

CS 3719 (Theory of Computation and Algorithms) – Lectures 2-4. Finite Automata and regular expressions

Antonina Kolokolova*

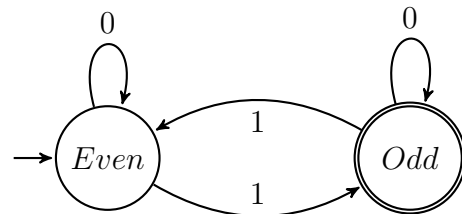
January 10, 2019

We start by talking about the simplest kind of a model of computation: finite automata, also known as finite state machines. Finite automata read the input symbol by symbol, and execute the computation by switching state (the only memory available in this system is the states). The answer is given by the kind of the state the automaton is in after reading the last letter of the input: if it ended in accepting state, the answer is “yes”, otherwise “no”.

1 Finite Automata

Let’s illustrate the finite automata with an example (hopefully it is a review for most of you).

Example 1. The following finite automaton receives a string of bits (0 and 1), and should finish in a final state (double-circle) if and only if the string contains an odd number of 1s.



More precisely, we define (deterministic) Finite Automata as follows.

Definition 1. A (deterministic) finite automaton (a DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- 1) Q is a finite set of states.
- 2) Σ is the alphabet (a finite set of symbols).

*The material in this set of notes came from many sources, in particular “Introduction to Theory of Computation” by Sipser and course notes of U. of Toronto CS 364.

- 3) $\delta : Q \times \Sigma \rightarrow Q$ is the transition function (pronounced as “delta”).
- 4) $q_0 \in Q$ is the start state
- 5) $F \subseteq Q$ is the set of accept (final) states.

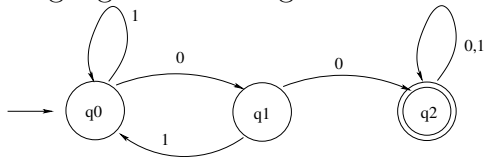
We say that a DFA accepts a string $w = \{w_0 \dots w_{n-1}\}$ if there exists a sequence of n states r_0, \dots, r_n such that $r_0 = q_0$, $r_n \in F$ and $\forall i, 0 \leq i < n, \delta(r_i, w_i) = r_{i+1}$. A DFA accepts (recognizes) a language L if it accepts every string in L , and does not accept any string not in L . A language is called a regular language if it is recognized by some finite automaton.

So to fully specify a finite automaton it is sufficient to say which states it is composed of (Q), what are the possible input symbols (Σ), where to start (q_0), where to end on good strings (F) and, the main part, what are the arrows and labels on them (δ). In example 1, $Q = \{even, odd\}$, $\Sigma = \{0, 1\}$, $q_0 = even$, $F = \{odd\}$ and δ is encoded by the following table:

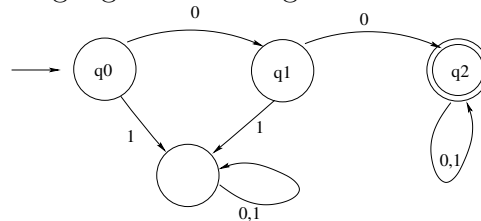
	0	1
even	even	odd
odd	odd	even

For example, on the string 1011001 it will go through the sequence of states “even, odd, odd, even, odd, odd, odd, even”, ending in a non-accepting state; but a string 101100 it will accept. So this automaton accepts a language of all strings over 0,1 in which the number of 1s is even.

Example 2. This automaton recognizes the language of all strings that contain 00.



Example 3. This automaton recognizes the language of all strings that start with 00.



1.1 Non-deterministic finite automata

A more general variant of finite automata, which is not implementable in practice (but often gives a more concise result) is a non-deterministic finite automaton (NFA).

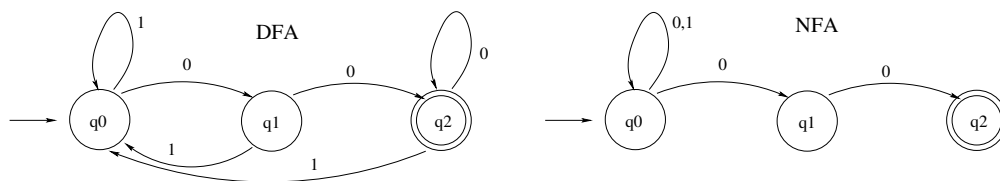
Definition 2. A non-deterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0 and F are as in the case of deterministic finite automaton (Q is the set of states, Σ an alphabet, q_0 is a start state and F is a set of accepting states), but the transition function δ is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$. Here, $\mathcal{P}(Q)$ is the powerset (set of all subsets) of Q .

