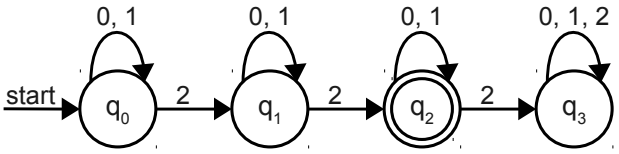
Problem Set 5 Solutions

Problem One: Constructing DFAs (24 Points)

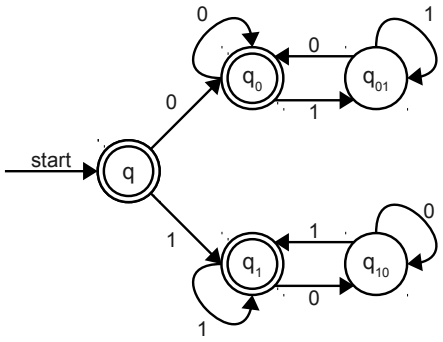
i. For the alphabet Σ = {`0`, `1`, `2`}, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ contains exactly two `2`s. $\}$



Here, the intuition is that each state represents having seen some number of copies of the `2`. q_0, q_1, and q_2 represent having seen zero, one, and two copies of `2`, respectively, while q_3 represents having seen more than two copies. q_3 is a dead state, because you cannot "unsee" one of the `2`s.

Why we asked this question: This question was designed to get you thinking about how to keep track of information with a DFA. Each state in the DFA represents some information the automaton needs to keep track of, and in this case those states correspond to the number of `2`'s seen.

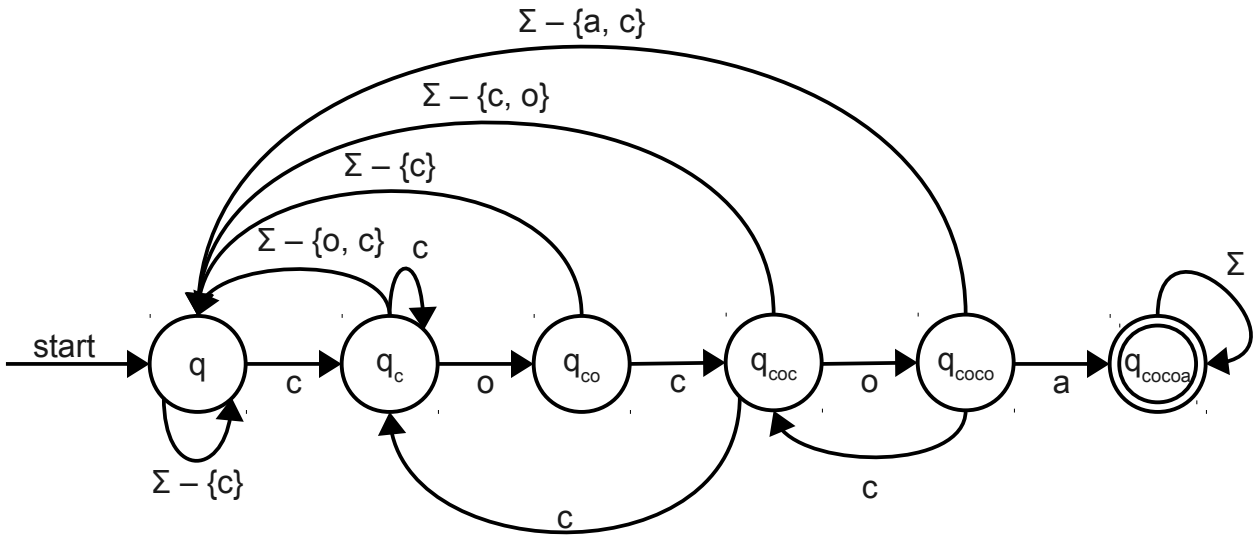
ii. For the alphabet Σ = {`0`, `1`}, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ contains the same number of instances of the substring `01` and the substring `10` $\}$



The intuition behind this solution is that a string of `0`s and `1`s can't have many more copies of `01` than `10` or of `10` than `01`. There can only be a difference of at most one, since if you have a string containing `01`, the only way to get another `01` to appear would be to introduce another `0`. This must follow the `1` from the `01`, which introduces another copy of `10`. Additionally, the first character of the string immediately tells you which substring (`01` or `10`), if any, will appear more frequently. This DFA see what the first character is, then checks the balance between the two substrings in the remainder of the string.

Why we asked this question: It's not at all obvious that this language is regular, especially since most languages that involve counting aren't. We wanted you to have the insight that you only need to store a difference of ±1.

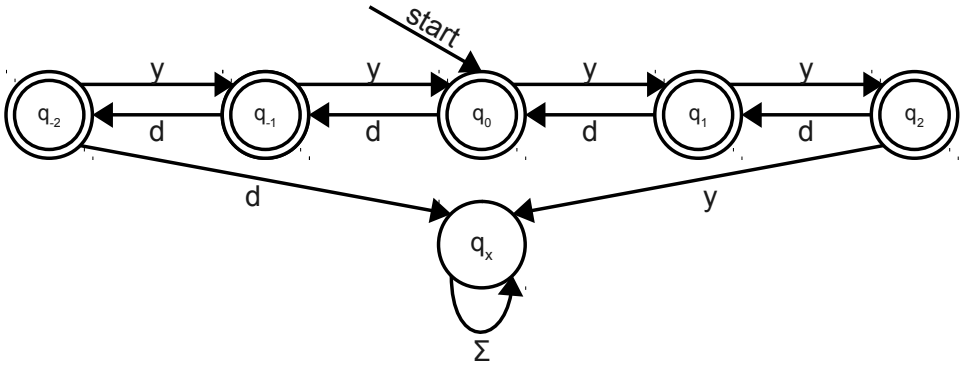
iii. For the alphabet Σ = {**a**, **b**, **c**, …, **z**}, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ contains the word "**cocoa**" as a substring $\}$. Remember that as a shorthand, you can specify multiple letters in a transition by using set operations on Σ (for example, Σ – {**a**, **b**})



The interesting trick of this DFA is noticing that if you read a **c** in state q_{coco}, then you do **not** transition back to the state q_c. Instead, you should transition back to state q_{coc}, since the **coc** that you have already read might be the real start of the string.

Why we asked this question: As mentioned in the handout, finite automata are often used in string-matching algorithms. It's a bit trickier than it might seem to build DFAs that match specific strings, especially if part of the string ends up being a prefix of the overall string.

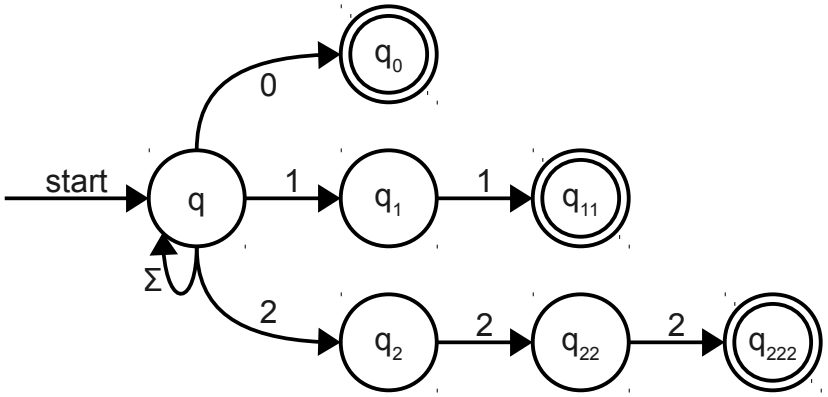
iv. Suppose that you are taking a walk with your dog along a straight-line path. Your dog is on a leash that has length two, meaning that the distance between you and your dog can be at most two units. You and your dog start at the same position. Consider the alphabet Σ = {**y**, **d**}. A string in Σ* can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string "**yydd**" means that you take two steps forward, then your dog takes two steps forward. Let $L = \{ w \in \Sigma^* \mid w$ describes a series of steps that ensures that you and your dog are never more than two units apart $\}$. Construct a DFA for *L*.



Why we asked this question: Again, we wanted to stress the intuition about DFAs as computers with finite memory. Each state here corresponds to some possible delta between your position and the dog's position.

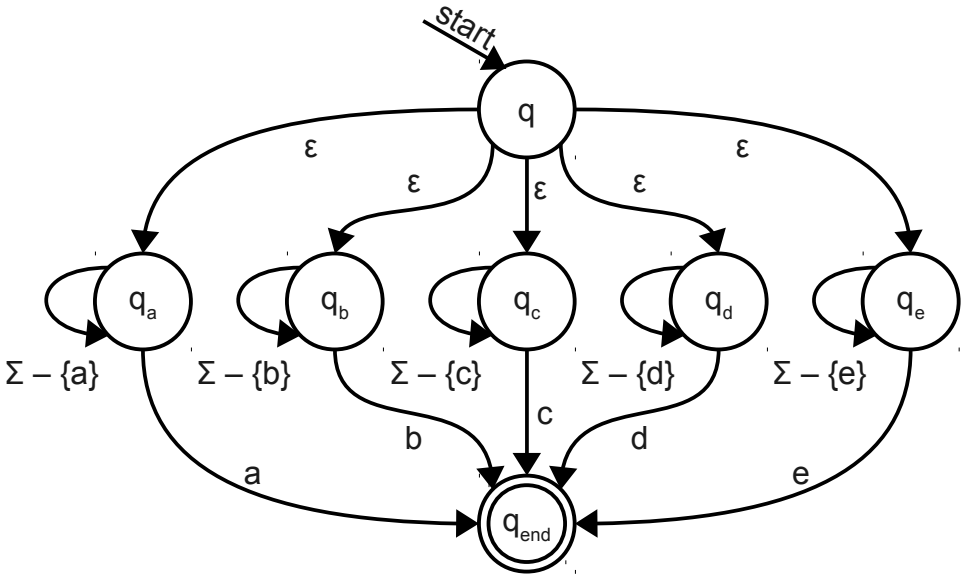
Problem Two: Constructing NFAs (20 Points)

i. For the alphabet Σ = {`0`, `1`, `2`}, construct an NFA for the language { *w* ∈ Σ* | *w* ends in `0`, `11`, or `222`. }



Why we asked this question: This is where nondeterminism really shines. The idea behind this machine is to sit in the start state waiting until the machine can guess that it's about to see the `0`, `11`, or `222` that will end the input. Since the machine is nondeterministic, we can just assume that it always knows how to guess correctly, so if the string does end this way, we can just transition to the appropriate branch.

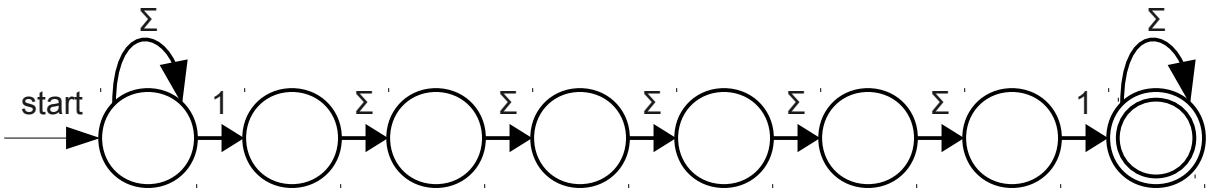
ii. For the alphabet Σ = {`a`, `b`, `c`, `d`, `e`}, construct an NFA for the language { *w* ∈ Σ* | the last character of *w* appears nowhere else in the string, and |*w*| ≥ 1 }



Here we see the power of ε-transitions. This automaton works by nondeterministically guessing which letter is going to be missing, then transitioning into a state that waits until the last character to nondeterministically transition into the accept state. As long as there's some choice the machine can make that accepts, the entire computation accepts.

Why we asked this question: If you were to try to construct a DFA for this language, it would have a huge number of states, since each state would need to track all letters that had been seen so far. We hoped that this would push you to build a nondeterministic machine.

iii. For the alphabet Σ = {**0**, **1**}, construct an NFA for the language { *w* ∈ Σ* | *w* contains two **1**'s with exactly five characters in-between them. } For example, **1000001** is in the language, as is **00100110100** and **01111010000**, but **11111** is not, nor are **11101** or **000101**.



The intuition here is to guess when we're going to see the **1** that will be matched with five intervening characters. If we do so correctly, we can end up in the accept state. If we do so incorrectly, that particular branch will fail.

Why we asked this question: Any DFA for this automaton would have to have a colossal number of states and it would be all but impossible to construct. This question was designed to force you to embrace nondeterminism.

Problem Three: Designing Regular Expressions (24 Points)

i. Let Σ = {**a**, **b**} and let *L* = { *w* ∈ Σ* | *w* does not contain **ba** as a substring }. Write a regular expression for *L*.

Since the alphabet here just contains **a** and **b**, any string that contains a **b** before an **a** will not be in the language *L* and any string that does not will be in *L*. Therefore, one regular expression is **a*b***.

Why we asked this question: There are many different ways of characterizing the same language. In this case, we wanted you to realize that excluding **ba** means that all **a**'s have to come before all **b**'s, at which point the regular expression **a*b*** makes sense.

ii. Let Σ = {**a**, **b**} and let *L* = { *w* ∈ Σ* | *w* does not contain **bb** as a substring }. Write a regular expression for *L*.

The main observation necessary to solve this problem is that if the string doesn't contain **bb** as a substring, then you can subdivide any string in the language into a number of "blocks" of **a**'s separated by **b**'s. Each **b** must be followed by at least one **a**, unless it's at the end of the string. This gives this regular expression:

a*(ba+)*b?

Here, we optionally can have any number of **a**'s at the start of the string. We then lay down blocks of **b**'s followed by one or more **a**'s, and finally conclude with an optional final **b**.

Why we asked this question: Often, designing a regular expression requires finding some way to build up a string in a language. Here, the observation about the strings in the language is that they can be partitioned into blocks delimited by **b**'s. We gave this problem to get you to see how to find the structure necessary and how to encode that as the regular expression.

iii. Let $\Sigma = \{\texttt{Y}, \texttt{D}\}$ and let $L = \{ w \in \Sigma^* \mid w$ represents a walk with your dog on a leash where you and your dog both end up at the same location $\}$. Write a regular expression for L.

We can think about this problem as follows: any string of this form can be broken down into a series of smaller walks, each of which ends up with you and your dog ending at exactly the same location. As a result, we can try to write a regular expression for smaller walks that balance out, then use the Kleene star to concatenate them together.

One possible option is

`((y(yd)*d) | (d(dy)*y))*`

Here, **`(y(yd)*d)`** means that if you take a step forward, there can be any number of iterations of you taking a second step forward, as long as your dog immediately goes forward. At the very end, to get back to the start, you need your dog to take another step forward. The other part of this regular expression is just the mirror image of this.

Why we asked this question: When building a DFA for the related language in Problem One, you could use multiple states to encode the difference between your position and the dog's position. When designing a regular expression, you cannot do this. Instead, you need to build the string up such that all mismatches are canceled out. We wanted you to see how DFA and regular expression representations of languages can often be quite different.

iv. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \neq \texttt{ab} \}$. Write a regular expression for L.

One possible option is

ε | a | b | aa | ba | bb | $\Sigma\Sigma^+$

This regular expression explicitly enumerates all strings of length two or less that aren't **`ab`** and also allows for any string of three or more characters.

Why we asked this question: It's easy to build a DFA for the complement of a language – just start out with a DFA for the language and flip the accepting and rejecting states. Regular expressions, however, don't support this, and to build a regular expression for a language of the form "everything except X" you need to explicitly show how to avoid anything that might produce X.

Problem Four: Finite and Cofinite Languages (16 Points)

i. Prove that any finite language is regular. *(Hint: Use induction.)*

Proof: By induction. Let $P(n)$ be "any language L where $|L| = n$" is regular. We prove $P(n)$ holds for all $n \in \mathbb{N}$.

For our base case, we prove $P(0)$, that any language L where $|L| = 0$ is regular. There is exactly one language with no strings in it, namely Ø, and Ø is regular because it is the language of the regular expression Ø.

For our inductive step, assume that for some $n \in \mathbb{N}$ that $P(n)$ holds and any language of cardinality n is regular. We prove $P(n + 1)$, that any language of cardinality $n + 1$ is regular. To do this, consider an arbitrary language of cardinality $n + 1$, call it L. Choose any string $w \in L$ and consider the language $L - \{w\}$. This language has cardinality n, so by our inductive hypothesis it is regular. Moreover, the language $\{w\}$ is regular, since it's the language of the regular expression w. Since regular languages are closed under union, $(L - \{w\}) \cup \{w\} = L$ is regular, as required. Thus $P(n + 1)$ holds, completing the induction. ■

Why we asked this question: We asked this question for a few reasons. First, the fact that all finite languages are regular is an extremely useful fact – as you'll see later in the quarter, it can help you build a better intuition for what problems are unsolvable, since any problem of the form "is the string one of the following n strings?" is regular and can be solved easily with a computer. Second, we wanted to give you additional practice with mathematical induction.

ii. Prove that any cofinite language is regular.

Proof: Consider any cofinite language L. By definition, $\overline{L}$ is a finite language, so it is regular. Since the regular languages are closed under complementation, this means $\overline{\overline{L}} = L$ is regular. Therefore, L is regular, as required. ■

Why we asked this question: We've discussed closure properties of regular languages before and wanted to give a question that required you to exercise those closure properties. One nuanced detail here is that the proof needs to start with L, show that $\overline{L}$ is regular, then conclude that $\overline{\overline{L}} = L$ is regular. Closure properties say that if you start with a regular language, you must end with a regular language as well, not that if you end with a regular language you must have started with one as well.

Problem Five: Testing Universality (20 Points)

Suppose you have a magic machine (an *oracle*) that takes as input two DFAs over the same alphabet Σ and returns whether every string is accepted by exactly one of the two DFAs. In other words, given two DFAs D_1 and D_2, the oracle reports whether $\mathscr{L}(D_1) = \Sigma^* - \mathscr{L}(D_2)$. You have no insight into how this oracle operates; from your perspective, it takes in two DFAs and then magically produces an answer. You cannot make any assumptions about how it works. What else could you do with this machine?

i. Suppose you have a DFA D over an alphabet Σ and want to determine whether $\mathscr{L}(D) = \Sigma^*$. Describe, in plain English, a procedure that uses the oracle to answer this question.

The procedure is the following. Let E be a DFA for Ø, which must exist since Ø is regular. Then, feed D and E into the oracle. If the oracle reports that $\mathscr{L}(D) = \Sigma^* - \mathscr{L}(E)$, return true. Otherwise, return false.

ii. Formally prove that your procedure from (i) is correct by proving the following: your procedure reports that $\mathscr{L}(D) = \Sigma^*$ iff it's actually the case that $\mathscr{L}(D) = \Sigma^*$. That is, prove that if $\mathscr{L}(D) = \Sigma^*$, then your procedure returns true and that if $\mathscr{L}(D) \neq \Sigma^*$, then your procedure returns false.

Proof: We prove both directions of implication.

($\Rightarrow$) First, we prove that if $\mathscr{L}(D) = \Sigma^*$, then the procedure reports true. Note that by definition the DFA E has language Ø. This means that $\Sigma^* - \mathscr{L}(E) = \Sigma^*$. Therefore, if $\mathscr{L}(D) = \Sigma^*$, it's true that $\mathscr{L}(D) = \Sigma^* = \Sigma^* - \mathscr{L}(E)$, so the procedure reports true.

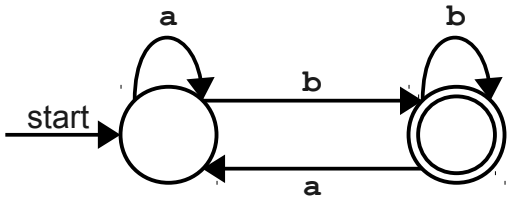
($\Leftarrow$) Next, we prove that if $\mathscr{L}(D) \neq \Sigma^*$, then the procedure reports false. Given our above logic, we know that $\mathscr{L}(D) = \Sigma^* - \mathscr{L}(E)$ iff $\mathscr{L}(D) = \Sigma^*$. Therefore, if $\mathscr{L}(D) \neq \Sigma^*$, we know $\mathscr{L}(D) \neq \Sigma^* - \mathscr{L}(E)$, so the procedure reports false. ■

Why we asked this question: This question is an example of a *reduction*. Given an oracle for one problem ("do two DFAs have complementary languages?"), we asked you to show that it would be possible to solve a different problem ("does D accept every string?"). Later in the quarter, we'll discuss reductions in more depth and how they let us find unsolvable problems.

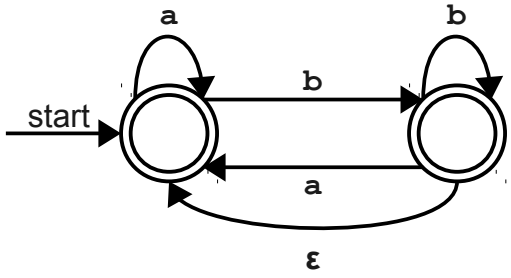
Problem Six: Why the Extra State? (12 Points)

Find a regular language L and an NFA N for L such that using the second construction does not create an NFA for L^*. Justify why the language of the new NFA isn't L^*.

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ ends with } \mathtt{b} \}$. Here's one possible NFA for L:



If we use the suggested construction, we get this machine:



This machine has language Σ^*, which is not the same as L^*. Note, for example, that $\mathtt{a} \in \Sigma^*$ but $\mathtt{a} \notin L^*$ because the string **a** cannot be decomposed into a series of smaller strings that all end with **b**.

Why we asked this question: The need for that extra state in the construction is subtle. The construction might fail if the start state isn't an accepting state, but it takes some legwork from there to figure out how exactly to cause it to break. We hoped that this question would give you a better appreciation for how nuanced some of these constructions can be.