Name: Daniel Xie
Student Number: 251075206

Question 1:

Theorem.

Let T be a complete binary tree of height h. The number of leaves of the tree L(T) is 2^h and the size of the tree N(T) is 2^{h+1}-1

Proof:

Induction base:

for h = 0. T has only one node (*root node*)

L(T) = $2^h = 2^0 = 1$ (*root node*). N(T) = 2^{h+1} - 1 = 2^{0+1} = 2-1 = 1 (*root node*)

Induction hypothesis:

Assume a complete binary tree T of height h, the number of leaves of the tree L(T) is 2^h and the size of the tree N(T) is 2^{h+1}-1

We have to show that a complete binary tree T of height h+1, the number of leaves of the tree L(T) is 2^{h+1} and the size of the tree N(T) is 2^{h+2}-1

Since a complete binary tree of height h + 1 consists of two complete binary trees (T_L and T_R) of height h whose roots are connected to a new root.

Therefore,

L(T) = L(T_L) + L(T_R)

= $2^h + 2^h$ (*by induction hypothesis*)

= 2^{h+1}

N(T) = N(T_L) + N(T_R) + 1 (*root node*)

= (2^{h+1}-1) + (2^{h+1}-1) + 1

= 2(2^{h+1}-1) + 1

= 2^{h+2}-2 + 1

= 2^{h+2}-1 □

Question 2:

Theorem.
Let N_n be a Fibonacci-like number sequence that is defined as $N_n = N_{n-1} + N_{n-2}$, $N_n = 2F_{n+1}$, $n \geq 0$ where $F_i \geq 1$ is the Fibonacci number

The definition of Fibonacci Number: $F_0=0$, $F_1=1$, and $F_n = F_{n-1} + F_{n-2}$ for $n>1$

Proof:

Induction base:

for $N_0 = 2$, $N_0 = 2F_{0+1} = 2F_1 = 2$ (*by definition of Fibonacci*)

for $N_1 = 2$, $N_1 = 2F_{1+1} = 2F_2 = 2(F_1 + F_0) = 2(1+0) = 2$ (*by definition of Fibonacci*)

Induction hypothesis:

Assume N_m is a Fibonacci-like number sequence that is defined as $N_m = N_{m-1} + N_{m-2}$, $N_m = 2F_{m+1}$ is true for $2 \leq m \leq n$ where $F_i \geq 1$ is the Fibonacci number

We have to show that $N_{n+1} = 2F_{n+2}$, $n \geq 0$ where $F_i \geq 1$ is the Fibonacci number

$N_{n+1} = N_n + N_{n-1}$ (*by definition of N_n*, $n \geq 0$)

$= 2F_{n+1} + 2F_{(n-1)+1}$ (*by induction hypothesis,* $2 \leq n-1 \leq n$)

$= 2F_{n+1} + 2F_n$

$= 2(F_{n+1} + F_n)$

$= 2F_{n+2}$ (*by definition of Fibonacci*) □

Question 3:

Exercise 2.3-5 (pp. 44) in the textbook.

Pseudocode:

```
RECURSIVE-INSERTION-SORT (A, n)
    if n > 1 do
        RECURSIVE-INSERTION-SORT (A, n-1) // Recursive call
        Key ← A[n]
        // Insertion
        j ← n – 1
        while j > 0 and A[j] > key do
                A[j + 1] ← A[j]
                j ← j – 1
        A[j + 1] ← key
```

Worse-case running time:

Base case checking (*if n>1*) takes 1

Inserting key A[n] into A[1: n-1] takes n in worse case

Recurrence Relation:

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + n, & n > 1 \end{cases}$$

$T(n) = 1 + 2 + \ldots + (n-1) + n = n(n+1)/2$

Therefore, $T(n) = \Theta(n^2)$

The worse-case running time is $\Theta(n^2)$

Question 4:

Problem 3-2 (pp. 71) in the textbook.