A non-deterministic finite automaton accepts a string $w = w_1 \dots w_m$ if there exists a sequence of states r_0, \dots, r_m such that $r_0 = q_0$, $r_m \in F$ and $\forall i, 0 \leq i < m, r_{i+1} \in \delta(r_i, w_i)$.

There are two differences between this definition of δ and the deterministic case. First, δ gives a (possibly empty) set of states, as opposed to exactly one state. Second, there can be a transition without seeing any symbols, on empty string ϵ . Also, now the acceptance condition is that there exists a good sequence of states leading to acceptance: there can be many computational paths, some of which would lead to rejecting states, some would “die” with no next state to go to, and, if the string is in the language, at least one will finish in an accepting state.

Another difference is that if you have a deterministic automaton for a language L , then making a DFA for the complement of L (that is, $\bar{L} = \Sigma^* - L$) is as easy as switching accepting and rejecting states. However, this does not work with a NFA.

Example 4. Consider the following two automata accepting a language of all strings ending with 00. The first one is deterministic, the second non-deterministic. Notice how in the non-deterministic automaton there are two arrows on 0 from q_0 (so, $\delta(q_0, 0) = \{q_0, q_1\}$), no arrows on 1 from q_2 and no arrows at all from q_2 (so $\delta(q_2, 0) = \delta(q_2, 1) = \emptyset$.) This example does not have ϵ -arrows.



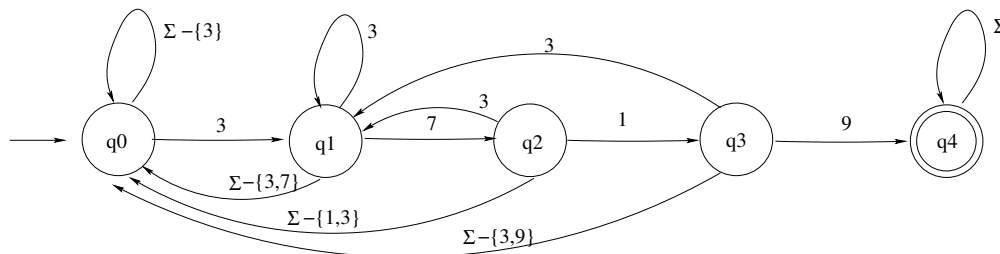
Consider possible executions of this NFA on string 101000. From q_0 on 1 there is just one choice: stay at q_0 . On the second symbol, there are two choices: stay at q_0 or go to q_1 . The computational branch that goes to q_1 dies at the next step: there is nowhere to go from q_1 on 1. The first computational branch survives, there the automaton stays in q_0 on the 3rd symbol. On the 4th symbol, 0, there is a choice again: you can see that moving to q_1 and then to q_2 will make it get stuck on the last 0. Similarly, not going to q_1 on the 5th symbol will make it finish in a reject state (either q_0 or q_1). Finally, an accepting sequence of states for string 101000 is: $q_0, q_0, q_0, q_0, q_0, q_1, q_2$.

