418. Sentence Screen Fitting

Medium String Dynamic Programming Simulation

Problem Description

In this problem, you are required to determine how many times a sentence can be fitted on a screen. The screen has dimensions defined by `rows` and `cols` where `rows` represent the number of lines on the screen and `cols` stand for the number of characters that can fit in a single line. The sentence is represented as a list of strings where each string is a word from the sentence.

A few rules need to be followed while placing the sentence on the screen:

1. Words must be put on the screen in the same order as they appear in the sentence.
2. No word should be split over two lines. If a word does not fit at the end of a line, it should be moved to the next line.
3. There must be a single space between two consecutive words on the same line.

The aim is to count the total number of times the sentence can be repeated on the screen under these constraints.

Intuition

The intuition behind the solution is to simulate the process of typing out the sentence on the screen by keeping track of the total number of characters that have been placed on the screen. We then use a variable to keep a count of where the next character would be placed in the sequence of the sentence that is repeated indefinitely.

Here's how the approach operates:

1. We first join all the words of the sentence separated by a space and append a space at the end to account for the separation between end and start of the repeated sentence.
2. This creates a string `s` which is a template of what we are trying to layout on the screen.
3. A variable `cur` is used to denote the position in the `s` string where the next character would be placed on the screen.
4. For each row on the screen, we increment `cur` by the number of columns `cols` since each row can hold this many characters.
5. We then check if at this `cur` position we are at a space character in the string `s`. If so, it means we can fit exactly up to this word on the current row, and we can move to one character past this position, which means the next word should start at the beginning of next line.
6. If `cur` doesn't land on a space, it means the last word doesn't fit, and we need to backtrack until we find the space that would mark the end of the last word that fits on the current row.
7. After completing all rows, we calculate how many full passes through the `s` string we made by dividing `cur` by the length of `s`, which tells us how many times the sentence was completely fitted on the screen.

By simulating the typing of the sentence row-by-row and handling the constraints, we can determine the total number of times the sentence can be repeated on the given screen.

Solution Approach

The solution provided is a smart linear time approach, which avoids the more obvious brute force method of trying to lay out the sentence on the screen as you would in real life. Here's how this solution works step by step, diving into the algorithm and the data structures:

- First step is to concatenate all words in the sentence with a space and also append a space at the end. This creates a single string `s` that represents the pattern of the sentence as it would be typed out.

```
s = " ".join(sentence) + " "
```

- The length of this string `m` is stored since we are going to use it often to wrap around when the end of the string is reached.

```
m = len(s)
```

- The algorithm uses a cursor `cur` to represent the current position in the string `s`. Initially, `cur` is set to 0, as we start at the beginning of the sentence.
- The main loop of the algorithm runs once for every row of the screen. In each iteration:

1. The cursor `cur` is moved forward by the number of columns `cols` as if typing out the characters in the current row.

```
cur += cols
```

2. If the character at the new cursor position `cur % m` (adjusted for wrapping around `s`) is a space, it means a word ends exactly at the end of the row and we can increment `cur` to look at the beginning of the next word (assuming it would start on the next row).

```
if s[cur % m] == " ":
    cur += 1
```

3. If we don't land on a space, we need to backtrack to the previous space. This represents moving the cursor back to the end of the last word that fit completely within the row. The while loop continues to decrement `cur` as long as the character at the decremented position isn't a space (and `cur` is not yet 0).

```
while cur and s[(cur - 1) % m] != " ":
    cur -= 1
```

- After the loop completes, `cur` represents the total number of characters that were fitted on the screen. By dividing this by the length of `s` (`s`), we get the total number of times the sentence has been fitted onto the screen.

```
return cur // m
```

This implementation efficiently simulates the typing of the sentence onto the screen, while handling cases where a word doesn't fit at the end of a row and needs to be moved to the next row. It does so without actually iterating over every single character, thereby being more efficient than a brute force approach, and doesn't require complex data structures or patterns—just a careful use of a string and character positioning.

