1. Recall that a Boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) is called **monotone** if changing any of the \( n \) input bits \( x_1, \ldots, x_n \) from 0 to 1 can only ever change the output \( f(x_1, \ldots, x_n) \) from 0 to 1, never from 1 to 0.

(a) Show that any monotone Boolean function is computable by a circuit containing only AND and OR gates. (Recall that we observed the converse statement in class. Also recall we proved that any Boolean function, not necessarily monotone, is computable using AND, OR, and NOT gates: for this problem, you’ll want to do something roughly analogous, but adapted to the monotone case. As usual, you may assume that the constants 0 and 1 are available as “freebies,” to be fed as input to any gate in the circuit.)

(b) [Extra credit] Prove that there is some monotone Boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) that requires a circuit of at least \( \sim 2^n / n^{3/2} \) AND and OR gates to compute. [Hint: Use a variant of Shannon’s counting argument from class. You only need to consider a special subset of monotone functions, those that can change from 0 to 1 only when the number of 1’s in the input is exactly \( n/2 \).]

2. Let \( f(x_1, \ldots, x_n) = 1 \) if \( x_1 + \cdots + x_n \) is divisible by 3, and \( f(x_1, \ldots, x_n) = 0 \) otherwise. Here \( x_1, \ldots, x_n \) are bits. Describe the design of a circuit, composed of AND, OR, and NOT gates, that computes \( f \) given \( x_1, \ldots, x_n \). The size of your circuit should be \( O(n) \). [Hint: How would you go about this if you were checking whether \( x_1 + \cdots + x_n \) was divisible by 2?]

3. Let \( L = \{ x \in \{a, b\}^* : \text{the number of b’s in } x \text{ is divisible by } 3 \} \). Find a regular expression for \( L \). [Reminder: Regular expressions can include nesting, \( | \) (OR), and concatenation. They cannot include NOT symbols.]

4. Let \( L = \{ x \in \{a, b\}^* : x \text{ does not contain } aba \text{ as a substring} \} \). Find a regular expression for \( L \).

5. Let \( L \subseteq \{a, b\}^* \) be the language consisting of all strings that consist of the same string repeated three times, like \( ababab \). Show that \( L \) is not regular. (You can use either the pigeonhole principle or the Pumping Lemma from Sipser’s book.)

6. Let \( L = \{1^n \mid n \text{ is prime} \} \).

   (a) Show that \( L \) is not regular. (You can use the fact that there are infinitely many prime numbers.)

   (b) Explain why the regular languages are closed under complement. Conclude that \( \overline{L} = \{1^n \mid n \text{ is composite} \} \) is not regular either.

7. Concatenation of regular languages

   (a) Let \( L \subseteq \{a, b, c\}^* \) be the language consisting of all strings \( w \) that can be expressed as \( w_1 \circ w_2 \), where \( w_1 \) contains an even number of \( b \)'s (and any number of \( a \)'s and \( c \)'s), \( w_2 \) contains a number of \( c \)'s that is divisible by 3 (and any number of \( a \)'s and \( b \)'s), and \( \circ \) denotes string concatenation. Show that \( L \) is regular, by constructing an NDFA that recognizes \( L \). [Note: Your NDFA can contain \( \varepsilon \)-transitions if you like.]
(b) Let \( L \subseteq \{a, b\}^* \) (different \( L \) than in part a.) be the language consisting of all strings \( w \) that can be expressed as \( w_1 \circ w_2 \), where \( w_1 \) contains an even number of \( b \)'s (and any number of \( a \)'s), and \( w_2 \) contains a number of \( b \)'s that is divisible by 3 (and any number of \( a \)'s). Construct a DFA that recognizes \( L \). \[ \text{Hint: You could do this by first constructing an NDFA and then using the simulation of NDFA’s by DFA’s, but that’s working way too hard!} \]

(c) Generalize part a. to show that, if \( L_1 \) and \( L_2 \) are any two regular languages, then

\[
L = \{ w_1 \circ w_2 | w_1 \in L_1, w_2 \in L_2 \}
\]

is also a regular language.

8. In this problem, you’ll study one of history’s first known examples of serious computational thinking, while also brushing up on useful material from previous courses. Euclid’s algorithm is a fast way to compute the greatest common divisor GCD \((A, B)\) of two positive integers \( A \geq B \) given as input. It’s given by the following recursive pseudocode:

\[
\text{if } B \text{ divides } A \text{ then return } B
\]
\[
\text{else return GCD}(B, A \mod B)
\]

Show that, if initialized on \( n \)-bit integers \( A \geq B \), Euclid’s algorithm halts after at most \( 2^n \) iterations. \[ \text{Hint: Let } A_t \geq B_t \text{ be the arguments to the GCD function at the } t^{th} \text{ iteration, so that } A_1 = A \text{ and } B_1 = B. \text{ What can you say about the decrease of } A_t, \text{ as a function of } t? \]