1.2 String matching

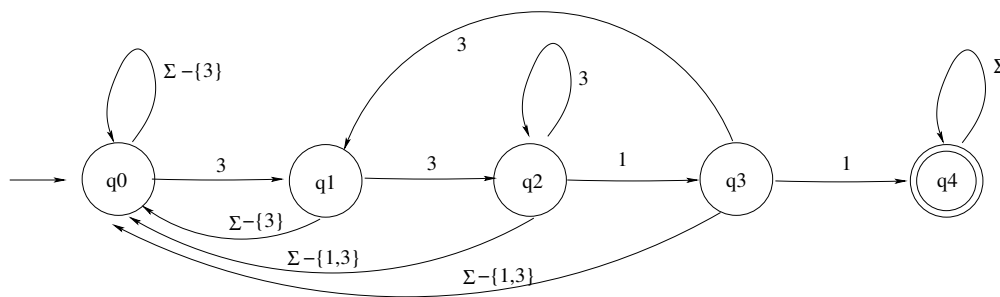
One area where automata have found much use is in string matching. Here, the pattern string (of length m is encoded by an automaton, which then gets the text (of usually much larger length n) as an input; the automaton should end in an accepting state if and only if this pattern occurs in the text. Such an algorithm runs very fast: in time $O(n|\Sigma|)$ plus time it takes to build an automaton: easy $O(m^3)$ (ignoring $|\Sigma|$); can be as efficient as $O(m)$. In particular, Knuth-Morris-Pratt algorithm you have seen in CS 2711 can be viewed as constructing a DFA in its preprocessing stage: think symbols of the pattern corresponding to the states $q_1 \dots q_m$, matching transitions going q_i to q_{i+1} and the partial match table

encoding non-matching transitions.

Example 5. Suppose you want to match a string “3719” in a file (here, take Σ to be all symbols that can occur in a text file). It is enough to feed the text from the file to this DFA.



Example 6. To make it more interesting, let’s build an automaton for a string with repeated letters, for the same Σ . Notice how backward transitions correspond to the “suffix matches the prefix” jumps in the KMP algorithm.



1.3 Regular expressions

Now that we defined finite automata, a natural question to ask is what kind of problems they can solve, that is, what kinds of languages are computable by DFAs and NFAs (and whether NFAs can do more than DFAs). First, we will describe what languages are computable by NFAs, and then talk about the equivalence of NFAs and DFAs in terms of computability.

Most of you have done Google searches for expressions like “MUN AND (CS3719 OR COMP3719)” to find pages that contain the string MUN and either the string CS3719 or the string COMP3719. Also, most of you have searched for a file with, say, 3719 in the name on a UNIX system by doing `ls '*3719*`'. In both cases you were using regular expressions to describe a set of strings you needed.

Formally, regular expressions are defined by the following recursive definition: R is a regular expression (over Σ) if R is one of the following:

- 1) A letter from an alphabet: $a \in \Sigma$, for some a (here, a is a generic name for a letter, does not have to be the symbol “a”).

- 2) An empty string ϵ .
- 3) The empty set \emptyset .
- 4) A union of two regular expressions $(R_1 \cup R_2)$ (that is, all strings that either match R_1 or match R_2 , or both).
- 5) A concatenation of two regular expressions $R_1 \circ R_2$ (that is, all strings that can be formed by writing a string of R_1 followed by a string of R_2).
- 6) A *Kleene star* R_1^* , which is zero or more concatenated strings from R_1 . This is the same $*$ as in Σ^* , where it means zero or more letters from Σ .

And only strings formed in this way are regular expressions. We would say a string “matches” a regular expression if it is a member of a language of this regular expression.

For example, suppose we want to define a set of all binary numbers (that is, $\Sigma = \{0, 1\}$) divisible by 4 by a regular expression. To make it more interesting, say we only want to list numbers starting with a 1 (e.g., we only consider strings starting with 1 as valid natural numbers for this example). Then we can do it as follows.

- 0 is a regular expression; so is 1, by rule 1.
- Then 00 is a regular expression by rule 5
- And $(0 \cup 1)$ is a regular expression by rule 4.
- By rule 6, $(0 \cup 1)^*$ is a regular expression (in fact, it matches all strings in Σ^* .)
- Finally, $1(0 \cup 1)^*00$ is a regular expression by two applications of rule 5

You can check yourself that the strings that are matched by this regular expression are binary numbers starting with 1 and ending with 00, that is, divisible by 4. For example, 100 is matched, and so is 10010101010100, but not 01100 or 1001. This language is infinitely countable: there are infinitely many numbers divisible by 4. Or, alternatively, you can see that there is a string in the regular expression for any string in $\Sigma^* = \{0 \cup 1\}^*$.

1.4 Regular Languages

Recall that a language is recognized by a finite automaton if each string in the language and no string not in the language are accepted by this automaton. There can be several different automata accepting the same language.

Definition 3. A language is called regular if there exists a finite automaton that accepts it. We will use the notation $\mathcal{L}(N)$ to mean the language accepted by the automaton N .

