CMPS 2200 Assignment 1

Name: Rhon Farber

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 $\lim_{n \to \infty} \frac{2^{n+1}}{2^n} = 2$, so 2^{n+1} is 2×2^n which is in 2^n.

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

  No, because $\lim_{n \to \infty} \frac{2^{2^n}}{2^n} = \lim_{n \to \infty} 2^{2^n - n} = \infty$, so 2^{2^n} is not in $O(2^n)$.

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

  No, because $\lim_{n \to \infty} \frac{n^{1.01}}{\log^2 n} = \infty$ because numerator is growing at polynomial growth, and denominator is slower at logarithmic growth, so $n^{1.01}$ is not in $O(\log^2 n)$.

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

  Yes, because $\lim_{n \to \infty} \frac{n^{1.01}}{\log^2 n} = \infty$, so $n^{1.01}$ grows asymptotically faster than $\log^2 n$ meaning $n^{1.01}$ is in $\Omega(\log^2 n)$.

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

  No, because $\lim_{n \to \infty} \frac{\sqrt{n}}{(\log n)^3} = \infty$ which means $\sqrt{n}$ grows asymptotically faster than $(\log n)^3$ meaning $\sqrt{n}$ is not in $O(\log^3 n)$.

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

  Yes, because $\lim_{n \to \infty} \frac{\sqrt{n}}{(\log n)^3} = \infty$, meaning $\sqrt{n}$ is in $\Omega((\log n)^3)$ because $\sqrt{n}$ grows asymptotically faster than $(\log n)^3$.

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 ...

```
foo x =
  if x ≤ 1 then
    x
  else
    let (ra, rb) = (foo (x − 1)) , (foo (x − 2)) in
      ra + rb
    end.
```

- 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?

The foo function takes in x as a parameter and then returns x if x is 0 or 1. Otherwise, it sets ra to x-1 and rb to x-2 and then calls the sum of ra and rb recursively as inputs to foo and then returns the sum of the two recursive calls.
the last call

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?

Work is the length of myarray which is O(n)
Span is also O(n) because work is sequential.

- 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?

The work of this classic divide and conquer algorithm is $O(n)$ because $2n$ nodes are created for the work which is the dominating factor. The span is the depth of the recursion so it is also $O(n)$

- 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 is the same, but span is faster due to threads

$W(n) = O(n)$

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