CMPS 2200 Problem Set 1

In this assignment, you will learn more about asymptotic notation, parallelism, functional languages, and algorithmic cost models.

As with the recitations, your code implementations will go in main.py. Please add your written answers to answers.md which you can convert to a PDF using convert.sh. Alternatively, you may scan and upload written answers to a file names answers.pdf.

1. **Asymptotic notation**

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

  Enter answers in answers.md .
  .
  .
  .
  .

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

  Enter answers in answers.md .
  . No because 2^2^n does not decompose into 2^n

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

  Enter answers in answers.md .
  . no because n^1.01 is effecively O(n) which is not the same as the right hand side.
  .
  .

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

  Enter answers in answers.md .
  . n^1.01 ∈ Ω(log(^2)n) is False. Because Ω(log(2)n) describes the lower bound on the growth rate of a function, but the growth rate of n^1.01 is much faster than log(2)n for n > 1.
  .
  .

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

  . √n ∈ O(log(^3)n) is True. Because The growth rate of √n is much slower than log(3)n, so it does not belong to the set O(log(3)n).
  .

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

  Enter answers in answers.md .

√n ∈ Ω(log(^3)n) is False. Because The definition of Ω(log(3)n) is a set of functions that grow no slower than log(3)n, and √n is one of them.

.

.

.

- 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.

Enter answers in answers.md .

. To prove: o(g(n)) ∩ ω(g(n)) is the empty set
Proof: For this, we need to show that it is not possible for a function h(n) to belong to both o(g(n)) and ω(g(n)) at the same time.
Definition of o(g(n)) and ω(g(n)) :
o(g(n)): For every positive constant c, there exists a constant n0 such that g(n) ≤ c * f(n) for all n ≥ n0.
ω(g(n)): For every positive constant c, there exists a constant n0 such that g(n) ≥ c * f(n) for all n ≥ n0.
Step 1: Assume h(n) ∈ o(g(n)) and h(n) ∈ ω(g(n))

h(n) ∈ o(g(n)): There exists a constant n1 such that h(n) ≤ c1 * g(n) for all n ≥n1.
h(n) ∈ ω(g(n)): There exists a constant n2 such that h(n) ≥ c2 * g(n) for all n≥n2.

Step 2: contradiction: Pick two positive constants c1 and c2 such that c1 < c2.
Let n0 = max(n1, n2). For all n ≥ n0, h(n) satisfies both h(n) ≤ c1 * g(n) and h(n) ≥ c2 * g(n).
This means c1 * g(n) ≤ h(n) ≤ c2 * g(n) for all n ≥ n0,
which contradicts either o(g(n)) or ω(g(n)).

Step 3: Since h(n) cannot simultaneously satisfy both o(g(n)) and ω(g(n)), we conclude that o(g(n)) ∩ ω(g(n)) = ∅

2. **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.
```

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

  This function is an implementation of the Fibonacci sequence, where it returns the nth number in the sequence given an input x. The Fibonacci sequence is defined such that each number is the

sum of the two preceding ones, starting from 0 and 1. If x is less than or equal to 1, the function returns x itself. If not, it recursively calls foo with x - 1 and x - 2, calculates the sum of the returned values and returns it.

.
.
.
.
.

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. First, implement an iterative, sequential version of longest_run in main.py.
- 3b. What is the Work and Span of this implementation?

Enter answers in answers.md

. The Work of this implementation is proportional to the length of the myarray list, as the loop runs once for each element in the list. The Span of this implementation is 1, as it requires only a single iteration over the list.

- 3c. 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. What is the Work and Span of this recursive algorithm?

- 3e. 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?

Enter answers in answers.md .

.
.
.
.
.
.
.