COMP 218
Decision, Computation and Language

Dominik Wojtczak

Department of Computer Science
University of Liverpool, U. K.

2019–20

based on slides from Paul Goldberg, Jeff Ullman, Andrej Bogdanov, M. Leucker, M. Andreina Francisco, and Raymond J. Mooney

General information

Module coordinator: Dominik Wojtczak

PhD (University of Edinburgh), postdoc (CWI Amsterdam and University of Oxford)

email: D.Wojtczak@liverpool.ac.uk

office hours: Monday 11am-11.50am, Ashton Building 217a

research interests: automata theory, game theory, probabilistic systems and verification

Tutor: Mr Joe Livesey

email: Joseph.Livesey@liverpool.ac.uk

COMP218 takes place in the 1st semester from now on.

COMP218 on VITAL

- lecture slides
- tutorial exercises
- extra material
- homework
- **discussion board**
  - please subscribe!!!
  - post questions there instead of by asking them via email or even better ask them during lectures (so the same questions are not asked repeatedly)
  - you can post anonymously, but please do not abuse that

Lectures

- three lectures (1h+2h) per week; I will try to record all of them;
- lecture notes are already available on VITAL (some changes may occur later though);
- unassessed exercises in lectures from time to time, so please bring pen and paper;
- online lectures on Automata Theory `https://www.coursera.org/course/automata`;
- please do not talk during the lectures, this really distracts the other students;
- lecture slides are not meant to replace your notes, many things will only be said but not written down;
- lectures will focus on discussing examples of various constructions and models, and mostly present just the intuition behind the proofs;
- formal proofs can be found in the following books...

Books

Recommended book

Michael Sipser. *Introduction to the Theory of Computation*, 3rd edition, 2012. Pages 1–244. (Other parts of it can also be useful for COMP202 and COMP309.)

Supplementary books

John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Automata Theory, Languages, and Computation*, 3rd edition, 2007 (for ϵ-NFA to NFA conversion, DFA minimisation, and CYK algorithm).

Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT press, 1999. (chapter 11 for probabilistic context-free grammars)

Tutorials

10 tutorial sessions starting tomorrow!

- an integral part of this module!
- it will be hard to follow this module without solving the tutorial exercises (or at least trying to);
- five tutorial groups;
- all tutorial exercises already available on VITAL;
- solutions available on VITAL one week after all tutorial groups of a given session are over;
- if you really cannot make it to your tutorial group in a given week, you can join one of the other groups, but let the tutor know about this.

Assessments

Two assessments each worth 10%. Each assessment consists of:

- a homework (2.5%);
- a 40 minutes long class test (7.5%).

The 1st homework will be conducted using Automata Tutor tool: `www.automatatutor.com` (You will be automatically registered within the next three weeks.)

The 2nd homework will be conducted using VITAL.

Class tests

The second class test will consists of 15 MCQ questions.

The first class test will either be an MCQ or a written one. You will decide through the mid-semester evaluation form on VITAL!

- MCQ: random guessing gives 20%, but no partial mark is given, no feedback apart from the score
- written one: partial marking, more feedback

Please let me know (ideally via email) at least two weeks in advance if you need any special provisions for the class tests.

Class tests will take place in **weeks 7 and 10 on Wednesday at 9am** (6th November and 27th November).

Timetable

UNIVERSITY OF LIVERPOOL

Module: COMP218 Decision, Computat & Language

Weeks: S1 01-S1 12, S1 13 (23 Sep 2019-12 Jan 2020)

	9:00	9:30	10:00	10:30	11:00	11:30	12:00	12:30	13:00	13:30	14:00	14:30	15:00	15:30	16:00	16:30	17:00	17:30
Mon			**COMP218 - Decision, Computat & Language [LECTURE]** COMP218 COMP218/LEC/A/01 MATH-027 Wojtczak, Dominik S1 01-S1 12															
Tue																		
Wed	**COMP218 - Decision, Computat & Language [CLASS TEST]** COMP218 COMP218/EXAM/01 ULMS-SR1, REN-LT6 Wojtczak, Dominik S1 07, S1 10				**COMP218 - Decision, Computat & Language [TUTORIAL]** COMP218 COMP218/TUT+/02 LIFS-SR4 Wojtczak, Dominik S1 01-S1 12													
Thu	**COMP218 - Decision, Computat & Language [TUTORIAL]** COMP218 COMP218/TUT+/03 502-TR5 Wojtczak, Dominik S1 01-S1 12												**COMP218 - Decision, Computat & Language [TUTORIAL]** COMP218 COMP218/TUT+/01 502-TR6 Wojtczak, Dominik S1 01-S1 12					
Fri					**COMP218 - Decision, Computat & Language [LECTURE]** COMP218 COMP218/LEC/B/01 ELEC-202[E2] Wojtczak, Dominik S1 01-S1 12								**COMP218 - Decision, Computat & Language [TUTORIAL]** COMP218 COMP218/TUT+/04 BROD-305a Wojtczak, Dominik S1 01-S1 12		**COMP218 - Decision, Computat & Language [TUTORIAL]** COMP218 COMP218/TUT+/05 BROD-305a Wojtczak, Dominik S1 01-S1 12			

Close Window Print Timetable Key to Room Codes Key to Weeks Date/Time: 9 Sep 2019 12:38

Feedback

- results and revision lectures after class tests
- tutorials!
- AutomataTutor
- the discussion board
- in-class questions

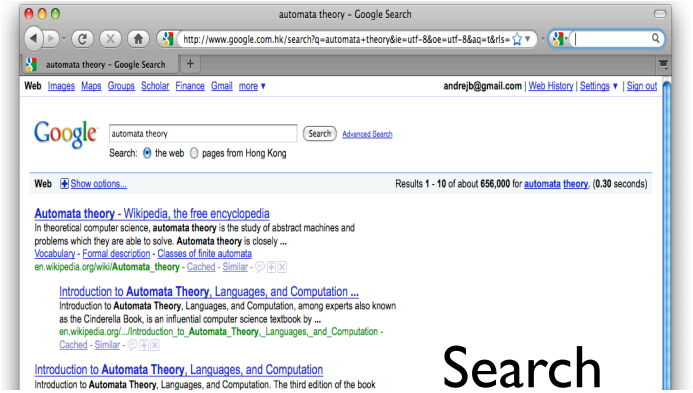
Exam

At the end of COMP218 there is an MCQ exam:

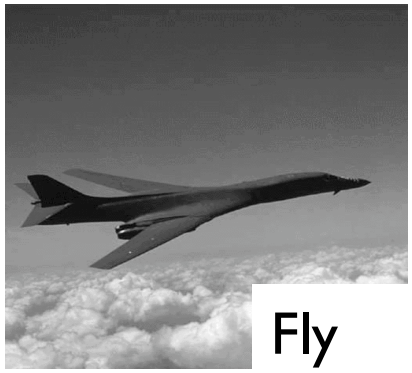
- two hours long, worth 80% of the final mark;
- 40 problem-solving questions;
- almost no bookwork questions, i.e., it is simply impossible to recall the correct answer;
- 5 possible answers, only one is correct, 20% by guessing, 40% required to pass; quite a bit of students failed last year;
- past MCQ exams/class tests are not available, but questions will be similar to previous written exams that are available on VITAL;
- the tutorials, homework and class tests should prepare you well for the exam;
- your final mark will suffer if you miss out on any of them!

Overview of COMP218: Decision, Computation and Language

Is there anything a computer cannot do?



Search



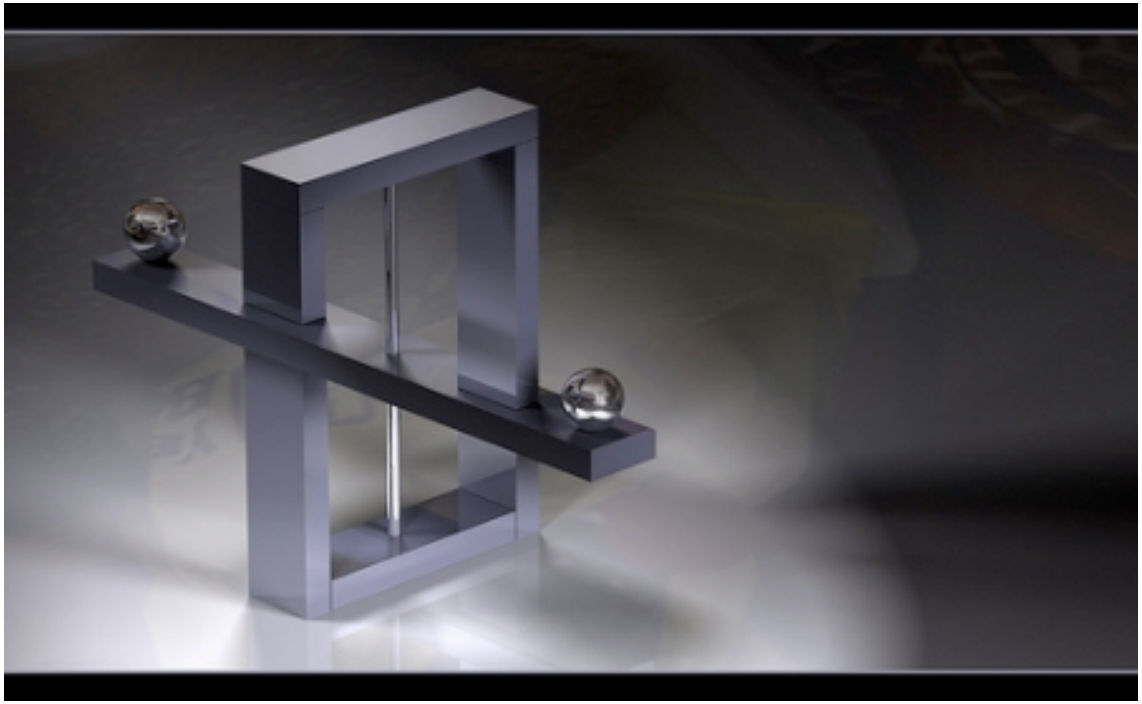
Fly

Drive?

Play Chess and Go

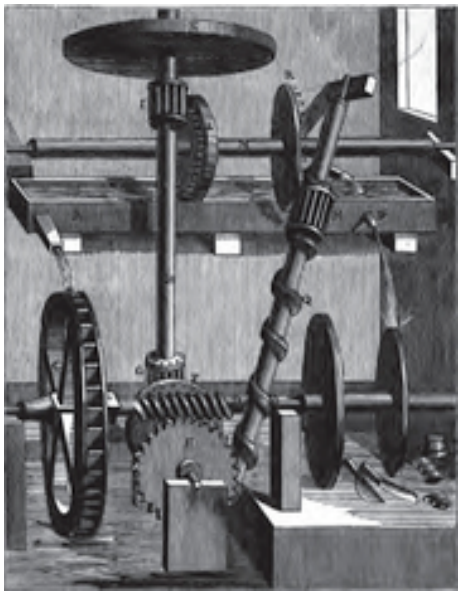
Translate

Impossibilities



Why do we care about the impossible?

Perpetual motion



In the middle ages, a lot of time and effort was invested into building a machine that can work indefinitely without any energy source (and trying to turn lead into gold).

It follows from the laws of thermodynamics discovered later that such a machine can never be built.

Understanding the impossible helps us channel our resources towards the more fruitful endeavors.

The laws of computation

Just like the laws of physics tell us
what is (im)possible for nature to do...

...the laws of computation tell us
what is (im)possible for computers.

Automata theory

Automata theory studies the laws of computation.

In reality, the laws of computation are not well understood,
but automata theory is a good start.

Initiated by Alan Turing in 1930s even before computers existed.

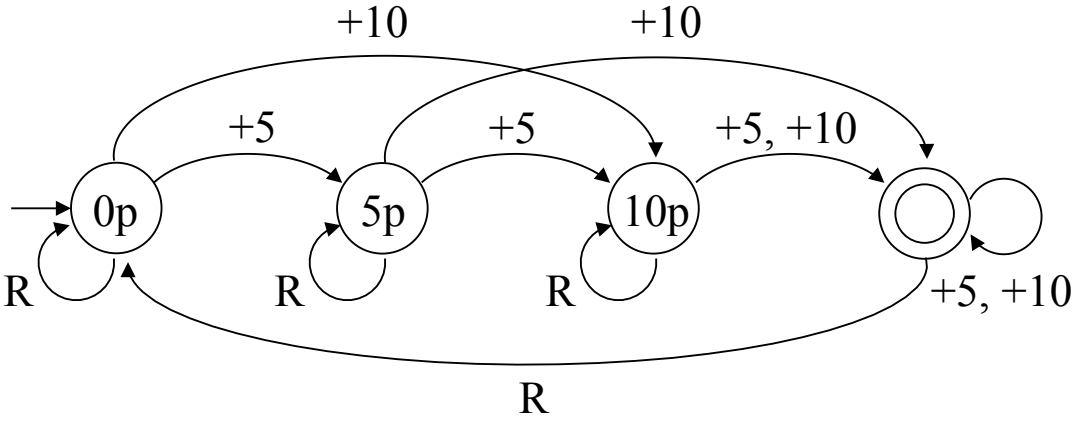
Formal languages and automata theory

A gumball machine

machine takes 5p and 10p coins

a gumball costs 15p

actions: +5, +10, R

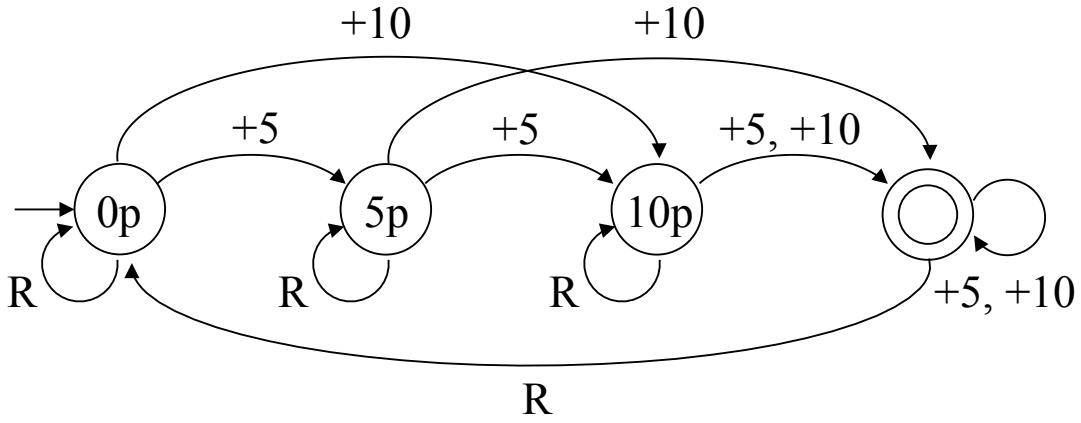


What can automata do

- They can describe the operation of a device/process
- They can be used to verify software
- They are used in lexical analysers to recognise expressions in programming languages:

`ab1` is a legal name of a variable in java
`5u=` is not

Different kinds of machines



- This was only one example of a machine
- We will look at different kinds of machines and ask:
  - What kind of problems can such machines solve?
  - What things are impossible?
  - Is one kind of machine more powerful than another?

Some kinds of machines

finite automata	Devices with a small amount of memory. Used to model very simple things.
pushdown automata	Devices with infinite memory that can be accessed in a restricted way. Used to parse grammars
Turing Machines	Devices with infinite memory. These are at least as powerful as computers
time-bounded Turing Machines	Infinite memory, but bounded running time. These are computers that run reasonably fast.

The Origins

- Finite automata
  - Warren McCulloch and Walter Pitss (1943). A Logical Calculus Immanent in Nervous Activity. (a neurophysiologist and mathematician)
  - Generalised by G. H. Mealy and E.F. Moore between 1955-56 (regular transducers)
- Context-Free Grammars (=pushdown automata)
  - Noam Chomsky (mid 1950s). (linguist and cognitive scientist)
- Turing Machines
  - Alan Turing (1937). On computable numbers, with an application to the Entscheidungsproblem. (logic)

Computer Science

NASA Computer Room (1949)

computer : a person that computes or calculates (since 17th century)

Some highlights of this module

- Finite automata
  - Automata are closely related to the task of searching for patterns in text

```
find (ab)*(ab) in abracadabra
```

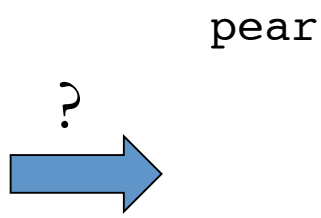
- Grammars
  - Grammars describe the meaning of sentences in English, and the meaning of programs in Java
  - We will see how to extract the meaning out of a program

Some highlights of this module

- Turing Machines
  - This is a general model of a computer, capturing anything we could ever hope to compute
  - But there are many things that computers cannot do:

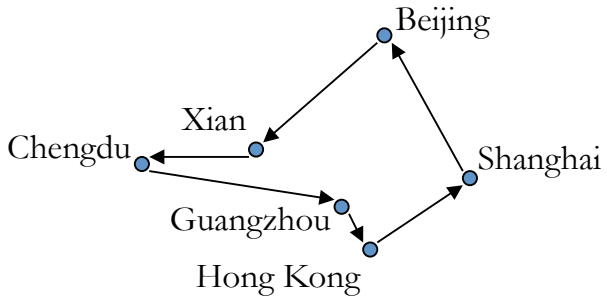
> Given the code of a computer program, can you tell if the program prints the string "`pear`"?

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')(){}#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```



Some highlights of this module

- Time-bounded Turing Machines
  - Many problems are possible to solve on a computer in principle, but take too much time in practice.
  - Traveling salesman: Given a list of cities, find the shortest way to visit them and come back home.



  - Hard in practice: For 200 cities, computing the best path even on the fastest computer would take more than the age of the universe!

Preliminaries of automata theory

- How do we ask the question

Can machine A solve problem B?

- First, we need a way of describing the problems that we are interested in solving.

Set Theory Recap

- $\varnothing \quad \{\varnothing\}$
- $\{a_1, a_2, a_3, a_4, \ldots, a_n\}$
- $a \in A \qquad B \subseteq A$
- $A \cup B \quad A \cap B \quad A \setminus B$
- $2^A = \{B \mid B \subseteq A\} = \{B : B \subseteq A\}$
- $(a_1, a_2, a_3, a_4, \ldots, a_n)$

Problems

- Examples of problems we will consider
  - Given a word s, does it contain "to" as a subword?
  - Given a number n, is it divisible by 7?
  - Given two words s and t, are they the same?

