Problem Set 1 solutions

https://github.com/spamegg1

May 21, 2022

Contents

1 Problem 1.

Prove that $\log_4(6)$ is irrational.

Proof. 1. Argue by contradiction and assume that $\log_4(6)$ is rational.

2. By definition of rationality, there exist integers m, n such that

$$\log_4(6) = \frac{m}{n}$$

where m and n are in lowest terms (have no common prime factors), and $n \neq 0$.

3. We'd like to get rid of the $\log_4$. Raising 4 to the power both sides, we get

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

$$6 = 4^{m/n}$$

4. Raising both sides to the power n we get

$$6^n = 4^m$$

5. Noticing that $6 = 2 \cdot 3$ and $4 = 2^2$ we get

$$(2 \cdot 3)^n = (2^2)^m$$

$$2^n \cdot 3^n = 2^{2m}$$

6. Dividing by 2^n we get

$$3^n = 2^{2m-n}$$

7. Since 2 and 3 are both prime numbers, by the Fundamental Theorem of Arithmetic (the uniqueness part), the only way to satisfy this equation is if both sides are equal to 1, in other words, if both sides have exponent 0.

8. So $n = 0$ and $2m - n = 0$, which gives us $m = n = 0$. This is a contradiction to $n \neq 0$ in Step 2.

9. So our initial assumption was false, hence $\log_4(6)$ is irrational. □

2 Problem 2.

Use the Well Ordering Principle to prove that

$$n \leq 3^{n/3}$$

for every nonnegative integer, n.

Hint: Verify for $n \leq 4$ by explicit calculation.

Proof. Let's verify the $n \leq 4$ cases first:

1. Case $n = 0$: $0 \leq 3^{0/3}$ is true, because LHS is 0 and the RHS is $3^0 = 1$.

2. Case $n = 1$: $1 \leq 3^{1/3}$ is true, because LHS is 1 and the RHS is $3^{1/3} = 1,44224957$ (approximately).

3. Case $n = 2$: $2 \leq 3^{2/3}$ is true, because LHS is 2 and the RHS is $3^{2/3} = 2,080083823$ (approximately).

4. Case $n = 3$: $3 \leq 3^{3/3}$ is true, because LHS is 3 and the RHS is $3^{3/3} = 3$.

5. Case $n = 4$: $4 \leq 3^{4/3}$ is true, because LHS is 4 and the RHS is $3^{4/3} = 4,326748711$ (approximately).

Now the rest of the proof.

6. Argue by contradiction and assume that there exists a nonnegative integer n such that $n > 3^{n/3}$. (We know from steps 1-5 that it must be that $n > 4$.) So $m > 3^{m/3}$.

7. Define the set of counterexamples

$$S = \{k \in \mathbb{N} \mid k > 3^{k/3}\}$$

8. By (6) the set S is nonempty, so by the Well-Ordering Principle, S has a smallest element m. (Again we know that it must be that $m > 4$.)

9. Since m is the smallest nonnegative integer with this property, for all $k < m$ it is true that $k \leq 3^{k/3}$.

10. In particular, it is true that $m - 1 \leq 3^{(m-1)/3}$.

11. Taking cubes of both sides we get $(m-1)^3 \leq 3^{m-1}$.

12. Multiplying both sides by 3 we get $3(m-1)^3 \leq 3^m$.

13. By (8) we have $3^m < m^3$, so combining this with (12) we get

$$3(m-1)^3 \leq 3^m < m^3$$

$$3(m-1)^3 < m^3$$

14. Dividing by $3m^3$ we get

$$\frac{(m-1)^3}{m^3} < \frac{1}{3}$$

$$\left(\frac{m-1}{m}\right)^3 < \frac{1}{3}$$

15. Taking cube roots we get

$$\frac{m-1}{m} < \left(\frac{1}{3}\right)^{1/3}$$

16. Notice that

$$\frac{m-1}{m} = 1 - \frac{1}{m}$$

so using this in (15) we get

$$1 - \frac{1}{m} < \left(\frac{1}{3}\right)^{1/3}$$

$$1 < \left(\frac{1}{3}\right)^{1/3} + \frac{1}{m}$$

$$1 - \left(\frac{1}{3}\right)^{1/3} < \frac{1}{m}$$

17. Taking reciprocals (which reverses the inequality) we get

$$\frac{1}{1 - \left(\frac{1}{3}\right)^{1/3}} > m$$

18. Let's calculate the LHS on a calculator, we get

$$\frac{1}{1 - \left(\frac{1}{3}\right)^{1/3}} = 3,261166697$$

approximately.

