Homework 1:

due January 20, 2019

Individual, Word set submissions only

In this homework, we start an experimental evaluation of different algorithms to calculate Fibonacci numbers.

The following Python code calculates recursively the Fibonacci numbers, defined by

$$f_n = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ f_{n-1} + f_{n-2} & \text{for } n \geq 2 \end{cases} .$$

```
def rec_fib(n):
    if n < 2:
        return n
    else:
        return rec_fib(n-1)+rec_fib(n-2)
```

If you want to, you can use an equivalent implementation in C or C++. However, Java code cannot be measured consistently and you **cannot** use Java. In order to measure the execution time of the code, we want to run this code many different times. We need to be careful with compiler optimizations, something that Python as an interpreted language does not offer. Thus, we are using something like

```
def measure(n):
    start = time.time()
    for _ in range(50):
        x = rec_fib(n)
    end = time.time()
    return (end-start)/50
```

which will measure 50 executions of the same code. An optimizing compiler might realize that the value stored in x is never used and optimize the call to rec_fib away and then the whole loop ending in an execution time of zero. If you use a compiler, make sure you understand the optimization settings.

Experimental results are useless without some idea of the error introduced. For a given argument *n*, we measure the average of 50 runs and put them into a spreadsheet or a Pandas data frame. Below is an example of what I get and port into excel:

As you can see, the arguments range from 20 to 24 and for each argument, I have collected 20 averages. Because the individual numbers are averages, we can assume with very little loss of precision that they are normally distributed. Using the Excel functions for each row, I create the

¢	21	22	23	24
0.002181082	0.00338392	0.0055643	0.00859566	0.01353488
0.002053404	0.0032958	0.00532308	0.00847372	0.01351704
0.001974921	0.00332874	0.0052554	0.00836268	0.01350424
0.002083135	0.00341684	0.00529878	0.0086008	0.01346298
0.002024961	0.0033226	0.0053118	0.00846503	0.0137813
0.002058864	0.00336964	0.00522604	0.00833566	0.01351563
0.002006254	0.0038552	0.00531948	0.0083283	0.01344252
0.002006545	0.00335502	0.00534484	0.00846822	0.01382258
0.002105517	0.0033311	0.00548464	0.00883558	0.0134329
0.002009063	0.00333486	0.00528978	0.00846166	0.01363992
0.002102284	0.00334436	0.0052618	0.00839112	0.01389652
0.00197516	0.0033129	0.00533824	0.00860694	0.01363578
0.002098684	0.00334868	0.00538214	0.00839832	0.01435782
0.00199966	0.00333528	0.00531572	0.00839966	0.01446568
0.0020507	0.00361898	0.0054035	0.0083988	0.01420108
0.001987734	0.00333842	0.0053153	0.00856208	0.01393395
0.002071166	0.00335232	0.00534248	0.0083988	0.0134236
0.002037902	0.0033919	0.00547316	0.0084745	0.01385018
0.002086115	0.00335876	0.0052628	0.00837992	0.01344764
0.001990895	0.00335156	0.00531972	0.00880532	0.01406328
0.002045202	0.00338734	0.00534165	0.00848714	0.01374648
5.39213E-05	0.00012856	8.4075E-05	0.00014229	0.00032209
3.10572E-05	7.4049E-05	4.8425E-05	8.1955E-05	0.00018551

average, the confidence value and finally the confidence interval at the $\alpha = 0.01$ level, that is, my confidence of 99% that the true value of the execution times is between average minus confidence value and average plus confidence value. You will have to do the same, gathering

and processing values for $n = 1$ to $n = 40$. You can see from the little sample that the execution times increase fast. On your setup, you might not quite make it to $n=40$. If the times start exceeding 10 seconds, you might stop.

Collect the results into a table, giving the argument, the average time and the confidence value.

You then are to add a graph that shows the same values with the confidence intervals. Excel, OpenOffice, LibreOffice, Pandas, Matplotpy, Mathematica, and Mathworks will all allow you to create such a table. You can export the table as a pdf file and use for instance pdf latex to generate the report.

If you look at the table values closely, you see that they will grow like the Fibonacci numbers themselves. This is because the run times are dominated by the number of recursive calls.

If the argument n is 0 or 1, there is no recursive call. If the argument is 2, then there are recursive calls with arguments 0 and 1 and no further calls. If the argument is 3, then there will be recursive calls with arguments 2 and 1, the former creating 2 more recursive calls. Thus, we have

Argument	**Recursive Calls** r_n
0	$r_0 = 0$
1	$r_1 = 0$
2	$r_2 = 2$
3	$r_3 = r_2 + r_1 + 2 = 4$

Develop a recurrence relation for the number of recursive calls r_n. Then prove by induction that $r_n = -2 + f_{n-1} + f_n + f_{n+1}$ for $n > 1$.