	O	o	Ω	ω	Θ
a.	yes	yes	no	no	no
b.	yes	yes	no	no	no
c.	no	no	no	no	no
d.	no	no	yes	yes	no
e.	yes	no	yes	no	yes
f.	yes	no	yes	no	yes

Justification:

Let A(n) be the function of column A and B(n) be the function of column B, C is a constant and C>0, $\exists\ n_0$ such that $\forall\ n \geq n_0$

(a) A(n) is polylogarithmic function and B(n) is a polynomial function, n>0
so, $A(n) < C \cdot B(n)$

(b) A(n) is polylogarithmic function and B(n) is an exponential function, n>0
so, $A(n) < C \cdot B(n)$

(c) $A(n)=\sqrt{n} = n^{1/2}$ and $B(n)= \sin n$ which is $-1<\sin n<1$; not able to get the result of >, <, or = between A(n) and B(n)

(d) $A(n) > C \cdot B(n)$ when n>0 and n is a very large number ($n \to \infty$)

(e) $A(n) = n^{\log c} = c^{\log n} = B(n)$

(f) $A(n) = \lg(n!) = \Theta(n \lg n)$ and $B(n) = \lg(n^n) = n \lg n$

Question 5:

Problem 4-6 b, c, and d (pp. 122-123) in the textbook.

(b)

Let X be a set to hold all the chips. Let Y be an empty set.
Let GG be all good chips pair set in X, BB be all bad chips pair set in X, and GB be one is good, one is bad chip pair set in X
Let *all_good* be the testing result that both chips are good
Let *not_all_good* be the testing results that either one is good, either one is bad, or all bad
Let c be the taking out chip if the number of chips is odd, initial to null (c=null)

Here are the steps to use ⌊n/2 ⌋ pairwise test to obtain a set with at most $\lceil n/2 \rceil$ chips that still has the property that more than half of the chips are good:

First put all chips into X

If n is an odd number, take c out, so the Length(X) = n-1
If n is an even number, keep as is, so the Length(X)=n

Then start to test

Take 2 chips at a time from X and do the pairwise testing

- If result = *all_good,* add one of the tested *two chips into Y*
- If result = *not_all_good,* do not add any of them to Y

When all chips are tested (Length(X)=0)

If the Length(Y) is even and c != null, then add c to Y

Now we have to show that Length(Y) $\leq \lceil n/2 \rceil$:

If n is odd, Length(X) = n-1; if n is even, Length(X)=n. Therefore, Length(X)≤n

Since we are doing the pairwise test, and when the testing result is *all_good*, we add one of the two chips into Y

so, Length(Y) ≤ Length(X)/2

$$= \begin{cases} \frac{n}{2} = \lceil n/2 \rceil, & n \text{ is even} \\ \frac{n-1}{2} + 1 = \lceil n/2 \rceil, & n \text{ is odd (and we added the chip c into Y)} \end{cases}$$

Therefore, Length(Y) ≤ $\lceil n/2 \rceil$

Now we will prove Length(Y)/2 of chips = good:

From above definition, we get:

Number of good chips in X = Length(GG) + Length(GB)/2

Number of bad chips in X = Length(BB) + Length(GB)/2

Number of good chips > Number of bad chips (∵ *more than n=2 of the chips is good*)

Case n is odd:

if c is a good chip:

Length(GG)+Length(GB)/2 + 1 > Length(BB) + Length(GB)/2
⇒ Length(GG) > Length(BB)

else if c is a bad chip

Length(GG)+Length(GB)/2 > Length(BB) + Length(GB)/2 + 1
⇒ Length(GG) > Length(BB)

Case n is even:

Length(GG)+Length(GB)/2 > Length(BB) + Length(GB)/2
⇒ Length(GG) > Length(BB)

Therefore, Length(GG) > Length(BB)

Now let's look at the chips in Y:

Let GG' be the chips in GG that is added to Y
Let BB' be the chips in BB that is added to Y

Since we only added testing result *all_good* chips into Y, that means it could be either from GG or from BB, but will not include any GB ⇒ Y = GG' + BB'

So, Length(GG') = Length (GG)/2 and Length (BB') = Length (BB)/2 (only add one of two into Y)

∵ we already proved that Length(GG) > Length(BB)
∴ Length(GG') > Length(BB')
and Length(GG') - Length(BB') ≥ 2 (at least two more good chips than bad chips)

We also need to consider the case when c is added to Y:

if c is a good chip (in GG' set) ⇒ Length(GG') > Length(BB')

else if c is a bad chip (in BB' set)

since we know Length(GG') - Length(BB') ≥ 2, even though add one bad c chip, we still have more good chip than bad ones.

