**Name: Sridevi Nataraj**
**Student ID: 40005572**
**Class: COMP 352**
**Assignment 1**
**Professor: Dr. Dhrubajyoti Goswami**
**Date: Monday, January 30, 2017**

## Part 1: Theory Questions

1. **What is the big-Oh (O) time complexity for the following algorithm (shown in pseudocode) in terms of input size *n*? Show all necessary steps:**

```
Algorithm MyAlgorithm (A,B)  Input:
Arrays A and B each storing n >= 1 integers.
Output: What is the output? (Refer to part b below)
Start:  count = 0
 for i = 0 to n-1 do {
     sum = 0
     for j = 0 to n-1 do {
        sum = sum + A[0]
        for k = 1 to j do
          sum = sum + A[k]
           }
           if B[i] == sum then count = count + 1
 }
return count
```

Answer:

```
    1. count = 0                                  -> O(1)
    2. for i = 0 to n-1 do {                      -> O(n)
    3.  sum = 0                                   -> O(n)
    4.   for j = 0 to n-1 do {                    -> O(n^2)
    5.    sum = sum + A[0]                        -> O(n^2)
    6.   for k = 1 to j do                        -> O(n^3)
    7.  sum = sum + A[k]                          -> O(n^3)
       }
    8. if B[i] == sum then count = count + 1      -> O(n)
 }
    1. return count                               -> O(1)
```

Hence, the time complexity is $O(n^3)$

b)

| | | i=0, Count=0 | i=1, Count=0 | i=2, Count=1 | i=3, Count=1 |
|---|---|---|---|---|---|
| J | K | SUM | SUM | SUM | SUM |
| 0 | X | 0+1=1 | 0+1=1 | 0+1=1 | 0+1=1 |
| 1 | X | 1+1=2 | 1+1=2 | 1+1=2 | 1+1=2 |
| 1 | 1 | 2+2=4 | 2+2=4 | 2+2=4 | 2+2=4 |
| 2 | X | 4+1=5 | 4+1=5 | 4+1=5 | 4+1=5 |
| 2 | 1 | 5+2=7 | 5+2=7 | 5+2=7 | 5+2=7 |
| 2 | 2 | 7+5=12 | 7+5=12 | 7+5=12 | 7+5=12 |
| 3 | X | 12+1=13 | 12+1=13 | 12+1=13 | 12+1=13 |
| 3 | 1 | 13+2=15 | 13+2=15 | 13+2=15 | 13+2=15 |
| 3 | 2 | 15+5=20 | 15+5=20 | 15+5=20 | 15+5=20 |
| 3 | 3 | 20+9=29 | 20+9=29 | 20+9=29 | 20+9=29 |
| B[i] | | B[0]=2<br>29≠2 | B[1]=29<br>29=29 | B[2]=40<br>29≠40 | B[3]=57<br>29≠57 |

The final output would be 1.

2. **Consider the following code fragments (a), (b) and (c) where *n* is the variable specifying data size and *C* is a constant. What is the big-Oh time complexity in terms of *n* in each case? Show all necessary steps.**

   **A) for (int i = 0; i < n; i = i + C)**
   **for (int j = 0; j < 10; j++)**
   **Sum[i] += j * Sum[i];**

Line 1:

| | |
|---|---|
| int i= 0 | It's a constant, Hence O(1) |
| i<n | It takes n+1 times, hence O(n) |
| i=i+c | Takes n times so O(n) |

Total Time: 1+(n+1) +n= 2n+2

Line 2:

| | |
|---|---|
| Int j=0 | It's a constant representation, hence O(1) |
| j<10 | It takes n+10 times, hence O(n) |
| J++ | Loops through 10 times hence O(n) |

Total time: 1+(10+10) n= 1+20n

Line 3:

| | |
|---|---|
| Sum[i]+=j*Sum[i] | Addition and multiplication takes a constant amount of time; hence O(n) |

Total time: 3n

T(n)= (2n+n)+(1+20n)+3n= 25n+3, therefor the answer is O(n)

**B) for (int i = 1; i < n; i = i * C)**
**for (int j = 0; j < i; j++)**
**Sum[i] += j * Sum[i];**

Line 1:

| | |
|---|---|
| Int i=1 | Takes constant amount of time, O(1) |
| I<n | Takes n+1 time, hence O(n) |
| i=i*c | Instead of incrementing by n, it is incrementing by c, hence its O (log n) |

