1451. Rearrange Words in a Sentence

Medium String Sorting

Problem Description

Given a sentence `text`, which is a string comprised of space-separated words, you are tasked with rearranging the words according to their lengths. Notably, the sentence follows certain formatting rules: the first character of the sentence is uppercase, and each subsequent word is separated by a single space. Your job is to reorganize the sentence so that shorter words come first, maintaining the order of words of the same length. After rearranging, the modified sentence should also adhere to the initial format with the first letter capitalized and space-separated words.

Intuition

To solve this problem, the initial step is to break the sentence into individual words. This can be easily achieved by using the `split` function in Python, which divides the string at each space and returns a list of words. Since the first word is capitalized, and the remaining words are not, we first need to convert the initial word to lowercase to ensure uniformity in formatting post-sorting.

Next, we sort the list of words based on their lengths. The `sort` function in Python is stable, which means that if multiple elements have the same sorting key—in this case, the length of the word—they will maintain their original order. This property is essential for meeting the requirement of preserving the order of words that have the same number of characters.

After sorting the words by length, we then need to capitalize the first word to match the sentence formatting rules. The `title` method is used to capitalize the first character of the first word.

Finally, we join the sorted words back into a string, separating them with spaces, to construct the rearranged sentence. This sentence now has its words sorted by length while preserving the order of words of the same length and maintaining the initial formatting rule with the capitalized first letter.

Solution Approach

The solution approach can be broken down into the following steps, aligning with the instructions in the provided Python code:

1. **Splitting the input sentence into words**: This makes use of Python's string `split()` function, which breaks the string at each space character and returns a list of words.
2. **Converting the first word to lowercase**: As the first letter of the input sentence is capitalized and we want to sort the words without being affected by their capitalization, the first word of the list is converted to lowercase using the `lower()` string method.
3. **Sorting the words by length**: The sorted `words` list is achieved by calling the `sort()` method with a key specifying `len`, which indicates that the sorting criterion is the length of each word. As the `sort()` method is stable, it retains the order of words that compare as equal (words of the same length).
4. **Capitalizing the first word of the sorted list**: After sorting, to adhere to the problem's output format criteria, the first word of the sorted list should be capitalized. This is accomplished by replacing the first word with its titled version, using the `title()` string method, which capitalizes the first character of the string.
5. **Joining the sorted words into a sentence**: The final sentence is constructed by combining the sorted words back together using the `join()` method with a `" "` space as the separator.

The key algorithms and data structures used in this solution are:

- String manipulation methods: `split()`, `lower()`, `title()`, and `join()` are used to divide and reformat the sentence accordingly.
- The sorting algorithm used by Python's `sort()` method: It rearranges the elements of the list in place and is known for its efficiency and stability.

This solution elegantly leverages built-in Python functions to address the problem, ensuring that the approach is both readable and efficient.

Example Walkthrough

Let's go through a small example to illustrate the solution approach described above. Imagine we have the sentence `text`:

"The quick brown fox jumps over the lazy dog."

We aim to sort this sentence by the length of the words while preserving the formatting.

1. **Splitting the input sentence into words**:

   We begin by splitting `text` into words using `split()`:

   ```
   words = text.split()
   ```

   results in:

   ```
   ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
   ```

2. **Converting the first word to lowercase**:

   We then convert the first word to lowercase:

   ```
   words[0] = words[0].lower()
   ```

   gives us:

   ```
   ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
   ```

3. **Sorting the words by length**:

   Next, we sort the words by their lengths using the `sort()` method:

   ```
   words.sort(key=len)
   ```

   sorts the list as:

   ```
   ["the", "fox", "the", "dog", "over", "lazy", "quick", "brown", "jumps"]
   ```

   Note that "the" appears before "fox" because it comes first in the original order.

4. **Capitalizing the first word of the sorted list**:

   We capitalize the first word in the sorted list to conform to the initial sentence's formatting:

   ```
   words[0] = words[0].title()
   ```

   modifies the list to:

   ```
   ["The", "fox", "the", "dog", "over", "lazy", "quick", "brown", "jumps"]
   ```

5. **Joining the sorted words into a sentence**:

   Finally, we join the sorted words into a sentence:

   ```
   sorted_text = ' '.join(words)
   ```

   produces the rearranged sentence:

   "The fox the dog over lazy quick brown jumps."

The sentence now adheres to the initial format with the first word capitalized and the rest of the words sorted by length, respecting the order of words with equal length. The approach shows the effectiveness of Python's string manipulations and the stability of the sort function.

