Intermediate Math Circles
Wednesday, November 14, 2018
Finite Automata II

Nickolas Rollick – nrollick@uwaterloo.ca

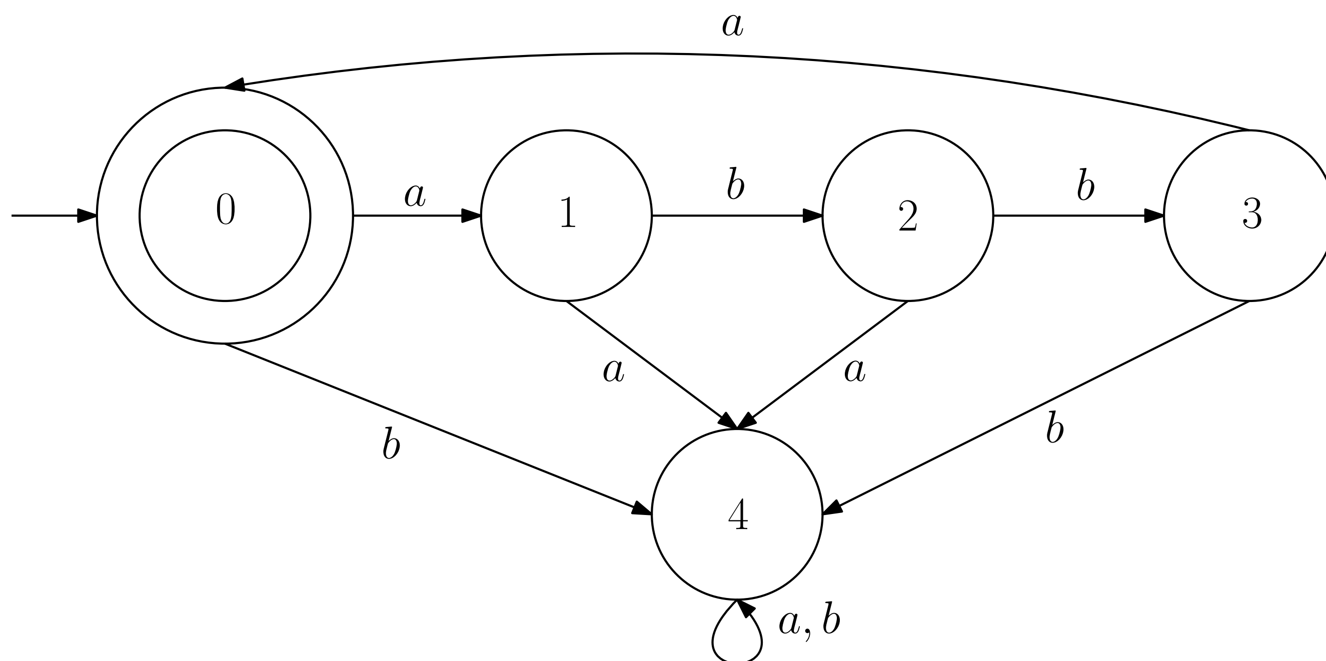
Regular Languages

Last time, we were introduced to the idea of a DFA (deterministic finite automaton), one of the simplest models of computing. We spent a good deal of time getting familiar with how they work, by constructing examples of DFAs accepting given languages.

Today, we want to take a closer look at the kind of languages accepted by DFAs. We give them a special name – *regular languages*. Ultimately, we want to get a feel for what regular languages look like, to identify them by sight. In that connection, our first task is to figure out how to build new regular languages out of old ones.

The first such construction is taking the *complement* of a language. In other words, if we have a regular language (one accepted by some DFA), we want to look at the language containing all and only the strings that are *not* in that first language. It turns out this language is always regular as well.

To get a sense for it, let's look at a specific example. Let's re-visit the ABBA machine from last time:

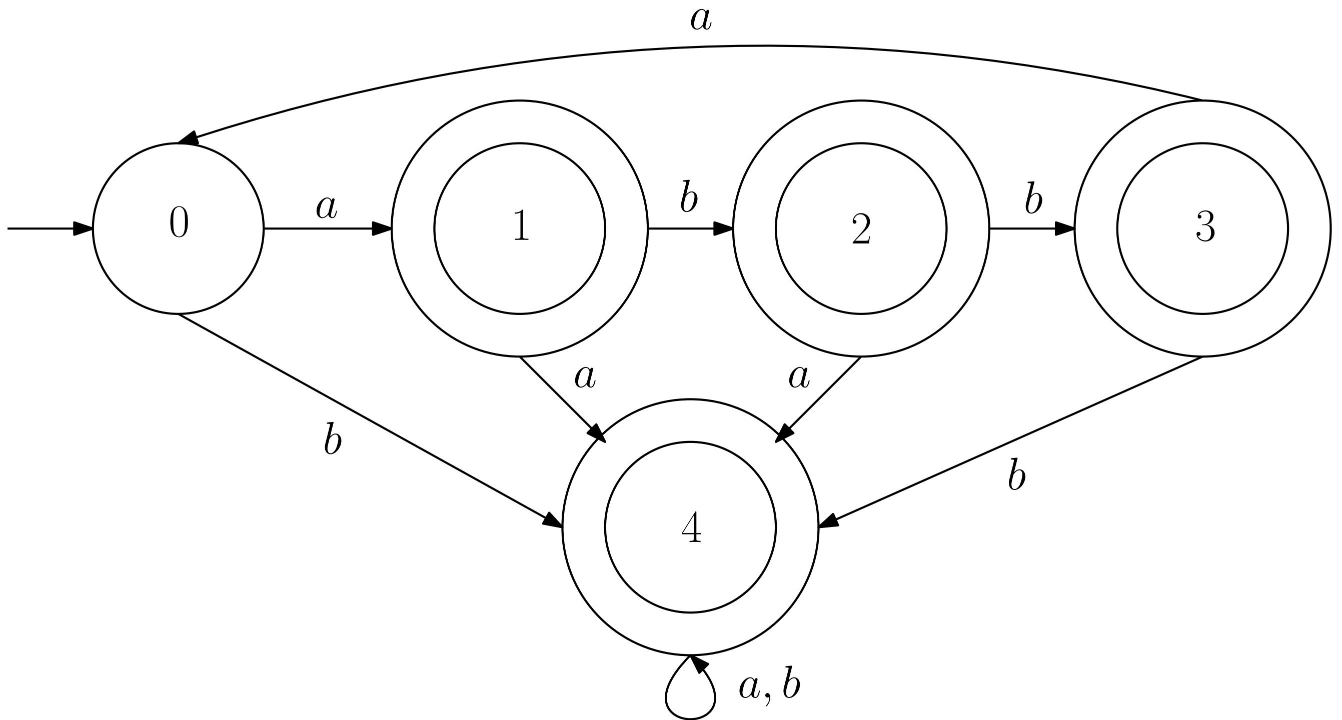


The language accepted by this machine is the set of strings where *abba* is repeated any number of times (including zero). Now, we want to build a DFA that accepts exactly the opposite kind of strings. In other words, this new DFA should reject the strings where *abba* is repeated any number of times, but it should accept everything else.

Is there an easy way to take the DFA we have and produce a DFA accepting this new language, the *complement* language?

The quick and easy solution is this: we want a DFA that accepts all the strings that used to be rejected,

and rejects all the strings that used to be accepted. If we just swap the accepting and rejecting states of the original machine, we'll get a DFA accepting the complement of the ABBA language:



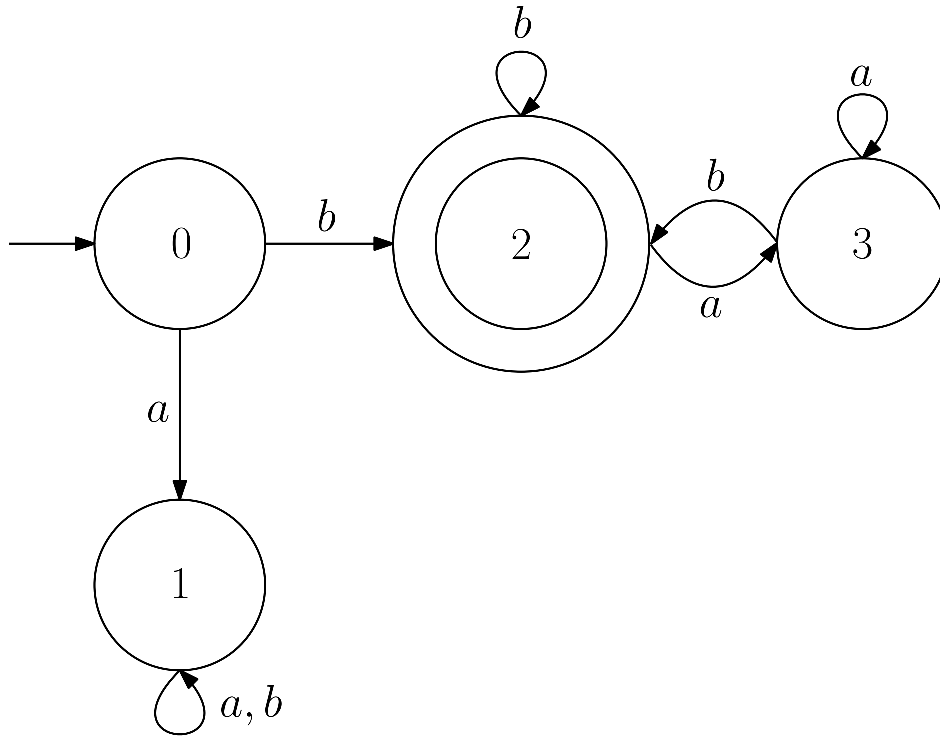
Given this idea, can you propose the way to do this in general? Given a DFA accepting a certain language, how can we modify it to create a DFA accepting the complementary language?