19. So we have $m < 3,261166697$, which is a contradiction to the fact that $m > 4$.

20. So our initial assumption in (6) was false, therefore $n \leq 3^{n/3}$ for all nonnegative integers n. □

3 Problem 3

3.1 (a)

Verify by truth table that (P IMPLIES Q) OR (Q IMPLIES P) is valid.

Proof. I am using the symbol $\implies$ for IMPLIES and the symbol $\vee$ for OR.

P	Q	$P \implies Q$	$Q \implies P$	$(P \implies Q) \vee (Q \implies P)$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

□

3.2 (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.

Proof. The formula we want for R is the biconditional: P IFF Q.

We know that P IFF Q is equivalent to: (P IMPLIES Q) AND (Q IMPLIES P).

We also know that A IMPLIES B is equivalent to NOT(A) OR B.

So (P IMPLIES Q) AND (Q IMPLIES P) is equivalent to:

R = (NOT(P) OR Q) AND (NOT(Q) OR P) □

3.3 (c)

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 is valid iff NOT(P) is not satisfiable.

Proof. Just some explanations first.

"Valid" means true under every possible assignment.

"Satisfiable" means that there exists AT LEAST ONE assignment which makes the formula true. (It could still be false for all the other possible assignments.)

We want to prove P is valid iff NOT(P) is not satisfiable. This is a biconditional statement, so let's prove each direction:

Proving: if P is valid, then NOT(P) is not satisfiable.

1. Assume P is valid.

2. So P is true under every possible assignment to its variables.

3. So NOT(P) is false under every possible assignment to P's variables.

4. So there is no assignment that makes NOT(P) true.

5. So by definition of "satisfiable", NOT(P) is not satisfiable.

Proving: if NOT(P) is not satisfiable, then P is valid.

1. Assume NOT(P) is not satisfiable.

2. By the definition of "satisfiable" there is no assignment to P's variables that makes NOT(P) true.

3. So under every possible assignment, NOT(P) is false.

4. So under every possible assignment, P is true.

5. Therefore P is valid. □

3.4 (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.

Proof. "Consistent" means that there is at least one assignment to the variables of the formulas that make them all true.

So... "not consistent" means that there is no assignment to the variables of the formulas that make them all true.

In other words, "not consistent" means that for every possible assignment to the variables, not all of the formulas are simultaneously true (at least one of the formulas is false).

How can we express this idea in a single formula?

"All the formulas being simultaneously true" is equivalent to the formula P_1 AND P_2 AND ... AND P_k.

So, "not all of the formulas being simultaneously true" is equivalent to the formula:

$S = \text{NOT}(P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_k)$.

Okay, we came up with a formula, but does it really do what we want? Let's make sure we prove this.

Again, we are trying to prove a biconditional statement: the set $P_1, \ldots P_k$ is not consistent iff S is valid. Let's prove each direction separately:

Proving: if the set $P_1, \ldots P_k$ is not consistent, then $S = \text{NOT}(P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_k)$ is valid.

1. Assume the set $P_1, \ldots P_k$ is not consistent.

2. By definition of "not consistent", for every assignment to the variables, at least one of $P_1, \ldots P_k$ is false.

3. So, for every assignment to the variables, P_1 AND P_2 AND ... AND P_k is false (because at least one of them is false, and AND requires all of them to be true for the overall statement to be true).

4. So, for every assignment to the variables, $\text{NOT}(P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_k)$ is true.

