Colleen Minor
HW #1

1) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8nˆ2 steps, while merge sort runs in 64nlgn steps. For which values of n does insertion sort beat merge sort?

First, we simplify the equation for the values of n for which merge sort time <= insertion sort time:

:

8 * n * n <= 64 * n * $\log_2(n)$.
8 * n * n <= 8 * 8 * n * $\log_2(n)$.
n <= 8 * log2(n).

Looking at the number line on Wolfram Alpha…

https://www.wolframalpha.com/input/?i=n+%3D+8+*+log(2,+n).

It looks like merge and insertion sort are at the same speed when there is just above 1 n and when there is between 43 and 44 n; the insertion sort beats the merge sort for integer values 2-43.

2) (CLRS) Problem 1-1 on pages 14-15. Fill in the given table. Hint: It may be helpful to use a spreadsheet or Wolfram Alpha to find the values.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg n	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{25920 \cdot 10^8}$	$2^{315360 \cdot 10^8}$	$2^{31556736 \cdot 10^8}$
$\sqrt{n}$	10^{12}	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$994519296 \cdot 10^{18}$	$994,519,296 \cdot 10^{18}$
n	10^6	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$31,536 \cdot 10^9$	$31,556,736 \cdot 10^8$
n lg n	62,746	2,801,417	133,378,058	2,755,147,513	71,870,856,404	797,633,893,349	68,654,697,441,062
n^2	1,000	7,745	60,000	293,938	1,609,968	5,615,692	56,175,382
n^3	100	391	1,532	4,420	13,736	31,593	146,677
2ˆn	19	25	31	46	41	44	51
n!	9	11	12	13	15	16	17

4) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.

a. f(n) = n^0.75; g(n) = n^0.5

https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7Bx%5E%7B0.75%7D%7D%7Bx%5E%7B0.5%7D%7D%5Cright)

Since the solution is infinity (meaning f(n) is growing faster):

f(n) is Ω(g(n).

This is what you would expect, since obviously f(n) puts n to a higher exponent, which means that we can expect higher exponential growth.

b. f(n) = n; g(n) = log n ^2

https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7Bx%7D%7B%5Clog_%7B10%7D%5Cleft(x%5Cright)%5Ccdot%5Clog_%7B10%7D%5Cleft(x%5Cright)%7D%5Cright)

Since the solution is infinity (meaning f(n) is growing faster),

f(n) is Ω(g(n).

This makes sense, since log n ^2 can be translated to,
log10(n) * log10(n),
and obviously 10 to the power of whatever number will result in an answer of n is going to be smaller than n.

c. f(n) = log n; g(n) = lg n

https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7B%5Clog_%7B10%7D%5Cleft(x%5Cright)%7D%7B%5Clog_%7B2%7D%5Cleft(x%5Cright)%7D%5Cright)

the limit, 0.30103, is greater than 0, therefore

$f(n)\ is\ \Theta(g(n))$

We can test this out:
$\log2(5)\ * 0.30103 = 0.698970014 \approx \log10(5).$

d. f(n) = e^n; g(n) = 2^n
f(n) = 2.72^n and g(n) = 2^n. Since 2.72 is greater than 2, f(n) will grow at a faster rate than g(n).

So the limit of e^n/2^n as n is infinity, meaning

$f(n) = \Omega(g(n)).$

e. f(n) = 2^n; g(n) = 2^(n-1)
https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7B2%5E%7Bx%7D%7D%7B2%5E%7Bx%20-1%7D%7D%5Cright)

the limit is greater than 0, therefor

$fn\ is\ \Theta(g(n))$

f. f(n) = 2^n; g(n) =2^2^n

https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7B2%5E%7Bx%7D%7D%7B2%5E%7B2%5Ex%7D%7D%5Cright)

The limit is 0, therefor,

$f(n)\ is\ O(g(n))$

g. f(n) = 2^n; g(n) = n!

https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7B2%5E%7Bx%7D%7D%7Bx!%7D%5Cright)

