Deterministic Finite Automata

1 Basic Definitions

Computational Model. Anything that processes information can be called a computer. However, there can be restrictions on how the information can be processed (either universal restrictions imposed by, say, the laws of physics, or restrictions imposed by the particular setting we are interested in).

A *computational model* is a set of allowed rules for information processing. Given a computational model, we can talk about the *computers* (or *machines*) allowed by the computational model. A computer is a specific instantiation of the computational model, or in other words, it is a specific collection of information processing rules allowed by the computational model.

Note that even though the terms "computer" and "machine" suggest a physical device, in this course we are not interested in physical representations that realize computers, but rather in the mathematical representations. The term *algorithm* is synonymous to computer (and machine) and makes it more clear that we are not necessarily talking about a physical device.

In this section we give the basic definitions regarding the computational model known as deterministic finite automata.

Definition (Deterministic Finite Automaton (DFA))**.** A *deterministic finite automaton* (*DFA*) M is a 5-tuple

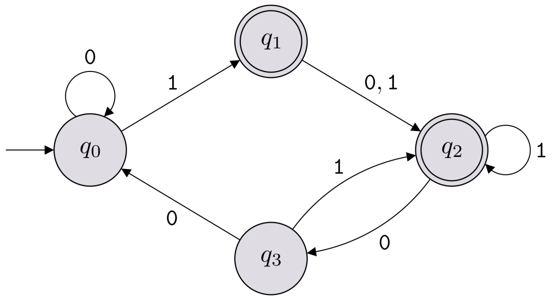
$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a non-empty finite set
  (which we refer to as the *set of states of the DFA*);
- Σ is a non-empty finite set
  (which we refer to as the *alphabet of the DFA*);

- δ is a function of the form $\delta : Q \times \Sigma \to Q$
  (which we refer to as the *transition function of the DFA*);
- $q_0 \in Q$ is an element of Q
  (which we refer to as the *start state of the DFA*);
- $F \subseteq Q$ is a subset of Q
  (which we refer to as the *set of accepting states of the DFA*).

Note (State diagram of a DFA)**.** It is very common to represent a DFA with a *state diagram*. Below is an example of how we draw a state diagram of a DFA:



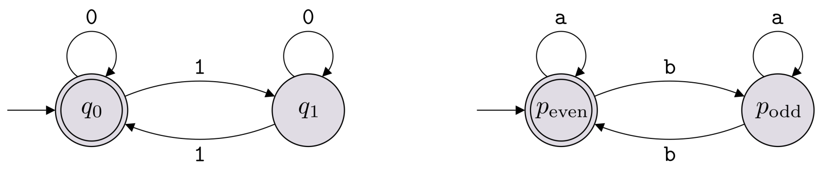
In this example, $\Sigma = \{\texttt{0}, \texttt{1}\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_1, q_2\}$. The labeled arrows between the states encode the transition function δ, which can also be represented with a table as shown below (row $q_i \in Q$ and column $b \in \Sigma$ contains $\delta(q_i, b)$).

	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2

The start state is labeled with q_0, but if another label is used, we can identify the start state in the state diagram by identifying the state with an arrow that does not originate from another state and goes into that state.

Remark (The labels of DFA components)**.** In the definition above, we used the labels $Q, \Sigma, \delta, q_0, F$. One can of course use other labels when defining a DFA as long as it is clear what each label represents.

Remark (Equivalence of DFAs)**.** We'll consider two DFAs to be equivalent/same if they are the same machine up to renaming the elements of the sets Q and Σ. For instance, the two DFAs below are considered to be the same even though they have different labels on the states and use different alphabets.



The flexibility with the choice of labels allows us to be more descriptive when we define a DFA. For instance, we can give labels to the states that communicate the "purpose" or "role" of each state.

Definition (Computation path for a DFA)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w = w_1 w_2 \cdots w_n$ be a string over an alphabet Σ (so $w_i \in \Sigma$ for each $i \in \{1, 2, \ldots, n\}$). The *computation path* of M with respect to w is a specific sequence of states $r_0, r_1, r_2, \ldots, r_n$ (where each $r_i \in Q$). We'll often write this sequence as follows.

$$\begin{array}{ccccc} & w_1 & w_2 & \ldots & w_n \\ \hline r_0 & r_1 & r_2 & \ldots & r_n \end{array}$$

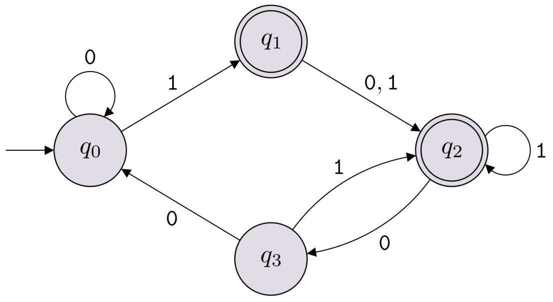
These states correspond to the states reached when the DFA reads the input w one character at a time. This means $r_0 = q_0$, and

$$\forall i \in \{1, 2, \ldots, n\}, \quad \delta(r_{i-1}, w_i) = r_i.$$

An *accepting computation path* is such that $r_n \in F$, and a *rejecting computation path* is such that $r_n \notin F$.

We say that M *accepts a string* $w \in \Sigma^*$ (or "$M(w)$ accepts", or "$M(w) = 1$") if the computation path of M with respect to w is an accepting computation path. Otherwise, we say that M *rejects the string* w (or "$M(w)$ rejects", or "$M(w) = 0$").

Example (An example of a computation path)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA in Note (State diagram of a DFA). For ease of reference, we present the state diagram once again here.



For input string $w =$ `110110`, the computation path of M with respect to w is

$$\begin{array}{ccccccc} & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline q_0 & q_1 & q_2 & q_3 & q_2 & q_2 & q_3 \end{array}$$

Since q_3 is not in F, this is a rejecting computation path, and therefore M rejects w, i.e. $M(w) = 0$.

