CMPS 6610 Problem Set 01 ## Answers

Name:___ Aaron Dumont___________________________________

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

1. **Asymptotic notation**

- 1a **TRUE. Yes**, $2^{n+1} \in O(2^n)$.

  Reasoning: Per Big O ntation, we must show that there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

  Let $f(n) = 2^{n+1}$ and $g(n) = 2^n$.

  $f(n) = 2^{n+1} = 2^1 \cdot 2^n = 2 \cdot 2^n$.

  $2 \cdot 2^n \leq c \cdot 2^n$.

  Divide both sides by 2^n, we get $2 \leq c$.

  We can choose $c \geq 2$ and this inequailty holds for all $n \geq 0$ (so we can set $n_0 = 0$).

  Since we found constants $c \geq 2$ and $n_0 = 0$ that satisfy the definition, 2^{n+1} is in $O(2^n)$.

- 1b **FALSE. No**, $2^{2^n} \in O(2^n)$.

  Reasoning: In class we have demonstrated that any polylogarithmic function grows slower than any polynomial function. Expressed differently, $log^i(n) \in O(n^j)\ \forall\ i, j > 0$. Conversely $n^j \in \Omega(log^i(n))\ \forall\ i, j > 0$

  Let $f(n) = 2^{2^n}$ and $g(n) = 2^n$.

  Inequality: $2^{2^n} \leq c \cdot 2^n$. We must check if there is a constant c that satisfies this.

  Let $a = 2^n$

  Then $2^a \leq c \cdot a$.

  Take log_2 of both sides.

  $a \leq log_2 c + log_2 a$

  Therefore there is no c that satisfies this inequality as the left polynomial which grows lineraly always grows faster than the right polylogarithmic function.

- 1c **FALSE. No**, $n^{1.01} \in O(log^2 n)$.

  Reasoning: As above, any polylogarithmic function grows slower than any polynomial function. Expressed differently, $log^i(n) \in O(n^j)\ \forall\ i, j > 0$. Conversely $n^j \in \Omega(log^i(n))\ \forall\ i, j > 0$. The polynomial function $n^{1.01}$ grows much faster than a constant multiple of $logn$ (in this case $log^2 n$).

- 1d **TRUE. Yes**, $n^{1.01} \in \Omega(log^2 n)$.

  Reasoning: As above, as we have shown in class, $n^j \in \Omega(log^i(n))\ \forall\ i, j > 0$.Therefore, the statement is true.

- 1e **FALSE. No**, $\sqrt{n} \in O(log^3 n)$.

  Reasoning: As above, any polylogarithmic function grows slower than any polynomial function. Expressed differently, $log^i(n) \in O(n^j)\ \forall\ i, j > 0$. Conversely $n^j \in \Omega(log^i(n))\ \forall\ i, j > 0$. The polynomial function $\sqrt{n}$ which is $n^{0.5}$ grows faster than a constant multiple of $logn$ (in this case $log^3 n$).

- 1f **TRUE. Yes**, $\sqrt{n} \in \Omega(log^3 n)$.

  Reasoning: As above,as we have shown in class, $n^j \in \Omega(log^i(n))\ \forall\ i, j > 0$.Therefore, the statement is true. This is a statement of lower bound so it is true.

- 1g We will assume that there exists a function, $f(n)$, in the intersection of $o(g(n))$ and $\omega(g(n))$. By definition:

  1. $f(n) \in o(g(n))$, for any constant $c_1 > 0$ and there is an n_1 such that $f(n) < c_1 g(n)$ for all $n \geq n_1$.
  2. $f(n) \in \omega(g(n))$, for any constant $c2 > 0$ and there is an n_2 such that $c_2 g(n) < f(n)$ for all $n \geq n_2$.

  The above must hold for every positive constant c. So let $c_1 = c_2$. We can also select $n_0 = max(n_1, n_2)$. By selecting $n_0 = max(n_1, n_2)$, we establish a threshold. For any $n \geq n_0$, we are guaranteed that $n \geq n_1$ AND $n \geq n_2$, meaning both inequalities must hold. Therfore:

  $c_2 g(n) < f(n) < c_1 g(n)$. $c_1 = c_2$. Therefore, $c_1 g(n) < f(n) < c_1 g(n)$. Since this is a strict inequality definition, there is no $f(n)$ that exists to satisfy this strict inequality.

