CMPS 2200 Assignment 1

Name: Viraj Choksi

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?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = 2^{n+1}$ $g(n) = 2^n$
  - $2^{n+1} = 2 \cdot 2^n$
  - $2 \cdot 2^n \leq c \cdot 2^n$ // take $c=2$ and condition is satisfied
  - $\therefore$ Yes, $2^{n+1} \in O(2^n)$
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = 2^{2^n}$ $g(n) = 2^n$
  - $\frac{2^{2^n}}{2^n} = 2^{2^n - n} \Rightarrow$ As $n \to \infty$, $2^{2^n - n}$ grows fast and tends to infinity
  - For sufficiently large n, no constant c can satisfy $2^{2^n} \leq c \cdot 2^n$
  - $\therefore$ No, $2^{2^n} \notin O(2^n)$
- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = n^{1.01}$ $g(n) = \log^2 n$
  - $\frac{n^{1.01}}{\log^2 n} \Rightarrow$ As $n \to \infty$, $n^{1.01}$ grows significantly faster than $\log^2 n$
  - For sufficiently large n, no constant satisfies $n^{1.01} \leq c \cdot \log^2 n$ // $\therefore$ No, $n^{1.01} \notin O(\log^2 n)$
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  - $f(n) \geq c \cdot g(n)$ // $f(n) = n^{1.01}$ $g(n) = \log^2 n$
  - As proved in 1c, for sufficiently large n, $n^{1.01} \geq c \cdot \log^2 n$ is satisfied.
  - $\therefore$ Yes, $n^{1.01} \in \Omega(\log^2 n)$
- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = \sqrt{n}$ $g(n) = (\log n)^3$
  - Polynomial fuctions generally grow faster than logarithmic functions

  $\lim_{n \to \infty} \frac{n^{1/2}}{(\log n)^3} = \infty$ // $\therefore$ No, $\sqrt{n} \notin O((\log n)^3)$
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  - $f(n) \geq c \cdot g(n)$ // $f(n) = \sqrt{n}$ $g(n) = (\log n)^3$

  As proved in 1e, Polynomials grow faster than logarithms // Yes, $\sqrt{n} \in \Omega((\log n)^3$

CMPS 2200 Assignment 1

Name: Viraj Choksi

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? .
  - $f(n) \leq c \cdot g(n)$ // $f(n) = 2^{n+1}$ $g(n) = 2^n$
  - $2^{n+1} = 2 \cdot 2^n$
  - $2 \cdot 2^n \leq c \cdot 2^n$ // take $c=2$ and condition is satisfied
  - $\therefore$ Yes, $2^{n+1} \in O(2^n)$
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = 2^{2^n}$ $g(n) = 2^n$
  - $\frac{2^{2^n}}{2^n} = 2^{2^n - n} \Rightarrow$ As $n \to \infty$, $2^{2^n - n}$ grows fast and tends to infinity
  - For sufficiently large n, no constant c can satisfy $2^{2^n} \leq c \cdot 2^n$
  - $\therefore$ No, $2^{2^n} \notin O(2^n)$
- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = n^{1.01}$ $g(n) = \log^2 n$
  - $\frac{n^{1.01}}{\log^2 n} \Rightarrow$ As $n \to \infty$, $n^{1.01}$ grows significantly faster than $\log^2 n$
  - For sufficiently large n, no constant satisfies $n^{1.01} \leq c \cdot \log^2 n$ // $\therefore$ No, $n^{1.01} \notin O(\log^2 n)$
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  - $f(n) \geq c \cdot g(n)$ // $f(n) = n^{1.01}$ $g(n) = \log^2 n$
  - As proved in 1c, for sufficiently large n, $n^{1.01} \geq c \cdot \log^2 n$ is satisfied.
  - $\therefore$ Yes, $n^{1.01} \in \Omega(\log^2 n)$
- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  - $f(n) \leq c \cdot g(n)$ // $f(n) = \sqrt{n}$ $g(n) = (\log n)^3$
  - Polynomial fuctions generally grow faster than logarithmic functions

  $\lim_{n \to \infty} \frac{n^{1/2}}{(\log n)^3} = \infty$ // $\therefore$ No, $\sqrt{n} \notin O((\log n)^3)$
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  - $f(n) \geq c \cdot g(n)$ // $f(n) = \sqrt{n}$ $g(n) = (\log n)^3$

  As proved in 1e, Polynomials grow faster than logarithms // Yes, $\sqrt{n} \in \Omega((\log n)^3)$

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 $x \le 1$ 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 function recursively calculates the Fibonacci sequence. The sequence is defined such that: $F(0)=1$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$ for $n \geq 2$
Each call to foo(x) checks if x is 0 or 1, and if not, recursively computes foo(x-1) and foo(x-2) and then returns their sum.

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: Since the list is iterated through once, the total number of operations is $O(n)$

Span: The longest chain of operations is also $O(n)$, as each element is processed in order

- 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:

$T(n) = 2T(n/2) + O(1) \Rightarrow T(n) = O(n)$ // 2 problems of $n/2$ size and $O(1)$ to combine

Span:

$S(n) = S(n/2) + O(1)$

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

- 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:

Remains the same because every element is still processed once

Span:

depth of recursion = $O(\log n)$ // $O(1)$ work at each level to combine

span: $O(\log(n))$