Topic Introduction

Anagram Problems and the Power of Character Counting

Today we're exploring a classic family of string interview problems: those that revolve around **anagrams**.

What is an anagram? An anagram is a rearrangement of the letters of a word or phrase to make a new word or phrase, using all the original letters exactly once. For example, “listen” and “silent” are anagrams.

Core Concept: Character Frequency Counting

The unifying theme for today's problems is character frequency counting—tallying how many times each character appears in a string. This is a common and powerful pattern for string problems, especially those involving anagrams, because:

- If two strings are anagrams, they have identical character counts for each letter.
- If we want to group words by their anagram-ness, or find all substrings that are anagrams of a pattern, character counts help us quickly compare and classify strings.

How does it work?

We often use a data structure like a hash map (dictionary in Python) or a fixed-size array (for lowercase letters) to keep track of character occurrences. This enables us to:

- Compare two strings in linear time.
- Build a signature (like a tuple of counts) to group or classify strings efficiently.

When is this useful in interviews?

- Detecting duplicates or anagrams.
- Grouping words or data by content, not order.
- Finding patterns within strings.

Simple Example

Suppose you want to check if "race" and "care" are anagrams.
Count characters:

- "race": r:1, a:1, c:1, e:1
- "care": c:1, a:1, r:1, e:1

All counts match: they are anagrams!

Today’s Problem Set

We'll tackle three popular LeetCode problems, all harnessing the magic of character counting for anagram detection:

- **Valid Anagram** (LeetCode 242): Are two strings anagrams?
- **Group Anagrams** (LeetCode 49): Group a list of strings into clusters of anagrams.
- **Find All Anagrams in a String** (LeetCode 438): Find every substring in a larger string that is an anagram of a given pattern.

Why group these together?

They all require you to compare or group strings based on their characters, not their order. While "Valid Anagram" is about comparing two strings, "Group Anagrams" is about classifying many strings, and "Find All Anagrams in a String" uses a sliding window to spot anagrams inside a larger string. All use the core idea of character counts or signatures.

Let’s dive in!

Problem 1: Valid Anagram

Problem Statement (Rephrased):

LeetCode 242: Valid Anagram

Given two strings `s` and `t`, determine if `t` is an anagram of `s`. Both strings contain only lowercase letters.

Example:

Input: s = "listen", t = "silent"

Output: True

Why? Both have the same letters with the same counts.

Thought Process:

How do we know if two strings are anagrams?

- They must be the same length.
- For every character, the count in `s` must equal the count in `t`.

Try This Test Case Yourself:

s = "triangle", t = "integral"

What should the answer be?

Brute-Force Approach:

- Generate all permutations of `s` and check if `t` is one of them.
- But generating all permutations is O(N!)—way too slow.

Optimal Approach:

Let’s use the character counting pattern!

- If lengths differ, return False immediately.
- Count the frequency of each character in both strings.
- Compare the two frequency maps. If they match, return True.

Why is this efficient?

Counting is linear in the length of the strings.

Python Solution:

```
def isAnagram(s: str, t: str) -> bool:
    # If lengths are different, cannot be anagrams
    if len(s) != len(t):
        return False
```

```
    # Use dictionaries to count character frequencies
    count_s = {}
    count_t = {}

    for char in s:
        count_s[char] = count_s.get(char, 0) + 1

    for char in t:
        count_t[char] = count_t.get(char, 0) + 1

    # Compare the two dictionaries
    return count_s == count_t
```

Time Complexity: O(N)

Space Complexity: O(1) (since only 26 lowercase letters, the dict size is capped)

What does each part do?

- We check lengths up front for a fast fail.
- We count each character occurrence in both strings using dictionaries.
- Finally, we compare the two dictionaries for equality.

Trace with Example:

s = "rat", t = "tar"

- count_s: {'r':1, 'a':1, 't':1}
- count_t: {'t':1, 'a':1, 'r':1}
- Both dictionaries are equal, so return True.

Try This Test Case:

s = "apple", t = "papel"

Take a moment to solve this on your own before peeking at the code!

Reflect:

Did you notice that sorting both strings and comparing is another valid O(N log N) approach? Try implementing that after finishing this!

Problem 2: Group Anagrams

Problem Statement (Rephrased):

LeetCode 49: Group Anagrams

Given a list of strings, group them into clusters of anagrams.

Example:

Input: `["eat", "tea", "tan", "ate", "nat", "bat"]`

Output: `[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]`

How is this different from Problem 1?

Now, instead of comparing just two strings, we're grouping all strings that are anagrams of each other.

Brute-Force Approach:

Compare each string to every other string and group those that are anagrams.

This leads to O(N^2 * K) time (N = number of strings, K = average length).

