CMPS 2200 Assignment 1

Name: Kevin Skelly

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? .

$2^{n+1} = O(2^n)$

$2 \cdot 2^n = O(2^n)$

$2 \cdot 2^n \le c \cdot 2^n \Rightarrow 2^{n+1} \in O(2^n)$

2^{n+1} is in $O(2^n)$ because for any $c > 2$, $c2^n$ dominates 2^{n+1}

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

Suppose $2^{2^n} \in O(2^n)$. Then there exists a constant c s.t. $\forall n$ beyond some n_0, $2^{2^n} \le c2^n$. Rewriting: $2^{2n} \le c2^n$. Taking log of both sides, we get $2n \le \log c + n \Rightarrow n \le \log c$. No such c can satisfy this for all $n > n_0$, so 2^{2^n} is not in $O(2^n)$

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

$$\lim_{n\to\infty} \frac{\log^2 n}{n^{1.01}} = \frac{\infty}{\infty} \xrightarrow{\text{L'hopital}} \lim_{n\to\infty} \frac{2\log n \frac{1}{n}}{1.01n^{0.01}} = \lim_{n\to\infty} \frac{2\log n}{1.01n^{1.01}} = \frac{\infty}{\infty} \xrightarrow{\text{L'hop}} \lim_{n\to\infty} \frac{2 \cdot \frac{1}{n}}{1.01^2 n^{0.01}} = \frac{2}{1.01^2 n^{1.01}} = 0$$

no

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

yes, by the result of last problem, $n^{1.01}$ asymptotically dominates $\log^2 n$

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

No, there does not exist a c such that $\sqrt{n} \le c(\log n)^3$ for sufficiently large values of n

$$\lim_{n\to\infty} \frac{\sqrt{n}}{(\log n)^3} = 0$$

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

Since $\lim_{n\to\infty} \frac{\sqrt{n}}{(\log n)^3} = 0$, $\sqrt{n}$ is thus 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 ...

```
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` Done in code
- 2b. (6 pts) What does this function do, in your own words?

This function implements the xth iteration of the fibonacci sequence. Given an integer x, the code recursively runs the foo(x-1) and foo(x-2), until the x plugged in is 0 or 1. Then it returns x and sums back up the tree. This returns outputs of the xth level of the fib. sequence, for example foo(4) returns 3, foo(5) returns(5), so foo(6) will return 5+3 = 8

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`. Done in code
- 3b. (4 pts) What is the Work and Span of this implementation?

$W(n) = W(n-1) + 1 \in O(n)$

$S(n) = W(n-1) + 1 \in O(n)$

- 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`. Done in code
- 3d. (4 pts) What is the Work and Span of this sequential algorithm?

$W(n) = O(n)$

$S(n) = O(n)$

work and span are the same because parallelism is not possible

- 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?

$W(n) = O(n)$

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

we can repeatedly divide work among computers, so span is reduced to $\log n$, or the depth of the tree