Week 3: Sugar Cookies

These problems are designed to help you become more comfortable with the Circuit Model of computing.

Collaboration: You should work on the problems yourself, before discussing with others, and with your cohorts are 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: Circuit Cycles

Why can't our AON-circuits have cycles in them (e.g. gate A connects to gate B which connects to gate C which connects to gate A)? Specifically, what about our Boolean circuits definition or execution rules would break?

Problem 2: Maximum Number of Inputs (Induction Practice)

The *depth* of a circuit is the length of the longest path (in the number of gates) from the an input to an output in the circuit. Prove using induction that the maximum number of inputs for a Boolean circuit (as defined by Definition 3.5 in the book) that produces one output that depends on all of its inputs with depth d is 2^d for all $d \geq 0$. (Note: there are ways to prove this without using induction, but the purpose of this problem is to provide induction practice, so only solutions that are well constructed proofs using the induction principle will be worth full credit.)

Problem 3: Why `LOOKUP_k`?

When discussing LOOKUP, we had to parameterize the function. That is, we did not just talk about `LOOKUP`, but defined `LOOKUP_k` (subscript `k`), where `k` referred to the number of bits in the index. This means that `LOOKUP_1` is a different function from `LOOKUP_2`, which is different from `LOOKUP_3`, etc.

Why did we need different versions of `LOOKUP` for different-sized arrays/indexes? Why couldn't we just have one `LOOKUP` which took in `k` as a parameter?

Problem 4: Cleaning Up Nate's Messes

In the lecture video *Cost of LOOKUP*, Professor Brunelle set up a proof by induction that `LOOKUP_k` requires $\leq 4 \cdot 2^k$ `NAND` gates to compute. His proof did not actually work, though, and his calls for help in the video were sadly unanswered.

Let's help him out. Write your own inductive proof to show that `LOOKUP_k` requires $\leq 4 \cdot 2^k$ `NAND` gates. (**Hint:** It may be easier to show that the number of gates required is upper-bounded by some other function that is itself upper-bounded by $4 \cdot 2^k$).

Problem 5: Straightline `IF` vs Python `if`

When we added the syntactic sugar `IF` to the AON or NAND Straightline programming language, we had to do a peculiar thing. With a Python `if`, code is either executed or not depending on the Boolean condition. When using `IF` within AON/NAND Straightline, the True "branch" and False "branch" were *both* always executed, and `IF` merely selected one of the two values to assign to a variable.

Why can't we "skip" code in straightline programs? Why doesn't Python execute `if` in a way similar to straightline (a helpful answer would include an example of a use of `if` in Python that would behave

differently with our `IF` definition)? Why is the `IF` function we defined okay for straightline programs but not for Python?

Problem 6: Full Adders (based on exercise 4.5)

First, complete the programming problems. They are necessary in helping you to address the written problem below.

Programming Problems

See *adders.py* for the programming problems.

Written Problem

Show that for every n there is a NAND-CIRC program to compute ADD_n with at most $9n$ lines where $ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ is the function that outputs the sum of two input n-bit numbers (where all inputs and outputs are represented in binary).

You may find the other parts of Exercise 4.5 in the book helpful, but it is not necessary to solve this problem using those steps.

Problem 7: *XOR* Circuits

Prove that the gate set *XOR*, $0, 1$ is *weaker* than *AND, OR, NOT*. In other words, show there is at least one function that can be computed by circuits using gates from *AND, OR, NOT* that cannot be computed with gates from *XOR*, $0, 1$. (You can use any strategy you want to prove this; see the book for one hint of a possible strategy, but we think you may be able to find easier ways to prove this, and it is not necessary to follow the strategy given in the book.)

