CMPS 2200 Assignment 1

Name: Maeren Hay

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, because $2^{n+1} = 2 \cdot 2^n$, pick $c = 2$ and $X_0 = 1$

  for all $n \geq 1$, $2^{n+1} = 2 \cdot 2^n \leq c \cdot 2^n$

  So $2^{n+1} \in O(2^n)$

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

  No, 2^{2^n} grows faster than the single exponential 2^n.

  The ratio $\frac{2^{2^n}}{2^n} = 2^{(2^n - n)}$, $\lim_{n \to \infty} (2^n - n) = \infty$, So $2^{2^n} \notin O(2^n)$

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

  No, $n^{1.01}$ grows faster than $(\log^2 n)$. $\frac{n^{1.01}}{(\log n)^2} \to \infty$

  So $n^{1.01} \notin O(\log^2 n)$

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

  Yes, $\frac{n^{1.01}}{(\log n)^2} \to \infty$, as $n \to \infty$. Meaning $n^{1.01}$ dominates $\log^2 n$

  So $n^{1.01} \in \Omega(\log^2 n)$

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

  No, $\sqrt{n} = n^{0.5}$, which grows faster than $(\log n)^3$

  So $\sqrt{n} \notin O((\log n)^3)$

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

  Yes, $\sqrt{n} = n^{0.5}$, which dominates $(\log n)^3$, therefore

  $\sqrt{n} \in \Omega((\log n)^3)$

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 . . .

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

- 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 mimicks the Fibonacci sequence because, if $x \leq 1$, it returns x. Otherwise it returns the sum of the two previous values. Calling foo(n) returns the n^{th} Fibonacci number.

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?

$W = O(n)$ (no parallelism)

$S = O(n)$

- 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?

work: $W(n) = 2W(n/2) + O(1) = \Theta(n)$

Span: $S(n) = 2S(n/2) + O(1) = \Theta(n)$

- 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?

work: $W(n) = 2W(n/2) + O(1) = \Theta(n)$

Span: $S(n) = S(n/2) + O(1) = \Theta(\log n)$