Jonathan Nicolosi

CS 325-400 Summer 16

Homework Assignment 1

1.

We want to find where $8n^2 = 64n \lg n$. This reduces to $n = 8 \lg n$.

n	8lg(n)
1	0
2	8
4	16
8	24
16	32
32	40
64	48

We can see from this table that insertion sort beats merge sort for all values of n below 32, and for values of n above 64 merge sort beats insertion sort. Evaluating the functions for n in the interval (32, 64), we find that for all values of n < 44 insertion sort beats merge sort.

2.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg n	$2^{1,000,000}$	$2^{60,000,000}$	$2^{360,000,000}$	$2^{8,640,000,000}$	$2^{259,200,000,000}$	$2^{94,608,000,000,000}$	$2^{9,460,800,000,000,000}$
$\sqrt{n}$	$1\text{x}10^{12}$	$3.6\text{x}10^{15}$	1.30E+17	7.46E+19	6.72E+22	8.95E+27	8.95E+31
n	1,000,000	60,000,000	360,000,000	8,640,000,000	259,200,000,000	94,608,000,000,000	9.46E+15
n lg n	62746.1	2.80142×10^6	1.50958×10^7	3.06479×10^8	7.88410×10^9	2.30374×10^12	1.99171×10^14
n^2	1000	7745.966692	18973.66596	92951.60031	509116.8825	9726664.382	97266643.82
n^3	100	391.486	711.37	2051.97	6375.95	45566.17	211499.47
2^n	19.932	25.838	28.423	33.008	37.915	46.427	53.071
n!	9	11	12	14	15	16	17

3.

Base Step:

If n = 2, then T(2) = 2 and 2lg2 = 2

Thus, T(2) = 2lg2

Hypothesis Step:

Assuming T(n) = nlgn is true if $n = 2^k$ for some integer k > 0

Induction step:

If $n = 2^{k+1}$, then

$T(2^{k+1})$

$= 2T(2^{k+1}/2) + 2^{k+1}$

$= 2T(2^k) + 2^{k+1}$

$= 2(2^k \lg(2^k)) + 2^{k+1}$

$= 2^{k+1}((\lg(2^k))+1)$

$= 2^{k+1}\lg(2^{k+1})$

4.

a. O(g(n)) because f(n) grows faster

b. Ω(g(n)) because g(n) grows faster

c. O(g(n)) and Ω(g(n)) so therefore also Θ(g(n)) because $\log_{10}(n) = (\frac{1}{\lg(10)})\lg(n)$

d. Ω(g(n)) because f(n) always grows faster

e. O(g(n)) and Θ(g(n)) because there exists a k1 and a k2 that will cause f(n) to be tightly bound.

f. O(g(n)) because f(n) is bound above by g(n)

g. O(g(n))

h. Ω(g(n))

5.

First sort the list using an algorithm like Merge Sort that sorts in n time. Then perform a binary search on the sorted list.

```
SortedArray = {12, 3, 4, 15, 11, 7};

int i = 0;

int j = 5;

while(i<j){

        if(SortedArray[i]+SortedArray[j] == 20){

                return true; //found a pair

        }

        else if(SortedArray[i]+SortedArray[j] > 20){

                j--;

        }

        Else

                i++;

}

return false; //pair does not exist
```

6.

a.

Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. This means that there exist constants $c_1, c_2 > 0$ such that $f_1(n) \le c_1g_1(n)$ and $f_2(n) \le c_2g_2(n)$ for all n > 0 integers. To prove the claim, we must find some constant c_3 that causes $f_1(n)+f_2(n) \le c_3 [g_1(n) + g_2(n)]$ for all n > 0 integers.

$f_1(n) + f_2(n) \le c_1g_1(n) + c_2g_2(n)$

$\le \max(c_1, c_2)g_1(n) + \max(c_1, c_2)g_2(n)$

$\le \max(c_1, c_2) [g_1(n) + g_2(n)]$

$= c_3 [g_1(n) + g_2(n)]$

We've found a $c_3 = \max(c_1, c_2)$ that satisfies the definition of big-Oh, proving the claim.

b.

This is true. If there exists a constant such that $f_1(n)$ will always lie below and to the right of $c*g_1(n)$ and there exists a constant such that $f_2(n)$ will always lie below and to the right of $cg_2(n)$, then there must be a constant that we can multiply with $\frac{g1(n)}{g2(n)}$ such that $\frac{f1(n)}{f2(n)}$ will always lie below and to the right of it.

c.

This means that there exists positive constants c_1, c_2, and n_0 such that,

$$0 \leq c_1(f_1(n) + f_2(n)) \leq \max(f_1(n), f_2(n)) \leq c_2(f_1(n) + f_2(n)) \text{ for all } n \geq n_0$$

Selecting $c_2 = 1$ clearly shows the third inequality since the maximum must be smaller than the sum. C_1 should be selected as ½, since the maximum is always greater than the weighted average of $f_1(n)$ and $f_2(n)$.

7.

a.

