# 1078. Occurrences After Bigram

Easy String

## Problem Description

The problem is about searching for a specific pattern in a string of text. This pattern is formed by three consecutive words: `first`, `second`, and `third`. The words `first` and `second` are given as inputs, and the goal is to find the word `third` that immediately follows each occurrence of the sequence `first second` within a given block of text. You're asked to collect all these `third` words into an array and return it. It's important to note that the pattern should be in the correct order, and the words must directly follow one another with no other words in between.

For example, if the input text is "alice is alice is there and alice is here", `first` is "alice" and `second` is "is", you need to return all the words that come immediately after each "alice is", which are "there" and "here".

## Intuition

To solve this problem, the solution approach starts with breaking down the text into individual words. This is accomplished by using the `split()` function which divides the text into a list of words based on whitespace.

Next, we iterate over this list of words with a running index. In each iteration, we check if the current word, the word right after it, and the word following these two match the pattern of `first`, `second`, and then any `third`. We are interested in this third word only when the first two match the given input words.

To perform this check efficiently, we look at slices of three words at a time using the current index: `words[i : i + 3]`. If we find a match for `first` and `second`, we take the third word and add it to our answer list.

## Solution Approach

The implementation of the solution to this problem involves a straightforward algorithm that leverages basic data structures and the concept of string manipulation.

Firstly, the text is converted into a list of words, this is done using the `split()` function which is a standard method in Python for dividing strings into a list based on a delimiter, which in this case, is space.

```
words = text.split()
```

Once we have our list of words, our goal is to iterate through this list and identify consecutive triplets where the first and second words match the given inputs.

The primary data structure used in the implementation is a simple list to keep track of our answers:

```
ans = []
```

The iteration starts at the beginning of the list and continues until the third-to-last word, allowing us to look at triplets without going out of the list's bounds:

```
for i in range(len(words) - 2):
```

At each step of the iteration, we consider a slice of three consecutive words:

```
a, b, c = words[i : i + 3]
```

This makes `a` the current word, `b` the word following it, and `c` the one right after `b`. We then compare `a` and `b` to our input `first` and `second`:

```
if a == first and b == second:
```

If they match, it means we've found an occurrence of the pattern, and we append the third word `c` to our answers list:

```
    ans.append(c)
```

The loop continues until all valid triplets have been considered. The resulting `ans` list, which now contains all the `third` words that followed each `first second` pattern, is then returned:

```
return ans
```

No additional or complex data structures are required, and the algorithm runs with a time complexity of O(n), where n is the number of words in the text, as it requires a single pass through the list of words. The space complexity is O(m), where m is the count of valid `third` words, since we store each one in the `ans` list.

### Example Walkthrough

Let's apply the solution approach with an example. Suppose our input text is "the quick brown fox jumps over the lazy dog", and we're looking for the `first` word "quick" and the `second` word "brown." Our goal is to find the `third` word that comes right after each "quick brown" sequence in the text.

Step by step, the algorithm does the following:

1. Split the text into individual words:

```
words = "the quick brown fox jumps over the lazy dog".split()
# words = ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

2. Initialize an empty list to store the answers:

```
ans = []
```

3. Iterate over the list of words, stopping two words before the last word to ensure we can look at groups of three:

```
for i in range(len(words) - 2): # This loops from 0 to len(words) - 3
```

4. In each iteration, create a slice of three words and assign them to variables `a`, `b`, and `c`:

```
# Iteration 0: i = 0
a, b, c = words[0 : 0 + 3] # a = 'the', b = 'quick', c = 'brown'
# Iteration 1: i = 1
a, b, c = words[1 : 1 + 3] # a = 'quick', b = 'brown', c = 'fox'
# And so on...
```

5. Compare `a` and `b` with the `first` and `second` words. If they match, append `c` to the `ans` list:

```
# Iteration 1: a = 'quick', b = 'brown', c = 'fox'
if a == "quick" and b == "brown": # This condition is True
    ans.append(c) # ans = ['fox']
```

6. Continue the loop until we have checked every group of three words. For this example, only one match exists, and the loop ends after iteration 6 (last index checked is 6 because `len(words) - 2` is 7, and range is exclusive on the end):

```
# After the loop ends
return ans # ans = ['fox']
```

So, in this case, the answer list `ans` contains the word "fox," which is the word that follows "quick brown" in the given text.

Using this method, the algorithm efficiently identifies the words that follow each occurrence of a specified two-word sequence in linear time.

## Solution Implementation

### Python

```
class Solution:
    def findOccurrences(self, text: str, first: str, second: str) -> List[str]:
        # Split input text into a list of words
        words = text.split()
        # Initialize an empty list to store the third words following the first and second words
        third_words = []

