CMPS 2200 Assignment 1

Name: Benjamin Horowitz

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? .
  - $2^{n+1} = 2^n \cdot 2$
  - We can treat the leading 2 as a constant c
  - It is evident that $c2^n \in O(2^n)$. Therefore $2^{n+1} \in O(2^n)$
  - .

- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  - For any n, the algorithm 2^{2^n} will grow at double exponential growth
  - Since $O(2^n)$ represents single exponential growth $2^{2^n} \notin O(2^n)$
  - .
  - .
  - .

- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  - For any exponent $c > 0$, a polynomial n^c will grow faster than logarithmic functions.
  - At large values of n, $\forall c \in \mathbb{R}$, $n^{1.01} > c\log^2 n$
  - Therefore $n^{1.01} \notin O(\log^2 n)$

- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  - As discussed in the last problem, $n^{1.01}$ grows faster than $\log^2 n$
  - Since we are being asked about lower bounds, we determine if $n^{1.01}$ grows at least as fast as $\log^2 n$
  - Since it does, $n^{1.01} \in \Omega(\log^2 n)$
  - .

- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  - Again, polynomial functions grow faster than logarithmic functions.
  - This is in form n^c where $c > 0$, so $\sqrt{n} \notin O(\log n^3)$
  - .

- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  - However, since $\sqrt{n}$ grows faster than $(\log n)^3$, $\sqrt{n} \in \Omega(\log n^3)$

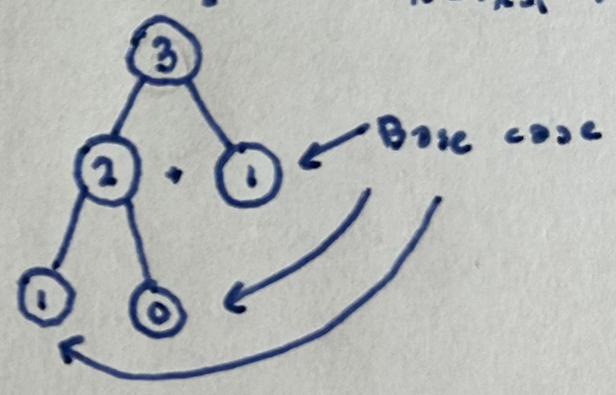
2. SPARC to Python

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 ...

```
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?

It generates the Fibonacci numbers using 2 cases
1) Base case: 1st Fib number. $F_1 = 1$, so this case returns 1.
2) Recursive case. Otherwise it will continuously recursively call $F_n = F_{n-1} + F_{n-2}$. This is the definition of the Fibonacci numbers



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?

The work $W(n)$ and span $S(n)$ are both $\in O(n)$. This is evident, as we only perform constant time operations, but visit every index of myList.

- 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?

Here, $W(n) = 2W(n/2) + O(n)$.
This is because it's divide and conquer, hence the $2W(n/2)$, but we also have to merge the results in $O(n)$.
Here $S(n) = S(n/2) + O(1)$
Again, this comes down to the list being divided, then the recombination step being done in $O(1)$ time.

- 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?

This would mean $W(n) \in O(n \log n)$, $S(n) \in O(\log n)$. This has been gone over in class for algorithms with this same $W(n)$ and $S(n)$ under parallelization.