- All of these have "yes/no" answers. These are called decision problems.
- There are other types of problems, like "Find this", "Optimise that", "How many of this" but we won't look at them here. (COMP202 part. However, sometimes we can deal with them using the solution to the decision problem.)

Alphabets and strings

- A common way to talk about words, numbers, pairs of words, etc. is by representing them as strings (aka words)
- To define strings, we start with an alphabet

> An alphabet is a finite set of symbols.

- Examples

$\Sigma_1 = \{a, b, c, d, \ldots, z\}$: the set of letters in English

$\Sigma_2 = \{0, 1, \ldots, 9\}$: the set of (base 10) digits

$\Sigma_3 = \{a, b, \ldots, z, \#\}$: the set of letters plus the special symbol #

Strings

A string over alphabet Σ is a finite sequence of symbols in Σ.

`abfbz` is a string over $\Sigma_1 = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \ldots, \texttt{z}\}$

`9021` is a string over $\Sigma_2 = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$

`ab#bc` is a string over $\Sigma_3 = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}, \#\}$

- The empty string will be denoted by ε
- We write Σ^* for the set of all strings over Σ
- We write Σ^+ for the set of all non-empty strings

Languages

A language is a set of strings (over the same alphabet).

- Languages define a "yes/no" problems:

$L_1 =$ All strings that contain the substring "`to`"

$\Sigma_1 = \{$`a`, `b`, …, `z`$\}$

`stop`, `to`, `toe` are in L_1

ε, `oyster` are not in L_1

$L_1 = \{x \in \Sigma_1^* \mid x$ contains the substring "`to`"$\}$

Examples of languages

$L_2 = \{x \in \Sigma_2^* \mid x \text{ is divisible by } 7\}$ $\quad \Sigma_2 = \{0, 1, \ldots, 9\}$

$= \{7, 14, 21, \ldots\}$

$L_3 = \{s\#s \mid s \in \{a, b, c, \ldots, z\}^*\}$ $\quad \Sigma_3 = \{a, b, \ldots, z, \#\}$

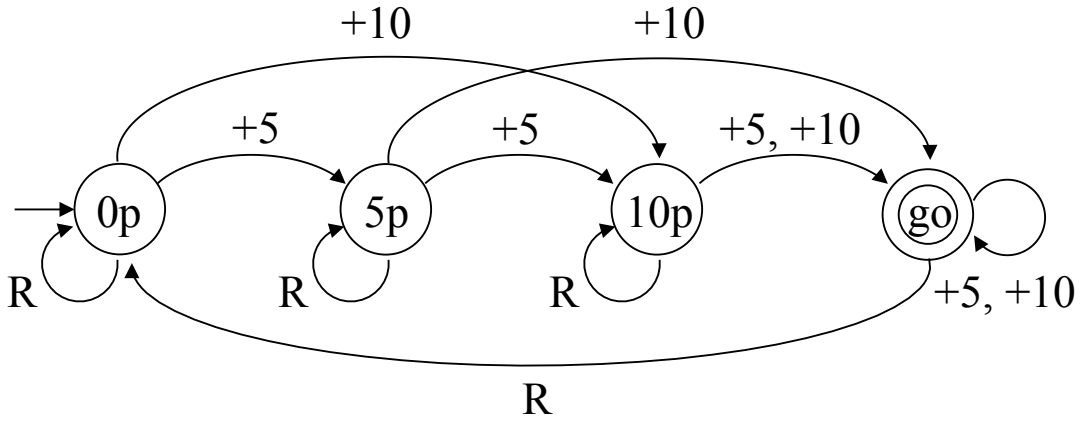
ab#ab in L_3

ab#ba not in L_3

a##a# not in L_3

Finite Automata

Example of a finite automaton



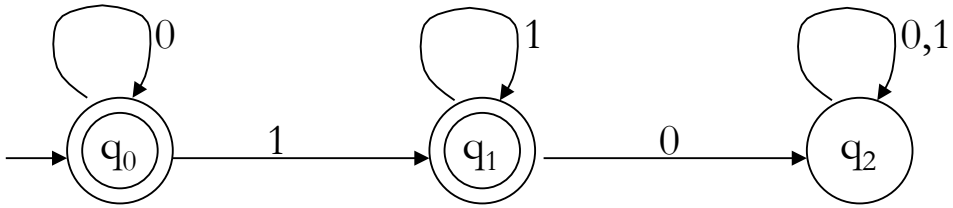
- There are states $0p$, $5p$, $10p$, go, the start state is $0p$
- The automaton takes inputs from $\{+5, +10, R\}$
- The state go is an accepting state
- There are transitions saying what to do for every state and every alphabet symbol

Deterministic finite automata

- A finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set of states
  - Σ is an alphabet
  - $\delta: Q \times \Sigma \rightarrow Q$ is a transition function
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$ is a set of accepting states (or final states).

- In diagrams, the accepting states will be denoted by double circles

Example



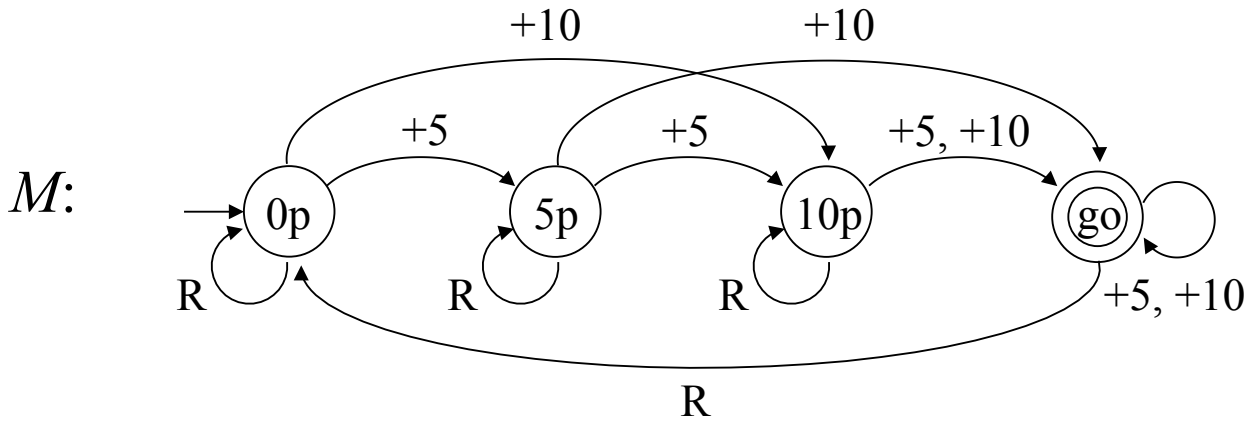
alphabet $\Sigma = \{0, 1\}$
states $Q = \{q_0, q_1, q_2\}$
initial state q_0
accepting states $F = \{q_0, q_1\}$

table of
transition function δ:

		inputs	
		0	1
states	q_0	q_0	q_1
	q_1	q_2	q_1
	q_2	q_2	q_2

Language of a DFA

The language of a DFA $(Q, \Sigma, \delta, q_0, F)$ is the set of all strings over Σ that, starting from q_0 and following the transitions as the string is read left to right, will reach some accepting state.

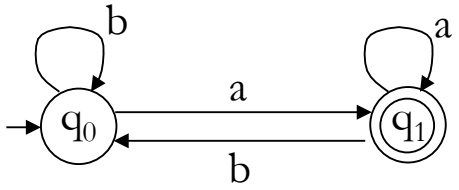


+5 +10 +5 R R +5 +10 are in the language

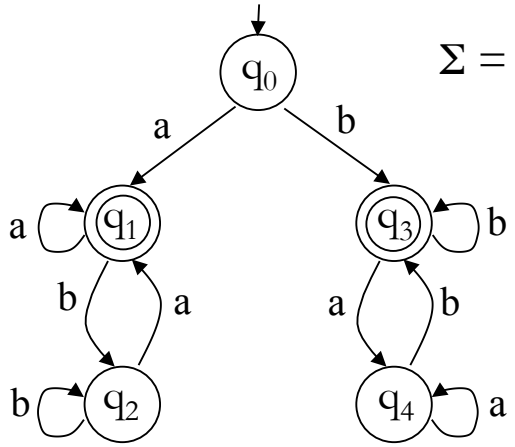
+5 +5 +10 R are not

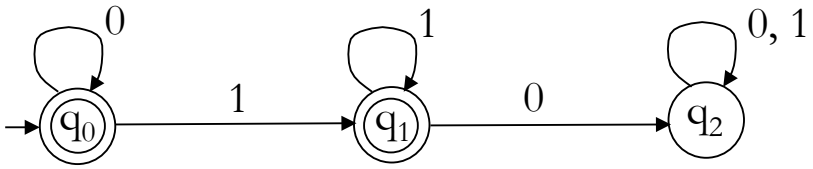
Examples

$\Sigma = \{a, b\}$



$\Sigma = \{a, b\}$





$\Sigma = \{0, 1\}$

What are the languages of these DFAs?

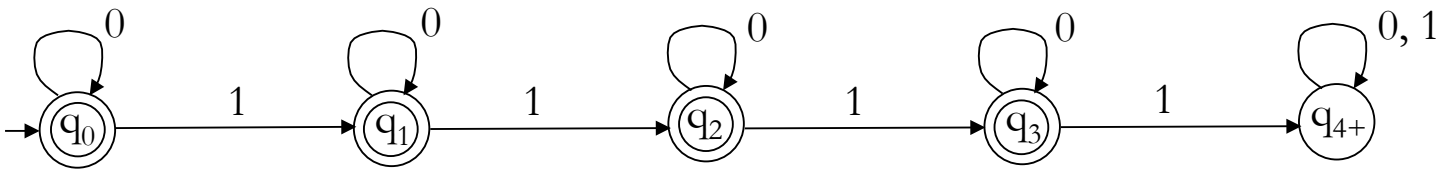
Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings with at most three 1s

Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings with at most three 1s

- Answer



Examples

- Construct a DFA that accepts the language

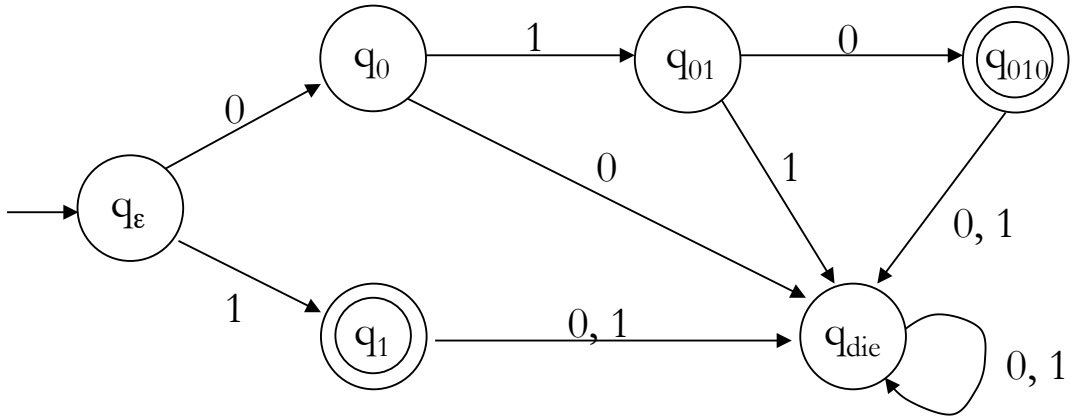
$$L = \{010, 1\} \qquad (\Sigma = \{0, 1\})$$

Examples

- Construct a DFA that accepts the language

$$L = \{010, 1\} \qquad (\Sigma = \{0, 1\})$$

- Answer

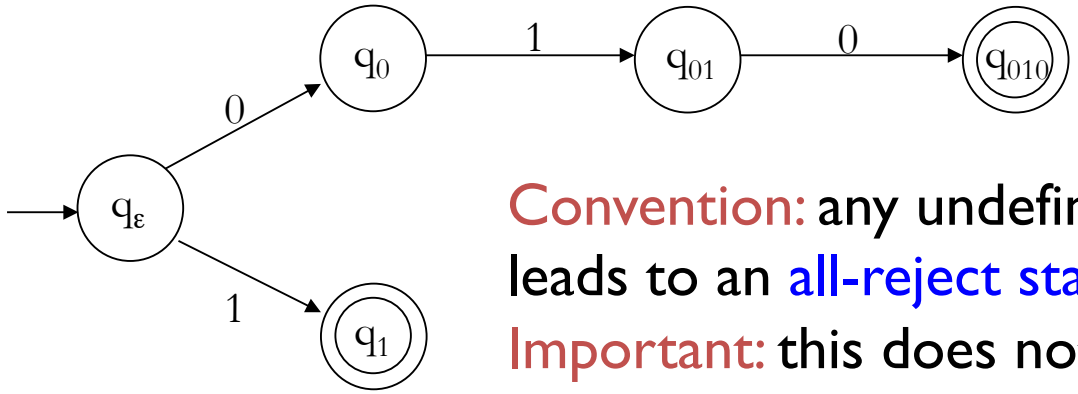


Examples

- Construct a DFA that accepts the language

$$L = \{010, 1\} \qquad (\Sigma = \{0, 1\})$$

- Answer



Convention: any undefined transition leads to an all-reject state (dead state)
Important: this does not add any power
Crucial for MCQs: this state still counts even though it is not drawn

Examples

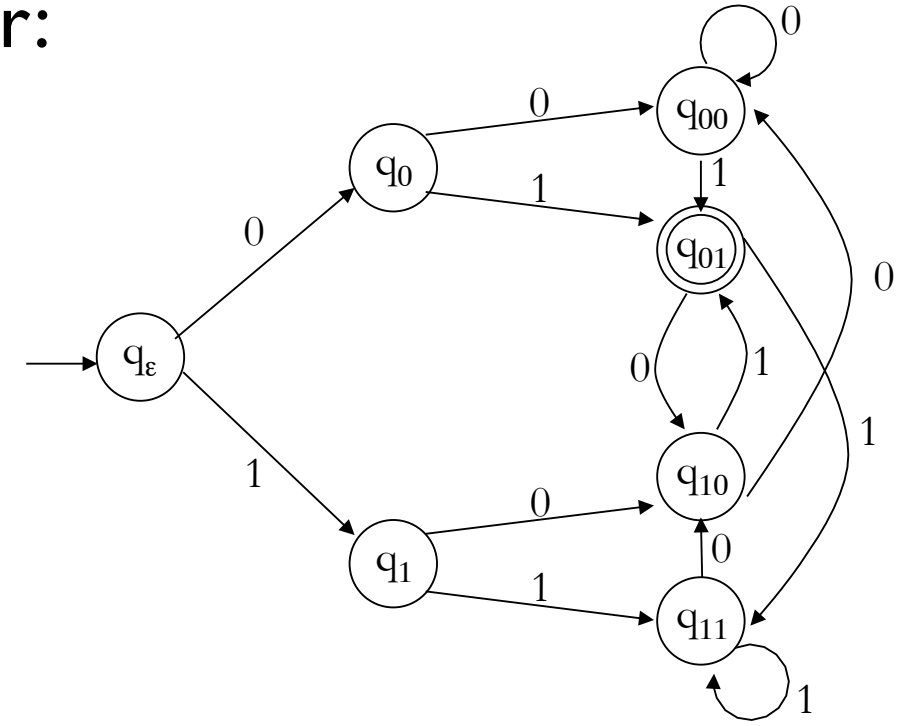
- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 01

Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 01
- Hint: The DFA must "remember" the last 2 bits of the string it is reading

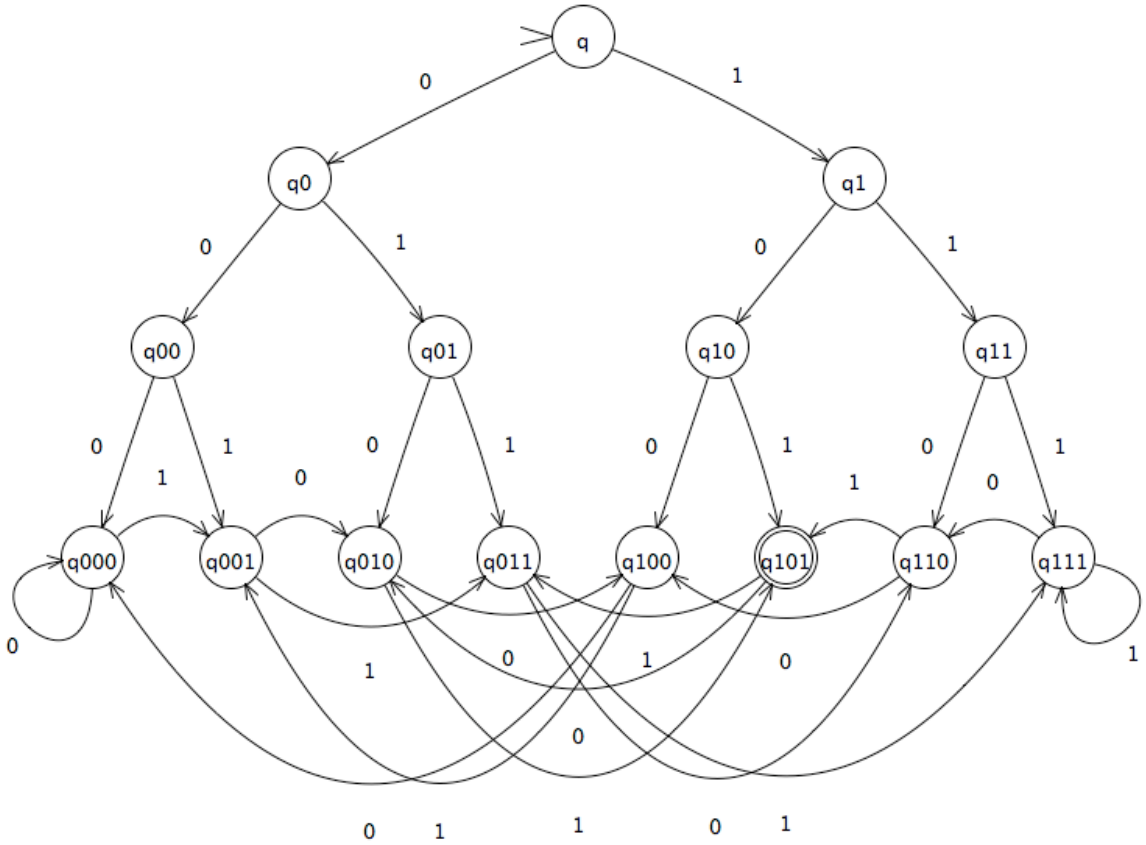
Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 01
- Answer:



Examples

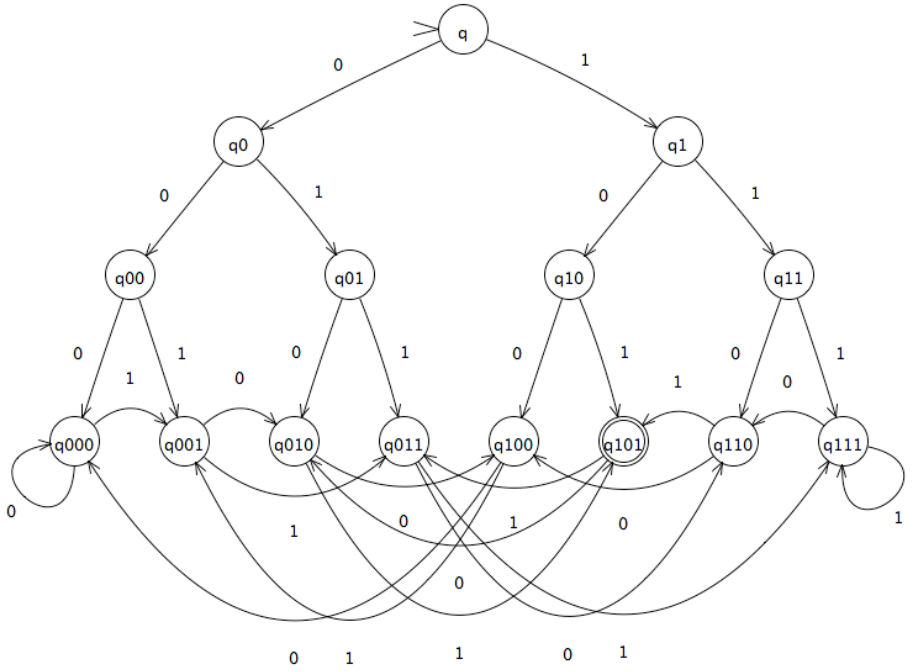
- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 101



Nondeterminism

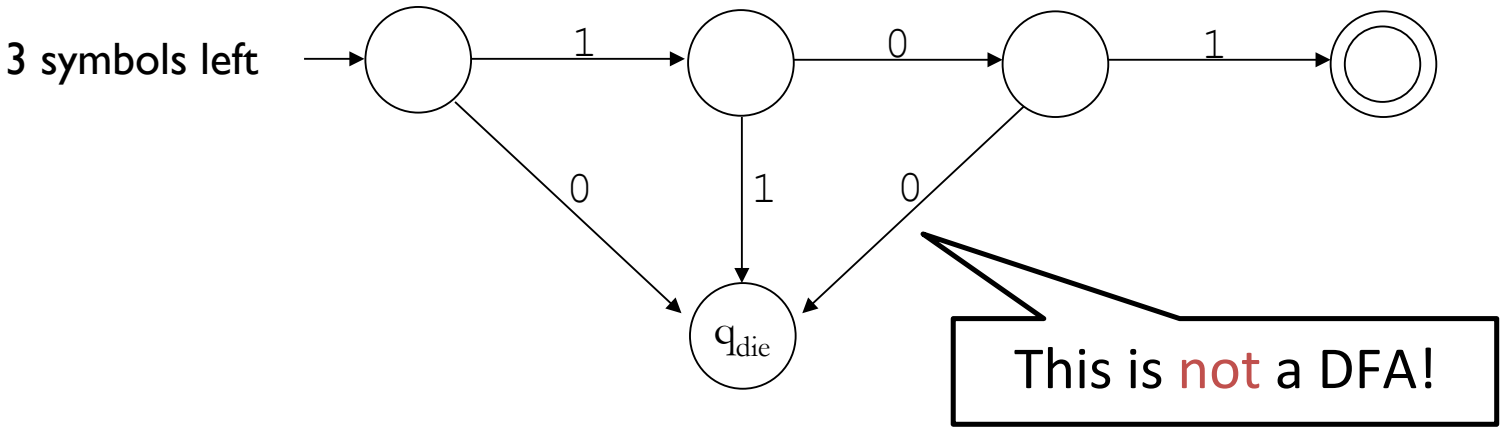
Example from last time

- Construct a DFA over alphabet $\{0, 1\}$ that accepts those strings that end in 101



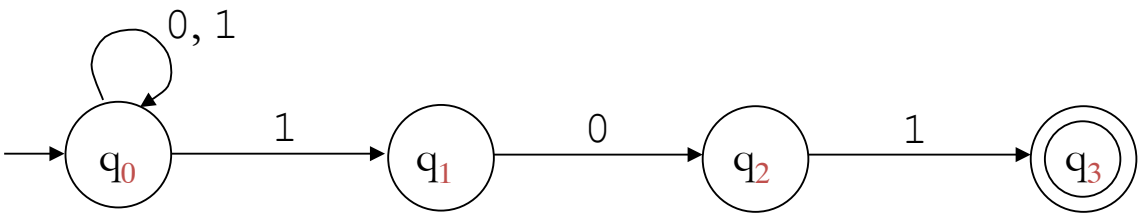
Would be easier if…

- Suppose we could guess when the string we are reading has only 3 symbols left



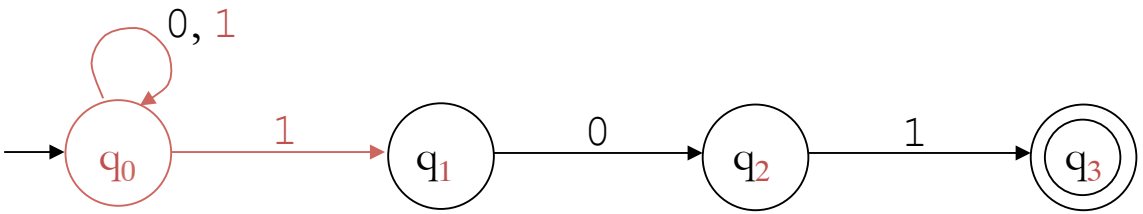
Nondeterministic finite automaton (NFA)

- This is a machine that allows us to make guesses



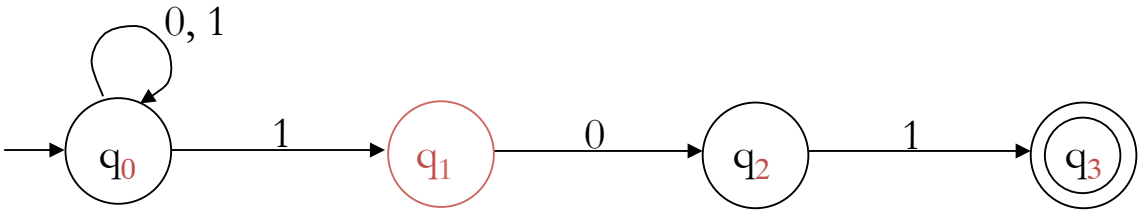
- Each state can have zero, one, or more outgoing transitions labeled by the same symbol

Choosing where to go



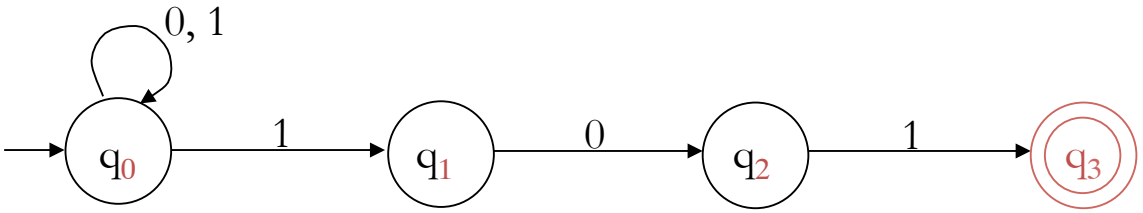
- State q_0 has two transitions labeled 1
- Upon reading 1, we have the choice of staying in q_0 or moving to q_1

The ability to make choices



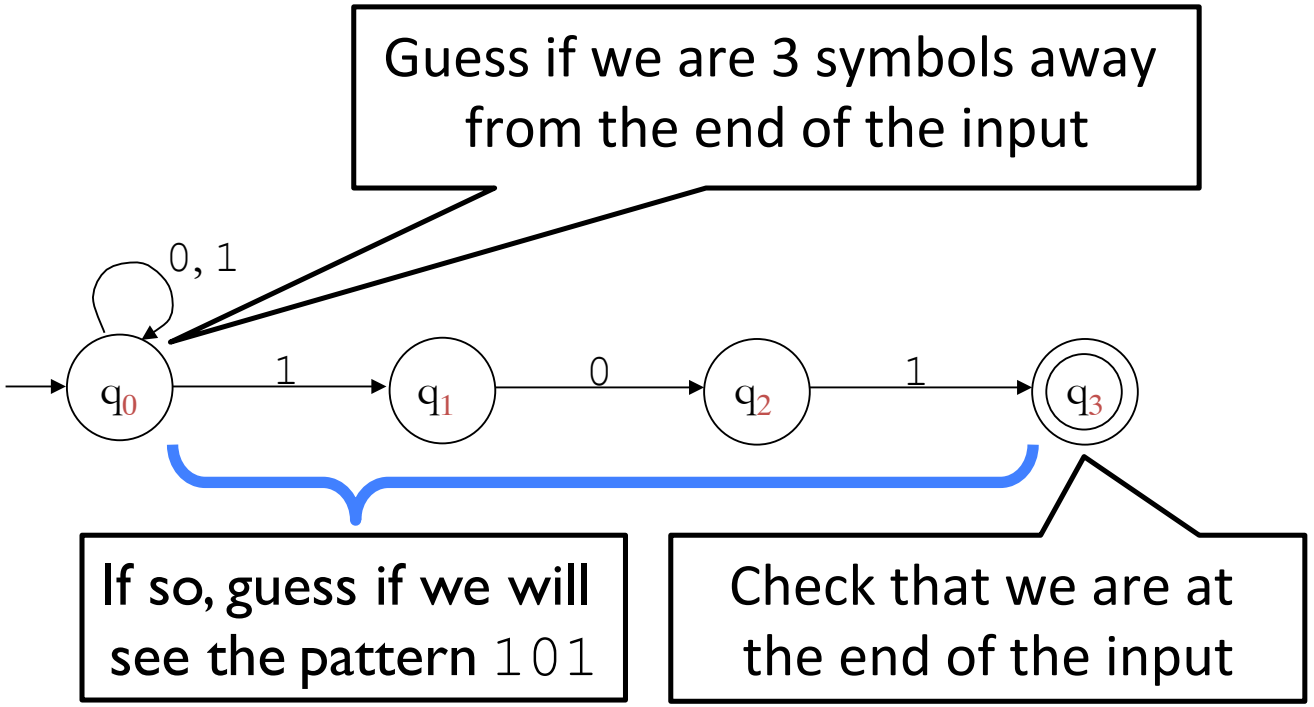
- State q_1 has no transition labeled 1
- Upon reading 1 in q_1, die; upon reading 0, continue to q_2

The ability to make choices

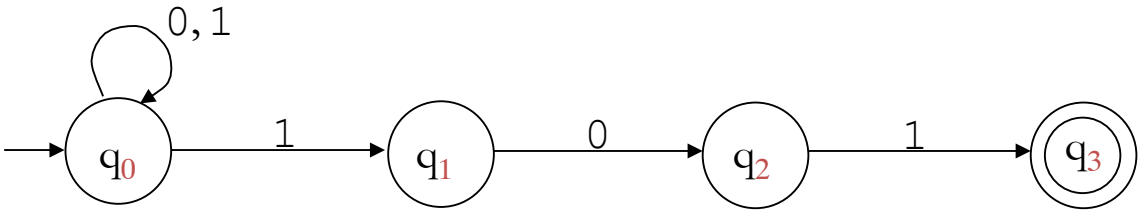


- State q_3 has no transition going out
- Upon reading 0 or 1 in q_3, die

Meaning of NFA



How to run an NFA



input: 0 1 1 0 1

An NFA can reach several different states
after reading a given word

Such a word is accepted by this NFA,
if at least one of these states is final

Example

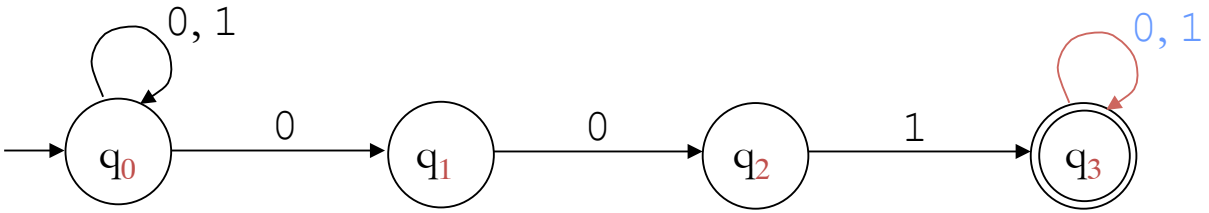
- Construct an NFA over alphabet $\{0, 1\}$ that accepts those strings that contain the pattern 001 somewhere

$11001010, 001001, 111001$	should be accepted
$\varepsilon, 000, 010101$	should not

Example

- Construct an NFA over alphabet $\{0, 1\}$ that accepts those strings that contain the pattern 001 somewhere

- Answer



Definition

- A nondeterministic finite automaton (NFA)

  is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

  - Q is a finite set of states
  - Σ is an alphabet
  - $\delta: Q \times \Sigma \rightarrow$ subsets of Q is a transition function
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$ is a set of accepting states (or final states).

- Differences from DFA:
  - transition function δ can go into several states

Definition

- An ε−nondeterministic finite automaton (ε-NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set of states
  - Σ is an alphabet
  - $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow$ subsets of Q is a trans. function
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$ is a set of accepting states (or final states).

- Differences from NFA:
  - It allows ε-transitions

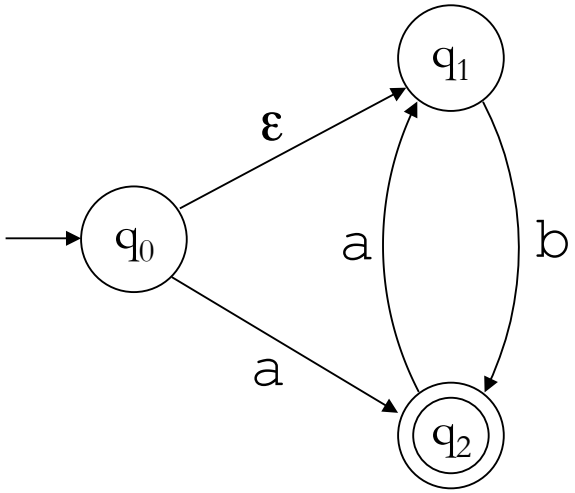
Language of an NFA / ε−NFA

The NFA / ε−NFA accepts string $x \in \Sigma^*$ if there is some path that, starting from q_0, leads to an accepting state as the string is read left to right.

The language of an NFA / ε−NFA is the set of all strings that the NFA / ε−NFA accepts.

ε-transitions

- These can be taken for free:



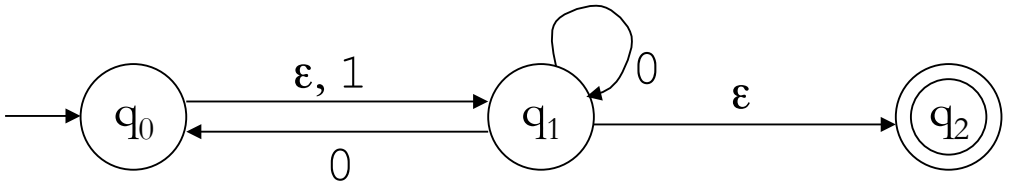
accepts:

a, b, aab, bab, aabab, …

rejects:

ε, aa, ba, bb, …

Example



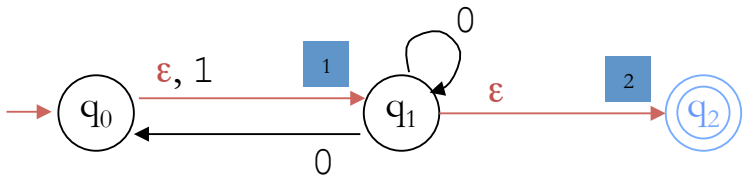
alphabet $\Sigma = \{0, 1\}$
states $Q = \{q_0, q_1, q_2\}$
initial state q_0
accepting states $F = \{q_2\}$

table of
transition function δ:

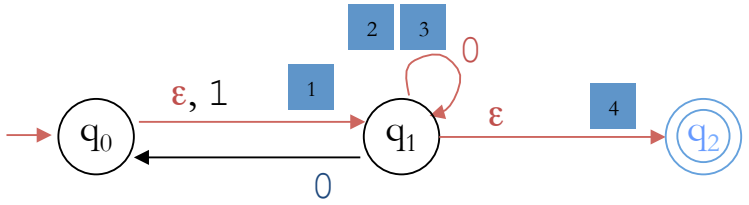
states \ inputs	0	1	ε
q_0	$\varnothing$	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_0, q_1\}$	$\varnothing$	$\{q_2\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$