The same idea applies: take the original DFA, turn all its accepting states into rejecting states, and turn all its rejecting states into accepting states. This argument tells us that given a regular language, the complement of that language is also regular – we've built a new regular language out of an old one.

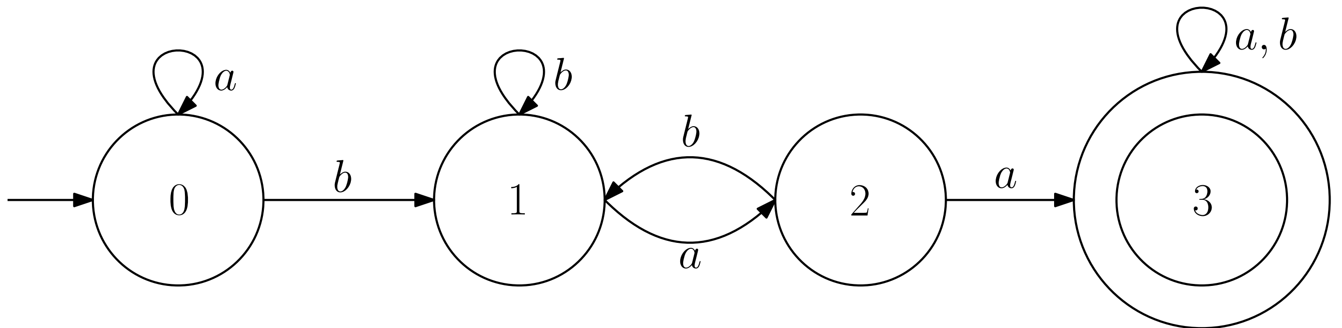
Of course, there are many more things of this kind that we can do. Given two regular languages, which we'll call L_1 and L_2, there are two related languages we can build. First, we might want to build a new DFA accepting only the strings belonging to *both* L_1 and L_2. Second, we may want to build a new DFA accepting the strings belonging to *either* L_1 or L_2. Like before, we will illustrate with an example.

Last time, you built a DFA that accepts the strings starting and ending with b, and you also built one accepting exactly the strings containing baa somewhere in the string. In other words, both these languages are regular.

One possible DFA accepting the first language is:



One possible DFA accepting the second language is:

