Sarah Ahmed

January 22 2026

Assignment 1 – Coding and Complexity

Problem 2: Algorithm for Common Substring (pseudocode)

```
Algorithm LongestCommonSubstring (text1, text2) {
    First convert text1 and text2 to character arrays A and B
    Initialize:  longestLengthSub = “” to store longest common substring
    Next, iterate over every starting position, called i, in A
for i from 0 to A.length - 1
for j from i to A.length - 1
length = j - i  +1
    Next check of this substring is found in B
    For q from 0 to B.length - length
    match = true
    for r from 0 to length –1
    If A[i + r] != B[q + r]
    match = false
    break
            If match == true
    If length > longestLengtSub.length
    longest = substring of A from i to j
            Break
    Return longest
```

Problem 6: Algorithm Analysis

Problem 1:

- Time Complexity is O(2^(m + n)) where m is the length of the first string and n is the length of the second input string, because the algorithm branches off into two more calls when the base case is not met.
- Space Complexity is O(m + n), where m and n are the lengths of the two input strings, because we are storing two input arrays.
- Big-Ω = Ω(m+n) in the best case, where m and n are the lengths of the two input strings. When all characters match, we only make one recursive call per character.

Problem 2:

- Time Complexity is O(n^(3)) where n is the length of the first string and m is the length of the second string, because there are three nested loops that check every substring of the first against all spots in string m
- Space Complexity is O(n+m), where n is the length of the first string, and m is the length of the second string, because the solution iterates over all of the elements one time and stores both arrays.
- Big-Ω = Ω(n^2) in the best case where n is the length of the first string, because a match is found immediately.

Problem 3:

- Time Complexity is O(n), where n is the size of the array, because the loop iterates n-2 times, doing constant-time operations each time.
- Space Complexity is O(n), where n is the size of the array, because an array size of n stores the sequence of terms.
- Big-Ω = Ω(n) where n is the size of the array, because in the best case, it still creates and stores all n elements, no matter what they are.

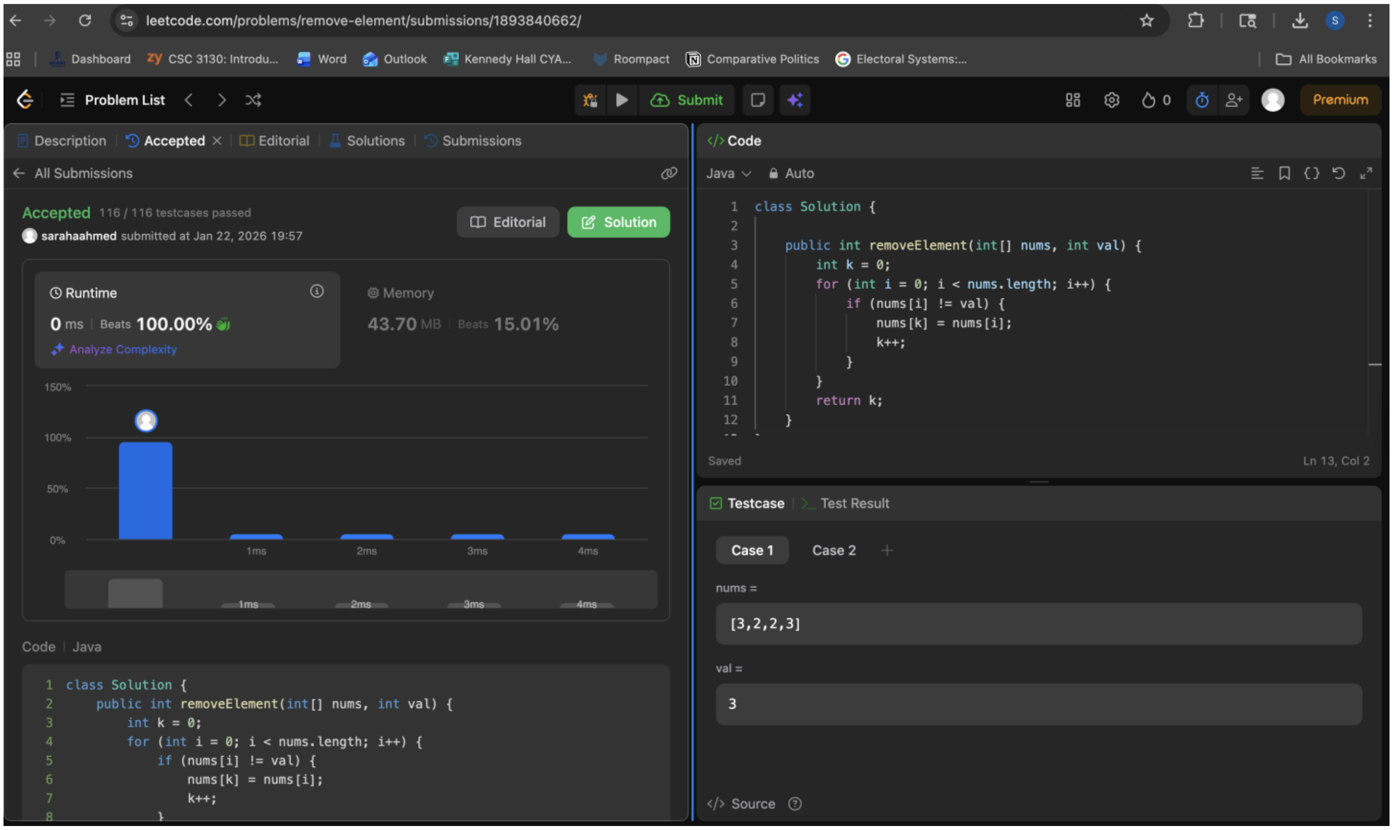
Problem 4:

- Time Complexity is O(n) where n is the size of the array because the NotFibonacci method takes O(n) and binary search takes O(logn), and n is higher order than logn.
- Space Complexity is O(n), where n is the size of the array, because the algorithm stores all n elements of the array.
- Big-Ω = Ω(n) where n is the size of the array, because the algorithm creates all n elements before doing the search.

Problem 5:

- Time Complexity is O(n) where n is the size of the array because each element is iterated through at least one time.
- Space Complexity is O(1), where n is the size of the array, because there are a constant number of variables, and the array is modified in-place.
- Big-Ω = Ω(n), where n is the size of the array, because all elements in it must be looked at least once to see if it will remain.

EC:



Accepted 116 / 116 testcases passed

sarahaahmed submitted at Jan 22, 2026 19:57

Runtime

0 ms | Beats 100.00%

Memory

43.70 MB | Beats 15.01%

Code | Java

```java
class Solution {
    public int removeElement(int[] nums, int val) {
        int k = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != val) {
                nums[k] = nums[i];
                k++;
            }
```

Code

```java
class Solution {

    public int removeElement(int[] nums, int val) {
        int k = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != val) {
                nums[k] = nums[i];
                k++;
            }
        }
        return k;
    }
```

nums =

[3,2,2,3]

val =

3