Examples

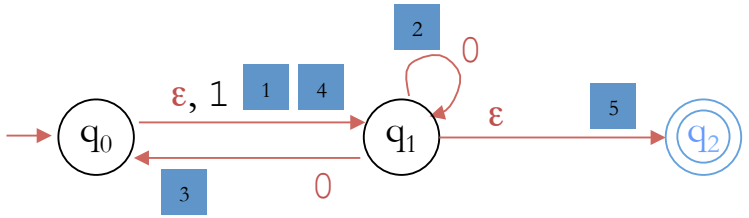
ε



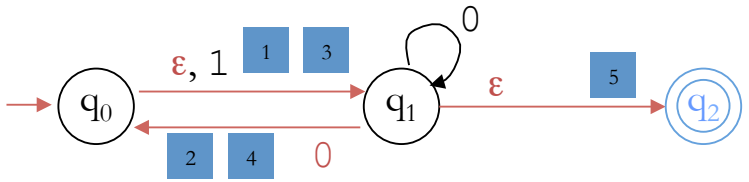
0 0



or

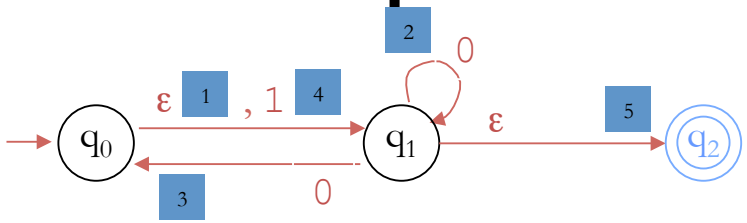


or

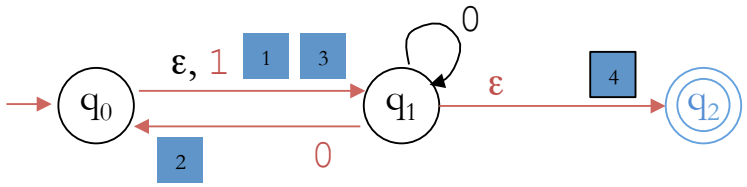


Examples

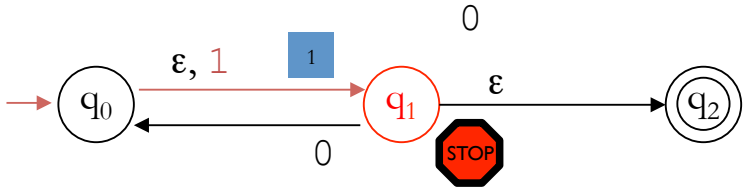
001

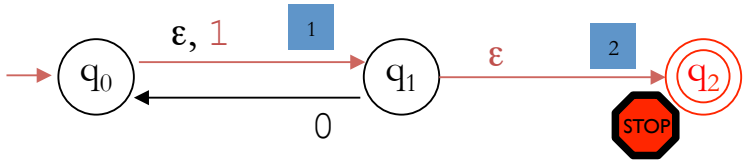


101

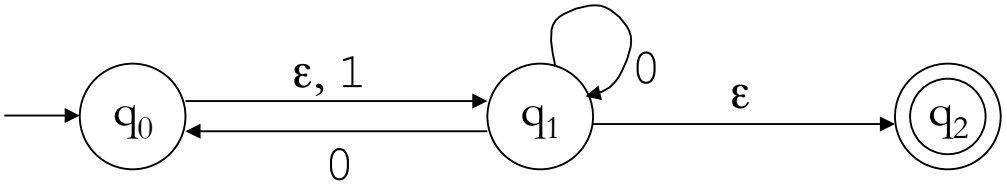


11





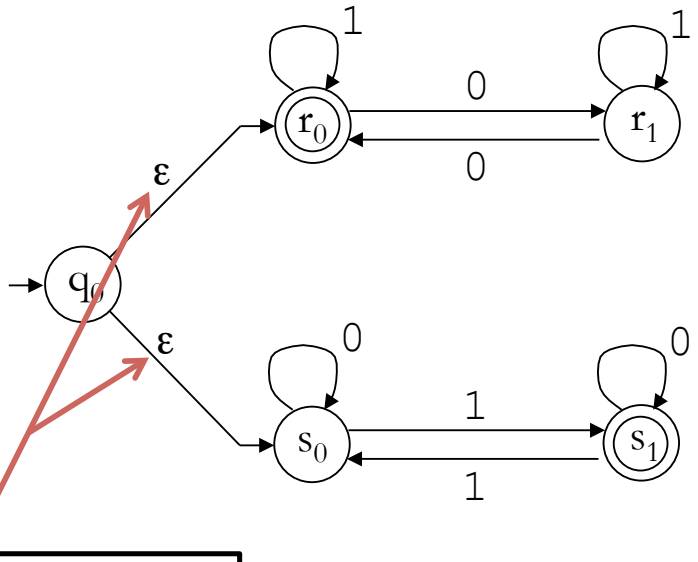
Language of this ε−NFA



Can you guess the language of this ε−NFA?

Example of ε-transitions

- Construct an ε−NFA that accepts all strings with an even number of `0`s or an odd number of `1`s



even number of `0`s

odd number of `1`s

ε-transitions can be taken for free (without reading input)

ε−NFA to DFA conversion and regular expressions

(ε−)NFAs are as powerful as DFAs

- An (ε−)NFA can do everything a DFA can do
- How about the other way?

YES

Every (ε−)NFA can be converted into a DFA for the same language.

ε−NFA → DFA in two steps

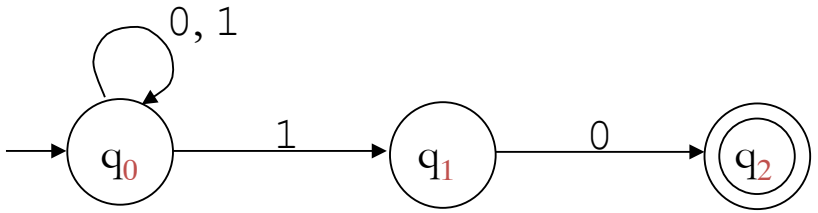
❶ Eliminate ε-transitions (ε−NFA → NFA)

❷ Convert NFA → DFA

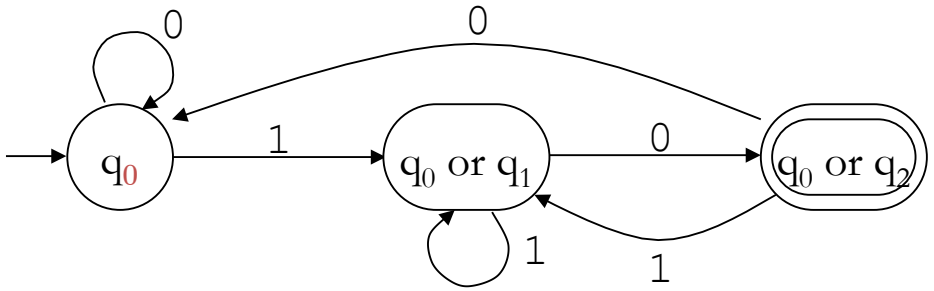
We do this first

NFA → DFA: intuition

NFA:

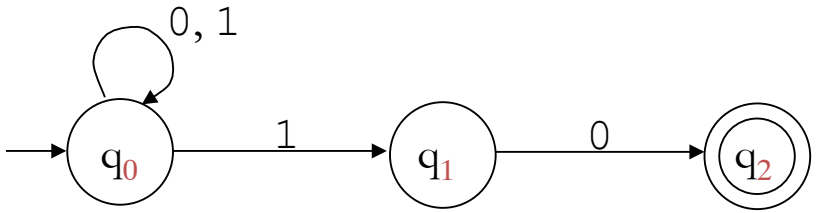


DFA:

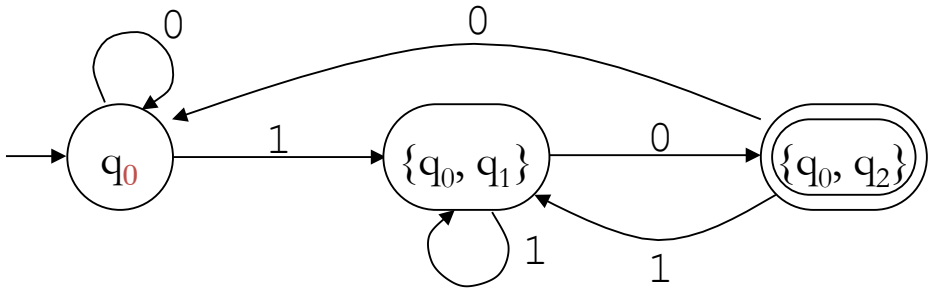


NFA → DFA: intuition

NFA:

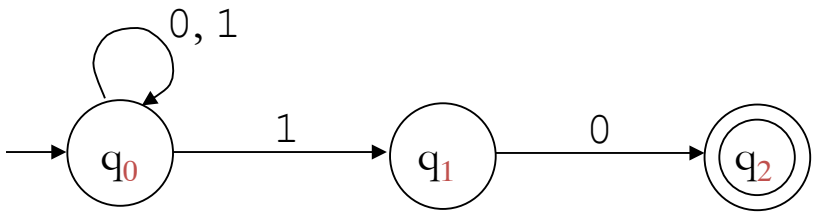


DFA:

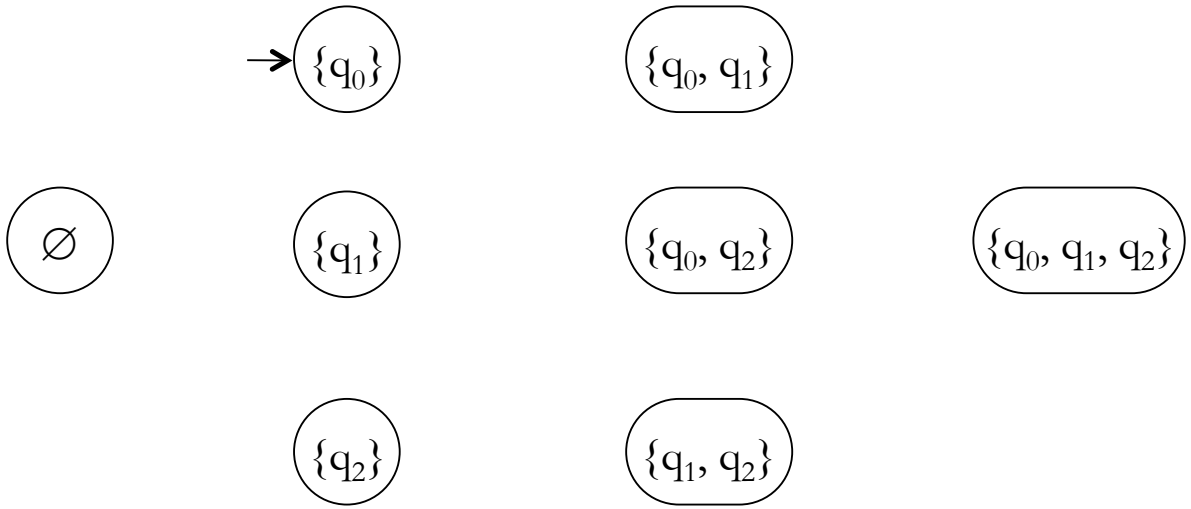


NFA → DFA: states

NFA:



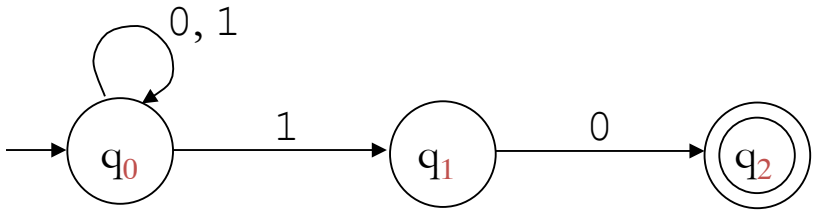
DFA:



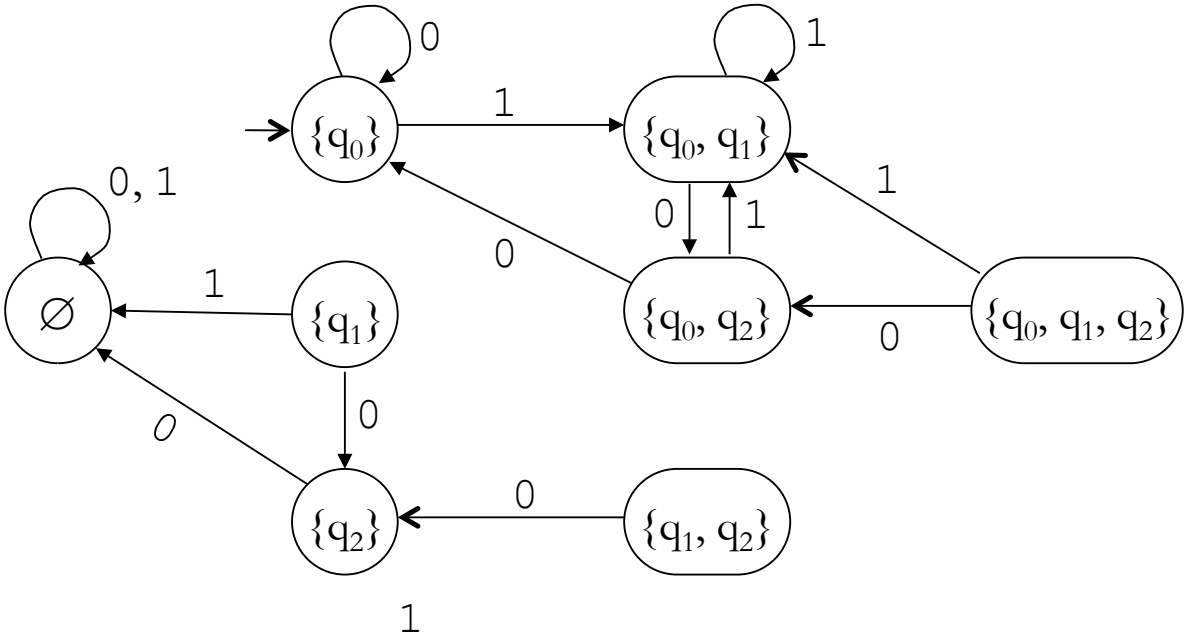
DFA has a state for every subset of NFA states

NFA → DFA: transitions

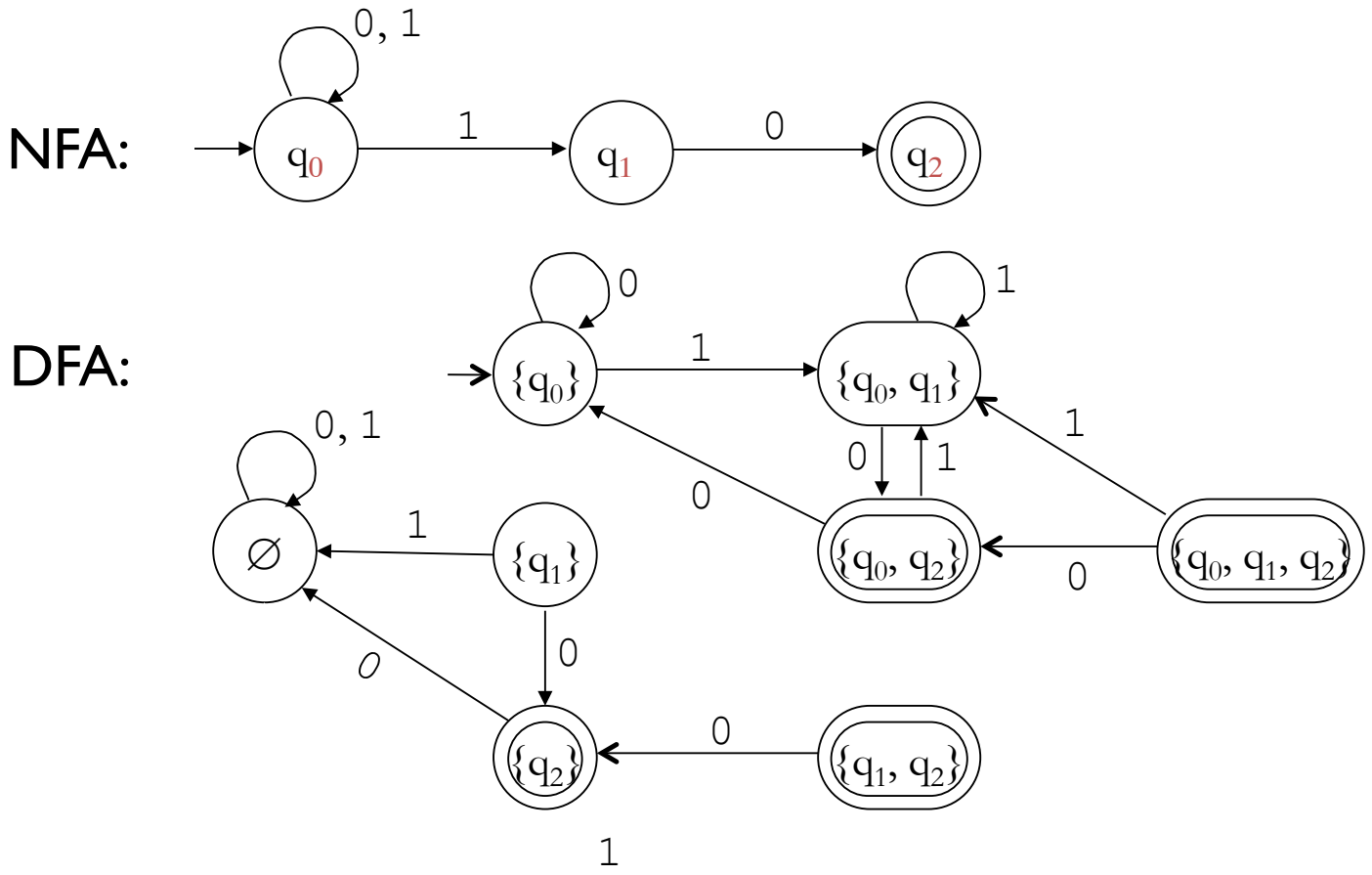
NFA:



DFA:



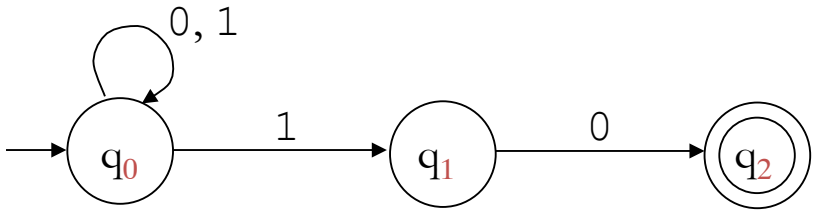
NFA → DFA: accepting states



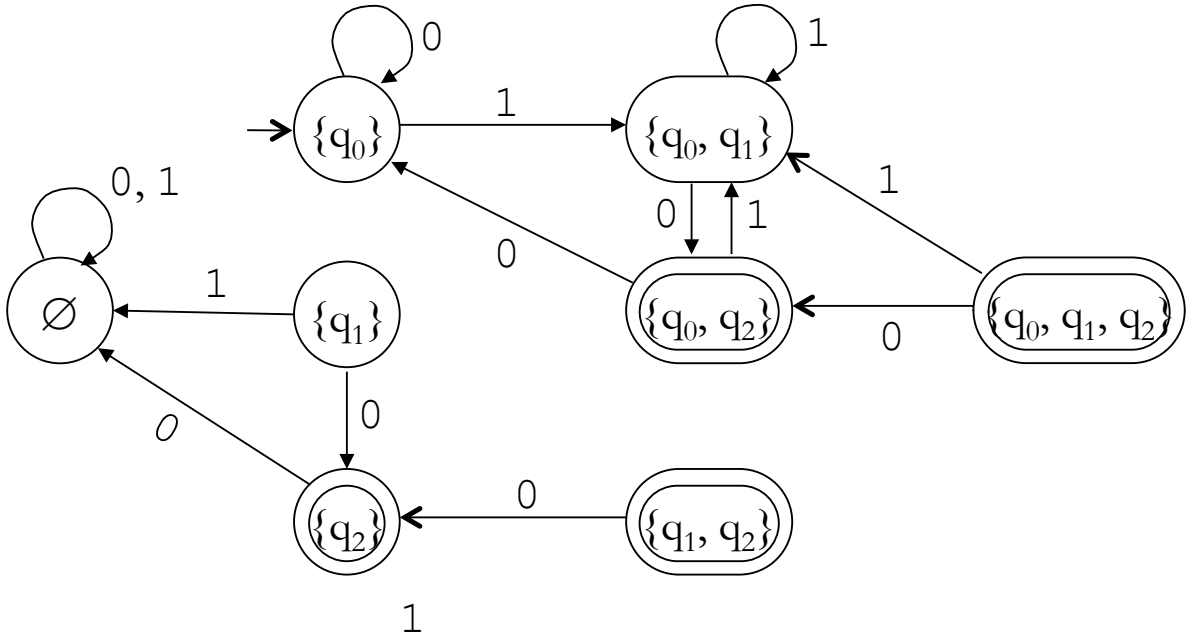
DFA accepts if the state contains a final state

NFA → DFA: dead state elimination

NFA:



DFA:



At the end, you can eliminate the unreachable states

The general method

	NFA	DFA
states	$q_0, q_1, \ldots, q_n$	$\varnothing, \{q_0\}, \{q_1\}, \{q_0,q_1\}, \ldots, \{q_0,\ldots,q_n\}$ one for each subset of states in the NFA
initial state	q_0	$\{q_0\}$
transitions	δ	$\delta'(\{q_{i1},\ldots,q_{ik}\}, a) =$ $\delta(q_{i1}, a) \cup \ldots \cup \delta(q_{ik}, a)$
accepting states	$F \subseteq Q$	$F' = \{S: S \text{ contains some state in } F\}$

This technique is called the subset construction and is also used in AI.

