Case Study #1

Members:

Chua, Kyle Matthew C.

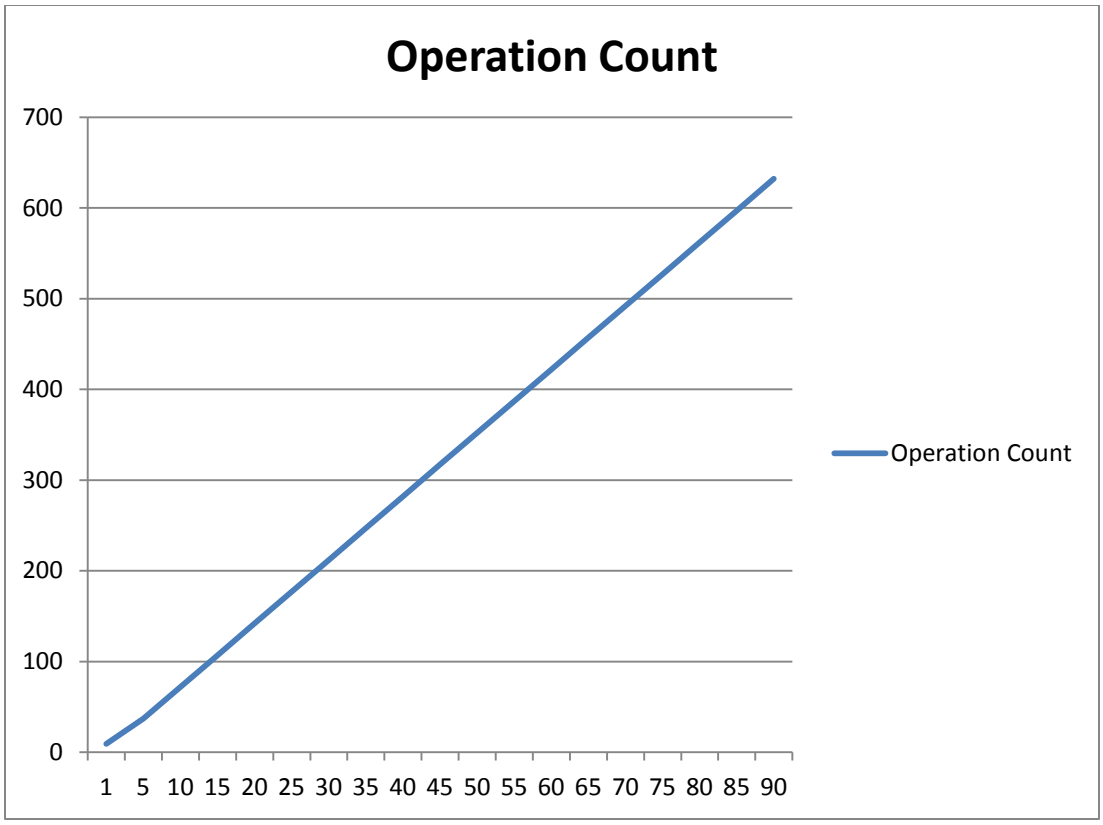
Santos, Kyle-Althea Francesca Anne M.

