Staff Solutions to Problem Set 1

Reading:

- Chapter 1. *What is a Proof?*
- Chapter 2. *The Well Ordering Principle* through 2.3; (omit 2.4. *Well Ordered Sets*).
- Chapter 3. *Logical Formulas* through 3.3, and 3.5; (optional: 3.4.)

These assigned readings do not include the Problem sections. (Many of the problems in the text will appear as class or homework problems.)

Reminder:

- Pset performance does not count toward the final grade, but the exams will be based heavily on psets. Devoting up to three hours working on the pset is important for learning the material.
- Problems should be submitted electronically by 11AM on the due date following the instructions on the class Stellar website. Graders will provide comments and grades (for feedback, not credit) on problems that are submitted on time.
- The class has a Piazza forum. With Piazza you may post questions—both administrative and content related—to the entire class or to just the staff. You are likely to get faster response through Piazza than from direct email to staff.

STAFF NOTE: Lectures covered: Intro, Proof by Contradiction, WOP, Propositional Formulas

Problem 1.
Prove that $\log_4 6$ is irrational.

Solution. *Proof.* Suppose to the contrary that $\log_4 6$ is rational, so $\log_4 6 = m/n$ for some positive integers $m$ and $n$. So $n \log_4 6 = m$.

Now raising 4 to each side of this equation gives

$$4^n \log_4 6 = 4^m,$$
$$4^{(\log_4 6) n} = 4^m,$$
$$6^n = 4^m$$

But the left hand side of (1) is divisible by 3 (since $n > 0$), and the right hand side is not. This contradiction implies that $\log_4 6$ must be irrational.
Problem 2.
Use the Well Ordering Principle to prove that

\[ n \leq 3^{n/3} \]  \hspace{1cm} (2)

for every nonnegative integer, \( n \).

*Hint:* Verify (2) for \( n \leq 4 \) by explicit calculation.

**Solution.** Suppose to the contrary that (2) failed for some nonnegative integer. Then by the WOP, there is a least such nonnegative integer, \( m \).

But \( 0 \leq 3^{0/3} \), so \( m \neq 0 \). Also, \( 1^3 \leq 3^1 \), so taking cube roots, \( 1 \leq 3^{1/3} \), which implies \( m \neq 1 \). Likewise, \( 2^3 \leq 3^2 \), so taking cube roots, \( 2 \leq 3^{2/3} \), which implies \( m \neq 2 \). Similar simple calculations show that \( m \neq 3, 4 \), so we know that \( m \geq 5 \).

Now since \( m > m - 3 \geq 0 \) and \( m \) is the least nonnegative integer for which the inequality (2) fails, the inequality must hold when \( n = m - 3 \). So

\[
3^{m/3} = 3 \cdot 3^{(m-3)/3} \\
\geq 3 \cdot (m - 3) \quad \quad \quad \quad \quad \text{(by (2) for } n = m - 3) \] \hspace{1cm} (3)

Also,

\[
3 \cdot (m - 3) = 3m - 9 \\
> 3m - 2m \\
= m. \quad \quad \quad \quad \quad \quad \quad \quad \text{since } m > 9/2 \] \hspace{1cm} (4)

Combining (3) and (4), we get

\[ m \leq 3^{m/3}, \]

contradicting the assumption that (2) fails for \( n = m \).

This contradiction implies that there cannot be a nonnegative integer for which (2) fails. By the WOP, this means that (2) must hold for all nonnegative integers.

Problem 3. (a) Verify by truth table that

\[(P \text{ implies } Q) \text{ or } (Q \text{ implies } P)\]

is valid.