Therefore, we have showed that Length(Y) ≤ $\lceil n/2 \rceil$ and Length(Y)/2 of chips = good

(c)

Using the recursive function to implement it.

Base case is n=1 (one chip left)

Input n chips and process the steps in section(b)

Initial set Y with Length(Y) $\leq \lceil n/2 \rceil$

Make a recursive call on set Y to identify one good chip

The recursive relation is

$$T(1) = 1$$

$$T(n) = T(n/2) + (n/2)$$

$$T(n) = n/2 + n/2^2 + n/2^3 + \cdots + n/2^k$$

$$= n \cdot (1/2 + 1/2^2 + 1/2^3 + \cdots + 1/2^k)$$

$$= n \cdot 1$$

$$= n$$

Therefore, $T(n) = \Theta(n)$

(d)

We can reuse function defined in section (c). Since we have already identified one good chip and a good chip always reports accurately whether the other chip is good or bad, we can use this good chip to test the other chips
The good chip needs to go through other n-1 chips, add n-1 to T(n)

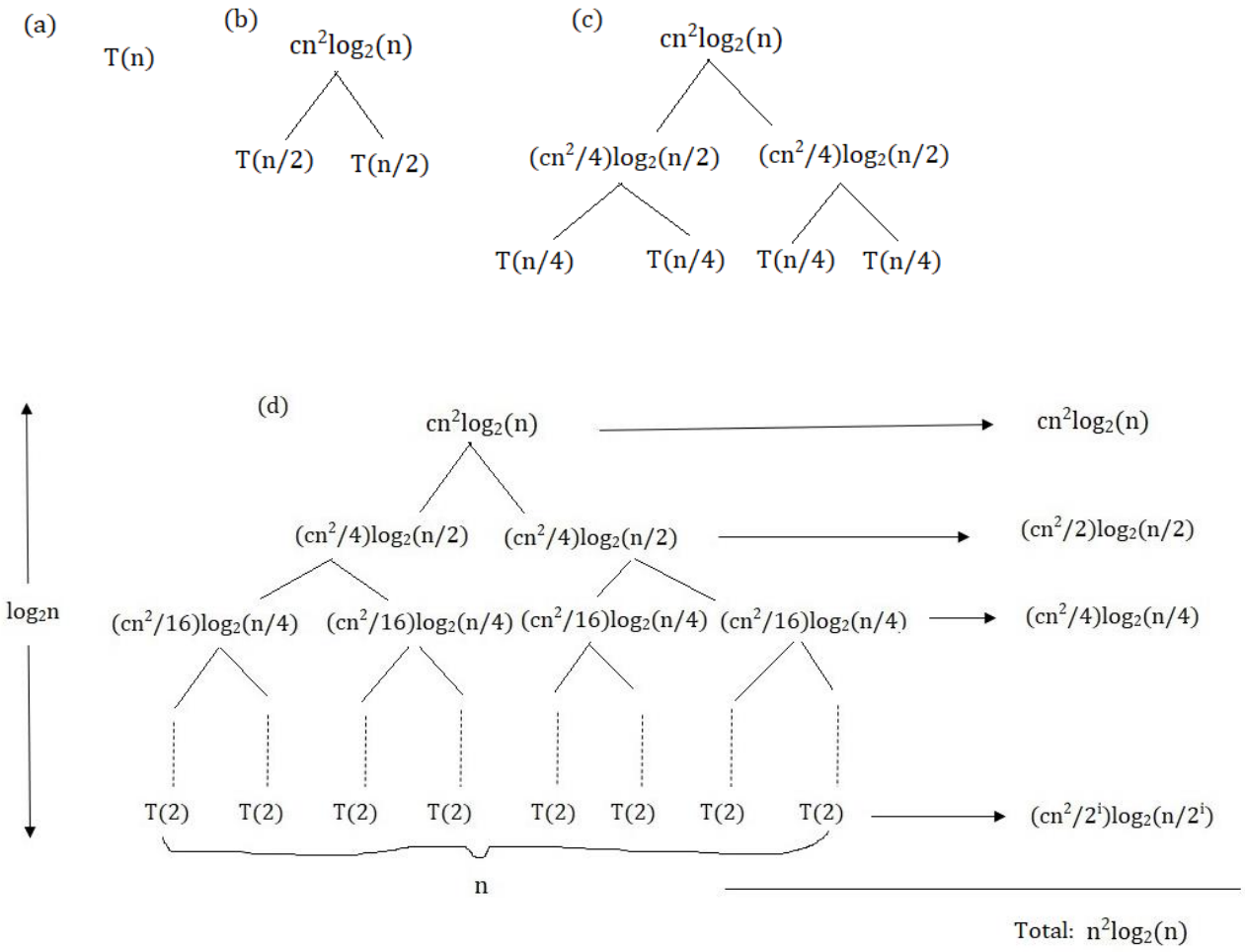
$T(n) = n + n - 1 = 2n$

Therefore, $T(n) = \Theta(n)$

Question 6:

$$T(2) \leq 2c$$
$$T(n) \leq 2T(n/2) + cn^2 \log_2(n) \quad n > 2$$

Recurrence Tree:



Base case is T(2), $n/2^i = 2$, therefore, $i = (\log_2 n) - 1$ the level $= \log_2 n$

Add up costs over all levels to determine cost for the entire tree:

$T(n) = cn^2\log_2(n) + (cn^2/2)\log_2(n/2) + (cn^2/4)\log_2(n/4) + \ldots +$

$(cn^2/2^{\log_2 n - 1})\log_2(n/2^{\log_2 n - 1}) + 2cn$

$= \sum_{i=0}^{(\log_2 n)-1} \left(\frac{c}{2^i}\right) n^2 \log_2\left(\frac{n}{2^i}\right) + 2cn$