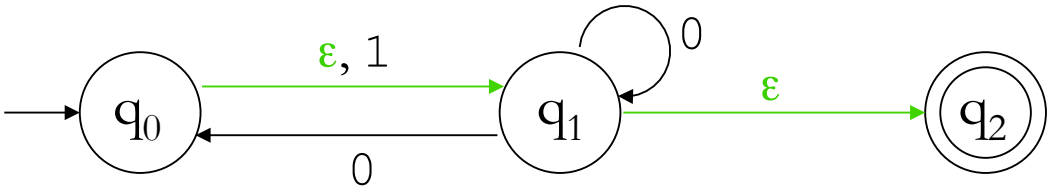
ε−NFA → DFA in two steps

❶ Eliminate ε-transitions (ε−NFA → NFA)

❷ Convert NFA → DFA ✔

Eliminating ε-transitions

ε-NFA:



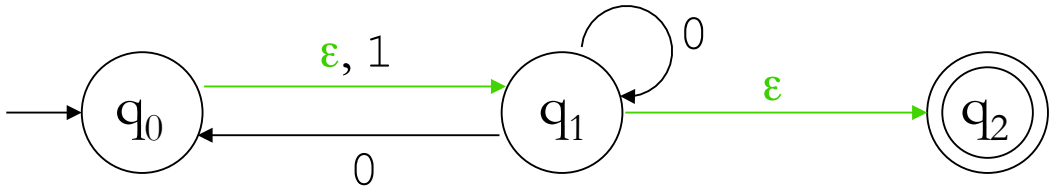
ε-NFA without εs:

	0	1
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
q_1	$\{q_0, q_1, q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$

Accepting states: q_2 , q_1, q_0

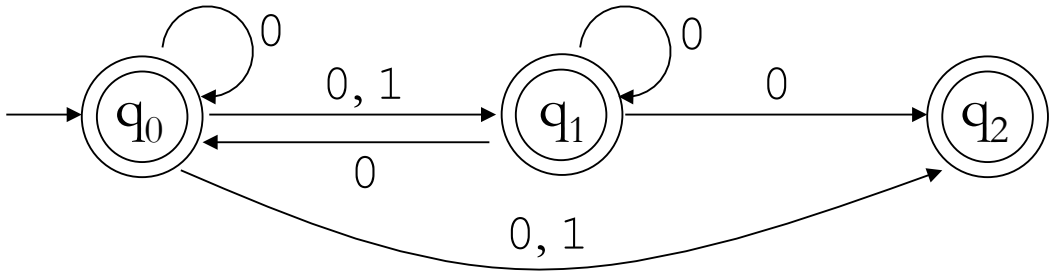
Eliminating ε-transitions

ε-NFA:



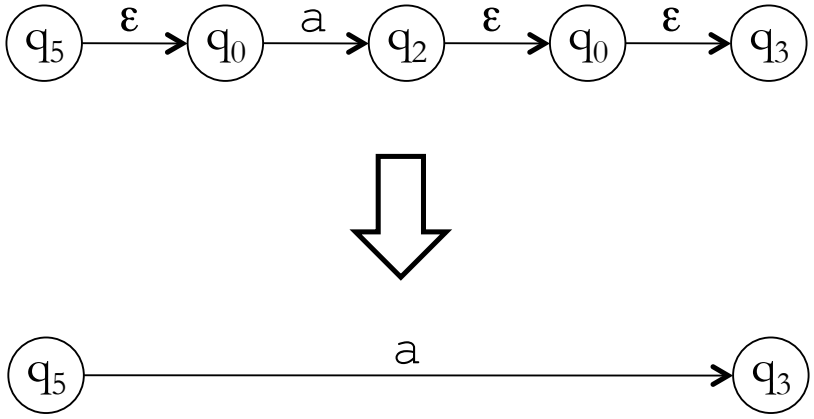
new NFA:

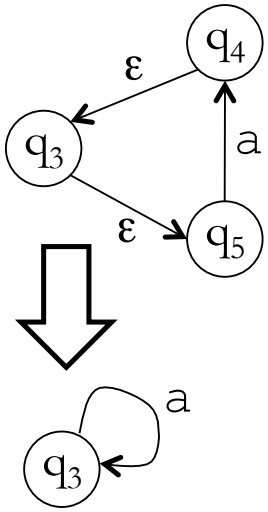
	0	1
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
q_1	$\{q_0, q_1, q_2\}$	$\varnothing$
q_2	$\varnothing$	$\varnothing$



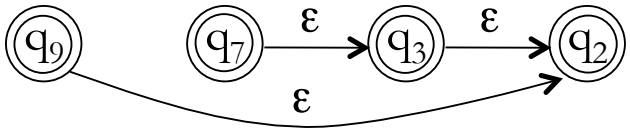
Eliminating ε-transitions

- Paths with εs are replaced by a single transition





- States that can reach final state by ε are all accepting

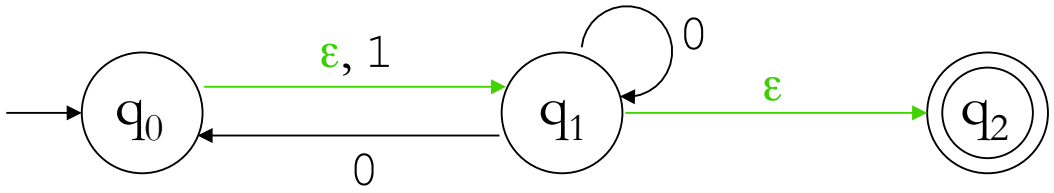


ε−closure

CL(A) = {q ∈ Q : q can be reached from any state in A
using 0 or more ε-transitions}

Note that always A ⊆ CL(A) and CL(Q) = Q.

ε-NFA:



$CL(\{q_0\}) = \{q_0, q_1, q_2\}$

$CL(\{q_1\}) = \{q_1, q_2\}$

$CL(\{q_2\}) = \{q_2\}$

$CL(\{q_0, q_1\}) = \{q_0, q_1, q_2\}$

$CL(\{q_0, q_1, q_2\}) = \{q_0, q_1, q_2\}$

$CL(\varnothing) = \varnothing$

General method

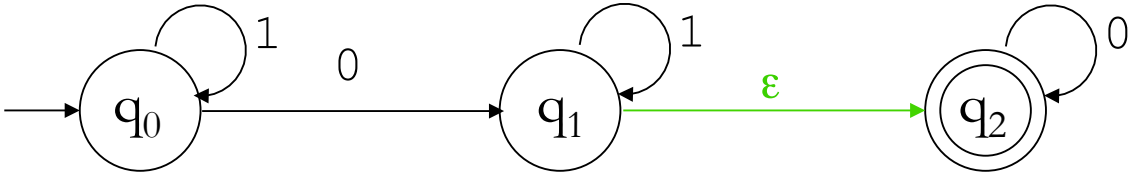
CL(A) = {q $\in$ Q : q can be reached from any state in A

using 0 or more ε-transitions}

	ε−NFA	NFA
states	$q_0, q_1, \ldots, q_n$	$q_0, q_1, \ldots, q_n$
initial state	q_0	$\{q_0\}$
transitions	δ	$\delta'(q, a) = CL(\delta(CL(\{q\}), a)) =$ $CL(\bigcup_{q' \in CL(\{q\})} \delta(q', a))$
accepting states	$F \subseteq Q$	$F' = \{q: CL(\{q\}) \cap F \neq \emptyset\}$

Example ε−NFA → NFA

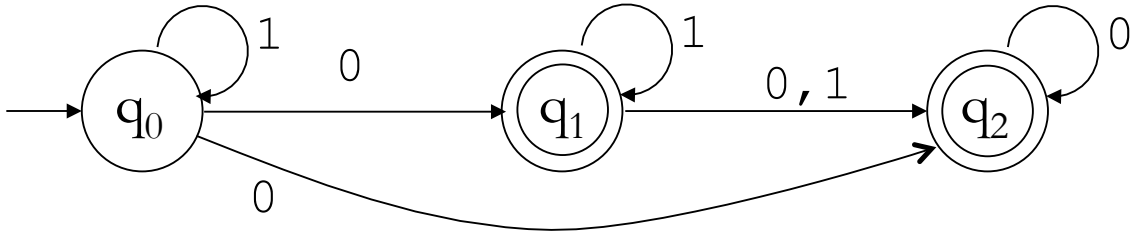
ε-NFA:



$CL(\{q_0\}) = \{q_0\}$ $CL(\{q_1\}) = \{q_1, q_2\}$ $CL(\{q_2\}) = \{q_2\}$

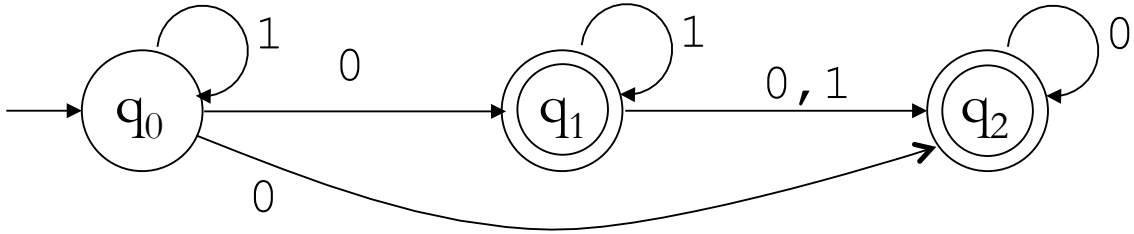
$$CL(\delta(CL(\{q_0\}), 0)) = CL(\delta(q_0, 0)) = CL(\{q_1\}) = \{q_1, q_2\}$$
$$CL(\delta(CL(\{q_0\}), 1)) = CL(\delta(q_0, 1)) = CL(\{q_0\}) = \{q_0\}$$
$$CL(\delta(CL(\{q_1\}), 0)) = CL(\delta(\{q_1, q_2\}, 0)) = CL(\{q_2\}) = \{q_2\}$$
$$CL(\delta(CL(\{q_1\}), 1)) = CL(\delta(\{q_1, q_2\}, 1)) = CL(\{q_1\}) = \{q_1, q_2\}$$
$$CL(\delta(CL(\{q_2\}), 1)) = CL(\delta(\{q_2\}, 1)) = CL(\varnothing) = \varnothing \qquad \delta'(q_2, 0) = \{q_2\}$$

NFA:



Example NFA → DFA

NFA:



Regular expressions

String concatenation

The concatenation (aka product) of two words is obtained by appending them together to form one long word.

s = abb t = bab

st = abbbab

ts = bababb

ss = abbabb

sst = abbabbbab

$s = x_1 \ldots x_n$ $t = y_1 \ldots y_m$	$st = x_1 \ldots x_n y_1 \ldots y_m$

Notation: s^k, $|s|$ (length, i.e. number of letters in s)

Note: $|st| = |s| + |t|$

Operations on languages

- The concatenation of languages L_1 and L_2 is

$$L_1L_2 = \{st \mid s \in L_1, t \in L_2\}$$

- The n-th power of L^n is

$$L^n = \{s_1s_2...s_n \mid s_1, s_2, ..., s_n \in L\}$$

- The union of L_1 and L_2 is

$$L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$$

Example

$L_1 = \{0, 01\}$

$L_2 = \{\varepsilon, 1, 11, 111, \ldots\}$

any number of 1s

$$L_1L_2 = \{0, 01, 011, 0111, \ldots\} \cup \{01, 011, 0111, \ldots\}$$
$$= \{0, 01, 011, 0111, \ldots\}$$

0 followed by any number of 1s

$$L_1^2 = \{00, 001, 010, 0101\}$$

$$L_2^2 = L_2$$
$$L_2^n = L_2 \quad (n \geq 1)$$

$$L_1 \cup L_2 = \{0, 01, \varepsilon, 1, 11, 111, ...\}$$

Operations on languages

- The star of L are all strings made up of zero or more chunks from L:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$$

- Example: $L_1 = \{01, 0\}$, $L_2 = \{\varepsilon, 1, 11, 111, \ldots\}$. What is ${L_1}^*$ and ${L_2}^*$?

Example

$L_1 = \{0, 01\}$

L_1^*: 00100001 is in L_1^*

00110001 is not in L_1^*

10010001 is not in L_1^*

$L_1^0 = \{\varepsilon\}$

$L_1^1 = \{0, 01\}$

$L_1^2 = \{00, 001, 010, 0101\}$

$L_1^3 = \{000, 0001, 0010, 00101,$
$0100, 01001, 01010, 010101\}$

L_1^* are all strings that start with 0 and do not contain consecutive 1s
(plus the empty string)

(Length, Number of strings) for L_1^*
(0, 1), (1,1), (2, 2), (3, 3), (4, 5), (5, 8), …
How many for length n?

Example

$$L_2 = \{\varepsilon, 1, 11, 111, \ldots\}$$

any number of 1s

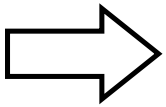
$$L_2^0 = \{\varepsilon\}$$
$$L_2^1 = L_2$$
$$L_2^2 = L_2$$
$$L_2^n = L_2 \quad (n \geq 1)$$

$$L_2^* = L_2^0 \cup L_2^1 \cup L_2^2 \cup \ldots$$
$$= \{\varepsilon\} \cup L_2^1 \cup L_2^2 \cup \ldots$$
$$= L_2$$

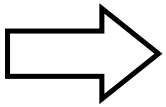
$$L_2^* = L_2$$

Combining languages

- We can construct languages by starting with simple ones, like $\{0\}$, $\{1\}$ and combining them

$\{0\}(\{0\} \cup \{1\})^*$  $0(0 + 1)^*$

all strings that start with 0

$(\{0\}\{1\}^*) \cup (\{1\}\{0\}^*)$  $01^* + 10^*$

0 followed by any number of 1s, or
1 followed by any number of 0s

Regular expressions

- A regular expression over Σ is an expression formed using the following rules:
  - The symbols $\emptyset$ and ε are regular expressions
  - Every a in Σ is a regular expression
  - If R and S are regular expressions, so are $R+S$, RS and R^*.

$\emptyset$ $\quad$ ε $\quad$ $0(0+1)^*$ $\quad$ $1^*(\varepsilon + 0)$ $\quad$ $01^* + 10^*$ $\quad$ $(0+1)^*01(0+1)^*$

A language is regular if it is represented by a regular expression

Analysing regular expressions

$\Sigma = \{0, 1\}$

$01* = 0(1*) = \{0, 01, 011, 0111, \ldots\}$

0 followed by any number of 1s

$(01*)(01) = \{001, 0101, 01101, 011101, \ldots\}$

0 followed by any number of 1s and then 01

Analysing regular expressions

$0+1 \quad = \{0, 1\}$ strings of length 1

$(0+1)^* \; = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$ any string

$(0+1)^*010$ any string that ends in 010

$(0+1)^*01(0+1)^*$ any string that contains the pattern 01

Analysing regular expressions

$((0+1)(0+1))^*+((0+1)(0+1)(0+1))^*$

all strings whose length is even or a mutliple of 3
= strings of length $0, 2, 3, 4, 6, 8, 9, 10, 12, \ldots$

$((0+1)(0+1))^*$ strings of even length

$(0+1)(0+1)$ strings of length 2

$((0+1)(0+1)(0+1))^*$ strings of length a multiple of 3

$(0+1)(0+1)(0+1)$ strings of length 3

Analysing regular expressions

$((0+1)(0+1)+(0+1)(0+1)(0+1))^*$

strings that can be broken into blocks,
where each block has length 2 or 3

$(0+1)(0+1)+(0+1)(0+1)(0+1)$ strings of length 2 or 3

$(0+1)(0+1)$ strings of length 2

$(0+1)(0+1)(0+1)$ strings of length 3

Analysing regular expressions

$((0+1)(0+1)+(0+1)(0+1)(0+1))^*$

strings that can be broken into blocks,
where each block has length 2 or 3

ε ✓ 1 ✗ 10 ✓ 011 ✓ $\underbrace{00}\underbrace{110}$ ✓ $\underbrace{01}\underbrace{10}\underbrace{10}\underbrace{110}$ ✓

this includes all strings except those of length 1

$((0+1)(0+1)+(0+1)(0+1)(0+1))^*$ = all strings except 0 and 1

Analysing regular expressions

$(1+01+001)^*(\varepsilon+0+00)$

ends in at most two 0s

there can be at most two 0s between consecutive 1s

---

there are never three consecutive 0s

Guess: $(1+01+001)^*(\varepsilon+0+00) = \{x \mid x \text{ does not contain } 000\}$

ε 00 $01|1|001|01|1|0$ $001|001|0$

Writing regular expressions

- Write a regular expression for all strings with two consecutive 0s.

