Name: Alex Miranda
Date: September 28, 2016
CS 325 Homework Assignment #1

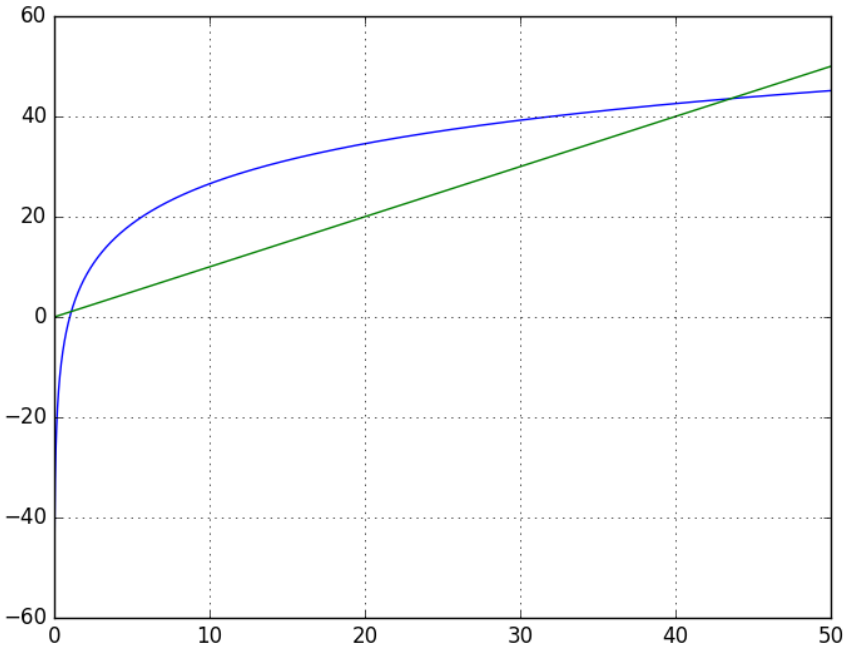
1) (CLRS) 1.2-2. Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n * log_2(n)$ steps. For which values of n does insertion sort beat merge sort?

In order to find the range of n where insertion sort beats merge sort I need to find when the functions are equal to each other, so I set them equal to each other and simplify for n.

$8n^2 = 64n * log_2(n)$
$n^2 = 8n * log_2(n)$ *(Divide both sides by 8)*
$n = 8 * log_2(n)$ *(Divide both sides by n)*

Now plotting the two expressions to see where they intersect to determine the range of n values:

(Substituting n with x)
$y_1 = x$
$y_2 = 8 * log_2(x)$



Based on the graph the values for which insertion sort outperforms merge sort would be when n is around 0 to 5 and when n is around 40 to 45. The values of the two graphs at those values of n are in the table below:

See end of homework for this table (There was a markup issue when converting to pdf)

As seen in the table the intersections occur when 1 < n < 2 and when 43 < n < 44 so for the sake of keeping n as a whole number the range will be defined as follows:

$2 < n < 43$

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.

See end of homework for this table (There was a markup issue when converting to pdf)

3) (CLRS) 2.3-3 on page 39. Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

Base step:

- $n = 2$ therefore
- $T(2) = 2$ so that
- $2 * log_2(2) = 2$

Hypothesis step:

- Assume $T(n) = n * log_2(n)$ is true if $n = 2^k$ for some integer $k > 0$

Induction step:

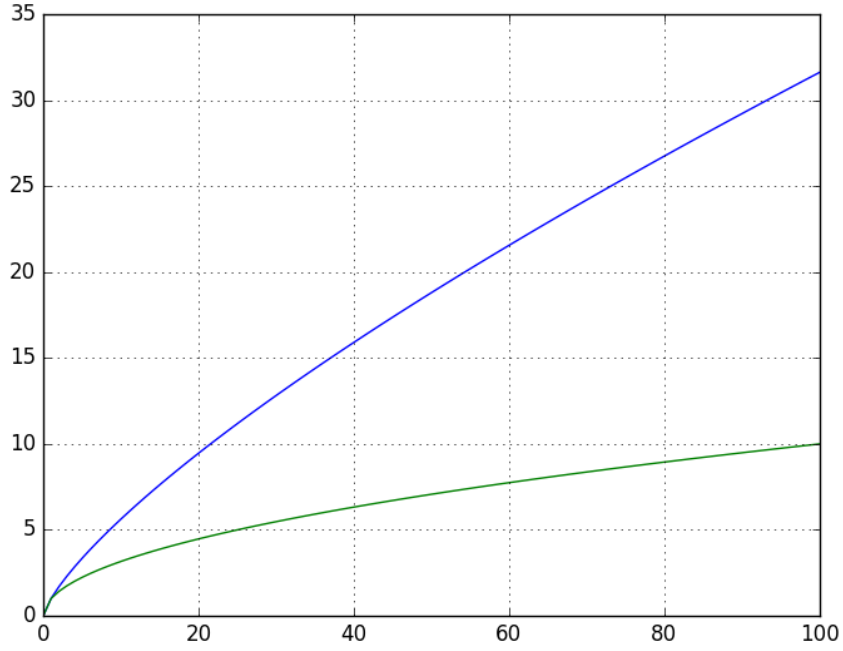
- If $n = 2^{k+1}$ then
- $T(2^{k+1}) = 2T(\frac{2^{k+1}}{2}) + 2^{k+1}$
- $2T(2^k) + 2^{k+1}$
- $2(2^k * log_2(2^k)) + 2^{k+1}$
- $2^{k+1}(log_2(2^k) + 1)$
- $2^{k+1}(log_2(2^k) + log_2(2))$
- $2^{k+1} * log_2(2^{k+1})$

This proves through induction that $T(n) = n * log_2(n)$ when n is an exact power of 2

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.

