## 30. Substring with Concatenation of All Words Sliding Window Hash Table **String** Hard

# **Problem Description** You are given a string s and an array of strings words. All strings within words have the same length. Your objective is to find all the

Consider words as a set of building blocks, where each word is a block of the same size. You have to find all position in s where you can construct a substring using every block exactly once, arranging them in any sequence.

starting indices of substrings in s where a substring is a concatenation of every word in any possible permutation of words.

A "concatenated substring" is such that:

- The words can be in any order. • Each word from words appears exactly as it is (unmodified, and in full).
  - For example, if words is ["ab", "cd", "ef"], then "abcdef", "abefcd", and "cdabef" are concatenated strings if they occur in s. On the
  - other hand, "acdbef" is not a concatenated substring because it doesn't represent any permutation of words.

It consists of all words from the words array.

Intuition

You need to return a list of starting indices of such concatenated substrings found in s, the order of indices does not matter.

computationally expensive. We need a more efficient way to check for concatenated substrings. The intuition behind the solution lies in the pattern recognition, sliding window, and hash table techniques:

starting indices.

**Solution Approach** 

words count.

**Example Walkthrough** 

length 4.

the cnt hash table in the code.

• Pattern Recognition: Since all words in words have the same length, our window of the substring to search in s will also be a constant size, which is the sum of lengths of all words in words.

• Hash Table: By using a hash table to count the occurrences of the words we've seen in the current window, we can keep track

To arrive at the solution for this problem, we have to consider that searching for each possible permutation of words in s would be

• Sliding Window: As we slide this window across s, we can incrementally check whether the substring window contains a valid concatenation of the words.

of the words and their counts to determine if the current window forms a valid concatenated substring.

- The sliding window starts from the beginning of s, and we move it to the right one word-length at a time. We continue this process for all possible positions the first word of the window could start from (i.e., 0 to word-length - 1). At each step, when a new word is included in the sliding window:
- If this new word isn't in the original words count hash table, we reset the current one, as it's not a valid continuation. If the count of the newly added word exceeds its expected count, we slide the window's left bound to the right to exclude

the starting index. By following this strategy, we avoid computing all permutations of words, with the sliding window efficiently narrowing down possible

Whenever the number of words within the sliding window is equal to the size of words and all word counts correspond, we record

1. Hash Table for Counting Words: A hash table is used to count the number of times each word appears in the words array. This is

2. Setting Up for Sliding Window: The key variables are set up for the sliding window algorithm such as the length of the string (m),

the number of words (n), and the length of each word which is assumed to be uniform (k). The variable ans is an array that will

The solution approach involves using a hash table and a sliding window, as already suggested by the intuition.

collect our answer - the starting indices of valid concatenated substrings.

forward because the word does not belong to any concatenation of words.

s, ensuring the constraints are satisfied before adding a starting index to the answer array.

Now let's follow the steps of the approach to find the starting indices of the concatenated substrings:

1. Create the Hash Table: Create a hash table (cnt) to store the frequency of each word in words.

3. Iterate Starting Points: Iterate with i from 0 to less than k (since k is 4, i will take values 0, 1, 2, and 3).

Let's go through a simple example to illustrate the solution approach:

Starting indices of ans will be collected in an array.

Since word is in cnt, add to cnt1 and increment t.

Continue shifting r and repeat the process.

 $\circ$  At this point, r = 20, l = 0, and t = 3.

 $\circ$  No match will be found starting at i = 1.

Similarly, initialize and check from index 2.

Initialize and start checking from index 3.

We add 9 to our ans array, now ans becomes [0, 9].

6. For i = 2 (third possible window position):

4. For i = 0: (First window position)

 $\circ$  Now, r = 4, l = 0, and t = 1.

We add it to a current word frequency count hash table.

enough occurrences, so the counts match.

3. Iterating Starting Points: We start iterating over the string s with variable i which represents the start point of the sliding window. This loop essentially allows us to accommodate words in s starting from different positions up to the word length, making sure we cover all possible alignments of words within s.