**Solution.**

<table>
<thead>
<tr>
<th>( P )</th>
<th>( Q )</th>
<th>((P \text{ implies } Q))</th>
<th>((Q \text{ implies } P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

(b) Let \( P \) and \( Q \) be propositional formulas. Describe a single formula, \( R \), using only AND’s, OR’s, NOT’s, and copies of \( P \) and \( Q \), such that \( R \) is valid iff \( P \) and \( Q \) are equivalent.
Solution.

\[ R := (P \land Q) \lor (\lnot P \land \lnot Q) \]

(e) A propositional formula is *satisfiable* iff there is an assignment of truth values to its variables—an *environment*—which makes it true. Explain why

\[ P \text{ is valid iff } \lnot(P) \text{ is not satisfiable.} \]

**Solution.** To prove the iff, we prove that the left hand statement implies the right hand one and vice-versa.

**(left-to-right case):** If \( P \) is valid, then \( \lnot(P) \) is not satisfiable.

**Proof.** Now \( P \) is true in an environment iff \( \lnot(P) \) is false in that environment. Since \( P \) is valid, it is true in every environment, which means that \( \lnot(P) \) is false in every environment. So no environment makes \( \lnot(P) \) true, which means that \( \lnot(P) \) is not satisfiable.

**(right-to-left case):** If \( \lnot(P) \) is not satisfiable, the \( P \) is valid.

**Proof.** Since \( \lnot(P) \) is not satisfiable, every truth assignment makes it false. This implies that every truth assignment makes \( P \) true, that is, \( P \) is valid.

(d) A set of propositional formulas \( P_1, \ldots, P_k \) is *consistent* iff there is an environment in which they are all true. Write a formula, \( S \), so that the set \( P_1, \ldots, P_k \) is not consistent iff \( S \) is valid.

**Solution.** Note that the set \( P_1, \ldots, P_k \) is consistent iff

\[ (P_1 \land P_2 \land \ldots \land P_k) \]

is satisfiable. So by part (e)

\[ S := \lnot(P_1 \land P_2 \land \ldots \land P_k) \]

is the desired formula. In more concise notation this would written

\[ \lnot \left( \bigwedge_{i=1}^{k} P_i \right) \]

or, using DeMorgan’s Law:

\[ \bigvee_{i=1}^{k} \lnot P_i. \]

Problem 4.

As in Problem 3.5, a digital circuit is called an \((n + 1)\)-bit *half-adder* when it has with \( n + 2 \) inputs

\[ a_n, \ldots, a_1, a_0, b \]

and \( n + 2 \) outputs

\[ c, s_n, \ldots, s_1, s_0. \]

The input-output specification of the half-adder is that, if the 0-1 values of inputs \( a_n, \ldots, a_1, a_0 \) are taken to be the \((n + 1)\)-bit binary representation of an integer, \( k \), then the 0-1 values of the outputs \( c, s_n, \ldots, s_1, s_0 \) are supposed to be the \((n + 2)\)-bit binary representation of \( k + b \).
For example suppose $n = 2$ and the values of $a_2 a_1 a_0$ were 101. This is the binary representation of $k = 5$. Now if the value of $b$ was 1, then the output should be the 4-bit representation of $5 + 1 = 6$. Namely, the values of $c s_2 s_1 s_0$ would be 0110.

There are many different circuit designs for half adders. The most straightforward one is the “ripple carry” design described in Problem 3.5. We will now develop a different design for a half-adder circuit called a parallel-design or “look-ahead carry” half-adder. This design works by computing the values of higher-order digits for both a carry of 0 and a carry of 1, in parallel. Then, when the carry from the low-order digits finally arrives, the pre-computed answer can be quickly selected.

We’ll illustrate this idea by working out a parallel design for an $(n + 1)$-bit half-adder.

Parallel-design half-adders are built out of parallel-design circuits called add1-modules. The input-output behavior of an add1-module is just a special case of a half-adder, where instead of an adding an input $b$ to the input, the add1-module always adds 1. That is, an $(n + 1)$-bit add1-module has $(n + 1)$ binary inputs

$$a_n, \ldots, a_1, a_0,$$

and $n + 2$ binary outputs

$$c, p_n, \ldots, p_1, p_0.$$

If $a_n \ldots a_1 a_0$ are taken to be the $(n + 1)$-bit representation of an integer, $k$, then $c p_n \ldots p_1 p_0$ is supposed to be the $(n + 2)$-bit binary representation of $k + 1$.

