Documentation on Find All Anagrams in a String

Table of Contents

1. Problem Statement
2. Intuition
3. Key Observations
4. Approach
5. Edge Cases
6. Complexity Analysis
    - Time Complexity
    - Space Complexity
7. Alternative Approaches
8. Test Cases
9. Final Thoughts

1. Problem Statement

Given two strings s and p, find all start indices of p's anagrams in s.

You may return the indices in any order.

Example 1:

Input:

s = "cbaebabacd"
p = "abc"

Output:[0, 6]

Explanation:

- The substring "cba" at index 0 is an anagram of "abc".
- The substring "bac" at index 6 is an anagram of "abc".

Example 2:

Input:

s = "abab"
p = "ab"

Output:[0, 1, 2]

Explanation: "ab" at index 0, "ba" at index 1, and "ab" at index 2 are all anagrams of "ab".

Constraints:

- $1 \le s.length, p.length \le 3 \times 10^4$
- s and p consist of lowercase English letters.

2. Intuition

The problem requires finding all substrings of s that are permutations (anagrams) of p.

Instead of checking every possible substring by sorting (which is inefficient), we can use a Sliding Window + HashMap approach to track character frequencies efficiently.

3. Key Observations

- An anagram contains the same frequency of characters as p, just in a different order.
- Instead of checking every substring manually, we can use a sliding window to maintain a character frequency count dynamically.
- If two frequency counts match, then the substring is an anagram.

4. Approach

i. Create a frequency map of p using Counter(p).
ii. Initialize a sliding window in s of size len(p) and compute its frequency count.
iii. Check if the first window matches p's frequency count. If yes, store the index 0.
iv. Slide the window through s, updating character frequencies dynamically:
    a. Remove the leftmost character from the count.
    b. Add the new rightmost character to the count.
    c. Compare the updated frequency map with p_count. If they match, store the starting index.
v. Return all valid indices.

5. Edge Cases

- s is shorter than p → Return [] since no anagram is possible.
- s and p contain distinct characters → No anagram matches.
- Multiple overlapping anagrams → Ensure we correctly slide the window.
- All characters in s are the same as p → Edge case where every substring is an anagram.
- Very large s and p (edge case for efficiency) → Should run in O(N), avoiding brute force.

6. Complexity Analysis

Time Complexity

- Constructing p_count → O(M) (M = length of p)
- Constructing initial window → O(M)
- Sliding window iteration through s → O(N - M) (N = length of s)
- Overall complexity: O(N)

Space Complexity

- p_count and s_count each store at most 26 lowercase letters → O(1).
- Result list can be O(N/M) in the worst case but is not a dominant factor.
- Overall space complexity: O(1) (ignoring output storage).

7. **Alternative Approaches**

Brute Force (Inefficient) – O(N*M log M)

1. Generate all substrings of s of length p.
2. Sort each substring and compare it with sorted p.
3. Issue: Sorting takes O(M log M), making it too slow for large N.

Using a Fixed-Size Hash Table – O(N)

Instead of sorting, use a frequency count (hash table) to compare character counts dynamically using a sliding window (our chosen approach).

8. **Test Cases**

Basic Cases:

assert Solution().findAnagrams("cbaebabacd", "abc") == [0,6]
assert Solution().findAnagrams("abab", "ab") == [0,1,2]

Edge Cases:

assert Solution().findAnagrams("abcd", "e") == []
assert Solution().findAnagrams("aaaaaa", "aa") == [0,1,2,3,4]
assert Solution().findAnagrams("a", "a") == [0]
assert Solution().findAnagrams("a", "b") == []

9. **Final Thoughts**

- This solution efficiently finds all anagrams in O(N) time.
- Sliding Window + Hash Map avoids unnecessary sorting, making it optimal.
- Alternative brute-force solutions are too slow for large inputs.
- Edge cases handled well, ensuring robustness.