Will show graphs of each function pair, $f(n)$ are blue and $g(n)$ are green

a) $f(n) = n^{0.75}$ $g(n) = n^{0.5}$



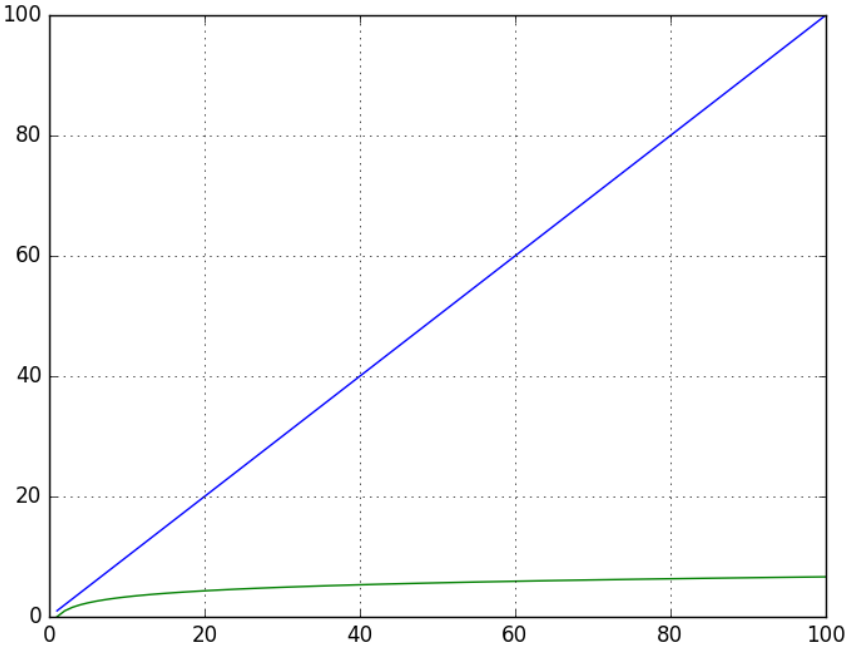
$f(n) = \Omega(g(n))$ because:

- $\frac{f(n)}{g(n)} = \frac{n^{0.75}}{n^{0.5}}$
- $\frac{n^{0.75}}{n^{0.5}} = n^{0.75-0.5} = n^{0.25}$

Taking the limit of $\frac{f(n)}{g(n)}$ as n approaches infinity the expression also goes to infinity showing that:

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

b) $f(n) = n \; g(n) = log_2(n)$



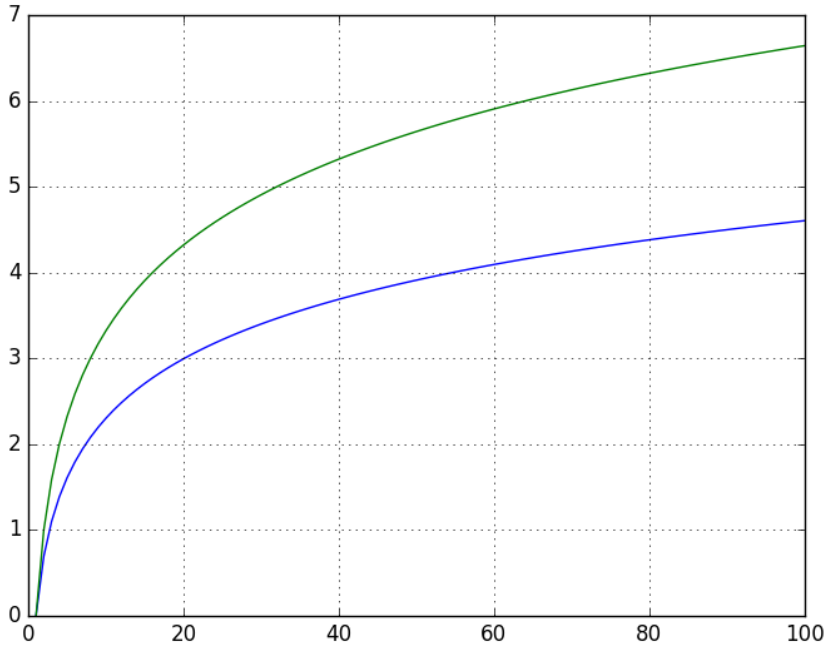
$f(n) = \Omega(g(n))$ because:

- $\frac{f(n)}{g(n)} = \frac{n}{log_2(n)}$
- $\frac{n}{log_2(n)} = \frac{1}{\frac{1}{n*ln(2)}}$ (L'hopital's Rule)
- $\frac{1}{\frac{1}{n*ln(2)}} = n * ln(2)$

As n approaches infinity so will $n * ln(2)$ proving that:

