<https://leetcode.com/problems/reorder-data-in-log-files/description/>

You are given an array of logs. Each log is a space-delimited string of words, where the first word is the identifier.

There are two types of logs:

Letter-logs: All words (except the identifier) consist of lowercase English letters.

Digit-logs: All words (except the identifier) consist of digits.

Reorder these logs so that:

The letter-logs come before all digit-logs.

The letter-logs are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.

The digit-logs maintain their relative ordering.

Return the final order of the logs.

**Example 1:**

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

**Output:** ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]

**Explanation:**

The letter-log contents are all different, so their ordering is "art can", "art zero", "own kit dig".

The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".

**Example 2:**

**Input:** logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]

**Output:** ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]

**Constraints:**

1 <= logs.length <= 100

3 <= logs[i].length <= 100

All the tokens of logs[i] are separated by a **single** space.

logs[i] is guaranteed to have an identifier and at least one word after the identifier.

**Attempt 1: 2025-02-25**

**Solution 1: Array + Sorting (30 min)**

class Solution {

    public String[] reorderLogFiles(String[] logs) {

        Arrays.sort(logs, new Comparator<String>() {

            @Override

            public int compare(String log1, String log2) {

                // Split each log into two parts: identifier and content

                String[] split1 = log1.split(" ", 2);

                String[] split2 = log2.split(" ", 2);

                boolean isDigit1 = Character.isDigit(split1[1].charAt(0));

                boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

                // Both logs are letter logs

                if(!isDigit1 && !isDigit2) {

                    // First compare the content

                    int cmp = split1[1].compareTo(split2[1]);

                    if(cmp != 0) {

                        return cmp;

                    }

                    // If content is the same, compare the identifiers

                    return split1[0].compareTo(split2[0]);

                // First log is letter log, second log is digit log

                // The letter log come before all digit log

                } else if(!isDigit1 && isDigit2) {

                    return -1;

                // First log is digit log, second log is letter log

                // The digit log comes after the letter log

                } else if(isDigit1 && !isDigit2) {

                    return 1;

                // Both logs are digit logs

                // Both logs are digit logs, maintain their original order

                } else {

                    return 0;

                }

            }

        });

        return logs;

    }

}

Time Complexity: O(M⋅N⋅logN)

First of all, the time complexity of the Arrays.sort() is O(N⋅logN), as stated in the API specification,

which is to say that the compare() function would be invoked O(N⋅logN) times.

For each invocation of the compare() function, it could take up to O(M) time, since we compare the contents of the logs.

Therefore, the overall time complexity of the algorithm is O(M⋅N⋅logN).

Space Complexity: O(M⋅logN)

For each invocation of the compare() function, we would need up to O(M) space to hold the parsed logs.

In addition, since the implementation of Arrays.sort() is based on quicksort algorithm whose space complexity is O(logn),

assuming that the space for each element is O(1)).

Since each log could be of O(M) space, we would need O(M⋅logN) space to hold the intermediate values for sorting.

In total, the overall space complexity of the algorithm is O(M+M⋅logN)=O(M⋅logN)

**Refer to**

<https://leetcode.com/problems/reorder-data-in-log-files/editorial/>

**Overview**

First of all, let us put aside the debate whether this problem is an easy or medium one.

The problem is a good exercise to practice the technique of **custom sort** in different languages.

The idea of custom sort is that we don't have to rewrite a sorting algorithm every time we have a different ***sorting criteria*** among the elements.

Each language provides certain interface that allows us to **customize** the sorting criteria of the sorting functions, so that we can reuse the implementation of sorting in different scenarios.

In this article, we will present two ways to specify the sorting order, namely by **comparator** and by **sorting key**.

**Approach 1: Comparator**

**Intuition**

Given a list of elements [*e*1​,*e*2​,*e*3​], regardless of the content of the elements, the first way to specify the *order* among the elements is to define the **pairwise** < (*"less than"*) **relationship** globally.

For instance, for the above example, we could define the **relationships** as *e*3​<*e*2​, *e*2​<*e*1​.

Then if we are asked to sort the list in the *ascending* order, the result would be [*e*3​,*e*2​,*e*1​].

**Note:** normally we should define all pairwise relationships among all elements, but due to the transitive property, we omit certain relationships that can be deduced from others, *e.g.* (*e*3​<*e*2​,*e*2​<*e*1​)→(*e*1​<*e*3​)

If we ever change the *order*, *e.g.* *e*1​<*e*3​, *e*3​<*e*2​, the final *sorted* result would be changed accordingly, *i.e.* [*e*1​,*e*3​,*e*2​].

**Algorithm**

The above pairwise *"less than"* relationship is also known as **comparator** in Java, which is a function object that helps the sorting functions to determine the orders among a collection of elements.

We show the [definition of the comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface as follows:

int compare(T o1, T o2) {

if (o1 < o2)

return -1;

else if (o1 == o2)

return 0;

else // o1 > o2

return 1;

}

As we discussed before, once we define the pairwise relationship among the elements in a collection, the **total order** of the collection is then fixed.

Now, what we need to do is to define our own proper **comparator** according to the description of the problem.

We can translate the problem into the following rules:

1). The *letter-logs* should be prioritized above all *digit-logs*.

2). Among the *letter-logs*, we should further sort them firstly based on their **contents**, and then on their **identifiers** if the contents are identical.

