CMPS 2200 Assignment 1

Name: Noelle Fox

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 big O tells us for $O(g(n))$: $n \geq n_0$, $f(n) \leq c \cdot g(n)$. If we write 2^{n+1} as $2 \cdot 2^n$ we can get $2 \cdot 2^n \leq 2 \cdot 2^n$ with $c = 2$ and $2^n = g(n)$. Thus, the formula stands and this is true.

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

2^{2^n} is ~~not~~ asymptotically bounded by $O(2^n)$. Using the same formula from 1a. with $f(n) = 2^{2^n}$ and $g(n) = 2^n$ we get $2^{2^n} \leq c \cdot 2^n$. However, that cannot be true because of how much faster 2^{2^n} grows compared to 2^n. There is no constant that we can use to make $f(n) \leq c \cdot g(n)$

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

NO, $n^{1.01}$ grows exponentially faster.

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

Yes because with Omega we need $f(n) \geq c \cdot g(n)$. And as I stated in 1c. $n^{1.01}$ grows much faster than $\log^2 n$.

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

No because $\sqrt{n}$ can also be written as $n^{.5}$ which is a polynomial and polynomials always grow quicker than logs.

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

Yes because as I discussed in 1e. $\sqrt{n}$ grows quicker than $(\log n)^3$ even when multiplied by some constant

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 repeatedly sums the previous two numbers in a sequence to get Fibonacci's number. It first checks to make sure x is greater than 1 because if it's not there will be no two numbers to sum. If it's greater than one it performs two recursive calls: one to get the value of the number directly before x and one to get the value of the number before that. Finally, it returns the sum of the two calculated values.

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?

$W(n) = O(n)$ → iterate through whole list once

$S(n) = O(1)$ → do one thing at a time, no branches to wait for

- 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?

  - $W(n) = O(n)$
    - → it is recursive so each step it is divided by 2
    - → results are merged which only adds $O(1)$ at each step

    $2T(n/2)+O(1)$ each time → $O(n)$

  - $S(n) = O(\log n)$
    - → list size is halved at each step: divide by 2 until base case
    - → recursive depth $= \log_2 n$ so longest sequence $= O(\log n)$

- 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?

  - $W(n) = O(n)$
    - → Being that the Work is the total number of operations performed across all recursive calls it remains $O(n)$
  - $S(n) = O(\log n)$
    - → Even when running the recursive calls in parallel, the only dependent step is when results are combined the recursion depth remains $O(\log n)$