CMPS 2200 Assignment 1

Name: Aaron Gershkovich

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** (12 pts)

- 1a. Is $2^{n+1} \in O(2^n)$? Why or why not?
. It is because 2^(n+1) = 2*2^(n), thus there exists some value, c, such that c * 2^n >= 2*2^n, such that c >= 2 such as c = 3.
.
.
.
.
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
. It is not because there is no value of c such that c*2^n is is greater than 2^(2^n)) for all n, as taking the log_2 of both sides would yield c* n >= 2^n, which can never be true for all n.
.
.
.
.
- 1c. Is $n^{1.01} \in O(\mathrm{log}^2 n)$?
. No, because N^1.01 grows faster than any function of the form log_x^2(N), meaning that for all n, there is no value of c such that c * log^2n >= n^(1.01). We know this because the limit as n approaches infinity of n^1.01/(log^2(n) is infinite, so n^1.01 dominates.
.
.
.

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

. YES, by the reverse of what was said for 1.3, since the limit as n approaches infinity of n^1.01/(log^2(n) is infinite, n^1.01 dominates, meaning.

.
.
.

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

. No because for sufficiently large n, we must consider the equation that as n approaches infinity for n^1/2/(log^3(n) must approach infinity because any n^(½) will eventually grow faster than any power of log(n). The general case is that any polynomial will grow faster than any power of log(n) at some N

.
.
.

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

. YES, since sqrt(n) increases faster than (log n)^3 at some point, there exists some c such that sqrt(n) > clog(n)^3.

- 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?

This function represents the fibonacci sequence, meaning that each value is the sum of the two values before it. I implemented this recursively, because when given a number, the function is called until the base case is reached with 0 and 1, and the numbers are added up until the the input.s

.
.
.
.
.
.
.
.

3. **Parallelism and recursion** (26 pts)

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?

The Work for this function is O(n) because it goes through all n terms of the list exactly once, as shown by the for loop

The Span is also O(n) because there is only one branch, and the depth of that branch is 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?

. The Work = O(n) because relationship is defined as W(n)=2W(n/2)+O(1), eventually reaching W(n)=2^(logn)W(1)+O(n), or just O(n)

. The Span = O(logn) because we are continually dividing the number of remaining values by 2 (S(n/2) + O(1)) eventually creating a tree with a depth of logn

.

.

.

.

.

.

.

.

- 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?

. If we parallelize it like we did with 'sum_list_recursive' , the characteristic function becomes W(n/2) + O(n), meaning that the work is O(nlogn)

.The span remains the same, S(n) = O(logn)

.

.

.