$= cn^2 \cdot \sum_{i=0}^{(\log_2 n)-1} \left(\frac{1}{2^i}\right) (\log_2 n - i) + 2cn$

$= cn^2 \cdot (\log_2 n \sum_{i=0}^{(\log_2 n)-1} \left(\frac{1}{2^i}\right) - \sum_{i=0}^{(\log_2 n)-1} \frac{i}{2^i}) + 2cn$

$= cn^2 \cdot (\log_2 n(1+1) - 2) + 2cn$

$= cn^2 \cdot (2 \log_2 n - 2) + 2cn$

$= O(n^2 \log_2 n)$

Proof:

We have to show that $T(n) \leq C \cdot (n^2 \log_2 n)$ for some constant $C > 0$.

Proof by substitution method:

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

$\leq 2c_1((n/2)^2 \log_2(\frac{n}{2})) + cn^2 \log_2 n$ (c_1 is a constant and $c_1 > 0$)

$= 2\, c_1\, ((n/2)^2(\log_2 n - 1)) + cn^2 \log_2 n$

$\leq 2\, c_1\, ((n/2)^2(\log_2 n)) + cn^2 \log_2 n$

$= (c_1/2+c)\, n^2 \log_2 n$

Let $C \geq c_1/2+c$

So, $T(n) \leq C \cdot (n^2 \log_2 n)$ □

Question 7:

(a)

This algorithm is implemented in **asn1_a.java** and the output is recorded in the **asn1.script**

```java
public class asn1_a {
    // Return the Fibonacci-like number by passed n
    public static long fibonacciLike(int n) {
        //base case: N0=2 & N1=2
        if (n == 0 || n == 1) {
            return 2;
        }
        // Recursive call: Nn = Nn-1 + Nn-2
        return fibonacciLike(n - 1) + fibonacciLike(n - 2);
    }

    public static void main(String[] args) {
        for (int i = 0; i <= 10; i++) {
            System.out.println("fib(" + i * 5 + "): " + fibonacciLike(i * 5));
        }
    }
}
```

(b)