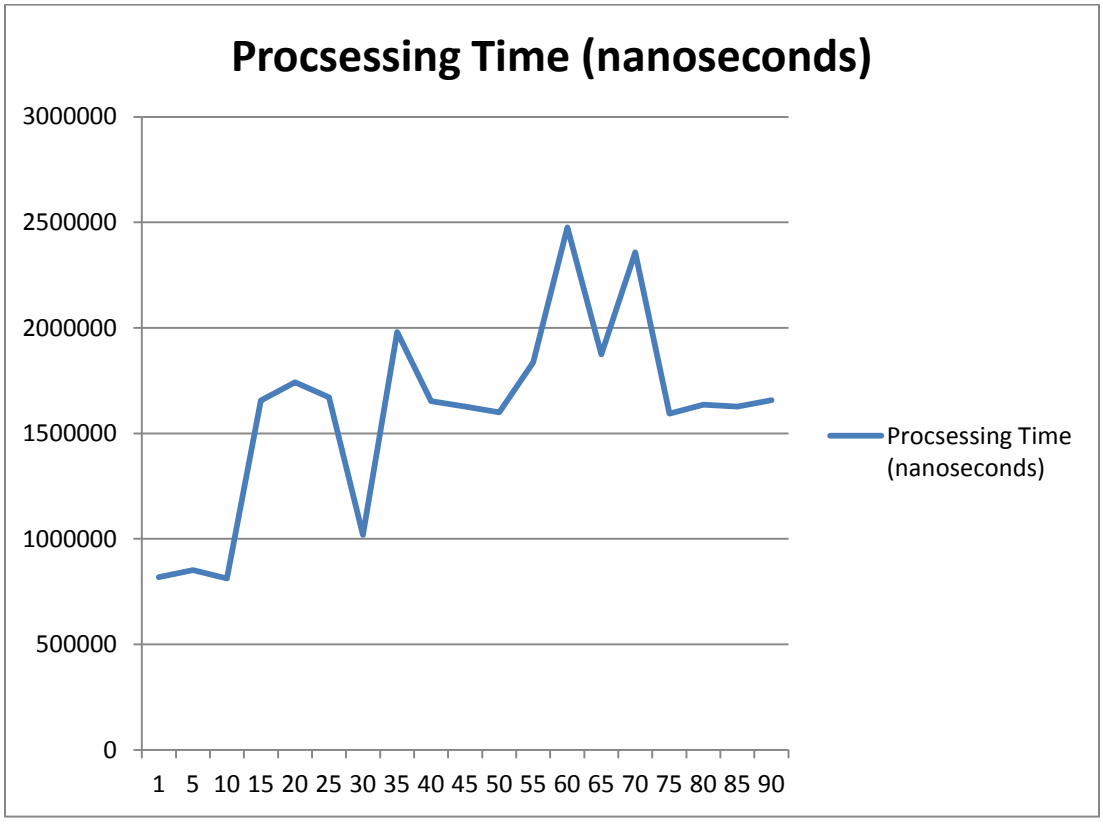
DASALGO S11

Non-Recursive Function

- **The data and the graphs of the performance of the non-recursive function:**

Performance of non-recursive program		
n	Operation Count	Processing Time (nanoseconds)
1	9	818097
5	37	851180
10	72	812899
15	107	1655100
20	142	1741588
25	177	1670696
30	212	1018959
35	247	1980731
40	282	1653209
45	317	1626743
50	352	1600276
55	387	1835638
60	422	2476977
65	457	1874393
70	492	2357879
75	527	1593660
80	562	1636668
85	597	1627215
90	632	1656990


Operation Count
700
600
500
400
300
200
100
0
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
Operation Count


Procsessing Time (nanoseconds)
3000000
2500000
2000000
1500000
1000000
500000
0
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
Procsessing Time (nanoseconds)

- **Computing the frequency count for each line of the function:**

```
import java.util.Scanner; ----------(0)

public class chua_santos{ ----------(0)
        public static void main(String[] args){ ----------(0)
        Scanner input = new Scanner(System.in); ----------(1)

        long a = 1, ----------(1)
        b = 1, ----------(1)
        temp; ----------(0)


        System.out.print("Enter nth number: "); ----------(1)
        int n = input.nextInt(); ----------(1)


        int ctr = 1; ----------(1)
        long time1 = System.nanoTime(); ----------(1)
        for(int i=1; i<n; i++){ ----------(1 + n + n - 1) = (2n)
                temp = a + b; ----------(n-1)
                a = b; ----------(n-1)
                b = temp; ----------(n-1)
                ctr += 7; ----------(n-1)
        }
        ctr++; ----------(1)
        System.out.println("\n" + n + " Fibonnaci number is: \n" + a); ----------(1)

        long time2 = System.nanoTime(); ----------(1)
        long elapsed = time2 - time1; ----------(1)
        System.out.println("\nTotal operation count: " + ctr); ----------(1)
        System.out.println("Elapsed time: " + elapsed + " nanoseconds"); ----------(1)


        input.close(); ----------(1)
        }
}
```

Total frequency count of the function: 6n+10.