We will show soon that this is the same “regular” as in “regular expressions”. In particular, the operations used to build regular expressions inductively out of smaller ones are called regular operations:

Definition 4. Let A, B be languages. We define three regular operations on them:

- **Union** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- **Concatenations** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star** $A^* = \{x_1 \dots x_k \mid k \geq 0 \text{ and } \forall 1 \leq i \leq k, x_i \in A\}$

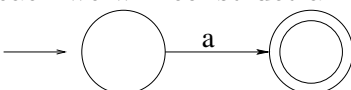
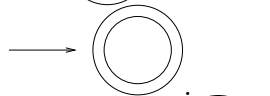
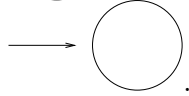
We are building up to showing that the class of languages expressible by regular expressions is the same as the class of languages accepted by finite automata: that is, in both cases it is the class of regular languages.

Theorem 1. The class of regular languages is closed under regular operations.

Corollary 2. For every regular expression, there is a finite automaton that recognizes the same language as this regular expression generates.

Proof. Recall the recursive definition of regular expressions: there were three base cases, and for the recursive step the three rules were union, concatenation and star. The theorem 1 (which we will prove later) gives us the recursive step. More precisely, if regular expressions R_1 and R_2 generate the same languages as recognized by finite automata N_1 and N_2 , respectively, then the closure under union operation gives us an automaton for a language $\mathcal{L}(N_1) \cup \mathcal{L}(N_2)$, which is the same set of strings as $R_1 \cup R_2$. Similarly, the theorem gives us automata for recognizing $R_1 \circ R_2$ and R_1^* given automata recognizing R_1 and R_2 . So all that is left to prove is the base case.

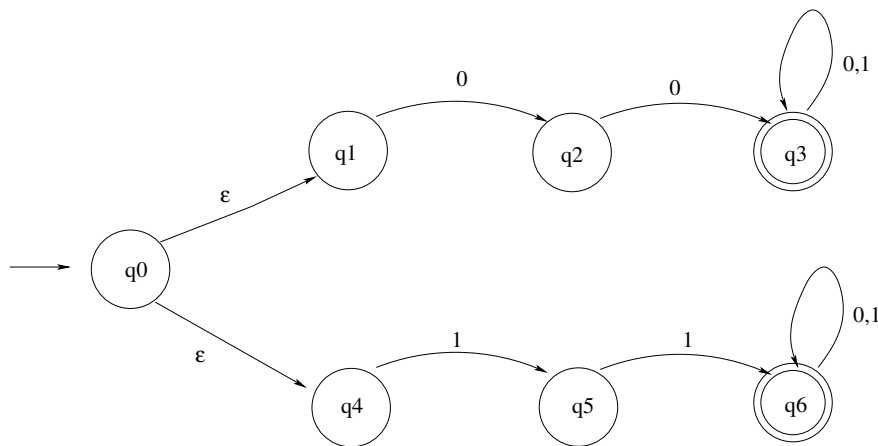
There are three parts of the base case, and for each we will construct a NFA accepting it.

- 1) $R = a$, for some $a \in \Sigma$. Take the NFA 
- 2) $R = \epsilon$ Define the corresponding NFA as 
- 3) $R = \emptyset$ Then, take an NFA accepting nothing. 

The proof follows by structural induction. Formally, suppose R is a regular expression; we will show how to construct a NFA recognizing the same language. If R is of the form a or ϵ or \emptyset then use the corresponding automaton from the previous paragraph. Otherwise $R = R_1 \cup R_2$ or $R = R_1 \circ R_2$ or $R = R_1^*$. By induction hypothesis, there exist NFAs N_1 and N_2 recognizing languages of R_1 and R_2 respectively. Now, by the theorem 1 there is an NFA N with $\mathcal{L}(N) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$. Thus, a regular expression $R = R_1 \cup R_2$ generates a language $\mathcal{L}(N)$. The arguments for $R_1 \circ R_2$ and R_1^* similarly follow from theorem 1. \square

To give an intuition for the proof of theorem 1 for the case of union, as well as illustrate the use of ϵ -arrows, consider the following example.

