CS3340 Assignment 1

Name: Michael Song

Student Number: 251101048

1. Base Case: The relation holds if h = 0 as the number of leaves is (2^h = 2^0 = 1) and the number of nodes is (2^(h+1) – 1 = 2^(0+1) – 1 = 2 – 1 = 1). Since a tree of height zero should only hold 1 node, the second check is confirmed. Since this node is by default the leaf, the first check is confirmed.

   Induction Case: The relation for leaves holds for a height of h = k + 1 instance as (2^(k+1) = 2^k * 2) matches the description that height of k + 1 creates two trees of height h = k, confirming the first check in regard to leaves. The relation of height of k+1 also holds for number of nodes in the instance of (2^(k + 1 + 1) – 1 = 2^(k + 2) – 1 = 2^k * 4 – 1 = 2(2 * 2^k) – 1 = 2^(k + 1) + 2^(k + 1) – 1 = 2^(k + 1) – 1 + 2^(k + 1) – 1 + 1) due to two binary trees of height h being in the equation, leading to an additional layer producing twice the amount of nodes as the original tree plus an extra node.

2. Base Case: Assume n = 2 as the n > 1 is a requirement. Therefore, LS = L_2 = 3 and RS = F_1 + F_3 = 1 + 2 = 3 and LS = RS.

   Induction Case: $L_{N+1} = F_N + F_{N+2} = (F_{N-1} + F_{N-2}) + (F_{N+1} + F_N) = (F_{N-1} + F_{N+1}) + (F_{N-2} + F_N) = L_N + L_{N-1}$
   Therefore, $L_{N+1} = F_N + F_{N-1}$

3. n! = (n - 0)(n - 1)(n - 2)(n – 3) … (n – (n – 1))

   = n(1 – 0/n) * n(1 – 1/n) * n(1 – 2/n) * n(1 – 3/n) * … * n(1 – (n – 1)/n)

   = n^n * (1 – 0/n) * (1 – 1/n) * (1 – 2/n) * (1 – 3/n) * … * (1 – (n – 1)/n)

   = $n^n \sum_{k=0}^{n-1} (1 - \frac{k}{n})$

   $\lg(n!) = \lg(n^n \sum_{k=0}^{n-1} (1 - \frac{k}{n}))$

   $= \lg(n^n) + \lg(\sum_{k=0}^{n-1} (1 - \frac{k}{n}))$

   $= n\lg(n) + \lg(\sum_{k=0}^{n-1} (1 - \frac{k}{n}))$

   = Θ(n·lg(n))

4. Base Case: If n = 1, T(n) = c_1.
   Induction case:

If $n > 1$, $T(n) = T(n-1) + c_2 * (n-1)/2 + c_3 = (T(n-2)+c_2 * (n-2)/2 + c_3) + c_2 * (n-1)/2 + c_3$
$= T(1) + \cdots + (c_2 * (n-2)/2 + c_3) + (c_2 * (n-1)/2 + c_3) = c_1 + c_2/2 * (1+2+\cdots+(n-1)) + c_3(n-1)$
$= c_1 + c_2/2 * n(n-1)/2 + c_3(n-1)$
$= \Theta(n^2)$

5. a) A is O of B, A is o of B, A is not Ω of B, A is not ω of B, A is not Θ of B
   b) A is O of B, A is o of B, A is not Ω of B, A is not ω of B, A is not Θ of B
   c) A is not O of B, A is not o of B, A is not Ω of B, A is not ω of B, A is not Θ of B
   d) A is not O of B, A is not o of B, A is Ω of B, A is ω of B, A is not Θ of B
   e) A is O of B, A is not o of B, A is Ω of B, A is not ω of B, A is Θ of B
   f) A is O of B, A is not o of B, A is Ω of B, A is not ω of B, A is Θ of B

   a) $A = \lg^k n$, $B = n^{\epsilon}$
   b) $A = n^k$, $B = c^n$
   c) $A = \sqrt{n}$, $B = n^{\sin(n)}$
   d) $A = 2^n$, $B = 2^{n/2}$
   e) $A = n^{\lg(c)}$, $B = c^{\lg(n)}$
   f) $A = \lg(n!)$, $B = \lg(n^n)$

6. Essentially, we are capable of ensuring that the last chip we sort through always ends up being good if we use the correct algorithm as there are always going to be more good chips than bad chips (as the question explicitly states this). We separate the chips into pairs of two. If the atleast one of the chips is found to be bad, we discard both. If both are found to be good, we throw away one from the pair. If we have a remaining chip that is set aside due to both being found as good, we use this chip with the undiscarded chips.

   In doing this, we can guarantee that we arrive at the answer after at least an amount of tests at minimum half of the total number of chips (assuming there are no situations in which both chips read each other as good) due to the fact that there will always be more good chips than bad chips, ensuring the last chip is a good one if the algorithm is applied correctly.

