# CMPS 2200 Assignment 1

**Name:** Lily Yee

In this assignment, you will learn more about asymptotic notation, parallelism, functional languages, and algorithmic cost models. As in the recitation, some of your answer will go here and some will go in `main.py`. You are welcome to edit this `assignment-01.md` file directly, or print and fill in by hand. If you do the latter, please scan to a file `assignment-01.pdf` and push to your github repository.

1. (2 pts ea) **Asymptotic notation**

- 1a. Is $2^{n+1} \in O(2^n)$? Why or why not? .
  - Yes, because $2^{n+1} = 2 \cdot 2^n$, so if a constant $c = 2$, then $2^{n+1} \in O(2^n)$. This can be seen by taking $2^{n+1}$ and seeing $2^1 \cdot 2^n = O(2^n)$.
  -
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  - No, because if you have $2^{2n} \leq c2^n$, you can go further and see that $2^n \leq c$. This is false because $2^n$ is not bounded, so there cannot be a constant $c \geq 1$ that satisfies $2^n \leq c$.
  -
- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  - $\lim_{n \to \infty} \frac{n^{1.01}}{\log^2 n} \longrightarrow \lim_{n \to \infty} \left( \frac{1.01 n^{.01}}{\frac{1}{n \ln 2}} \right) \longrightarrow \infty$
  - No, through the limit method I found that $n^{1.01} \notin O(\log^2 n)$.
  -
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  - Using the limit method shown above, it is found that $\log^2 n$ asymptotically dominates $n^{1.01}$. This means $\log_2 n$ is the lower bound, so $n^{1.01} \in \Omega(\log^2 n)$.
  -
- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  - No, because $O((\log n)^3)$ is not an upper bound for $\sqrt{n}$.
  -
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  - Yes, because polynomial functions will grow faster than polylogarithmic ones.

2. **SPARC to Python**

Consider the following SPARC code of the Fibonacci sequence, which is the series of numbers where each number is the sum of the two preceding numbers. For example, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 ...

$$
\begin{array}{l}
foo\ x = \\
\quad \texttt{if}\ \ x \leq 1\ \ \texttt{then} \\
\quad\quad x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ \ (ra, rb) = (foo\ (x-1))\ \ ,\ \ (foo\ (x-2))\ \ \texttt{in} \\
\quad\quad\quad ra + rb \\
\quad\quad \texttt{end}.
\end{array}
$$

- 2a. (6 pts) Translate this to Python code – fill in the `def foo` method in `main.py`
- 2b. (6 pts) What does this function do, in your own words?

The function takes x and first checks to see if it equals 1, in which case it will return 1. If x does not equal 1, the function takes x-1 and x-2, adds them together, and then returns the value.

3. **Parallelism and recursion**

Consider the following function:

```
def longest_run(myarray, key)
    """
     Input:
       `myarray`: a list of ints
       `key`: an int
     Return:
       the longest continuous sequence of `key` in `myarray`
    """
```

E.g., `longest_run([2,12,12,8,12,12,12,0,12,1], 12) == 3`

- 3a. (7 pts) First, implement an iterative, sequential version of `longest_run` in `main.py`.
- 3b. (4 pts) What is the Work and Span of this implementation?

- 3c. (7 pts) Next, implement a `longest_run_recursive`, a recursive, divide and conquer implementation. This is analogous to our implementation of `sum_list_recursive`. To do so, you will need to think about how to combine partial solutions from each recursive call. Make use of the provided class `Result`.
- 3d. (4 pts) What is the Work and Span of this sequential algorithm?
- 3e. (4 pts) Assume that we parallelize in a similar way we did with `sum_list_recursive`. That is, each recursive call spawns a new thread. What is the Work and Span of this algorithm?