Solution Review: Problem Challenge 4

We'll cover the following

- Words Concatenation (hard)
- Solution
- Code
  - Time Complexity
  - Space Complexity

Words Concatenation (hard)#

Given a string and a list of words, find all the starting indices of substrings in the given string that are a **concatenation of all the given words** exactly once **without any overlapping** of words. It is given that all words are of the same length.

0% completed

Search Course

- Longest Substring with Same Letters after Replacement (hard) (/courses/grokking-the-coding-interview/R8DVgjq78yR)
- Longest Subarray with Ones after Replacement (hard) (/courses/grokking-the-coding-interview/B6VypRxPolJ)
- Problem Challenge 1 (/courses/grokking-the-coding-interview/N8vB7OVYo2D)
- Solution Review: Problem Challenge 1 (/courses/grokking-the-coding-interview/N0o9QnPLKNv)
- Problem Challenge 2 (/courses/grokking-the-coding-interview/YQ8N2OZq0VM)
- Solution Review: Problem Challenge 2 (/courses/grokking-the-coding-interview/xl2g3vxrMq3)
- Problem Challenge 3 (/courses/grokking-the-coding-

Example 1:

```
Input: String="catfoxcat", Words=
["cat", "fox"]
Output: [0, 3]
Explanation: The two substring co
ntaining both the words are "catf
ox" & "foxcat".
```

Example 2:

```
Input: String="catcatfoxfox", Wor
ds=["cat", "fox"]
Output: [3]
Explanation: The only substring c
ontaining both the words is "catf
ox".
```

Solution#

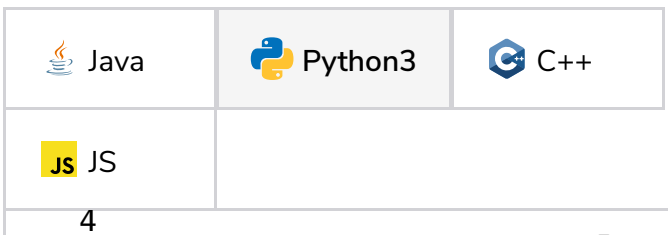
This problem follows the **Sliding Window** pattern and has a lot of similarities with Maximum Sum Subarray of Size K (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5177043027230720/). We will keep track of all the words in a **HashMap**

and try to match them in the given string. Here are the set of steps for our algorithm:

1. Keep the frequency of every word in a **HashMap**.
2. Starting from every index in the string, try to match all the words.
3. In each iteration, keep track of all the words that we have already seen in another **HashMap**.
4. If a word is not found or has a higher frequency than required, we can move on to the next character in the string.
5. Store the index if we have found all the words.

Code#

Here is what our algorithm will look like:



```
 6
 7  r word in words:
 8  if word not in word_frequency:
 9    word_frequency[word] = 0
10  word_frequency[word] += 1
11
12  sult_indices = []
13  rds_count = len(words)
14  rd_length = len(words[0])
15
16  r i in range((len(str1) - words_count
17  words_seen = {}
18  for j in range(0, words_count):
19    next_word_index = i + j * word_lengt
20    # Get the next word from the string
21    word = str1[next_word_index: next_wo
22    if word not in word_frequency:  # Br
23      break
24
25    # Add the word to the 'words_seen' m
26    if word not in words_seen:
27      words_seen[word] = 0
28    words_seen[word] += 1
29
30    # No need to process further if the
31    if words_seen[word] > word_frequency
```

Run Save Reset

Time Complexity#

The time complexity of the above algorithm will be $O(N * M * Len)$ where 'N' is the number of characters

in the given string, ‘M’ is the total number of words, and ‘Len’ is the length of a word.

Space Complexity#

The space complexity of the algorithm is $O(M)$ since at most, we will be storing all the words in the two **HashMaps**. In the worst case, we also need $O(N)$ space for the resulting list. So, the overall space complexity of the algorithm will be $O(M + N)$.

← Back

Next →

Problem Challenge 4

Introduction

Mark as Completed

'tag/solution-review-problem-challenge-4__pattern-sliding-
ing-interview-patterns-for-coding-questions?
:he-coding-interview-patterns-for-coding-

9520&cid=5671464854355968&pid=5223876420173824)