1181. Before and After Puzzle

Medium Array Hash Table String Sorting

Problem Description

The problem involves creating a set of new phrases called "Before and After puzzles." To form these puzzles, one should merge pairs of given phrases under the condition that the last word of the first phrase is *identical* to the first word of the second phrase. This merging has to consider the order of the phrases as swapping them yields different results. The expected outcome is to produce a unique set of these merged phrases, without any duplications and sorted in lexicographical (dictionary) order.

Additionally, the constraints are as follows:

- Each phrase consists only of lowercase letters and spaces.
- Spaces do not appear at the beginning or the end of a phrase.
- Phrases do not contain consecutive spaces.

Intuition

The intuition for solving this problem includes the idea that we only need to look at the first and last words of each phrase to determine if a valid "Before and After puzzle" can be formed. To efficiently check for mergeable phrases, we can follow these steps:

1. Preprocess the input list of phrases by splitting each phrase into its first and last word, noting that these are the crucial points of intersection for merging.
2. Iterate over the pairs of phrases (excluding pairs of the same phrase) and check if the last word of one phrase matches the first word of another phrase. If they do, we have a potential puzzle.
3. Merge the phrases by concatenating them, excluding the duplicate word that serves as a joint between them.
4. Rather than adding all the puzzles into a list sequentially, we can add them to a set to ensure that there are no duplicates.
5. After considering all the pairs and merging where possible, convert the set of puzzles into a list and sort it lexicographically.

This logical sequence of operations leads us to a solution that not only finds all possible "Before and After puzzles" but also ensures the correct and concise order of the final output.

Solution Approach

The implementation of the solution follows a straightforward, brute-force approach with some optimizations used to process the phrases and store the results. The algorithm, data structures, and pattern used in the reference solution are explained below:

1. **Data Preprocessing:** Initially, each phrase is split into its constituent words using `split()`. Saving only the first and last words of each phrase, since these are the points of potential merging, reduces unnecessary computation. These tuples of (first, last) words are stored in a list called `ps`.
2. **Iterative Checking:** After preprocessing, the code uses nested `for` loops to iterate over all possible pairs (excluding self-pairing). The condition `if i != j` ensures no phrase is paired with itself.
3. **Condition Validation and Merging:** For each pair, the last word of the first phrase (`ps[i][1]`) is compared with the first word of the second phrase (`ps[j][0]`) to check if they are the same. If they are, the two phrases are merged by concatenating the full first phrase with the second one, starting right after its first word. This is done by slicing the second phrase from the length of its first word: `phrases[j][len(ps[j][0]):]`.
4. **Elimination of Duplicates and Sorting:** Instead of adding the merged phrases directly to the result list, they are added to a set `ans`. Using a set automatically removes any duplicate entries, taking advantage of the properties of sets where each element is unique. After processing all pairs, the set is converted into a list (because sets cannot be sorted) which is then sorted using the built-in sorted function to ensure the result is in lexicographic order.
5. **Returning the Result:** Finally, the sorted list is returned as the desired output.

The solution uses basic Python data structures: the list for storing phrases and tuples for first and last words, and the set for ensuring unique results. The approach does not use any advanced algorithms but relies on comparing the vital elements (first and last words) of phrases combined with standard operations offered by Python's data structures.

One could argue that there is room for optimization by using a hash map to store phrases by their first and last words for O(1) lookups, potentially reducing the overall time complexity, but the provided solution is straightforward and sufficient for the problem at hand.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we are given the following list of phrases:

```
["writing code", "code rocks", "daily challenges", "challenges code"]
```

First, we apply data preprocessing. We need only the first and last words of each phrase:

- "writing code": first word "writing", last word "code".
- "code rocks": first word "code", last word "rocks".
- "daily challenges": first word "daily", last word "challenges".
- "challenges code": first word "challenges", last word "code".

Then we iterate checking pairs of phrases to merge:

1. "writing code" (last word "code") can be merged with "code rocks" (first word "code") to form "writing code rocks".
2. "writing code" (last word "code") can also be merged with "challenges code" (first word "code") to form "writing challenges code".
3. "daily challenges" (last word "challenges") can be merged with "challenges code" (first word "challenges") to form "daily challenges code".

During merging, we're careful not to pair a phrase with itself and we avoid duplicating the common word.

Next, we add the merged phrases to a set to avoid duplicates:

- The set will contain ("writing code rocks", "writing challenges code", "daily challenges code").

