Problem 2:

Pseudo code:

Input: text1, text2 (Strings)
Output: longest common substring (String)
1. Convert text1 and text2 into character arrays: chars1, chars2
2. Initialize maxLength = 0 and startIndex = -1
3. For i from 0 to length(chars1) - 1:
    4. For j from 0 to length(chars2) - 1:
        5. Initialize length = 0
        6. While i + length < length(chars1) AND j + length < length(chars2) AND
            chars1[i + length] == chars2[j + length]:
            7. Increment length
        8. If length > maxLength:
            9. maxLength = length
            10. startIndex = i
11. If maxLength == 0, return ""
12. Return substring of text1 starting at startIndex with length maxLength

Code:

```
public class LongestCommonSubstring {
    public static String longestCommonSubstring(String text1, String text2) {
        // First, you convert the strings to character arrays
        char[] chars1 = text1.toCharArray();
        char[] chars2 = text2.toCharArray();

        int maxLength = 0; // stores the max length of the common substring
        int startIndex = -1; // stores the starting index of the common substring in text1

        // Iterate through all starting points in text1 and text2
        for (int i = 0; i < chars1.length; i++) {
            for (int j = 0; j < chars2.length; j++) {
                int length = 0; // Length of current common substring

                // Compare characters while they match and within bounds
                while (i + length < chars1.length && j + length < chars2.length &&
                        chars1[i + length] == chars2[j + length]) {
                    Length++; }

                // Update maxLength and startIndex if a longer substring is found
                if (length > maxLength) {
                    maxLength = length;
                    startIndex = i; } } }
        // If no common substring is found, return an empty string
        if (maxLength == 0) {
            return ""; }
```

```
        // Construct the longest common substring from text1
        string result = new String();
        for (int i = startIndex; i < startIndex + maxLength; i++) {
            result.append(chars1[i]); }
        return result.toString();   }

    public static void main(String[] args) {
        String text1 = "spy family";
        String text2 = "jujutzu";
        System.out.println("Longest Common Substring: " + longestCommonSubstring(text1,
text2));

        text1 = "gears of war";
        text2 = "History of warriors";
        System.out.println("Longest Common Substring: " + longestCommonSubstring(text1,
text2));

        text1 = "spy family";
        text2 = "jujutsu kaisen";
        System.out.println("Longest Common Substring: " + longestCommonSubstring(text1,
text2));           } }
```

Problem 6:

- #1: The time complexity in Big-O notation is O(m*n) because it is upper bound & since there is a nested for loop, the other loop iterates m times, m being the length of text1. Then, the inner loop iterates n times, n being the length of text2. The body in each loop are constant time so it is O(1). Thus, the total amount of operations is O(text1.length*text2.length), which is O(m*n). Even in the best case scenario, you still have to iterate through the for loops m and n times so in Big-Ω notation it is still Ω(m*n), since it is still lower bound.
- #2: For the nested for loops, the outer loop runs m times, where m is text1.length, and the inner loop runs n times, where n is text2.length. Thus, in the worst case scenario, the combined complexity of these loops is O(m*n). The best case time complexity is when the characters in the strings have a big difference which causes the loop to end, so it reduces to Ω(m*n) as well.
- #3: The for loop iterates n-2 times, which means the number of iterations is directly proportional to n, when you take out the 2. The atomics inside the loop, or the body, performs multiplication and addition operations and time complexity depends on the size of the number, because the loop runs n times. So the worst case scenario is O(n^2/2), which equals O(n^2), when you cross off the constant. In the best case scenario, it will be the same thing, because the loop still iterates n-2 times and the size of the numbers increases as the sequence continues so in Big-Ω notation, the time complexity is still Ω(n^2).

- #4: In the worst case scenario, the for loop will just iterate through the array until it goes through all of the terms in the sequence, which is 1000. Since it is a constant, the number of iterations is O(1). Inside the loop, each iteration performs multiplication and addition, which means it depends on the size of i. Each iteration is proportional to O(i) & the loop runs n times so the time complexity in Big-O notation would be O(n^2/2) = O(n^2). For the best case scenario, the number of iterations could be less than 1000 and the loop could end early & run fewer iterations so the time complexity would be Ω(m), m being the index/position of the input in the sequence when it stops.
- #5: The loop runs n times, where n is the length of the array nums. In the body of the loop, there are 3 atomics, where each operation takes O(1) per second which is constant. So, the time complexity is O(n). In the best case scenario, the condition in the loop will always be true and the loop would still run through all elements, so the time complexity is Ω(n). Since the number of iterations depends only on n, the size of the array, the time complexity is linear in both best and worst case scenarios.

EXTRA CREDIT:

The problem is the speed of the numbers being executed due to the sequence multiplying previous terms by constants so it grows exponentially. Therefore, this will cause conflict when plotting due to the growth being extremely rapid. To reduce the rapid growth, you could reduce the range of values by taking the logarithm of each term before plotting and showing the growth pattern.

Pseudo code:

Input: Number of terms, n = 1000
Output: Logarithmic plot of the first 1000 NotFibonacci numbers
1. Initialize an array `sequence` of size `n`
2. Set sequence[0] = 0, sequence[1] = 1
3. For i = 2 to n - 1:
    sequence[i] = sequence[i-1] * 3 + sequence[i-2] * 2
4. Initialize an empty array `logValues`
5. For each value in `sequence`:
    If value > 0:
        Append log10(value) to `logValues`
6. Plot the `logValues` array (e.g., using matplotlib in Python)

Code:

```
import matplotlib.pyplot as plt
import math

def generate_not_fibonacci(n):
    sequence = [0] * n
    sequence[0] = 0
    sequence[1] = 1
    for i in range(2, n):
        sequence[i] = sequence[i - 1] * 3 + sequence[i - 2] * 2
```

```
    return sequence

# Generate the first 1000 terms
n = 1000
sequence = generate_not_fibonacci(n)

# Apply logarithm (base 10)
log_values = [math.log10(x) if x > 0 else 0 for x in sequence]

# Plot
plt.figure(figsize=(10, 6))
plt.plot(log_values, label='Log10(NotFibonacci)', color='blue')
plt.title('Logarithmic Plot of NotFibonacci Sequence (First 1000 Terms)')
plt.xlabel('Index (n)')
plt.ylabel('Log10(Value)')
plt.legend()
plt.grid()
plt.show()
```