2. **SPARC to Python**

- 2a - See main.py
- 2b - This function takes two numbers as input, checks if the base case is true and than recursively returns the maximum of the two numbers and the remainder of y maximum divided by the mininum number. The recursion only works on the second argument, which is repeatedly replaced by y % (current_second_arg). Eventually, the second argument will become 0, and the function will return the first argument, which has been max(a, b) all along. In my own words, the function computes the maximum of two non-negative integers. For any inputs a and b, foo(a, b) will return max(a, b).

Note: If the function in SPARC had (foo x, y mod x) instead of (foo y, y mod x), this would be the Euclidean algorithm and would return the greatest common divisor (GCD). The function foo(a, b) calculates the Greatest Common Divisor (GCD) of two non-negative integers a and b under this modified algorithm.

- 2c - **Analysis**:

Algorithm: $y = max(a, b)$ is constant. $foo(y, b_i)$ calls $foo(y, y\%b_i)$.

In each recursive call, a constant number of operations is performed: 2 comparisons, a min, a max, a mod, and the function call itself. This is constant work O(1). In terms of the number of recursive calls, the second argument gets smaller with each step until it becomes 0.The sequence if $b_0 = \min(a, b)$, $b_1 = y\%b_0$, $b_2 = y % b_1, etc. The number of steps is logarithmic in the value of the inputs ie. $O(log(min(a, b)))$.

Work: The total number of operations, equivalent to the time it would take on a single processor. Work = work per step * # of steps.

$W(a, b) = O(1) * O(log(min(a, b))) = O(log(min(a, b)))$

Span: Span (or depth) is the longest chain of dependent operations, which represents the execution time on an infinite number of processors. This is a completely sequential algorithm as each call depends on the result of the next one. So the Span will be the same as work.

$S(a, b) = O(1) * O(log(min(a, b))) = O(log(min(a, b)))$

3. **Parallelism and recursion**

- 3a - See longest_run in main.py
- 3b - The work and span of this sequential algorithm: The work of an algorithm is the total number of operations it performs. In the for loop, we iterate through the list (all n elements of the array) and inside the loop it performs a constant number of operations (a comparison, an addition, a max operation, an assignment). Therefore total work is linear and proportional to n.

  $W(n) = O(n)$

  The span is the longest chain of dependent operations, representing the runtime with infinite processors. Our implementation in this algorithm is sequential (no parallelism). Therefore, span is also proportional to n.

  $S(n) = O(n)

- 3d

  n is the length of mylist.

  Work: Work is the total number of operations. This function splits the problem into 2 subproblems of size $n/2$ and then does a constant amount of work ($O(1)$) to combine them.

  Therefore $W(n) = 2W(n/2) + O(1)$.

  Can use brick method that is leaf dominated. n= s^h so $log_2(n) = log_2(2^h)$ and $h = log_2(n)$. Number of leaves = 2^h which is $2^(log_2(n))$ which is n. This simplifies to $W(n) \in O(n)$.

  Span: Span is the longest dependency path. As a result of the two recursive calls, longest_run_recursive(left_half, ...) and longest_run_recursive(right_half, ...) are still being run **sequentially**. Therefore the Span and Work are the same.

  $S(n) = 2S(n/2) + O(1)$. This simplifies to $S(n) \in O(n)$.

- 3e **Work**: Work is the total number of operations. This function splits the problem into 2 subproblems of size $n/2$ and then does a constant amount of work ($O(1)$) to combine them. The work **remains the same**.

  Therefore $W(n) = 2W(n/2) + O(1)$. This simplifies to $W(n) \in O(n)$ as reasoned above.

  Span: Span is the longest dependency path. As a result of the two recursive calls, longest_run_recursive(left_half, ...) and longest_run_recursive(right_half, ...) are now being run **in parallel**. Therefore the Span is the height of the tree as the algorithm is operating in parallel. We can use the brick method that is balanced with the first level having a cost of 1 and the number of leaves equal to h.

  $S(n) = S(n/2) + O(1)$. We have shown above that the height of the tree, h, is $log_2(n)$ and the first level has a cost of 1. This simplifies to $S(n) \in O(logn)$.

4. **GCD**