CMPS 2200 Assignment 1

Name:___________________________________

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? .

No, this is because 2^n+1 = 2*2^n, meaning it grows twice as fast 2^n.

- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?

No, this is because 2^2^n is a double exponential function, which grows much faster than 2^n, a single exponential function.

- 1c. Is $n^{1.01} \in O(\log^2 n)$?

No, this is because despite being slightly greater than 1, n^1.01 grows polynomially, which is faster than the logarithmic growth of log^2 n.

- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?

Yes, this is because n^1.01 dominates log^2n, meaning it grows at least as fast as log^2 n.

- 1e. Is $\sqrt{n} \in O((\log n)^3)$?

No, this is because sqrt(n) grows polynomially, which is faster than the logarithmic growth of (log n)^3.

- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?

Yes, this is because sqrt(n) dominates (log n)^3, meaning it grows at least as fast as (log n)^3.

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 . . .

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

- 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?

.
. This code returns the xth value in the Fibonacci sequence.
.
.
.
.
.
.

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?

.
.
.

The work of this implementation is O(n), this is because for each element a constant number of operations are performed, so the work is proportional to the size of the list.

The span of the implementation is also O(n), this is because every operation depends upon the previous operations to be performed. This makes the span proportional to the length of the list.

- 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?

  The work of this sequential algorithm is O(n log n), this is because it makes two recursive calls per level and merges in O(n) time.

  The span of this sequential algorithm is O(log n) , this is because the recursion depth is O(log n), with each level requiring only O(1) time to merge.

- 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?

The work of the algorithm is O(n log n), this is because even with the use of paraellization, the number of operations across all recursive calls remains the same.

The span of the algorithm is O(log n), this is because the recursive calls can be executed in parallel. Since the recursion depth is O(log n), the span is O(log n).