2788. Split Strings by Separator

Easy Array String

Leetcode Link

Problem Description

In this problem, we have an input array `words` which consists of strings, and a `separator`, which is a single character. Our task is to take each string in `words` and split it into substrings based on where the `separator` occurs. Once we split the strings, we need to exclude any empty substrings that may result from this process. The output should be a new array of strings which represents the initial strings split by the separator, maintaining the original order of the strings in `words`. Importantly, the `separator` itself should not be included in the resulting strings.

Intuition

The intuition behind the solution is to use the built-in string function `split()` which is available in Python. The `split()` function takes a character or substring and splits the string at every occurrence of this character or substring. This works perfectly for our use case with the `separator`. The idea is to iterate through each string in the `words` array, apply the `split(separator)` function to divide it into substrings, and collect the resulting substrings in the new output array. However, because we should not include any empty strings in the output, we add a condition to only include substrings that are not empty (`if s`).

This approach allows us to construct the desired output array with each string being split by the `separator` and empty strings being excluded, all while keeping the order of the originally provided strings intact.

Solution Approach

The solution leverages the power of list comprehension in Python, which is a compact way to process elements in a list and construct a new list based on certain criteria. There are three main parts to our list comprehension:

1. Iterating over each word in the input `words` list.
2. Splitting each word by the given `separator` to create a list of substrings.
3. Filtering out any resulting empty strings.

These steps can be broken down as follows:

- **Iteration**: The outer loop `for w in words` goes through each string `w` present in the input list `words`.
- **Split Operation**: For each string `w`, `w.split(separator)` is called. This function returns a list of substrings obtained by splitting `w` at every occurrence of the character `separator`.
- **Filtering Empty Strings**: Inside the list comprehension, after the split operation, there's another loop `for s in w.split(separator)`. Here, `s` represents each substring result from the split. The `if s` part ensures that only non-empty `s` (substrings) get included in the final list. This filtering is important to meet the problem's requirement to exclude empty strings.
- **Creating the Output List**: The list comprehension `[s for w in words for s in w.split(separator) if s]` then constructs the output list. This list includes all non-empty substrings obtained from splitting each of the strings in `words` by the `separator`.

Code Analysis: The given solution code uses no additional data structures other than the list to hold the final result. It's a direct implementation of string manipulation methods and list comprehension, which are very efficient for this kind of task. The overall time complexity is O(n * m), where n is the number of strings in the input list and m is the average length of the strings, assuming `split()` is O(m) in time.

In conclusion, this implementation is a straightforward and effective way of achieving the desired functionality using Python's string manipulation capabilities and list comprehension.

Example Walkthrough

Let's walk through the provided solution approach with a small example to better understand how it works. Suppose we have the following inputs:

- `words`: ["apple,berry,cherry", "lemon##lime", "banana--melon"]
- `separator`: "-"

We want our output to exclude the separator and not to include any empty strings that may come from split operation. Here's how the solution approach applies to our example:

Step 1: Iterating over each word We start by taking the first string in the `words` list which is "apple,berry,cherry". Since our separator is "-", this string does not get split as it does not contain the separator.

Step 2: Splitting each word We then move to the second string "lemon##lime". Again, this does not contain our separator, so it remains unsplit.

Step 3: Filtering out empty strings Next, we take the third string "banana--melon". Using the `split('-')` function on this string gives us ["banana", "", "melon"]. We find that there is an empty string present as a result of the split operation where two separators were adjacent.

Step 4: Applying the list comprehension As per the list comprehension `[s for w in words for s in w.split(separator) if s]`, we construct a new list by including each non-empty substring. For our example, the list comprehension step would look like this:

```
result = [s for w in ["apple,berry,cherry", "lemon##lime", "banana--melon"] for s in w.split("-") if s]
```

This will iterate over each word, split it by the separator, and only include the non-empty substrings. In this case, the first two words won't get split because they do not contain the separator. In the third word, "banana--melon", it will split into ["banana", "", "melon"] and exclude the empty string.

The final output array after applying the list comprehension to our example would therefore be:

```
["apple,berry,cherry", "lemon##lime", "banana", "melon"]
```

Conclusion: The initial strings are split wherever the separator is encountered, and any strings that would have been empty after the split are removed from the output. The order of the elements is preserved from the original list to the output list.

