CMPS 2200 Assignment 1

Name: Shira Rozenthal

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, it is in O(2^n) because, as n approaches infinity,

  the limit of 2^(n+1)/2^n = 2, and so 2^n+1 is 2 * 2^n .

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

  No, it is not in O(2^n) because the limit of their quotient

  approaches infinity as n approaches infinity, so there is no

  constant > 0 that for all n satisfies f(n) <= g(n),

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

  No, it is not in O(log^2 n) because the power function will eventually surpass

  the logarithmic function, so the limit of their quotient approaches infinity.

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

  Yes, it is in Omega(log^2 n) because, as n approaches infinity, the

  limit of n^1.01 / log^2n = infinity, and so for all n there exists a

  constant such that f(n) >= g(n).

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

  No, it is in O((log n)^3) because the limit of their quotient approaches

  infinity as n approaches infinity, and so the logarithmic function will not

  asymptotically dominate the square root for all values of n.

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

  Yes, it is in Omega((log n)^3) because the limit of their quotient approaches

  infinity as n approaches infinity, and so (in contrast to the question above)

  sqrt(n) will asymptotically dominate ((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 ...

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

- 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?

```
The function takes an integer x as an argument and returns the xth Fibonacci
number. The function uses recursion, where it calls itself to calculate the
Fibonacci number of the previous two numbers until it reaches the base case of
x < 1, in which case it returns x. In the end, it returns the sum of the two
previous Fibonacci numbers, which is the xth Fibonacci number.
```

3. **Parallelism and recursion**

Consider the following function:

```python
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) because the function is iterative, so work grows linearly with input size.
S(n) = O(n) because there is no parallelism involved in the function, so S(n) = W(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?
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .

- 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?

.
.
.
.
.
.
.
.