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

c) $f(n) = log(n)\ g(n) = log_2(n)$



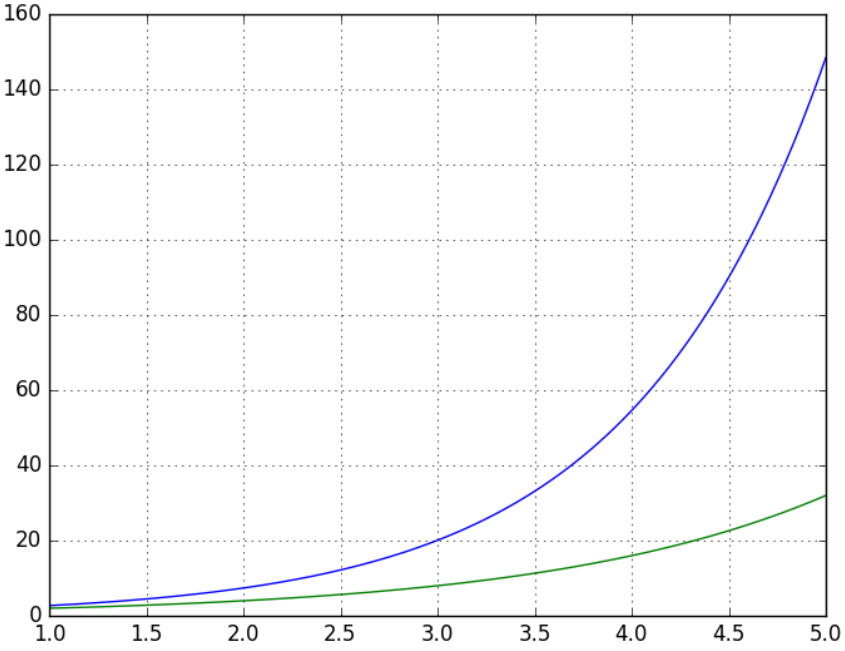
$f(n) = \Theta(g(n))$ because:

- $\frac{f(n)}{g(n)} = \frac{log(n)}{log_2(n)}$
- $\frac{log(n)}{log_2(n)} = \frac{\frac{1}{n*ln(10)}}{\frac{1}{n*ln(2)}}$ (L'hopitals Rule)
- $\frac{\frac{1}{n*ln(10)}}{\frac{1}{n*ln(2)}} = \frac{n*ln(2)}{n*ln(10)}$
- $\frac{n*ln(2)}{n*ln(10)} = \frac{ln(2)}{ln(10)}$

Taking the limit of the quotient of the two functions as n approaches infinity shows that the limit approaches a constant value other than zero so it shows that $g(n)$ can bound $f(n)$ above and below for certain constant factors therefore:

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

d) $f(n) = e^n\ \ g(n) = 2^n$



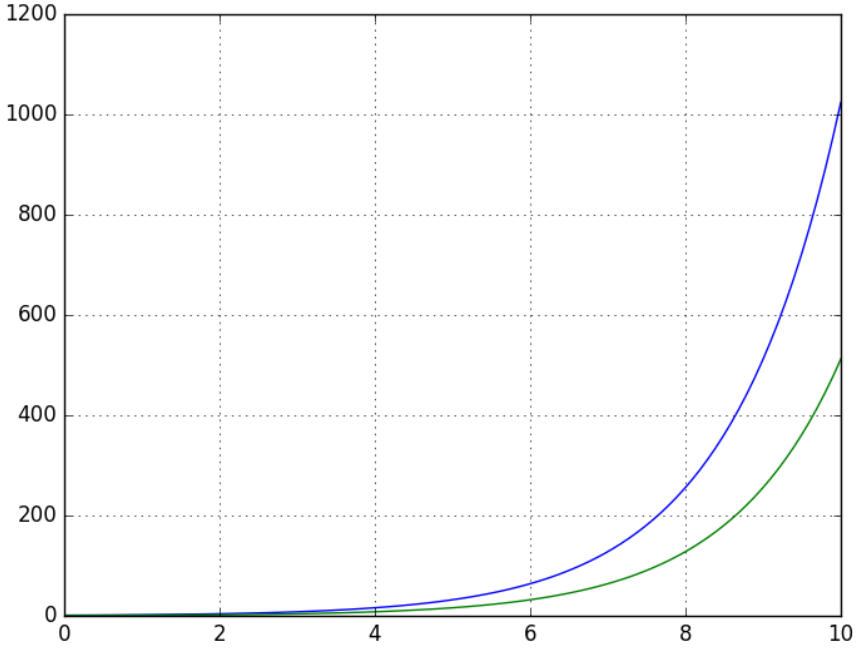
$f(n) = \Omega(g(n))$ because:

- $\frac{f(n)}{g(n)} = \frac{e^n}{2^n}$
- $\frac{e^n}{2^n} = (\frac{e}{2})^n$ (Properties of exponents)
- The base of the exponential function is $(\frac{e}{2}) > 1$ because $e \approx 2.7$ therefore as n goes to infinity so will $(\frac{e}{2})^n$ because the base of the function is greater than one. (Properties of expontential functions)

So we showed that as n approaches infinity $\frac{f(n)}{g(n)}$ also approaches infinity showing that:

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

e) $f(n) = 2^n \ g(n) = 2^{n-1}$



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

The reason this is true is because:

- $2^{n-1} = \frac{1}{2} * 2^n$

This shows that $g(n)$ is $cf(n)$ where $c = \frac{1}{2}$

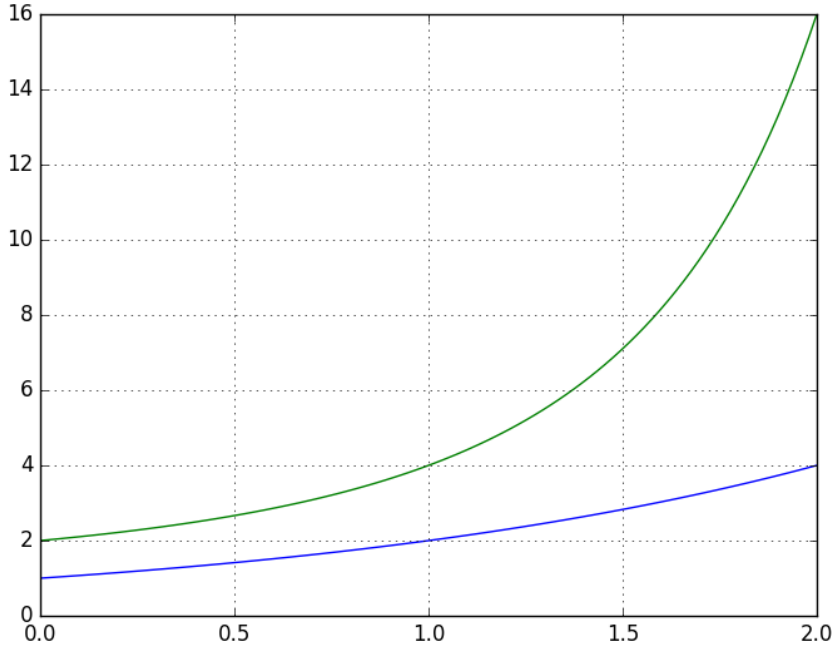
So $g(n)$ can bound $f(n)$ both above and below for certain constant values. An example would be:

- $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ where $c_1 = 1$ and $c_2 = 4$
- $2^{n-1} \leq 2^n \leq 2^{n+1}$

Therefore:

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

f) $f(n) = 2^n \;\; g(n) = 2^{2^n}$



$f(n) = O(g(n))$

The reason for this is because:

- $\frac{2^n}{2^{2^n}} = 2^{n-2^n}$ (Property of exponents)

So as n approaches infinity will n or 2^n dominate?

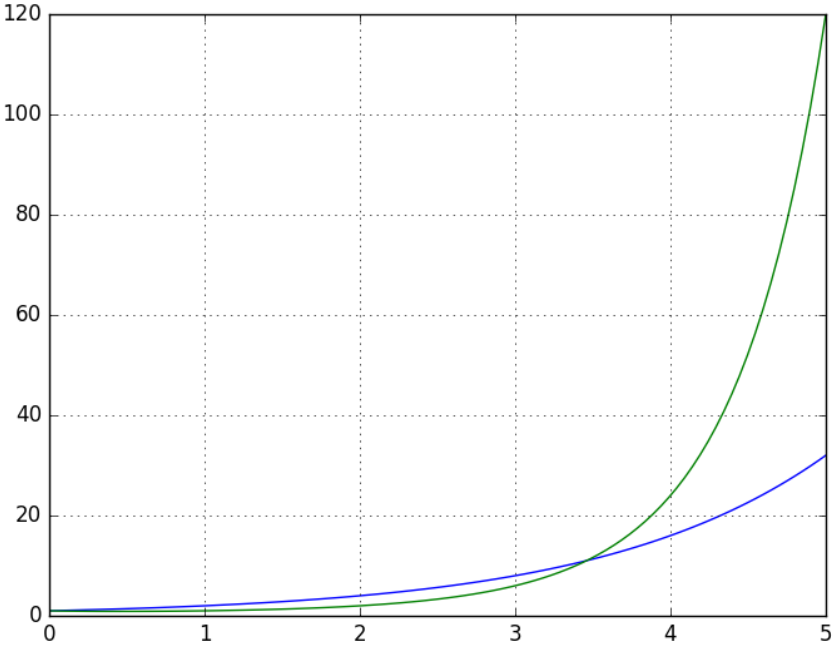
- $\frac{n}{2^n} = \frac{1}{ln(2)2^n}$ (L'hopital's rule)

So as n approaches infinity the quotient of $\frac{n}{2^n}$ will approach 0 showing that 2^n will dominate in the exponent in 2^{n-2^n} therefore the value of the exponent will become more and more negative as n grows larger towards infinity thus showing that as n approaches infinity:

- 2^{n-2^n} will go to zero showing that $g(n)$ will be an upper bound of $f(n)$ thus proving:

$f(n) = O(g(n))$

g) $f(n) = 2^n \ g(n) = n!$



$f(n) = O(g(n))$

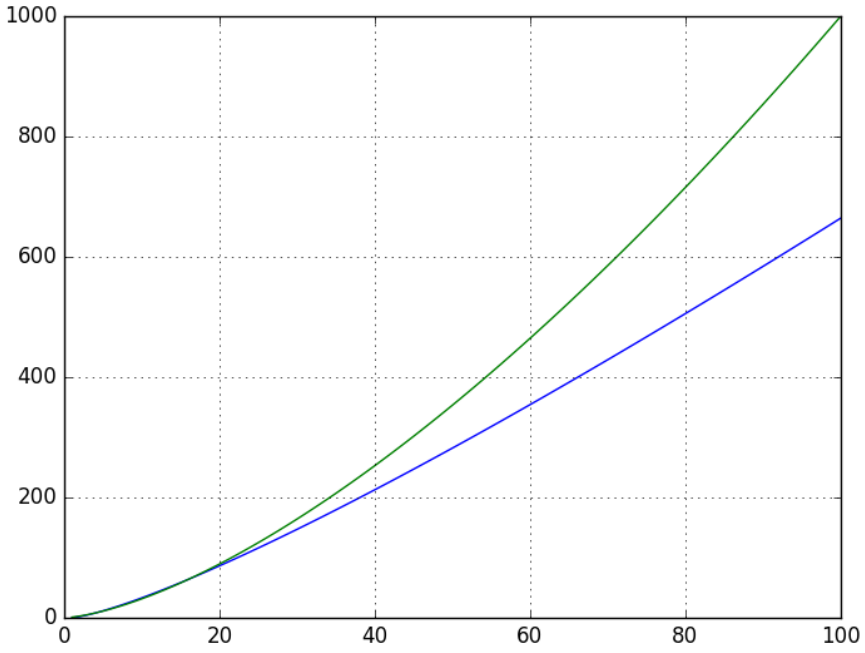
The reason for this is because:

- $\frac{f(n)}{g(n)} = \frac{2^n}{n!}$
- $0 < \frac{2^n}{n!} = \frac{2}{1} * \frac{2}{2} * \frac{2}{3} \ldots \frac{2}{n}$
- $\frac{2}{1} * \frac{2}{2} * \frac{2}{3} \ldots \frac{2}{n} \leq \frac{2}{1} * \frac{2}{2} * \frac{2}{3} \ldots \frac{2}{3}$
- $\frac{2}{1} * \frac{2}{2} * \frac{2}{3} \ldots \frac{2}{3} = \frac{2}{1} * \frac{2}{2} * (\frac{2}{3})^{n-2}$
- $\frac{2}{1} * \frac{2}{2} * (\frac{2}{3})^{n-2} = 2 * (\frac{2}{3})^{n-2}$

As n approaches infinity $2 * (\frac{2}{3})^{n-2}$ approaches 0 therefore by the squeeze theorem it is shown that $\frac{2^n}{n!}$ also approaches 0 as n goes to infinity thus showing that:

$f(n) = O(g(n))$

h) $f(n) = n * log_2(n)\ g(n) = n\sqrt{n}$



$f(n) = O(g(n))$

The reason for this is because:

- $\frac{f(n)}{g(n)} = \frac{n*log_2(n)}{n\sqrt{n}}$
- $\frac{n*log_2(n)}{n\sqrt{n}} = \frac{log_2(n)}{\sqrt{n}}$ (Divide numerator and denominator by n)
- $\frac{log_2(n)}{\sqrt{n}} = \frac{\frac{1}{n*ln(2)}}{\frac{1}{2\sqrt{n}}}$ (L'hopital's Rule)
- $\frac{\frac{1}{n*ln(2)}}{\frac{1}{2\sqrt{n}}} = \frac{2\sqrt{n}}{n*ln(2)}$ (L'hopital's Rule)
- $\frac{2\sqrt{n}}{n*ln(2)} = \frac{2}{2*ln(2)*\sqrt{n}}$ (L'hopital's Rule)
- $\frac{2}{2*ln(2)*\sqrt{n}} = \frac{1}{ln(2)\sqrt{n}}$ (Eliminanting common terms)

As n approaches infinity $\frac{f(n)}{g(n)}$ approaches 0 therefore:

$f(n) = O(g(n))$

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].

My original algorithm was doing 2n comparisons in the worst case. The implementation that does at most 1.5n comparisons is below. Because array A has an odd length both the min and max values are initialized as the first term in the array and set the starting index to be 1. Then the iteration through the array compares adjacent values in the array so the first round of iteration does comparison between 3 and 5 which is true for the arr[i] < arr[i + 1] case. Going into that if it then checks if 3 is less than the min_val which is true because min_val is set to 9 at the time. So the min_val is then set to 3. 5 is then compared to max_val which is also 9 at the time but

that fails so max_val remains 9 for that iteration. The iterations continue until i reaches the array's length - 1 and then returns an object identifying the min and max values of the passed in array. The total number of comparisons for this algorithm is:

- $\frac{n}{2}$ For the first comparison to find the local min/max of the pair
- $\frac{n}{2}$ For the comparison to find the global minimum in the innermost if's in the for loop
- $\frac{n}{2}$ For the comparison to find the global maximum in the innermost if's in the for loop

Leading to a total of $3 * \frac{n}{2}$ comparisons which is $1.5n$ comparisons.

```
def find_min_max(arr):
    start_index = None

    if len(arr) % 2 == 0:
      if arr[0] > arr[1]:
        max_val = arr[0]
        min_val = arr[1]
      else:
        max_val = arr[1]
        min_val = arr[0]
      start_index = 2
    else:
      max_val = arr[0]
      min_val = arr[0]
      start_index = 1

    for i in range(start_index, len(arr) - 1):

        if arr[i] < arr[i + 1]:
            if arr[i] < min_val:
                min_val = arr[i]
            if arr[i + 1] > max_val:
                max_val = arr[i + 1]
        else:
            if arr[i + 1] < min_val:
                min_val = arr[i + 1]
            if arr[i] > max_val:
                max_val = arr[i]

    return {'max': max_val, 'min': min_val}
```

6) Let f1 and f2 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 $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

By definition there exists two integers n_1, n_2 and two constants c_1, c_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq n_1$ and $f_2(n) \leq c_2 g_2(n)$ for $n \geq n_2$.