So a 1-bit add1-module just has input $a_0$ and outputs $c, p_0$ where

$$p_0 := a_0 \text{ XOR 1, (or more simply, } p_0 := \text{ NOT}(a_0)),
\quad c := a_0.$$

In the ripple-carry design, a double-size half-adder with $2(n + 1)$ inputs takes twice as long to produce its output values as an $(n + 1)$-input ripple-carry circuit. With parallel-design add1-modules, a double-size add1-module produces its output values nearly as fast as a single-size add1-modules. To see how this works, suppose the inputs of the double-size module are

$$a_{2n+1}, \ldots, a_1, a_0,$$

and the outputs are

$$c, p_{2n+1}, \ldots, p_1, p_0.$$

We will build the double-size add1-module by having two single-size add1-modules work in parallel. The setup is illustrated in Figure 1.

Namely, the first single-size add1-module handles the first $n + 1$ inputs. The inputs to this module are the low-order $n + 1$ input bits $a_n, \ldots, a_1, a_0$, and its outputs will serve as the first $n + 1$ outputs $p_n, \ldots, p_1, p_0$ of the double-size module. Let $c_{(1)}$ be the remaining carry output from this module.

The inputs to the second single-size module are the higher-order $n + 1$ input bits $a_{2n+1}, \ldots, a_{n+2}, a_{n+1}$. Call its first $n + 1$ outputs $r_n, \ldots, r_1, r_0$ and let $c_{(2)}$ be its carry.

(a) Write a formula for the carry, $c$, of the double-size add1-module solely in terms of carries $c_{(1)}$ and $c_{(2)}$ of the single-size add1-modules.

**Solution.**

$$c = c_{(1)} \text{ AND } c_{(2)}.$$

(b) Complete the specification of the double-size add1-module by writing propositional formulas for the remaining outputs, $p_{n+i}$, for $1 \leq i \leq n + 1$. The formula for $p_{n+i}$ should only involve the variables $a_{n+i}$, $r_{i-1}$, and $c_{(1)}$. 


Solution. The \( n + 1 \) high-order outputs of the double-size module are the same as the inputs if there is no carry from the low-order \( n + 1 \) outputs, and otherwise is the same as the outputs of the second single-size \textit{add1} module. So

\[
p_{n+i} ::= (\text{NOT}(c_{(i)}) \text{ AND } a_{n+i}) \text{ OR } (c_{(i)} \text{ AND } r_{i-1}).
\]  

(5)

for \( 1 \leq i \leq n + 1 \).

(c) Explain how to build an \((n + 1)\)-bit parallel-design half-adder from an \((n + 1)\)-bit add1-module by writing a propositional formula for the half-adder output, \( s_i \), using only the variables \( a_i, p_i, \) and \( b \).

Solution.

\[
s_i ::= (b \text{ AND } p_i) \text{ OR } (\text{NOT}(b) \text{ AND } a_i)
\]

(d) The speed or \textit{latency} of a circuit is determined by the largest number of gates on any path from an input to an output. In an \( n \)-bit ripple carry circuit, there is a path from an input to the final carry output that goes through about \( 2n \) gates. In contrast, parallel half-adders are exponentially faster than ripple-carry half-adders. Confirm this by determining the largest number of propositional operations, that is, gates, on any path from an input to an output of an \( n \)-bit add1-module. (You may assume \( n \) is a power of \( 2 \).)

Solution. In a look-ahead carry circuit, no path from an input to an output has more than \( 4 \log_2 n \) gates. The most operations for an output are those specified in formula (5). This shows that it takes at most 4 additional operations to get from the outputs of a single-size look-ahead carry circuit to the outputs of the double-size next level. It takes \( \log_2 n \) doublings to get from 1-bit to \( n \)-bit modules, so the number of levels of nested half-adder circuits is \( \log_2 n \). Hence, the largest number of operations on a path from any one input bit to output bit is \( 4 \log_2 n \).

Also, since each level has a total of at most \( 4n \) gates, it follows that the \textit{total} number of operations used in the parallel half-adder to calculate \textit{all} the output digits is at most \( 4n \log_2 n \). This is larger than the total for a ripple-carry adder by a factor proportional to \( \log_2 n \). So we get an exponential speed-up—from proportional \( n \) to \( \log n \)—if we increase the size of the circuit by a factor which grows very slowly—proportional to \( \log_2 n \).
Figure 1  Structure of a Double-size add1 Module.