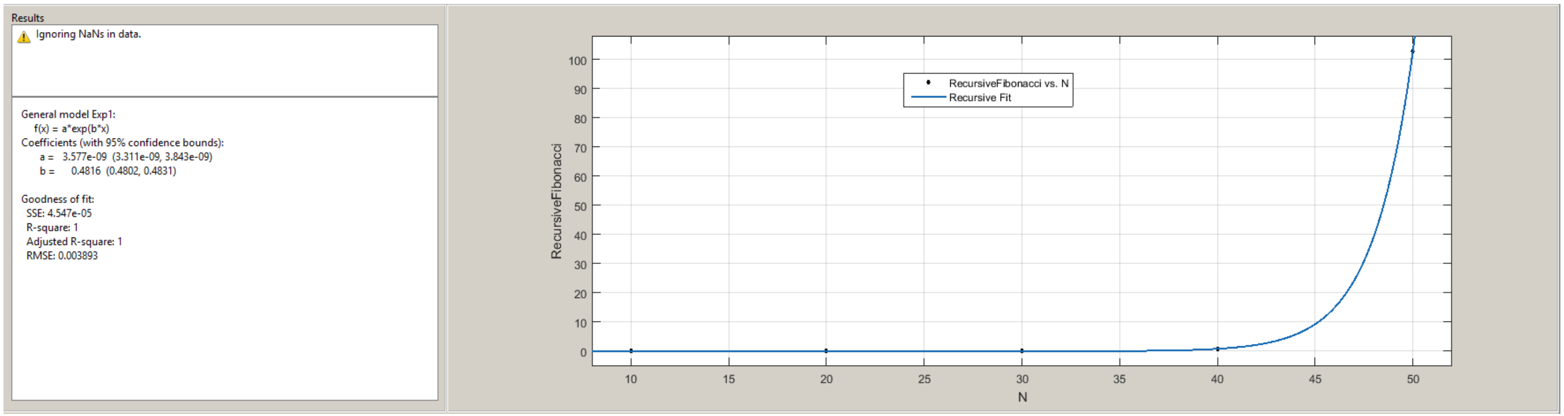
```
unsigned long long int fibRecursive(int n){
    if(n==0){
        return 0;
    }
    else if(n==1){
        return 1;
    }
    else{
        return fibRecursive(n-1) + fibRecursive(n-2);
    }
}
unsigned long long int fibIterative(int n){
int fib = 0;
```

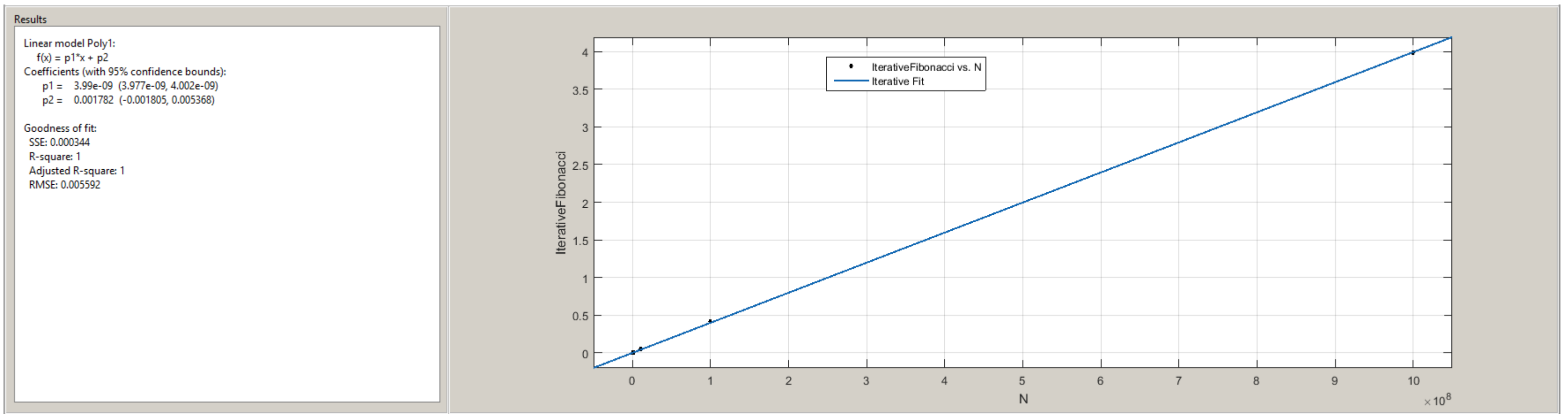
```
    int a = 1;

    int t = 0;

    int k;

    for(k = 0 ; k<n; k++){

        t = fib + a;

        a = fib;

        fib = t;

    }

    return fib;

}
```

b.

N	Iterative Fibonacci	Recursive Fibonacci
10	0	0
20	0	0
30	0	0
40	0	.833
50	0	102.888
100	0	Indeterminate
1,000	0	Indeterminate
10,000	0	Indeterminate
100,000	0	Indeterminate
1,000,000	0	Indeterminate
10,000,000	.053	Indeterminate
100,000,000	.413	Indeterminate
1,000,000,000	3.99	Indeterminate

c.





d.

The recursive Fibonacci is best fit by an exponential function:

f(x) = a*exp(b*x) where

a = 3.577e-09 (3.311e-09, 3.843e-09)

b = .4816 (.4802, .4831)

The iterative Fibonacci is best fit by a polynomial function:

f(x) = p1*x + p2 where

p1 = 3.99e-09 (3.977e-09, 4.002e-09)

p2 = .001782 (-.001805, .005368)

Recursive Fibonacci is much slower because you are adding redundant calls by recalculating the same values over and over again.