CMPS 6610 Problem Set 01 ## Answers

Name: Md. Ahsan Habib

Place all written answers from `assignment-01.md` here for easier grading.

1. **Asymptotic notation**

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

    $O(f(n)) = \{g(n) \mid f(n) \text{ asymptotically dominates } g(n)\}$

    To determine if 2^n asymptotically dominates 2^{n+1}, we need to find constants $c > 0$ and n_0 such that:

    $$2^{n+1} \leq c \cdot 2^n \quad \text{for all } n \geq n_0$$

    Dividing both sides by 2^n, we get:

    $$2 \leq c$$

    So, any $c \geq 2$ and any $n_0 \geq 0$ will work. For example, if we set $c = 2$ and $n_0 = 0$, the inequality holds for all $n \geq 0$:

    $$2^{n+1} \leq 2 \cdot 2^n$$

    Thus, $2^{n+1} \in O(2^n)$.

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

    $O(f(n)) = \{g(n) \mid f(n) \text{ asymptotically dominates } g(n)\}$

    To determine if $2^{2^n} \in O(2^n)$, we need to check if there exist constants $c > 0$ and n_0 such that:

    $$2^{2^n} \leq c \cdot 2^n \quad \text{for all } n \geq n_0$$

    However, 2^{2^n} grows much faster than 2^n. In fact, for large n, $2^{2^n} \gg 2^n$.

    Taking logarithms of both sides:

    $$\log_2(2^{2^n}) = 2^n \quad \text{and} \quad \log_2(c \cdot 2^n) = \log_2 c + n$$

    For large n, 2^n grows much faster than n, so the inequality cannot be satisfied for any constant c.

    Therefore, $2^{2^n} \notin O(2^n)$.

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

    $O(f(n)) = \{g(n) \mid f(n) \text{ asymptotically dominates } g(n)\}$

    To determine if $n^{1.01} \in O((\log n)^2)$, we need to check if there exist constants $c > 0$ and n_0 such that:

    $$n^{1.01} \leq c \cdot (\log n)^2 \quad \text{for all } n \geq n_0$$

However, for large n, any positive power of n grows much faster than any power of $\log n$. In particular, $n^{1.01} \gg (\log n)^2$ as $n \to \infty$.

Taking logarithms of both sides:

$$\log(n^{1.01}) = 1.01 \log n \quad \text{and} \quad \log(c(\log n)^2) = \log c + 2 \log\log n$$

For large n, $1.01 \log n$ grows much faster than $2 \log\log n$, so the inequality cannot be satisfied for any constant c.

Therefore, $n^{1.01} \notin O((\log n)^2)$.

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

    $\Omega(f(n)) = \{g(n) \mid g(n) \text{ asymptotically dominates } f(n)\}$

    To determine if $n^{1.01} \in \Omega((\log n)^2)$, we need to check if there exist constants $c > 0$ and n_0 such that:

    $$n^{1.01} \geq c \cdot (\log n)^2 \quad \text{for all } n \geq n_0$$

    For large n, any positive power of n grows much faster than any power of $\log n$. In particular, $n^{1.01} \gg (\log n)^2$ as $n \to \infty$.

    Taking logarithms of both sides:

    $$\log(n^{1.01}) = 1.01 \log n \quad \text{and} \quad \log(c(\log n)^2) = \log c + 2 \log\log n$$

    For large n, $1.01 \log n$ grows much faster than $2 \log\log n$, so the inequality is satisfied for sufficiently large n and some $c > 0$.

    Therefore, $n^{1.01} \in \Omega((\log n)^2)$.

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

    $O(f(n)) = \{g(n) \mid f(n) \text{ asymptotically dominates } g(n)\}$

    To determine if $\sqrt{n} \in O((\log n)^3)$, we need to check if there exist constants $c > 0$ and n_0 such that:

    $$\sqrt{n} \leq c \cdot (\log n)^3 \quad \text{for all } n \geq n_0$$

    For large n, any positive power of n grows much faster than any power of $\log n$. In particular, $\sqrt{n} = n^{1/2} \gg (\log n)^3$ as $n \to \infty$.

    Taking logarithms of both sides:

    $$\log(\sqrt{n}) = \frac{1}{2} \log n \quad \text{and} \quad \log(c(\log n)^3) = \log c + 3 \log\log n$$

    For large n, $\frac{1}{2} \log n$ grows much faster than $3 \log\log n$, so the inequality cannot be satisfied for any constant c.

