438. Find All Anagrams in a String

Medium · Hash Table · String · Sliding Window · Leetcode Link

Problem Description

The task is to find all the starting indices of substrings in string `s` that are anagrams of string `p`. An anagram is a rearrangement of letters in a word to form a new word, using all the original letters exactly once. The two input strings, `s` and `p`, contain only lowercase English letters. The output should be a list of indices, and these indices don't need to be in any specific order.

For example, if `s` is "cbaebabacd" and `p` is "abc", then the function should return `[0, 6]`, since the substrings starting at indices 0 ("cba") and 6 ("bac") are anagrams of "abc".

Intuition

The intuition behind the solution is to use a sliding window approach combined with character counting. The idea is to maintain a window of length equal to the length of `p` and slide it across `s`, while keeping track of the frequency of characters within the window and comparing it to the frequency of characters in `p`. If at any point the frequency counts match, we've found an anagram and can record the starting index of the window.

To implement this efficiently, we use two `Counter` objects (from Python's collections module) to store the frequency counts for `p` and for the current window in `s`. The key steps in the algorithm are as follows:

1. If `s` is shorter than `p`, immediately return an empty list as no anagrams could exist.
2. Create a `Counter` for string `p` which holds the count of each character.
3. Also, create a `Counter` for the first window of size `len(p) - 1` in `s`.
4. Iterate through `s` starting at the index where the first full window would begin. For each index `i`:
   - Increment the count of character `s[i]` in the current window counter.
   - Check if the current window's character counts match those of `p`. If they do, append the starting index of the window `(i - len(p) + 1)` to the result list.
   - Decrement the count of character `s[i - len(p) + 1]` in the current window counter, effectively sliding the window one position to the right.
5. After the loop, return the list of starting indices, which contains all the starting points of anagram substrings within `s`.

Solution Approach

The solution approach utilizes a sliding window algorithm combined with character frequency counting.

Sliding Window Algorithm

The sliding window algorithm is quite efficient for problems that require checking or computing something among all substrings of a given size within a larger string. This technique involves maintaining a window that slides across the data to examine different subsections of it.

In this problem, we slide a window of length `len(p)` across the string `s` and at each step, we compare the frequency of characters within the window to the frequency of characters in `p`.

Character Frequency Counting

To track the characters' frequencies efficiently, we use Python's `Counter` class from the collections module. A `Counter` is a dictionary subclass that allows us to count hashable objects. The elements are stored as dictionary keys, and the counts of the objects are stored as the values.

Here's how the code implements the above techniques:

1. We create a `Counter` called `cnt1` for the string `p`, which will remain constant throughout the algorithm.
2. Initialize a second `Counter` called `cnt2` for the first `len(p) - 1` characters of string `s`. We start with one less than `len(p)` because we will be adding one character to the counter in the main loop, effectively scanning windows of size `len(p)`.

```python
cnt1 = Counter(p)
cnt2 = Counter(s[: len(p) - 1])
```

3. The main loop starts from index `len(p) - 1` and iterates until the end of the string `s`. In each iteration, we:
   - Include the next character in the current window's count by incrementing `cnt2[s[i]]`.
   - Check if `cnt1` and `cnt2` are equal, which means we found an anagram. If they are equal, we append `i - len(p) + 1` to our answer list `ans`.
   - Before moving to the next iteration, we decrement the count of the character at the start of the current window since the window will shift to the right in the next step.

```python
for i in range(len(p) - 1, len(s)):
    cnt2[s[i]] += 1
    if cnt1 == cnt2:
        ans.append(i - len(p) + 1)
    cnt2[s[i - len(p) + 1]] -= 1
```

Notice how the algorithm elegantly maintains the window size by incrementing the end character count and decrementing the start character count in each iteration, making the time complexity for this operation O(1), assuming that the inner workings of the Counter object and comparing two Counters take constant time on average.

By employing the sliding window technique and frequency counting, we avoid the need to re-compute the frequencies from scratch for each window, thereby achieving an efficient solution.

Example Walkthrough

Let's consider a small example to illustrate how the sliding window approach and character frequency counting is applied in the implementation.

Suppose our string `s` is "abab" and our pattern string `p` is "ab". We want to find all starting indices of substrings in `s` that are anagrams of `p`.