Python Solution

```python
from typing import List

class Solution:
    def splitWordsBySeparator(self, words: List[str], separator: str) -> List[str]:
        # Initialize an empty list to store the split words
        split_words = []

        # Iterate through each word in the input list 'words'
        for word in words:
            # Split the word by the given separator
            word_parts = word.split(separator)
            # Append each non-empty part of the split word to 'split_words'
            for part in word_parts:
                if part:  # Ensure to add only non-empty strings
                    split_words.append(part)

        # Return the list containing all split, non-empty words
        return split_words

```

Java Solution

```java
import java.util.regex.Pattern; // Importing the Pattern class for regex operations.
import java.util.ArrayList; // Importing the ArrayList class for dynamic array operations.
import java.util.List; // Importing the List interface for using lists.

// Class name should be capitalized and descriptive
class WordSplitter {

    // Method to split strings in a list by a given separator and return a list of substrings
    public List<String> splitWordsBySeparator(List<String> words, char separator) {
        // Creating a list to store the resulting substrings
        List<String> splitWords = new ArrayList<>();

        // Iterating through each string in the list of words
        for (String word : words) {
            // Splitting the current word by the separator and escaping it if it's a special regex character
            String[] parts = word.split(Pattern.quote(String.valueOf(separator)));

            // Adding each non-empty part to the result list
            for (String part : parts) {
                if (!part.isEmpty()) {
                    splitWords.add(part);
                }
            }
        }

        // Returning the list of split substrings
        return splitWords;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Splits words in the vector by a specified separator and
    // returns a vector of the subsequently separated strings.
    // Empty strings resulting from the split are not included in the results.
    std::vector<std::string> splitWordsBySeparator(std::vector<std::string>& words, char separator) {
        std::vector<std::string> results;

        // Iterate over each word in the input vector
        for (auto& word : words) {
            // Split the word by the provided separator
            std::vector<std::string> splitParts = split(word, separator);

            // Add non-empty parts to the result vector
            for (auto& part : splitParts) {
                if (!part.empty()) {
                    results.emplace_back(part);
                }
            }
        }

        return results;
    }

    // Splits a string by a given delimiter and returns a vector of the parts.
    std::vector<std::string> split(std::string inputString, char delimiter) {
        std::vector<std::string> parts;
        std::stringstream stream(inputString);
        std::string segment;

        // Use getline to split the string by the delimiter and add each part to the vector
        while (getline(stream, segment, delimiter)) {
            parts.push_back(segment);
        }

        return parts;
    }
};

```

Typescript Solution

```typescript
// This function takes an array of strings and a separator, then splits each
// string in the array by the separator, and returns a new array with the split segments.
// Empty strings resulted from the split are excluded from the result.
//
// @param words - An array of strings to be split.
// @param separator - A string representing the separator to be used for splitting the words.
// @returns An array of strings containing the non-empty segments after splitting.
function splitWordsBySeparator(words: string[], separator: string): string[] {
    // Initialize an array to store the result segments.
    const splitWords: string[] = [];

    // Loop through each word in the input array.
    for (const word of words) {
        // Split the current word by the given separator.
        const splits = word.split(separator);

        // Loop through the split segments of the current word.
        for (const split of splits) {
            // Add the non-empty segments to the result array.
            if (split.length > 0) {
                splitWords.push(split);
            }
        }
    }

    // Return the result array containing all non-empty segments.
    return splitWords;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by the number of operations it needs to perform in relation to the input size. Here, we consider both the number of words `n` and the average length of the words `m`.

The function consists of a list comprehension with a nested loop - for every word `w` in `words`, it performs a `.split(separator)` operation. The `.split(separator)` operation is `O(m)` where `m` is the length of the word being split. This is because `.split()` goes through each character in the word `w` to check if it matches the `separator`.

After splitting, the words are iterated over again to filter out empty strings. This operation can potentially include each character from the previous step, which keeps the complexity at the same level - `O(m * n)`.

Therefore, the overall time complexity is `O(m * n)`, where `n` is the number of words and `m` is the average length of those words.

Space Complexity

The space complexity of the function depends on the space required to store the output list.

In the worst case, if the separator is not present in any of the words, then the list comprehension will include all the original words. In this case, the space complexity is equal to the total number of characters in all words (plus the space required for list overhead), which is `O(m * n)`.

However, if the separator is present, then additional strings will be added to the output for each split occurrence. In the absolute worst case, if every character is a separator, the size of the output list could be up to twice the number of characters (since you get an empty string before and after each character), which leads to a space complexity of `O(2 * m * n)`. Since constant factors are dropped in Big O notation, this also simplifies to `O(m * n)`.

So, the space complexity is also `O(m * n)` under the assumption that the separator can be arbitrarily often in the words.