This algorithm puts the big integer into an array, every digit is stored as an array item, we add each digit from two array at the same position. So, it iterates from 1 to n (input Fibonacci number position) and does the adding. So the time complexity is O(A(n))

This algorithm is implemented in **asn1_b.java** and the output is recorded in the **asn1.script**

```java
/*
 * Question 7 (b)
 * Return the Fibonacci-like number by passed n
 *
 * Each fibonacci-like number is stored in an array,
 * every single digit is an element of the array.
 * Adding two arrays (fib_1 and fib_2) digit by digit and
 * store the result into array fib to create a new big integer
 */
public static int[] fibonacciLike(int n) {
    // array length for fib, fib_1, and fib-2
    int length = 120;

    //base case: N0=2 & N1=2
    if (n == 0 || n == 1) {
        int[] fib = createArray(length);
        fib[length - 1] = 2;
        return fib;
    }

    // Recursive call: Nn = Nn-1 + Nn-2
    //F(n)
    int[] fib = createArray(length);
    // F(n-1)
    int[] fib_1 = createArray(length);
    // F(n-2)
    int[] fib_2 = createArray(length);

    fib_1[length - 1] = 2;
    fib_2[length - 1] = 2;

    for (int i = 2; i <= n; i++) {
        //F(n) = F(n-1) + F(n-2)
        fib = add(fib_1, fib_2);
        fib_2 = fib_1;
        fib_1 = fib;
    }
    return fib;
}
```

(c)

The algorithm b is much faster than algorithm a according to the time facility track. Algorithm a takes $O(2^n)$ while algorithm b only take O(A(n)) which is for array adding.

```
[dxie32@compute asn1]$ time java asn1_a
fib(0): 2
fib(5): 16
fib(10): 178
fib(15): 1974
fib(20): 21892
fib(25): 242786
fib(30): 2692538
fib(35): 29860704
fib(40): 331160282
fib(45): 3672623806
fib(50): 40730022148

real    1m15.018s
user    1m13.157s
sys     0m0.358s
[dxie32@compute asn1]$
```

```
[dxie32@compute asn1]$ time java asn1_b
fib(0): 2
fib(20): 21892
fib(40): 331160282
fib(60): 5009461563922
fib(80): 75778124746287812
fib(100): 1146295688027634168202
fib(120): 1734001479701589731610384 2
fib(140): 26230240268816379067306864 9732
fib(160): 39678484281238388644956121483 92122
fib(180): 60021642909926907815061334295658979762
fib(200): 90794738833061590639459393939482123846 7652
fib(220): 13734520083255583906104114706164126578641192042
fib(240): 20776208439145982941702103676555080336028407355 1682
fib(260): 31428170368550927563356933170483722962668906019751015 72
fib(280): 47541393108744903733630203389969690960078450775793287927962
fib(300): 71915865041316712192353133034437819810473442861853446451117960 2
fib(320): 10878712857258585944592354700489205612760626741634120068867325911492
fib(340): 16456228867259197917068142763076888295985180261596490566322157455195988 2
fib(360): 24893337298715860116563120111782861920444726054110743863334326893801712235 22
fib(380): 37656151167205192925733052622412507596287854142201519626100930629032275546257412
fib(400): 56962459621697922351597787536292199123076017756460978197295439129193854280806464780 2
fib(420): 86167112293180935469210043948818683389352016097319980297033614509722237080253183810434 42
fib(440): 13034499019627020486729480996540014700015080351965575663071296669590243673936044834197949933 2
fib(460): 19717286580822681597094750434256036287894128659066310208207970175055547095840817940438055053517 22
fib(480): 29826339280465480255655024114604296127297301422418803932300439853093559395975968559140197537475999362
fib(500): 45118303232387266174502539007214414409202264982751638117727773283694925547737376681003197410559393699725 2

real    0m0.103s
user    0m0.074s
sys     0m0.027s
[dxie32@compute asn1]$
```

(d) No. The bytes integer type doesn't work because N50 exceeds the maximum unsigned 4-byte integer value.

Program b can handle N50, or even N500. Since it uses two arrays as a data structure and adds every digit of the big integer into the array as an array item, and then does the addition based on its index, the integer can handle it with no issue.