

# CMPS 6610 Problem Set 1

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

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 named `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` .

.  
.  
.  
.

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

Enter answers in `answers.md` .

.  
.  
.

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

Enter answers in `answers.md` .

.  
.  
.

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

Enter answers in `answers.md` .

.  
.  
.

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

Enter answers in `answers.md` .

.  
.  
.

- 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 .**

.  
.  
.  
.  
.  
.  
.  
.  
.  
.  
.  
.

## 2. SPARC to Python

Consider the following SPARC specification:

```
foo a b =
  if a = 0 then
    b
  elseif b = 0 then
    a
  else
    let (x,y) = (min(a,b) , (max(a,b)) in
      (foo y , y mod x)
    end.
```

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

**Enter answers in answers.md .**

.  
.  
.

- 2c. What is the work and span of `foo`?

**Enter answers in answers.md .**

.  
.  
.  
.  
.

## 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`**

.

.

.

.

.

.

.

.

.

- 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?

**Enter answers in `answers.md`**

.

.

.

.

.

.

.

.

.

- 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`**

.

.

.

.

.

.

.