The limit is 0, therefor,

$f(n)\ is\ O(g(n))$

h. f(n) = nlgn; g(n) = $n\sqrt{n}$
https://www.symbolab.com/solver/limit-calculator/%5Clim_%7Bx%5Cto%5Cinfty%7D%5Cleft(%5Cfrac%7Bx%20%5Ccdot%20%5Clog_%7B2%7D%5Cleft(x%5Cright)%7D%7Bx%20%5Ccdot%20%5Csqrt%7Bx%7D%7D%5Cright)

The limit is 0, therefor,

$f(n)\ is\ O(g(n))$

5) Design an algorithm that given a list of n numbers, returns the largest and smallest numbers in the list. How many comparisons does your algorithm do in the worst case? Instead of asymptotic behavior suppose we are concerned about the coefficients of the running time, can you design an algorithm that performs at most 1.5n comparisons? Demonstrate the execution of the algorithm with the input A= [9, 3, 5, 10, 1, 7, 12].

Q5 Part 1: Design an algorithm that given a list of n numbers, returns the largest and smallest numbers in the list. How many comparisons does your algorithm do in the worst case?

First, a 2n comparison algorithm:

```
var maxMinNumFinder= function(arr){
   var max = arr[0];
   var min = arr[0];
   for(var i = 1; i < halfLength; i++){
     if(arr[i] > max){
        max = arr[i];
     }
    if(arr[i] < min){
        min = arr[i];
     }


   }
   return [max, min];
};
```

The above loop will make exactly 2n comparisons— one for each number, compared for both greater-than and less-than in separate if statements. Because the coefficient 2 makes little difference when n is very large, we drop it, making the worst-case for this one O(n) .

Q5 Part 2: Instead of asymptotic behavior suppose we are concerned about the coefficients of the running time, can you design an algorithm that performs at most 1.5n comparisons?

A 1.5n + 1 comparison algorithm:

```
var maxMinNumFinder= function(arr){
   var max = arr[0];
   var min = arr[0];
   var highNums = {};
   var lowNums = {};

//Loop that makes 1/2 n comparisons:
for (i = 0, i < arr.length, i+=2)
{
```

```
        a = arr[i];
        b = arr[i+1];
         if (a > b)
         {
                highNum.push(a);
                 lowNum.push(b);
         }
        else    {
                highNum.push(b);
                lowNum.push(a);
         }
}//end for loop

//If odd number in array, add last value to smaller list (1 comparison)
if (arr.length % 2 != 0){

        lowNumList.push(arr[arr.length - 1]));
        highNumList.push(arr[arr.length - 1]));


}
```

//Loop that makes a total of 1n comparisons: For the two sublists, each
// of length 1/2n, a comparison is made for each item, totaling n comparisons:

```
for (i = 0, i < Math.floor(arr[i]/2), i++)
{
   if (highNumList[i] < highNumList[i+1])
      max = highNumList[i+1]
   if (lowNumList[i] > lowNumList[i+1])
      min = lowNumList[i+1]
}

return [min, max];
} //end of function/algorithm
```

Q5 Part 3: Demonstrate the execution of the algorithm with the input **A= [9, 3, 5, 10, 1, 7, 12]**.

First for-loop:

1st iteration:

1 comparison.
highNumList{9} ;
lowNumList{3} ;

2nd iteration:

2 comparisons.
highNumList{9, 10} ;
lowNumList{3,5} ;

3rd iteration:
2 comparisons.
highNumList{9, 10, 7} ;
lowNumList{3,5, 1};

Comparison to determine that list has odd number of elements, add last value (12) to both lists:
highNumList{9, 10, 7, 12} ;
lowNumList{3,5, 1, 12};

Second for-loop:

First iteration:
2 comparisons. Max = 10, min = 3.

Second iteration:
2 comparisons. Max = 10, min = 1.

Third iteration:
2 comparisons. Max = 12, min = 1.