Note (Extended transition function)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The transition function $\delta : Q \times \Sigma \to Q$ extends to $\delta^* : Q \times \Sigma^* \to Q$, where $\delta^*(q, w)$ is defined as the state we end up in if we start at q and read the string $w = w_1 \ldots w_n$. Or in other words,

$$\delta^*(q, w) = \delta(\ldots \delta(\delta(\delta(q, w_1), w_2), w_3) \ldots, w_n).$$

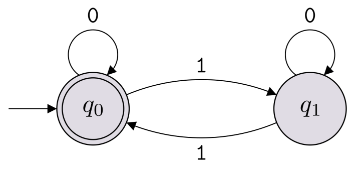
The star in the notation can be dropped and δ can be overloaded to represent both a function $\delta : Q \times \Sigma \to Q$ and a function $\delta : Q \times \Sigma^* \to Q$. Using this notation, a word w is *accepted* by the DFA M if $\delta(q_0, w) \in F$.

Definition (DFA solving a decision problem or a language)**.** Let $f : \Sigma^* \to \{0, 1\}$ be a decision problem and let M be a DFA over the same alphabet. We say that M *solves* (or *decides*, or *computes*) f if the input/output behavior of M matches f exactly, in the following sense: for all $w \in \Sigma^*$, $M(w) = f(w)$.

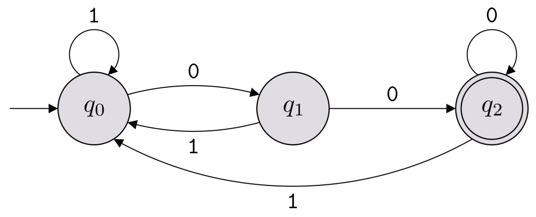
If L is the language corresponding to f, the above definition is equivalent to saying that M *solves* (or *decides*, or *computes*) L if the following holds:

- if $w \in L$, then M accepts w (i.e. $M(w) = 1$);
- if $w \notin L$, then M rejects w (i.e. $M(w) = 0$).

Example (Even number of `1`'s)**.** The following DFA solves the language consisting of all binary strings that contain an even number of `1`'s.



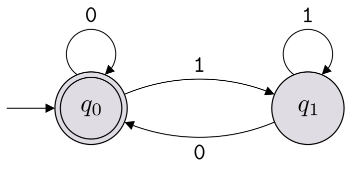
Example (Ends with `00`)**.** The following DFA solves the language consisting of all binary strings that end with `00`.

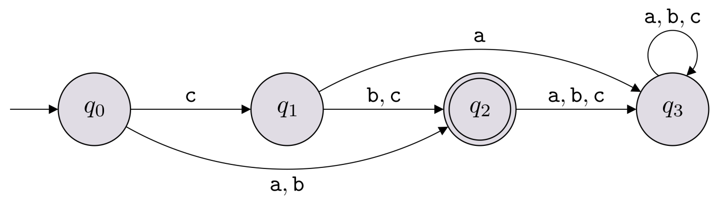


Note (Uniqueness of the language of a DFA)**.** For a DFA M, there is a unique language L that M solves. This language is often denoted by $L(M)$[1] and is referred to as the language of M. Note that

$$L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}.$$

Exercise (Describe the language of a DFA)**.** For each DFA M below, define $L(M)$.

1. 

2. 

Solution. 1. $L(M) = \{x \in \{0, 1\}^* : x \text{ ends with a } 0\} \cup \{\epsilon\}$.

2. $L(M) = \{\mathtt{a}, \mathtt{b}, \mathtt{cb}, \mathtt{cc}\}$.

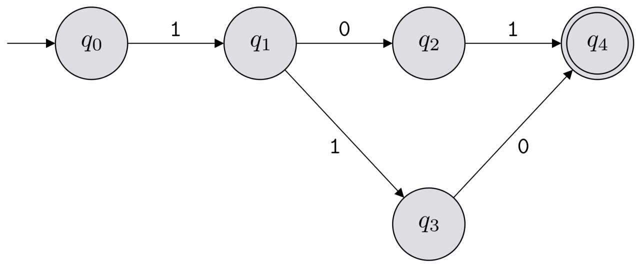
■

[1]The letter L can be overloaded: we often use it to denote a language $L \subseteq \Sigma^*$, but in this notation, it represents a function that maps a DFA to a language. Given the context, this overloading should not create any ambiguity.

Exercise (Draw DFAs)**.** For each language below (over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$), draw a DFA solving it.

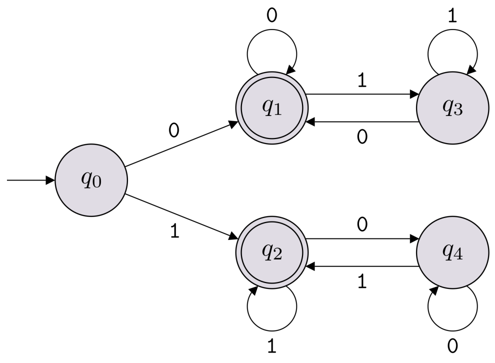
1. $\{\texttt{101}, \texttt{110}\}$
2. $\{\texttt{0}, \texttt{1}\}^* \setminus \{\texttt{101}, \texttt{110}\}$
3. $\{x \in \{\texttt{0}, \texttt{1}\}^* : x \text{ starts and ends with the same bit}\}$
4. $\{\texttt{110}\}^* = \{\epsilon, \texttt{110}, \texttt{110110}, \texttt{110110110}, \ldots\}$
5. $\{x \in \{\texttt{0}, \texttt{1}\}^* : x \text{ contains } \texttt{110} \text{ as a substring}\}$

Solution. 1. Below, all missing transitions go to a rejecting sink state (so the DFA actually has 6 states in total).

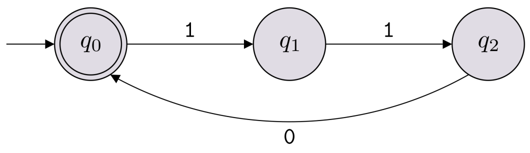


2. Take the DFA above and flip the accepting and rejecting states.

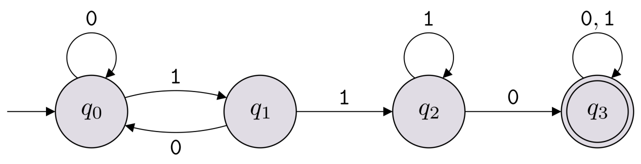
3.



4. Below, all missing transitions go to a rejecting sink state.



5.



■

Exercise (Finite languages can be solved by DFAs)**.** Let L be a finite language, i.e., it contains a finite number of words. At a high level, describe why there is a DFA solving L.

