -= 1  # Move to the previous position

        # Once the remaining_value is <= 25, place a character that bridges the gap to the target numeric_value
        smallest_string[index] = chr(ord('a') + remaining_value)

        # Join all characters in the list to form the resulting string and return it
        return ''.join(smallest_string)

# Example Usage:
# Create an instance of the Solution class
solution_instance = Solution()
# Call getSmallestString() method with the specified 'length' and 'numeric_value'
result = solution_instance.getSmallestString(5, 73)
# Output the result
print(result)  # Output should be 'aaszz'
```

Java Solution

```
class Solution {

    // Function to find the lexicographically smallest string with a given length n and numeric value k
    public String getSmallestString(int n, int k) {
        // Initialize a character array to hold the answer
        char[] smallestString = new char[n];

        // Fill the array with the letter 'a', which is the smallest character
        Arrays.fill(smallestString, 'a');

        // Pointer to track the position in the array from the end
        int index = n - 1;

        // Compute the total value we need to distribute among the characters
        int valueToDistribute = k - n;

        // Loop to assign the character 'z' as long as the value to distribute is more than 25,
        // because 'z' represents the value 26 (1 for 'a' + 25)
        while (valueToDistribute > 25) {
            smallestString[index--] = 'z'; // Assign 'z' to current position
            valueToDistribute -= 25; // Decrease the value to distribute by the value of 'z'
        }

        // Assign the last character that uses the remaining valueToDistribute
        // We add the valueToDistribute to 'a' to get the correct character
        smallestString[index] = (char) ('a' + valueToDistribute);

        // Convert the character array to a string and return it
        return String.valueOf(smallestString);
    }
}
```

C++ Solution

```
class Solution {
public:
    // Function constructs the lexicographically smallest string with length n
    // and numeric value k.
    string getSmallestString(int n, int k) {
        // Initialize the answer string with all 'a's since 'a' has the smallest
        // lexicographical value.
        string answer(n, 'a');

        // Start from the end of the string and work backwards.
        int index = n - 1;

        // Calculate the numeric value required to achieve exactly k
        // given that all 'a's have already contributed a value equal to n.
        int numericValueNeeded = k - n;

        // Fill the answer string from the end with 'z's (each 'z' has a numeric value of 26)
        // until the required numeric value is less than or equal to the value that can be represented
        // by a single character (i.e., 25 or less since we already have 'a' included).
        while (numericValueNeeded > 25) {
            answer[index--] = 'z'; // Replace 'a' with 'z' at the current index.
            numericValueNeeded -= 25; // Subtract 25 from the remaining value needed as we replaced 'a' with 'z'.
        }

        // There can be at most one character that is not 'a' or 'z' to perfectly reach the required value k.
        // Adjust the character at the current index by increasing its value to match the
        // remaining numericValueNeeded so that the total numeric value of the string equals k.
        answer[index] = String.fromCharCode(answer[index].charCodeAt(0) + numericValueNeeded);

        // Join the array of characters back into a string and return the result.
        return answer.join("");
    }
};
```

Typescript Solution

```
function getSmallestString(n: number, k: number): string {
    // Initialize the answer string as an array of 'a' characters since 'a' has the smallest
    // lexicographical value and each 'a' contributes a value of 1.
    let answer: string[] = new Array(n).fill('a');

    // Start from the end of the string and work backwards since it is more efficient
    // to reduce the required numeric value by replacing 'a' with higher valued characters from the end.
    let index = number = n - 1;

    // Calculate the numeric value required to achieve exactly k
    // given that all 'a's have already contributed a value equal to n.
    let numericValueNeeded = k - n;

    // Fill the answer string from the end with 'z's (each 'z' has a numeric value of 26)
    // until the required numeric value is less than or equal to the value that can be represented
    // by a single character (i.e., 25 or less since we already have 'a' included).
    while (numericValueNeeded > 25) {
        answer[index--] = 'z'; // Replace 'a' with 'z' at the current index.
        numericValueNeeded -= 25; // Subtract 25 from the remaining value needed as we replaced 'a' with 'z'.
    }

    // There can be at most one character that is not 'a' or 'z' to perfectly reach the required value k.
    // Adjust the character at the current index by increasing its value to match the
    // remaining numericValueNeeded so that the total numeric value of the string equals k.
    answer[index] = String.fromCharCode(answer[index].charCodeAt(0) + numericValueNeeded);

    // Join the array of characters back into a string and return the result.
    return answer.join("");
}
```

Time and Space Complexity

Time Complexity

The given code's time complexity depends on the number of operations it performs in relation to the input parameters `n` and `k`.

- Initializing the answer list with 'a's: $O(n)$
- While loop decrementing `d` by at least 26 each time until `d` is no more than 25: This will run at most $(k-n)/26$ times, which is $O(k-n)/26)$. Since `-` can be at most `+ 26`, this simplifies to $O(n)$.
- Assigning a character other than 'a' to the appropriate index once in the answer list, which is a constant time operation.

Thus, the time complexity is the sum of these operations, which is $O(n + (k-n)/26)$. Given that $(k-n)/26 <= n$, the dominant term is $O(n)$. Therefore, we can consider the time complexity to be $O(n)$.

Space Complexity

The space complexity is determined by the amount of extra memory used relative to the input:

- The answer list `ans` of size `n`: $O(n)$
- A few integer variables `i` and `d`, which use a constant amount of space: $O(1)$

Considering both, the space complexity of the code is $O(n)$.