First Programming Assignment

In this assignment, you will measure the speed of a recursive version of the Fibonacci number on your system and compare it to a non-recursive version.

Measuring Time:

In principle, we could measure the timing of computer programs using a stop-watch. This would involve determining exactly when a program terminates, which could be difficult to achieve. In general, we are better off using the system time. System time has a better resolution than a stopwatch or a phone application and human reaction times do not have to be taken into consideration. Almost all programming environments allow you to measure time well, for example, using the time module in Python 3 or the <chrono> library in C++.

Unfortunately, other processes in a system can have a large influx on measured times. For example, in a Java environment, measurements are almost useless because the garbage collector can start and slow down any program. This is why you are not allowed to use Java for this programming assignment. Even in a modern multi-core architecture with maybe a dozen of threads that can run in parallel, contention for the RAM-cache interface, contention for shared caches, or a sudden burst of system processes can slow down any single thread. It is therefore best to measure performance several times. In what follows, we will use a for loop to execute a process to be measured several times. Then we will repeat several times to get a number of timings. Finally, we will use some statistics to find confidence intervals for the timing.

When you measure timing, you are really measuring the timing of an implementation. If you are using a compiler, you can set its optimization levels. At one setting, the compiler might figure out that you are not actually using the result of the function whose timing you are measuring and optimize the function call away. If you get very good runtimes, this might be the reason. You will still see counter-intuitive timings that are attributable to such things as cold cache misses.

Fibonacci Numbers:

The Fibonacci numbers are defined by

$$f_i = \begin{cases} i & \text{if } i < 2 \\ f_{i-1} + f_{i-2} & \text{if } i \geq 2 \end{cases}$$

The recursive implementation uses exactly this definition. As each function call with an argument larger than one generates at least two function call, which in turn can each generate two function calls, the number of function calls even for moderate argument values is very high. In contrast, maintaining two variables and updating them is much more efficient. After initializing two variables, `cur` and `pre`, we just update them using

```
cur, pre = cur+pre, cur.
```

If you use C or C++, you need to implement this tuple assignment using a temporary variable.

```cpp
int main(int argc, const char * argv[]) {
    unsigned int x = 0;
    for(unsigned int value = 0; value != 36; value++)   {
        auto start = std::chrono::high_resolution_clock::now();
        for(int i=0; i<10; i++) {
            x += fib(value);
        }
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = duration_cast<std::chrono::nanoseconds>(end-start);  Use of function template name with no...
        std::cout << value << "\t" << duration.count()/10 << std::endl;
    }
    std::cout << x << std::endl;
    return 0;
}
```

Figure 1: C++20 implementation of the timer

Statistical Processing

All measurements are subject to measurement errors. We usually use a statistical model in order to extract information on measurement errors. We **assume** that our runtimes consists of the true runtime plus an error component that is normally distributed. This is certainly not the case, but it is a good enough assumption in our case. We repeat each measurement several times, a good value would be 25 times. We then calculate the sample mean (average) and sample standard deviation. From these and the count, we can calculate the confidence interval size of the student-t distribution. We finally graph average-confidence interval size and average+confidence interval size.

The reason for this procedure is that the average value of a number of runs is much closer to being normally distributed. However, we also need to measure the sample standard deviation, which creates its own error, so we use the student-t distribution instead of the normal distribution.

```
import time

for i in range(1,25):
    print(i)
    for j in range(20):
        start_time = time.perf_counter()
        for _ in range(50):
            x = fibonacci(i)
        duration = (time.perf_counter() - start_time)/50
        print("{:12.10f}".format(duration))
    print("\n")
```

Figure 2: Python implementation of a timer. This will measure the performance for each value in 25 batches of 50 runs each.

Hand-In:

You need to submit a single pdf file with:

(1) A title and your name
(2) A description of your code (one paragraph)
(3) A listing of your code (as figures or embedded in the text).
(4) A table with your results **after** statistical processing
(5) A graphical representation of your results. Extra credit if you figure out how to use error-bars, either using excel (very difficult), matlab or Mathematica (not so easy), or seaborn (simple, but you need to know to use numpy, matplotlib.pyplot, and seaborn).
(6) A short text that summarizes your findings and references the table and the figure.

Here is a Table with some values as an example on how to format results:

Value	Recursive Fibonacci (nsec)	Good Fibonacci (nsec)
0	17.245 ± 2.398	10.012± 0.238
1	11.650 ± 0.134	11.592± 0.024
2	13.650 ± 0.282	11.452± 0.109
3	15.783 ± 1.094	12.761± 0.823