Example Walkthrough

Here's a small example to illustrate the solution approach:

Let's say we have a sentence represented by the list of strings `sentence = ["hello", "world"]` and a screen with dimensions `rows = 2` and `cols = 8`. Now, let's walk through the algorithm to determine how many times the sentence can be fitted on the screen.

1. We first join the words of the sentence separated by a space and append a space at the end to simulate the space at the end of the sentence when it repeats. Our string `s` will look like this:

```
s = "hello world "
```

2. The length of `s` is calculated:

```
m = len(s)  # m is 12
```

3. We initialize the cursor `cur` to zero since we start at the beginning of the sentence:

```
cur = 0
```

4. Now, for each row on the screen, we do the following:

   - For the first row, we start with `cur = 0` and move the cursor forward by `cols` (**8** in this case):

   ```
   cur += cols  # cur becomes 8
   ```

   At this position (`cur % m`), we find that `s[8 % 12]` (which is `s[8]`) is a space, indicating the word "world" fitting perfectly at the end of the row. So we increment `cur` to move to the start of the next word:

   ```
   cur += 1  # cur is now 9
   ```

   - For the second row, we again move `cur` forward by `cols` (**8**):

   ```
   cur += cols  # cur becomes 17
   ```

   But now, `s[17 % 12]` (which is `s[5]`) is not a space; it's the character 'o' from the word "hello". This means the word "hello" does not fit completely and needs to continue on the next line. So we backtrack until we find a space:

   ```
   while cur and s[(cur - 1) % m] != " ":
       cur -= 1
   ```

   After backtracking, `cur` becomes `12`, which is the next position after a space, indicating the start of the next word on a new line.

5. Having processed both rows, we determine the number of times the sentence fitted on the screen by dividing `cur` by the length of `s` (**6**):

```
count = cur // m  # count is 1
```

The result is `1`, meaning the sentence "hello world" was completely fitted on the screen only once given the dimensions `rows = 2` and `cols = 8`.

This example demonstrates the solution approach to the presented problem efficiently using simple string operations to simulate the process of typing a sentence onto a screen with specific dimensions.

Solution Implementation

Python

```
class Solution:
    def wordsTyping(self, sentence: List[str], rows: int, cols: int) -> int:
        # Combine the sentence into a single string with spaces and an extra space at the end
        concatenated_sentence = " ".join(sentence) + " "
        sentence_length = len(concatenated_sentence)
        current_position = 0  # Start at the beginning of the concatenated sentence

        # Iterate over each row
        for _ in range(rows):
            # Move the current position forward by the number of columns
            current_position += cols

            # If the current character is a space, it fits perfectly and move to next character
            if concatenated_sentence[current_position % sentence_length] == " ":
                current_position += 1
            # If not, backtrack to the last space so that the word doesn't break
            while current_position > 0 and concatenated_sentence[(current_position - 1) % sentence_length] != " ":
                current_position -= 1

        # Return the number of times the sentence was fully fitted in the rows and columns
        return current_position // sentence_length
```

Java

```
class Solution {

    public int wordsTyping(String[] sentence, int rows, int cols) {
        // Join all the words in the sentence array with spaces and add an extra space at the end.
        String joinedSentence = String.join(" ", sentence) + " ";

        // Calculate the total length of the joined sentence.
        int sentenceLength = joinedSentence.length();

        // Initialize character pointer to track the total characters that fit on the screen.
        int charPointer = 0;

        // Loop through each row.
        while (rows-- > 0) {
            // Add the number of columns to the character pointer because each row can contain `cols` number of characters.
            charPointer += cols;

            // If the character immediately after the last character that fits in the row is a space,
            // we can simply increment the character pointer since we're at the end of a word.
            if (joinedSentence.charAt(charPointer % sentenceLength) == ' ') {
                charPointer++;
            } else {
                // If it's not a space, we're in the middle of a word and need to backtrack
                // until we find the beginning of the word, so the whole word can fit on the next line.
                while (charPointer > 0 && joinedSentence.charAt((charPointer - 1) % sentenceLength) != ' ') {
                    charPointer--;
                }
            }
        }

        // The number of times the sentence can be repeated is the total characters that fit
        // divided by the sentence length.
        return charPointer / sentenceLength;
    }
}
```

