Travis Robinson
CS325
Homework 1

1) For values of 43 and less, insertion sort beats merge sort, as shown in the table below.

n	Insertion	Merge
40	12800	13624.135923
41	13448	14058.21646
42	14112	14494.549232
43	14792	14933.080605
44	15488	15373.759438
45	16200	15816.536917
46	16928	16261.366399
47	17672	16708.203266
48	18432	17157.004802
49	19208	17607.730071
50	20000	18060.339807

2)

	second	minute	hour	day	month	year	century
lg n	2^1000000	2^6000000	2^3600000000	2^86400000000	2^2592000000000	2^31536000000000	2^315360000000000
sqrt(n)	1.00E+12	3.60E+15	1.29600000000E+19	7.46E+21	6.72E+24	9.60E+32	9.95E+30
n	1000000	60000000	3600000000	86400000000	2592000000000	31536000000000	3153600000000000
n lg n	62746	2801417	133378058	2755147513	71870856404	797633893349	68654697441062
N^2	1000	7745.9666924	60000	293938.769133981	1609968.94379985	5615692.29926284	56156922.9926284
N^3	100	391.48676412	1532.6188647871	4420.8377983685	13736.57091064	31593.8245690286	146645.543330724
2^n	19.931568569	25.838459165	31.7453497605	36.3303122613	41.2372028569	44.842064915	51.4859211048
n!	9	11	12	13	15	16	17

3)
Base Case: T(2) is 2, as provided in the problem
Base Case is True

Inductive Case:
Assume that for $n=2^k$ $k>1$, $T(n) = 2T(n/2)+n = n \lg n$
$= 2T(2^k/2)+2^k = 2^k \lg 2^k$
For n = k+1:
$2T(2^{k+1}/2)+2^{k+1} = 2T(2^k)+2^{k+1} = 2*2^k \lg 2^k + 2^{k+1} = 2^{k+1} \lg 2^k + 2^{k+1} = 2^{k+1} (\lg 2^k + 1)$
$= 2^{k+1} (\lg 2^k + \lg 2) = 2^{k+1} (\lg 2^k * 2)$
$= 2^{k+1} (\lg 2^{k+1})$
Inductive case is true, proving that the recurrence is correct

4)
a. O(g(n)) because $n^{.5}$ will always yield a larger number than $n^{.25}$ for n>1, meaning that as n gets large, no matter what the constant is, $n^{.5}$ will always surpass $n^{.25}$
b. Ω(g(n)) because for n, $\log^2$ n there will always be a point where n surpasses, regardless of c
c. Θ(g(n)), because lg can be converted to log, which requires a coefficient, and then there are also coefficients that can cause it to be a bound above or below.
d. Ω(g(n)), e is greater than 2, so for all n>1, f(n) will always end up greater regardless of constant
e. Θ(g(n)), $2^{n+1} = 2*2^n$, so that we know there are coefficients that can cause to be an upper or lower

boundary (ie 3 for an upper boundary, 1/3 for a lower)
f. O(g(n)), 2^n will always yield a smaller value than 2^{2^n} for all n, so it will always surpass regardless of constants
g. O(g(n)), once n>4, n! will always surpass 2^n, so that there's no constant that can prevent n! from growing larger than 2^n
h. Ω(g(n)), the only way to make (n+1)! smaller than n! for all n is to divide (n+1)! by (n+1), which is not a constant

