CMPS 2200 Assignment 1

Name: Ella Moses

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? . using the limit method
  - $\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to\infty} \frac{2^n \cdot 2}{2^n} = \lim_{n\to\infty} 2 = 2$
  - $2 > 0$ so $2^{n+1} \in \Theta(2^n)$
  - yes, $\Theta(2^n) = O(2^n) \cap \Omega(2^n)$ so it is both $O(2^n)$ and $\Omega(2^n)$
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not? using the limit method
  - $\lim_{n\to\infty} \frac{2^{2^n}}{2^n} = \lim_{n\to\infty} \frac{2^n}{n} = \infty$
  - No, $2^{2^n} \in \Omega(2^n)$
- 1c. Is $n^{1.01} \in O(\log^2 n)$? using the limit method
  - $\lim_{n\to\infty} \frac{n^{1.01}}{\log^2 n} = \infty$
  - No, $n^{1.01} \in \Omega(\log_2 n)$
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$? using the limit method
  - $\lim_{n\to\infty} \frac{n^{1.01}}{\log^2 n} = \infty$
  - Yes, $n^{1.01} \in \Omega(\log_2 n)$
- 1e. Is $\sqrt{n} \in O((\log n)^3)$? using the limit method
  - $\lim_{n\to\infty} \frac{n^{1/2}}{(\log n)^3} = \infty$
  - No, $n^{0.5} \in \Omega(\log_2 n)$
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$? using the limit method
  - $\lim_{n\to\infty} \frac{n^{1/2}}{(\log n)^3} = \infty$

  Yes, $n^{0.5} \in \Omega(\log_2 n)$

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?

This function finds the xth element of the Fibonacci sequence by adding together the previous two elements. The base cases are x=0 and x=1. Since these are the first two elements of the sequence, we can't add the previous two elements, so we just return x. For any element after the second element, this function will return the sum of the previous two elements. When you input a number x, this function will find the xth element of the Fibonacci sequence by recursively calling the function until the base case is reached, then the function will add all of the elements leading up to the xth element together.

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 function is O(n) since each element must be compared to the key as we iterate through the list. The worst case span is also O(n) because this is a sequential algorithm and the elements are compared one at a time.

- 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?

$W(n) = \underbrace{2\left(\frac{n}{2}\right)}_{\text{Split into 2 branches of size } n/2 \text{ in each recursive step}} + \underbrace{O(1)}_{\text{Work at each level (creating results)}}$

root $1 = 1$

$L_1 = 2 \cdot 1 = 2$

$L_2 = 2^2 \cdot 1 = 4$

leaf dominated

$W(n) = O(n)$ where n = # of leaves

$S(n) = \underbrace{2S\left(\frac{n}{2}\right)}_{\text{2 since this is not parallel}} + O(1)$

root $1 = 1$

$L_1 = 2 \cdot 1 = 2$

$L_2 = 2^2 \cdot 1 = 4$

leaf dominated

$S(n) = O(n)$ where n = # of leaves

- 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?

$W(n) = \underbrace{2\left(\frac{n}{2}\right)}_{\text{Split into 2 branches of size } n/2 \text{ in each recursive step}} + \underbrace{O(1)}_{\text{Work at each level (creating results)}}$

root $1 = 1$

$L_1 = 2 \cdot 1 = 2$

$L_2 = 2^2 \cdot 1 = 4$

leaf dominated

$W(n) = O(n)$ where n = # of leaves

$S(n) = \underbrace{S\left(\frac{n}{2}\right)}_{\text{1 since this is not parallel}} + O(1)$

$S(n) = O(\log n)$

Span = depth of tree = $\log n$