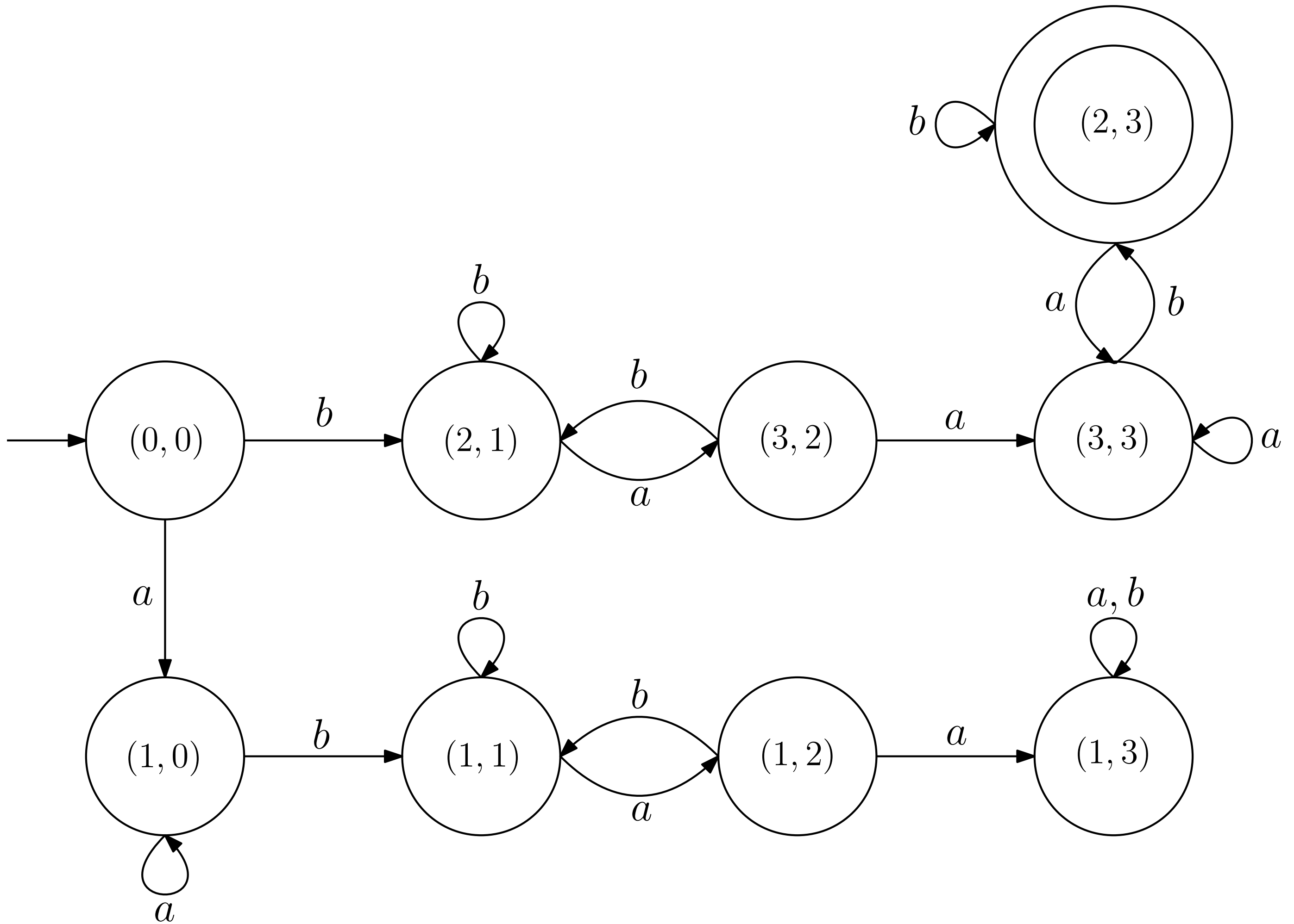


Given these two DFAs, is there a way to build a DFA accepting the *intersection* of the two languages? In other words, we want a DFA accepting only the strings belonging to both languages. More concretely, it will accept the strings starting and ending with b that also contain baa somewhere in the string. Take a few minutes to chat this over. We'll take it up as a group shortly.

The key to this is that we need to keep track of what's happening in both machines at the same time. You can imagine running both machines in parallel, giving them both the same string independently. Each machine will either accept or reject the string, and we only want to accept the string in the end if *both* machines have accepted the string.

The cool part is that we can model this with a single DFA. In this new DFA, we keep track of *pairs* of states, one for each of the two original DFAs. Each state in the new DFA matches up with a pair of states from the old DFAs. Then, when the new DFA reads a letter, each state in the pair changes in the same way that the two old DFAs would have. The accepting states of the new DFA match up with pairs of accepting states from the two old DFAs, since we only want to accept the strings that *both* the old DFAs liked.

Here's what we get when all is said and done:



In this new DFA, all the states are have pairs of numbers in the labels. The first number represents a state in the first DFA, and the second number represents a state in the second DFA. For example, state $(2, 1)$ represents the first DFA being in state 2, and the second being in state 1. If you give a to both machines, the first machine moves to state 3 and the second moves to state 2, so the new DFA moves to state $(3, 2)$. If you give b to both machines instead, both of them stay in the same state, so the DFA stays in state $(2, 1)$. Notice that $(2, 3)$ is the only accepting state in this new machine. This is because the only way both of the original DFAs accept a string is if the first ends up in state 2 and the second ends up in state 3.