5)
My algorithm starts by sorting the given set, using mergeSort which has n*lg(n) time complexity. Then it goes through each element of the set, and adds it to the last element. If it's smaller than x, it moves on, because there's no reason to search if adding it to the largest number still makes it too small. If it's larger than x, it uses a method similar to a binary search to find if one of the other values in the set will work; that is it creates a search range from the element to and adds the one in the middle to see if they sum to x. If it does, it stops. If it's less, it divides the search range in half and checks the lower half. If it's more, it checks the upper half. It continues checking through this divide and conquer method until it rules out all elements or finds a pair that sums to x. It then moves on to the second element and repeats.

```
mergeSort(S)
sum(S,x)
        for (index = 0 to length)
                min = index
                max = S.length

                while (S[index]+S[(min+max)/2] != x || index != (min+max)/2)
                        if (index >= S.length)
                                break
                        if (index + (min+max)/2 > S.length)
                                break
                        if (S[index]+S[S.length] > x)
                                break
                        if (index == (min+max)/2)
                                min = min+1
                        if (min > max)
                                break
                        if (min == max)
                                if (index == (min+max)/2)
                                        break
                                else if (S[index]+S[(min+max)/2] == x)
                                        return
                                else
                                        break

                        if (S[index]+S[(min+max)/2] == x)
                                return

                        else if (S[index]+S[(min+max)/2] < x)
                                min = (min+max+1)/2
```

```
                else if (S[index]+S[(min+max)/2] > x)
                        max = (min+max-1)/2
```

Demonstration:
S = {12,3,4,15,11,7}
After mergeSort:
S= {3,4,7,11,12,15}

```
Index 1 (the first index, I know in most languages it would be 0):
        Min = 1
        Max = 6
        S[index]+S[S.length] = 3 + 15 = 18 < 20
                break
Index 2:
        Min = 2
        Max = 6
        S[index] +S[S.length] = 4 + 15 = 19 < 20
                break
Index 3:
        Min = 3
        Max = 6
        S[index] +S[S.length] = 7 + 15 = 22 >20
                don't break
        S[index] + S[(min+max)/2] = S[index] + S[4] = 7 + 11 = 18 < 20
        Min = (min+max+1)/2 = (3+6+1) = 5
        Max = 6
        S[index] + S[(min+max)/2] = S[index] + S[5] = 7 + 12 = 19 < 20
        Min = (5+6+1)/2= 6
        Max = 6
        S[index] + S[(min+max)/2] = S[index] + S[6] = 7 + 15 = 22 >20
                min == max
                break
Index 4:
        Min = 4
        Max = 6
        S[index]+S[S.length] = 11 + 15 = 26 >20
                don't break
        S[index] + S[(min+max)/2] = S[index] + S[5] = 11 + 12 = 23 > 20
        Min = 4
        Max = (min+max-1)/2 = (4+6-1)/2 = 4
        Min == Max
        Min incremented
        Min > Max
                break
```

Index 5:
Min = 5
Max = 6
S[index]+S[S.length] = 12 + 15 = 27 >20
don't break
Index == (min+max)/2
Min incremented
Min = 6
Max = 6
Min == Max
S[index] + S[S.length] = 12 + 15 = 27 >20
break
Index 6:
Min = 6
Max = 6
Min == Max
index == (min+max)/2
break

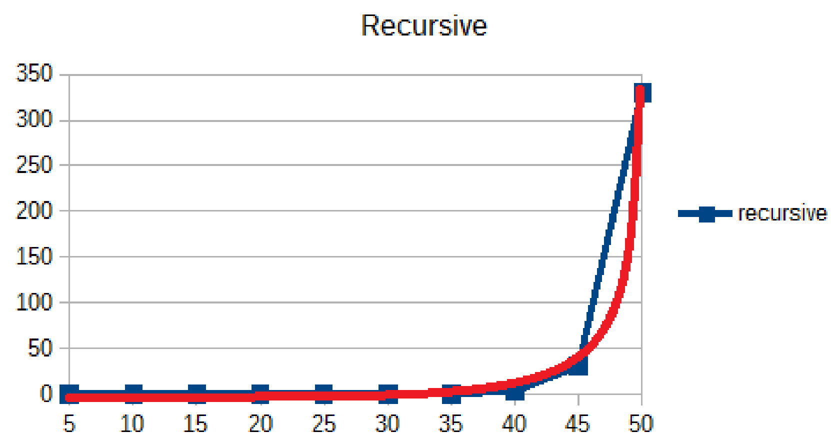
6)
a. This is not true; O-complexity is the upper limit and it's not possible for functions to be the upper limit of each other, unless they both have Θ-complexity.
So f1(n) = O(f2(n)) does not imply f2(n) = O(f1(n))
Also f1(n) = O(f2(n)) iff f2(n) = Ω(f1(n)), which is not O(f1(n))
b. This is correct; when looking at complexity, we look at the highest order (ie $n^2 + n$ would be $O(n^2)$).
Since we're multiplying, the orders of f_n may change, but so will the order of g_n, and since g_n both g_n are larger, multiplying them will only increase the magnitude of the difference, per the rules of algebra.
c. This is incorrect; Θ is also the bottom-most boundary, the upper-most boundary is O