5. Therefore S is valid.

Proving: if $S = \text{NOT}(P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_k)$ is valid, then the set $P_1, \ldots P_k$ is not consistent.

1. Assume $S = \text{NOT}(P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_k)$ is valid.

2. By definition of "valid", for every assignment to the variables, S is true.

3. So, for every assignment to the variables, P_1 AND P_2 AND ... AND P_k is false.

4. So, for every assignment to the variables, at least one of $P_1, \ldots P_k$ is false (otherwise P_1 AND P_2 AND ... AND P_k would have to be true).

5. Therefore the set $P_1, \ldots P_k$ is not consistent. □

4 Problem 4.

4.1 (a)

A 1-bit *add1* module just has input a_0 . Write propositional formulas for its outputs c and p_0.

Proof. Remember the convention that T corresponds to 1 and F to 0.

According to the description, the output will be a 2-bit representation of "one plus the input".

If the input bit is $a_0 = 0$, and we add 1 to it, we get 1, which would be represented as 01. In this case $c = 0$ and $p_0 = 1$.

If the input bit is $a_0 = 1$, and we add 1 to it, we get 2, which would be represented as 10. In this case $c = 1$ and $p_0 = 0$.

So we can see that the carry c should be the same as the input, whereas p_0 is the opposite (negation) of the input.

So $c = a_0$ and $p_0 = \text{NOT}(a_0)$. □

4.2 Warning

For the rest of the problem, first we need to know how a half-adder works. Then we need to figure out how a parallel half-adder works. And THEN we can do the rest of the problem.

First I'll try to understand a 2-bit half-adder. Then I'll understand the $(n + 1)$-bit half-adder.

4.3 2-bit Half-adder

This is taken from the textbook. Here is how a 2-bit half-adder works.

a_1, a_0, b are the input bits.

Together a_1a_0 represents a 2-bit number from 0 to 3.

Then b is either 0 or 1, and it gets added to a_1a_0.

o_1, o_0, c are the output bits.

They represent the result, which is a 3-bit number co_1o_0.

$$\begin{array}{rcl} c_0 & = & b \\ o_0 & = & a_0 \text{ XOR } c_0 \\ c_1 & = & a_0 \text{ AND } c_0 \quad \text{(the carry into column 1)} \\ o_1 & = & a_1 \text{ XOR } c_1 \\ c_2 & = & a_1 \text{ AND } c_1 \quad \text{(the carry into column 1)} \\ c & = & c_2 \end{array}$$

Here is an example. Let's say $a_1 = 1$, $a_0 = 1$ and $b = 1$.

So the binary input number is 11 which is the integer 3. We will add 1 to it, and get 4. Then it will become 100.

First we compute c_0, which is $c_0 = b = 1$.

Then we compute o_0 by using a_0 XOR $c_0 = 1$ XOR $1 = 0$.

Then we compute c_1 by using a_0 AND $c_0 = 1$ AND $1 = 1$.

Then we compute o_1 by using a_1 XOR $c_1 = 1$ XOR $1 = 0$.

Then we compute c_2 by using a_1 AND $c_1 = 1$ AND $1 = 1$.

Finally we compute $c = c_2 = 1$. So the result is the 3-bit binary number $co_1o_0 = 100$, which is the integer 4.

The computation is sequential, it goes $c_0 \to o_0 \to c_1 \to o_1 \to c_2$.

o_0 and c_1 both depend on c_0 so they have to wait for c_0 to be computed first.

4.4 (n+1)-bit Half-adder

This is taken from 2010 Lecture solutions (the file named "MIT6-042JS10-lec04-sol.pdf").

$a_{n+1}, a_n, \ldots, a_0, b$ are the input bits.

Together $a_{n+1}a_n \ldots a_1a_0$ represent a $(n+1)$-bit number from 0 to $2^{n+1} - 1$.

Then b is either 0 or 1, and it gets added to $a_{n+1}a_n \ldots a_1a_0$.

$o_n, \ldots, o_0, c$ are the output bits.