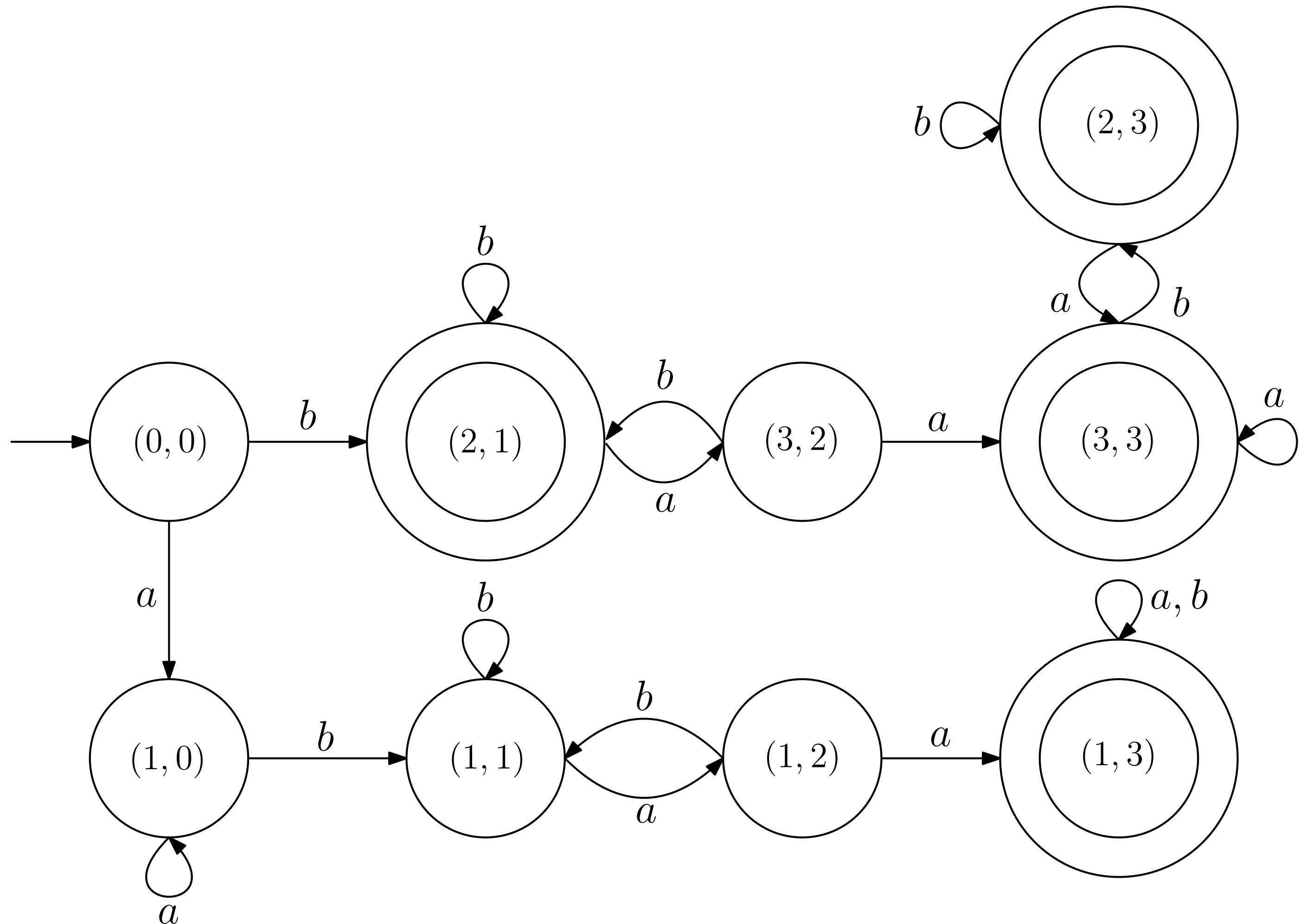
You can convince yourself that this DFA accepts the intersection of the two languages: all the strings starting and ending with b, and which contain baa somewhere.