7)
a.
```
#include <stdio.h>
#include <time.h>

using namespace std;

int fibRecur (int n)
{
   if (n == 0)
   {
      return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return (fibRecur(n-1)+fibRecur(n-2));
   }
```

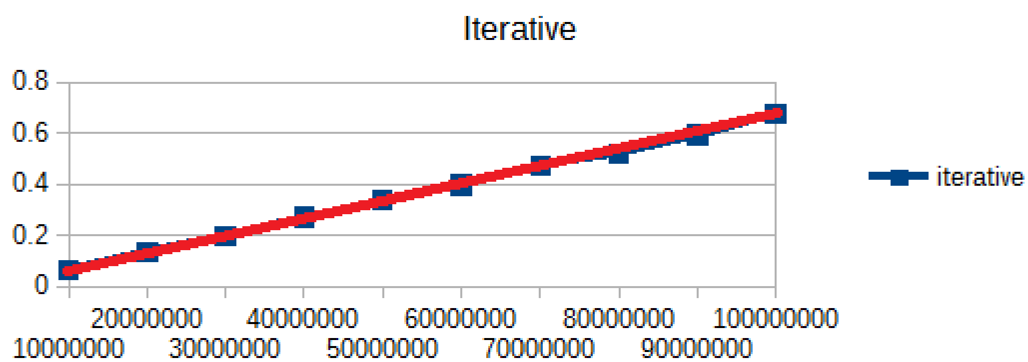
```
}

int fibIter (int n)
{
    int fib = 0;
    int a = 1;
    int t = 0;
    for (int k = 1; k < n; k++)
    {
        t = fib + 1;
        a = fib;
        fib = t;
    }
    return fib;
}

int main ()
{
    int n, result;

    for (int count = 1; count < 11; count++)
    {
        clock_t tStart = clock();
        fibIter(count*10000000); //change to fibRecur
        printf("n = %d",(count*10000000));
        printf("Time taken: %.10fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    }
    return 0;
}
```

c.

n	recursive
5	0
10	0
15	0
20	0.001
25	0.003
30	0.023
35	0.288
40	3.564
45	31.56
50	329.637



n	iterative
10000000	0.065
20000000	0.135
30000000	0.199
40000000	0.271
50000000	0.337
60000000	0.397
70000000	0.473
80000000	0.522
90000000	0.596
100000000	0.678



For the iterative data set, a linear function (y = cx) works best. For the recursive, it looks like a factorial function (y = c(n!) due to the sudden and sharp growth. (The red lines on the graph show what the functions look like)

It's interesting the large difference between these two methods. I would have thought that they'd both be linear, since they're doing the same thing; I figured they'd be linear because doing arithmetic is largely a constant function for computers. I was surprised that the recursive function took so much longer; however, it makes sense that it does. While the iterative function is just performing arithmetic, that is not the case with the recursive; this is because it needs to keep track of each function call that is made, all the way down to the base case, and then it must do the arithmetic as it backtracks all the way back to n. This will be a large drain on system resources as n gets larger, so it makes sense that it will take a while to complete this algorithm.

Sources:
Referred to http://clrs.skanev.com/index.html for confirming answers