Week 5:

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. By the Assessed Cohort Meeting, you and all of your cohortmates, should be prepared to present and discuss solutions to all of the assigned problems. In addition to discussing with your cohortmates, you may discuss the problems with any other current CS3102 students you want, and use any resources you want except for any materials from previous offerings of this course or complete solutions that might be available on the web, which are not permitted. **At the end of your assessed cohort meeting, your Cohort Coach will assign one of these problems as a writeup. You may not collaborate on this writeup with your cohort-mates, but you may use notes taken before or during your assessed cohort meeting.**

Problem 1: Cohort Communication

Please discuss any questions or concerns that came up as a result of your peer evals. If you have something you would like to discuss with your cohort, but are not sure how to mention it, Professor Brunelle would be happy to help you make a plan. Just show up to office hours, or send him a DM to schedule a time to chat.

Problem 2: Complexity by Circuit Depth

In the *Complexity Classes: SIZE* video we defined the complexity $SIZE(s)$ to be the set of all functions which can be implemented as a NAND circuit containing s or fewer gates. We defined $SIZE^{AON}(s)$ to be the set of all functions which can be implemented as an AON circuit containing s or fewer gates.

Once a circuit has been implemented in hardware, all gates at the same level will evaluate in parallel with one another, but each gate must "wait" on gates at shallower levels before it can be evaluated. The *depth* of a circuit is defined as the maximum number of gates along a path from an input to output. For this reason, circuit depth is actually a better metric for estimating running time than circuit size would be.

For this problem we will measure complexity by circuit depth rather than by number of gates. These two definitions define sets of complexity classes based on circuit depth:

Definition 1 ($DEPTH^{NAND}$) *A function $f : \{0,1\}^n \to \{0,1\}^m$ belongs to class $DEPTH^{NAND}(d)$ if it can be implemented as a NAND circuit with depth d or less.*

Definition 2 ($DEPTH^{AON}$) *A function $f : \{0,1\}^n \to \{0,1\}^m$ belongs to class $DEPTH(d)$ if it can be implemented as an AON circuit with depth d or less.*

Answer the following using these complexity classes:

(a) What is the smallest natural number d for which `OR` $: \{0,1\}^2 \to \{0,1\}$ will be in class $DEPTH^{NAND}(d)$

(b) What is the smallest natural number d for which `AND` $: \{0,1\}^2 \to \{0,1\}$ will be in class $DEPTH^{NAND}(d)$

(c) What is the smallest natural number d for which `NOT` $: \{0,1\} \to \{0,1\}$ will be in class $DEPTH^{NAND}(d)$

(d) What is the smallest natural number d for which `NAND` $: \{0,1\}^2 \to \{0,1\}$ will be in class $DEPTH^{AON}(d)$

(e) In the *First Complexity Proof lecture* we showed $SIZE(\frac{s}{2}) \subseteq SIZE^{AON}(s) \subseteq SIZE(3s)$. Use your answers above to perform a similar argument for $DEPTH$ by identifying functions f and g that will allow you to show that $DEPTH^{NAND}(f(d)) \subseteq DEPTH^{AON}(d) \subseteq DEPTH^{NAND}(g(d))$.

Problem 3: Implementations are not unique

Show that for any function $f : \{0,1\}^n \to \{0,1\}$ there are an infinite number of NAND circuits which implement that function.

With this proof in mind, explain why we define the complexity of a function in terms of its most efficient implementation.

Problem 4: Asymptotic Operators

For each sub-problem, indicate if the statement is *true* or *false* and support your answer with a convincing argument.

(a) $17n \in O(723n + \log n)$

(b) $\min(n^n, 3012) \in O(1)$

(c) $n^2 \in \Theta(n^3)$

(d) $2.0001^n \in O(2^n)$

(e) $(\frac{\star}{2}) \log_n 10 \in \Theta(\log_{2n} 17)$

Problem 5: Constant "time"

Show whether $O(1) = \Theta(1)$.

It is left up to you to determine if the two sets are equivalent (the proposition may be either true or false), and provide a convincing proof to support your answer.

Problem 6: little-o

Another useful notation is "little-o" which is designed to capture the notion that a function g grows much faster than f:

Definition 3 (o) *A function $f(n) : \mathbb{N} \to \mathbb{R}$ is in $o(g(n))$ for any function $g(n) : \mathbb{N} \to \mathbb{R}$ if and only if for every positive constant c, there exists an $n_0 \in \mathbb{N}$ such that:*

$$\forall n > n_0 . f(n) < cg(n).$$

In other words, $f(n) \in o(g(n))$ provided that no matter what positive constant that is chosen for c, eventually there comes a point were $c \cdot g(n) > f(n)$ forever more.

Provide a proof for each of the following sub-problems.

(a) Prove that for any function f, $f \notin o(f)$.

(b) Prove that $n \in o(n \log n)$.

Problem 7: Actually Linear Time

(This problem is related to the *Common Misuses of Asymptotic Notation* video.)

Typically when evaluating the running time of programs we will count the number of times a particular operation might be performed. For example, in your algorithms class you might say that this program runs in linear time $\Theta(x)$, where x is the value of the input `x`, by counting the number of additions performed and assuming they all take constant time:

```
def mult(x, y):
    product = 0
    for _ in range(x):
        product += y
    return y
```

If we wanted to be more formal and determine the number of total clock cycles your CPU might perform relative to the number of bits given as input, we would see that the running time of this program is not linear. There are several reasons for this, but we'll describe two of them here.

First, running "`product += y`" does not take the same amount of time for every value of `y`. Once our values get large enough, that sum will begin to take more time to compute as the values get larger (note that unlike Java, Python does not have an arbitrary low bound on the maximum value of an integer, but can scale integers to use multiple words of memory).

Second, saying that the running time is linear is the *value* of `y` does not mean it is linear in the number of bits given as input. For a number with value `y`, the number of bits required to represent it is $\lceil \log_2 y \rceil$, so the implementation requires a number of addition operations that is exponential in the size (number of *bits*) of the input `y`.

For this problem, your goal is to write a Python program whose running time is as close as possible to $\Theta(n)$ where n represents the size of its input. You may want to use the Python *timeit* module to time your program. If it has running time in $\Theta(n)$, you should be able to find a constant c such that the running time of your program is approximately cn milliseconds for all inputs of size n, about some n_0.