    Therefore, $\sqrt{n} \notin O((\log n)^3)$.

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

- ***Answer***

  $\Omega(f(n)) = \{g(n) \mid \exists\ c > 0,\ n_0 \text{ such that } g(n) \geq c \cdot f(n) \text{ for all } n \geq n_0\}$

  To determine if $\sqrt{n} \in \Omega((\log n)^3)$, we need to check if there exist constants $c > 0$ and n_0 such that:

  $$\sqrt{n} \geq c \cdot (\log n)^3 \quad \text{for all } n \geq n_0$$

  For large n, any positive power of n grows much faster than any power of $\log n$. In particular, $\sqrt{n} = n^{1/2} \gg (\log n)^3$ as $n \to \infty$.

  Taking logarithms of both sides:

  $$\log(\sqrt{n}) = \frac{1}{2} \log n \quad \text{and} \quad \log(c(\log n)^3) = \log c + 3 \log \log n$$

  For large n, $\frac{1}{2} \log n$ grows much faster than $3 \log \log n$, so the inequality is satisfied for sufficiently large n and some $c > 0$.

  Therefore, $\sqrt{n} \in \Omega((\log n)^3)$.

- 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 $g(n) < c \cdot f(n)$ 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.

- ***Proof***
- $f(n) \in o(g(n))$ means that for **every** $c > 0$, there exists n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.
  - $f(n) \in \omega(g(n))$ means that for **every** $c > 0$, there exists n_0 such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.

  Suppose, for contradiction, that there exists a function $f(n)$ such that $f(n) \in o(g(n)) \cap \omega(g(n))$.

  Then, by definition:

  - For every $c > 0$, $f(n) < c \cdot g(n)$ for all sufficiently large n.
  - For every $c > 0$, $f(n) > c \cdot g(n)$ for all sufficiently large n.

  Take $c = 1$. Then for large n,

  $$f(n) < g(n) \quad \text{and} \quad f(n) > g(n)$$

  So $f(n) = g(n)$ cannot be possible.

  Therefore, no function can be in both $o(g(n))$ and $\omega(g(n))$, so:

  $$o(g(n)) \cap \omega(g(n)) = \emptyset$$

2. **SPARC to Python**

- 2b. What does the *foo* function do, in your own words?
  - ***Answer***

    The *foo* function is simply giving the maximum value of those given two numbers.

    Note: However, if we put the min value (here x) when we can the *foo* recursively, it will work as a *GCD* function.

- 2c. what is the work and span of *foo*?

- ***Answer***
    - Work:
      The work is total number of recursive calls of foo function. In the worst case, the number of recursive calls is $O(logn)$, where n = max(a, b). Each call (min, max, mod, and comparison) does constant c work.
      Formally, we can write it like:
        - Base: $W(0) = c$
        - Step: $W(n) = W(n \bmod m) + c$

      So work = $O(logn)$.
    - Span: The span (critical path length) is the longest sequence of dependent computations. Since each recursive call depends on the result of the previous call, the calls are sequential, not parallel. Formally, we can write it like:
        - Base: $S(0) = c$
        - Step: $S(n) = S(n \bmod m) + c$

      So, span = $O(logn)$.

3. **Parallelism and recursion**

- 3b. What is the Work and Span of the iterative *longest_run* algorithm?
  - ***Answer***
    - Work: The work is total number of iterations throughout the entire list. So it should be O(n) for n length of *mylist*. Formally, we can write it like:
        - $W(n) = W(n-1) + 1$

      So the work should be $O(n)$.
    - Span: As it is purely sequential, so span is also $O(n)$.
- 3d. What is the Work and Span of the iterative *longest_run_recursive* algorithm?
  - ***Answer***
    - Work: Every time we split the list into two parts until there is a single item and then we combine them with some additional operations. So, for n length of *mylist*, we can write:
        - $W(n) = 2 \cdot W(n/2) + c$

      So, the work should be $O(n)$.
    - Span: Here, the recursive calls are independent, so can run in parallel. We can write:
        - $S(n) = S(n/2) + c$

      So, the span should be $O(logn)$.
- 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?
  - ***Answer***
    - By parallelizing the function, I think the work and span should be the same as 3d. As per definition of work, we only have one processor to complete the task, so parallelization doesn't benifit here. On the other hand, the span is calculated while considering the as many processor as it requires by definition. So, both are same. That means,
        - $W(n) = 2 \cdot W(n/2) + c = O(n)$
        - $S(n) = S(n/2) + c = O(logn)$