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(Log(N))

Input: Fn

Like previous functions, it begins with constant time operations, O(1), where variables are initialized and some initial conditions are checked. Although the loop has the condition that it iterates while i >= 3 and i < Fn, which would suggest the loop iterates Fn-2 times, which would result in a time complexity of O(N), Fn grows exponentially and requires fewer loop iterations to find the position in the sequence that is nearest to Fn.

By representing the notFibonnaci sequence using matrix operations

| Fn+1 | | 2   3 | ^ n | Fn | where Fn = 1 and Fn-1 =0

| Fn | = | 1   0 | | Fn-1 |

After performing singular value decomposition and finding USH^T, the diagonal matrix S can be rewritten as S^n because US^nH^T is equivalent to the previous expression.

| λ1^n 0 |

| 0   λ2^n |

Now, after solving for eigenvalues and vectors, Fn in the bottom row of the original left-hand matrix can be rewritten with the explicit formula ((-3)^(n-1) -1) / (4(-1)^(n-1)) = Fn.

As the equation demonstrates, Fn grows exponentially with respect to n, and thus, n corresponds to the n-2 loops required to find Fn. Thus, the loops executed grow logarithmically with respect to Fn, which results in a time complexity of O(log(N)) + O(1) or O(log(N)).

Extra Credit)

The first 100 entries in the Fibonacci sequence

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 upper bound of the data type are overflowing and becoming values in its lower bound.

0

1

2

7

20

61

182

547

1640

4921

14762

44287

132860

398581

1195742

3587227

10761680

32285041

96855122

290565367

871696100

2615088301

7845264902

23535794707

70607384120

211822152361

635466457082

1906399371247

5719198113740

17157594341221

51472783023662

154418349070987

463255047212960

1389765141638881

4169295424916642

12507886274749927

37523658824249780

112570976472749341

337712929418248022

1013138788254744067

3039416364764232200

9118249094292696601

8908003209168538186

8277265553796062943

6385052587678637212

708413689326360021

2125241067979080062

6375723203937240187

680425538102168944

2041276614306506833

6123829842919520498

-75254544950990121

-225763634852970364

-677290904558911091

-2031872713676733274

-6095618141030199821

159889650618952152

479668951856856457

1439006855570569370

4317020566711708111

-5495682373574427284

1959696952986269765

5879090858958809294

-8094714968331237333

-2428414490499371200

-7285243471498113599

-3408986340784789182

8219785051355184071

6212611080356000596

191089167358450173

573267502075350518

1719802506226051555

5159407518678154664

-2968521517675087623

-8905564553025262870

-8269949585366236993

-6363104682389159364

-642569973457926475

-1927709920373779426

-5783129761121338277

1097354790345536784

3292064371036610353

-8570550960599720558

-7264908808089610057

-3347982350559278556

8402797022031715949

6761646992385596230

1838196903447237075

5514590710341711224

-1902971942684417943

-5708915828053253830

1319996589549790127

3959989768649370380

-6566774767761440475

-1253580229574769810

-3760740688724309429

7164522007536623328

3046821948900318369

9140465846700955106

8974653466393313703