937. Reorder Data in Log Files

Medium Array String Sorting

Problem Description

In this problem, we are dealing with an array of strings called `logs`, where each log entry is composed of an identifier followed by either a string of lowercase English letters (a letter-log) or a string of digits (a digit-log). The identifier and the log content are separated by a space. Our task is to reorder these logs based on certain rules:

1. Letter-logs must be sorted and come before all digit-logs.
2. Letter-logs should be ordered lexicographically (i.e., alphabetically) by their content. If two letter-logs have identical content, they should then be sorted by their identifiers in lexicographic order.
3. Digit-logs should retain their original order as they appeared in the input.

The expected output is the reordered array of the logs following these rules.

Intuition

To solve this, we need a way to differentiate between letter-logs and digit-logs and then sort them according to the given criteria. We can achieve this through a custom sorting function.

When we look at the identifier and the log content, we observe that letter-logs contain alphabetical characters while digit-logs contain numbers. We can exploit this characteristic to split logs into two categories by checking if the first character of the log content is a letter or a digit.

Given the sorting requirements, we know that all letter-logs should be sorted lexicographically by their content, and if these are equal, by their identifiers. Digit-logs do not need sorting among themselves, but they need to be placed after all letter-logs. To keep them in their original order, we can just ignore their content and identifiers during the sorting process.

With these observations, we craft a sorting function that:

1. Splits each log into its identifier and content.
2. Checks whether it's a letter-log or digit-log by inspecting the first character of the content.
3. If it's a letter-log, the function returns a tuple `(0, content, identifier)`. The zero indicates that it's a letter-log for sort precedence, and sorting by content and then identifier follows naturally.
4. If it's a digit-log, the function returns a tuple `(1,)`. The `1` ensures that all digit-logs are ordered after letter-logs, and since no other sorting keys follow, they maintain their relative input order.

Finally, by calling the `sorted` function and passing our custom comparison function, we obtain a sorted list that meets all the criteria outlined in the problem description. The `sorted` function rearranges the elements according to the tuples returned by our comparison function `cmp`, effectively pushing letter-logs to the front and sorting them while keeping digit-logs in their original order at the end of the resulting list.

Solution Approach

The implementation of the solution primarily revolves around the Python `sorted` function and a custom sorting key function, which is the `cmp` function in the reference solution. The `sorted` function is an efficient sorting algorithm that uses the TimSort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

The `cmp` function is where the distinction between letter-logs and digit-logs is made and the sorting logic is customized. It takes a log entry as an argument and uses the `split` method to divide it into two parts, the identifier and the content, with a space being the delimiter. Since we need both parts separately for the sorting criteria, splitting them does the job efficiently.

Here's the step-by-step approach within the `cmp` function:

1. Split the log into identifier `a` and content `b`:

```
a, b = x.split(' ', 1)
```

2. Check whether the log is a letter-log or a digit-log:

```
if b[0].isalpha():
    ...
else:
    ...
```

3. For letter-logs, return a tuple `(0, b, a)`, where `b` is the content of the log without the identifier and `a` is the identifier. The leading `0` ensures these come before any digit-logs.
4. For digit-logs, return a tuple `(1,)`, which only has the digit `1`. It ensures digit-logs are placed after letter-logs and since no other values are in the tuple, their original order (relative to other digit-logs) is maintained.

The `sorted` function uses this `cmp` key function to sort the entire list of logs. The logs are compared based on the tuples returned by `cmp`; if the first elements of these tuples are different (which differentiate between letter and digit logs), it sorts based on this. If they are the same (as with all letter-logs), it proceeds to sort based on the subsequent elements in the tuple, which are the content and, if necessary, the identifier.

To sum up, the algorithm utilizes a sorting key function to create sortable tuples for each log entry, which Python's sorted function then utilizes to sort the entire list accordingly. This approach does not require any additional data structures or complex algorithms, making it an efficient and elegant solution to the problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following `logs` array:

```
logs = ["dig1 8 1 5 1", "let1 art can", "dig2 3 6", "let2 own kit dig", "let3 art zero"]
```

We are tasked with reordering these logs based on the rules provided in the problem description.

Step 1: Differentiating logs

Firstly, we want to differentiate the logs into letter-logs and digit-logs. We can see that "let1 art can", "let2 own kit dig", and "let3 art zero" are letter-logs because they contain letters after the first space. "dig1 8 1 5 1" and "dig2 3 6" are digit-logs because they contain numbers after the first space.

Step 2: Applying the custom sorting function

We implement a custom sorting key function, `cmp`, to sort the logs. This function will handle logs differently based on their type.