$\Sigma = \{0, 1\}$

(anything) 00 (anything else)

$(0+1)^*00(0+1)^*$

Writing regular expressions

- Write a regular expression for all strings that do not contain two consecutive 0s.

$\Sigma = \{0, 1\}$

$$\underbrace{111}_{\text{some 1s at the beginning}}\underbrace{011011011010}_{\text{every 0 followed by one or more 1s}}\underbrace{0}_{\text{maybe a 0 at the end}}$$

... 0 followed by one or more 1s (011^*)

... and at most one 0 in the last block $(\varepsilon + 0)$

$$1^*(011^*)^*(\varepsilon + 0)$$

Writing regular expressions

- Write a regular expression for all strings with an even number of 0s.

$\Sigma = \{0, 1\}$

even number of zeros = (two zeros)*

two zeros = 1*01*01*

(1*01*01*)*

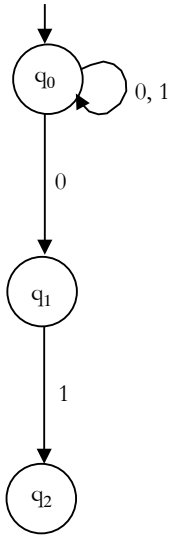
How about all strings with an odd number of 0s?

DFAs, NFAs, εNFAs, and regular expressions

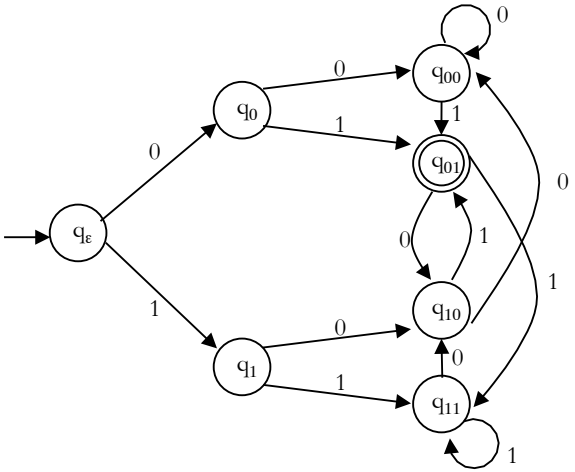
Four ways of doing it

$L = \{x \in \Sigma^* : x \text{ ends in } 01\}$

$\Sigma = \{0, 1\}$



εNFA / NFA

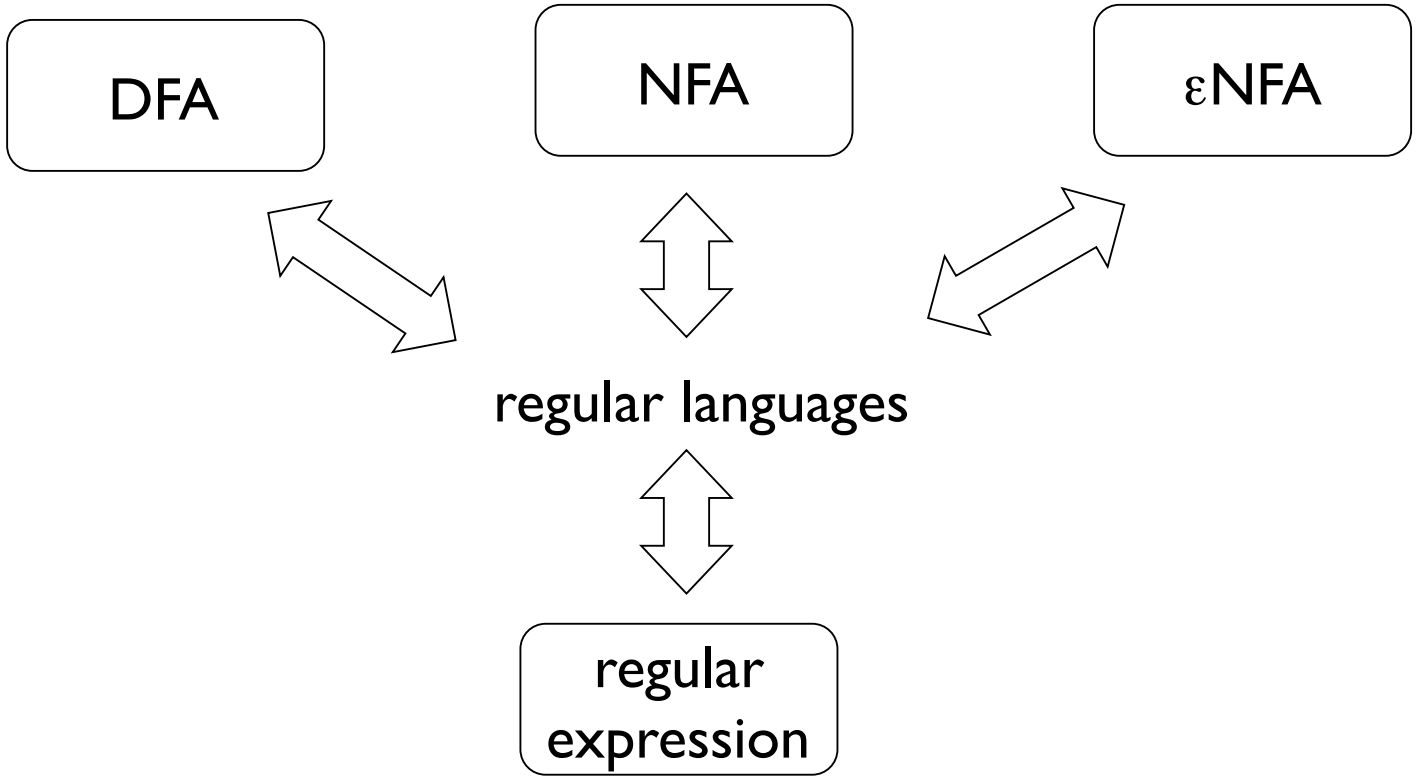


DFA

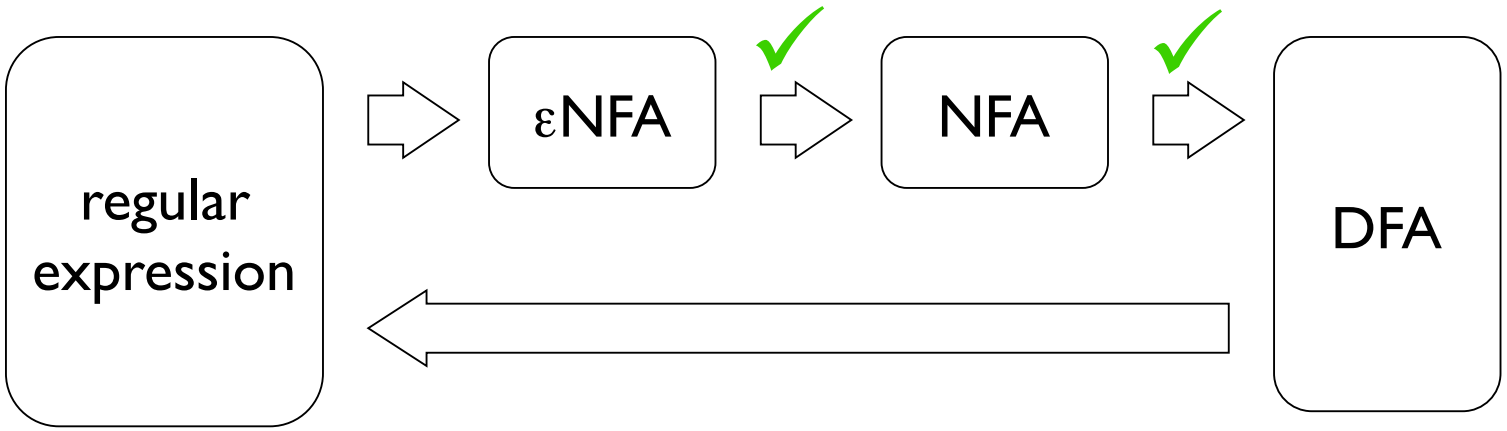
$(0+1)^*01$

regular expression

They are all the same

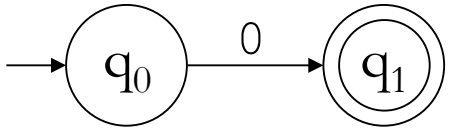


Road map

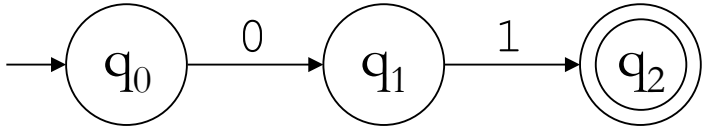


Examples: regular expression → εNFA

$R_1 = 0$

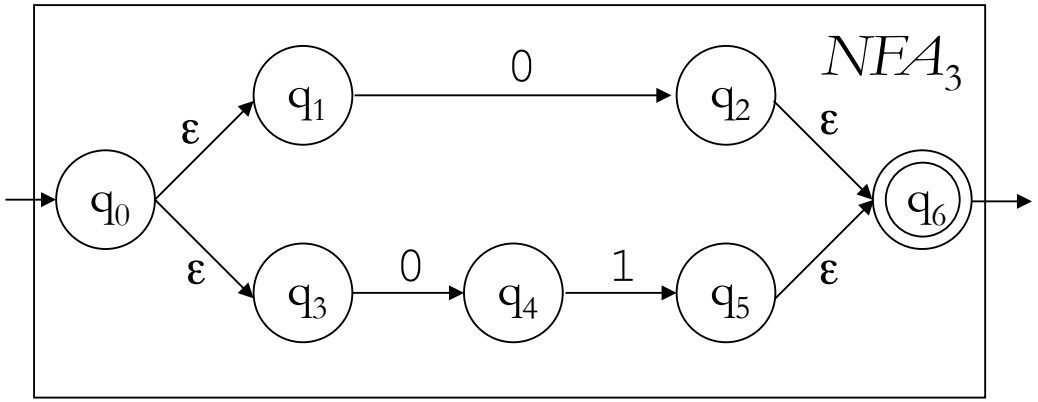


$R_2 = 01$

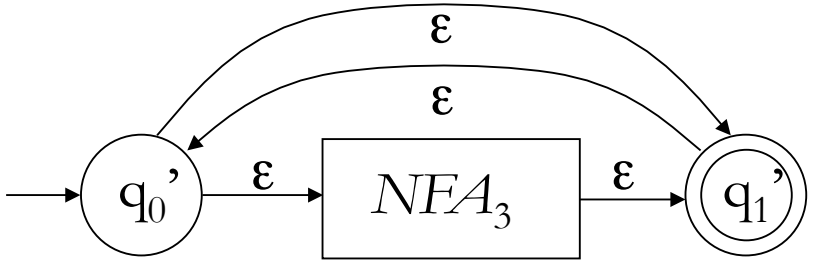


Examples: regular expression → εNFA

$R_3 = 0 + 01$



$R_4 = (0 + 01)^*$



Regular expressions

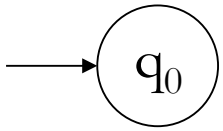
In general, how do we convert a regular expression to an εNFA?

- A regular expression over Σ is an expression formed using the following rules:
  - The symbols $\emptyset$ and ε are regular expressions
  - Every a in Σ is a regular expression
  - If R and S are regular expressions, so are $R+S$, RS and R^*.

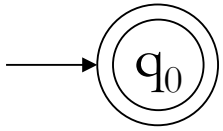
General method

regular expr ⇨ εNFA

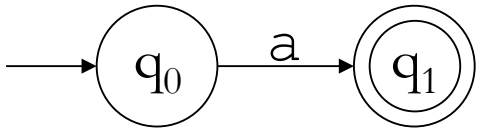
$\emptyset$



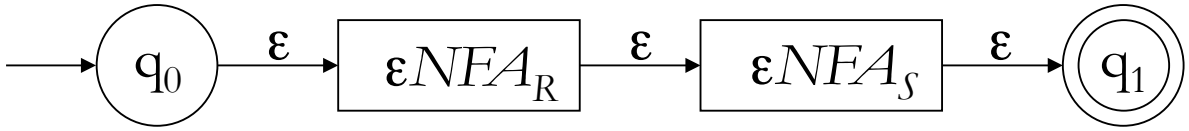
ε



$a \in \Sigma$



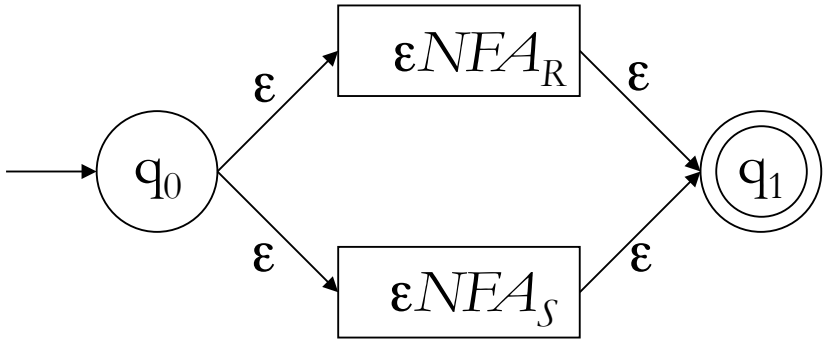
RS



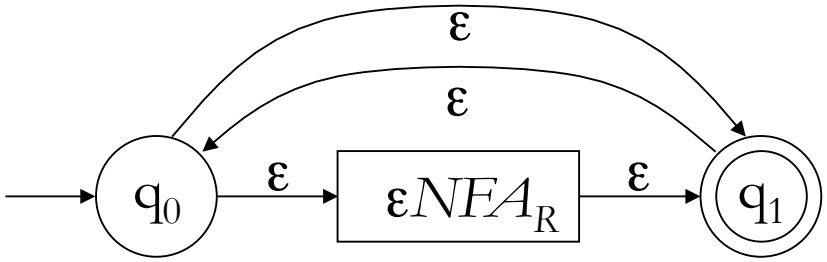
General method continued

regular expr ⇨ εNFA

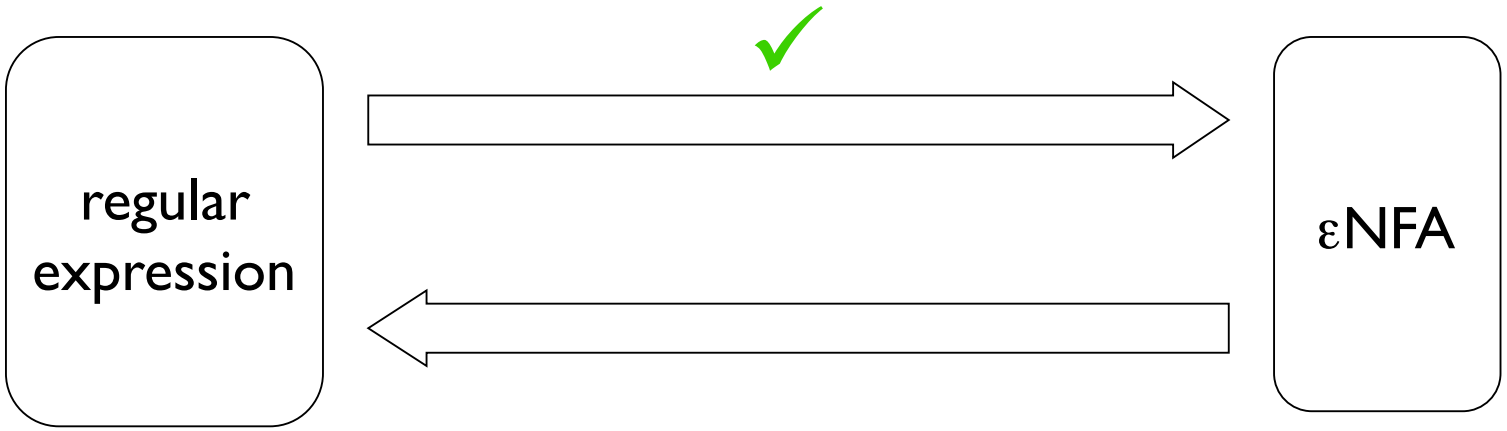
$R + S$



R^*

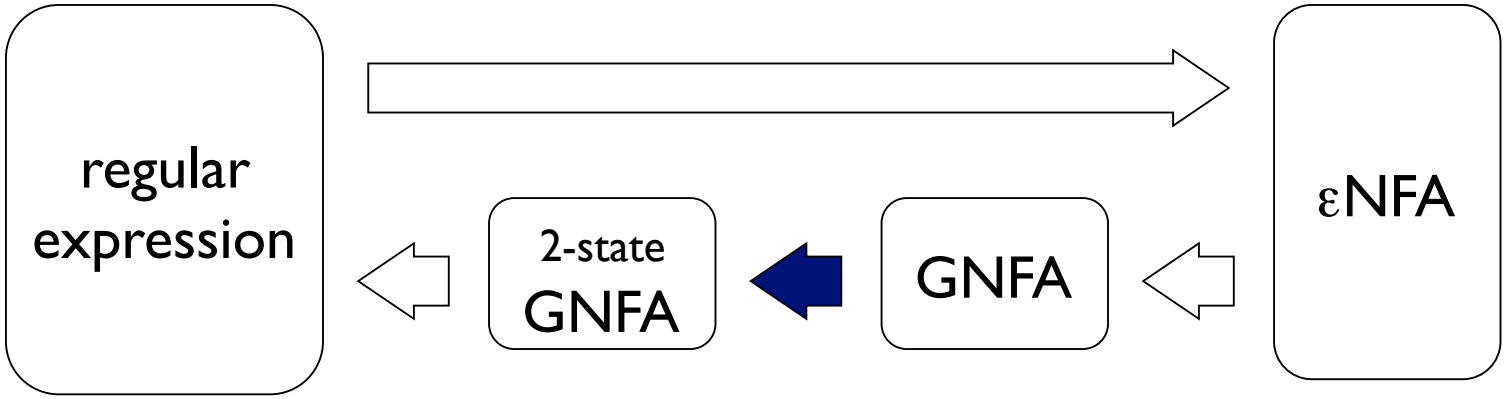


Road map



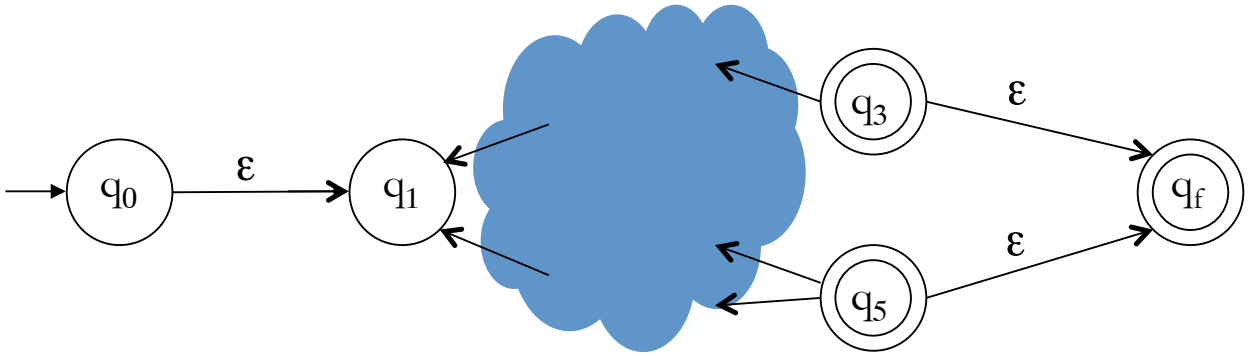
Road map

Road map

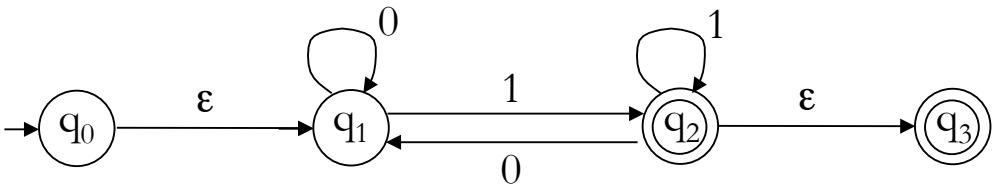


Simplify the εNFA

- First, we simplify the εNFA so that
  - ✓ – It has exactly one accept state
  - ✓ – No arrows come into the start state
  - ✓ – No arrows go out of the accept state