Example 7. Consider the language of strings over $\{0, 1\}$ starting with either 00 or 11. Here, the ϵ -arrows at the beginning allow us to start either at the automaton recognizing strings starting with 00 or at the automaton recognizing strings starting with 11.



1.5 Equivalence of NFAs and DFAs

When we defined regular languages, we said that a language is regular if it is accepted by some finite automaton, without really specifying whether it is a DFA or an NFA. It is clear that if a language can be recognized by some DFA can also be recognized an NFA, at least because DFAs are a special case of NFAs. But what about the other direction, are there languages that can be done by NFAs but not DFAs? In general, does non-determinism make the model of computation more powerful?

In this section we will show how to simulate NFAs by DFAs, although the simulation is not very efficient. To the most general question, about the power of non-determinism, there is no single answer: sometimes non-determinism does not help much (as for the finite automata or Turing machines), sometimes it allows the model to recognize languages it would not be able to recognize without (that will be the case for pushdown automata, which we will talk about

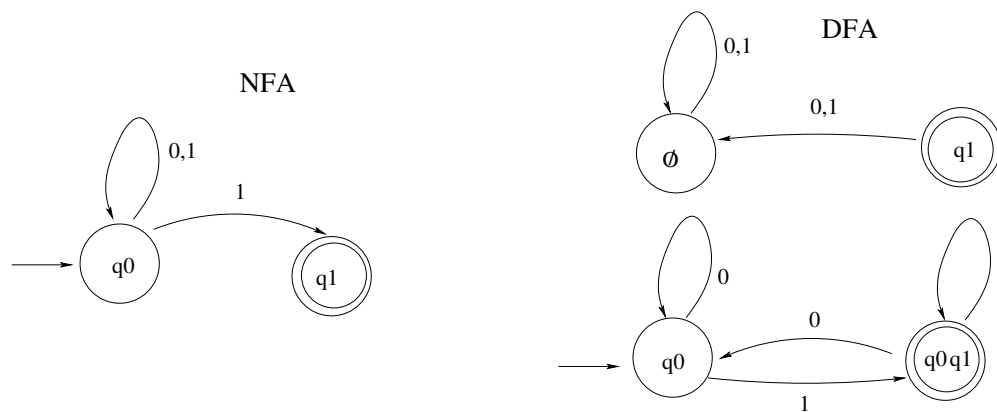
next week) and sometimes the answer is not known: a major, literally million-dollar, problem P vs. NP asks whether non-deterministic Turing machines can be simulated efficiently by deterministic ones. We will talk about this problem at length at the last part of the course.

Theorem 3. For every NFA N there is a DFA D such that $\mathcal{L}(N) = \mathcal{L}(D)$.

Corollary 4. The class of languages accepted by NFAs is the same as the class of languages accepted by NFAs (regular languages.)

Proof. The idea of the construction is as follows. Think of tracking an execution of N on a string. On every step of the NFA, maintain a set of states in which N could be at the moment. Then, on a next symbol, see where it is possible to get from any of the current states on that symbol. If there are ϵ -transitions, then every time after computing the states reachable from the current also add to the set any states reachable from the computed ones on ϵ -arrows.

Example 8. Here is an example of an NFA and a corresponding DFA. Consider an execution of this NFA on a string 1101. The first state is q_0 , after that, on seeing a 1, it can go to either q_0 or q_1 ; record it as being in a state $\{q_0, q_1\}$. After seeing another 1, it can again go to either q_0 or q_1 already from q_0 , so both states are possible. From there it will go back to q_0 on a 0 since there are no transitions from q_1 on 0, and from q_0 on 0 N stays in q_0 . And, finally, it will go again to $\{q_0, q_1\}$ on 1. Since this means it is possible to end in an accepting state q_1 , we treat $\{q_0, q_1\}$ as an accepting state.



