806. Number of Lines To Write String

Easy Array String Leetcode Link

Problem Description

In this problem, we are given a string `s` consisting of lowercase English letters and an array `widths` which indicates the pixel width of each lowercase letter. The pixel width of the letter 'a' is at `widths[0]`, the width of 'b' is at `widths[1]`, and so on up to 'z'. We need to write this string across several lines where each line can hold up to 100 pixels.

The task is to find out how many lines would be needed to write the entire string `s` and the width of the last line in pixels. The conditions given are that starting from the beginning of `s`, we place as many letters as we can on the first line without exceeding 100 pixels. When we can't fit a letter because it would exceed the line limit, we move to the next line and continue from there. This process is repeated until all the characters of the string `s` are written.

We are required to return an array `result` of length 2, where:

- `result[0]` is the total number of lines needed.
- `result[1]` is the width of the last line in pixels.

Intuition

The intuition behind the solution is to iterate over each character in the given string `s` and add up their widths while keeping track of the total width accumulated on the current line. We use the widths array to look up the width of a particular character.

We start with the first character in the string and keep a tally of the line width (`last`). We also initialize a counter for the number of lines (`row`) to 1 since we start with the first line.

For each character `c` in the string `s`, we perform the following steps:

1. Retrieve the character's width by finding its pixel width from the `widths` array using the ASCII value of 'a' as the base (`widths[ord(c) - ord('a')]`).
2. Check if adding this character's width to the current line's total width (`last`) will exceed 100 pixels.
3. If it does not exceed 100 pixels, we add this character's width to `last` and continue with the next character.
4. If it does exceed, we must start a new line. Therefore, we increment our line counter (`row`) by 1, and set `last` to the width of the current character which will be the starting width of the new line.

We repeat this process for each character until we reach the end of the string `s`. The result will be the total number of lines used (`row`) and the width of the last line (`last`). This pair is then returned as the solution.

Solution Approach

The implementation of the solution involves a simple iterative approach, checking each character's width and keeping a tally for the current line's total width. No complex data structures or intricate patterns are used; the solution is straightforward and efficient. Here's how it's done in detail:

1. Initialize two variables: `last` is set to 0, which will keep track of the current line's width in pixels, and `row` is set to 1, representing the initial line count since we start writing on the first line.

2. Iterate through each character `c` in the string `s` using a `for` loop to process one character at a time.

3. For each character `c`, find the width assigned to it in the `widths` array. This is done using `widths[ord(c) - ord('a')]`, which computes the correct index in the `widths` array based on the ASCII value of 'a' and the character `c`. This gives the pixel width of `c`.

4. Check if adding this character's width to the total width of the current line (`last`) will exceed the maximum allowed pixels per line (100 pixels). There are two possible scenarios:

   - If `last + w <= 100`, the current character can fit in the current line without exceeding the limit. In this case, we add the width of `c` to the current line total (`last`).

   - If `last + w > 100`, the current character cannot fit in the current line and we need to start a new line. Therefore, increase the line count `row` by 1, and reset `last` to the width of `c` since it will be the first character on the new line.

5. Continue this process until the end of the string is reached.

6. Finally, return the result as a list containing the total number of lines (`row`) and the width of the last line (`last`). This is done with the statement `return [row, last]`.

The algorithm completes in O(n) time complexity where 'n' is the length of the string `s`, as it needs to iterate through the entire string once. The space complexity is O(1), as the space used does not grow with the input size but remains constant, only needing to store the two variables `last` and `row`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following string `s` and `widths`:

- `s = "abcdefghijk"`
- `widths = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]`

This implies that each character has a width of 10 pixels, and since there are only lowercase characters, this rule applies uniformly to all.

Now let's walk through the solution step by step:

1. Initialize `last` as `0`, and `row` as `1`. These variables track the width of the current line and the total number of lines, respectively.

2. Start iterating over `s`, character by character:

   1. First character is `'a'`, with a width of `widths[0]` which is `10`. Add this to `last`, now `last = 10`.
   2. Next character `'b'` also has a width of `10`. `last + 10 <= 100`, so we add it. Now `last = 20`.
   3. We continue adding characters `c`, `d`, `e`, `f`, `g`, `h`, `i`, and `j`, each adding `10` to `last`, which becomes `100`.
   4. We reach character `'k'`, and `last` is currently `100`. If we add the width of `k`, `last` would become `110`, which exceeds our max line width of `100`. So, we start a new line.
   5. Increment `row` to `2`. We place `k` as the first character on this new line, so `last` is now `10`, the width of `k`.
3. At the end of the string, we have used `2` lines in total, and the width of the last line is `10` pixels.

