CMPS 2200 Assignment 1

Name: Gabe Epstein

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. **Asymptotic notation**

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

  Yes because c * 2^(n) ≥ 2^(n+1) for c ≥ 2

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

  No. Assuming it were O(2^n) then a value c would exist such that c * 2^n ≥ 2^(2^n)).
  If you divide both sides by 2^n you get c * 1 ≥ 2^n and there is no value c for which
  that is true.

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

  No.

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

  Yes.

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

  No.

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

  Yes.

- 1g. Consider the definition of "Little o" notation:

$g(n) \in o(f(n))$ means that for **every** positive constant c, there exists a constant n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. There is an analogous definition for "little omega" $\omega(f(n))$. The distinction between $o(f(n))$ and $O(f(n))$ is that the former requires the condition to be met for **every** c, not just for some c. For example, $10x \in o(x^2)$, but $10x^2 \notin o(x^2)$.

.

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

If we assume that 0(g(n)) ∩ w(g(n)) is not the empty set and call it f(n) we can see that it is in fact the empty set.

If it were the empty set it would imply a constant C such that g(n) * C is both <f(n) and > f(n), and that f(n)/g(n) must be both < and > C which can't happen.

2. **SPARC to Python**

Consider the following SPARC code:

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

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

This is 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. First, implement an iterative, sequential version of `longest_run` in `main.py`.
- 3b. What is the Work and Span of this implementation?

- 3c. 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. What is the Work and Span of this recursive algorithm?
- 3e. 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?