Simplify the εNFA



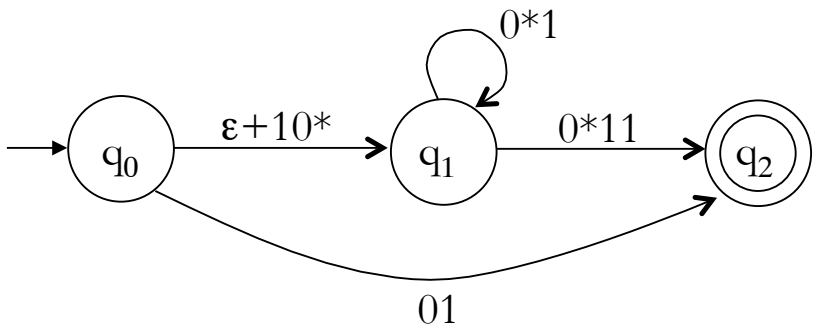
✓ – It has exactly one accept state

✓ – No arrows come into the start state

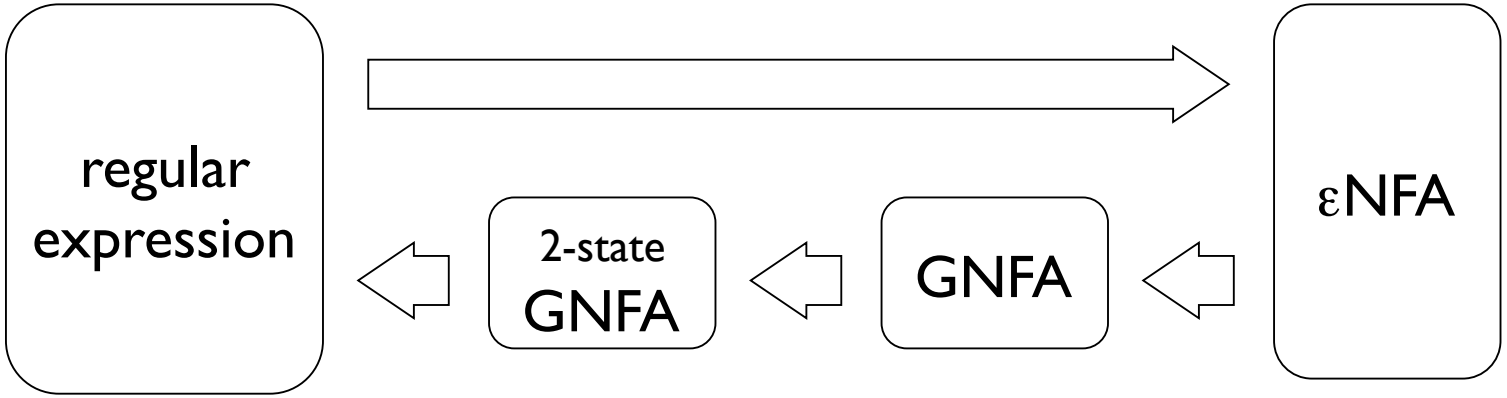
✓ – No arrows go out of the accept state

Generalized NFAs

- A generalized NFA is an εNFA whose transitions are labeled by regular expressions, like

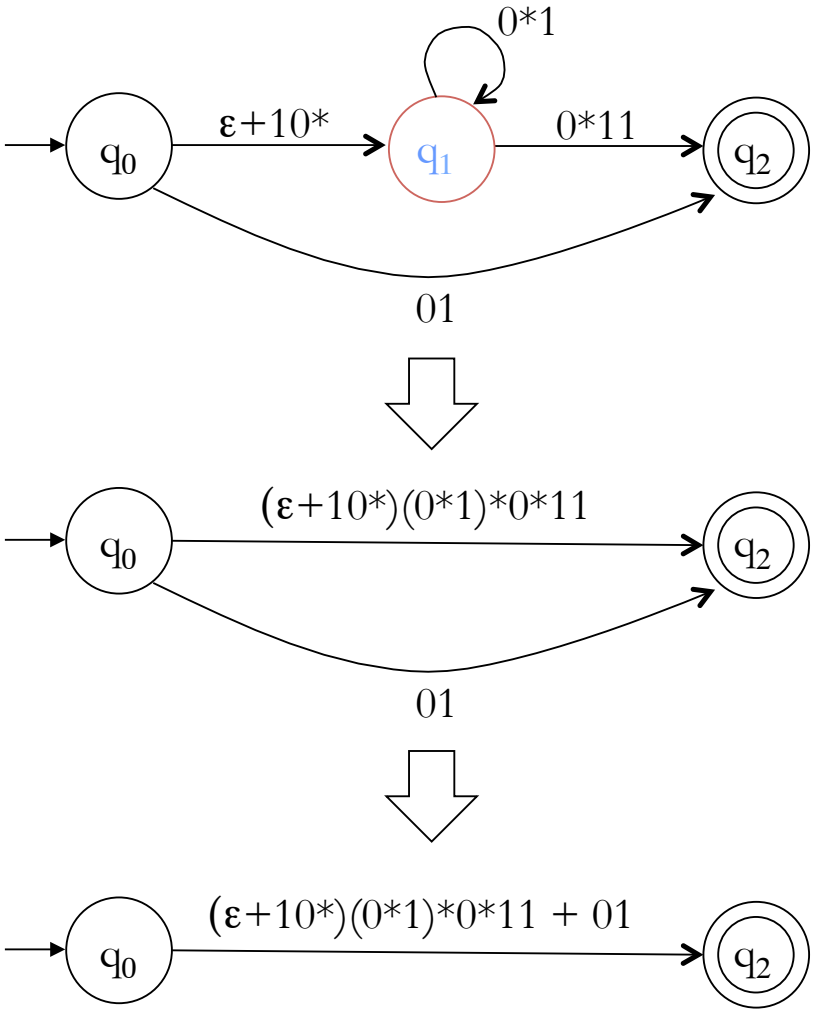


GNFA state reduction



We will eliminate every state but the start and accept states

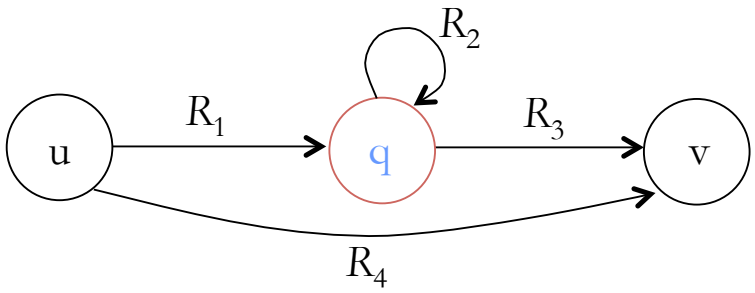
State elimination



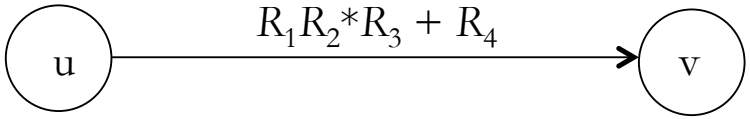
State elimination – general method

- To eliminate state q, for every pair of states (u, v)

Replace

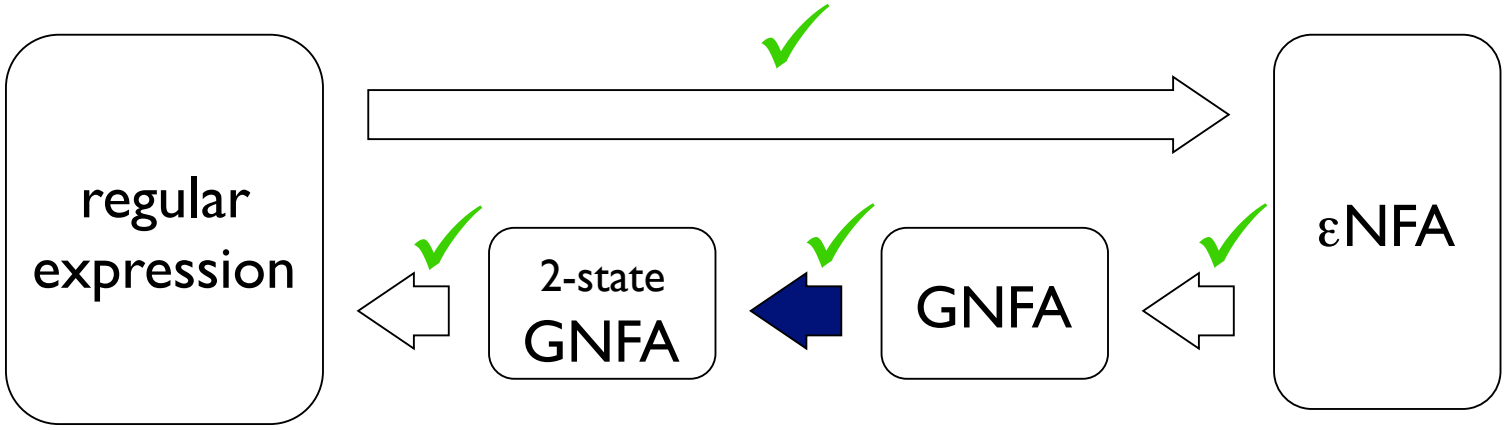


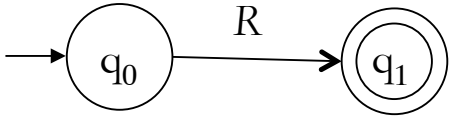
by



remember to do this even when $u = v$!

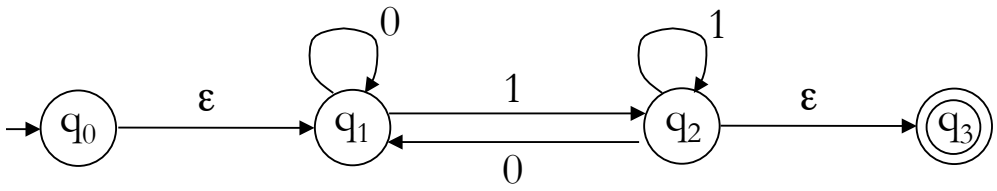
Road map





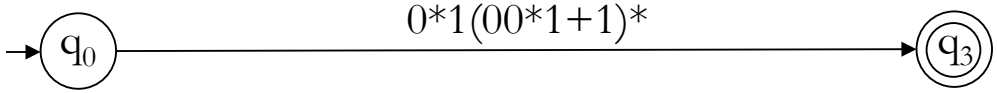
A 2-state GNFA is the same as a regular expression R!

Conversion example



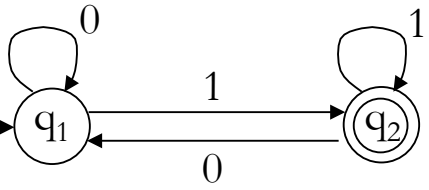
Eliminate q_1:

Eliminate q_2:

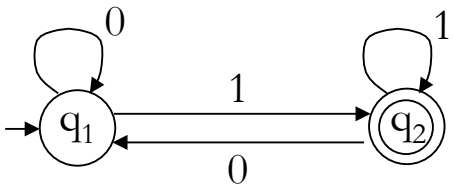


Check:

$$0^*1(00^*1+1)^* \stackrel{?}{=}$$



Check your answer!



All strings that end in 1

$(0 + 1)^*1$

$0^*1(00^*1+1)^*$

$=$

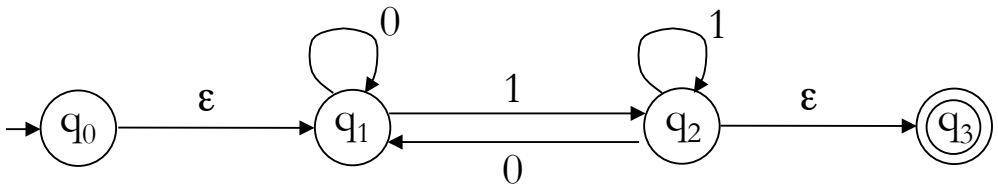
$0^*1(0^*1)^*$

Always ends in 1

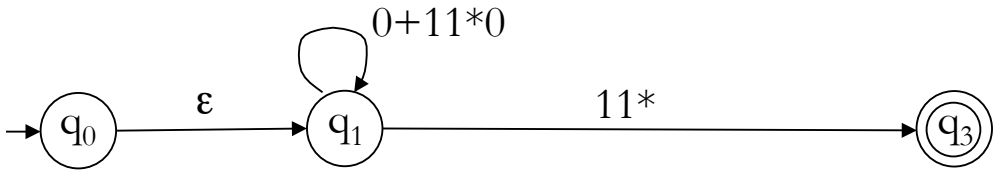
Does every string that ends in 1 have this form?

$01|1|001|0001|01$ Yes!

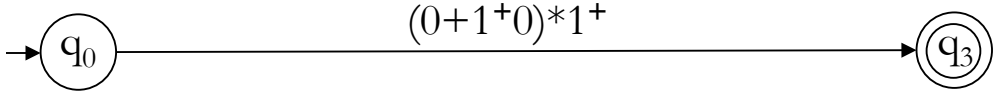
The order matters!



Eliminate q_2:

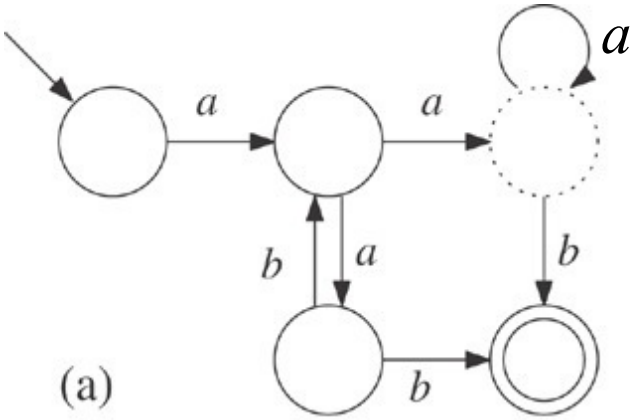


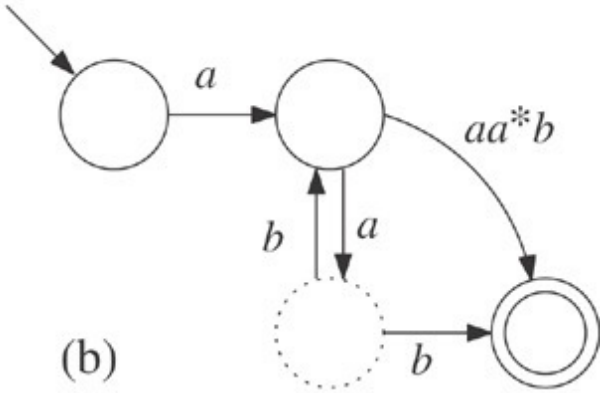
Eliminate q_1:

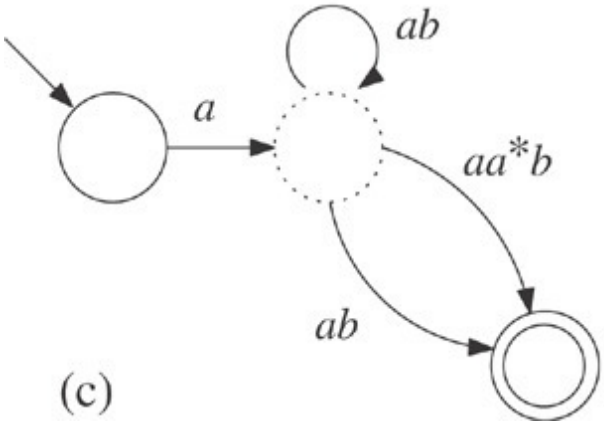


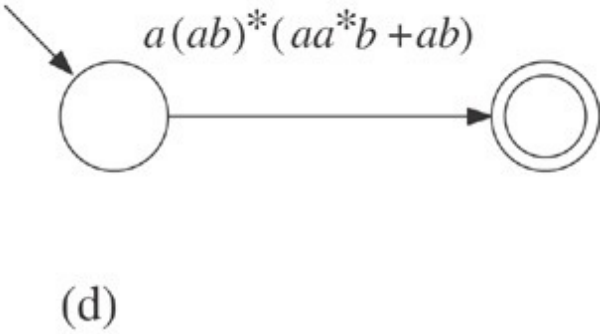
Check: $0^*1(0^+1+1)^* \stackrel{?}{=} (0+1^+0)^*1^+ \stackrel{?}{=} (0+1)^*1$

Another example

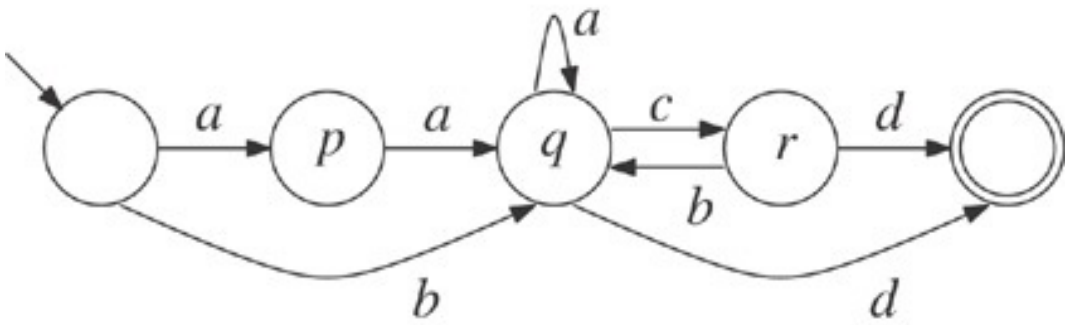








Yet another example



- $E_1 = (aa + b)\ (a + cb)^*\ (cd + d)$ is the output of state elimination in $p \rightarrow r \rightarrow q$ order.
- $E_2 = (aa + b)\ a^*c\ (ba^*c)^*\ (ba^*d + d) + (aa + b)\ a^*d$ is the output of state elimination in $p \rightarrow q \rightarrow r$ order.
- Of course $L(E_1) = L(E_2)$, but try to check it by hand!

