CMPS 2200 Assignment 1

Name: Shayne Shelton

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; if $c = 3$, then $3 \cdot 2^n \geq 2 \cdot 2^n$ for all $n \geq n_0$

  $2^n \cdot 2^1 = 2 \cdot 2^n$

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

  No; the exponetial behavior of the exponent will result in the first statement surpassing 2^n no matter what c it is multiplied by

  $(2^{2^n} \leq c \cdot 2^n) \log_2$

  $2^n \leq cn$ for $n \leq n_0$ ← not possible

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

  No; $n^{1.01}$ grows faster than a just n. Linear functions have larger growth rates than log functions. Even if log is squared, the linear fctn will dominate it

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

  Yes; the first statement's somewhat linear nature means that it dominates over the log fctn. With $\log^2$ as the best case, $n^{1.01}$ can still be within the domain

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

  No; any root fctn will dominate over a log fctn. So since $\sqrt{n}$ is greater than the worse case scenario, it is not in the domaine

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

  Yes, with $\log^3$ as the best case, the root function is greater than the second statement

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?

Given x, the function outputs the x_{th} term of the fibonacci. It recursively calls foo to calculate the term. The base case is when x = 0,1. Summing the "previous two terms" each time eventually reveals the 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?

$W(n) = O(n)$

$S(n) = O(n)$

.
.
.
.
.
.

- 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?

.
.
.
.
.
.
.
.