4. Show that the following language is regular:

\[
\left\{ w \in \{0,1\}^* \left| \begin{array}{l}
\text{there is a way to insert the substring 011 (once) into } w \\
\text{so that the result is a multiple of 3 in binary}
\end{array} \right. \right\}
\]

**Example:** The string \( w = 1011 \) should be accepted since 
\( 101011_{\text{bin}} = 87_{\text{dec}} \), a multiple of 3. The string \( w = 1000 \) should not be accepted since none of the following are multiples of 3:

\[
\{0111000, 1011000, 1001100, 1000110, 1000011\}
\]

**Solution:** The idea is that the NFA you design should keep track of a remainder mod 3 (i.e., keep track of a state in the famous 3-state DFA for multiples of 3). The NFA should keep track of what would happen if the NFA’s input was put into the DFA. But at exactly one point, the NFA should imagine what happens if the DFA were advanced on input ‘011’.

Suppose \( w \) is a string that should be accepted. That means there’s a way to insert 011 into \( w \) that results in a multiple of 3. However, the NFA gets only \( w \) as input, with no indication of where to insert the 011 to result in a multiple of 3.

So let’s make things easier. Suppose the input did have a special character (let’s call it \( c \)) that indicates where to insert 011:

\[
A = \left\{ w \in \{0,1,c\}^* \left| \begin{array}{l}
w \text{ contains exactly one } c \text{ character, and} \\
\text{replacing } c \text{ with 011 in } w \text{ results in a multiple of 3 in binary}
\end{array} \right. \right\}
\]

It is easier to design an NFA for this language. Suppose we have read the string \( x \in \{0,1\}^* \) so far, and we are keeping track of \( \text{bin}(x) \mod 3 \). Then we see the character \( c \). We now need to compute \( \text{bin}(x011) \mod 3 \) — can you see why?

Remember this rule?

\[
\text{bin}(xb) = 2\text{bin}(x) + b, \quad \text{for } b \in \{0,1\}
\]

Let’s apply it 3 times:

\[
\text{bin}(x011) = 2(2(\text{bin}(x) + 0) + 1) + 1 = 8\text{bin}(x) + 3
\]

From this expression you can verify the following:

- if \( \text{bin}(x) \mod 3 \) is 0 then \( \text{bin}(x011) \mod 3 \) is 0
- if \( \text{bin}(x) \mod 3 \) is 1 then \( \text{bin}(x011) \mod 3 \) is 2
- if \( \text{bin}(x) \mod 3 \) is 2 then \( \text{bin}(x011) \mod 3 \) is 1

Alternatively, we could have observed that if the states of the multiples-of-3 DFA are \( q_0, q_1, q_2 \), then:

\[
\delta^*(q_0,011) = q_0 \\
\delta^*(q_1,011) = q_2 \\
\delta^*(q_2,011) = q_1
\]

With these rules in mind, the following NFA for language \( A \) should make sense:
The “top half” and “bottom half” are both the famous remainder-mod-3 DFAs. Computation starts in the top half and transitions to the bottom half when a $c$ character is read. Only the bottom half accepts. The transitions on character $c$ follow the $\text{bin}(x011)$ rule given above.

Now, back to the language in the homework problem. The input does not contain a special character to say where to insert “011”. Instead, we just have to guess where the special character $c$ might have been. At any point, without reading any input character, we should be allowed to do whatever the previous NFA did on input $c$. This effectively turns $c$-transitions into $\varepsilon$-transitions:

So this NFA uses its states to keep track of a remainder mod 3 (this is like the “column” of a state) and a single bit (this is the distinctions between states in top row and bottom row). This bit says whether we have “imagined” a 011 yet. At any time, if this internal state bit is 0, then we can set it to 1 by “imagining” a 011 string, and advancing the mod-3 DFA on 011, all without consuming any input.