They represent the result, which is an $(n+1)$-bit number $co_n \ldots o_0$.

$$\begin{array}{rcl} c_0 & = & b \\ o_i & = & a_i \text{ XOR } c_i \quad \text{for } 0 \le i \le n \\ c_{i+1} & = & a_i \text{ AND } c_i \quad \text{for } 0 \le i \le n \\ c & = & c_{n+1} \end{array}$$

Let's think about the order of execution.

First we compute $c_0 = b$ from the input b.

Then we compute o_0 by using: a_0 XOR c_0. Notice that the computation of o_0 has to wait for the computation of c_0 to finish first.

Then we compute c_1 by using: a_0 AND c_0. Notice that the computation of c_1 has to wait for the computation of c_0 to finish first.

We can now compute o_1 by using a_1 XOR c_1. Again notice that the computation of o_1 has to wait for the computation of c_1 to finish first.

Then we repeat this procedure.

We compute c_2 from: a_1 AND c_1 (again, c_2 has to wait for c_1).

We compute o_2 from: a_2 XOR c_2 (again, it has to wait for c_2).

And so on. The computation order goes: $c_0 \to o_0 \to c_1 \to o_1 \to \ldots c_{n+1} \to c$. As you can see this is a sequential computation, not parallel.

4.5 Where to go next?

I don't see how to go from an $(n+1)$-bit add1 module to an $(n+1)$-bit parallel half-adder, as they ask us in part (b) of the problem. So first I'm going to try writing a 2-bit parallel half-adder. This uses a 2-bit add1 module. So let's talk about that first.

4.6 2-bit PARALLEL add1 module

The assignment says:

Parallel half-adders are built out of parallel add1 modules.

OK, so the add1 module itself is parallel! I guess that the $(n+1)$-bit add1 module would use n (or maybe $n+1$) 1-bit add1 modules in parallel.

I actually tried writing out the formulas for this 2-bit parallel add1 module, but it gets quite complicated! p_0 was still the negation of a_0 but p_1 was not clear at all. It required the introduction of two auxiliary variables, one that computes p_1 under the assumption that $p_0 = 0$, one that $p_0 = 1$, and then combined with a "select logic."

So I understand why the assignment does not make us write the $(n+1)$-bit add1 module explicitly. The logic seems quite complicated.

An $(n+1)$-bit add1 module takes as input the $(n+1)$-bit binary representation, $a_n \ldots a_1 a_0$, of an integer, s, and produces as output the binary, $cp_n \ldots p_1 p_0$, of $s+1$.

So let's take a look when $n = 1$ (the 2-bit case):

A 2-bit add1 module takes as input the 2-bit binary representation, $a_1 a_0$, of an integer, s, and produces as output the binary, $cp_1 p_0$, of $s+1$.

We just need to keep this in mind to write the 2-bit parallel half-adder below.

Note that the parallelism comes from the add1 module itself, not based on how we'll be writing the "parallel" half-adder. Once again, the assignment says "Parallel half-

adders are built out of parallel add1 modules." So we just assume that the add1 module is parallel already.

4.7 From a 2-bit add1 module, to a 2-bit half-adder

The assignment says:

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

Let's consider $n = 1$.

We assume that we are given a 2-bit parallel add1 module:

A 2-bit add1 module takes as input the 2-bit binary representation, a_1a_0, of an integer, s, and produces as output the binary, cp_1p_0, of $s+1$.

We want to make a 2-bit half-adder out of it:

a_1, a_0, b are the input bits. Together a_1a_0 represents a 2-bit number from 0 to 3. Then b is either 0 or 1, and it gets added to a_1a_0. o_1, o_0, c are the output bits. They represent the result, which is a 3-bit number co_1o_0.

So... the only difference between the 2-bit parallel half-adder (what we want) and the 2-bit parallel add1 module is that, for the half-adder, the input might be 0; whereas for add1 the input is always 1.

So if $b = 1$ we want to just use the add1 directly. If $b = 0$ then we simply return the input (with the 3rd bit carry being 0).

Let's think about some examples. Let's say the input is $a_1a_0 = 11$ which is 3.