Following the steps described in the solution approach:

1. Since `s` is not shorter than `p`, we proceed with creating a `Counter` object for `p`.

```python
cnt1 = Counter("ab")  # This will hold {'a': 1, 'b': 1}
```

2. Next, initialize a `Counter` object for the first `len(p) - 1` characters of `s`. In our case, the initial window will be of size 1 (2-1).

```python
cnt2 = Counter(s[:1])  # This will hold {'a': 1}
```

3. Now, we will iterate from index 1 to the length of `s`.

   - For index 1, we include character 'b' in `cnt2` and increment its count.
   - We compare `cnt1` with `cnt2`. Since at this point `cnt1` is {'a': 1, 'b': 1} and `cnt2` is {'a': 1, 'b': 1}, they are equal, which indicates that we have found an anagram starting at index 0.
   - The list `ans` will now include this index:

```python
ans = [0]
```

```
- Before moving to the next index, we decrement the count of the character at the start of the current window (which is 'a') in `cnt2`
```

4. At index 2, we include character 'a' in `cnt2` then repeat the steps. `cnt1` and `cnt2` will still be equal, so we add index 1 to `ans`.

```python
ans = [0, 1]
```

```
- Decrement the count of 'b' in `cnt2` before the next iteration.
```

5. At index 3, we include character 'b' in `cnt2`. The Counters are still equal, add index 2 to `ans`.

```python
ans = [0, 1, 2]
```

```
- Finally, decrement the count of 'a' in `cnt2` as the loop is completed.
```

6. Since we have no more characters in `s` to iterate through, we have completed our search. The list `ans` now contains all the starting indices of the anagrams of `p` in `s`.

The final output for our example `s = "abab"` and `p = "ab"` is `[0, 1, 2]`, indicating that anagrams of `p` start at indices 0, 1, and 2 in `s`.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        # Length of the string 's' and the pattern 'p'
        s_length, p_length = len(s), len(p)
        # List to store the starting indices of the anagrams of 'p' in 's'
        anagram_indices = []

        # Early return if 's' is shorter than 'p', as no anagrams would fit
        if s_length < p_length:
            return anagram_indices

        # Count the occurrences of characters in 'p'
        p_char_count = Counter(p)
        # Count the occurrences of characters in the first window of 's'
        # This window has the size one less than the size of 'p'
        s_window_count = Counter(s[:p_length - 1])

        # Iterate over 's', starting from index where first full window can begin
        for i in range(p_length - 1, s_length):
            # Add the current character to the window's character count
            s_window_count[s[i]] += 1

            # If the character counts match, it's an anagram; record the start index
            if p_char_count == s_window_count:
                anagram_indices.append(i - p_length + 1)

            # Move the window to the right: subtract the count of the oldest character
            s_window_count[s[i - p_length + 1]] -= 1
            # Remove the character count from the dict if it drops to zero
            if s_window_count[s[i - p_length + 1]] == 0:
                del s_window_count[s[i - p_length + 1]]

        # Return the list of starting indices of anagrams
        return anagram_indices

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        // Initialize the length of both strings
        int stringLength = s.length(), patternLength = p.length();
        List<Integer> anagramsStartIndices = new ArrayList<>();

        // If the pattern is longer than the string, no anagrams can be found
        if (stringLength < patternLength) {
            return anagramsStartIndices;
        }

        // Count the frequency of characters in the pattern
        int[] patternCount = new int[26];
        for (int i = 0; i < patternLength; ++i) {
            patternCount[p.charAt(i) - 'a']++;
        }

        // Initialize character frequency counter for window in 's'
        int[] windowCount = new int[26];
        // Initialize the window with the first (patternLength - 1) characters
        for (int i = 0; i < patternLength - 1; ++i) {
            windowCount[s.charAt(i) - 'a']++;
        }

        // Slide the window across the string 's' one character at a time
        for (int i = patternLength - 1; i < stringLength; ++i) {
            // Add the new character to the window
            windowCount[s.charAt(i) - 'a']++;

            // If the window matches the pattern counts, record the start index
            if (Arrays.equals(patternCount, windowCount)) {
                anagramsStartIndices.add(i - patternLength + 1);
            }

            // Remove the leftmost (oldest) character from the window
            windowCount[s.charAt(i - patternLength + 1) - 'a']--;
        }

        return anagramsStartIndices;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find all the start indices of p's anagrams in s.
    std::vector<int> findAnagrams(std::string s, std::string p) {
        int sLength = s.size(), pLength = p.size();
        std::vector<int> startingIndices; // Holds the starting indices of anagrams.