- $f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$ (Substitution from the definition above)

Assume that the expression $max(a, b)$ returns the maximum between the pair from now on therefore:

Suppose there are $c_0 = 2max(c_1, c_2)$ and $n_0 = max(n_1, n_2)$ therefore:

- $c_1 g_1(n) + c_2 g_2(n) = c_0(g_1(n) + g_2(n))/2$ where $n_0 \leq n$
- $c_0(g_1(n) + g_2(n))/2 = c_0(g_1(n) + g_2(n))$ Where the $\frac{1}{2}$ could be incorporated into the c_0

Therefore there exists some c_0 and n_0 such that:

- $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ when $n \geq n_0$

b) If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $\frac{f_1(n)}{f_2(n)} = O(\frac{g_1(n)}{g_2(n)})$

Suppose that $f_1(n) = n, f_2(n) = n^3$, $g_1(n) = n^2$, and $g_2(n) = n^5$ (both c_1 and c_2 are 1) and $f_1(n) = O(g_1(n)), n_1 \leq n$ and $f_2(n) = O(g_2(n)), n_2 \leq n$. Therefore if the conjecture is true:

- $\frac{f_1(n)}{f_2(n)} = \frac{n}{n^3} \leq \frac{n^2}{n^5} max(n_1, n_2) \leq n$
- $\frac{f_1(n)}{f_2(n)} = \frac{n}{n^3} = \frac{1}{n^2}$ (Simplifying terms)
- $\frac{g_1(n)}{g_2(n)} = \frac{n^2}{n^5} = \frac{1}{n^3}$ (Simplifying terms)

So we are left with:

- $\frac{1}{n^2} \leq \frac{1}{n^3}$ for $max(n_1, n_2) \leq n$

However in order for this to be true the quotiant of the g functions divided by the quotiant of the f functions must approach infinity as n approaches infinity

- $\frac{\frac{1}{n^3}}{\frac{1}{n^2}} = \frac{n^2}{n^3} = \frac{1}{n}$ (Algebraic manipulation)

When simplified we can see the resulting expression will approach 0 as n goes to infinity therefore:

$\frac{f_1(n)}{f_2(n)} = \Omega(\frac{g_1(n)}{g_2(n)})$ for this example that fulfills assumptions of the if-clause

Thus the conjecture is contradicted.

7) **Fibonacci Numbers:**

The Fibonacci sequence is given by: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... By definition the Fibonacci sequence starts at 0 and 1 and each subsequent number is the sum of the previous two. In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0$ and $F_1 = 1$

An algorithm for calculating the n^{th} Fibonacci number can be implemented either recursively or iteratively.

Example Recursive:

```
fib (n) {
    if (n = 0) {
        return 0;
```

```
    } else if (n = 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Example Iterative:

```
fib (n) {
    fib = 0;
    a = 1;
    t = 0;
    for (k = 1 to n) {
        t = fib + a;
        a = fib;
        fib = t;
    }
    return fib;
}
```

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.

Both implementations are in Python 3

Recursive implementation:

```
def fib_recur(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n - 1) + fib_recur(n - 2)
```

Iterative implementation:

```
def fib_iter(n):
    cur_term = 0
    next_term = 1
    total = 0

    for k in range(0, n):
        total = cur_term + next_term
        next_term = cur_term
        cur_term = total