If the other input b is 1, then the result will be 4. So $co_1o_0 = 100$ and this is exactly what we get from the add1 module $cp_1p_0 = 100$. So in this case $c = c$, $o_1 = p_1$, $o_0 = p_0$.

If the other input b is 0, then the result will be 3. So $co_1o_0 = 011$. So in this case $c = 0$, $o_1 = a_1$, $o_0 = a_0$.

The assignment is not asking us to handle c, but let's do it anyway. Let's try to isolate the c output of the half-adder. It's equal to the c output of the add1 module when $b = 1$, and 0 when $b = 0$. We can express this as:

$c_{half-adder} = c_{add1}$ AND b.

The logic is similar for o_1 and o_0. When $b = 1$, $o_1 = p_1$ and $o_0 = p_0$. When $b = 0$, $o_1 = a_1$ and $o_0 = a_0$. So it selects between a_is and p_is depending on b. How can we express this?

$o_1 = (p_1$ AND $b)$ OR $(a_1$ AND NOT$(b))$

$o_0 = (p_0$ AND $b)$ OR $(a_0$ AND NOT$(b))$

You can convince yourself that this is correct on all 8 possible input combinations.

4.8 (b)

Explain how to build an $n+1$-bit parallel half-adder from an $n+1$-bit *add1* module by writing a propositional formula for the half-adder output, o_i, using only the variables a_i, p_i, and b.

Proof. The solution we came up with above for the 2-bit parallel half-adder generalizes:

$$o_i = (p_i \text{ AND } b) \text{ OR } (a_i \text{ AND NOT}(b))$$

□

5 Double-size add1

Now we have two $(n+1)$-bit add1 modules put together. The input has $2n+2$ bits and the output has $2n+3$ bits. c is the highest bit of the entire output (of the double-size add1 module). $c_{(1)}$ is the highest bit of the "right" add1 module that handles the lower digits, and $c_{(2)}$ is the highest bit of the "left" add1 module that handles the higher digits. Let's do an example to understand:

$n = 1$

Input: $a_3a_2a_1a_0$

a_1a_0 is handled by the right add1 module, its output is $c_{(1)}p_1p_0$.

a_3a_2 is handled by the left add1 module, its output is $c_{(2)}r_1r_0$.

The final output of the double add1 module: $cp_3p_2p_1p_0$.

Notice that r_1r_0 can be different than p_3p_2.

5.0.1 Neither half carries.

Let's say that the input is $a_3a_2a_1a_0 = 1010$. This is the integer 10. So we should get the integer 11 when we add 1 to it. This is 1011 in binary. So the result should be 01011 which means $c = 0$.

The right add1 module (that handles the lower bits) takes the input 10 (the integer 2), adds 1 to it (the integer $3 = 11$) and returns the output $c_{(1)}p_1p_0 = 011$ in binary.

The left add1 module (that handles the higher bits) takes the input 10 (the integer 2), adds 1 to it (the integer $3 = 011$) and returns the output $c_{(2)}r_1r_0 = 011$ in binary.

In this case we cannot take the lower 2 bits of these results (11 and 11) and concatenate them together (1111) to get the right result. We also cannot get the right result by adding a $c = 0$ (01111) or $c = 1$ (11111) to the left of it.

We only add 1 to the lower bits, but since the lower bits do not produce a carry in this case, we should not be adding 1 to the higher half. The higher half should stay the same as its input, namely 10, instead of becoming 011.

So the values of the higher bits $c_{(2)}r_1r_0$ depend on the lower half's carry $c_{(1)}$. This can be handled with a select logic like we did in the 2-bit parallel half-adder. If the lower bits produce a carry $c_{(1)} = 1$ then $c_{(2)}r_1r_0$ should be the result of using add1 on them. Otherwise, $c_{(2)}r_1r_0$ should be the same as $0a_3a_2$.

5.0.2 Right half carries, left does not

Let's say that the input is $a_3a_2a_1a_0 = 1011$. This is the integer 11. So we should get the integer 12 when we add 1 to it. This is 1100 in binary. So the result should be 01100 which means $c = 0$.