Letter-logs

For letter-logs, `cmp` will return a tuple of the form `(0, content, identifier)`. Here's how it will look for our letter-logs in python code:

```
# "let1 art can" splits into identifier "let1" and content "art can"
("let1 art can".split(" ", 1)) => ("let1", "art can") => (0, "art can", "let1")

# "let2 own kit dig" splits into identifier "let2" and content "own kit dig"
("let2 own kit dig".split(" ", 1)) => ("let2", "own kit dig") => (0, "own kit dig", "let2")

# "let3 art zero" splits into identifier "let3" and content "art zero"
("let3 art zero".split(" ", 1)) => ("let3", "art zero") => (0, "art zero", "let3")
```

Digit-logs

For digit-logs, `cmp` will return a tuple just containing `1`, which places all digit-logs after the letter-logs.

```
# "dig1 8 1 5 1" splits but we only need to recognize it as a digit-log
("dig1 8 1 5 1".split(" ", 1)) => ("dig1", "8 1 5 1") => (1,)

# "dig2 3 6" also splits but we only need to mark it as a digit-log
("dig2 3 6".split(" ", 1)) => ("dig2", "3 6") => (1,)
```

Step 3: Sorting logs

After applying the `cmp` function, the `sorted` function sorts the logs based on these tuples:

```
# Letter-logs sorted by content and then by identifier if needed
sorted_letter_logs = sorted(["let1 art can", "let2 own kit dig", "let3 art zero"], key=cmp)

# Output for letter-logs is: ["let3 art zero", "let1 art can", "let2 own kit dig"]

# Digit-logs maintain their original order
# Therefore, after letter-logs they are just appended: ["dig1 8 1 5 1", "dig2 3 6"]

# Combined sorted logs
sorted_logs = sorted_letter_logs + ["dig1 8 1 5 1", "dig2 3 6"]

# Final output: ["let3 art zero", "let1 art can", "let2 own kit dig", "dig1 8 1 5 1", "dig2 3 6"]
```

This final output respects all the sorting rules mentioned in the problem description and uses the power of Python's `sorted` function along with a custom comparison key to effectively reorder the logs.

Solution Implementation

Python

```
from typing import List

class Solution:
    def reorderLogFiles(self, logs: List[str]) -> List[str]:
        def get_log_key(log: str) -> tuple:
            # Split each log into two parts: identifier and content
            identifier, content = log.split(' ', 1)

            # We return a tuple that will help us sort the logs:
            # If it's a letter-log (i.e., content starts with an alphabetic character),
            # we want it sorted by (0, content, identifier) to ensure that it appears
            # before any digit-logs and that letter-logs are sorted firstly by content
            # and secondly by identifier if there's a tie in content.
            # For digit-logs (content starts with a digit), we use (1,)
            # so use (1,) so that it appears after all the letter-logs,
            # and within the digit-logs subarray, it maintains the original order
            # Since we don't provide any further sorting keys for digit-logs,
            return (0, content, identifier) if content[0].isalpha() else (1,)

        # Sort logs using sort() function, the key parameter takes a function
        # that extracts sorting keys from the log entries.
        # The logs are sorted according to the keys provided by the get_log_key function.
        return sorted(logs, key=get_log_key)
```

Java

```
class Solution {
    public String[] reorderLogFiles(String[] logs) {
        // Sort the given logs with specified ordering rules
        Arrays.sort(logs, this::compareLogs);
        return logs;
    }

    private int compareLogs(String log1, String log2) {
        // Split each log into the words (identifier and content
        String[] split1 = log1.split(" ", 2);
        String[] split2 = log2.split(" ", 2);

        // Check if log1 and log2 contents start with a digit
        boolean isDigit1 = Character.isDigit(split1[1].charAt(0));
        boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

        // Both logs are letter-logs
        if (!isDigit1 && !isDigit2) {
            // Compare the contents of the logs
            int contentComparison = split1[1].compareTo(split2[1]);
            if (contentComparison != 0) {
                return contentComparison;
            }
            // If contents are identical, compare the identifiers
            return split1[0].compareTo(split2[0]);
        }

        // Both logs are digit-logs, in which case we return 0 to retain their original order
        if (isDigit1 && isDigit2) {
            return 0;
        }

        // One log is a digit-log and the other is a letter-log
        // Letter-logs should come before digit-logs, thus returning 1 if the first log is a digit-log, else -1
        return isDigit2 ? 1 : -1;
    }
}
```

