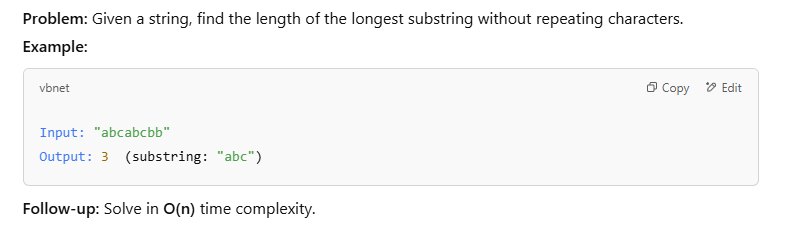
1:- Longest Substring without repeating character



**package** test;

**import** java.util.HashSet;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String str = "abcabcbb";

**int** start = 0;

**int** maxStart = 0;

**int** maxlength = 0;

HashSet<Character> hs = **new** HashSet<>();

**for** (**int** end = 0; end < str.length(); end++) {

**char** ch = str.charAt(end);

**while** (hs.contains(ch)) {

hs.remove(str.charAt(start));

start++;

}

hs.add(ch);

**if** (end - start + 1 > maxStart) {

maxlength = end - start + 1;

maxStart = start;

}

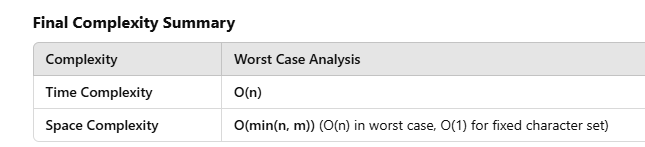
}

String longestSubstring = str.substring(maxStart, maxlength + maxStart);

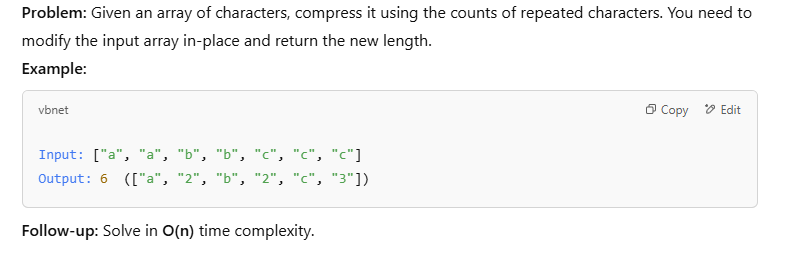
System.***out***.println(longestSubstring);

}

}



2:- String compression



**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String str = "aabbbcccd";

**int** count = 1;

**for** (**int** i = 1; i <=str.length(); i++) {

**if** ( i<str.length() && str.charAt(i) == str.charAt(i - 1)) {

count++;

} **else** {

System.***out***.print(str.charAt(i - 1) + "" + count);

count = 1; // Reset count for the next character

}

}

}

}

**Time Complexity: O(n)**

* **Explanation:**
  + The loop runs once for each character in the string str, starting from index 1 up to str.length().
  + Inside the loop, there are constant-time operations (if check, character comparison, and printing).
  + As a result, the time complexity is **O(n)** where n is the length of the string.
  + **Note:** The charAt(i) operation is **O(1)**, so it does not affect the overall time complexity.

**Space Complexity: O(1)**

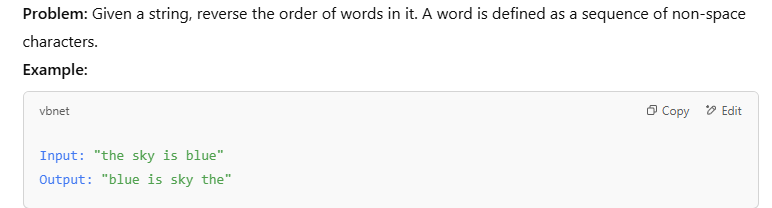
* **Explanation:**
  + The program uses a few **integer variables (count, i)** and prints the result directly. No additional data structures (such as arrays or lists) are used to store intermediate results.
  + The space used does not depend on the size of the input string and remains constant.
  + Thus, the space complexity is **O(1)** (constant space).

**Summary:**

| **Complexity** | **Analysis** |
| --- | --- |
| **Time** | O(n) |
| **Space** | O(1) |

This solution efficiently handles the string compression in **linear time** and **constant space**.

3:- Reverse Words in a String



**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String input = "the sky is blue";

String[] words = input.trim().split("\\s+");

StringBuffer sb = **new** StringBuffer();

**for** (**int** i = words.length - 1; i >= 0; i--) {

sb.append(words[i]).append(" ");

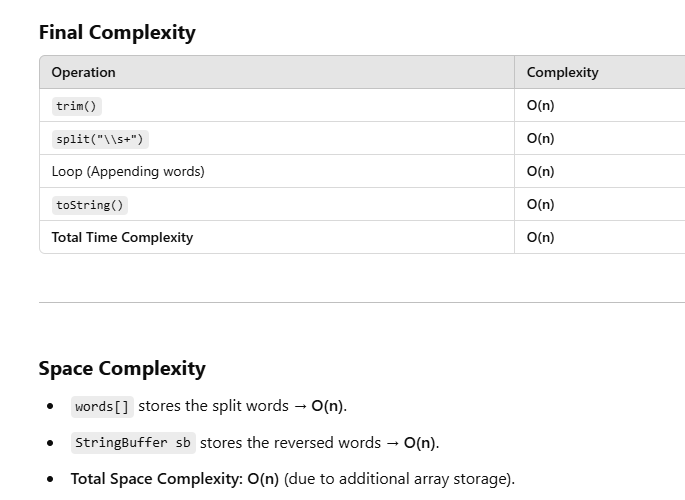
}

System.***out***.println(sb.toString());