The Big-Oh of the function is O(n).

To prove that the total frequency count is indeed the Big-Oh we have concluded:

$$G(n) <= c\ f(n)$$

$$6n + 10 <= cn$$

We then assign the value of 7 to c and we get.

$$6n + 10 <= 7n$$

$$10 <= 7n - 6n$$

$$10 <= n$$

We then test the equation for different values of n:

- **If n = 10**

  6(10) + 10 <= 7(10)

  70 <= 70 (True)

- **If n = 11**

  6(11) + 10 <= 7(11)

  76 <= 77 (True)

So the frequency count and big-Oh is true iff c = 7 and n > = 10.

1. **What observations can be made from it? Correlate the count and processing time on all runs made (hint: look at the table and graphs defined)**

We have observed that as the input increases, the operation count also increases. Mainly because as the input gets bigger, the more times it has to undergo the loop; meaning more operations needed to be done. Since the operation count increases as the input gets bigger, it should take more time to generate the nth fibonacci number but as our data and graph show, the running time is inconsistent. It does not explicitly show that as n increases, the running time also increases because the nanoTime() method is not stable; therefore it is giving different measurements on different runs (Yang, 2015).

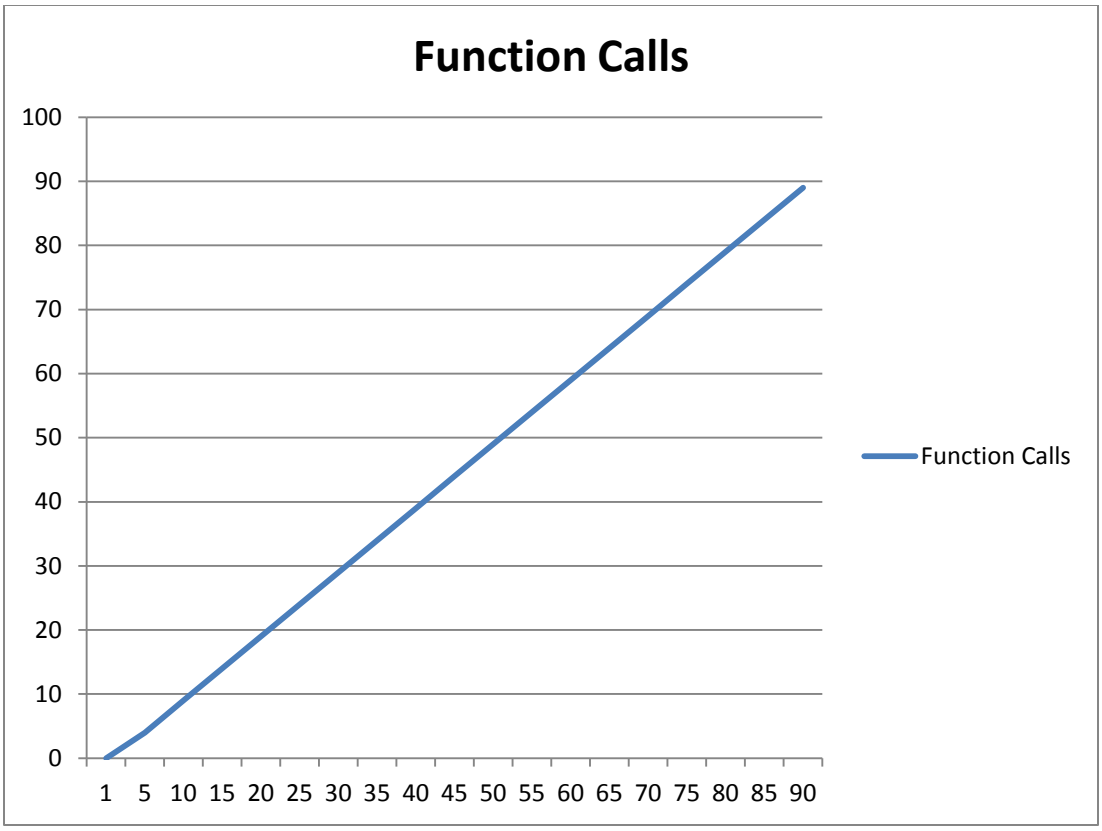
2. **Correlate the graphs with the big-Oh defined. Do these two information consistent in illustrating the efficiency of the non-recursive Fibonacci function? Explain elaborately.**

