Solutions for Practice with Automata

DFAs, States, and Information

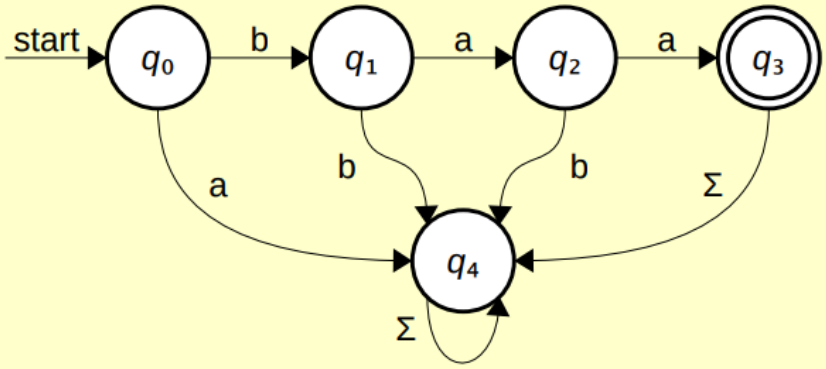
Because this problem was really designed to get you exploring around with DFAs, we haven't included solutions per se. However, here are a few thoughts that we hoped you'd have in the course of working through this problem:

- The language L_1 consists of all strings whose length leaves a remainder of three when divided by five.
- For L_1, the DFA needs to remember the remainder of the length of the string when that length is divided by five. It's probably best to make the states correspond to the different remainders.
- For L_2, the DFA needs to remember both the remainder of the number of a's when divided by two and the remainder of the number of b's when divided by three. Each state corresponds to a pair of an "a remainder" and a "b remainder." There are six possible combinations. You can think about building the DFA by having each transition move between states by updating either the number of a's or the number of b's, depending on what was read.

Designing DFAs

i. Let $\Sigma = \{a, b\}$ and let $L = \{baa\}$. Design a DFA for L.

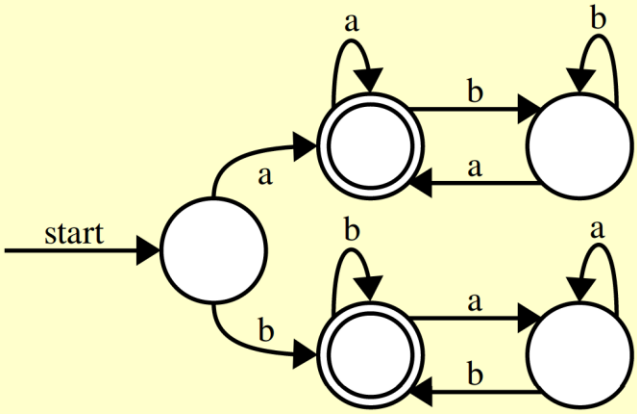
Here is one possible option:



This basically consists of a chain of states leading up to baa with a dead state for any deviations from that string.

ii. Let $\Sigma = \{a, b\}$ and let $L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$. Design a DFA for L.

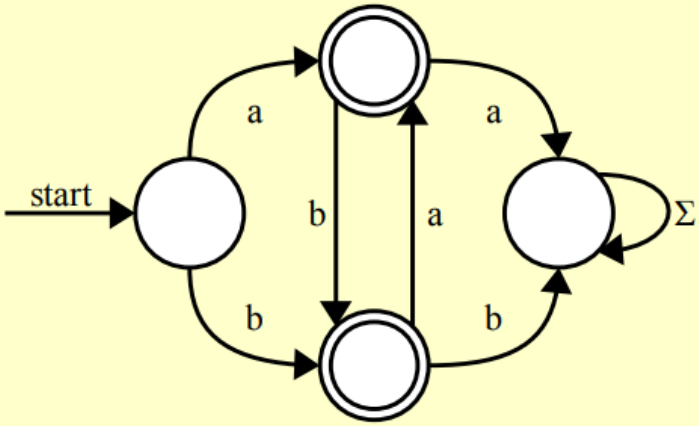
Here is one possible option:



In the start state, we wait to see what the first character is. We then transition either to the top branch (first character *a*) or the bottom branch (first character *b*). At that point, we just need to remember what the last character we read was and can use that to decide whether to accept.

iii. Let $\Sigma = \{a, b\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a nonempty string whose characters alternate between a's and b's $\}$. Design a DFA whose language is L.

Here's one possible option:



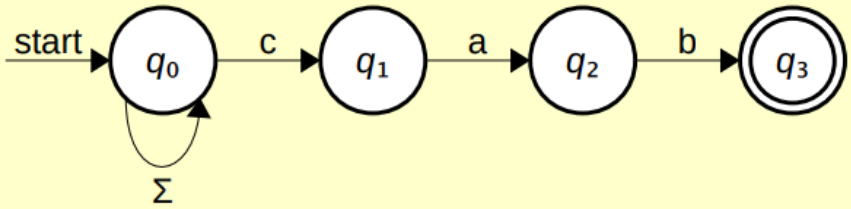
In the start state, we wait to see what the first character is. We then transition either to the top state (first character *a*) or the bottom state (first character *b*). The transitions then permit us to bounce back and forth between the two states as long as we alternate, and if we ever deviate from the rule we enter the dead state on the right.

