Austin Tackett

Assignment 1

1. Common Subsequence:

```
private static int LCSequence(char[] s1, char[] s2, int s1Len, int s2Len) {
    if ( s1Len == 0 || s2Len == 0)
        return 0;
    else if (s1[s1Len-1] == s2[s2Len-1])
        return 1 + LCSequence(s1, s2, s1Len: s1Len-1, s2Len: s2Len-1);

    return Math.max(LCSequence(s1, s2, s1Len: s1Len-1, s2Len), LCSequence(s1, s2, s1Len, s2Len: s2Len-1));
}
```

Complexity: O(2^N)

Input: s1.length or s2.length

The worst-case performance of each recursive function call

T(M, N) = k + T(M-1, N) + T(M, N-1)

For arbitrarily large values of M and N, each recursive call creates two additional function calls. Also, half of the branches have a depth of M, and the other half has N. Viewing it as a recursive tree, each root node creates two more child nodes, and if the current layer has N nodes, the next layer in depth will have 2^N nodes (likewise with M). Therefore, assuming a worst-case complexity where N = M, the complexity is O(2^N).

2. Common Substring

```java
public static char[] LCSubstring(char[] s1, char[] s2) {
    int[][] lenMatrix = new int[s1.length][s2.length];
    int largestCol = 0;
    int largestDiagonal = 0;

    // Find length of longest substring
    for (int row = 0; row < s1.length; row++) {
        for (int col = 0; col < s2.length; col++) {
            if(s1[row] == s2[col]) {
                // Look at diagonal direction
                lenMatrix[row][col] = 1;

                // Add diagonal values not outside of matrix bounds
                if (row-1 >= 0 && col-1 >= 0)
                    lenMatrix[row][col] = lenMatrix[row][col] + lenMatrix[row-1][col-1];

                // Find largest diagonal
                if (lenMatrix[row][col] > largestDiagonal) {
                    largestDiagonal = lenMatrix[row][col];
                    largestCol = col;
                }
            }
        }
    }

    // Traverse backwards to find substring by counting columns backwards
    // will not access out of bounds indices of s2 b/c, the largest col <= length of smallest char[] input
    char[] subString = new char[largestDiagonal];
    for (int i = 0; i <= largestDiagonal-1; i++)
        subString[largestDiagonal-1-i] = s2[largestCol - i];


    return subString;
}
```

Complexity: O(N^2)

Input: s1.length or s2.length

The function begins with initializing variables (O(1)). Next, two nested loops occur where the outer loop has a time complexity of O(M), where M = s1.length, and the inner loop has a time complexity of O(N), where N = s2.length, Inside the inner loop, constant time operations occur, so the time complexity of the inner loop is still O(N). The time complexity of both loops together is O(N^2) because worst case, M=N, and the loop must have O(N^2) or O(M^2) operations Finally, another loop occurs after both previous loops end. The loop only iterates the same amount of times as the length of the largest substring. A substring will always be at most the

length of either input. The worst-case substring length occurs when M = N and both the strings are identical; hence, the loop will iterate N times (O(N)). Finally, the return statement is a constant time operation. The final time complexity is O(N^2) + O(N) + O(1), which can be simplified as only O(N^2).

3. Not Fibonacci

```java
public static void notFib(Long n) {
    Long num1 = 0L;
    Long num2 = 1L;
    Long sum = 0L;

    if (n >= 1)
        System.out.println(0);
    if (n >= 2)
        System.out.println(1);

    int i = 3;
    while(i <= n) {
        sum = 3 * num1 + 2 * num2;
        System.out.println(sum);
        num1 = num2;
        num2 = sum;
        i++;
    }
}
```

Complexity: O(N)

Input: n

The function begins with constant time operations before entering a while loop, and assuming the worst case where n > 3, the while loop will iterate times n - 2 with a resulting O(N-2) + O(1) time complexity, which simplifies down to O(N).

4. Remove Element

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

    return newIndex;
}
```

Complexity: O(N)

Input: nums.length

Like previous functions, it begins with constant time operations where variables are initialized (O(1)). Subsequently, a loop occurs that will always iterate n times. Inside the loop are 2 constant time operations; the total time complexity of the loop is O(N + 2) or O(N). Finally, a return statement happens, which is also a constant time operation, O(1). By dropping the constant time operations, the total time complexity of the algorithm is O(N).

5. Where In Sequence

```java
public static int whereInSequence(Long Fn) {
    Long num1 = 0L;
    Long num2 = 1L;
    Long sum = 0L;

    if (Fn <= 0)
        return 1;
    if (Fn == 1)
        return 2;

    int i = 3;
    while(i <= Fn) {
        sum = 3 * num1 + 2 * num2;
        if (sum > Fn) {
            return i-1;
        };
        num1 = num2;
        num2 = sum;
        i++;
    }

    return -1;
}
```

Complexity: O(N)

Input: Fn

Like previous functions, it begins with constant time operations, O(1), where variables are initialized and some initial conditions are checked. The loop has the condition that it iterates while i >= 3 and i < Fn, which would suggest the loop iterates Fn-2 times when Fn >= 2, resulting in a time complexity of O(N). Although Fn grows exponentially compared to n, its position in the sequence is still a linear search algorithm and hence has a time complexity of O(N).

Extra Credit)

Long is an 8-byte data type and stores integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The sequence is growing too fast; hence, numbers above the previously stated upper bound of the data type are experiencing overflows and underflows. The Long data type is stored as a two's complement. The largest non-negative value is 011….1 in binary, but if you add one, that number becomes 100….0, the smallest negative value, and vice versa. That means that once the maximum value is reached, the behavior of the algorithm, after it reaches its first overflow, will be bouncing around within the bounds of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.