This is a total of 12 comparisons. 7 times 1.5 = 10.5 ≈ 11. 11+ 1 = 12.

6) Let f_1 and f_2 be asymptotically positive functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$.

What "$f_1(n)+f_2(n)$ is $O(g_1(n)+g_2(n))$" means is that there is an upper bound, meaning that there must exist positive numbers C and N where, when n >= N, $f_1(n)+f_2(n) < C * g_1(n)+g_2(n)$.

Let's start from here:

$f_1(n) = O(g_1(n))$

From this, we know that there exists a value C_1 of which $f_1(n) \leq C_1 * g_1(n)$.

$f_2(n) = O(g_2(n))$

From this, we know that there exists a value C_2 of which $f_2(n) \leq C_2 * g_2(n)$.

Putting these two together,

$f_1(n)+f_2(n) \leq c1g1(n) + c2g2(n)$

Because multiplying by a larger coefficient should yield a larger result, it is safe to say that

$f_1(n)+f_2(n) \leq \max(c1, c2)g1(n) + \max(c1, c2)g2(n)$

.

The right-hand side of the equation has the same coefficient (which is max(c1, c2)) for the two products being added, so we can simplify it to:

$f_1(n)+f_2(n) \leq \max(c1, c2)\,[g1(n) + g2(n)]$

We'll call whatever max(c1, c2) is c3:

$f_1(n)+f_2(n) \leq c3[g1(n) + g2(n)]$

This satisfies the statement that we originally set out to prove ($f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$) according to the definition of big-O, proving the statement to be true.

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) / f_2(n) = O(g_1(n) / g_2(n))$
False. Counter example:
Let's say that f1 = x and g1 = x^2, while f2 = x^2 and g2 = x^3.

These conditions do fulfill the first two conditions, making $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ true.

However, the condition that we are trying to prove, that

$$f_1(n) / f_2(n) = O(g_1(n) / g_2(n))$$

is **not** true with these values, which we can see here:

$$\lim_{x\to\infty} \left(\frac{\frac{x}{x^2}}{\frac{x^2}{x^3}} \right) = 1$$

Thus we have proved through the use of counter-example that statement b is false.

7) Fibonacci Numbers:

a) Implement both recursive and iterative algorithms to calculate Fibonacci Numbers in the programming language of your choice. Provide a copy of your code with your HW pdf. We

will not be executing the code for this assignment. You are not required to use the flip server for this assignment.

//RECURSIVE

```
#include <iostream>
#include <cstdlib>

using namespace std;

int rFibNum(int x);

int main(int argc, char *argv[])
{
    int nth =  (atoi)(argv[1]);

    cout << "The Fibonacci number at position " << nth
     << " is: " << rFibNum(nth)
    << endl;

    return 0;
}

int rFibNum(int x) {
    if (x == 0)
        return 0;

    if (x == 1)
        return 1;

    return rFibNum(x-1)+rFibNum(x-2);
}
```

//ITERATIVE

```
#include <iostream>
#include <cstdlib>

using namespace std;
int fibonacci(int n);

int main(int argc, char *argv[])

{
    int nth =  (atoi)(argv[1]);

    int fibo = fibonacci(nth);
    cout << "The Fibonacci number at position " << nth
     << " is: " << fibo << endl;

    return 0;
```

```
}

int fibonacci(int n){
    int oldFib = 0, curFib = 1, temp, idx, fib;
    for (idx = 0; idx < n; idx++){
     temp = oldFib + curFib;
     oldFib = curFib;
     curFib = temp;
    }
    return oldFib;
}
```

b) Use the system clock to record the running times of each algorithm for n = 5, 10, 15, 20, 30, 50, 100, 1000, 2000, 5000, 10,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. If you program in C your algorithm will run faster than if you use python. The goal of this exercise is to collect run time data. You will have to adjust the values of n so that you get times greater than 0.