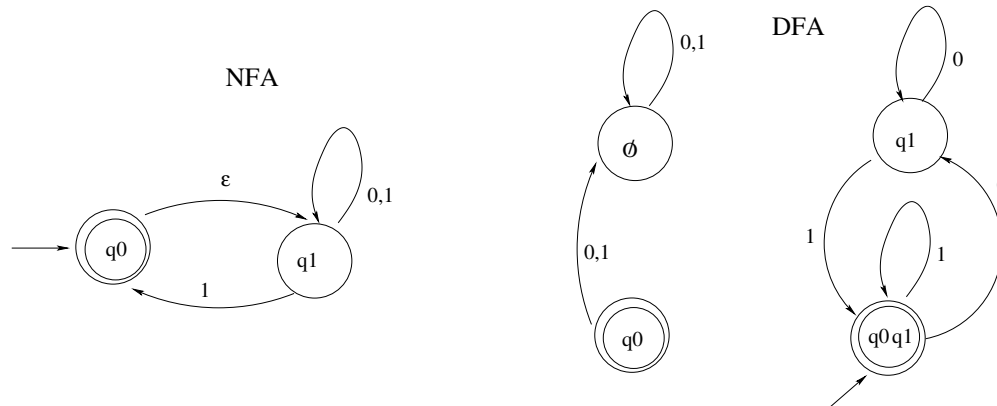
To formalize this intuition, let's describe $D(Q', \Sigma, \delta', q_0, F)$. We will have to talk about sets of states reachable from a given set by following ϵ -arrows: let's call, for a set of states Q_i , a $E(Q_i) = \{q \in Q \mid \exists q' \in Q \text{ such that } q \text{ is reachable from } q' \text{ by following } \epsilon\text{-arrows}\}$.

- Set $Q' = \mathcal{P}(Q)$, that is, a set of all subsets of Q . Thus, this conversion is not very efficient: the new DFA is exponentially larger than the original NFA.
- The new start state will be all states accessible from q_0 by following ϵ -arrows. That is, $q'_0 = E(\{q_0\})$.

- The final states of D are all sets that contain at least one final state of N . That is, $F' = \{Q_i \mid \exists q \in F, q \in Q_i\}$.
- Finally, we need to define the transition function of D , $\delta'(Q_i, a)$. It should be a set of states reachable from states in Q_i by doing δ -transitions on a , together with everything else that can be reached from them by ϵ -arrows. More formally, $\delta'(Q_i, a) = \{q \mid q \in E(\delta(q', a)) \text{ for some } q' \in Q_i\}$.

□

Example 9. Here is an example of an NFA with ϵ -transitions and a corresponding DFA. Note that the start state is now $\{q_0, q_1\}$, since q_1 is reachable from q_0 by an ϵ -transition. Also, q_1 is reachable from q_1 two different ways: by the self-loop, and by going to q_0 and back on ϵ -arrow.



Note that in these two examples there are groups of states that are not reachable from the start. If this is the case, then we often just draw the part of the automaton which is reachable, ignoring the states we never get to.

1.6 Nonregular languages

A natural question to ask is whether there are languages that are not regular, that is, whether any language (any subset of Σ^* for a finite Σ) can be described by a regular expression. In this lecture, we will show some examples of languages for which it is not possible to build a finite automaton or write a regular expression; we will call such languages nonregular.

How would one go about proving that some language is *not* regular? We cannot check all possible finite automata to make sure that none of them accepts the language, there are infinitely many automata to check. So instead we will identify a property that all regular languages satisfy, and show languages that do not satisfy this property.

Example 10. Consider the automaton accepting all strings with the number of 1s divisible by 3. There is one property you can notice about this automaton: if you take a string it accepts, and repeat any substring of it in which the number of 1s is divisible by 3, you again obtain a string in the language. For example, if 111011010 is in the language, so is 11 1011 1011 010, as well as 111 111 011010. If you look at the automaton, the reason becomes clear: repeating such a substring the automaton returns to the same state. In case of 11 1011 1011 010, the automaton was in the state q_2 after seeing the first 11, and it is again in q_2 after seeing 1011, and back to q_2 after seeing the second block of 1011... and after that its execution on the remaining 010 is the same as it would be without the additional 1011.

This observation will lead us to a property that holds for all regular languages.

Lemma 5 (Pumping lemma). *If A is a regular language, then there exists a number $p \in \mathbb{N}$ called “pumping length” such that for any string $s \in A$, if $|s| \geq p$ then $s = xyz$, where*

- 1) $\forall i \geq 0, xy^iz \in A$. That is, if s is in A then so is any string obtained from s by repeating the block y 0 or more times.
- 2) $|y| > 0$. Otherwise the lemma would be trivial: you can always repeat an empty substring as many times as you like.
- 3) $|xy| \leq p$. This property comes out of the proof of the lemma and helps with some nonregularity proofs.

