**Solution: Valid Anagram**

**Problem Statement**

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1**:

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

Output: true

**Example 2**:

Input: s = "rat", t = "car"

Output: false

**Example 3**:

Input: s = "hello", t = "world"

Output: false

**Constraints:**

* 1 <= s.length, t.length <= 5 \* 105
* s and t consist of lowercase English letters.

**Solution**

We can solve this problem by calculating how many times each character appears in both the strings. If the frequency of each character is the same in both the given strings, we can conclude that the strings are anagrams of each other.

We can use a hash map to store the frequency of each character in both strings.

For each character in the string, the frequency of that character in string s is incremented and the frequency of that character in string t is decremented. After iterating over all characters in the strings, we will check if the frequency of all characters is 0. If it is, the strings are anagrams of each other and the function returns true. Otherwise, it returns false.

**Code**

Here is the code for this algorithm:

class Solution:

  def isAnagram(self, s, t):

    # Check if the lengths of both strings are equal. If not, return false.

    if len(s) != len(t):

      return False

    # Create a hash map to store the frequency of characters in both strings.

    freq\_map = {}

    for i in range(len(s)):

      # Increment the frequency of the character in string s.

      if s[i] in freq\_map:

        freq\_map[s[i]] += 1

      else:

        freq\_map[s[i]] = 1

      # Decrement the frequency of the character in string t.

      if t[i] in freq\_map:

        freq\_map[t[i]] -= 1

      else:

        freq\_map[t[i]] = -1

    # Check if the frequency of all characters is 0.

    for char in freq\_map:

      if freq\_map[char] != 0:

**. . . .**

Execute

**Complexity Analysis**

**Time Complexity**

* **Length comparison**: First, the algorithm checks whether the lengths of the two strings s and t are the same. This comparison takes  time.
* **Hash map population (first loop)**:
  + The algorithm then iterates over both strings simultaneously, processing each character in both s and t. For each character, it updates the frequency in the hash map. The put() and getOrDefault() operations on a HashMap have an average time complexity of .
  + Since the loop runs for N iterations, where N is the length of the strings, this part of the algorithm takes  time.
* **Hash map validation (second loop)**:
  + After the hash map is populated, the algorithm iterates over the keys of the hash map to check if any frequency is non-zero. This loop iterates at most N times, where N is the number of distinct characters in the strings.
  + Each get() operation on the hash map is  on average.
  + This part of the algorithm also runs in  time.

**Overall time complexity**: , where N is the length of the input strings.

**Space Complexity**

* The algorithm uses a HashMap to store the frequency of characters. In the worst case, the hash map will contain all unique characters from the input strings. Hence, the space complexity is proportional to the number of distinct characters.
* In the worst case, there can be 26 distinct lowercase characters, so the space complexity is .

**Overall space complexity**: .