Solution: Find All Anagrams in a String

Let's solve the Find All Anagrams in a String problem using the Knowing What to Track pattern.

We'll cover the following

- Statement
- Solution
  - Time complexity
  - Space complexity

Statement

Given two strings, `a` and `b`, return an array of all the start indexes of anagrams of `b` in `a`. We may return the answer in any order.

> An **anagram** is a word or phrase created by rearranging the letters of another word or phrase while utilizing each of the original letters exactly once. For example, "act" is the anagram of "cat", and "flow" is the anagram of "wolf".

Constraints:

- $1 \leq$ `a.length`, `b.length` $\leq 3 \times 10^3$
- Both `a` and `b` consist only of lowercase English letters.

Solution

In this problem, we need to find the consecutive characters in string `a` that make up any of the anagrams of string `b`. For this, we'll iterate over string `a` with a window of the same size as the length of string `b`. Along the way, if we encounter any of the anagrams of string `b`, we'll store the start index of the substring that represents the anagram. How will we know we've encountered the anagram of string `b` in the string `a`? If the characters appearing in the sliding window are the same as those in string `b`, they represent the anagram of `b` in `a`. Now, as the anagram is created by rearranging the characters in a certain string, the order of the characters in the anagram doesn't need to be the same as the original string. Therefore, instead of directly comparing the substring, represented by the sliding window, with the string `b`, we compare the count of characters in the sliding window with the count of the characters in string `b`. For storing the count of characters, we use the hash maps.

An important case to note is that to find the anagrams of string `b` in string `a`, the length of string `a` should be greater than or equal to string `b`. If string `b` is longer than string `a`, it's impossible to find any anagrams. Therefore, the first step in our solution will be to check if the length of string `a` is greater than or equal to that of string `b`. If it's not, return an empty array.

We initialize an array, `ans`, to store the output, i.e., all the start indexes of anagrams of string `b` in string `a`. We also create two hash maps, `hashA` and `hashA`, which act as counters. While `hashB` stores the count of characters in string `b`, `hashA` stores the count of characters in the sliding window. We'll update the count in `hashA` every time the sliding window moves ahead.

Now, to iterate over string `a`, we'll start a loop that will last till the length of string `a`. In each iteration, we'll do the following:

- Move the sliding window ahead by adding one character to its right. The length of the sliding window can't exceed the length of string `b`, so if it exceeds, we remove one character from its left.
- Update the count of characters in `hashA`.
- Check if the count of characters in `hashA` equals the count of characters in `hashB`. If it's equal, we found an anagram, so add the start index of the substring in the sliding window representing the anagram in the output array, `ans`. Otherwise, move to the next iteration.

Once the loop is over, return the array, `ans`, as the output.

4 of 4

Let's take a look at the code for this solution below:

```java
class Anagrams {

    public static List<Integer> findAnagrams(String a, String b) {

        // list to store the output, i.e., start indexes of all anagrams of string b in string a
        List<Integer> ans = new ArrayList<Integer> ();

        // if length string b is greater than the length of string a, return an empty list
        if (b.length() > a.length()) return ans;

        // create the hash maps
        HashMap<Character, Integer> hashA = new HashMap<>();
        HashMap<Character, Integer> hashB = new HashMap<>();

        // populate hashB with the count of characters in string b
        for (int i = 0; i<b.length(); i++) {
            if (!hashB.containsKey(b.charAt(i))) hashB.put(b.charAt(i), 1);
            else hashB.put(b.charAt(i), hashB.get(b.charAt(i)) + 1);
```

```java
        // traverse string a: in each iteration, move the window rightward by one character
        while (windowEnd < a.length()) {

            // to move the window rightward, add a new element in it, i.e.,
            // add this new element and its count in the hash map, hashA
            if (!hashA.containsKey(a.charAt(windowEnd))) hashA.put(a.charAt(windowEnd), 1);
            else hashA.put(a.charAt(windowEnd), hashA.get(a.charAt(windowEnd)) + 1);
```

Time complexity

We traverse both strings, `a` and `b`, only once, so the time complexity for this solution is $O(n + m)$, where n is the length of string `a`, and m represents the length of string `b`. However, since the length of string `a` $\geq$ length of string `b`, the time complexity simplifies to $O(n)$.

Space complexity

The count in both `hashA` and `hashB` can be, at most, 26 elements because they only contain lowercase English letters. Therefore, the space complexity of the solution above is $O(1)$.

← Back
Find All Anagrams in ...

Next →
Ransom Note

✓ Mark as Completed