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/>

package com.iimtiaz.day\_02;  
  
import java.util.Arrays;  
import java.util.HashMap;  
import java.util.Map;  
  
public class TwoSum {  
 public static void main(String[] args) {  
 int[] nums = {3, 2, 4};  
 int target = 6;  
 int array\_size = nums.length;  
 System.*out*.println(Arrays.*toString*(new Solution\_1().twoSum(nums, target)));  
 System.*out*.println(Arrays.*toString*(new Solution\_2().twoSum\_2(nums, target)));  
  
 }  
}  
  
*/\*\*  
 \* Time complexity: O(n^2)  
 \* O(1): Initializing variables like result.  
 \* O(n \* n): Nested loop iterates through each element of the array  
 \* twice (n times for the outer loop and n times  
 \* for the inner loop).  
 \* O(1): Checking if i != j and comparing if nums[i] + nums[j] == target  
 \* are both constant time operations.  
 \* <p>  
 \* Space complexity: O(1)  
 \* Only a few constant-size variables are used (e.g., i, j, result).  
 \*/*class Solution\_1 {  
 public int[] twoSum(int[] nums, int target) {  
 int[] result = null;  
 for (int i = 0; i < nums.length; i++) {  
 for (int j = i; j < nums.length; j++) {  
 if (i != j) {  
 if (nums[i] + nums[j] == target) {  
 result = new int[]{i, j};  
 }  
 }  
 }  
 }  
 return result;  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(n) (linear time)  
 \* We iterate through the array once, which takes O(n) time.  
 \* For each element, we calculate its complement (the value needed to reach the target).  
 \* The hash map allows constant-time lookups, so checking if the complement exists in the  
 \* map also takes O(1) time. Overall, the time complexity is O(n).  
 \* <p>  
 \* Space Complexity: O(n) (linear space)  
 \* We use a hash map (numToIndex) to store encountered elements and their indices.  
 \* The space required depends on the number of items stored in the hash map.  
 \* In the worst case, when all elements are unique, the map stores exactly n elements.  
 \* Therefore, the space complexity is O(n).  
 \*/*class Solution\_2 {  
 public int[] twoSum\_2(int[] nums, int target) {  
 // Create a hash map to store encountered elements and their indices  
 Map<Integer, Integer> numToIndex = new HashMap<>();  
 for (int i = 0; i < nums.length; i++) {  
 int complement = target - nums[i];  
 if (numToIndex.containsKey(complement)) {  
 // Found a valid pair  
 return new int[]{numToIndex.get(complement), i};  
 }  
 // Add the current number and its index to the map  
 numToIndex.put(nums[i], i);  
 }  
 // No valid pair found  
 return new int[0];  
 }  
}  
  
  
*/\*\*  
 Time complexity: O(n) (linear time)  
 We iterate through the array twice:  
 First, we build the hash map by adding each element to it. This takes O(n) time.  
 Next, we find the complement for each element and check if it exists in the map.  
 This also takes O(n) time. Therefore, the overall time complexity is O(n).  
  
 Space complexity: O(n) (linear space)  
 We use a hash map (numMap) to store encountered elements and their indices.  
 The space required depends on the number of items stored in the hash map.  
 In the worst case, when all elements are unique, the map stores exactly n elements.  
 Therefore, the space complexity is O(n).  
 \*/*class Solution\_3 {  
 public int[] twoSum(int[] nums, int target) {  
 Map<Integer, Integer> numMap = new HashMap<>();  
 int n = nums.length;  
 // Build the hash table  
 for (int i = 0; i < n; i++) {  
 numMap.put(nums[i], i);  
 }  
 // We check if the hash map (numMap) contains the calculated complement.  
 // Additionally, we ensure that the complement’s index is not the same as the current index i.  
 // If both conditions are met, we found a valid pair of indices.  
 for (int i = 0; i < n; i++) {  
 int complement = target - nums[i];  
 if (numMap.containsKey(complement) && numMap.get(complement) != i) {  
 return new int[]{i, numMap.get(complement)};  
 }  
 }  
 // No solution found  
 return new int[]{};  
 }  
}  
  
  
// Note: 1. Return if no pair is found then not found, 2. If found multiple pair then return first pair and pair count,  
// 3, If found then return all pair and pair count, 4. Assume for sorted array and unsorted array  
// Keyword: array.length vs string.length()  
// Learned: Arrays.toString(),  
// Leetcode link: https://leetcode.com/problems/two-sum/