1 Why Reversal Preserves Mod-3-equals-0

I prove by example, and leave a general proof for you.

1.1 Consider odd-length strings

\texttt{abcde} vs \texttt{edcba} in binary. The values are, respectively

- $2^4a + 2^3b + 2^2c + 2^1d + 2^0e$, versus
- $2^4c + 2^1d + 2^2c + 2^1b + 2^0a$

After taking mod-3 (i.e. $\%3$), and asking the question of interest (i.e. is the modulus equal to 0), we have

- $(2^4\%3a + 2^3\%3b + 2^2\%3c + 2^1\%3d + 2^0\%3e)\%3 = 0$, versus
- $(2^4\%3c + 2^1\%3d + 2^2\%3c + 2^1\%3b + 2^0\%3a)\%3 = 0$

This reduces to

- $(a + 2b + c + 2d + e)\%3 = 0$, versus
- $(e + 2d + c + 2b + a)\%3 = 0$

For odd-length strings, the situation is the same! So reversal preserves mod-3-equal-0. (Generalize this to $N$ bits.)

1.2 Consider even-length strings

\texttt{abcd} vs \texttt{dcba} in binary. The values are, respectively

- $2^4a + 2^3b + 2^1c + 2^0d$, versus
- $2^3d + 2^2c + 2^1b + 2^0a$

After taking mod-3 (i.e. $\%3$), and asking the question of interest (i.e. is the modulus equal to 0), we have

- $(2a + b + 2c + d)\%3 = 0$, versus
\[(2d + c + 2b + a) \equiv 0 \mod 3\]

Not quite the same. However, take \texttt{abcd} vs. \texttt{dcba0} (double the second number) because doubling preserves \(\mod 3 = 0\). (If \(n\) is divisible by 3 without remainder then so is \(2n\).)

Then we have

\[
\begin{align*}
(2a + b + 2c + d) &\equiv 0 \mod 3, \text{ versus} \\
(d + 2c + b + 2a + 0) &\equiv 0 \mod 3
\end{align*}
\]

Now the situation is the same.

## 2 Half Language

Define the convention of starting an NFA in a set of states. This is a simple idea: if an NFA is required to start at states \(\{a, b, c\}\), it can be easily achieved in our style of NFAs (so far required to begin operation in a single state) as follows: Introduce a new initial state; introduce \(\epsilon\) jumps to each state in \(\{a, b, c\}\). Hereafter, we will assume this ability, and often employ \(q_0\) to be a set of states.

Given a DFA \(D\) with

\[
(Q, \Sigma, \delta, q_0, F)
\]

build an NFA with its own \(\langle Q, \Sigma, \delta, q_0, F \rangle\) tuple as follows (there was a typo in the earlier version; now fixed):

\[
\begin{align*}
(Q \times Q \times Q, \Sigma, \delta_N, \{(q_0, q, q) \mid q \in Q\}, \{(q, q, q1) \mid q1 \in F\}) \\
\delta_N((p, q, r), q) = \{(\delta(p, a), q, \delta(r, x) \mid x \in \Sigma)\}
\end{align*}
\]

Let’s use the analogy of frogs with synchronized jumps. Then, the above \(\delta_N\) is saying:

- Start a frog at \(q_0\)
- Put a sign-post at every possible \(q\) (in the NFA initial state)
- Start another frog at every sign post \(q\) also

You can see how the first position and the third position of the tuple are moving forward. \textit{The first position moves according to the input. The third position moves chaotically in all directions—all it needs to do is describe the same length, regardless of the string processed.} The final states are those where the first and second positions are the same and the third position is at a final state. This means that the first position has traced out a path of the same length as the third position has while going to a final state.

## 3 Middle-third Language

The ideas are similar and you should try and build this \(\delta_N\) yourself, claiming extra credits from me! Hint: use a four-tuple and two frogs.