3). Among the *digit-logs*, they should remain in the same order as they are in the collection.

One can then go ahead and implement the comparator based on the above rules. Here is an example.

class Solution {

public String[] reorderLogFiles(String[] logs) {

Comparator<String> myComp = new Comparator<String>() {

@Override

public int compare(String log1, String log2) {

// split each log into two parts: <identifier, content>

String[] split1 = log1.split(" ", 2);

String[] split2 = log2.split(" ", 2);

boolean isDigit1 = Character.isDigit(split1[1].charAt(0));

boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

// case 1). both logs are letter-logs

if (!isDigit1 && !isDigit2) {

// first compare the content

int cmp = split1[1].compareTo(split2[1]);

if (cmp != 0)

return cmp;

// logs of same content, compare the identifiers

return split1[0].compareTo(split2[0]);

}

// case 2). one of logs is digit-log

if (!isDigit1 && isDigit2)

// the letter-log comes before digit-logs

return -1;

else if (isDigit1 && !isDigit2)

return 1;

else

// case 3). both logs are digit-log

return 0;

}

};

Arrays.sort(logs, myComp);

return logs;

}

}

**Stable Sort**

One might notice that in the above implementation one can find the logic that corresponds each of the rules, except the **Rule (3)**.

Indeed, we did not do anything explicitly to ensure the order imposed by the Rule (3).

The short answer is that the Rule (3) is ensured *implicitly* by an important property of sorting algorithms, called **stability**.

It is stated as "stable sorting algorithms sort equal elements in the same order that they appear in the input."

Not all sort algorithms are *stable*, *e.g.* ***merge sort*** is stable.

The Arrays.sort() interface that we used is stable, as one can find in the [specification](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html).

Therefore, the Rule (3) is implicitly respected thanks to the stability of the sorting algorithm that we used.

**Complexity Analysis**

Let *N* be the number of logs in the list and *M* be the maximum length of a single log.

Time Complexity: O(*M*⋅*N*⋅log*N*)

First of all, the time complexity of the Arrays.sort() is O(*N*⋅log*N*), as stated in the [API specification](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-), which is to say that the compare() function would be invoked O(*N*⋅log*N*) times.

For each invocation of the compare() function, it could take up to O(*M*) time, since we compare the contents of the logs.

Therefore, the overall time complexity of the algorithm is O(*M*⋅*N*⋅log*N*).

Space Complexity: O(*M*⋅log*N*)

For each invocation of the compare() function, we would need up to O(*M*) space to hold the parsed logs.

In addition, since the implementation of Arrays.sort() is based on quicksort algorithm whose space complexity is O(log*n*), assuming that the space for each element is O(1)).

Since each log could be of O(*M*) space, we would need O(*M*⋅log*N*) space to hold the intermediate values for sorting.

In total, the overall space complexity of the algorithm is O(*M*+*M*⋅log*N*)=O(*M*⋅log*N*)

**Refer to Deepseek**

Certainly! The problem "Reorder Data in Log Files" on LeetCode requires you to reorder the logs such that all letter-logs come before digit-logs. The letter-logs should be sorted lexicographically, and if they have the same content, they should be sorted by their identifiers. The digit-logs should maintain their original order.

Here's a Java solution for the problem:

class ReorderLogFiles {

public String[] reorderLogFiles(String[] logs) {

Arrays.sort(logs, new Comparator<String>() {

@Override

public int compare(String log1, String log2) {

// Split each log into two parts: identifier and content

String[] split1 = log1.split(" ", 2);

String[] split2 = log2.split(" ", 2);

boolean isDigit1 = Character.isDigit(split1[1].charAt(0));

boolean isDigit2 = Character.isDigit(split2[1].charAt(0));

// Case 1: Both logs are letter-logs

if (!isDigit1 && !isDigit2) {

// First compare the content

int cmp = split1[1].compareTo(split2[1]);

if (cmp != 0) {

return cmp;

}

// If content is the same, compare the identifiers

return split1[0].compareTo(split2[0]);

}

// Case 2: One of the logs is a digit-log

if (!isDigit1 && isDigit2) {

// The letter-log comes before the digit-log

return -1;

} else if (isDigit1 && !isDigit2) {

// The digit-log comes after the letter-log

return 1;

} else {

// Case 3: Both logs are digit-logs, maintain their original order

return 0;

}

}

});

return logs;

}

public static void main(String[] args) {

ReorderLogFiles solution = new ReorderLogFiles();

String[] logs = {

"dig1 8 1 5 1",

"let1 art can",

"dig2 3 6",

"let2 own kit dig",

"let3 art zero"

};

String[] result = solution.reorderLogFiles(logs);

for (String log : result) {

System.out.println(log);

}

}

}

**Explanation:**

Splitting the Logs: Each log is split into two parts: the identifier and the content. This is done using split(" ", 2), which splits the string into two parts based on the first space.

Identifying Digit-Logs and Letter-Logs: We check if the first character of the content is a digit or a letter using Character.isDigit().

Sorting Logic:

If both logs are letter-logs, we compare their content. If the content is the same, we compare their identifiers.

If one log is a letter-log and the other is a digit-log, the letter-log comes first.

If both logs are digit-logs, they maintain their original order.

Returning the Result: The sorted logs are returned as the final result.

**Example Output:**

For the input:

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

The output will be:

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

This solution efficiently sorts the logs according to the given rules.