package com.iimtiaz.day\_01;  
  
import java.util.\*;  
  
public class Anagram {  
 public static void main(String[] args) {  
 System.*out*.println(new Solution\_1().isAnagram("anagram", "nagaram"));  
 System.*out*.println(new Solution\_1().isAnagram("rat", "car"));  
 System.*out*.println(new Solution\_2().isAnagram("anagram", "nagaram"));  
 System.*out*.println(new Solution\_2().isAnagram("rat", "car"));  
 System.*out*.println(new Solution\_3().isAnagram("anagram", "nagaram"));  
 System.*out*.println(new Solution\_3().isAnagram("rat", "car"));  
 }  
}  
  
*/\*\*  
 Time complexity: O(n log n)  
 O(1): Checking the lengths of the strings.  
 O(n): Converting the strings to character arrays.  
 O(n log n): Sorting both character arrays. This is the dominant term due to the use of sorting algorithms like merge  
 sort or quicksort.  
 O(n): Comparing the sorted arrays.  
  
 Space complexity: O(n)  
 O(n): Two character arrays to store the converted strings.  
 O(n): Temporary space used by the sorting algorithm (depends on the specific algorithm used).  
 \*/*

class Solution\_1 {  
 public boolean isAnagram(String s, String t) {  
 if (s.length() != t.length()) {  
 return false;  
 }  
 char[] str1 = s.toCharArray();  
 char[] str2 = t.toCharArray();  
 Arrays.*sort*(str1);  
 Arrays.*sort*(str2);  
 return Arrays.*equals*(str1, str2);  
 }  
}

*/\*\*  
 Time complexity: O(n)  
 O(1): Checking the lengths of the strings.  
 O(n): Iterating through each character in both strings and updating the character counts.  
 O(n): Iterating through the character count array to check for non-zero values.  
  
 Space complexity: O(1)  
 O(1): Fixed-size character count array with 26 elements (assuming ASCII characters).  
 \*/*class Solution\_2 {  
 public boolean isAnagram(String s, String t) {  
 if (s.length() != t.length()) {  
 return false;  
 }  
 int[] charCounts = new int[26];  
 for (int i = 0; i < s.length(); i++) {  
 charCounts[s.charAt(i) - 'a']++;  
 charCounts[t.charAt(i) - 'a']--;  
 }  
 for (int count : charCounts) {  
 if (count != 0) {  
 return false;  
 }  
 }  
 return true;  
 }  
}  
  
*/\*\*  
 Time complexity: O(n!)  
 O(n!): The permute function generates all possible permutations of the t string, which involves a recursive nested  
 loop that iterates over each character position and performs further permutations on the remaining string.  
 This leads to a factorial growth in the number of operations as the string length increases.  
 O(n): Checking each permuted string against s involves iterating through both strings once, creating an additional  
 O(n) complexity layer for each permutation.  
  
 Space complexity: O(n!)  
 O(n!): Each recursive call to permute creates a new string object to store the current prefix, leading to a stack  
 of such strings during the permutation process. As the number of permutations grows factorially, the space complexity  
 also increases dramatically.  
 \*/*class Solution\_3 {  
 public boolean isAnagram(String s, String t) {  
 List<String> tPermutations = new ArrayList<>();  
 permute(t, "", tPermutations);  
 for (String permutation : tPermutations) {  
 if (permutation.equals(s)) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 private void permute(String str, String prefix, List<String> permutations) {  
 if (str.isEmpty()) {  
 permutations.add(prefix);  
 } else {  
 for (int i = 0; i < str.length(); i++) {  
 String rem = str.substring(0, i) + str.substring(i + 1);  
 permute(rem, prefix + str.charAt(i), permutations);  
 }  
 }  
 }  
}  
  
// Topic:  
// Why it is bad to find every combination and match  
// Why substring did not work  
// char[] vs String[] vs String, int[]  
// https://leetcode.com/problems/valid-anagram/