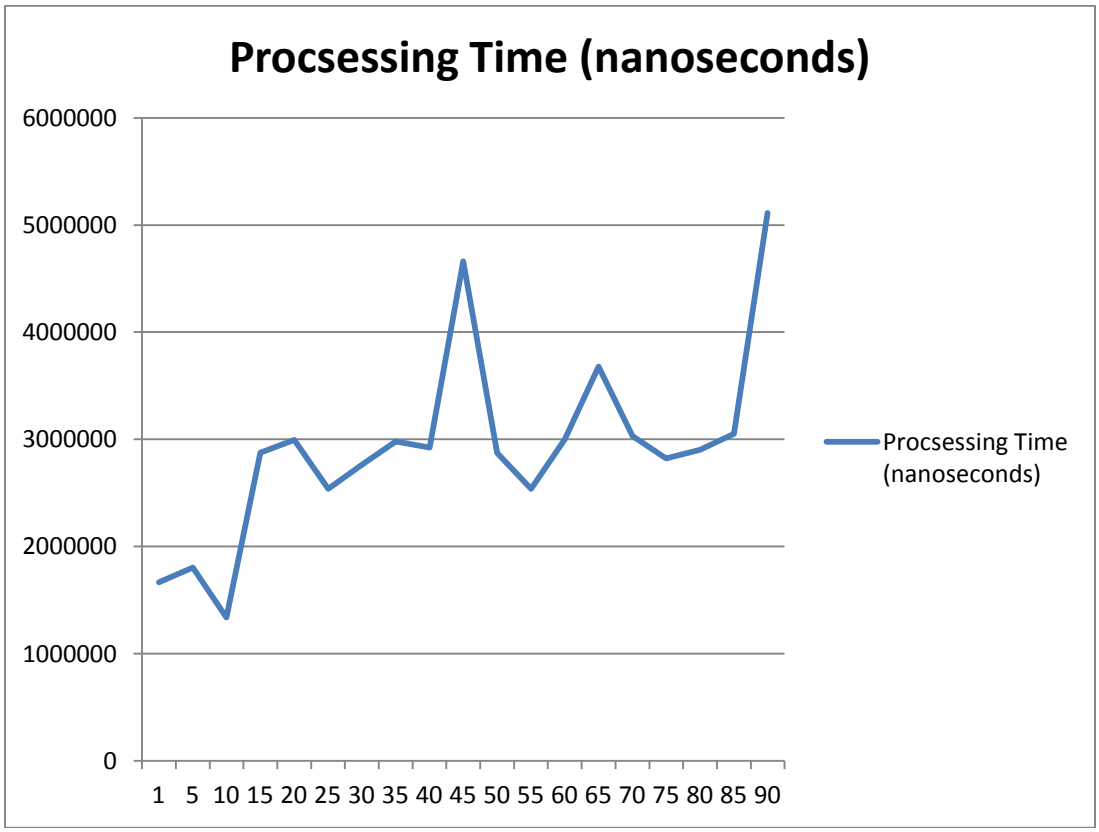
The graph of the operation count and the big-Oh defined is consistent as the graph shows. As the n increases, the operation count also increases showing an increasing trend in the graph. On the other hand, the graph of the processing time does not show complete consistency with the big-Oh defined, but bringing aside the faulty results, the graph still shows an increasing trend.