Finally, we sort the results lexicographically and return them:

- The sorted list will be ["daily challenges code", "writing challenges code", "writing code rocks"].

This is the expected output of the program based on our small example, illustrating the solution approach described in the content.

Solution Implementation

Python

```python
class Solution:
    def beforeAndAfterPuzzles(self, phrases):
        # Store the first and last word of each phrase
        first_and_last_words = []
        for phrase in phrases:
            words = phrase.split()
            first_and_last_words.append((words[0], words[-1]))

        num_phrases = len(first_and_last_words)  # The number of phrases
        results = []  # Store the results

        # Loop through all combinations of phrases
        for i in range(num_phrases):
            for j in range(num_phrases):
                # Check if i and j are not the same, and the last word of phrase i matches the first word of phrase j
                if i != j and first_and_last_words[i][1] == first_and_last_words[j][0]:
                    # Append the combined phrase, removing the duplicate word, to the results
                    combined_phrase = phrases[i] + phrases[j][len(first_and_last_words[j][0]):]
                    results.append(combined_phrase)

        # Sort the results and remove duplicates
        return sorted(set(results))
```

Java

```java
import java.util.*;

class Solution {
    public List<String> beforeAndAfterPuzzles(String[] phrases) {
        // Determine the number of phrases.
        int numPhrases = phrases.length;

        // This array will hold the first and last words of each phrase.
        String[][] firstLastWords = new String[numPhrases][];
        for (int i = 0; i < numPhrases; ++i) {
            // Split the phrase into words.
            String[] words = phrases[i].split(" ");
            // Store the first and the last word of each phrase.
            firstLastWords[i] = new String[] {words[0], words[words.length - 1]};
        }

        // Use a set to avoid duplicate phrases and to store the final phrases.
        Set<String> uniquePhrases = new HashSet<>();
        for (int i = 0; i < numPhrases; ++i) {
            for (int j = 0; j < numPhrases; ++j) {
                // Exclude same phrase comparisons and check if last word of i is
                // equal to first word of j to form a new phrase.
                if (i != j && firstLastWords[i][1].equals(firstLastWords[j][0])) {
                    // Concatenate the phrases by removing the overlapping word.
                    uniquePhrases.add(phrases[i] + phrases[j].substring(firstLastWords[j][0].length()));
                }
            }
        }

        // Convert the phrase set into a list.
        var sortedPhrases = new ArrayList<>(uniquePhrases);
        // Sort the list in alphabetical order.
        Collections.sort(sortedPhrases);

        return sortedPhrases;
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <utility>
#include <unordered_set>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to assemble phrases such that the last word of one phrase is the first word of another.
    vector<string> beforeAndAfterPuzzles(vector<string>& phrases) {
        int phraseCount = phrases.size();
        pair<string, string> firstLastWords[phraseCount];

        // Separate each phrase into first and last words.
        for (int i = 0; i < phraseCount; ++i) {
            int firstSpacePosition = phrases[i].find(' ');
            if (firstSpacePosition == string::npos) {
                // If the phrase has only one word, both first and last are the same.
                firstLastWords[i] = {phrases[i], phrases[i]};
            } else {
                // Find the last space position to separate the last word.
                int lastSpacePosition = phrases[i].rfind(' ');
                firstLastWords[i] = {
                    phrases[i].substr(0, firstSpacePosition),
                    phrases[i].substr(lastSpacePosition + 1)
                };
            }
        }

        unordered_set<string> uniquePhrases; // To prevent duplicate phrases.

        // Iterate through each pair and combine phrases where the last word of one
        // is the first word of another.
        for (int i = 0; i < phraseCount; ++i) {
            for (int j = 0; j < phraseCount; ++j) {
                if (i != j && firstLastWords[i].second == firstLastWords[j].first) {
                    uniquePhrases.insert(phrases[i] +
                                         phrases[j].substr(firstLastWords[i].second.size()));
                }
            }
        }

        // Transfer and sort the results into a vector for output.
        vector<string> resultUniquePhrases.begin(), uniquePhrases.end());
        sort(result.begin(), result.end());
        return result;
    }
};
```