C++

```
#include <vector>
#include <string>
#include <algorithm>

// Function to reorder log files based on specified rules
std::vector<std::string> reorderLogFiles(std::vector<std::string>& logs) {
    // Lambda function to check if a character is a digit
    auto isDigit = [](char c) { return c >= '0' && c <= '9'; };

    // Custom sort function for the logs
    std::sort(logs.begin(), logs.end(), [&](const std::string& log1, const std::string& log2) {
        // Find the first space in both logs to separate identifier from content
        size_t firstSpaceLog1 = log1.find(' ');
        size_t firstSpaceLog2 = log2.find(' ');

        // Check if the first character of the content of log1 and log2 is a digit
        bool isDigitLog1 = isDigit(log1[firstSpaceLog1 + 1]);
        bool isDigitLog2 = isDigit(log2[firstSpaceLog2 + 1]);

        // If both logs are digit logs, maintain their relative order
        if (isDigitLog1 && isDigitLog2) {
            return false;
        }

        // If log1 is a digit log, then it should come after log2 if log2 is a letter log
        if (isDigitLog1) {
            return false;
        }

        // If log2 is a digit log, then it should come after log1 if log1 is a letter log
        if (isDigitLog2) {
            return true;
        }

        // If both logs are letter logs, compare their contents excluding identifier
        std::string contentLog1 = log1.substr(firstSpaceLog1 + 1);
        std::string contentLog2 = log2.substr(firstSpaceLog2 + 1);

        // If the contents are equal, sort based on the lexicographical order of the entire log
        if (contentLog1 == contentLog2) {
            return log1 < log2;
        }

        // Otherwise, sort based on the lexicographical order of just the contents
        return contentLog1 < contentLog2;
    });

    // Return the reordered logs
    return logs;
}
```

TypeScript

```
function reorderLogFiles(logs: string[]): string[] {
    // Function to check if a character is a digit
    const isDigit = (char: string): boolean => char >= '0' && char <= '9';

    // Custom sort function for the logs
    return logs.sort((log1, log2) => {
        // Check last character of log1 and log2 to determine if they are digit logs
        const lastCharLog1 = log1[log1.length - 1];
        const lastCharLog2 = log2[log2.length - 1];

        // If both logs are digit logs, maintain their relative order
        if (isDigit(lastCharLog1) && isDigit(lastCharLog2)) {
            return 0;
        }

        // If log1 is a digit log, it should come after log2 if log2 is a letter log
        if (isDigit(lastCharLog1)) {
            return 1;
        }

        // If log2 is a digit log, it should come after log1 if log1 is a letter log
        if (isDigit(lastCharLog2)) {
            return -1;
        }

        // If both logs are letter logs, compare based on content excluding identifier
        const contentLog1 = log1.split(' ').slice(1).join(' ');
        const contentLog2 = log2.split(' ').slice(1).join(' ');

        // If contents are equal, sort based on lexicographical order of the entire log
        if (contentLog1 === contentLog2) {
            return log1.localeCompare(log2); // Changed to use standard localeCompare for string comparison
        }

        // Otherwise, sort based on the lexicographical order of just the content
        return contentLog1.localeCompare(contentLog2); // Changed to use standard localeCompare for string comparison
    });
}
```

```
from typing import List

class Solution:
    def reorderLogFiles(self, logs: List[str]) -> List[str]:
        def get_log_key(log: str) -> tuple:
            # Split each log into two parts: identifier and content
            identifier, content = log.split(' ', 1)

            # We return a tuple that will help us sort the logs:
            # If it's a letter-log (i.e., content starts with an alphabetic character),
            # we want it sorted by (0, content, identifier) to ensure that it appears
            # before any digit-logs and that letter-logs are sorted firstly by content
            # and secondly by identifier if there's a tie in content.
            # For digit-logs (content starts with a digit),
            # we use (1,) so that it appears after all the letter-logs,
            # and within the digit-logs subarray, it maintains the original order
            # Since we don't provide any further sorting keys for digit-logs,
            return (0, content, identifier) if content[0].isalpha() else (1,)

        # Sort logs using sort() function. The key parameter takes a function
        # that extracts sorting keys from the log entries.
        # The logs are sorted according to the keys provided by the get_log_key function.
        return sorted(logs, key=get_log_key)
```

Time and Space Complexity

The time complexity of the given code is $O(N \log N)$, where N is the number of logs. This arises from the use of the sorted function, which typically has $O(N \log N)$ time complexity for sorting an array. Specifically, every comparison between two logs requires string comparison, and in the worst case, each comparison can take $O(M)$ time, where M is the maximum length of a log file. However, assuming M is relatively small and therefore the comparison time is constant, the overall time complexity still remains $O(N \log N)$.

The space complexity of the code is $O(N)$. This is because the sorting algorithm used in Python (Timsort) is a hybrid stable sorting algorithm derived from merge sort and insertion sort characterizes $O(N)$ space complexity. This space is needed to hold the intermediate results for the sorted array. Furthermore, the extra space needed for the lambda function's return values is not significant compared to the space needed for these intermediate results as they do not depend on the number of logs N.