Recursive version:

n = 5:

```
real 0m0.007s
user 0m0.002s
sys  0m0.003s
```

n = 10:

```
real 0m0.006s
user 0m0.002s
sys  0m0.002s
```

n = 15:

```
real 0m0.007s
user 0m0.002s
sys  0m0.003s
```

n = 20:

```
real 0m0.006s
user 0m0.002s
sys  0m0.003s
```

n = 30:

```
real 0m0.012s
user 0m0.007s
sys  0m0.003s
```

n = 50:

```
real 1m17.919s
user 1m17.721s
sys  0m0.130s
```

Iterative version:

n = 5:

```
real	0m0.006s
user	0m0.002s
sys	0m0.003s
```

n = 10:

```
real	0m0.006s
user	0m0.002s
sys	0m0.003s
```

n = 15:

```
real	0m0.007s
user	0m0.002s
sys	0m0.003s
```

n = 20:

```
real	0m0.006s
user	0m0.002s
sys	0m0.003s
```

n = 30:

```
real	0m0.006s
user	0m0.002s
sys	0m0.003s
```

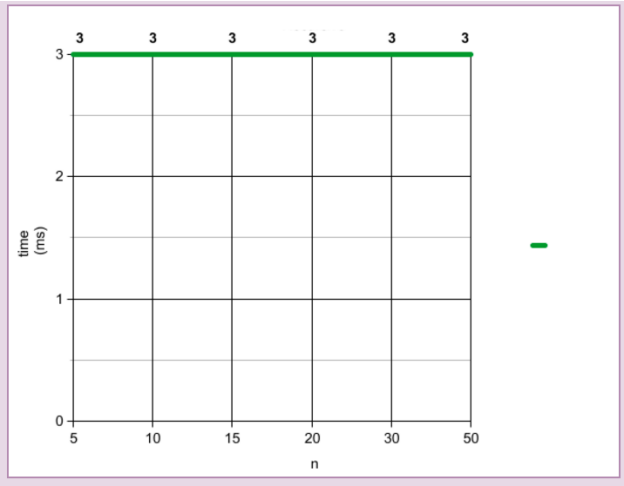
n = 50:

```
real	0m0.007s
user	0m0.002s
sys	0m0.003s
```

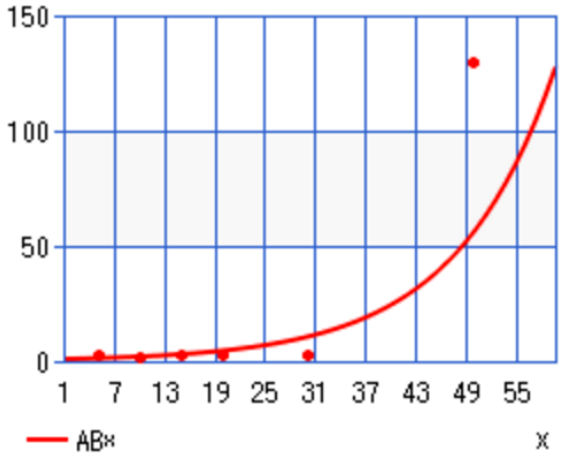
c) Plot the running time data you collected on graphs with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software.

I used the sys times for both. (graphs on next page)

Recursive data graph:



Iterative data graph:



d) What type of function (curve) best fits each data set? Again you can use Excel, Matlab, any software or a graphing calculator to calculate the regression curve. Give the equation of the function that best "fits" the data and draw that curve on the data plot. Why is there a difference in running times?

(Both graphs are pictured above)
The recursive data best fits the function y=ABx. However, I think in theory, the worst-case for the recursive function should look something like O(work^n), based on the fact that each increase in n multiplies the amount of work that must be done by the amount that must be done on the numbers before it.

The iterative data best fits the function y = 0x + 3, simply because the results were always 3 ms in this trial. However, I think that it should be something like O(work * n), (or in line form, y = bX) considering that more work must be done on higher numbers.

The reason the iterative function is much quicker is because the recursive implementation includes many redundant calls by recomputing the same values over and over again.