TypeScript

```typescript
function beforeAndAfterPuzzles(phrases: string[]): string[] {
    // Split each phrase into an array of its first and last words.
    const firstLastWords: string[][] = [];
    for (const phrase of phrases) {
        const words = phrase.split(' ');
        firstLastWords.push([words[0], words[words.length - 1]]);
    }

    // Count of phrases to process.
    const phraseCount = firstLastWords.length;

    // Set to store unique combined phrases.
    const combinedPhrasesSet: Set<string> = new Set();

    // Loop through each pair of phrases.
    for (let i = 0; i < phraseCount; ++i) {
        for (let j = 0; j < phraseCount; ++j) {
            // Check if the phrases are not the same phrase and the last word of one
            // phrase is the same as the first word of the other.
            if (i !== j && firstLastWords[i][1] === firstLastWords[j][0]) {
                // Combine phrases by appending the substring of the second phrase that
                // excludes the overlapping first word to the first phrase.
                const combinedPhrase = `${phrases[i]}${phrases[j].substring(firstLastWords[j][0].length)}`;
                // Add the combined phrase to the set to ensure uniqueness.
                combinedPhrasesSet.add(combinedPhrase);
            }
        }
    }

    // Convert the set to an array and sort it in lexicographical order.
    return Array.from(combinedPhrasesSet).sort();
}
```

```python
class Solution:
    def beforeAndAfterPuzzles(self, phrases):
        # Store the first and last word of each phrase
        first_and_last_words = []
        for phrase in phrases:
            words = phrase.split()
            first_and_last_words.append((words[0], words[-1]))

        num_phrases = len(first_and_last_words)  # The number of phrases
        results = []  # Store the results

        # Loop through all combinations of phrases
        for i in range(num_phrases):
            for j in range(num_phrases):
                # Check if i and j are not the same, and the last word of phrase i matches the first word of phrase j
                if i != j and first_and_last_words[i][1] == first_and_last_words[j][0]:
                    # Append the combined phrase, removing the duplicate word, to the results
                    combined_phrase = phrases[i] + phrases[j][len(first_and_last_words[j][0]):]
                    results.append(combined_phrase)

        # Sort the results and remove duplicates
        return sorted(set(results))
```

Time and Space Complexity

Time Complexity:

The time complexity of the algorithm is determined by several nested operations.

1. Splitting phrases into tuples of first and last words: This operation is `O(n * m)`, where `n` is the average length of a phrase, and `m` is the total number of phrases. Splitting a string into words is an `O(n)` operation (since it depends on the length of the phrase), and it is done for each phrase.
2. Double `for` loop to compare each phrase with every other phrase: There are `n` phrases, and each phrase is compared with every other phrase, resulting in `O(n^2)` comparisons.
3. Concatenation of phrases when a match is found: The worst-case time complexity of string concatenation in Python can be `O(k)` where `k` is the length of the resulting string. Since the length of the phrases can vary, also considering the worst-case scenario where all phrases are concatenated, this step can contribute significantly to the time complexity. However, in practice for large strings, Python's string concatenation is usually more efficient due to optimizations. Therefore, we can consider it to be relatively constant for this analysis.
4. Removal of the first word from the second phrase when concatenating: This operation has a time complexity of `O(m)`, but since it's only done when a match is found and is part of the concatenation process, its impact is included in the previous step.
5. Sorting the final list of answers: Sorting a list of `r` results has a time complexity of `O(r log r)`. The value of `r` can be at most `n * (n - 1) / 2`, in the case where each phrase can be combined with every other phrase, which simplifies to `O(n^2/2) = O(n^2 log n^2/2) = O(n^2 log n)`.

Considering all the aforementioned steps, the dominant term for the time complexity is the double loop with the potential concatenation of phrases: `O(n^2 * k) + O(n^2 log n)`. The overall time complexity in terms of big-O notation is `O(n^2 * k + n^2 log n)`; however, we usually represent the time complexity with the most significant term, which is `O(n^2 * k)` if `k` is significantly large, or `O(n^2 log n)` if phrase concatenation is not a dominant factor (assuming strings are concatenated efficiently).

Space Complexity:

1. Intermediate list `ps` of tuples: It has a space complexity of `O(n)` since it stores a tuple for each phrase.
2. The `ans` list: This list will store the combined phrases. In the worst case, where every phrase can be combined with every other phrase, the number of combinations is `n * (n - 1) / 2`, leading to a space complexity of `O(n^2)`.
3. Set used in `sorted(set(ans))`: Converting the list `ans` to a set to eliminate duplicates, and then sorting it also requires `O(n^2)` space in the worst case. The sorted list will also occupy `O(n^2)` space.

With both the intermediate list and the final answer list considered, the predominant term for space complexity is `O(n^2)` for storing the potential phrase combinations.