Recursive Function

- **The data and the graphs of the performance of the non-recursive function:**

Performance of recursive function		
	Function Calls	Processing Time (nanoseconds)
1	0	1665024
5	4	1802556
10	9	1338447
15	14	2875865
20	19	2993547
25	24	2537945
30	29	2762438
35	34	2981258
40	39	2923127
45	44	4660934
50	49	2871139
55	54	2539363
60	59	2997800
65	64	3679311
70	69	3030883
75	74	2822933
80	79	2903277
85	84	3050733
90	89	5112755


Function Calls
100
90
80
70
60
50
40
30
20
10
0
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
Function Calls


Procsessing Time (nanoseconds)
6000000
5000000
4000000
3000000
2000000
1000000
0
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
Procsessing Time (nanoseconds)

From our recursive function,

```
public static void getFibonacci(long a, long b, int n, int ctr){
        if(n==1){
                System.out.println(a);
                System.out.println("\nTotal function call/s: " + ctr);
        }
        else
                getFibonacci(b, a+b, n-1, ctr+1);
}
```

the recurrence of our function is:

$$T(n) = \begin{cases} 3, & n = 1 \\ T(n-1) + 4, & n > 1 \end{cases}$$

T(n) = T(n-1) + 4 [1st call]
T(n-2) + 4 + 4 [2nd call]
T(n-3) + 4 + 4 + 4 [3rd call]

ith = T(n-i) + 4i

Since our base case states that n should be equal to 1, then

n – i = 1
n = i + 1
i = n – 1

We then substitute n with i + 1 and we obtain,

T(i + 1 – i) + 4i
T(1) + 4i

We then substitute I with n – 1 and also T(1) with 3 since if n is 1, the frequency count is 3

3 + 4(n-1)
3+4n – 4
4n – 1

We have obtained the closed-form of our recurrence relation which is **4n - 1** and the big-Oh is O(n).

1. **What observations can be made from it? Correlate the count and processing time on all runs made (hint: look at the table and graphs defined)**

Similar to the non-recursive function, as n increases, the function call also increases as shown in the graph. However, same with the non-recursive function, the processing time is still not consistent because of the factor we have stated before.

2. **Correlate the graphs with the big-Oh defined. Do these two information consistent in illustrating the efficiency of a recursive Fibonacci function? Explain elaborately.**

The graph of the function call and the big-Oh defined is consistent as the graph suggests. As the n increases, the function call also increases showing an increasing trend in the graph. On the other hand, the graph of the processing time does not show complete consistency with the big-Oh defined, but bringing aside the faulty results, the graph still shows an increasing trend.

3. **What can you conclude in terms of efficiency given the 2 implementations of Fibonacci? Kindly elaborate/justify your answer.**

We therefore conclude that the non-recursive function is more efficient than the recursive function because based on the data gathered, the processing time of the non-recursive function is less than that of the recursive function in all inputs of n. Another reason why recursive function is less efficient is because it takes up more memory space than the non-recursive function.

References:

Yang, D. (2014, January 1). Current Time in Milliseconds and Nanoseconds. Retrieved February 2, 2015, from http://www.herongyang.com/JVM/System-Current-Time-in-Milliseconds-and-Nanoseconds.html

Contributions of each member:

Chua, Kyle Matthew C.

- Created the recursive program
- Researched on how to get timestamp in nanoseconds
- Graphing
- Proved big-Oh of non-recursive and recursive
- Written the discussion of non-recursive and recursive

Santos, Kyle-Althea Francesca Anne M.

- Created non-recursive program
- Written the discussion of non-recursive and recursive
- Got the Processing Time of both non-recursive and recursive