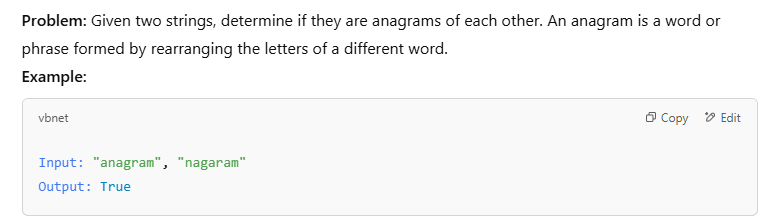
}

}

Time and space complexity is o(n)



4:- Anagram Check



**package** test;

**import** java.util.Arrays;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

System.***out***.println(*isAnagram*("anagram", "nagaram"));

}

**private** **static** **boolean** isAnagram(String s, String t) {

**if** (s.length() != t.length()) {

**return** **false**;

}

**char**[] s1 = s.toLowerCase().toCharArray();

**char**[] s2 = t.toLowerCase().toCharArray();

Arrays.*sort*(s1);

Arrays.*sort*(s2);

**return** Arrays.*equals*(s1, s2);

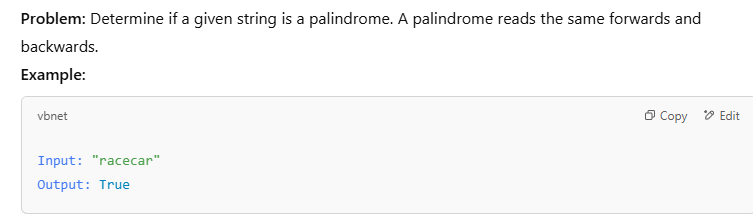
}

}

**Time & Space Complexity**

* **Time Complexity:** O(n log n) → Due to sorting.
* **Space Complexity:** O(1) → Sorting is in-place.

5:- Palindrome Check



**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

System.***out***.println(*isPalindrome*("racecar"));

}

**private** **static** **boolean** isPalindrome(String s) {

**int** start = 0;

**int** end = s.length() - 1;

**while** (start < end) {

**if** (s.charAt(start) != s.charAt(end)) {

**return** **false**;

} **else** {

start++;

end--;

}

}

**return** **true**;

}

}

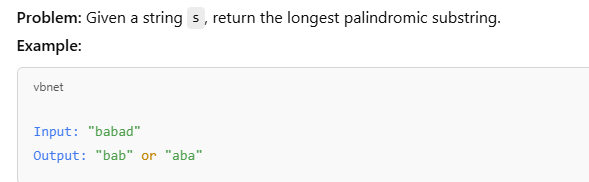
**🕒 Time Complexity:**

* **O(n)** → We compare at most **n/2** characters.

**🛠 Space Complexity:**

* **O(1)** → No extra storage used.

6:- Longest Palindromic Substring



**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String str = "babad";

String longestPalindrome = "";

**for** (**int** i = 0; i < str.length(); i++) {

**for** (**int** j = i + 1; j < str.length(); j++) {

String subStr = str.substring(i, j);

**if** (*isPalindrome*(subStr)) {

**if** (subStr.length() > longestPalindrome.length()) {

longestPalindrome = subStr;

}

}

}

}

System.***out***.println(longestPalindrome);

}

**private** **static** **boolean** isPalindrome(String s) {

**int** left = 0, right = s.length() - 1;

**while** (left < right) {

**if** (s.charAt(left) != s.charAt(right)) {

**return** **false**;

}

left++;

right--;

}

**return** **true**;

}

}

 **Time Complexity**: O(n³)

 **Space Complexity**: O(n)

7:-- Find All Substrings of a String

**package** test;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String str = "abc";

List<String> allSubstring = *findAllSubstring*(str);

System.***out***.println(allSubstring);

}

**private** **static** List<String> findAllSubstring(String str) {

List<String> result = **new** ArrayList<>();

**for** (**int** i = 0; i < str.length(); i++) {

**for** (**int** j = i + 1; j <= str.length(); j++) {

result.add(str.substring(i, j));

}

}

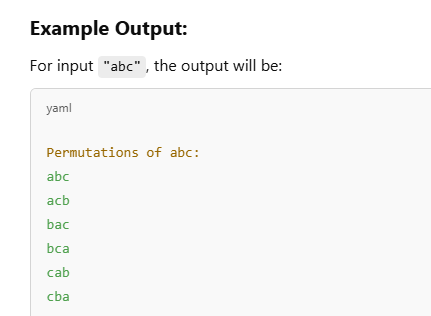
**return** result;

}

}

Output :- [a, ab, abc, b, bc, c]

8:- Permutation of string



**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

String str = "abc";

*findPermutation*(str,"");

}

**private** **static** **void** findPermutation(String ques, String ans) {

**if**(ques.length()==0) {

System.***out***.println(ans);

**return**;

}

**for**(**int** i=0;i<ques.length();i++) {

**char** ch= ques.charAt(i);

String remaining = ques.substring(0,i)+ques.substring(i+1);

*findPermutation*(remaining, ans+ch);

}

}

}

**Time and Space Complexity:**

* **Time Complexity:** **O(n!)** (Factorial time complexity)
  + Each character has n choices initially, then (n-1), (n-2), etc., leading to n! permutations.
* **Space Complexity:** **O(n!)** (for storing recursive call stack)

9:-