C++

```
class Solution {
public:
    int wordsTyping(vector<string>& sentence, int rows, int cols) {
        // Construct a single string from the sentence, with spaces between words
        string sentenceString;
        for (const auto& word : sentence) {
            sentenceString += word + " ";
        }

        // Get the length of the constructed string
        int sentenceLength = sentenceString.size();
        int currentPos = 0; // Represents the current position in the sentenceString

        // Loop through each row
        while (rows--) {
            // Add the number of columns to current position as we can type as many chars
            currentPos += cols;

            // If the current position is at a space, it's the end of a word so move to the next character
            if (sentenceString[currentPos % sentenceLength] == ' ') {
                currentPos++;
            } else {
                // Move back to find the end of the previous word if it's not perfectly fitted
                while (currentPos > 0 && sentenceString[(currentPos - 1) % sentenceLength] != ' ') {
                    --currentPos;
                }
            }
        }

        // Dividing the total characters typed by the sentence length gives the number of times
        // the sentence has been typed fully
        return currentPos / sentenceLength;
    }
};
```

TypeScript

```
// Function to calculate how many times a sentence can be fitted on a screen
function wordsTyping(sentence: string[], rows: number, cols: number): number {
    // Combine words of the sentence into a single string separated by spaces,
    // and append a space to mark the end of the sentence.
    const sentenceString = sentence.join(' ') + ' ';

    // Initialize a pointer to track the current position in the sentenceString.
    let currentPosition = 0;

    // Get the total length of the sentence string including the trailing space.
    const sentenceLength = sentenceString.length;

    // Loop through each row of the screen.
    for (let row = 0; row < rows; ++row) {
        // Move the start of each row, we can add 'cols' number of characters.
        currentPosition += cols;

        // If the current position is at a space, it means we have exactly
        // completed a word and can move to the next character.
        if (sentenceString[currentPosition % sentenceLength] === ' ') {
            currentPosition++;
        } else {
            // Move the current position backwards until a space is found,
            // which indicates the end of a word. This step ensures that a
            // word is not cut off in the middle when the row ends.
            while (currentPosition > 0 && sentenceString[currentPosition - 1] % sentenceLength] !== ' ') {
                currentPosition--;
            }
        }
    }

    // Calculate and return the number of times the sentence can be fitted on the screen
    // by dividing the total number of characters added by the length of the sentence string.
    return Math.floor(currentPosition / sentenceLength);
}
```

```
class Solution:
    def wordsTyping(self, sentence: List[str], rows: int, cols: int) -> int:
        # Combine the sentence into a single string with spaces and an extra space at the end
        concatenated_sentence = " ".join(sentence) + " "
        sentence_length = len(concatenated_sentence)
        current_position = 0  # Start at the beginning of the concatenated sentence

        # Iterate over each row
        for _ in range(rows):
            # Move the current position forward by the number of columns
            current_position += cols

            # If the current character is a space, it fits perfectly and move to next character
            if concatenated_sentence[current_position % sentence_length] == " ":
                current_position += 1
            # If not, backtrack to the last space so that the word doesn't break
            while current_position > 0 and concatenated_sentence[(current_position - 1) % sentence_length] != " ":
                current_position -= 1

        # Return the number of times the sentence was fully fitted in the rows and columns
        return current_position // sentence_length
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(rows * cols)` in the worst case. This is because the code iterates over the number of `rows` given, and for each row, it may potentially iterate over a number of characters up to the number of `cols` when adjusting the `cur` variable to point to the start of the next word. However, the average case time complexity can be better since it's not always that we traverse back the full length of cols for each row.

Space Complexity

The space complexity of the code is `O(n)` where `n` is the total length of the string constructed by joining all words in the sentence with spaces, plus an additional space at the end. This additional space is used to signify the separation between the end of the sentence and the beginning when the sentence is to be repeated. No other significant space is used, as the variables used for iteration and indexing are of constant size.