CMPS 2200 Assignment 1

Name: Camden Yale

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? .

  True. Since, $2^{n+1} = 2 * 2^n$, pick $c \geq 2$ and $n_0 = 0$

  $2^{n+1} \in O(2^n)$

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

  False. $2^{2^n} \neq 2^n \cdot 2^n$

  can't find c and n_0 such that $2^{2^n} \leq c \cdot 2^n$ for all $n \geq n_0$

  $2^{2^n} \notin O(2^n)$

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

  False. Any polylog grows slower than any polynomial.

  $\log^i n \in O(n^j)\ \forall i,j > 0$ and converselv, $n^j \in \Omega(\log^i n)\ \forall j,i > 0$

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

  True. Any polylog grows slower than any polynomial.

  $\log^i n \in O(n^j)\ \forall i,j > 0$ and converselv, $n^j \in \Omega(\log^i n)\ \forall j,i > 0$

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

  False. $\log n = \log(\sqrt{n})^2 = 2\log\sqrt{n}$

  So, $\sqrt{n} = 2^{(\log_2 n)/2} >> \log n$

  Supported by 1d, $\sqrt{n} = n^{.5}$ is a polynomial so it grows faster than any polylog

  $\sqrt{n} \notin O((\log n)^3)$

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

  True. Supported by the previous question.

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  $x \leq 1$  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?

Recursive function that calculates the Fibonacci Sequence.
Adds the result of two previous function calls.
Continues until it reaches the base case of 1 or 0

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: $O(n)$

Span: $O(n)$

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

Work: O(n), n is the length of the list, elements processed after the list is recursivly divided

Span: O(logn) because of the divide and conquer. The depth of the recursion tree is logarithmic

- 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: O(n), total number of nodes in the recursion tree

Span: O(logn), worst case span is equal to the number of levels in the recursion tree