        // Base case: if the length of string s is smaller than the length of string p,
        // no anagrams of p can be found in s.
        if (sLength < pLength) {
            return startingIndices;
        }

        // Counters for the characters in p and the current window in s.
        std::vector<int> pCharCount(26, 0); // Assuming only lowercase English letters.
        std::vector<int> windowCharCount(26, 0);

        // Count the occurrences of each character in p.
        for (char c : p) {
            ++pCharCount[c - 'a']; // Increment the count for this character.
        }

        // Initialize windowCharCount with the first window in s.
        for (int i = 0; i < pLength - 1; ++i) {
            ++windowCharCount[s[i] - 'a'];
        }

        // Slide the window over string s and compare with character counts of p.
        for (int i = pLength - 1; i < sLength; ++i) {
            ++windowCharCount[s[i] - 'a']; // Add the current character to the window count.

            // If the window has the same character counts as p, it's an anagram starting at (i - pLength + 1).
            if (pCharCount == windowCharCount) {
                startingIndices.push_back(i - pLength + 1);
            }

            // Move the window forward by one: decrease the count of the character leaving the window.
            --windowCharCount[s[i - pLength + 1] - 'a'];
        }

        return startingIndices; // Return the collected starting indices of anagrams.
    }
};

```

Typescript Solution

```typescript
function findAnagrams(source: string, pattern: string): number[] {
    const sourceLength = source.length;
    const patternLength = pattern.length;
    const resultIndices: number[] = [];

    // If the pattern is longer than the source string, no anagrams are possible
    if (sourceLength < patternLength) {
        return resultIndices;
    }

    // Character count arrays for the pattern and the current window in the source string
    const patternCount: number[] = new Array(26).fill(0);
    const windowCount: number[] = new Array(26).fill(0);

    // Helper function to convert a character to its corresponding index (0-25)
    const charToIndex = (c: string) => c.charCodeAt(0) - 'a'.charCodeAt(0);

    // Initialize the character count for the pattern
    for (const char of pattern) {
        patternCount[charToIndex(char)]++;
    }

    // Initialize the character count for the first window in the source string
    for (const char of source.slice(0, patternLength - 1)) {
        windowCount[charToIndex(char)]++;
    }

    // Slide the window over the source string and compare counts of characters
    for (let i = patternLength - 1; i < sourceLength; ++i) {
        // Include the current character into the window count
        windowCount[charToIndex(source[i])]++;

        // Compare the pattern character count array with the current window count array
        if (arrayEqual(patternCount, windowCount)) {
            // If they match, add the starting index to the result array
            resultIndices.push(i - patternLength + 1);
        }

        // Remove the first character of the current window from the window count
        windowCount[charToIndex(source[i - patternLength + 1])]--;
    }

    // Helper function to check if two arrays are equal
    function arrayEqual(arr1: number[], arr2: number[]): boolean {
        for (let i = 0; i < arr1.length; i++) {
            if (arr1[i] !== arr2[i]) {
                return false;
            }
        }
        return true;
    }

    return resultIndices;
}

```

Time and Space Complexity

The time complexity of the given code snippet is `O(m + n)`, where `m` is the length of string `s` and `n` is the length of string `p`. Here's a breakdown of the complexity:

- Initializing `cnt1` and `cnt2` takes `O(n)` time because we are counting characters in strings of length `n` and `n - 1`, respectively.
- The `for` loop runs from `n - 1` to `m`, resulting in `O(m - n + 1)` iterations.
- Inside the loop, we increment the count of `s[i]` in `O(1)` time, compare `cnt1` and `cnt2` in `O(1)` time assuming the alphabet is fixed and thus the Counter objects have a constant number of keys, and decrement the count of a character in `O(1)` time.
- Therefore, loop operations have a complexity of `O(m - n + 1)`, and setting up the Counter objects has `O(n)`, resulting in a total time complexity of `O(n + m - n + 1)` which simplifies to `O(m + n)`.

The space complexity is `O(1)` or `O(k)` where `k` is the size of the character set (e.g., the alphabet size), assuming the alphabet is fixed and does not grow with `n` or `m`. This is because the space that `cnt1` and `cnt2` take up does not depend on the length of `s` or `p` but only on the number of distinct characters, which is constant for a fixed alphabet.