02_Strings

- Counting Characters in a String
- Anagram
- Palindrome

Counting Characters in a String

h e l l o   w o r l d

Initial state (no characters processed yet)

Current counts: {}

Need to know: Dictionary , for loop

```python
def count_characters(string):
    counts = {}
    for char in string:
        if char in counts:
            counts[char] = counts[char] + 1
        else:
            counts[char] = 1
    return counts


# Example usage:
input_string = "hello world"
result = count_characters(input_string)

print("Character counts:")
for char, count in result.items():
    print(f"'{char}': {count}")
```

If you need to keep a counter of characters, a common mistake is to say that the space complexity required for the counter is O(n). The space required for a counter of a string of latin characters is O(1) not O(n). This is because the upper bound is the range of characters, which is usually a fixed

constant of 26. The input set is just lowercase Latin characters.

Anagram

An anagram is word switch or word play. It is the result of rearranging the letters of a word or phrase to produce a new word or phrase, while using all the original letters only once.

In interviews, usually we are only bothered with words without spaces in them.

Let's break down the three common approaches in very simple terms:

1. **Sorting Both Strings**
   - **How it works:**
     Rearrange (sort) the letters of both strings alphabetically. If the sorted versions are exactly the same, then the strings are anagrams.
   - **Time Cost:**
     Sorting takes about O(n log n) time, where *n* is the number of characters.
   - **Space Cost:**
     It typically uses extra space proportional to the depth of recursion (about O(log n)).
   - **Simple Analogy:**
     Imagine you have two mixed-up decks of cards. If you sort them by suit and number, and they end up in the same order, then both decks originally had the same cards.
2. **Prime Multiplication Method**
   - **How it works:**
     Assign each letter a unique prime number (like a secret code). For each string, multiply the primes corresponding to its letters. Thanks to the unique nature of prime factors, two strings will have the same product if and only if they have the exact same letters.
   - **Time Cost:**
     You only go through the string once, so it takes O(n) time.
   - **Space Cost:**
     It uses constant extra space, O(1), since the mapping of letters to primes is fixed.
   - **Simple Analogy:**
     Think of it like each letter is a unique ingredient with a special "number." Mix them all together (multiply), and if two recipes (strings) yield the same "flavor number," they contain exactly the same ingredients.
3. **Frequency Counting**
   - **How it works:**
     Count how many times each letter appears in each string using a hash (dictionary). Then, compare these counts. If they match for every letter, the strings are anagrams.
   - **Time Cost:**
     This also takes O(n) time, as you just make one pass through each string.

- **Space Cost:**
  It uses constant extra space, O(1), because the number of letters (for example, 26 for the English alphabet) is fixed.
- **Simple Analogy:**
  Imagine you have two baskets of fruit. If you count the number of apples, oranges, etc., in both baskets and the counts match exactly, then both baskets have the same mix of fruits.

Each method has its tradeoffs between speed and space:

- **Sorting** is straightforward but slightly slower due to the sorting process.
- **Prime multiplication** is fast and space-efficient, but it can run into issues with very long strings because the multiplication might produce huge numbers.
- **Frequency counting** is both fast and efficient, and it's often the simplest and most reliable method.

Palindrome

Palindrome

Worse approaches

- **Reversing the String:**
  You can reverse the string (using slicing like `s[::-1]` in Python) and then compare it to the original. This is very concise but uses O(n) additional space.
- **Recursion:**
  A recursive approach can check the first and last characters and then recurse on the substring that excludes them. However, this method uses extra space for the call stack and is generally less efficient.

Overall, the two-pointer approach is typically considered the best way to solve the palindrome problem due to its efficiency in both time and space.

Starting palindrome check.

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as `madam` or `racecar`.

Here are ways to determine if a string is a palindrome:

- Reverse the string and it should be equal to itself.
- Have two pointers at the start and end of the string. Move the pointers inward till they meet. At every point in time, the characters at both pointers should match.

The order of characters within the string matters, so hash tables are usually not helpful.

When a question is about counting the number of palindromes, a common trick is to have two pointers that move outward, away from the middle.

- Note that palindromes can be even or odd length. For each middle pivot position, you need to check it twice - once that includes the character and once without the character.
- For substrings, you can terminate early once there is no match

```python
def is_palindrome(s: str) -> bool:
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True


# Example usage:
input_string = "racecar"
if is_palindrome(input_string):
    print(f"'{input_string}' is a palindrome.")
else:
    print(f"'{input_string}' is not a palindrome.")
```