6. Updating Word Count: If the word is valid, we increment its count in cnt1 and also the total count t.

window. 5. Processing Words: We then continue to shift our window to the right word by word, capturing each word using s[r:r+k]. We

also keep checking if the captured word is in the cnt hash table. If it's not in cnt, we reset our counters and move the window

7. Validating Counts: If there are too many occurrences of the word in the current window, we increment the left boundary of the

8. Storing Valid Indices: If the total number of words t in the sliding window equals the number of words in words (n), it means we

window 1 to reduce the count of the word in the window. This is necessary to match the word count exactly to that of the input

4. Sliding Window Mechanism: Inside the loop, we initialize a counter for the current window (cnt1), the left (1) and right (r)

boundaries of the window, and the variable t to keep track of the total number of valid words encountered in the current

found a valid concatenation starting at index 1. This index is stored in the answer array ans. By the end of the outer loop, we have considered all possible alignments and have added all valid starting points to the ans array, which is then returned.

The sliding window algorithm is made efficient by the fact that it maintains a balance of counts dynamically as it shifts over the string

cnt = {"word": 1, "good": 1, "best": 1} 2. Set Up Sliding Window Variables: The length of each word (k) is 4, s length (m) is 22, and there are 3 words (n) in words.

Suppose the input string s is "wordgoodgoodgoodbestword" and words is ["word", "good", "best"], where each word in words is of

- Initialize cnt1 (current window frequency), 1 (left boundary of window), r (right, initially i), and t (total valid words in current window, initially 0). Start the right boundary r at index 0 and 1 at index 0.
- Now ans array has [0]. 5. For i = 1 (second possible window position):

Since t equals the number of words in words, we have a valid starting index at 1, which is 0.

Similar process, we initialize the variables again, but we start checking from the first index.

Slide r by k to capture a word. (first word word from indices 0 to 4).

 $\circ$  At r = 16, the window captures best, add it to cnt1 and increment t.

 $\circ$  No match will be found starting at i = 2. 7. For i = 3 (fourth possible window position):

ans array contains [0, 9], which are the starting indices where the concatenated substrings appear in s.

slides over s, and validates the counts of encountered words against cnt to determine valid starting points.

 $\circ$  A match is found for the second occurrence of the valid concatenated substring starting at index 1 = 9.

By the end of the algorithm, we've checked all possible alignments within s for concatenations of all words in words. Thus, the final

The solution is efficient because it does not calculate all permutations of words; instead, it builds the concatenated substring as it

class Solution: def findSubstring(self, s: str, words: List[str]) -> List[int]: # Count the frequency of each word in the 'words' list word\_count = Counter(words) 6 total\_length = len(s)

word\_length = len(words[0])

for offset in range(word\_length):

total\_matched\_words = 0

window\_word\_count = Counter()

right += word\_length

left = right

continue

return start\_indices

for (String word : words) {

if word not in word\_count:

left += word\_length

total matched\_words -= 1

if total\_matched\_words == num\_words:

public List<Integer> findSubstring(String s, String[] words) {

int strLength = s.length(), numOfWords = words.length;

Map<String, Integer> currentCount = new HashMap<>();

String sub = s.substring(right, right + wordLength);

// Create and populate a map with the count of each unique word

int wordLength = words[0].length(); // Assume all words are the same length

// Iterate over all possible word start indices to check for valid substrings

// Expand the window to the right, adding words into current window count

// If the word is not in the original word list, reset the window

Map<String, Integer> wordCount = new HashMap<>();

wordCount.merge(word, 1, Integer::sum);

while (right + wordLength <= strLength) {</pre>

if (!wordCount.containsKey(sub)) {

currentCount.clear();

++windowCount[currentWord];

left += wordSize;

--totalCount;

--windowCount[wordToRemove];

if (totalCount == wordCountSize) {

function findSubstring(s: string, words: string[]): number[] {

wordCountMap.set(word, (wordCountMap.get(word) || 0) + 1);

// Iterate through the string in increments of word length

while (right + wordLength <= stringLength) {</pre>

if (!wordCountMap.has(currentWord)) {

tempCountMap.clear();

const tempCountMap: Map<string, number> = new Map();

// Scan the string in chunks the size of the words' length

// Skip the word if it's not in the frequency map

const currentWord = s.slice(right, right + wordLength);

const wordToLeft = s.slice(left, left + wordLength);

// Check if all words match; if so, add to results

if (matchedWordCount === wordArrayLength) {

store up to m / k starting indices if every substring is a valid concatenation.

const wordCountMap: Map<string, number> = new Map();

// Create a map to store the frequency of words.

