1859. Sorting the Sentence

Easy String Sorting Leetcode Link

Problem Description

In this problem, we are given a sentence that has been jumbled in a particular way. Initially, the sentence was a list of words separated by spaces with no leading or trailing spaces. However, it has been 'shuffled' by appending a 1-indexed position number at the end of each word, and then the words have been rearranged. Our goal is to reconstruct the original sentence from this shuffled version.

For example, the original sentence "I have a cat" could be converted to "I1 have2 a4 cat3" in its shuffled form, and we would need to rearrange it back to the original order.

The constraints tell us that the shuffled sentence will contain no more than 9 words, and each word consists only of lowercase or uppercase English letters followed by the position number.

Intuition

The solution involves taking the following steps:

1. Split the shuffled sentence into individual words, so we can process each word separately.
2. Each word ends with a digit that indicates its original position in the sentence before shuffling. We need to create a structure, like a list, to hold the words in their correct positions.
3. Loop over each word from the shuffled sentence. Extract the last character of each word, which is the digit. Since we know that the position number is 1-indexed and Python lists are 0-indexed, we subtract 1 from the extracted digit to find the correct index for placing the word in our list.
4. Remove the digit from each word as we place the word into the correct position in our list.
5. Once all words are placed in their correct positions, we join them back into a sentence using a space as the separator.

The rationale for this approach is that we can use the position numbers as indices for an array or a list. By doing so, we ensure that each word is placed exactly in the position it was before shuffling. Since the words can only be shuffled by their positional indices and the indices are unique, this method guarantees that we can recover the original sentence.

Solution Approach

The implementation of the solution uses Python's built-in string and list operations to deconstruct and reconstruct the sentence. Here are the steps following the algorithm:

1. **String Splitting**: The given shuffled sentence `s` is split using the `split()` method, which divides the sentence into a list of words based on spaces.

   ```
   words = s.split()
   ```

2. **List Initialization**: A new list `ans` is created with the same length as the number of words. Each element is initialized as `None` to be filled in with the corresponding sorted word.

   ```
   ans = [None] * len(words)
   ```

3. **Loop and Parsing**: Each word in the list `words` is processed in a loop.

   ```
   for w in words:
   ```

   For each word `w`, the last character (which is a digit representing the word's original position) is converted to an integer and 1 is subtracted to account for Python's 0-based indexing.

   ```
   i = int(w[-1]) - 1
   ```

   Then the digit is removed from the word, and the resulting word (with the digit removed) is placed at the correct index in `ans`.

   ```
   ans[i] = w[:-1]
   ```

4. **Joining the List**: Finally, the list `ans` is joined into a string using `' '.join(ans)` with spaces between elements to form the reconstructed sentence.

   ```
   return ' '.join(ans)
   ```

In terms of data structures, a list is used to hold the words at the indices corresponding to their original positions. This means we make use of the array index as a direct access table.

In terms of patterns, the solution utilizes simple parsing of strings and the strategy of using an auxiliary data structure (list) to organize items based on specific keys (in this case, the number appended to each word). The algorithm complexity is O(n) where n is the number of words in the sentence, as each word is visited exactly once to place it in its correct position.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose the shuffled sentence given is `"is2 This1 an4 example3"`. Our goal is to rearrange it back to `"This is an example"`.

Here is how the algorithm works:

1. **String Splitting**: We first split the shuffled sentence into words:

   ```
   words = "is2 This1 an4 example3".split()  # words = ['is2', 'This1', 'an4', 'example3']
   ```

2. **List Initialization**: We prepare a new list with the length equal to the number of words, which is 4 in our example:

   ```
   ans = [None] * len(words)  # ans = [None, None, None, None]
   ```

3. **Loop and Parsing**: We then loop through each word in our list of words and parse each one.

   ```
   for w in words:
       i = int(w[-1]) - 1  # For 'is2', i = 2 - 1 = 1
       ans[i] = w[:-1]     # The word without the last character ('is') is placed in 'ans' at index 1
   ```

   Following this step for each word places them in their correct positions on the `ans` list:

   - `'This1'` goes to `ans[0]`
   - `'is2'` goes to `ans[1]`
   - `'example3'` goes to `ans[2]`
   - `'an4'` goes to `ans[3]`

   Now our `ans` list looks like this: `['This', 'is', 'example', 'an']`

4. **Joining the List**: The last step is to join these words with spaces to form the reconstructed sentence:

   ```
   return ' '.join(ans) # "This is example an"
   ```

However, we notice that the output doesn't quite match the expected sentence `"This is an example"`. Upon reviewing the loop, we realize that we have incorrectly sequenced `'example'` and `'an'` due to a mistake in their indices. Correctly placing `'an'` in `ans[2]` and `'example'` in `ans[3]` during the parsing loop we get the right order:

- `'This1'` goes to `ans[0]`
- `'is2'` goes to `ans[1]`
- `'an4'` goes to `ans[2]`
- `'example3'` goes to `ans[3]`

Now `ans` is `['This', 'is', 'an', 'example']`, and after joining, it indeed forms the correct original sentence `"This is an example"`.

