Week 3: Sweet

The purpose of this assignment is to develop your understanding of finite computation, focusing on the material in Chapter 3 of the textbook and what we covered in the Week 3 Videos.

We have also included some problems to provide more practice with proof techniques including induction, since it is apparent from Problem Set 1 that most students will benefit from these.

Collaboration Policy: 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 (including the programming problems). In addition to discussing with your cohortmates, you may discuss the problems with anyone you want, and use any resources you want except for any materials from previous offerings of this course, which are not permitted.

Problem 1 *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 *Fixing Brunelle's Mistakes*

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 *Universality Checkup*

Prove that $\{\texttt{MAJ}, \texttt{0}, \texttt{1}\}$ is not a universal gate set (where `MAJ` is the majority of three inputs function, and `0` and `1` are constants).

Problem 7 *How* universal *is "Universal"?*

We introduced in lecture that some sets of gates are "Universal". For example, the set $\{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$ is universal, as is the set $\{\texttt{NAND}\}$. We've also proven that no finite model of computing can implement all Boolean functions. If $\{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$ being Universal doesn't mean it can be used to compute *any* function, what precisely do we mean by "Universal"?

Problem 8 *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.

> **Starred Problems:** The remaining problems on this problem set are "starred" challenge problems. When a problem is marked with a $\star$, it means we think this problem is challenging enough that students are not expected to be able to solve it. We still hope everyone will attempt these problems and learn from trying to solve them, but you shouldn't get overly frustrated if you are not able to solve a $\star$ problem. Unlike the non-starred problems, you will not be "cold-called" to present a solution to a $\star$-level problem at an Assessed Cohort Meeting. Instead, if there is time in the meeting, students will have an opportunity to volunteer to discuss the problem (and receive potential bonus points for outstanding solutions).

Problem 9 ($\star$) *Perceptron*

A *perceptron* is a single layer neural network (Section 3.4.5) that can be modeled by the following function:

$$f(x_0, x_1, \ldots, x_{k-1}) = \sigma(\sum_i w_i x_i)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is an activation function. For this question, you may assume the activation function is a rectified linear unit (ReLU), commonly used in deep learning:

$$ReLU(x) = \max(0, x)$$

Prove that there is no way to define *XOR* using a perceptron. That is, show that there is no way to assign the values of w_i such that $f(x_0, x_1) = ReLU(w_0 x_0 + w_1 x_1)$ implements the *XOR* function. You can interpret the output of f ias a Boolean value with values below 0.5 interpreted as False and values ≥ 0.5 interpreted as True.

Historical and resource policy note: The proof that a perceptron cannot compute *XOR* is of some historical importance, and it doesn't take much cleverness to *find proofs of this* (don't click on this link until after attempting this problem on your own). You should not search for solutions to this problem since the goal of it is for you to think about this yourself and come up with a proof. The historical significance of this problem, which is often overblown to the point where some refer to it as the "XOR affair", is that it has been attributed by some as one of the reasons why research in neural networks mostly ceased in the 1980s, except for a few die-hard believers who kept working on it, eventually leading to the explosion of "deep learning" over the past decade, and being awarded the Turing Award in 2018.

Problem 10 ($\star\star$) *Prove that a two-layer perceptron is universal.*

This will require a bit of creativity and thinking carefully about our definitions. As in Problems 4 and 5, *universal* means that a Boolean circuit where the gates are all two-layer perceptrons can compute the same function as any Boolean circuit.

