CMPS 2200 Assignment 1

Name: Max Curl

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 because as the limit goes to infinity both the numerator and denominator are both infinity. The derivative of both shows that as n goes to infinity the limit becomes 2 which means 2^n+1 is in Theta(2^n)

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

  No because when you take the derivative of the top and bottom as the n goes to infinity the answer is still infinity. This means that 2^(2^n)) is in Omega(2^n)

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

  For certain values of n log^2(n) does dominate n^1.01 but as n approches infinity n^1.01 is in O(Log^2(n))

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

  For certain values log^2(n) is larger than n^1.01 but as n approaches infinity n^1.01 is not in Omega(log^2(n))

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

  No because when you keep taking the derivative the final answer as n goes to infinity is infinity. This shows that sqrt(n) is in Omega((log(n))^3)

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

  Yes because as you keep taking the derivative the final answer is infinity as n goes to infinity. This shows that sqrt(n) is in Omega((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 ...

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

- 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?

Whenever x is greater than 1 the function recursivley calls itself until it reaches the first if condition. In the end the program is just counting the number of 1's until it reaches the final answer.

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?