Using a similar idea, we can build a DFA accepting the *union* of these two languages: all the strings belonging to *either* language. In our particular example, this means the collection of strings that start and end with a b, OR contain baa somewhere. In our minds, we get the same intuitive idea: imagine running the two original DFAs with the same input, and keeping track of which states both are in at any given time. After both finish reading the string, we accept it if *either* DFA has accepted it.

With this in mind, look back at the DFA accepting the intersection of the two languages. What small adjustment can we make to it so that it accepts the union instead?

All we have to change are the accepting states of the machine. We now want the DFA to accept the string if *either* state in the pair is an accepting state for the matching DFA. Since state 2 is an accepting state for the first DFA, any state in the new machine starting with a 2 should be an accepting state. Likewise, since

state 3 is an accepting state in the second DFA, any state in the new machine ending with a 3 should be an accepting state as well. The end result looks like this:



This is the machine accepting the strings that either start and end with b, or else contain baa somewhere inside of them.

All right, let's recap to make sure you've got the gist. Let's say I give you two DFAs, accepting two different languages. How do I build a DFA accepting only the strings belonging to both languages? To either language?

Great! We have now described several different ways to make new regular languages out of old ones. We can take the *complement* of a language, or the *union* or *intersection* of two languages.

But now the all-important question: is every language a regular language? In other words, if you write down any old collection of strings, can we always build a DFA accepting exactly that collection? Take five minutes to play with this question now, chatting it over with your friends.

The key to this question is that DFAs only have a fixed, finite number of states, and the states are the only memory the DFA has. Any language requiring us to store an unbounded amount of information can never be handled by a DFA. The simplest example of this phenomenon is the language of strings of the form $a^n b^n$, where n can be any whole number bigger than or equal to 0. In other words, the strings in the language are the empty string, ab, $aabb$, $aaabbb$, and so on. The important thing is that there is some number of as, followed by an equal number of bs.

Now that I've given you this example, can you explain why this language can't be regular? Intuitively, we need to keep a count of how many as have been read so far, in order to match up the corresponding number of bs. But since any number of as are allowed, we seem to need infinitely many states to keep track of it all.

But let's convince ourselves beyond a shadow of all doubt. Let's write down a mathematical *proof* that this language is not regular. This will be an argument no one can disagree with. Suppose there *was* a DFA that accepted this language. Necessarily, this DFA has only a finite number of states. Now, let's track what states the DFA passes through as it reads a bunch of as in a row. There are only so many states to be in, so eventually, we'll be able to find two whole numbers m and n, where the DFA is in the same state after reading a^m and a^n.

Mathematicians like to call this the *pigeonhole principle*: if you are trying to put pigeons into boxes, and you have more pigeons than boxes, at least two pigeons must end up in the same box. In the language of our problem: we're trying to put strings of the form a^k into states, and we have more strings than states, so at least two of the strings end up associated to the same state.

Let's say that the DFA ends up in state q after reading both a^m and a^n. Without doing any harm, we can assume m is smaller than n. Since this DFA accepts the language we're interested in, it accepts the string $a^m b^m$. Therefore, if you start in state q and read m letter bs in a row, you'll end up in some accepting state, which we'll call r. But now we have a problem. After reading a^n, the machine also ends up in state q, so if the input is $a^n b^m$, we'll end up in state q, then read m letter bs in a row, ending in state r. This means the DFA has to accept $a^n b^m$, but this can't be!

All in all, this means no DFA can accept this language – we've found a language that's not regular! Another very natural language (for computers) that isn't regular is the language of *legal bracketings*. For this, pretend that a represents a left bracket, and b represents a right bracket (we could actually use brackets as our symbols, but letters are easier to read). When you type an expression with brackets into a computer or calculator, what's the rule you need to follow? You can't have a right bracket without a matching left bracket coming first. In terms of letters, you can't have a b without a matching a that comes first. So *abaababb* is allowed, but anything starting with a b is not, and *abba* isn't either.

I'll let you come up with an argument for why this isn't regular on your own, but my hint is to take a close look at the argument we just finished using for the other language!

Conclusion

Today, we got a much better sense of how regular languages work. First, we explored how to build new regular languages from old ones, by taking complements, unions, and intersections. Then, we discovered that not all languages are regular: some are just too complicated to be accepted by any DFA. The natural question is: how can we change the design of the DFA to accept more languages? Next time, we'll explore one possible option – making the DFA *non*-deterministic, allowing it to make choices. This has some interesting tie-ins with the modern day concept of *quantum computers*.