Solution. It is a good idea to first think about whether languages of size 1 are regular. Are languages of size 2 regular? When you draw DFAs for such languages, the basic idea is to "hard-code" the words in the language into the state-diagram of the DFA (this is what we did for part 1 of Exercise (Draw DFAs)). So for each word in the language, there would be a path in the DFA corresponding to that word that ends in an accepting state. This idea works whenever the language in consideration has a finite number of words. The number of states in the DFA will depend on the number of words in the language and the length of those words, but since the language is finite (and each word has finite-length), all the words in the language can be hard-coded using a finite number of states. ■

Definition (Regular language)**.** A language $L \subseteq \Sigma^*$ is called a *regular language* if there is a deterministic finite automaton M that solves L.

Example (Some examples of regular languages)**.** All the languages in Exercise (Draw DFAs) are regular languages.

Exercise (Equal number of 01's and 10's)**.** Let $\Sigma = \{0, 1\}$. Is the following language regular?

$$L = \{w \in \{0,1\}^* : w \text{ has an equal number of occurrences of } 01 \text{ and } 10 \text{ as substrings}\}$$

Hint. The language is regular. Go through some examples to see if you can notice a pattern and come up with an alternate description for the language.

Solution. The answer is yes because the language is pretty much the same as the language in Exercise (Draw DFAs), part 3 (except that the starting state should also be accepting). ■

Definition (Complexity class REG)**.** We denote by REG the set of all regular languages (over the default alphabet $\Sigma = \{0, 1\}$).

2 Non-Regular Languages

Theorem (0^n1^n is not regular)**.** *Let $\Sigma = \{0, 1\}$. The language $L = \{0^n1^n : n \in \mathbb{N}\}$ is **not** regular.*

Proof. Our goal is to show that $L = \{0^n1^n : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let's assume that L is regular.

Since L is regular, by definition, there is some deterministic finite automaton M that solves L. Let k denote the number of states of M. Consider the following set of $k + 1$

strings: $P = \{\texttt{0}^n : n \in \{0, 1, \ldots, k\}\}$. Each string in P, when fed as input to M, ends up in one of the k states of M. By the pigeonhole principle[2] (thinking of the strings as pigeons and the states as holes), we know that there must be two strings in P that end up in the same state. In other words, there are $i, j \in \{0, 1, \ldots, k\}$, with $i \neq j$, such that the string $\texttt{0}^i$ and the string $\texttt{0}^j$ end up in the same state. This implies that for any string $w \in \{\texttt{0}, \texttt{1}\}^*$, $\texttt{0}^i w$ and $\texttt{0}^j w$ end up in the same state. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $\texttt{0}^i w$ and $\texttt{0}^j w$ end up in different states.

Consider the string $w = \texttt{1}^i$. Then since M solves L, we know $\texttt{0}^i w = \texttt{0}^i \texttt{1}^i$ must end up in an accepting state. On the other hand, since $i \neq j$, $\texttt{0}^j w = \texttt{0}^j \texttt{1}^i$ is not in the language, and therefore cannot end up in an accepting state. This is the desired contradiction. □

Exercise (Would any set of pigeons work?)**.** In the proof of the above theorem, we defined the set $P = \{\texttt{0}^n : n \in \{0, 1, \ldots, k\}\}$ and then applied the pigeonhole principle. Explain why picking the following sets would not have worked in that proof.

1. $P = \{\texttt{1}^n : n \in \{1, 2, \ldots, k+1\}\}$
2. $P = \{\texttt{1}, \texttt{11}, \texttt{000}, \texttt{0000}, \texttt{00000}, \ldots, \texttt{0}^{k+1}\}$

Solution. 1. With $P = \{\texttt{1}^n : n \in \{1, 2, \ldots, k+1\}\}$ we can still apply the pigeonhole principle to conclude that there are two strings $\texttt{1}^i$ and $\texttt{1}^j$, $i \neq j$, that must end up in the same state. However, to reach a contradiction, we need to find a string $w \in \{\texttt{0}, \texttt{1}\}^*$ such that exactly one of $\texttt{1}^i w$ and $\texttt{1}^j w$ is in the language, and the other is not. On the other hand, any string starting with a `1` is not in the language.

2. The argument is the same as above, with the key observation being the following. When we apply the pigeonhole principle to conclude that two strings in P must end up in the same state, we have no control over which two strings in P end up in the same state. Therefore, our argument must work no matter which two strings in P end up in the same state. For the P given to us, the two strings that end up in the same state could be `1` and `11`, and so we would run into the same problem as in the previous part.

■

Exercise ($\texttt{c}^{251}\texttt{a}^n\texttt{b}^{2n}$ is not regular)**.** Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Prove that $L = \{\texttt{c}^{251}\texttt{a}^n\texttt{b}^{2n} : n \in \mathbb{N}\}$ is not regular.

Solution. Our goal is to show that $L = \{\texttt{c}^{251}\texttt{a}^n\texttt{b}^{2n} : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let's assume that L is regular.

Since L is regular, by definition, there is some deterministic finite automaton M that solves L. Let k denote the number of states of M. For $n \in \mathbb{N}$, let r_n denote the state that M reaches after reading $\texttt{c}^{251}\texttt{a}^n$. By the pigeonhole principle, we know that there must be a repeat among $r_0, r_1, \ldots, r_k$. In other words, there are $i, j \in \{0, 1, \ldots, k\}$ with $i \neq j$ such that $r_i = r_j$. This means that the string $\texttt{c}^{251}\texttt{a}^i$ and the string $\texttt{c}^{251}\texttt{a}^j$ end up in the same state in M. Therefore, $\texttt{c}^{251}\texttt{a}^i w$ and $\texttt{c}^{251}\texttt{a}^j w$, *for any* string $w \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^*$, end up in the same state in M. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $\texttt{c}^{251}\texttt{a}^i w$ and $\texttt{c}^{251}\texttt{a}^j w$ end up in different states.

Consider the string $w = \texttt{b}^{2i}$. Then since M solves L, we know $\texttt{c}^{251}\texttt{a}^i w = \texttt{c}^{251}\texttt{a}^i\texttt{b}^{2i}$ must end up in an accepting state. On the other hand, since $i \neq j$, $\texttt{c}^{251}\texttt{a}^j w = \texttt{c}^{251}\texttt{a}^j\texttt{b}^{2i}$ is not in the language, and therefore cannot end up in an accepting state. This is the desired contradiction. ■

[2] The *pigeonhole principle* states that if n items are put inside m containers, and $n > m$, then there must be at least one container with more than one item. The name *pigeonhole principle* comes from thinking of the items as pigeons, and the containers as holes. The pigeonhole principle is often abbreviated as PHP.

Exercise (A fooling pigeon set)**.** In Exercise (Would any set of pigeons work?) we saw that the "pigeon set" that we use to apply the pigeonhole principle must be chosen carefully. We'll call a pigeon set a *fooling pigeon set* if it is a pigeon set that "works". That is, given a DFA with k states that is assumed to solve a non-regular L, a fooling pigeon set of size $k+1$ allows us to carry out the contradiction proof, and conclude that L is non-regular. Identify the property that a fooling pigeon set should have.

Solution. A fooling pigeon set P is such that **for all** $x, y \in P$, there exists a $z \in \Sigma^*$ such that exactly one of xz and yz is in L, and the other is not. (Note that the choice of z for different pairs x and y may be different.)

The crux of a non-regularity proof is to show that for any $k \in \mathbb{N}$ (which denotes the number of states of a DFA that is assumed to solve L), there is a fooling pigeon set of size at least $k+1$. Since k is arbitrary, showing that a language L is non-regular really amounts to finding an infinite-size fooling pigeon set. In the case of the language $L = \{\texttt{0}^n\texttt{1}^n : n \in \mathbb{N}\}$, the infinite-size fooling pigeon set is $\{\texttt{0}^n : n \in \mathbb{N}\}$. ■

Theorem (A unary non-regular language)**.** *Let $\Sigma = \{\texttt{a}\}$. The language $L = \{\texttt{a}^{2^n} : n \in \mathbb{N}\}$ is **not** regular.*

Proof. Our goal is to show that $L = \{\texttt{a}^{2^n} : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let's assume that L is regular.

Since L is regular, by definition, there is some deterministic finite automaton M that solves L. Let k denote the number of states of M. For $n \in \mathbb{N}$, let r_n denote the state that M reaches after reading $\texttt{a}^{2^n}$ (i.e. $r_n = \delta(q_0, \texttt{a}^{2^n})$). By the pigeonhole principle, we know that there must be a repeat among $r_0, r_1, \ldots, r_k$ (a sequence of $k+1$ states). In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i < j$ such that $r_i = r_j$. This means that the string $\texttt{a}^{2^i}$ and the string $\texttt{a}^{2^j}$ end up in the same state in M. Therefore, $\texttt{a}^{2^i}w$ and $\texttt{a}^{2^j}w$, *for any* string $w \in \{\texttt{a}\}^*$, end up in the same state in M. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $\texttt{a}^{2^i}w$ ends up in an accepting state but $\texttt{a}^{2^j}w$ ends up in a rejecting state (i.e. they end up in different states).

Consider the string $w = \texttt{a}^{2^i}$. Then $\texttt{a}^{2^i}w = \texttt{a}^{2^i}\texttt{a}^{2^i} = \texttt{a}^{2^{i+1}}$, and therefore must end up in an accepting state. On the other hand, $\texttt{a}^{2^j}w = \texttt{a}^{2^j}\texttt{a}^{2^i} = \texttt{a}^{2^j+2^i}$. We claim that this word must end up in a rejecting state because $2^j + 2^i$ cannot be written as a power of 2 (i.e., cannot be written as 2^t for some $t \in \mathbb{N}$). To see this, note that since $i < j$, we have

$$2^j < 2^j + 2^i < 2^j + 2^j = 2^{j+1},$$

which implies that if $2^j + 2^i = 2^t$, then $j < t < j+1$ (which is impossible). So $2^j + 2^i$ cannot be written as 2^t for $t \in \mathbb{N}$, and therefore $\texttt{a}^{2^j+2^i}$ leads to a reject state in M as claimed. □

Note (Lower-bounding the number of states)**.** In the next exercise, we'll write the proof in a slightly different way to offer a slightly different perspective. In particular, we'll phrase the proof such that the goal is to show no DFA solving L can have a finite number of states. This is done by identifying an infinite set of strings that all must end up in a different state (this is the fooling pigeon set that we defined in Exercise (A fooling pigeon set)). Once you have a set of strings S such that every string in S must end up in a different state, we can conclude any DFA solving the language must have at least $|S|$ states.

This slight change in phrasing of the non-regularity proof makes it clear that even if L is regular, the technique can be used to prove a lower bound on the number of states of any DFA solving L.

Exercise ($\texttt{a}^n\texttt{b}^n\texttt{c}^n$ is not regular)**.** Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Prove that $L = \{\texttt{a}^n\texttt{b}^n\texttt{c}^n : n \in \mathbb{N}\}$ is not regular.

Solution. Our goal is to show that $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \in \mathbb{N}\}$ is not regular. Consider the set of strings $P = \{\mathtt{a}^n : n \in \mathbb{N}\}$. We claim that in any DFA solving L, no two strings in P can end up in the same state. To prove this claim, let's go by contradiction, and assume that there are two strings in P, $\mathtt{a}^i$ and $\mathtt{a}^j$, $i \neq j$, that end up in the same state. Then for any $w \in \Sigma^*$, the strings $\mathtt{a}^i w$ and $\mathtt{a}^j w$ must end up in the same state. But for $w = \mathtt{b}^i\mathtt{c}^i$, $\mathtt{a}^i w = \mathtt{a}^i\mathtt{b}^i\mathtt{c}^i$ must end up in an accepting state, whereas $\mathtt{a}^j w = \mathtt{a}^j\mathtt{b}^i\mathtt{c}^i$ must end up in a rejecting state. This is a contradiction.

Since we have identified an infinite set of strings that all must end up in a different state, we conclude that there cannot be a DFA solving L, since by definition, DFAs have a finite number of states. ■

3 Closure Properties of Regular Languages

In this section we will be interested in the following question. Given regular languages, what operations can we apply to them (e.g. union, intersection, concatenation, etc.) so that the resulting language is also regular?

Exercise (Are regular languages closed under complementation?)**.** Is it true that if L is regular, then its complement $\Sigma^* \setminus L$ is also regular? In other words, are regular languages *closed* under the complementation operation?

Hint. The answer is yes. How can you modify the DFA for L to construct a DFA for $\Sigma^* \setminus L$?

Solution. Yes. If L is regular, then there is a DFA $M = (Q, \Sigma, \delta, q_0, F)$ solving L. The complement of L is solved by the DFA $M = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Take a moment to observe that this exercise allows us to say that a language is regular *if and only if* its complement is regular. Equivalently, a language is not regular if and only if its complement is not regular. ■

Exercise (Are regular languages closed under subsets?)**.** Is it true that if $L \subseteq \Sigma^*$ is a regular language, then any $L' \subseteq L$ is also a regular language?

Hint. The answer is no. Try to think of a counter example.

Solution. No. For example, $L = \Sigma^*$ is a regular language (construct a single state DFA in which the state is accepting). On the other hand, by Theorem ($\mathtt{0}^n\mathtt{1}^n$ is not regular), $\{\mathtt{0}^n\mathtt{1}^n : n \in \mathbb{N}\} \subseteq \{\mathtt{0}, \mathtt{1}\}^*$ is not regular. ■

Theorem (Regular languages are closed under union)**.** *Let Σ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cup L_2$ is also regular.*

Proof. Given regular languages L_1 and L_2, we want to show that $L_1 \cup L_2$ is regular. Since L_1 and L_2 are regular languages, by definition, there are DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ that solve L_1 and L_2 respectively (i.e. $L(M) = L_1$ and $L(M') = L_2$). To show $L_1 \cup L_2$ is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q''_0, F'')$ that solves $L_1 \cup L_2$. The definition of M'' will make use of M and M'.

The idea behind the construction of M'' is as follows. To figure out if a string w is in $L_1 \cup L_2$, we can run $M(w)$ and $M'(w)$ to see if at least one of them accepts. However, that would require scanning w twice, which is something a DFA cannot do. Therefore, the main trick is to simulate both $M(w)$ and $M'(w)$ simultaneously. For this, we can imagine having one thread for $M(w)$ and another thread for $M'(w)$ that we run together. We can write the computation paths corresponding to the threads as follows.

		w_1	w_2	w_3	$\dots$	w_n
thread 1:	r_0	r_1	r_2	r_3	$\dots$	r_n
thread 2:	s_0	s_1	s_2	s_3	$\dots$	s_n

Here, thread 1 represents a computation path for M and thread 2 represents a computation path for M'. We want the above to represent the computation path of a single DFA M'' (and we want to know if one of the threads is an accepting computation path). We can accomplish this by viewing the combination of states (r_i, s_i) as a single state of M''. And when we read a symbol w_{i+1}, we update r_i according to the transition function of M, and we update s_i according to the transition function of M'. In more detail:

- The set of states is $Q'' = Q \times Q' = \{(q, q') : q \in Q, q' \in Q'\}$. (Note that in the definition of a DFA, we can use any fixed finite set as the set of states. It does not matter if a state is represented as a tuple or some other object; you could rename them to be of the form q''_i if you wanted.)
- The transition function δ'' is defined such that for all $(q, q') \in Q''$ and for all $\sigma \in \Sigma$,
$$\delta''((q, q'), \sigma) = (\delta(q, \sigma), \delta'(q', \sigma)).$$
(Note that for $w \in \Sigma^*$, $\delta''((q, q'), w) = (\delta(q, w), \delta'(q', w))$.)
- The initial state is $q''_0 = (q_0, q'_0)$.
- The set of accepting states is $F'' = \{(q, q') : q \in F \text{ or } q' \in F'\}$.

This completes the definition of M''.[3] It remains to show that M'' indeed solves the language $L_1 \cup L_2$, i.e. $L(M'') = L_1 \cup L_2$. We will first argue that $L_1 \cup L_2 \subseteq L(M'')$ and then argue that $L(M'') \subseteq L_1 \cup L_2$. Both inclusions will follow easily from the definition of M'' and the definition of a DFA accepting a string.

$L_1 \cup L_2 \subseteq L(M'')$: Suppose $w \in L_1 \cup L_2$, which means w either belongs to L_1 or it belongs to L_2. Our goal is to show that $w \in L(M'')$. Without loss of generality, assume w belongs to L_1, or in other words, M accepts w (the argument is essentially identical when w belongs to L_2). So we know that $\delta(q_0, w) \in F$. By the definition of δ'', $\delta''((q_0, q'_0), w) = (\delta(q_0, w), \delta'(q'_0, w))$. And since $\delta(q_0, w) \in F$, $(\delta(q_0, w), \delta'(q'_0, w)) \in F''$ (by the definition of F''). So w is accepted by M'' as desired.

$L(M'') \subseteq L_1 \cup L_2$: Suppose that $w \in L(M'')$. Our goal is to show that $w \in L_1$ or $w \in L_2$. Since w is accepted by M'', we know that $\delta''((q_0, q'_0), w) = (\delta(q_0, w), \delta'(q'_0, w)) \in F''$. By the definition of F'', this means that either $\delta(q_0, w) \in F$ or $\delta'(q'_0, w) \in F'$, i.e., w is accepted by M or M'. This implies that either $w \in L(M) = L_1$ or $w \in L(M') = L_2$, as desired. □

Corollary (Regular languages are closed under intersection)**.** *Let Σ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cap L_2$ is also regular.*

Proof. We want to show that regular languages are closed under the intersection operation. We know that regular languages are closed under union and closed under complementation. The result then follows since $A \cap B = \overline{\overline{A} \cup \overline{B}}$. □

Exercise (Direct proof that regular languages are closed under difference)**.** Give a direct proof (without using the fact that regular languages are closed under complementation, union and intersection) that if L_1 and L_2 are regular languages, then $L_1 \setminus L_2$ is also regular.

[3] At this point, it would be reasonable to end the proof and say that the construction of M'' is correct based on our discussion earlier that motivates the construction. It is also reasonable to spell out the proof of correctness, as we do here. In situations like these, in an introductory course like this one, we recommend you err on the side of caution and spell out the correctness proof.

Solution. The proof is very similar to the proof of Theorem (Regular languages are closed under union). The only difference is the definition of F'', which now needs to be defined as

$$F'' = \{(q, q') : q \in F \text{ and } q' \in Q' \setminus F'\}.$$

The argument that $L(M'') = L(M) \setminus L(M')$ needs to be slightly adjusted in order to agree with F''. ■

Exercise (Finite vs infinite union)**.**

1. Suppose $L_1, \ldots, L_k$ are all regular languages. Is it true that their union $\bigcup_{i=0}^{k} L_i$ must be a regular language?
2. Suppose $L_0, L_1, L_2, \ldots$ is an infinite sequence of regular languages. Is it true that their union $\bigcup_{i\geq 0} L_i$ must be a regular language?

Hint. The first one is "yes" (think induction) and the second one is "no" (give a counter-example).

Solution. In part 1, we are asking whether a finite union of regular languages is regular. The answer is yes, and this can be proved using induction, where the base case corresponds to a single regular language, and the induction step corresponds to Theorem (Regular languages are closed under union). In part 2, we are asking whether a countably infinite union of regular languages is regular. The answer is no. First note that any language of cardinality 1 is regular, i.e., $\{w\}$ for any $w \in \Sigma^*$ is a regular language. In particular, for any $n \in \mathbb{N}$, the language $L_n = \{\texttt{0}^n\texttt{1}^n\}$ of cardinality 1 is regular. But

$$\bigcup_{n\geq 0} L_n = \{\texttt{0}^n\texttt{1}^n : n \in \mathbb{N}\}$$

is not regular. ■

Exercise (Union of non-regular languages)**.** Suppose L_1 and L_2 are not regular languages. Is it always true that $L_1 \cup L_2$ is not a regular language?

Hint. No. Come up with a counter-example. You can try to find one such that $L_1 \cup L_2$ is Σ^*.

Solution. The answer is no. Consider $L = \{\texttt{0}^n\texttt{1}^n : n \in \mathbb{N}\}$, which is a non-regular language. Furthermore, the complement of L, which is $\overline{L} = \Sigma^* \setminus L$, is non-regular. This is because regular languages are closed under complementation (Exercise (Are regular languages closed under complementation?)), so if $\overline{L}$ was regular, then $\overline{\overline{L}} = L$ would also have to be regular. The union of L and $\overline{L}$ is Σ^*, which is a regular language. ■

Exercise (Regularity of suffixes and prefixes)**.** Suppose $L \subseteq \Sigma^*$ is a regular language. Show that the following languages are also regular:

$$\text{SUFFIXES}(L) = \{x \in \Sigma^* : yx \in L \text{ for some } y \in \Sigma^*\},$$
$$\text{PREFIXES}(L) = \{y \in \Sigma^* : yx \in L \text{ for some } x \in \Sigma^*\}.$$

Solution. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA solving L. We define $S \subseteq Q$ to be the set of states that are "reachable" starting from the initial state q_0. More precisely,

$$S = \{s \in Q : \exists y \in \Sigma^* \text{ such that } \delta(q_0, y) = s\}.$$

Define a DFA for each $s \in S$ as follows: $M_s = (Q, \Sigma, \delta, s, F)$, so the only difference between M_s and M is the starting/initial state. We claim that

$$\text{SUFFIXES}(L) = \bigcup_{s\in S} L(M_s).$$

We now prove this equality by a double containment argument.

First, if $x \in$ SUFFIXES(L) then you know that there is some $y \in \Sigma^*$ such that $yx \in L$. So $M(yx)$ accepts. Note that this y, when fed into M, ends up in some state. Let's call that state s. Then M_s accepts x, i.e. $x \in L(M_s)$. And therefore $x \in \bigcup_{s \in S} L(M_s)$.

On the other hand suppose $x \in \bigcup_{s \in S} L(M_s)$. Then there is some state $s \in S$ such that $x \in L(M_s)$. By the definition of S, there is some string y such that $M(y)$ ends up in state s. Since x is accepted by M_s, we have that yx is accepted by M, i.e. $yx \in L$. By the definition of SUFFIXES(L), this implies $x \in$ SUFFIXES(L). This concludes the proof of the claim.

Since $L(M_s)$ is regular for all $s \in S$ and S is a finite set, using Exercise (Finite vs infinite union) part 1, we can conclude that SUFFIXES(L) is regular.

For the second part, define the set

$$R = \{r \in Q : \exists x \in \Sigma^* \text{ such that } \delta(r, x) \in F\}.$$

Now we can define the DFA $M_R = (Q, \Sigma, \delta, q_0, R)$. Observe that this DFA solves PREFIXES(L), which shows that PREFIXES(L) is regular. ■

Theorem (Regular languages are closed under concatenation). *If $L_1, L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 L_2$ is also regular.*

Proof. Given regular languages L_1 and L_2, we want to show that $L_1 L_2$ is regular. Since L_1 and L_2 are regular languages, by definition, there are DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ that solves L_1 and L_2 respectively. To show $L_1 L_2$ is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ that solves $L_1 L_2$. The definition of M'' will make use of M and M'.

Before we formally define M'', we will introduce a few key concepts and explain the intuition behind the construction.

We know that $w \in L_1 L_2$ if and only if there is a way to write w as uv where $u \in L_1$ and $v \in L_2$. With this in mind, we make the following definition. Given a word $w = w_1 w_2 \ldots w_n \in \Sigma^*$, a *concatenation thread* with respect to w is a sequence of states

$$r_0, r_1, r_2, \ldots, r_i, s_{i+1}, s_{i+2}, \ldots, s_n,$$

where $r_0, r_1, \ldots, r_i$ is an accepting computation path of M with respect to $w_1 w_2 \ldots w_i$, and $q_0', s_{i+1}, s_{i+2}, \ldots, s_n$ is a computation path (not necessarily accepting) of M' with respect to $w_{i+1} w_{i+2} \ldots w_n$. A concatenation thread like this corresponds to simulating M on $w_1 w_2 \ldots w_i$ (at which point we require that an accepting state of M is reached), and then simulating M' on $w_{i+1} w_{i+2} \ldots w_n$. So a concatenation thread is really a concatenation of a thread in M with a thread in M'.

For each way of writing w as uv where $u \in L_1$, there is a corresponding concatenation thread for it. Note that $w \in L_1 L_2$ if and only if there is a concatenation thread in which $s_n \in F'$. Our goal is to construct the DFA M'' such that it keeps track of all possible concatenation threads, and if one of the threads ends with a state in F', then M'' accepts.

At first, it might seem like one cannot keep track of all possible concatenation threads using only *constant* number of states. However, this is not the case. Let's identify a concatenation thread with its sequence of s_j's (i.e. the sequence of states from Q' corresponding to the simulation of M'). Consider two concatenation threads (for the sake of example, let's take $n = 10$):

$$\begin{array}{r} s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10} \\ s_5', s_6', s_7', s_8', s_9', s_{10}' \end{array}$$

If, say, $s_i = s_i' = q' \in Q'$ for some i, then $s_j = s_j'$ for all $j > i$ (in particular, $s_{10} = s_{10}'$). At the end, all we care about is whether s_{10} or s_{10}' is an accepting state of M'. So at index i, we do not need to remember that there are two copies of q'; it suffices to keep track of one copy. In general, at any index i, when we look at all the possible concatenation threads, we want to keep track of the unique states that appear at that index, and not worry about duplicates. Since we do not need to keep track of duplicated states, what we need to remember is a *subset* of Q' (recall that a set cannot have duplicated elements).

The construction of M'' we present below keeps track of all the concatenation threads using a constant number of states. The strategy is to keep a single thread for the machine M, and then separate threads for machine M' that will correspond to the M' portions of the different concatenation threads. So the set of states is[4]

$$Q'' = Q \times \wp(Q') = \{(q, S) : q \in Q, S \subseteq Q'\},$$

where the first component keeps track of the state we are at in M, and the second component keeps track of all the unique states of M' that we can be at if we are following one of the possible concatenation threads.

Before we present the formal definition of M'', we introduce one more definition. Recall that the transition function of M' is $\delta' : Q' \times \Sigma \to Q'$. Using δ' we define a new function $\delta'_{\wp} : \wp(Q') \times \Sigma \to \wp(Q')$ as follows. For $S \subseteq Q'$ and $\sigma \in \Sigma$, $\delta'_{\wp}(S, \sigma)$ is defined to be the set of all possible states that we can end up at if we start in a state in S and read the symbol σ. In other words,

$$\delta'_{\wp}(S, \sigma) = \{\delta'(q', \sigma) : q' \in S\}.$$

It is appropriate to view $\delta'_{\wp}$ as an extension/generalization of δ'.

Here is the formal definition of M'':

- The set of states is $Q'' = Q \times \wp(Q') = \{(q, S) : q \in Q, S \subseteq Q'\}$.

  (The first coordinate keeps track of which state we are at in the first machine M, and the second coordinate keeps track of the set of states we can be at in the second machine M' if we follow one of the possible concatenation threads.)

- The transition function δ'' is defined such that for $(q, S) \in Q''$ and $\sigma \in \Sigma$,

$$\delta''((q, S), \sigma) = \begin{cases} (\delta(q, \sigma), \delta'_{\wp}(S, \sigma)) & \text{if } \delta(q, \sigma) \notin F, \\ (\delta(q, \sigma), \delta'_{\wp}(S, \sigma) \cup \{q'_0\}) & \text{if } \delta(q, \sigma) \in F. \end{cases}$$

  (The first coordinate is updated according to the transition rule of the first machine. The second coordinate is updated according to the transition rule of the second machine. Since for the second machine, we are keeping track of all possible states we could be at, the generalized transition function $\delta'_{\wp}$ gives us all possible states we can go to when reading a character σ. Note that if after applying δ to the first coordinate, we get a state that is an accepting state of the first machine, a new thread in M' must be created, which corresponds to a new concatenation thread that we need to keep track of. This is accomplished by adding q'_0 to the second coordinate.)

- The initial state is

$$q''_0 = \begin{cases} (q_0, \varnothing) & \text{if } q_0 \notin F, \\ (q_0, \{q'_0\}) & \text{if } q_0 \in F. \end{cases}$$

  (Initially, if $q_0 \notin F$, then there are no concatenation threads to keep track of, so the second coordinate is the empty set. On the other hand, if $q_0 \in F$, then there is already a concatenation thread that we need to keep track of – the one corresponding to running the whole input word w on the second machine – so we add q'_0 to the second coordinate to keep track of this thread.)

- The set of accepting states is $F'' = \{(q, S) : q \in Q, S \subseteq Q', S \cap F' \neq \varnothing\}$.

  (In other words, M'' accepts if and only if there is a state in the second coordinate that is an accepting state of the second machine M'. So M'' accepts if and only if one of the possible concatenation threads ends in an accepting state of M'.)

[4]Recall that for any set Q, the set of all subsets of Q is called the *power set* of Q, and is denoted by $\wp(Q)$.

This completes the definition of M''. To see that M'' indeed solves the language L_1L_2, i.e. $L(M'') = L_1L_2$, note that by construction, M'' with input w, does indeed keep track of all the possible concatenation threads. And it accepts w if and only if one of those concatenation threads ends in an accepting state of M'. The result follows since $w \in L_1L_2$ if and only if there is a concatenation thread with respect to w that ends in an accepting state of M'. □

We defined a generalized transition function in the proof of the above theorem. This is a useful definition and we will be using it multiple times in what comes next. Therefore we repeat the definition below.

Definition (Generalized transition function)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We define the *generalized transition function* $\delta_\wp : \wp(Q) \times \Sigma \to \wp(Q)$ as follows. For $S \subseteq Q$ and $\sigma \in \Sigma$,

$$\delta_\wp(S, \sigma) = \{\delta(q, \sigma) : q \in S\}.$$

Exercise (Regular languages are closed under concatenation - another construction)**.** In the proof of Theorem (Regular languages are closed under concatenation), we defined the set of states for the DFA solving L_1L_2 as $Q'' = Q \times \wp(Q')$. Construct a DFA for L_1L_2 in which Q'' is equal to $\wp(Q \cup Q')$, i.e., specify how δ'', q_0'' and F'' should be defined with respect to $Q'' = \wp(Q \cup Q')$. Proof of correctness is not required.

Solution. As before, we have DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ solving L_1 and L_2 respectively. The construction for L_1L_2 is as follows. We are given that the set of states is

$$Q'' = \wp(Q \cup Q').$$

To define the transition function, let $S'' \in Q'' = \wp(Q \cup Q')$ and $\sigma \in \Sigma$. Write S'' as $S \cup S'$, where $S \subseteq Q$ and $S' \subseteq Q'$. Then,

$$\delta''(S \cup S', \sigma) = \begin{cases} \delta_\wp(S, \sigma) \cup \delta'_\wp(S', \sigma) \cup \{q_0'\} & \text{if } \delta_\wp(S, \sigma) \cap F \neq \varnothing, \\ \delta_\wp(S, \sigma) \cup \delta'_\wp(S', \sigma) & \text{otherwise.} \end{cases}$$

The initial state is

$$q_0'' = \begin{cases} \{q_0, q_0'\} & \text{if } q_0 \in F, \\ \{q_0\} & \text{if } q_0 \notin F. \end{cases}$$

And the set of accepting states is

$$F'' = \{S \in Q'' : S \cap F' \neq \varnothing\}.$$

Note that this construction is not really much different from the construction given in the proof of Theorem (Regular languages are closed under concatenation) because the way the initial state and the transition function is defined, a state S'' will always have a single element from Q. So one can view S'' as an element of $Q \times \wp(Q')$. ■

Exercise (Regular languages are closed under star - wrong proof)**.** Critique the following argument that claims to establish that regular languages are closed under the star operation, that is, if L is a regular language, then so is L^*.

> Let L be a regular language. We know that by definition $L^* = \bigcup_{n \in \mathbb{N}} L^n$, where
>
> $$L^n = \{u_1u_2 \ldots u_n : u_i \in L \text{ for all } i\}.$$
>
> We know that for all n, L^n must be regular using Theorem (Regular languages are closed under concatenation). And since L^n is regular for all n, we know L^* must be regular using Theorem (Regular languages are closed under union).

Solution. It is true that using induction, we can show that L^n is regular for all n. However, from there, we cannot conclude that $L^* = \bigcup_{n\in\mathbb{N}} L^n$ is regular. Even though regular languages are closed under finite unions, they are not closed under infinite unions. See Exercise (Finite vs infinite union). ■

Exercise (Regular languages are closed under star - correct proof)**.** Show that regular languages are closed under the star operation as follows: First show that if L is regular, then so is L^+, which is defined as the union

$$L^+ = \bigcup_{n\in\mathbb{N}^+} L^n.$$

For this part, given a DFA for L, show how to construct a DFA for L^+. A proof of correctness is not required. In order to conclude that L^* is regular, observe that $L^* = L^+ \cup \{\epsilon\}$, and use the fact that regular languages are closed under union.

Solution. We construct a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ solving L^+ using a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that solves L. The construction is as follows.

$$Q' = \wp(Q).$$

So the elements of Q' are subsets of Q. To define the transition function, for any $S \subseteq Q$ and any $\sigma \in \Sigma$, let

$$\delta'(S,\sigma) = \begin{cases} \delta_\wp(S,\sigma) \cup \{q_0\} & \text{if } \delta_\wp(S,\sigma) \cap F \neq \varnothing, \\ \delta_\wp(S,\sigma) & \text{otherwise.} \end{cases}$$

The initial state is $q_0' = \{q_0\}$. And the set of accepting states is

$$F' = \{S \subseteq Q : S \cap F \neq \varnothing\}.$$

■

4 Bonus: Non-Deterministic FA

To be added.

5 Check Your Understanding

Problem.
1. What are the 5 components of a DFA?
2. Let M be a DFA. What does $L(M)$ denote?
3. Draw a DFA solving $\varnothing$. Draw a DFA solving Σ^*.
4. True or false: Given a language L, there is at most one DFA that solves it.
5. Fix some alphabet Σ. How many DFAs are there with exactly one state?
6. True or false: For any DFA, all the states are "reachable". That is, if $D = (Q, \Sigma, \delta, q_0, F)$ is a DFA and $q \in Q$ is one of its states, then there is a string $w \in \Sigma^*$ such that $\delta^*(q_0, w) = q$.
7. What is the set of all possible inputs for a DFA $D = (Q, \Sigma, \delta, q_0, F)$?
8. Consider the set of all DFAs with k states over the alphabet $\Sigma = \{\mathtt{a}\}$ such that all states are reachable from q_0. What is the cardinality of this set?
9. What is the definition of a regular language?

10. Describe the general strategy (the high level steps) for showing that a language is not regular.

11. Give 3 examples of non-regular languages.

12. Let $L \subseteq \{\mathtt{a}\}^*$ be a language consisting of all strings of a's of odd length except length 251. Is L regular?

13. Let L be the set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that contain at least 251 0's and at most 251 1's. Is L regular?

14. Suppose you are given a regular language L and you are asked to prove that any DFA solving L must contain at least k states (for some k value given to you). What is a general proof strategy for establishing such a claim?

15. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA solving a language L and let $M' = (Q', \Sigma, \delta', q'_0, F')$ be a DFA solving a language L'. Describe the 5 components of a DFA solving $L \cup L'$.

16. True or false: Let $L_1 \oplus L_2$ denote the set of all words in either L_1 or L_2, but not both. If L_1 and L_2 are regular, then so is $L_1 \oplus L_2$.

17. True or false: For languages L and L', if $L \subseteq L'$ and L is non-regular, then L' is non-regular.

18. True or false: If $L \subseteq \Sigma^*$ is non-regular, then $\overline{L} = \Sigma^* \setminus L$ is non-regular.

19. True or false: If $L_1, L_2 \subseteq \Sigma^*$ are non-regular languages, then so is $L_1 \cup L_2$.

20. True or false: Let L be a non-regular language. There exists $K \subset L$, $K \neq L$, such that K is also non-regular.

21. By definition a DFA has finitely many states. What is the motivation/reason for this restriction?

22. Consider the following decision problem: Given as input a DFA, output True if and only if there exists some string $s \in \Sigma^*$ that the DFA accepts. Write the language corresponding to this decision problem using mathematical notation, and in particular, using set builder notation.

6 High-Order Bits

Important. Here are the important things to keep in mind from this chapter.

1. Given a DFA, describe in English the language that it solves.

2. Given the description of a regular language, come up with a DFA that solves it.

3. Given a non-regular language, prove that it is indeed non-regular. Make sure you are not just memorizing a template for these types of proofs, but that you understand all the details of the strategy being employed. Apply the Feynman technique and see if you can teach this kind of proof to someone else.

4. In proofs establishing a closure property of regular languages, you start with one or more DFAs, and you construct a new one from them. In order to build the new DFA successfully, you have to be comfortable with the 5-tuple notation of a DFA. The constructions can get notation-heavy (with power sets and Cartesian products), but getting comfortable with mathematical notation is one of our learning objectives.

5. With respect to closure properties of regular languages, prioritize having a very solid understanding of closure under the union operation before you move to the more complicated constructions for the concatenation operation and star operation.