4. Return the result as `[row, last]`, which is `[2, 10]` in our example.

Using this example, we went through the string and checked the width of each character, adding characters to the current line until we could no longer do so without exceeding 100 pixels, at which point a new line was started. The result gives us the total number of lines required and the width of the last line.

Python Solution

```python
class Solution:
    def numberOfLines(self, widths: List[int], text: str) -> List[int]:
        # Initialize variables to store the current width count and the row number
        current_width = 0
        row_count = 1

        # Iterate through each character in the provided text
        for char in text:
            # Calculate width of the current character based on its position in the alphabet
            char_width = widths[ord(char) - ord('a')]
            # Check if adding this character would exceed the maximum width of 100
            if current_width + char_width <= 100:
                # If not, add its width to the current line
                current_width += char_width
            else:
                # Otherwise, we need to move to a new line
                row_count += 1
                # Reset current width to the width of the new character
                current_width = char_width

        # Return the total number of lines and the width of the last line
        return [row_count, current_width]

```

Java Solution

```java
class Solution {
    // Define the constant for the maximum width of a line.
    private static final int MAX_WIDTH = 100;

    // Method to calculate the number of lines used and the width of the last line
    // when typing a string using widths provided for each character.
    public int[] numberOfLines(int[] widths, String s) {
        int currentLineWidth = 0; // Maintain the current width of the line.
        int numberOfRows = 1; // Start with one row.

        // Iterate through each character in the input string.
        for (char character : s.toCharArray()) {
            // Determine the width of the current character based on the widths array.
            int charWidth = widths[character - 'a'];

            // Check if the current character fits in the current line.
            if (currentLineWidth + charWidth <= MAX_WIDTH) {
                // Add the character width to the line if it fits.
                currentLineWidth += charWidth;
            } else {
                // If character doesn't fit, move to the next line and
                // reset the current line width to the width of this character.
                numberOfRows++; // Increment the number of rows.
                currentLineWidth = charWidth;
            }
        }

        // Return the number of rows used and the width of the last line.
        return new int[] {numberOfRows, currentLineWidth};
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Define a constant for the maximum width of a line.
    const int MAX_WIDTH = 100;

    // Function to determine the number of lines required to write a string 's',
    // and the width of the last line, given the widths for each lowercase letter.
    vector<int> numberOfLines(vector<int>& letterWidths, string s) {
        int currentWidth = 0, // Current accumulated width on the last line
            totalRows = 1;    // Start with one row

        // Iterate over each character in the given string 's'
        for (char c : s) {
            // Get the width of the current character based on the 'letterWidths' map
            int charWidth = letterWidths[c - 'a'];

            // Check if adding the current character exceeds the max width
            if (currentWidth + charWidth <= MAX_WIDTH) {
                // If it doesn't exceed, add to the current line width
                currentWidth += charWidth;
            } else {
                // If it exceeds, start a new line and set the current character's width as the starting width
                ++totalRows;           // Increment the row count
                currentWidth = charWidth; // Reset the width for the next line
            }
        }
        // Return a vector with two elements: the total number of lines and the width of the last line
        return {totalRows, currentWidth};
    }
};

```

Typescript Solution

```typescript
// Define a constant for the maximum width of a line.
const MAX_WIDTH = 100;

// Function to determine the number of lines required to write a string 'text',
// and the width of the last line, given the widths for each lowercase letter.
function numberOfLines(letterWidths: number[], text: string): [number, number] {
    let currentWidth = 0,   // Current accumulated width on the last line
        totalLines = 1;     // Start with one line

    // Iterate over each character in the given string 'text'
    for (const char of text) {
        // Get the width of the current character based on the 'letterWidths' array
        // ASCII value of 'a' is 97, substracing it from the ASCII value of char
        // will give us the index related to the character in the 'letterWidths' array.
        const charWidth = letterWidths[char.charCodeAt(0) - 'a'.charCodeAt(0)];

        // Check if adding the current character exceeds the max width
        if (currentWidth + charWidth <= MAX_WIDTH) {
            // If it doesn't exceed, add its width to the current line
            currentWidth += charWidth;
        } else {
            // If it exceeds, start a new line and set the current character's width as the starting width
            totalLines++;                       // Increment the line count
            currentWidth = charWidth;           // Reset the width for the new line
        }
    }

    // Return a tuple with two elements: the total number of lines and the width of the last line
    return [totalLines, currentWidth];
}

// Example usage:
// Depending on how you implement the code, you might need to export the function if it's part of a module
// export { numberOfLines };

```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the string `s`. This is because the algorithm iterates over each character in the input string exactly once, and the operations inside the loop (calculating width and checking/updating `last` and `row`) are constant time operations.

The space complexity of the code is `O(1)`. The amount of extra space used does not depend on the input size but is fixed, with a small number of integer variables being used to keep track of the rows (`row`) and the current width (`last`).