Total time: 1+n+log n

Line 2:

| | |
|---|---|
| Int= j=0 | Takes constant time, O(1) |
| j<1 | Loops through, O(n) |
| j++ | Loops through n number of times. Hence O(n) |

Total time: 1+2n

Line 3:

| | |
|---|---|
| Sum[i] += j * Sum[i]; | 3n |

T(n)= (1+n+log n) +(1+2n) + 3n= 6n+2+logn, hence O(log n)

**C) for (int i = 1; i < n; i = i * 2)**
**for (int j = 0; j < n; j = j + 2)**
**Sum[i] += j * Sum[i];**

Line 1:

| | |
|---|---|
| int i=1 | Takes constant time, O(1) |
| I<n | Loops through, hence O(n) |
| I=i*2 | Instead of incrementing by n, it is incrementing by c, hence its O (log n) |

Total time: 1+n+log n

Line 2:

| | |
|---|---|
| Int j=0 | Takes constant time, O(1) |
| j<n | Loops through, hence O(n) |
| j=j+2 | It increments by 2, hence O(n^2) |

Total time: $1+2n^2$

Line 3:

| | |
|---|---|
| Sum[i] += j * Sum[i]; | 3n |

Total time: 3n

T(n)= 1+n+log n + $1+2n^2$ + 3n is hence, O(n log n)

**3. The number of operations executed by algorithms A and B are $12n^3 + 40n \log n$ and $5n^4 - 100n^2$ respectively. Determine an $n_0$ such that B is greater than A for $n \geq n_0$.**

$12n^3 + 40n \log n = 5n^4 - 100n^2$
$40n \log n = 5n^4 - 100n^2 - 12n^3$
$n=6.08$
$n=0.38$
Solving for n, we get 6.08 ~6, and 0.38.

Hence, $n_0=7$ and $n_0=1$, since for all n>= 7 and n>=1, A will be faster than B. Also, at ~6 and ~1 they are going to be equal.

**4. Answer the following questions:**

**a)**

d(n) is o(f(n)), and e(n) is O(g(n))

➔ d(n) <= k f(n) for n >= N, and e(n) <= L g(n) for n>=M

➔ d(n) +e(n) <= k f(n) + L g(n) <= (K+L) (f(n) +g(n)), for n >= max (N, M)

**b)**

let d(n)= 2n and e(n)=n. Then
d(n)-e(n)=n,
since d(n)= O(n) and e(n) = O(n) we have that
f(n)= g(n) = n,

so O(f(n)-g(n)) = O(n-n) =O(1),
n is not equal to O(1)

**c)**

$2^{n+1} + n^2$, because $2^{n+1} = 2 \times 2^n = O(2^n)$

$2^{n+1} + n^2 = 2 \times 2^n + n^2$

$2 \times 2^n + n^2 <= 2 \times 2^n + 2^n$

and since, $n^2 < 2^n$ for $n \geq 0$ and c=2.

f(n) is $O(2^n)$

**d)**

$f(n) = \sum i^2$

$= (n(n+1)(2n+1))/6) = (((n^2+n)(2n+1))/6)$

$= (2n^3+3n^2+n)/6$

To show that the sum is O(n3) find constant c and k

So; $| (2n^3+3n^2+n)/6| <= c|n^3|$ when $n>k$

For positive integer n;

$(2n^3+3n^2+n)/6 < 2n^2+3n^3+n^3<6n^3$

$c=1, n \geq n_o$

Hence, $T(n)$ is $O(n^3)$

## Part 2: Programming Questions

**Pseudo Codes For:**

***Binary Recursion (BinaryFib)***
**Algorithm** Fibonacci (n)
**Input:** nonnegative number n
**Output:** the $n^{th}$ Fibonacci number