    return total
```

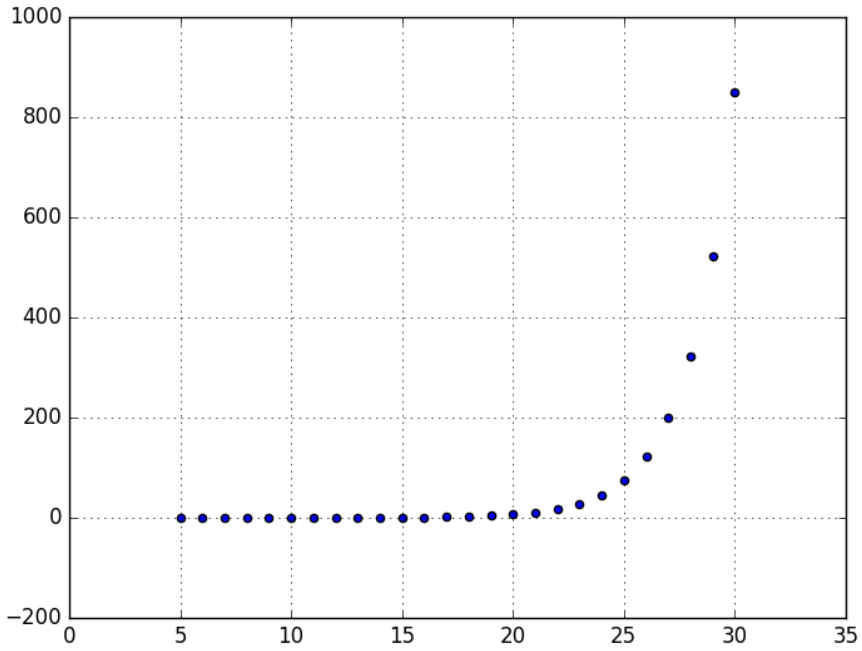
b) Use the system clock to record the running times of each algorithm for n = 5, 10, 15, 20, 30, 50, 100, 1000, 2000, 5000, 10,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. If you program in C your algorithm will run faster 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.

The entries in the table are in milliseconds.

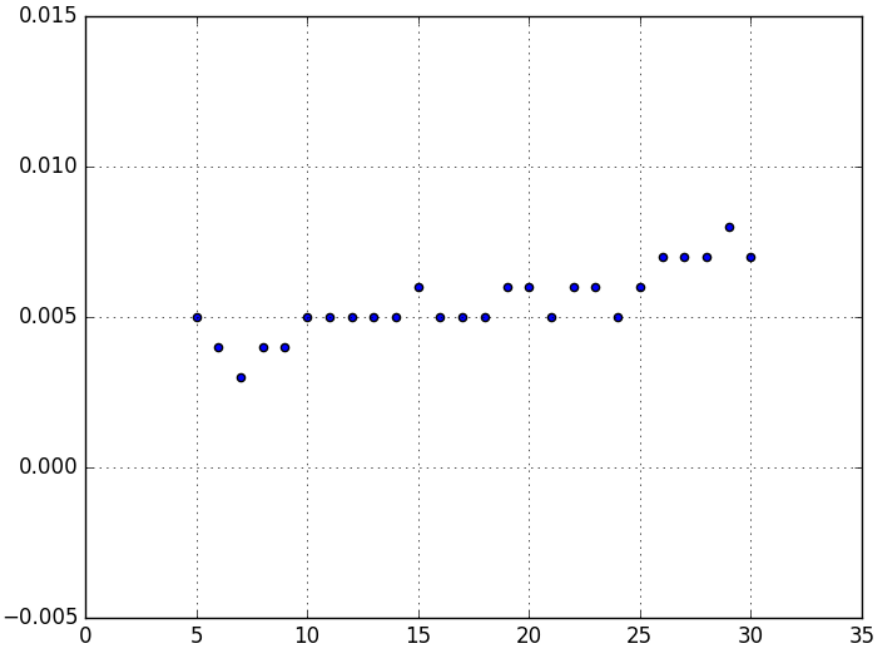
See end of homework for the tables (There was a markup issue when converting to pdf)

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.

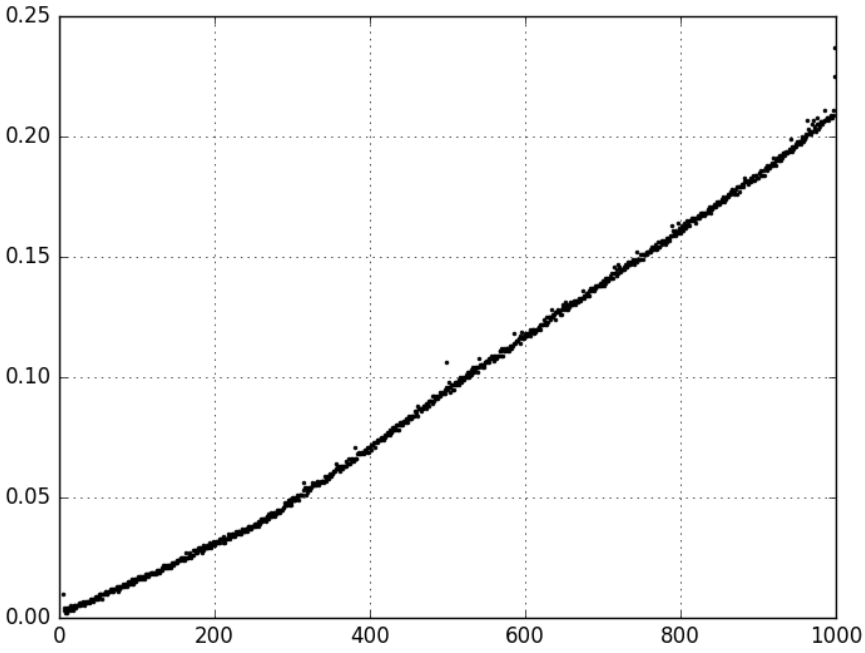
Recursive runtime plot:



Iterative runtime plot:

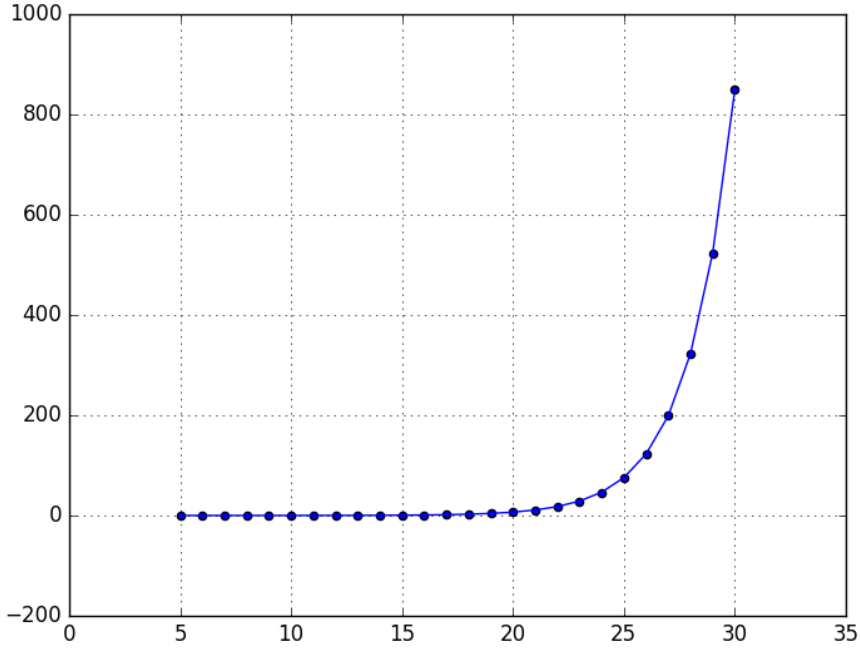


Because the trend isn't that apparent for the iterative case for such a small range of n I created another plot that plots the runtimes for the iterative case from 5 to 1000



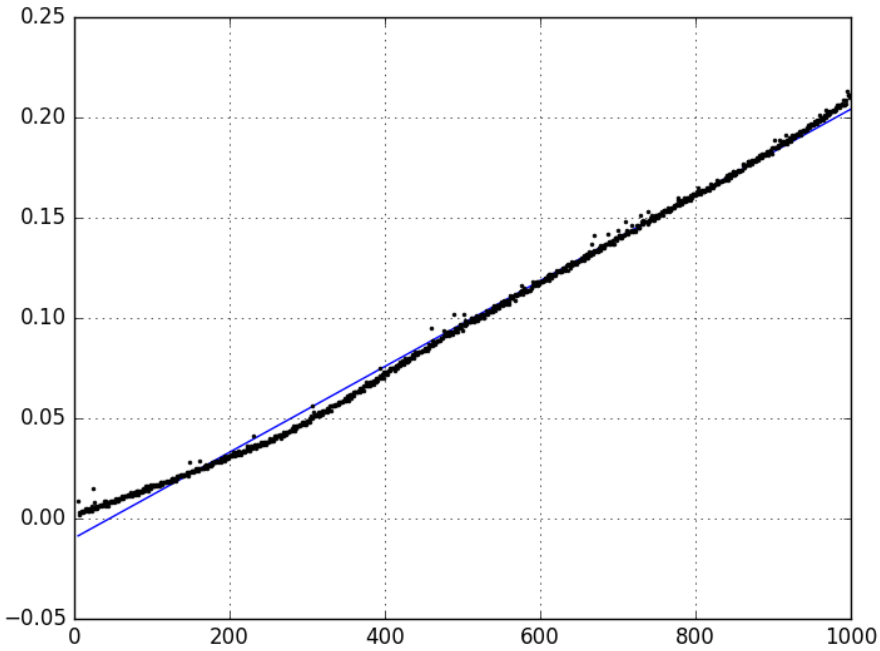
d) What type of function (curve) best fits each data set? Again you can use Excel, Matlab, any software or a graphing calculator to calculate the regression curve. Give the equation of the function that best “fits” the data and draw that curve on the data plot. Why is there a difference in running times?

For the recursive case an exponential curve best fits the runtime data as shown in the plot below:



The literal equation is: $y = (4.222 * 10^{-4}) * e^{-0.483x} + 8.37 * 10^{-2}$

For the iterative case a linear curve best fits the runtime data as shown in the plot below:



The equation for the best fit line is:

$$y = 0.00021384x - 0.00970961$$