7. **Time Complexity:**

T(2) <= 2c so T(1) <= c

$T(n) <= 2T(n/2) + cn\log_2(n)$

Height =

a) T(n)

b) $cn\log_2(n)$

T(n/2) T(n/2)

c) $cn\log_2(n)$

$c(n/2)\log_2(n/2)$ $c(n/2)\log_2(n/2)$

T(n/4) T(n/4) T(n/4) T(n/4)

d) Last Line: T(2) + T(2) + T(2) + ….

.

.

.

.

.

Totals: a) $cn\log_2(n)$

b) $cn\log_2(n/2) = cn(\log_2(n) - 1)$

c) $cn\log_2(n/4) = cn(\log_2(n) - 2)$

$cn\log_2(n/8) = cn(\log_2(n) - 3)$

.

.

.

.

Last Line: $cn\log_2(n/(n/2)) = cn$

The time complexity seems to be of $O(n * (\log_2 n)^2$ based on the available information as $n * (\log_2 n)^2$ will end up being the common constant among all numbers.

Substitution Method:

$O(n^2)$:

Base Case: $T(2) = 2c <= c(2)^2 = 4c$

Induction Case: $T(n) <= 2T(n/2) + cn\log_2(n)$

$<= 2(n/2)^2 + cn\log_2(n)$

$<= cn^2$ **(possible)**

$O(n\log_2 n)$:

Base Case: $T(2) = 2c <= c2\log_2(2)$

Induction Case: $T(n) <= 2T(n/2) + cn\log_2(n)$

$<= 2(n/2)\log_2(n/2) + cn\log_2 n$

$<= cn\log_2 n$ **(not possible)**

$O(n)$:

Base Case: $T(2) = 2c <= cn = c(2) = 2c$

Induction Case: $T(n) <= 2T(n/2) + cn\log_2(n)$

$<= 2c(n/2) + cn\log_2(n)$

$<= cn$ **(not possible)**

$O(n * (\log_2 n)^2)$:

Base Case: $T(2) = 2c <= c2* (\log_2 2)^2 = 2c$

Induction Case: $T(n) <= 2T(n/2) + cn\log_2(n)$

$<= 2(n/2)(\log_2(n/2))^2 + cn\log_2(n)$

$<= cn* (\log_2 n)^2$ (**possible**)

As $O(n * (\log_2 n)^2$ is much more restrictive than $O(n^2)$, we will choose to use the former to denote time complexity.

8. a)
Code:

```
#include <stdio.h>



long long store1less = 1; //used to be int
long long store2less = 2;
long long curAnswer = 0;

int main() {

      for (int i = 0; i <= 50; i++) {

            if (i == 0) {
                printf("%d\n", store2less);
            }

            if (i == 2) {

                  curAnswer = store1less + store2less;

            }

            if (i > 2) {
                        formula();
            }


            if (i % 5 == 0 && i != 0) {
                   printf("%ld\n", curAnswer);
            }
      }
}

void formula() {
    store2less = store1less;
    store1less = curAnswer;
    curAnswer = store2less + store1less;

}
```

Answers:
2
11

123
1364
15127
167761
1860498
20633239
228826127
2537720636
28143753123

b)

Code:

```
#include <stdio.h>

long long store1less = 1;
long long store2less = 2;
long long curAnswer = 0;



int main() {

	for (int i = 0; i <= 500; i++) {

		if (i == 0) {
		    printf("%d\n", store2less);
		}

		if (i == 2) {

			curAnswer = store1less + store2less;

		}

		if (i > 2) {
		    formula();
		}


		if (i % 20 == 0 && i != 0) {
			printf("%lld\n", curAnswer);
		}
	}
}
```

```
void formula() {
    store2less = store1less;
    store1less = curAnswer;
    curAnswer = store2less + store1less;

}
```

Answers:

2
15127
228826127
3461452808002
52361396397820127
-1139155321138466361
-2795939413257253630
5347811994664660839
-8271524584511622593
-4435149930091013822
8566728179384764591
4955201673824879479
-352179712341101566
-1268672955607851337
-6249783109692392593
362950400237129026
-529246468703212673
-387354483789832153
7082585619549648130
-29558693758691065
6951655735469402015
-7162095078750790846
8210776971223662927
-8789253581020694569
881586227177639938
7548146805174218327

c) For 8.a) real time was 0.002 seconds, user time was 0.001 seconds, and system time was 0.000 seconds. For 8.b) real time was 0.003 seconds, user time was 0.000 seconds, system time was 0.001 seconds. In terms of real time, I would assume that the algorithm of 8.b) proved to be slower as a result of the fact that the program is required to utilize many more loops in order to reach and output the results.

d) No, we cannot compute L_{50} because it falls out of the range that a 4 byte int variable can register due to the amount of digits (arithmetic overflow), leading to a negative number to be produced. We can calculate L_{500} due to the nature of the unique order of the recursive function.