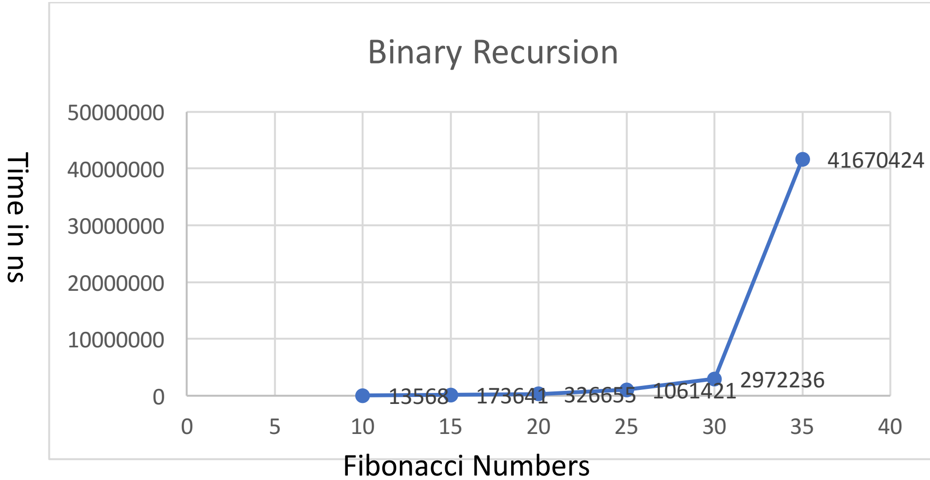
**if** n ≤ 2 **then**
    **return** 1
**else**
    **return** BinaryFib(n-1) + BinaryFib(n-2)

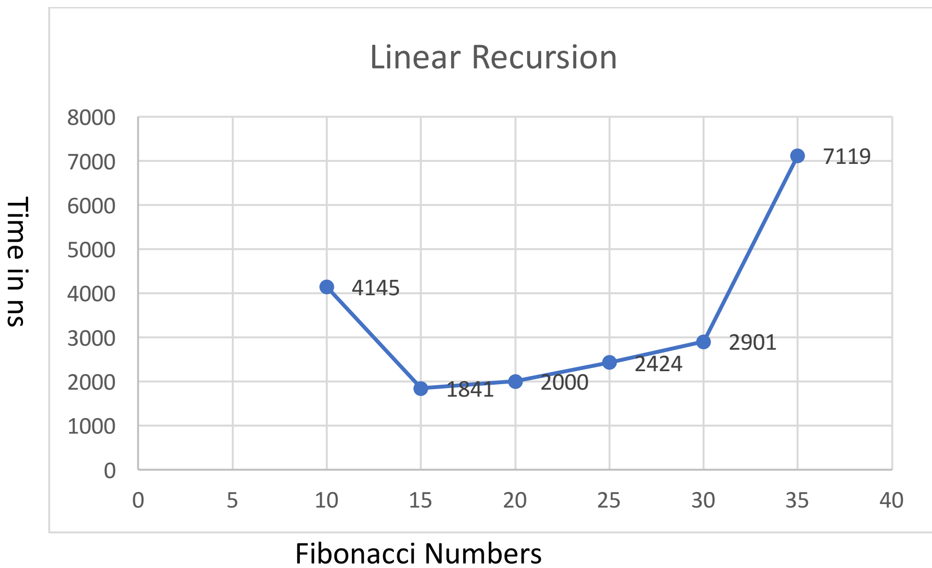
***Linear Recursion (LinearFibonacci)***
**Algorithm** FibonacciLinear(n, a, b)
**Input:** nonnegative number n and the value of the first three Fibonacci numbers
**Output:** the $n^{th}$ Fibonacci number

**if** n = 1 **then**
    **return** the first Fibonacci (a)
**if** n = 2 **then**
    **return** the second Fibonacci (b)
**else**
    **return** Fibonacci (n-1, b (a + b))

**Observation of timing measurements:**

| **Fibonacci number** | **Binary Recursion (nanoseconds)** | **Linear Recursion (nanoseconds)** |
|---|---|---|
| Fibonacci (10) | 13568 | 4145 |
| Fibonacci (15) | 173641 | 1841 |
| Fibonacci (20) | 326655 | 2000 |
| Fibonacci (25) | 1061421 | 2424 |
| Fibonacci (30) | 2972236 | 2901 |
| Fibonacci (35) | 41670424 | 7119 |




**b) Briefly explain why the first algorithm is of exponential complexity and the second one is linear (more specifically, how the second algorithm resolves some specific bottleneck(s) of the first algorithm). You can write your answer in a separate file and submit it together with the other submissions.**

**<u>Binary recursion:</u>**

The $n^{th}$ Fibonacci number depends on the two-preceding values Fibonacci (n - 1), and Fibonacci (n – 2). After computing Fibonacci (n - 2), we must compute it again in the recursive calls of Fibonacci (n – 1). In other words, double work.

**Linear recursion:**

This is much more efficient way to compute Fibonacci numbers, as each invocation makes only one recursive call. This algorithm runs in O(n) time.

c) **Do any of the previous two algorithms use tail recursion? Why or why not? Explain your answer.**

The second algorithm (linear recursion) uses tail recursion because it makes its recursive call as its last step.