CS 107
Divesh Otwani
11/2/15

Lab 6: Complexity

The four parts are numbered and highlighted red.

1.

First we create adds(n):

- def adds(n):
  - precondition(is_int(n) and n>0)
  - if n <3:
    - return 0
  - else:
    - return adds(n-1) + adds(n-2) + 1

We conjecture the postcondition is {adds(n) == fib(n) -1}.

We prove the function with three steps (for now).

First, we can say each function call satisfies the precondition of the called function. Note, here we only have two recursive function calls. By the second conditional above the recursive calls, we know n>= 3 (and n is an integer by the precondition). This means { (n-1) >= 2 and (n-1) is an integer} and {(n-2) >= 1 and n-2 is an integer}. In either recursive call, the precondition of adds, of n being an integer greater than 0, is satisfied. Hence, condition one is satisfied.

Condition two shows that each return satisfies the postcondition:

Proof by cases:

1. In the base case, we only have two values of n (since the precondition makes n an int greater than zero and the if conditional makes n <2.) which are 1 and 2. We know fib(2) and fib(1) =1. So, according to the postcondition, we should have adds(2) and adds(1) = 0. Since that is what the function returns, this case has been shown.
2. In the else block we must use the recursive function substitution rule. To use this rule we need to show both recursive calls satisfy the precondition and make progress toward a base case. We already showed the former in step one of the three part correctness proof of adds. We show the latter by observing that subtracting one or two from an integer n >=3 must eventually reach an integer n < 3. If so, by the conditional of the base case, we approach a base case. So, we use the recursive function substitution rule to substitute the return expression as: {adds(n-1) + adds(n-2) + 1} →{fib(n-1) -1 + fib(n-2) -1 + 1}. We can now use basic math to say fib(n-1) + fib(n-1) = fib(n) and -1 -1 +1 = -1. Thus, the returned expression is {fib(n) -1} which satisfies the postcondition.

The third condition is that for recursive functions we need to show that 1) the function approaches a base case (which has already been shown) and that 2) when a progress expression is <= 0, a base case is executed. We make the progress expression (n-2). So, when (n-2) <= 0 or equivalently (n-3) <0, it is obvious that the base case's conditional is true and the base case is executed.

CS 107
Divesh Otwani
11/2/15

Therefore, the conjecture is true and the number of adds of fib(n) for the given algorithm is

Adds(n) = fib(n) -1

or,

$$\text{Adds}(n) = \left(\frac{1}{\sqrt{5}}\right)\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) - 1$$

2.

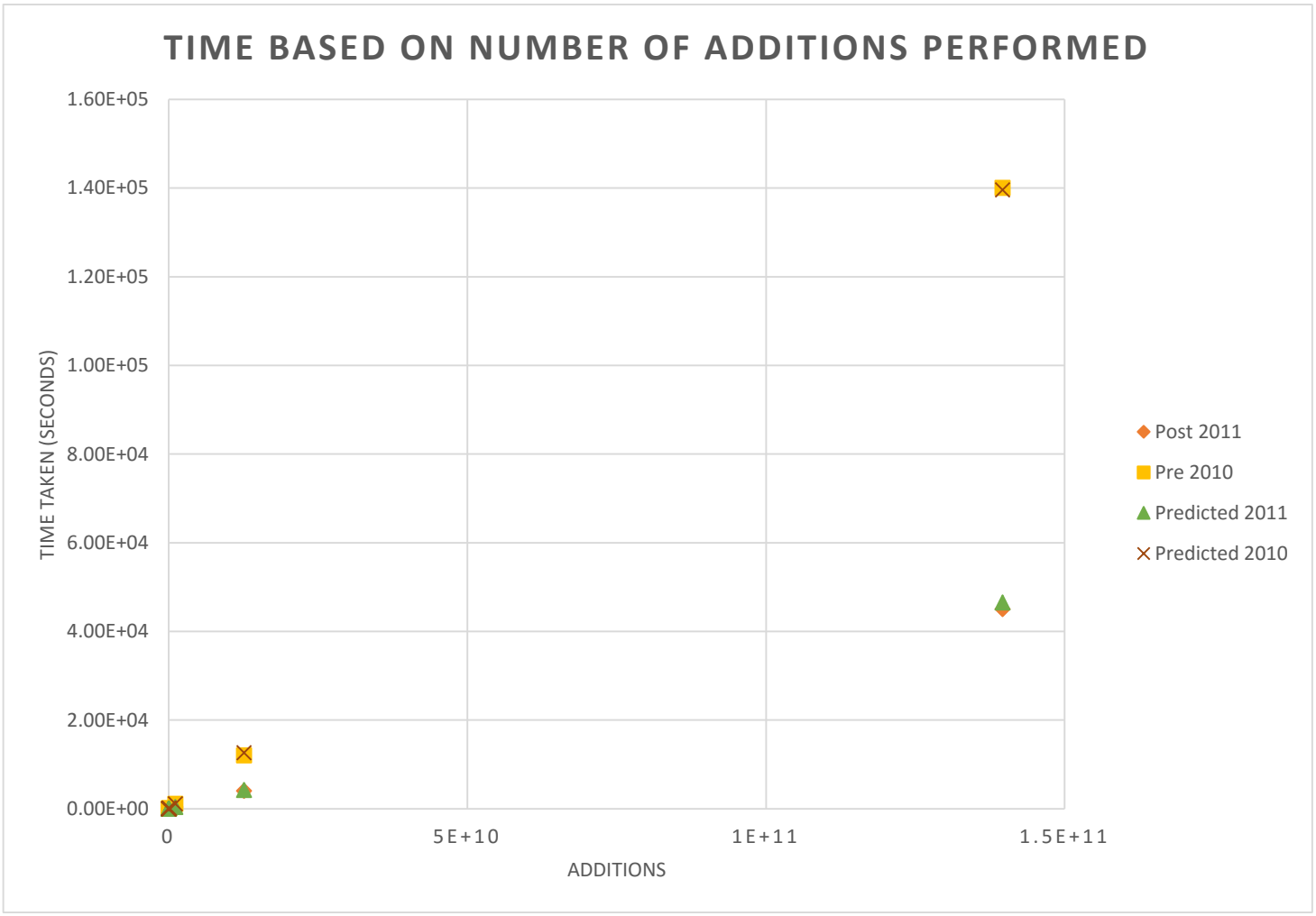
N	Time post 2011 computers (secs)	Time pre 2010 Computers (secs)	Adds(n)	Ratio 1: Adds(n)/2011	Ratio 2: Adds(n)/2010
5	1.60E-06	4.29E-06	4	2.50E+06	9.32E+05
10	1.80E-05	5.24E-05	54	3.00E+06	1.03E+06
15	0.0002	0.00058	609	3.05E+06	1.05E+06
20	0.0022	0.0063	6764	3.07E+06	1.07E+06
25	0.024	0.071	75024	3.13E+06	1.06E+06
30	0.27	0.8	832039	3.08E+06	1.04E+06
35	2.9	8.7	9227464	3.18E+06	1.06E+06
40	33	96	1E+08	3.10E+06	1.07E+06
45	360	1060	1.1E+09	3.15E+06	1.07E+06
50	4000	12000	12586269024L	3.15E+06	1.05E+06
55	45000	140000	139583862444L	3.10E+06	9.97E+05

Since the ‘ratio 1’ column and ‘ratio 2’ column are roughly consistent around one value, our “Adds” corresponds to the time needed by each computer.

These columns also give an addition rate for each type of computer. The addition rate of 2011 computers is *3E06 additions per second* while the addition rate of 2010 computers is *1E06 additions per second*.

CS 107
Divesh Otwani
11/2/15

Graphs:



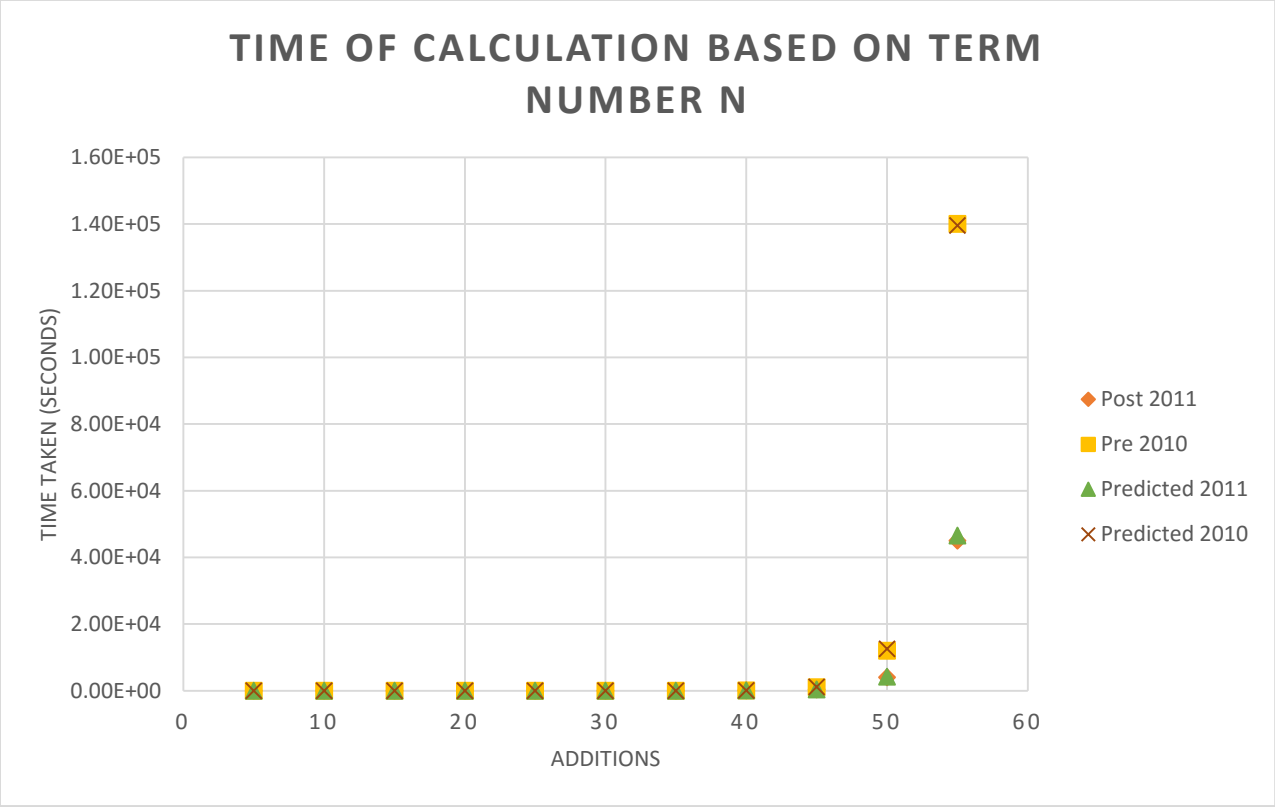
The predicted time values of post 2011 are calculated as f(additions) = additions / (3E06). For pre 2010, f(additions) = additions/(1E06).

CS 107
Divesh Otwani
11/2/15

Since the graph is probably hard to read, here are the values:

Additions	Post 2011	Pre 2010	Predicted 2011	Predicted 2010	Error in 2011 prediction	Error in 2010 prediction
4	1.60E-06	4.29E-06	1.33333E-06	0.000004	1.67E-01	6.76E-02
54	1.80E-05	5.24E-05	0.000018	0.000054	0.00E+00	3.05E-02
609	0.0002	0.00058	0.000203	0.000609	1.50E-02	5.00E-02
6764	0.0022	0.0063	0.002254667	0.006764	2.48E-02	7.37E-02
75024	0.024	0.071	0.025008	0.075024	4.20E-02	5.67E-02
832039	0.27	0.8	0.277346333	0.832039	2.72E-02	4.00E-02
9227464	2.9	8.7	3.075821333	9.227464	6.06E-02	6.06E-02
1.00E+08	33	96	33.33333333	100	1.01E-02	4.17E-02
1.10E+09	360	1060	366.6666667	1100	1.85E-02	3.77E-02
12586269024	4000	12000	4195.423008	12586.26902	4.89E-02	4.89E-02
1.39584E+11	45000	140000	46527.95415	139583.8624	3.40E-02	2.97E-03

(I wasn't sure if you wanted this graph too :)



CS 107
Divesh Otwani
11/2/15

3.

The predictions are calculated in this table:

N	Adds(n)	Time post 2011 = Adds(n) / (3E06) (seconds)	Time pre 2010 =Adds(n) / (1E06) (seconds)
60	1548008755919	516002.9	1548008.8
100	354224848179263111168	118074949393087.7	354224848179263.1

So, for post 2011 computers fib(60) takes about 6 days and fib(100) takes about 3744132 years. With pre 2010 computers fib(60) takes about 18 days and fib(100) takes about 11232396 years. This is incredibly slow.

4.

This has three parts: the chart, the graphs and the upgrade vs algorithm discussion. Each part header is green.

Chart for strange_fib:

n	Time taken on hopper (seconds)	Number of operations /Cost measure	Operations per Second
5	1.34E-06	4	2.99E+06
10	9.02E-06	36	3.99E+06
15	6.16E-05	256	4.15E+06
20	0.000416573	1742	4.18E+06
25	0.002812391	11790	4.19E+06
30	0.019000375	79728	4.20E+06
35	0.1285055	539082	4.20E+06
40	0.871316	3644944	4.18E+06
45	5.886621	24644824	4.19E+06
50	39.796624	166632768	4.19E+06
55	268.970102	1126665694	4.19E+06

The computer seems to do 4.2E06 operations per second- this is the operation rate. Thus, we can predict the time it takes to do x operations with f(x) = x / (4.2E06).

CS 107
Divesh Otwani
11/2/15

Explanation of creation of chart:

I calculated the number of operations (multiplications and minuses) with this loop based function:

- def operations_strange_fib(n):
  - if n<4:
    - return 0
  - else:
    - start = [0,0,2]
    - for i in range(n-4):
      - save = start[0]
      - start[0] = start[1]
      - start[1] = start[2]
      - start[2] = 2 + start[1] + save
    - return start[2]

This works because it is a simple translation of the obvious recursive function for operations(n). This basic recursive function has a base case of {n<4, returns 0} while other cases have {n>=4, return 2+ operations(n-1) + operations(n-3)}. (We add two because we multiply and subtract in each recursive call.) We translate this into a loop by 1) keeping the base case the same and 2) starting with a list of the operations of n=2,3,4 (which is [operations(2), operations(3), operations(4)]) and update it as many times as needed, which is just n-4 times. I labeled the list 'start' in my function and do the update carefully without dropping any needed values.

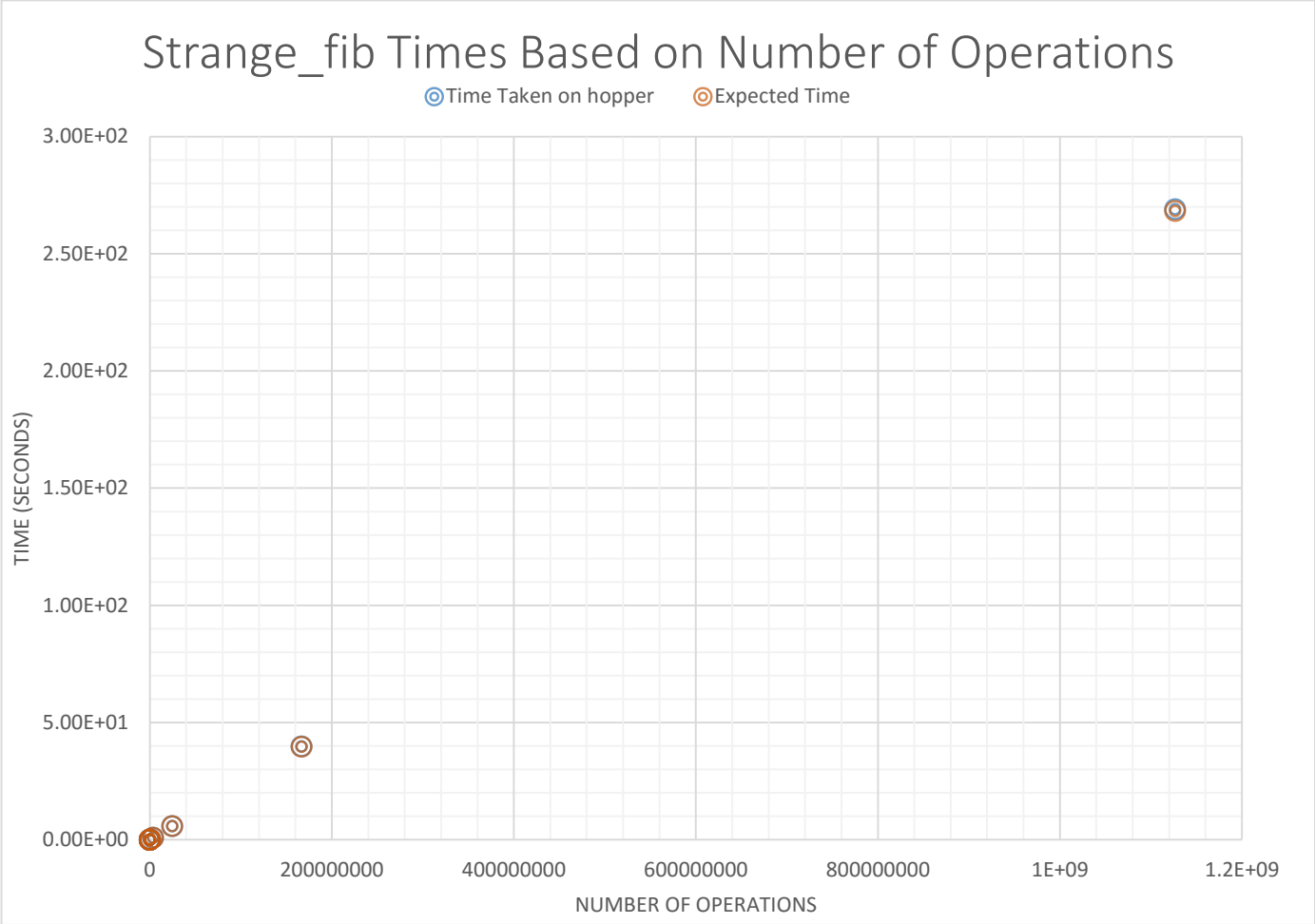
The times are calculated by importing strange_fib in the file `fib_time.py` and making the following changes:

- def fib_time(n, repeats=0):
  - def fib_n(): return strange_fib(n)
  - return func_time(fib_n, repeats)
- for i in range(5,60,5):
  - print "The time for strange_fib of ",i," is this: ", fib_time(i)

The last column is self-explanatory.

CS 107
Divesh Otwani
11/2/15

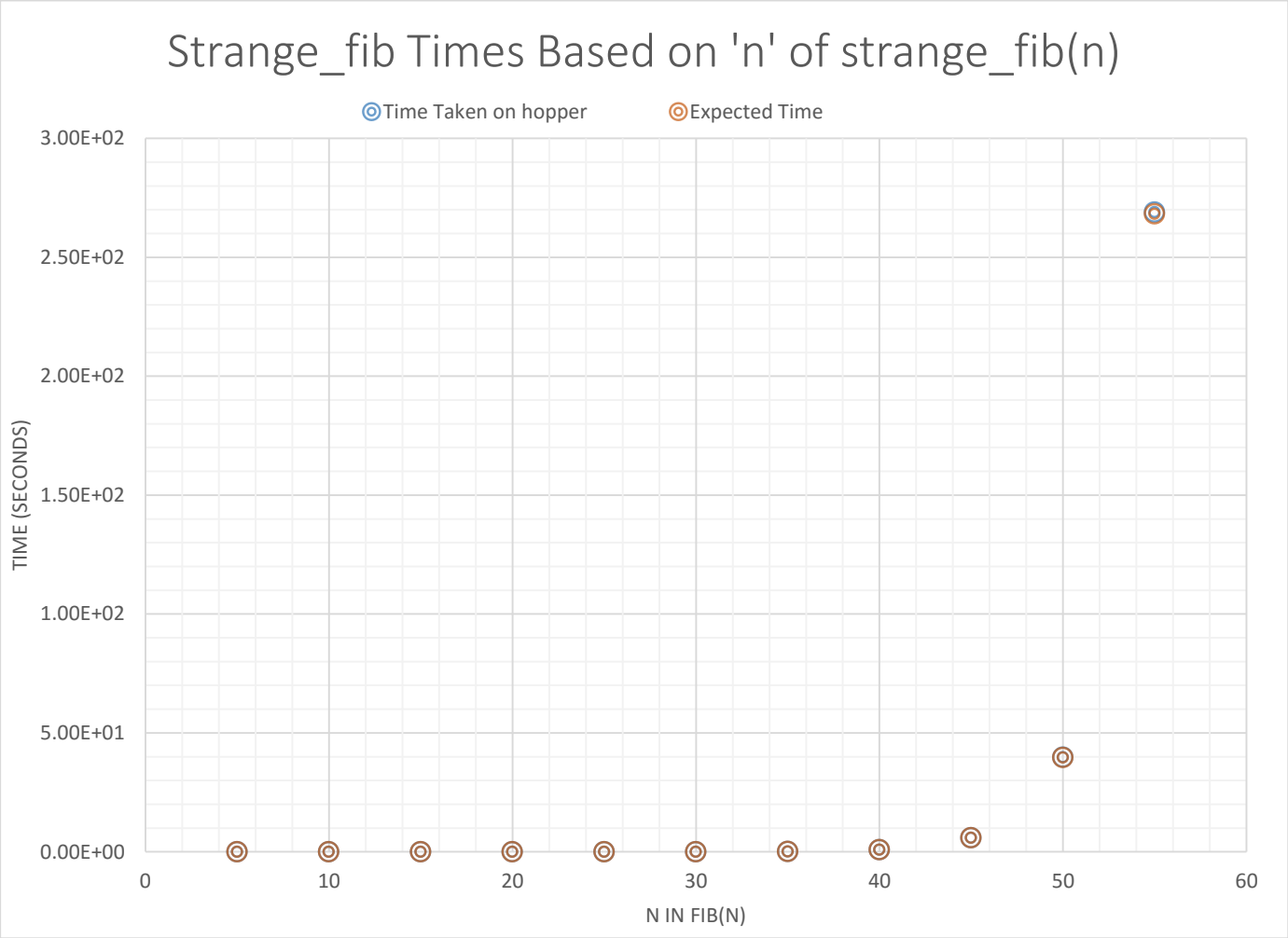
Graphs:



Again, the graph is hard to see so looking at the raw data is helpful:

Number of Operations	Time Taken on hopper	Expected Time	Error
4	1.34E-06	9.52381E-07	2.89E-01
36	9.02E-06	8.57143E-06	4.97E-02
256	6.16E-05	6.09524E-05	1.05E-02
1742	0.000416573	0.000414762	4.35E-03
11790	0.002812391	0.002807143	1.87E-03
79728	0.019000375	0.018982857	9.22E-04
539082	0.1285055	0.128352857	1.19E-03
3644944	0.871316	0.86784381	3.98E-03
24644824	5.886621	5.867815238	3.19E-03
166632768	39.796624	39.67446857	3.07E-03
1126665694	268.970102	268.2537367	2.66E-03

CS 107
Divesh Otwani
11/2/15



We use the previous formula, {f(x) = x / (4.2E06) | x = operations(n)} to compute the expected times of strange_fib(60) and strange_fib(100). These are about, respectively, 1813 seconds or 30 minutes and 7922417022 seconds or 251 years. These are still huge amounts of time.

Comparing Upgrades

The algorithm upgrade is far more important than the hardware upgrade here. We see the computer upgrade within fib makes very little difference, approximately cutting the time in a quarter (e.g., fib(60):18 days to 6 days). The algorithm change is drastic. Strange_fib calculates the 100^{th} fibonacci number in one 14916^{th} of the time of the basic recursive fib, even on the newer machines (e.g., fib(100):3744132 years to 251 years). This indicates a larger point about this lab; the drastic improvement of an algorithm's complexity, especially if from say an exponential shape to a quadratic shape, is far more important than improving hardware. This is not to say what works best on some hardware should not be considered. It simply points to the value of a much "better" algorithm/ recipe/ method with respect to some resource using factor- in this case, adding or other basic operations.