The reason for the difference in the running times is because the recursive implementation requires multiple calls to the fibonacci function to obtain the previous values in the sequence when calculating the sum, this call stack causes the runtime to become exponential. The iterative implementation has a linear runtime because each term

In the sequence is only computed once when computing the sum of all of the terms. If the recursive call cached the previous values rather than recomputing them for each function call it's runtime would also be linear

Item	1 second	1 miniute	1 hour	1 day	1 month	1 year	1 century

Table Appendix:

Question 1:

$f(n)$	1	2	3	4	5
x	1	2	3	4	5
$8log_2(n)$	0	8	12.67970	16	18.57542

$f(n)$	40	41	42	43	44
x	40	41	42	43	44
$8log_2(n)$	42.5754	42.8604	43.1385	43.4101	43.6754

Question 2:

Item	1 second	1 miniute	1 hour	1 day	1 month	1 year	1 century
lgn	2^{10^6}	2^{6*10^7}	2^{36*10^8}	2^{864*10^8}	$2^{25920*10^8}$	$2^{315360*10^8}$	$2^{31556736*10^8}$
$n^{1/2}$	10^{12}	$36 * 10^{14}$	$1296 * 10^{16}$	$746496 * 10^{16}$	$6718464 * 10^{18}$	$994519296 * 10^{18}$	$995827586973696 * 10^{16}$
n	10^6	$6 * 10^7$	$36 * 10^8$	$864 * 10^8$	$2592 * 10^9$	$31536 * 10^9$	$31556736 * 10^8$
$nlgn$	62746	2801417	133378058	2755147513	71870856404	797633893349	68654697441062
n^2	1000	7745	60000	293938	1609968	5615692	56175382
n^3	100	391	1532	4420	13736	31593	146677
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Question 7:

Function	5	6	7	8	9	10	11	12	13	14	15	16	17
Recursive	0.015	0.015	0.022	0.048	0.056	0.083	0.132	0.209	0.341	0.533	0.818	1.189	1.92
Iterative	0.005	0.004	0.003	0.004	0.004	0.005	0.005	0.005	0.005	0.005	0.006	0.005	0.005

Function	18	19	20	21	22	23	24	25	26	27	28	29	30
Recursive	3.087	4.407	6.66	10.73	17.35	28.34	45.86	75.97	122.9	201.0	322.8	521.5	850.0
Iterative	0.005	0.006	0.006	0.005	0.006	0.006	0.005	0.006	0.007	0.007	0.007	0.008	0.007

In [ ]: