CMPS 2200 Assignment 1

Name: Abe

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 never eclipses 2^{n+1}~~
Yes, ~~eventually~~, there will exist a constant that you can multiply 2^{n+1} by so that it will eclipse 2^n.

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

Yes. you can take a constant, say 1000000, where you can take another constant, say 100, and when you multiply the by 2^n and 2^{2^n} respectively, 2^n will eclipse 2^{2^n}.

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

no. these functions will continue on their paths forever and never cross no matter what constants they are multiplied by.

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

Yes. for the same reason listed above.

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

no. At tail ends, $\sqrt{n}$ will always eclipse $(\log_n 3)^3$ for large constant values

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

Yes. For the same reason as above

- 1g. Consider the definition of "Little o" notation:

$g(n) \in o(f(n))$ means that for **every** positive constant c, there exists a constant n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. There is an analogous definition for "little omega" $\omega(f(n))$. The distinction between $o(f(n))$ and $O(f(n))$ is that the former requires the condition to be met for **every** c, not just for some c. For example, $10x \in o(x^2)$, but $10x^2 \notin o(x^2)$.

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

need to consider it is not the empty set.

$f(n) \in o(g(n)) \cap \omega(g(n))$

$f(n) = \omega(g(n))$ iff

$g(n) = o(f(n))$ $\longrightarrow o(f(n)) = o(g(n))$

$+ \quad f(n) = o(g(n))$

$c > 0 \quad f(n) < c \cdot f(n)$

So long as c is less than 1, this will lead to a contradiction, so it will not be correct.

2. (3 pts) **Python to SPARC**

Recall the `sum_list_recursive_parallel` function from lecture 2. Specify our implementation in SPARC here.

```
Sum x =
  if |x| = 1 then
    x[0]
  let
  (res1, res2) = in-parallel( ((l2 = splitMid x)/2)
                              ((l2 = splitMid x)/2))
    res1 + res2
  end.
```

CANNOT FIND IN LECTURE 2

```
Sum x =
  if x = 1 then
    x
  end.
else
```

3. **SPARC to Python**

Consider the following SPARC code:

```
foo x =
  if x ≤ 1 then
    x
  else
    let (ra, rb) = (foo (x − 1)) , (foo (x − 2)) in
      ra + rb
    end.
```

- 3a. (4 pts) Translate this to Python code – fill in the `def foo` method in `main.py`
- 3b. (2 pts) What does this function do, in your own words?

```python
def foo(x):
    if x <= 1:
        return x
    else:
        ra = foo(x-1)
        rb = foo(x-2)
        return ra+rb
```

This function checks to see if an input is less than or equal to 1: if it is, it returns that value. Otherwise, it returns the sum of the value decremented by 1 and decremented by 2.

4. 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`

- 4a. (8 pts) First, implement an iterative, sequential version of `longest_run` in `main.py`.
- 4b. (4 pts) What is the Work and Span of this implementation?

```
def longest_run(myarray, key):
    lst = []
    for i in myarray:
        ind = 1
        cnt = 0
        while True:
            if i != len(myarray) and myarray[i] == myarray[i+1]:
                cnt += 1
                ind += 1
                lst.append(cnt)
    return max(lst)
```

Span = $O(n^2)$

Work = $O(n)$

- 4c. (8 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`. — ?
- 4d. (4 pts) What is the Work and Span of this sequential algorithm?

```
def longest_run(nums):
    maxCnt = 0
    cnt = 1
    for i in range(len(nums)):
        if nums[i] != nums[i-1] and cnt > maxCnt:
            cnt = maxCnt
            print(maxCnt)
        else:
            cnt = cnt + 1

elif len(nums > 0)
    if cnt > maxCnt:
        cnt = maxCnt
    print(maxCnt)

def main()
    longest_run[nums]
main():
```

Work = $O(n \log n)$

Span = $O(n \log n)$

Input list here.

- (3 pts) 4e. 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?

Span = $O(\log n)$

Work = $O(n \log n)$