The right add1 module (that handles the lower bits) takes the input 11 (the integer 3), adds 1 to it (the integer $4 = 100$) and returns the output $c_{(1)}p_1p_0 = 100$ in binary.

The left add1 module (that handles the higher bits) takes the input 10 (the integer 2), adds 1 to it (the integer $3 = 11$) and returns the output $c_{(2)}r_1r_0 = 011$ in binary.

So in this case we can take the outputs of the individual add1 modules (ignoring the carries), which are 11 and 00, and concatenate them together to get the correct result 1100 (and set $c = 0$).

In this case the lower bits produced a carry, so the higher bits are the result of using add1 on them.

5.0.3 Left half carries, right does not

Let's say that the input is $a_3a_2a_1a_0 = 1110$. This is the integer 14. So we should get the integer 15 when we add 1 to it. This is 1111 in binary. So the result should be 01111 which means $c = 0$.

The right add1 module (that handles the lower bits) takes the input 10 (the integer 2), adds 1 to it (the integer $3 = 11$) and returns the output $c_{(1)}p_1p_0 = 011$ in binary.

The left add1 module (that handles the higher bits) takes the input 11 (the integer 3), adds 1 to it (the integer $4 = 100$) and returns the output $c_{(2)}r_1r_0 = 100$ in binary.

In this case we cannot take the lower 2 bits of these results (11 and 00) and concatenate them together (0011) to get the right result. We also cannot get the right result by adding a $c = 1$ to the left of it (10011).

Again, like the first case, since the lower bits did not produce a carry, the higher bits should remain the same as their inputs (instead of the result of add1 on them).

5.0.4 Both halves carry

Let's say that the input is $a_3a_2a_1a_0 = 1111$. This is the integer 15. So we should get the integer 16 when we add 1 to it. This is 10000 in binary. So the result should be 10000 which means $c = 1$.

The right add1 module (that handles the lower bits) takes the input 11 (the integer 3), adds 1 to it (the integer $4 = 100$) and returns the output $c_{(1)}p_1p_0 = 100$ in binary.

The left add1 module (that handles the higher bits) takes the input 11 (the integer 3), adds 1 to it (the integer $4 = 100$) and returns the output $c_{(2)}r_1r_0 = 100$ in binary.

So in this case we can take the outputs of the individual add1 modules (ignoring the carries), which are 00 and 00, and concatenate them together, and add $c = 1$ on the left, to get the correct result 10000 in binary.

5.1 (c)

Write a formula for the carry, c, in terms of $c_{(1)}$ and $c_{(2)}$.

Proof. From the 4 cases we have seen earlier, we can conclude that if both halves carry, then c should be 1, and c should be 0 otherwise. The formula is:

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

□

5.2 (d)

Complete the specification of the double-size module by writing propositional formulas for the remaining outputs, p_i, for $n + 1 \leq i \leq 2n + 1$. The formula for p_i should only involve the variables $a_i, r_{i-(n+1)}$ and $c_{(1)}$.

Proof. From the 4 cases we have seen earlier, we can figure this out.

If $c_{(1)} = 0$ then the higher half should not change, it should be the same as the input. Namely, for $n + 1 \leq i \leq 2n + 1$, p_i should just be a_i (the same as the input).

If $c_{(1)} = 1$ then there is a carry of 1 coming from the lower half. So the higher half should be the result of using add1 on it. Namely, for $n + 1 \leq i \leq 2n + 1$, p_i should just be $r_{i-(n+1)}$ (the result of the higher add1 on the higher half of the input).

Combining them together we get: for $n + 1 \leq i \leq 2n + 1$

$$p_i = (c_{(1)} \text{ AND } r_{i-(n+1)}) \text{ OR } (\text{NOT}(c_{(1)}) \text{ AND } a_i)$$

□

5.3 (e)

Parallel half-adders are exponentially faster than ripple-carry half-adders. Confirm this by determining the largest number of propositional operations required to compute any one output bit of an n-bit add module. (You may assume n is a power of 2.)

Proof. □