CMPS 2200 Assignment 1

Name: Caroline Casella

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
  - we say f(n) dominates g(n) if g(n) is less than or equal $c \cdot f(n)$
  - $2^{n+1} = 2 \cdot 2^n$
  - there has to be a constant C such that $2^{n+1} \leq C \cdot 2^n$
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  - No
  - 2^{2^n} is greater than 2^n
  - $\log_2(2^{2^n}) = 2^n \Rightarrow \log_2(2^n) = n$
  - $\log 2^{2^n} \leq \log C(2^n)$

    $\frac{2^n}{n} \leq \frac{C \cdot n}{n}$

    $2^n/n \leq \log C \longrightarrow$ incorrect: no C that satisfies
- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  - No
  - $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n)$
  - Then: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{g'(n)}{f'(n)}$
  - $= \lim_{n\to\infty} \frac{n^{1.01}}{\log^2 n} = \lim \frac{1.01 - n^{1.01-1}}{\frac{1}{n}} = \lim_{n\to\infty} 1.01 \cdot n^{0.01} = \infty \rightarrow$ doesn't make sense
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  - Yes
  - $n^{1.01} \geq C \cdot \log^2 n \Rightarrow \frac{1}{\log n} \cdot n^{1.01} \leq C \cdot \log n$
  - $n^{1.01} \leq C \cdot \log(n)$
  - $n^{1.01}$ will always dominate log n
- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  - Yes
  - $(g(n))^b \in O(a^{g(n)})$
  - $g(n) = \log(n)$, $b = 3$, $a = \sqrt{2}$
  - $(\log(n))^3 \in O(\sqrt{2}^{\log(n)}) = O(\sqrt{n})$

    $(2^{1/2})^{\log n} = n^{1/2}$
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  - $\sqrt{n} \geq c \cdot (\log n)^3 \Rightarrow \lim \frac{\sqrt{2}}{(\log n)^3} \geq \lim C$
  - $\infty \geq c$
  - $\sqrt{n}$ is $\Omega(\log n)^3$
- 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.

- assume $O(g(n)) \cap \omega(g(n))$ is not empty
- $o(g(n)) \Rightarrow g(n) \cdot c > f(n) \Rightarrow c > \frac{f(n)}{g(n)}$
- $\omega(g(n)) \Rightarrow g(n) \cdot c < f(n) \Rightarrow c < \frac{f(n)}{g(n)}$
- we know that it is not possible that for every c, it is both greater than and less than $\frac{f(n)}{g(n)}$
- our initial assumption must then be incorrect since there exits no value of c that is both greater than and less than $\frac{f(n)}{g(n)}$
- so therefore $o(g(n)) \cap \omega(g(n))$ must be an empty $\blacksquare$

2. **SPARC to Python**

Consider the following SPARC code:

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

- 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 function prints out the Fibonacci sequence. The Fibonacci sequence forms a sequence of numbers where each number is the sum of the two numbers before it.

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?

- Work: $O(n)$
- Span: $O(n)$

.
.
.
.
.

- 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 sequential algorithm?

  .
  . Work: $O(n \log n)$
  . Span: $O(\log(n))$
  .
  .
  .
  .
  .
  .
  .
  .

- 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?

. Work: $O(n \log n)$
. Span: $O(n)$
.
.
.
.
.
.