Optimal Approach:

We'll use a **signature** for each string—a unique identifier that is identical for anagrams.

Two Common Signature Patterns:

- Sort the string (e.g., "eat" → "aet").
- Use a tuple of character counts (e.g., (1,0,0,1,1,...) for "eat").

We'll go with sorting for simplicity.

Step-by-step:

- Create a dictionary to group lists of words by their sorted character signature.
- For each string, sort its letters and use the result as a key.
- Append the word to the list for that key.
- Return all the grouped lists.

Pseudocode:

```
initialize empty dict: anagram_groups
for each word in input list:
    signature = sort(word)
    append word to anagram_groups[signature]
return all values in anagram_groups
```

Example Input/Output:

Input: ["listen", "silent", "enlist", "google", "gogole"]

Output: [["listen", "silent", "enlist"], ["google", "gogole"]]

Try This Test Case:

Input: ["abc", "bca", "cab", "dog", "god"]

What should the groups be?

Trace with Example:

For "abc", signature is "abc".

For "bca", signature is "abc".

For "cab", signature is "abc".

For "dog", "dgo".

For "god", "dgo".

So, two groups: ["abc", "bca", "cab"], ["dog", "god"]

Potential Time Complexity:

- Sorting each word: O(K log K) for each.
- For N words: O(N*K log K)

- Space: O(N*K) for the result.

Takeaways:

Sorting is straightforward, but the count-tuple signature using a fixed array is also popular (especially for constraints like only lowercase letters).

Problem 3: Find All Anagrams in a String

Problem Statement (Rephrased):

LeetCode 438: Find All Anagrams in a String

Given strings `s` and `p`, return all starting indices in `s` where the substring is an anagram of `p`.

Example:

Input: s = "cbaebabacd", p = "abc"

Output: [0, 6]

Why? Substrings "cba" at index 0 and "bac" at index 6 are anagrams of "abc".

What's Different or More Challenging?

Now we're looking for *all substrings* of length `len(p)` in `s` that are anagrams of `p`.

We need to check every window of length `len(p)`—but efficiently.

Brute-Force Approach:

For every possible substring of length `len(p)`, check if it's an anagram using the character counting method from Problem 1.

This is O(N * K), where N = len(s), K = len(p).

Optimal Approach:

Use a **sliding window** with character counting:

- Count the frequencies of characters in `p`.
- Slide a window of size `len(p)` across `s`, maintaining a count of characters in the window.
- After updating the window, compare the counts. If they match, the window is an anagram.

Pseudocode:

```
initialize result list
count_p = character counts of p
count_window = counts for first len(p)-1 characters in s

for i from len(p)-1 to len(s)-1:
    add s[i] to count_window
    if count_window == count_p:
        append (i - len(p) + 1) to result
    remove s[i - len(p) + 1] from count_window (decrement count)
return result
```

Example:

s = "abab", p = "ab"

Window 0-1: "ab" -> matches -> add 0

Window 1-2: "ba" -> matches -> add 1

Window 2-3: "ab" -> matches -> add 2

Output: [0, 1, 2]

Try This Test Case:

s = "afdgacda", p = "da"

What indices should be returned?

Potential Time and Space Complexity:

- Time: O(N), because each update is O(1) and we scan through `s` once.
- Space: O(1), since only 26 letters' counts are tracked.

Nudge:

How could you further optimize if the input used only digits, or supported unicode? Try thinking about the tradeoffs!

Summary and Next Steps

Recap:

Today, you've dug into three essential anagram problems:

- **Valid Anagram:** Compare two strings.
- **Group Anagrams:** Cluster a list by anagram groups.
- **Find All Anagrams in a String:** Use a sliding window and counting to spot all anagram substrings.

Key Patterns:

- Character counting (hash maps or fixed arrays) is a must-know trick for anagram problems.
- Sliding windows let you efficiently scan for substring matches.
- Sorting can serve as a simple signature for grouping, but counting is more flexible and efficient for fixed-size alphabets.

Common Pitfalls:

- Forgetting to compare string lengths up front.
- Not resetting the window properly for sliding window problems.
- Using inefficient O(N^2) grouping when hashing or sorting signatures is much better.

Action List

- **Solve all 3 problems on your own:** Even the one with code provided—typing it out builds muscle memory.
- **Try the "tuple of counts" approach** for Problem 2, instead of sorting.
- **Tackle Problem 3 with a different alphabet** (like uppercase letters or digits) to practice generalizing.
- **Check your solutions against LeetCode discussions:** Spot optimizations and edge-case handling.
- **Reflect:** Could you reuse parts of your code between these problems?
- **Don't stress if you get stuck:** The core idea is to keep practicing and learning.

Happy coding! Tomorrow, we'll tackle another concept together.