// Populate the word frequency map.

const stringLength: number = s.length;

for (let i = 0; i < wordLength; ++i) {</pre>

let matchedWordCount = 0;

right += wordLength;

left = right;

++matchedWordCount;

left += wordLength;

--matchedWordCount;

indices.push(left);

const wordArrayLength: number = words.length;

const wordLength: number = words[0].length;

for (const word of words) {

const indices: number[] = [];

let left = i;

let right = i;

substrIndices.push\_back(left);

while (windowCount[currentWord] > wordCount[currentWord]) {

string wordToRemove = s.substr(left, wordSize);

++totalCount;

return substrIndices;

Typescript Solution

List<Integer> indices = new ArrayList<>();

for (int i = 0; i < wordLength; ++i) {</pre>

right += wordLength;

left = right;

continue;

totalWords = 0;

int left = i, right = i;

int totalWords = 0;

start\_indices.append(left)

# This will hold the start indices of the substrings

while right + word\_length <= total\_length:</pre>

window\_word\_count.clear()

total\_matched\_words = 0

word = s[right: right + word\_length]

# Check every word\_length characters to find valid substrings

# Move the window rightwards in the string 's' by word\_length

# If the word is not in our word\_count, reset window

# Increase the count for the new word in our window

window\_word\_count[word\_to\_remove] -= 1

left = right = offset # Initialize two pointers

num\_words = len(words)

start\_indices = []

**Python Solution** 

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

39

40

41

43

44

45

46

47

48

9

10

11

12

13

14

15

16

17

18

20

21

22

23

24

25

26

27

28

29

30

31

32

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

55

54 };

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

Java Solution

class Solution {

1 from collections import Counter

33 window\_word\_count[word] += 1 34 total\_matched\_words += 1 35 36 # If there are more instances of the word than needed, shrink window 37 while window\_word\_count[word] > word\_count[word]: word\_to\_remove = s[left: left + word\_length] 38

# If the window contains exactly 'num\_words' words, we found a substring starting at 'left'

## 33 34 // Increase the count for the current word in the window 35 36 37

currentCount.merge(sub, 1, Integer::sum); ++totalWords; // If a word count exceeds its count in wordCount, reduce from left side 38 while (currentCount.get(sub) > wordCount.get(sub)) { 39 String removed = s.substring(left, left + wordLength); 40 left += wordLength; currentCount.merge(removed, -1, Integer::sum); 43 --totalWords; 45 // If the total words reached the number of words, a valid substring is found 46 if (totalWords == numOfWords) { indices.add(left); 48 49 50 51 return indices; 52 53 54 } 55 C++ Solution 1 class Solution { 2 public: // This function searches for all starting indices of substring(s) in 's' that is a concatenation of each word in 'words' exactly vector<int> findSubstring(string s, vector<string>& words) { // Count the frequency of each word in 'words'. unordered\_map<string, int> wordCount; for (auto& word : words) { ++wordCount[word]; 10 int stringSize = s.size(), wordCountSize = words.size(), wordSize = words[0].size(); 11 12 vector<int> substrIndices; 13 14 // Iterate over the string 's'. 15 for (int i = 0; i < wordSize; ++i) {</pre> unordered\_map<string, int> windowCount; 16 int left = i, right = i; int totalCount = 0; 19 20 // Slide a window over the string 's'. 21 while (right + wordSize <= stringSize) {</pre> 22 string currentWord = s.substr(right, wordSize); right += wordSize; 24 25 // Skip the current segment if the word is not in 'words'. 26 if (!wordCount.count(currentWord)) { 27 windowCount.clear(); left = right; 28 29 totalCount = 0; 30 continue; 31 32 33 // Update the count for the current word in the window.

// If there are more occurrences of 'currentWord' in the window than in 'words', remove from the left.

// If the total count of words match and all words frequencies are as expected, add to result.

### 29 matchedWordCount = 0; 30 continue; 31 32 33 // Update the temporary count map tempCountMap.set(currentWord, (tempCountMap.get(currentWord) || 0) + 1); 34

return indices;

Time and Space Complexity The time complexity of the given code is 0(m \* k) where m is the length of the string s and k is the length of each word within the words list. This stems from the fact that we iterate over the string s in increments of k for loops starting at each of the first k characters. The space complexity is 0(n \* k) where n is the number of words in the given list words and k is the length of each word. This is due to two counters cnt and cnt1 storing, at most, n different words of k length each, along with the list ans that in the worst case could

// If the current word has been seen more times than it is present in words array, slide the window to the right

while ((tempCountMap.get(currentWord)! - wordCountMap.get(currentWord)!) > 0) {

tempCountMap.set(wordToLeft, tempCountMap.get(wordToLeft)! - 1);