Why we asked this question: There are a couple of canonical tricks in the design of DFAs. Part (i) was a check to make sure you remembered to include a transition on every state/symbol combination. Part (ii) demonstrates how to build an automaton that can remember one of finitely many pieces of information persistently (namely, what the first character is), and part (iii) combines the two.

Designing NFAs

i. Let $\Sigma = \{a, b, c\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ ends in } cab \}$. Design an NFA for L.

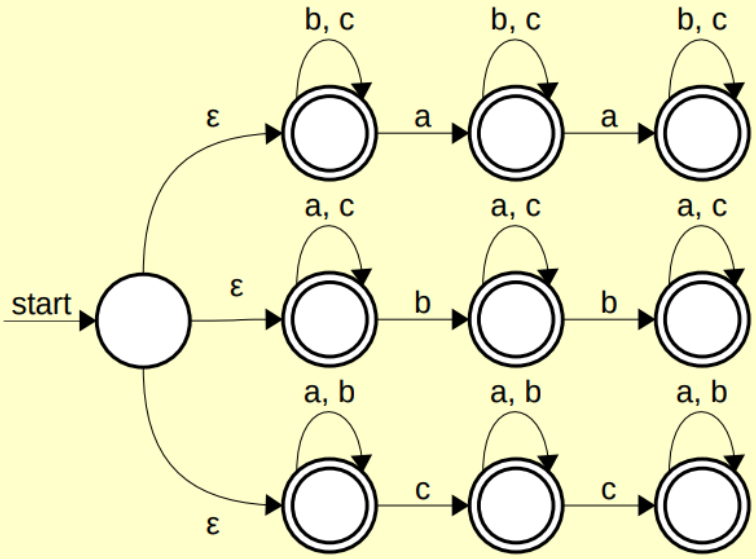
Here is one possibility:



This NFA uses nondeterminism to guess when it's three characters from the end. If we guess correctly and the input does end in *cab*, we accept. The machine otherwise doesn't accept.

ii. Let $\Sigma = \{a, b, c\}$ and let $L = \{ w \in \Sigma^* \mid$ some character in Σ appears at most twice in $w \}$. Design an NFA for L.

Here's one option:



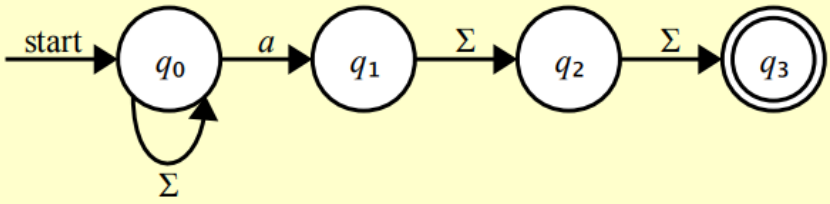
This NFA uses nondeterminism to guess which character will appear at most twice. Once it does, it transitions into one of three different chains (one per possible character) that count the number of times that character appears. If we exceed the limit, the automaton dies off in that branch. This means that to accept, we have to guess the branch correctly, then survive long enough to accept.

Why we asked this question: Unlike DFA design, NFA design often focuses on determining what information you need to nondeterministically guess. These two problems show off the hybrid model of using nondeterminism to guess some information, then determinism (in some form) to check it. Part (ii) was specifically designed to get you thinking about how to count with states.

Automata Transformations

i. Let Σ = {*a*, *b*} and let *L* = { *w* ∈ Σ* | the third-from-last character of *w* is *a* }. Design an NFA for *L*. Your NFA should use at most four states.

Along the lines of the "ends with *cab*" problem, this NFA stays in the start state until it nondeterministically guesses that it's three characters from the end:



ii. Using the subset construction, convert your NFA from part (i) into an equivalent DFA. (Although you could just directly design a DFA for this language, we want you to practice using the subset construction to get a sense of how it works.) You may find it useful to construct the transition table for the DFA rather than to write out a state-transition diagram.

For readability's sake, we've encoded this one as a transition table. It's shown below:

state	***a***	***b***
{q_0}	{q_0, q_1}	{q_0}
{q_0, q_1}	{q_0, q_1, q_2}	{q_0, q_2}
{q_0, q_1, q_2}	{q_0, q_1, q_2, q_3}	{q_0, q_2, q_3}
{q_0, q_2}	{q_0, q_1, q_3}	{q_0, q_3}
*{q_0, q_1, q_2, q_3}	{q_0, q_1, q_2, q_3}	{q_0, q_2, q_3}
*{q_0, q_2, q_3}	{q_0, q_1, q_3}	{q_0, q_3}
*{q_0, q_1, q_3}	{q_0, q_1, q_2}	{q_0, q_2}
*{q_0, q_3}	{q_0, q_1}	{q_0}

Why we asked this question: The beauty of the automata transformations we've seen so far is that it lets us study the same class of objects – the regular languages – using one of three totally different lenses. The constructions are a bit tricky, though, and to help you get a handle for how they work, we wanted you to practice working through them on a few special cases. This particular case, in our opinion, wasn't too tough of a sample input.