        # Iterate through the list of words, stopping two words before the end
        for i in range(len(words) - 2):
            # Unpack the current triplet of words for easy comparison
            current_first, current_second, following_word = words[i : i + 3]
            # Check if the current first two words match the provided first and second words
            if current_first == first and current_second == second:
                # If yes, append the following word to the results list
                third_words.append(following_word)

        # Return the list of third words that follow the first and second words
        return third_words
```

### Java

```
class Solution {
    public String[] findOccurrences(String text, String first, String second) {
        // Split the input text into words.
        String[] words = text.split(" ");

        // Create a list to store the third words following the 'first' and 'second' words.
        List<String> thirdWordsList = new ArrayList<>();

        // Iterate through the words, stopping two words before the last to avoid out-of-bounds access.
        for (int i = 0; i < words.length - 2; ++i) {
            // Check if the current word is equal to 'first' and the next word is equal to 'second'.
            if (first.equals(words[i]) && second.equals(words[i + 1])) {
                // If the condition is met, add the word that comes after 'second' to the list.
                thirdWordsList.add(words[i + 2]);
            }
        }

        // Convert the list of third words to an array and return it.
        return thirdWordsList.toArray(new String[0]);
    }
}
```

### C++

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

class Solution {
public:
    // Function to find all occurrences of third word that immediately follow
    // the first and second words in the given text.
    std::vector<std::string> findOccurrences(std::string text, std::string first, std::string second) {
        // Create an input string stream from the given text
        std::istringstream inputStream(text);
        std::vector<std::string> words; // Vector to store all words from the text
        std::string word; // Variable to hold each word while extracting from stream

        // Read words from the stream and emplace them to the words vector
        while (inputStream >> word) {
            words.emplace_back(word);
        }

        std::vector<std::string> result; // Vector to store the result
        int numWords = words.size(); // Get the total number of words

        // Iterate over all words, stopping 2 words before the last
        for (int i = 0; i < numWords - 2; ++i) {
            // If the current and next word match 'first' and 'second', respectively
            if (words[i] == first && words[i + 1] == second) {
                // Add the word immediately following them to the result
                result.emplace_back(words[i + 2]);
            }
        }

        return result; // Return the final result vector
    }
};
```

### TypeScript

```
/**
 * This function finds all the occurrences of a triplet pattern in a sentence, where the first two
 * words in the triplet are given as inputs 'first' and 'second', and returns an array containing
 * the third words of those triplets.
 * @param text - The string text in which to search for the triplets.
 * @param first - The first word in the triplet sequence to match.
 * @param second - The second word in the triplet sequence to match.
 * @returns An array of the third words following each found 'first' and 'second' word pair.
 */
function findOccurrences(text: string, first: string, second: string): string[] {
    // Split the input text into an array of words.
    const words = text.split(' ');
    // Determine the number of words in the array.
    const wordCount = words.length;
    // Initialize an array to store the results.
    const matches: string[] = [];

    // Iterate through each word in the array until the second-to-last word.
    for (let i = 0; i < wordCount - 2; i++) {
        // Check if a sequence matches the 'first' and 'second' words.
        if (words[i] === first && words[i + 1] === second) {
            // If a match is found, add the third word to the results array.
            matches.push(words[i + 2]);
        }
    }

    // Return the array of matching third words.
    return matches;
}
```

```
class Solution:
    def findOccurrences(self, text: str, first: str, second: str) -> List[str]:
        # Split input text into a list of words
        words = text.split()
        # Initialize an empty list to store the third words following the first and second words
        third_words = []

        # Iterate through the list of words, stopping two words before the end
        for i in range(len(words) - 2):
            # Unpack the current triplet of words for easy comparison
            current_first, current_second, following_word = words[i : i + 3]
            # Check if the current first two words match the provided first and second words
            if current_first == first and current_second == second:
                # If yes, append the following word to the results list
                third_words.append(following_word)

        # Return the list of third words that follow the first and second words
        return third_words
```

## Time and Space Complexity

### Time Complexity

The time complexity of the given function is `O(n)`, where `n` is the number of words in the input string `text`. This complexity arises because the function goes through all the words in the text sequentially only once, examining if each sequence of three words starts with the `first` and `second` words accordingly. With the number of iterations equal to `len(words) - 2`, and because string splitting, comparisons, and appending to the list are all `O(1)` operations for each iteration, the overall time complexity remains linear.

### Space Complexity

The space complexity of the function is `O(m)`, where `m` is the number of triplets found that match the given pattern. This is because the space required is directly proportional to the number of third words "c" that follows a matching pair ("a", "b"). We also have additional `O(n)` space complexity from creating the `words` list where `n` is the number of words in the given text. However, since `m` is bounded by `n` (i.e., in the worst case, `m` equals `n-2` when every triplet in the text is a match), the dominant term is still `O(n)`. Therefore, we simplify this and express the overall space complexity as `O(n)`.