Vlad Predovic

CS325

Due 4/03/2016

Homework 1

1)

Insertion sort: ~~8n \*~~ n steps

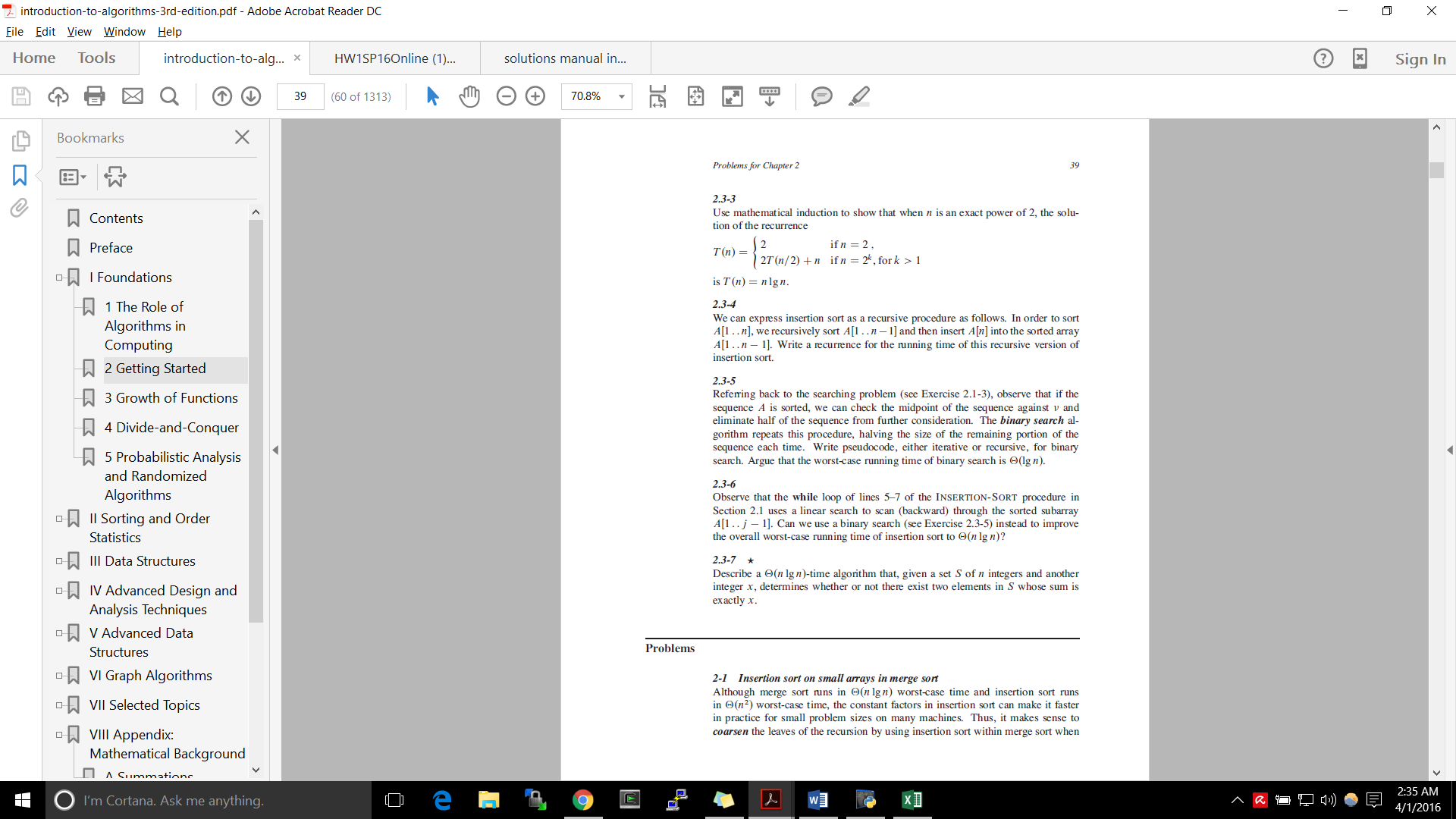
Merge sort: ~~64n~~ 8\* log2(n) steps

Python Solution:

|  |
| --- |
| >>> num1=2  >>> num2=3  >>> while num2 > num1:  ... num1 += .1  ... num2 = math.log(num1, 2)\*8  >>> num2  43.57004983911661  >>> num1  43.60000000000035 |

Therefore n can be a maximum of 43(if Integer). Merge sort becomes more efficient at higher numbers due to having a logarithmic complexity.

2)

3)

For all n,k in https://www.mathsisfun.com/images/symbols/set-natural-lg.gif that are an exact power of 2, T(n) = nlgn.

**Base Case**: When n = 2 🡪 2lg(2) = 2\*1 = 2

Assume T(n)= nlogn when n=2k then for a given number k:

If n = 2k+1 then

T(2k+1) = 2T(2k+1/2)+2k+1 = 2T(2k\*2 /2) + 2k+1 = 2T(2k) + 2k+1

T(2k+1) = 2\*2k(log(2k)) + 2k+1

= 2k+1(log2k + 1)

= 2k+1(log2k+1)

4)

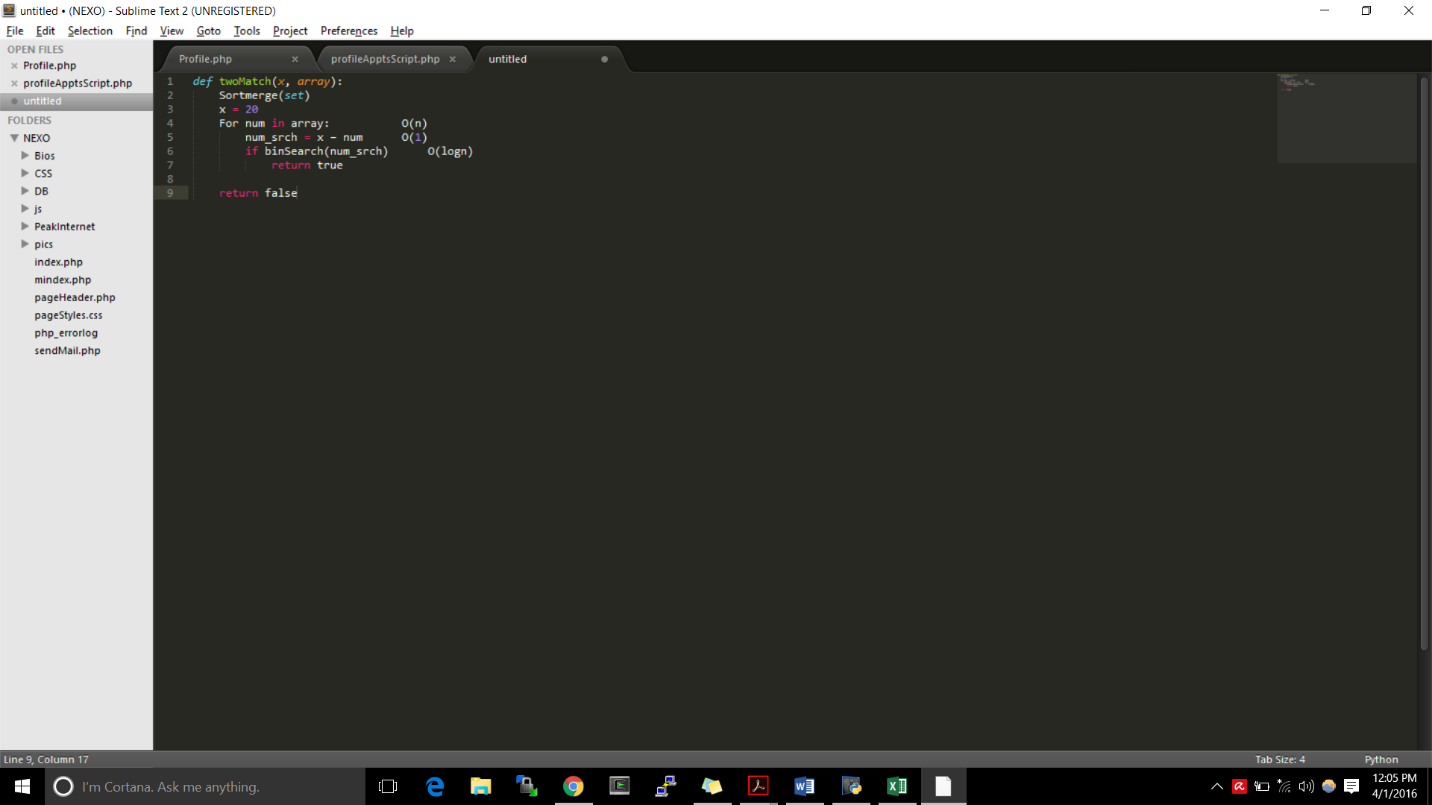
|  |
| --- |
| a. f(n) = n0.75; g(n) = n0.5 🡪 f(n) = Ω(g(n))  since .75 > .5 and it is known that for example n3 will always we greater than n3-x for all positive numbers n and x, we can assume that f(n) will at the very least grow as fast as g(n) when n is greater than 1. |
| b. f(n) = n; g(n) = log2n 🡪 f(n) = Ω(g(n))  Linear will always be equal to or slower than logarithmic after a set constant c=1 |
| c. f(n) = log(n); g(n) = lg(n) 🡪 f(n) = Θ(g(n))  When speaking asymptotically, the base of logarithms is irrelevant. This is because of the identity loga b logb n = logan |
| d. f(n) = en; g(n) = 2n  🡪 f(n) = Ω (g(n))  The value of e is approximately 2.7. This will always be greater than 2 therefore at best f(n) will increment at a speed equal to g(n) |
| e. f(n) = 2 n; g(n) = 2n-1 🡪 f(n) = Θg(n)  2n-1 = 2n/2 O(2n/2) = 2n as it only differs by a constant/ |
| f. f(n) = 2n; g(n) =22 ^ n 🡪 f(n) = O (g(n))  For all n greater than 1 2n >n since 2n> 2\*n > n. Therefore f(n) at worst will equal g(n). |
| g. f(n) = 2 n; g(n) = n! 🡪 f(n) = O (g(n))  = 0, therefore 2n is bounded above by n! |
| h. f(n) = nlgn; g(n) = n√𝑛 🡪 f(n) = O (g(n)) |

5)

This algorithm needs to determine if there are two numbers in a set S that equal another number X with a time complexity of O(nlogn).

I first use a Mergesort which has a worst case time complexity of O(nlogn) satisfying the requirements.

In order to create this algorithm I nested a search algorithm within an outer loop that would iterate through the set with a complexity of O(n). The algorithm inside this loop would then search for the corresponding number that when summed with the first number. For this I used the binary search.

My codeish pseudocode can be found below..

S = {3, 4, 7, 11, 12, 15} after Mergesort.

Step through each number in the array.

For 3 🡪 binary search for 20-3 (17)

Not found, proceed to next number at next index in array.

For 4 🡪 search for 20-4 (16)

Not found

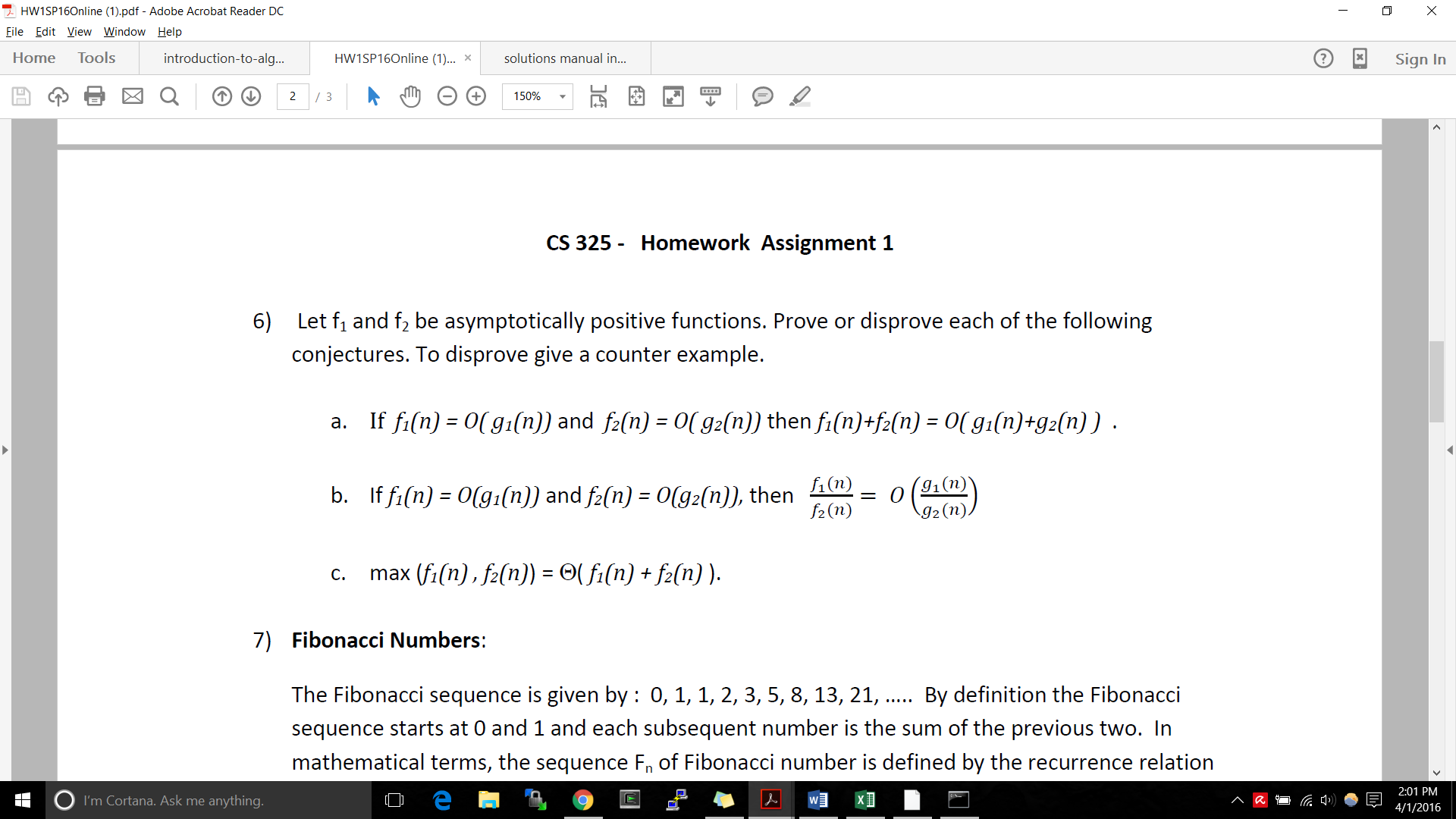
For 7 🡪 search for 20-7 (13)

Not found

Repeat…….

Loop ends without a match.

Function returns false as no two numbers equaling 20 were found.

6)

a. Applying the rule of sums results in:

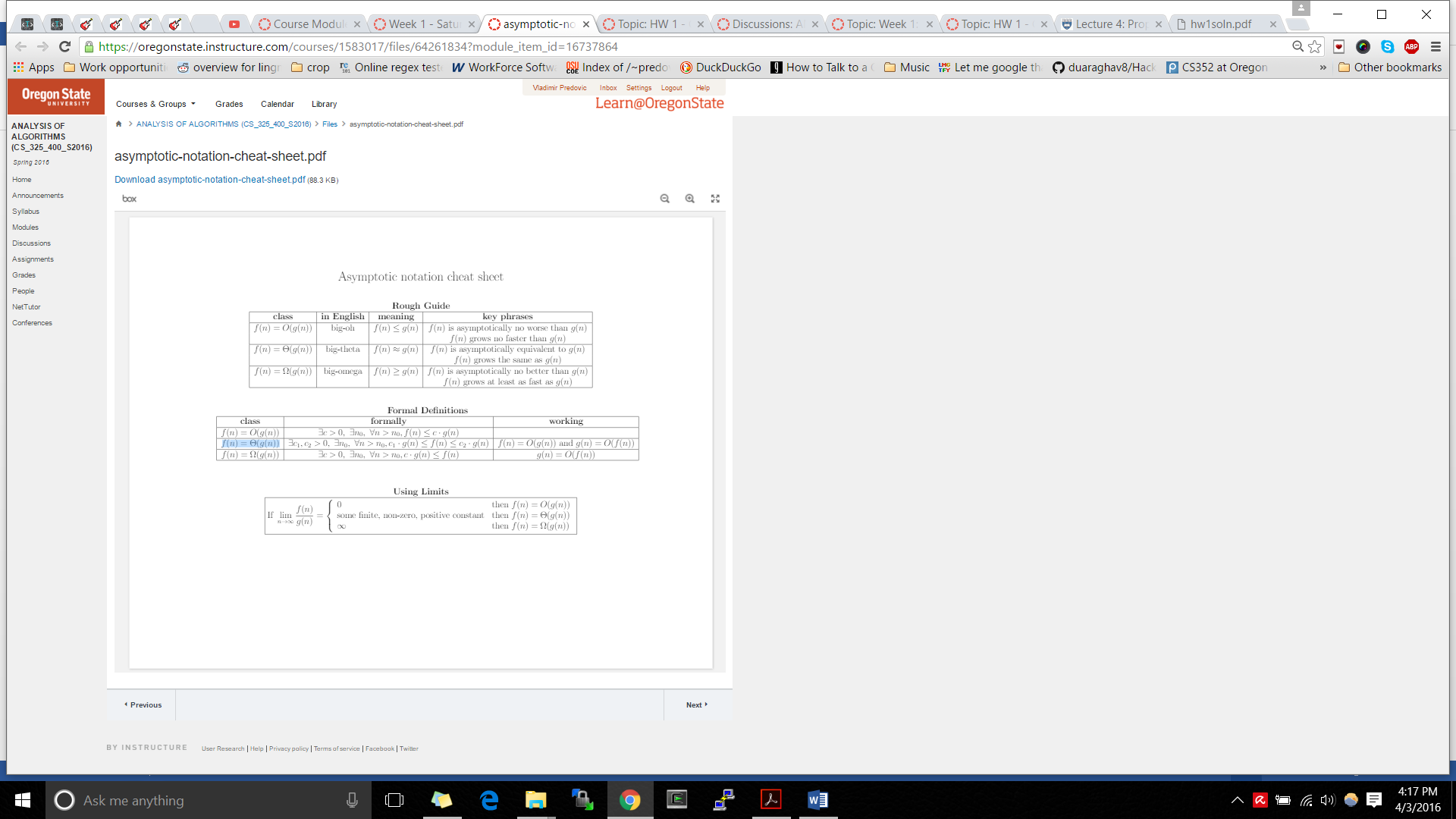
Let f1(n) = O(g1(n)) and f2(n) = O(g2(n))

There exist constants c1,c2>0 where 0 **≤** f1(n) **≤** c1 g1(n)  **and** 0 **≤** f2(n) **≤** c2 g2(n)

There must exist a constant c3 where (f1(n) + f2(n)) **≤** c3**(**g1(n) + g2(n)) for all positive integers greater than a given n0.

f1(n) + f2(n)) **≤** c1g1(n) + c2g2(n))

f1(n) + f2(n)) **≤** c3(g1(n) + g2(n)) satisfies the formal definition below



b. f1(n) = n2 and f2(n) = n. g1(n)=n3and g2(n) = n!

= while =

Therefore ≠ O( ) so this is false

c. To prove this conjecture we first assert the following statements

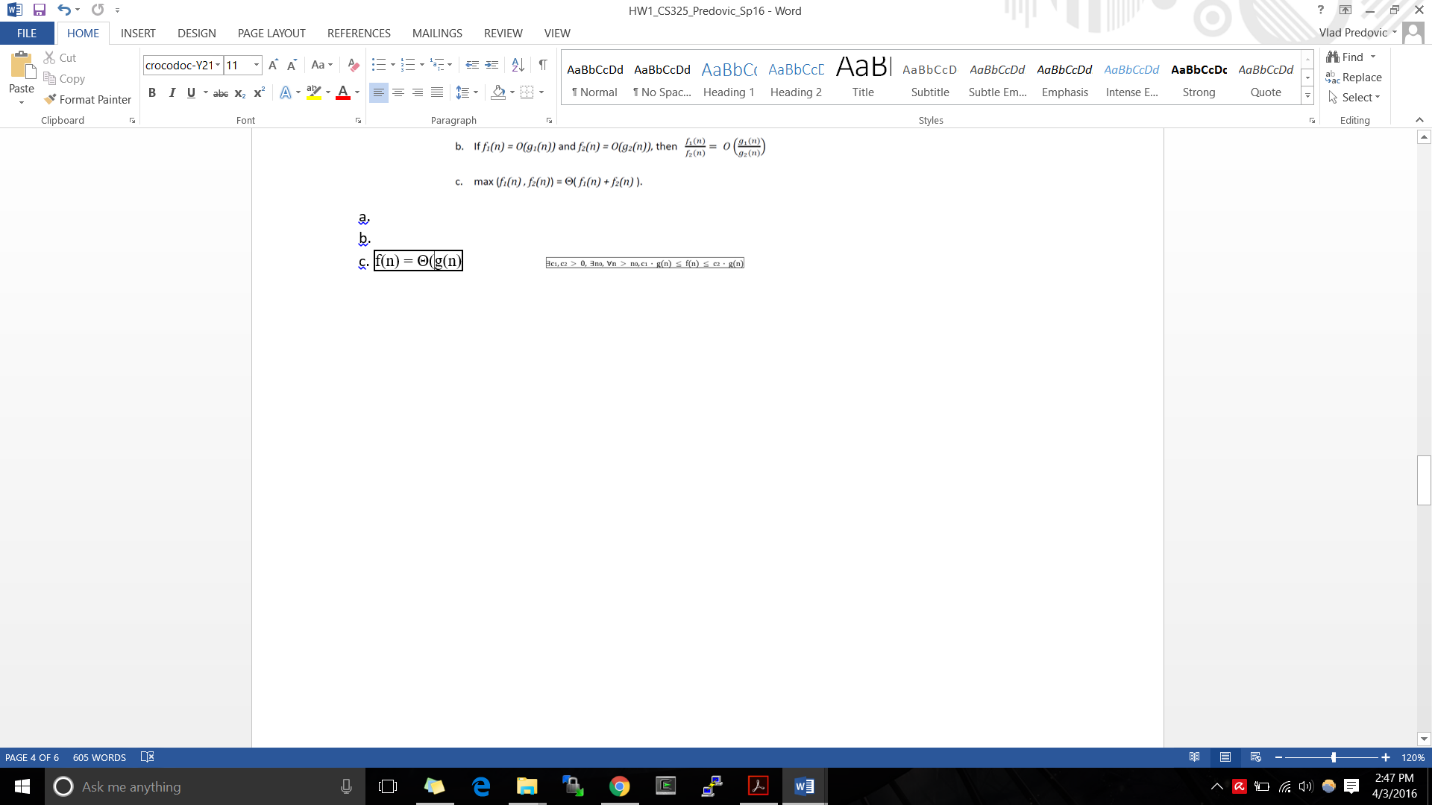
**f(n)≤ max(f(n),g(n))** **g(n)≤ max(f(n),g(n))**

Therefore it can be said that: **(f(n)+g(n))/2 ≤ max(f(n),g(n))**

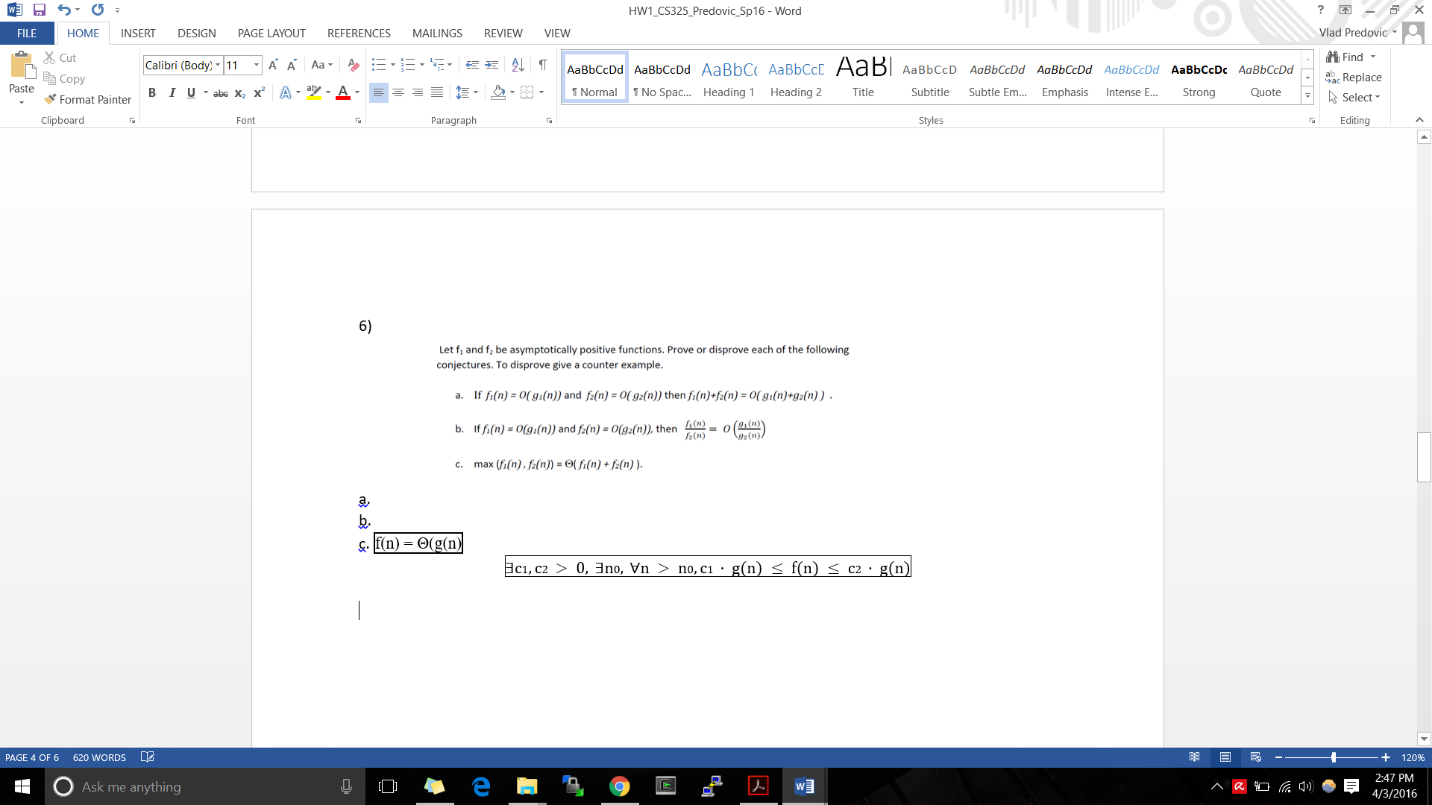
Due to the properties of max it can also be stated that since both f(n) and g(n) are positive functions **max(f(n),g(n)) ≤ 1(f(n) + g(n))**

