Introduction to Computer Science CS105

Lab 6: Complexity (Fall 2014 edition)

This week's lab focuses on analysis of existing algorithms, rather than programming. You may obtain the sample data and programs mentioned below by checking out the `resounce_complexity-Fall_2014` project in Eclipse, but you are not required to submit modified program files: You should **hand in your answers on paper or Team→Commit a PDF file of your results** this week.

1. Consider the definition of the `fib` function, for finding numbers in the Fibonacci sequence, that is given in the file `fib.py` and shown below. Note that this algorithm arises from "basic recursive design" and is probably the one you used in Lab 2. **Give a formula** that indicates (in terms of the function's parameter n) the total number of times this function adds elements of the Fibonacci sequence, as it works to find `fib(n)`. (So, the number of operations for fib(3) would be 1, and the number for fib(4) would be 2.)

   You may either give an exact formula, or find two functions that enclose the actual number and have the same "shape" (e.g. both linear, both quadratic, both exponential, etc.). If you don't know how to get started, try drawing call trees for fib(4), fib(5), and fib(6), and write down the *number of additions* performed in each. Then see if you can come up with a way to find *the number of additions* needed for fib(7) without writing down the whole tree, and generalize this process. You can then either try to find a precise formula for the sequence of numbers, or find a class of "larger trees" and "smaller trees" that are easier to analyze (and these will then give an upper bound and a lower bound for the amount of work).

2. See if the number of additions given by your formula in Part 1 really corresponds to the amount of time needed for this function, by using the table of running times shown below the function. The table of running times (below and in `fibonacci_time*.txt` in the repository) shows the amount of processing time needed to compute `fib(n)` for several values of n, both before and after an upgrade of computer hardware.

   If you have an exact formula for the number of additions in Part 1, see if the number of additions corresponds to the processing time on each computer. To do this, check to see if the ratio of the number of additions to the processing time should be roughly the same for all n (on a given computer). You may want to use a spreadsheet (such as the libreoffice spreadsheet) to do these calculations, and **record the addition rate** (the typical number of additions per seconds) that best fits the computations for each generation of computer. Note that, if the addition rate is consistent, you can use it to predict the running time of values not given in the table.

   If your answer to Part 1 has an upper and lower bound with the same general shape, you don't need to find a single addition rate, but you should **find a single function** that has the *same shape* as your upper and lower bounds, and fits the timing data. Note that a linear function will have the same *difference* between each pair of successive values, and this difference will tell you the slope of the linear function that fits the data. An exponential function will have the same *ratio* between successive values, and this ratio tells you the base for the exponential function.

3. **Produce a graph (or two)** showing the measured timing numbers for each computer (plotted as points) and your predicted time for that computer (plotted as a line). Use a logarithmic scale for the (vertical) time axis of your graph. You may use graphing software if you know some already, or your lab instructor may provide information about tools such as libreoffice or gnuplot, or you may use pencil and paper.

4. Based on your answer to Part 2, **give an estimate** of how long it would take each generation of computer to find the 60^{th} and 100^{th} numbers in the Fibonacci sequence with this algorithm. Give the estimate both in seconds and in some more convenient unit (e.g., the $n=5$ and $n==55$ times on pre-2010 computers could be listed as "4 microseconds" and "1.6 days", respectively).

5. Optional (extra credit): Use the `func_time` function of the file `fib_time.py` to collect timing data for `strange_fib(n)` for values of n from 5 to 55 (in steps of 5). If you're not sure how to time strange_fib, look at the `fib_time` function, which shows an example of using `func_time`. You can even copy it and then just change your copy to time `strange_fib` instead. The file `fib_time_lots.py` includes the loop that was used to gather timing data for our table; you are welcome to copy or modify it too. Once you have data, make **a chart and graph** for the `strange_fib` function, and give an estimate for the **times to compute** elements 60 and 100. In other words, gather data and then repeat the previous exercises with `strange_fib`. Then, **briefly discuss** the relative importance of the computer upgrade vs. the algorithm upgrade.

6. Optional: Write a function that uses tail recursion to find elements of the Fibonacci sequence, and do a similar evaluation for it. If you know how to use loops or other relevant programming techniques in Python, feel free to compare them as well.

Note that the way to compute elements of the Fibonacci sequence with the smallest number of additions is to use the "closed form" equation for the Fibonacci sequence, which is discussed at `http://ulcar.uml.edu/~iag/CS/Fibonacci.html`:

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

Unfortunately, describing the computing resources needed for this computation is surprisingly tricky.

For reference, the code for the "basic recursive design" version of fib(n) is:

```
# fib(n) is the n'th element of the Fibonacci sequence
def fib(n):
    if n<=2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

The run-times needed for the Python function using the basic recursive algorithm, for the pre-2010 and post-2011 lab computers, are shown in seconds below:

n	**seconds needed to find fib(n)**	
	(pre-2010)	(post-2011)
5	0.000004	0.0000016
10	0.00005	0.000018
15	0.0006	0.00020
20	0.006	0.0022
25	0.07	0.024
30	0.8	0.27
35	9.	3.0
40	100.	33.
45	1100.	360.
50	12000.	4000.
55	140000.	45000.
60	???	???
100	???	???