Solution Implementation

Python

```python
class Solution:
    def arrangeWords(self, text: str) -> str:
        # Split the sentence into words
        words = text.split()

        # Convert the first word to lowercase to standardize sorting and capitalization
        words[0] = words[0].lower()

        # Sort the words based on their lengths
        words.sort(key=len)

        # Capitalize the first word of the arranged sentence
        words[0] = words[0].capitalize()

        # Join the words back into a sentence and return
        return " ".join(words)
```

Java

```java
class Solution {
    public String arrangeWords(String text) {
        // Split the input string into an array of words.
        String[] words = text.split(" ");

        // Convert the first word of the array to lowercase to standardize case for comparison.
        words[0] = words[0].toLowerCase();

        // Sort the array of words by the length of each word.
        Arrays.sort(words, Comparator.comparingInt(String::length));

        // Capitalize the first letter of the first word in the sorted array.
        words[0] = words[0].substring(0, 1).toUpperCase() + words[0].substring(1);

        // Join the words back into a single string with spaces and return the result.
        return String.join(" ", words);
    }
}
```

C++

```cpp
#include <vector>
#include <sstream>
#include <algorithm>
#include <cctype>

class Solution {
public:
    // Function to rearrange words such that they are sorted by length
    string arrangeWords(string text) {
        vector<string> words; // Vector to hold individual words
        stringstream ss(text); // Stringstream to break text into words
        string currentWord;

        // Extract words from the stream and push them to the vector
        while (ss >> currentWord) {
            words.push_back(currentWord);
        }

        // Convert the first letter of the initial word to lowercase
        words[0][0] = tolower(words[0][0]);

        // Sort the words by their length using stable_sort to maintain
        // the original order if lengths are equal
        stable_sort(words.begin(), words.end(), [](const string& a, const string& b) {
            return a.size() < b.size();
        });

        // Concatenate words back into a single string with spaces
        string result;
        for (const auto& word : words) {
            result += word + " ";
        }

        // Remove the trailing space added from the last iteration
        result.pop_back();

        // Capitalize the first letter of the result
        result[0] = toupper(result[0]);

        // Return the final rearranged string
        return result;
    }
};
```

TypeScript

```typescript
function arrangeWords(text: string): string {
    // Split the input text into an array of words
    let words: string[] = text.split(' ');

    // Change the first word to lowercase to ensure uniformity before sorting
    words[0] = words[0].toLowerCase();

    // Sort the array of words based on their length in ascending order
    words.sort((firstWord, secondWord) => firstWord.length - secondWord.length);

    // Capitalize the first character of the first word in the sorted array
    words[0] = words[0].charAt(0).toUpperCase() + words[0].slice(1);

    // Join the words back into a string with spaces in between and return the result
    return words.join(' ');
}
```

```python
class Solution:
    def arrangeWords(self, text: str) -> str:
        # Split the sentence into words
        words = text.split()

        # Convert the first word to lowercase to standardize sorting and capitalization
        words[0] = words[0].lower()

        # Sort the words based on their lengths
        words.sort(key=len)

        # Capitalize the first word of the arranged sentence
        words[0] = words[0].capitalize()

        # Join the words back into a sentence and return
        return " ".join(words)
```

Time and Space Complexity

The given Python code takes a string `text`, splits it into words, and rearranges those words in ascending order of their lengths. The first word of the input text is converted to lowercase, and after sorting, the first word of the result is capitalized.

Time Complexity

The time complexity of the method mainly comes from three operations: splitting the text, sorting the words, and joining them back into a single string.

1. Splitting the text: This operation is $O(n)$ where n is the length of the input string since it needs to go through the entire string once.
2. Sorting the words: Python uses TimSort (a hybrid sorting algorithm derived from merge sort and insertion sort) for its sort() function, which has a worst-case time complexity of $O(m * \log(m))$, where m is the number of elements in the list to sort. In this case, m is the number of words. Note that since the sort key is the length of the words, and lengths can be computed in constant time, the complexity of the sorting step is purely based on the number of words.
3. Joining the words: This is again $O(n)$ as we need to combine all the words to form a single string, inserting a space in between each word.

Combining these, the overall time complexity of this function is $O(n + m * \log(m))$. In the worst case where m is proportional to n (i.e., n is very close or equal to m when the text has only space-separated single-character words), the time complexity simplifies to $O(n * \log(n))$.

Space Complexity

1. Additional List `words`: We store all m words of the input text in a separate list, which gives us $O(m)$ space complexity.
2. Internal workings of the sort function: Python's TimSort requires $O(m)$ space in the worst case, however this is a stable sort, so it does not require extra space that is proportional to n (the total number of characters).

Therefore, the space complexity is also $O(n)$. If we again assume that n is proportional to n in the worst-case scenario, then the space complexity can be considered $O(n)$.