Python Solution

```python
class Solution:
    def sortSentence(self, sentence: str) -> str:
        # Split the sentence into a list of words
        words = sentence.split()

        # Initialize a list with the same length as the number of words
        sorted_words = [None] * len(words)

        # Iterate over each word in the list
        for word in words:
            # The index is the last character of the word, converted to an integer and decremented by 1
            index = int(word[-1]) - 1
            # Assign the word (without the last character) to the correct position
            sorted_words[index] = word[:-1]

        # Join the list of words into a sentence with spaces between words
        return ' '.join(sorted_words)

```

Java Solution

```java
class Solution {
    public String sortSentence(String s) {
        // Split the input string into words based on spaces
        String[] words = s.split(" ");

        // Create an array to hold the sorted words
        String[] sortedWords = new String[words.length];

        // Iterate over the array of words
        for (String word : words) {
            // Find the index for the sorted position from the last character of the word
            // '1' is subtracted because the positions in the task are 1-based and array indices are 0-based
            int index = word.charAt(word.length() - 1) - '1';

            // Assign the word without the number to the correct position in the sortedWords array
            // The substring method excludes the last character which is the digit
            sortedWords[index] = word.substring(0, word.length() - 1);
        }

        // Join the sorted words into a single string with spaces and return it
        return String.join(" ", sortedWords);
    }
}

```

C++ Solution

```cpp
#include <string>
#include <sstream>
#include <vector>

class Solution {
public:
    // The function takes a scrambled sentence with words followed by their position
    // in the original sentence and returns the sorted sentence.
    string sortSentence(string s) {
        istringstream iss(s); // Used to split the string into words based on whitespace
        string word; // Temporary string to hold each word from the stream
        vector<string> words; // To store words as they are read from the stream

        // Reading words from the stringstream into the vector
        while (iss >> word) {
            words.push_back(word);
        }

        // Create a result vector with the same size as the number of words
        vector<string> result(words.size());

        // Placing words at their correct index as indicated by the last character
        for (auto& currWord : words) {
            // Find the index by subtracting '1' to convert char to int (0-based indexing)
            int index = currWord.back() - '1';
            // Add the word to the result without the number
            result[index] = currWord.substr(0, currWord.size() - 1);
        }

        // Construct the sorted sentence by concatenating words in the result
        string sortedSentence; // To store the final sorted sentence
        for (auto& sortedWord : result) {
            sortedSentence += sortedWord + " "; // Add each word followed by a space
        }

        // Remove the trailing space from the last word
        sortedSentence.pop_back();

        // Return the sorted sentence
        return sortedSentence;
    }
};

```

Typescript Solution

```typescript
// This function sorts a sentence based on the numerical indices attached to its words.
function sortSentence(sentence: string): string {
    // Split the sentence into words based on space delimiter.
    const words = sentence.split(' ');

    // Initialize an array to hold the sorted words. Its length is the same as the number of words.
    const sortedWords = new Array<string>(words.length);

    // Iterate over each word to place them in the correct position.
    for (const word of words) {
        // The index for the word's position is at the end of the word (one digit).
        // Subtract the char code of '1' to convert from char to actual numerical index.
        const index = word.charCodeAt(word.length - 1) - '1'.charCodeAt(0);

        // Assign the word (without the numerical index) to the correct position.
        sortedWords[index] = word.slice(0, -1); // Slice off the last character, the numerical index.
    }

    // Join the sorted words back into a sentence with spaces and return the sorted sentence.
    return sortedWords.join(' ');
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given `sortSentence` function can be analyzed as follows:

1. Splitting the sentence into words with `s.split()`. In the worst case, this operation has a time complexity of `O(n)`, where `n` is the length of the input string `s`.
2. Initializing the answer list `ans` with `None` * the number of words. The time complexity of this operation is `O(m)`, where `m` is the number of words in `s`.
3. Looping over each word `w` to place it at the correct position in `ans`. This is another `O(m)` operation, as it iterates through all words once.
4. Within the loop, extracting the position `i` and word without the number `w[:-1]` is `O(1)` for each word since the word is sliced at a constant index from the end and assigning the position is a direct index access.
5. Joining the words in `ans` into a single string with `' '.join(ans)`. The time complexity of this operation can be seen as `O(m * k)`, where `k` is the average length of a word without the trailing number, as it has to concatenate each word with a space in between.

The dominant factors are the splitting of the initial string and the joining of the words in the list. Therefore, the total time complexity is `O(n + m + m * k)`, which can be simplified to `O(n + m * k)` because both `n` and `k` are related to the length of the input string `s`.

Space Complexity

The space complexity of the `sortSentence` function can be analyzed as follows:

1. The list of words obtained from splitting the string `s.split()` requires `O(m)` space, where `m` is the number of words.
2. The `ans` list also requires `O(m)` space to hold the words in their sorted order.
3. Temporary variables such as `w` and `i` use a constant `O(1)` space as they are just references to elements in the list or integers.

Hence, the total space complexity is `O(m) + O(m)` which equals `O(m)`, where `m` is the number of words in the sentence.