If substituted **h(n) = f(n) + g(n)**

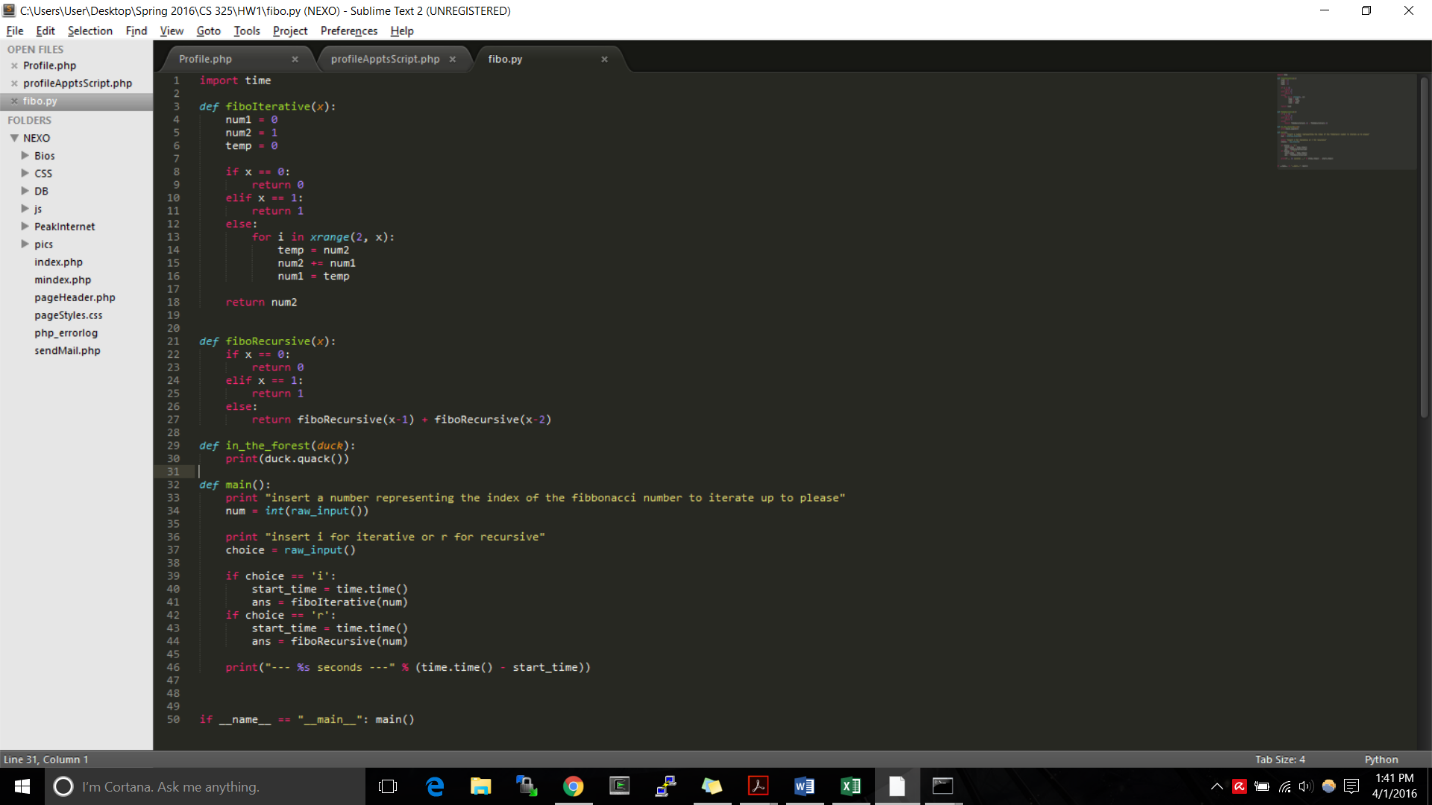
Then it can be concluded that **c1(h(n)) ≤ max(f(n),g(n)) ≤ c2(h(n))**

Where **c1 = ½ and c2 = 1**

This satisfies the formal definition given below for the class



7)



Iterative

Fibb Index

Time

For my data I found very aggressive differences in running times. The recursive function fit best with an exponential graph while the iterative Fibonacci function fit best with a linear graph. The running times are so different due to the algorithmic complexity of each version.   
In the iterative version there is one loop in which three constant time actions are performed. This ultimately results in a linear time complexity as shown by the data.

In the recursive version, every time a call to the function is made it spawns two additional functions until an end case is met. Since in each step we call the function twice the result is a complexity of O(t) = 2n which is incredibly slower than the linear counterpart.