Proof. Since A is regular, there exists a DFA D , $\mathcal{L}(D) = A$. Suppose D has p states; this number will be the pumping length.

Consider any string s of length at least p . Since during a computation on a string of length p the DFA goes through a sequence of $p + 1$ states, by the Pigeonhole Principle at least one state in this sequence repeats. Now, take y to be the substring between the first two repetitions of the first repeating state. Then repeating this y 0 or more times does not change the acceptance of s by D : just add or remove the sequence of states of D on y from the corresponding position in the sequence of states. The $|y| > 0$ because this is a DFA with no ϵ -transitions. And the last condition is satisfied because there are at most p states the DFA can go through before repeating a state, again by the Pigeonhole Principle: here, x is a string from the start of s to the first occurrence of the first repeating state. \square

Now we will use this lemma to show that several languages are not regular; in particular, we will show that “counting” cannot be done by finite automata.

Example 11. The language $A = \{0^n1^n\}$ is not regular.

For the sake of contradiction, suppose that it is regular. Then there exists a natural number p such that any string longer than p satisfies the condition of the pumping lemma. Here, we will present a string that is in A , but cannot be “pumped”.

Consider $s = 0^p 1^p$. Clearly, $|s| \geq p$. By the lemma, then $s = xyz$ such that $\forall i \geq 0, xy^i z \in A$. There are three possibilities: y can consist just of 0s, or just of 1s, or of a mix of 0s and 1s. In the first case, repeating y would increase the number of 0s, but not the number of 1s, so $xyyz \notin A$. Similarly, if y consists of only 1s, repeating it would make a string not in A . Finally, suppose that y consists of some 0s followed by some 1s. But then the string $xyyz$ has some 1s before 0s, which again makes it not in the language.¹

Therefore, s cannot be written as $s = xyz$ in a way that satisfies the pumping lemma. Thus, the language $\{0^n 1^n\}$ is not regular.

Example 12. The language $A = \{ww | w \in \{0, 1\}^*\}$ is not regular. This also holds for any other Σ with $|\Sigma| > 1$.

Again, assume for the sake of contradiction that A is regular; then by the pumping lemma there exists a pumping length p , such that for any $s \in A$, $|s| > p$, $s = xyz$ where y 's can be repeated.

Take $s = 0^p 10^p 1$. By the 3rd condition of the pumping lemma, $|xy| \leq p$. And therefore the repeating part y would consist of only 0s. But adding 0s in the first half of the string will make the first half different from the second half, and thus not in the language.

Note, by the way, that if we would have taken a string $s = 0^p 0^p$, then the argument would not work: take a y of a small even length, and it could be pumped indefinitely. So the choice of a string matters: it is not that we need to show that every string in the language cannot be pumped, we just need to present one such string.

Example 13. The language $A = \{w | w \in \{0, 1\}^* \text{ contains the same number of 0s and 1s}\}$ is not regular.

Assume for the sake of contradiction that A is regular; let p be the pumping length. Take $s = 0^p 1^p$, like in the first example. But now the case of "mixed 0s and 1s" does not apply: such a string would still be in the language. Here, we do need to refer to the $|xy| \leq p$ condition of the pumping lemma: by that condition, y would have to contain only 0s, so repeating y would make a string with more 0s than 1s.

Also note that here taking a string, for example, $(01)^p$ would not be useful: this string can be pumped.

Example 14. The language $A = \{0^i 1^j | i > j\}$ is not regular.

(Again assume A is regular and get p from the pumping lemma). Take a string $s = 0^{p+1} 1^p$. Now saying that $|xy| \leq p$ and so y consists of just 0s does not help: increasing the number of 0s would give us a string in the language. However, notice that in the first condition of the pumping lemma $i \geq 0$. That is, a string xz with no repetitions of y should still be in the

¹Note that the last two cases can be also eliminated by using the $|xy| < p$ condition of the pumping lemma.

language (think of skipping the first loop on that first repeating state of the automaton). However, here if we will remove even one 0 from the string the resulting string will not be in the language. Therefore, this language is also not regular.