COMP 218
Decision, Computation and Language

Dominik Wojtczak

Department of Computer Science
University of Liverpool, U. K.

2019–20

based on slides from Paul Goldberg, Jeff Ullman, Andrej Bogdanov, M. Leucker, M. Andreina Francisco, and Raymond J. Mooney

Algorithm for converting (m+1)-GFA to m-GFA

Input: a GFA M with dedicated start and accept states and at least one additional state.

Output: a GFA M' with dedicated start and accept states and one less state that M, and such that $L(M') = L(M)$.

Notation: Let q_0 be an additional state of the automaton M. Let $q_1, \ldots, q_m$ be the remaining states of M. Let $R_{i \to j}$ be the label of the transition from q_i to q_j, or if there is no such transition, let $R_{i \to j} = \emptyset$.

Construction of M': As the set of states of M' take $\{q_1, \ldots, q_m\}$ and for all $i, j \geq 1$, label the transition from q_i to q_j in M' by $R'_{i \to j} = R_{i \to j} + R_{i \to 0} R^*_{0 \to 0} R_{0 \to j}$.

Proof of correctness: The reason is that every computation path in M that uses q_0 can be shortcut to an equivalent computation path in M'. Similarly, every computation path in M' can be extended to an equivalent computation path in M.

observations, exercise

Recall: any formal language is a well-defined set of words over some given alphabet.

A regular language is one that is given by some regular expression or accepted by some DFA (or NFA).

A regular language will have many different but equivalent DFAs or regexes that represent it.

Which of the following regular expressions over alphabet $\{\mathtt{a}, \mathtt{b}\}$ are equivalent:
$\mathtt{b}(\mathtt{a}^*)\mathtt{b}$, $\mathtt{bb} + \mathtt{ba}(\mathtt{a}^*)\mathtt{b}$,
$\mathtt{ba}(\mathtt{a}^*)$, $\{\mathtt{b}, \epsilon\}\mathtt{a}(\mathtt{a}^*)$, $\{\mathtt{ba}, \mathtt{a}\}(\mathtt{a}^*)$

Equivalences amongst regular expressions

Let R, S and T denote regular expressions. We can note some general rules governing equivalence of regular expressions such as

$$R + S = S + R$$
$$R + (S + T) = (R + S) + T$$
$$R(ST) = (RS)T$$
$$R(S + T) = RS + RT$$

Notice that all of these are essentially the same as the laws of arithmetic, where the alternation corresponds to the sum, and the concatenation to the multiplication.

We can further notice that ϵ behaves like 1 and $\emptyset$ as 0:

$$\emptyset + R = R + \emptyset = R$$
$$\emptyset R = R\emptyset = \emptyset$$
$$\epsilon R = R\epsilon = R$$

However, note that typically $RS \neq SR$ while it is true for regular multiplication.

Equivalences amongst regular expressions

Some properties of closure

$(R^*)^* = R^*$
$R(R^*) = (R^*)R$
$R(R^*) + \epsilon = R^*$
$R(SR)^* = (RS)^*R$
$(R + S)^* = (R^* + S^*)^* = (R^*S^*)^* = R^*(S(R^*))^*$

Example proof

Prove that $(R^*)^* = R^*$.

(*Part*1. $\subseteq$) Suppose $w \in (R^*)^*$.

Then $w = w_1 w_2 \dots w_n$ for $w_i \in R^*$.

$w_i \in R^*$ means that $w_i = w_{i,1} w_{i,2} \dots w_{i,n(i)}$ for $w_{i,j} \in R$.

Hence w is a concatenation of words that belong to R, so $w \in R^*$.

(*Part*2. $\supseteq$) Then, prove that if $w \in R^*$ then $w \in (R^*)^*$.

If $w \in R^*$ then it follows immediately that $w \in (R^*)^*$, since the closure of a language contains all words in that language and possibly more.

Text search

The program `egrep`

`egrep` *`regexp file (or grep -E ...)`*

Searches for the occurrence of patterns matching a regular expression

`cat|12`	{`cat`, `12`}	union
`[abc]`	{`a`, `b`, `c`}	shorthand for `a|b|c`
`[ab][12]`	{`a1`, `a2`, `b1`, `b2`}	concatenation
`(ab)*`	{ε, `ab`, `abab`, ...}	star
`[ab]?`	{ε, `a`, `b`}	zero or one
`(cat)+`	{`cat`, `catcat`, ...}	one or more
`[ab]{2}`	{`aa`, `ab`, `ba`, `bb`}	{*n*} copies

Regular expressions in egrep

- Say we have a file `w.txt`

```
1/1/09   14C   rain,
2/1/09   17C   sunny,
3/1/09   18C   sunny,
                …
```

- Want to know all sunny days

```
> egrep 'sunny' w.txt
```

- Any cloudy days in April '09?

```
> egrep '[0-9]*/4/09 ([0-9]|C| )* cloudy' w.txt
```

- Any consecutive string of 7 sunny days?

```
> egrep '(([0-9]|/|C| )* sunny,){7}' w.txt
```

Searching with `grep`

Words containing `savor` or `savour`

```
egrep "savou?r" words
```

```
outsavor        savorsome
savor           savory
savored         savour
savorer         unsavored
savorily        unsavoredly
savoriness      unsavoredness
savoringly      unsavorily
savorless       unsavoriness
savorous        unsavory
```

Words with 5 consecutive `a` or `b`

```
egrep "[ab]{5}" words
```

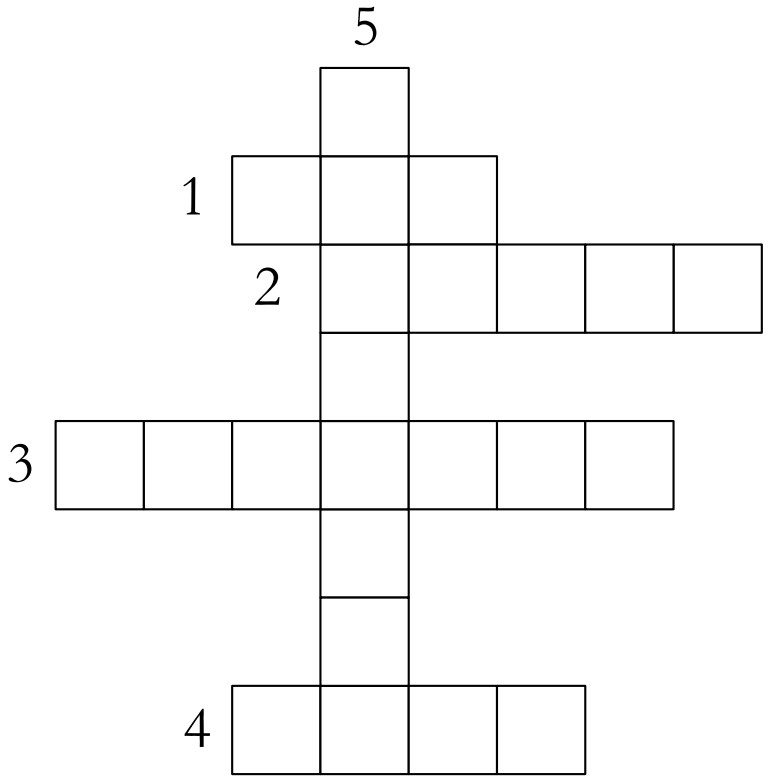
```
grabbable
```

```
egrep "zo+zo+" words
```

```
zoozoo
```

More `egrep` commands

`.`	any symbol
`[a-z]`	anything in a range
`\<`	beginning of line
`$`	end of line



1 If a DFA is too hard, I do an ...

2 Time ... when studying COMP218

3 If a DFA likes it, it is ...

4 $\$10000000 \in \$10^{?}$

5 I study COMP218 hard because it will make me ...

How do you look for...

Words that start in `cat` and have another `cat`

```
egrep `\<cat.*cat` words
```

Words with at least ten vowels?

```
egrep `([aeiouy].*){10}` words
```

Words without any vowels?

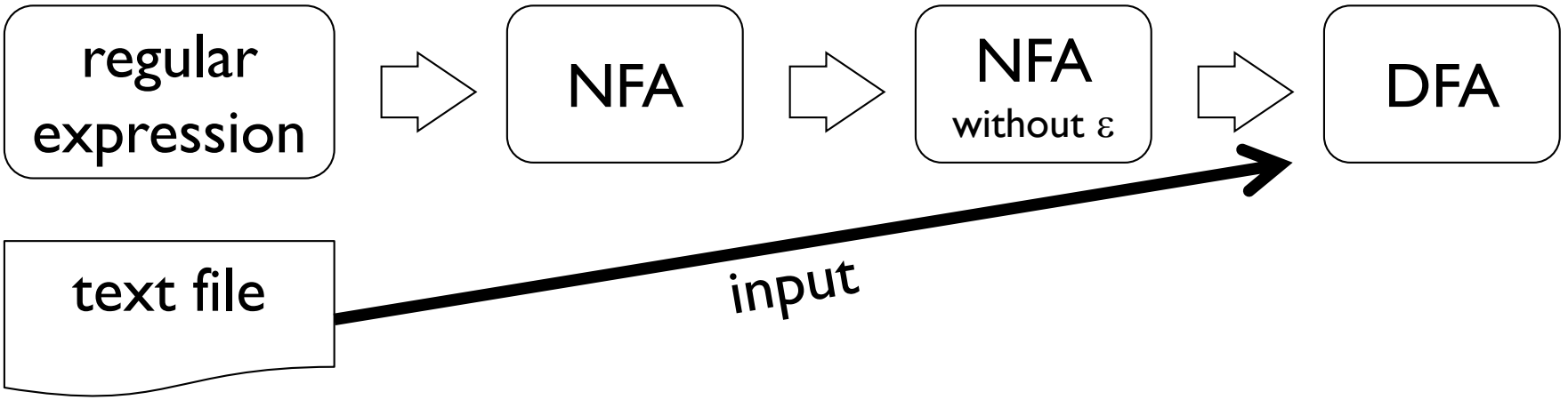
`[^R]`	does not contain R

```
egrep `\<[^AEIOUYaeiouy]*$` words
```

Words with exactly ten vowels?

```
egrep `\<[^AEIOUYaeiouy]*
       ([aeiouy][^AEIOUYaeiouy]*){10}$` words
```

How `egrep` (could) work



differences	in class	In `egrep`
`[ab]?, a+, (cat){3}`	not allowed	allowed
input handling	matches whole	looks for pattern
output	accept/reject	finds pattern

Implementation of egrep

- How do you handle expressions like:

`[ab]?` → `()|[ab]` — zero or one

$R? \rightarrow \varepsilon | R$

`(cat)+` → `(cat)(cat)*` — one or more

$R+ \rightarrow RR*$

`a{3}` → `aaa` — $\{n\}$ copies

$R\{n\} \rightarrow \underbrace{RR...R}_{n \text{ times}}$

`[^aeiouy]` ? — not containing any

Decision and closure properties

Properties of Language Classes

- A language class is a set of languages.
- We have one example: the regular languages.
- We'll see many more later.
- Language classes have two important kinds of properties:
  - Decision Properties
  - Closure Properties

Decision Properties

- A decision property for a class of languages is an algorithm that takes a formal description of a language (e.g., a DFA) and tells whether or not some property holds.
- Examples:
  - Is the language of a given DFA empty?
  - Are all words of its language finite?
  - Do all words end with a given letter?
  - etc.

Why Decision Properties?

- Example: we can use DFA to represent the behavior of a communication protocol. We can check whether such a protocol is correct by examining the language of the corresponding DFA:
  - "Can this protocol succeed?" = "Is this language nonempty?"
  - "Does it always end with an acknowledgment signal?" = "Do all words end with ACK?"

Closure Properties

- A *closure property* of a language class says that given languages in the class, an operator (e.g., union) produces another language in the same class.
- Example: the regular languages are obviously closed under union, concatenation, and (Kleene) closure.
  - Use the RE representation of languages.

Closure Under Union

- If L and M are regular languages, so is L ∪ M.
- Proof: Let L and M be the languages of regular expressions R and S, respectively.
- Then R+S is a regular expression whose language is L ∪ M.

Closure Under Concatenation and Kleene Closure

- Same idea:
  - RS is a regular expression whose language is LM.
  - R* is a regular expression whose language is L*.

Why Closure Properties?

- Gives us a better understanding of a given language class and its limitations
- Allow us to know that a language is in a given class even when we have no idea how to represent it, e.g., the language of words with an even number of 0s and five 1s that start with 01 and end with 010 is regular

Example

- The language L of strings that end in 101 is regular

$$(0+1)^*101$$

- How about the language $\overline{L}$ of strings that do not end in 101?

Example

- **Hint:** w does not end in 101 if and only if it ends in:

$$000, 001, 010, 011, 100, 110 \text{ or } 111$$

or it has length 0, 1, or 2

- So $\overline{L}$ can be described by the regular expression

$$(0+1)^*(000+001+010+010+100+110+111)$$
$$+ \varepsilon + (0 + 1) + (0 + 1)(0 + 1)$$

Complement

- The complement $\overline{L}$ of a language L contains those strings that are not in L

- Examples ($\Sigma = \{0, 1\}$)
  - L_1 = all strings that end in 101
  - $\overline{L}_1$ = all strings that do not end in 101
    = all strings end in $000, \ldots, 111$ or have length 0, 1, or 2

  - $L_2 = 1^* = \{\varepsilon, 1, 11, 111, \ldots\}$
  - $\overline{L}_2$ = all strings that contain at least one 0
    $= (0 + 1)^*0(0 + 1)^*$

Example

- The language L of strings that contain 101 is regular

$$(0+1)^*101(0+1)^*$$

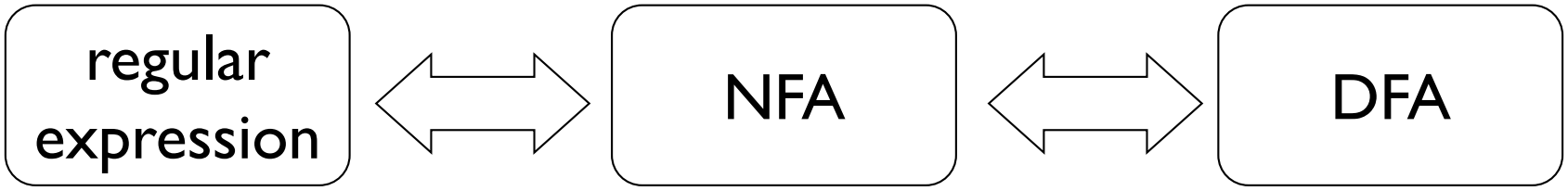
- How about the language $\overline{L}$ of strings that do not contain 101?

You can write a regular expression, but it is a lot of work!

Closure under complement

If L is a regular language, so is $\overline{L}$.

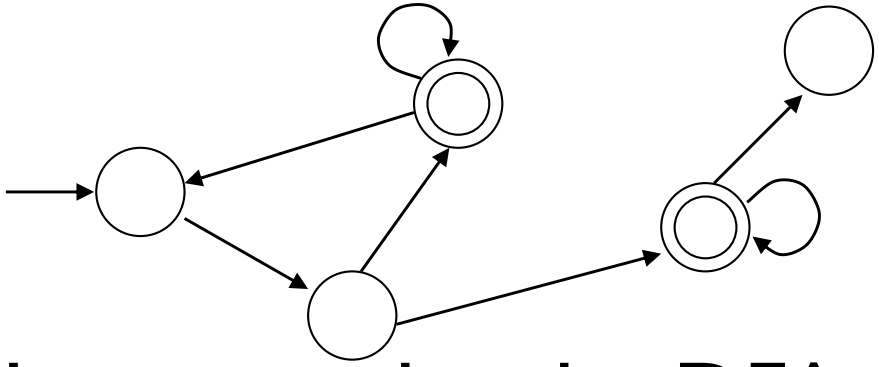
- To argue this, we can use any of the equivalent definitions for regular languages:



- The DFA definition will be most convenient
  - We assume L has a DFA, and show $\overline{L}$ also has a DFA

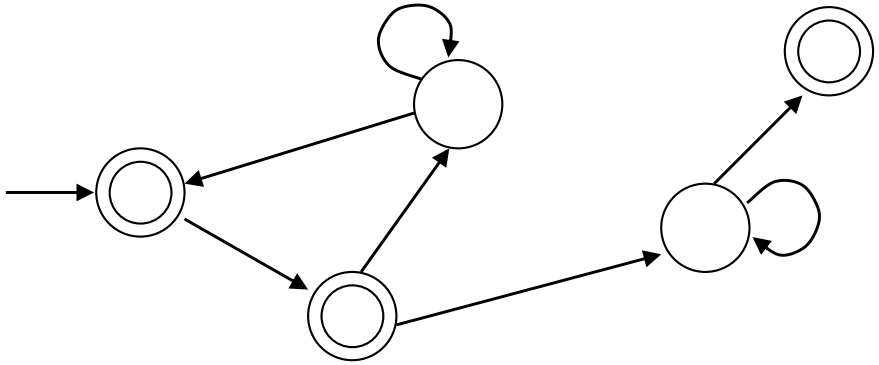
Arguing closure under complement

- Suppose L is regular, then it has a DFA M



accepts L

- Now consider the DFA M' with the accepting and rejecting states of M swapped

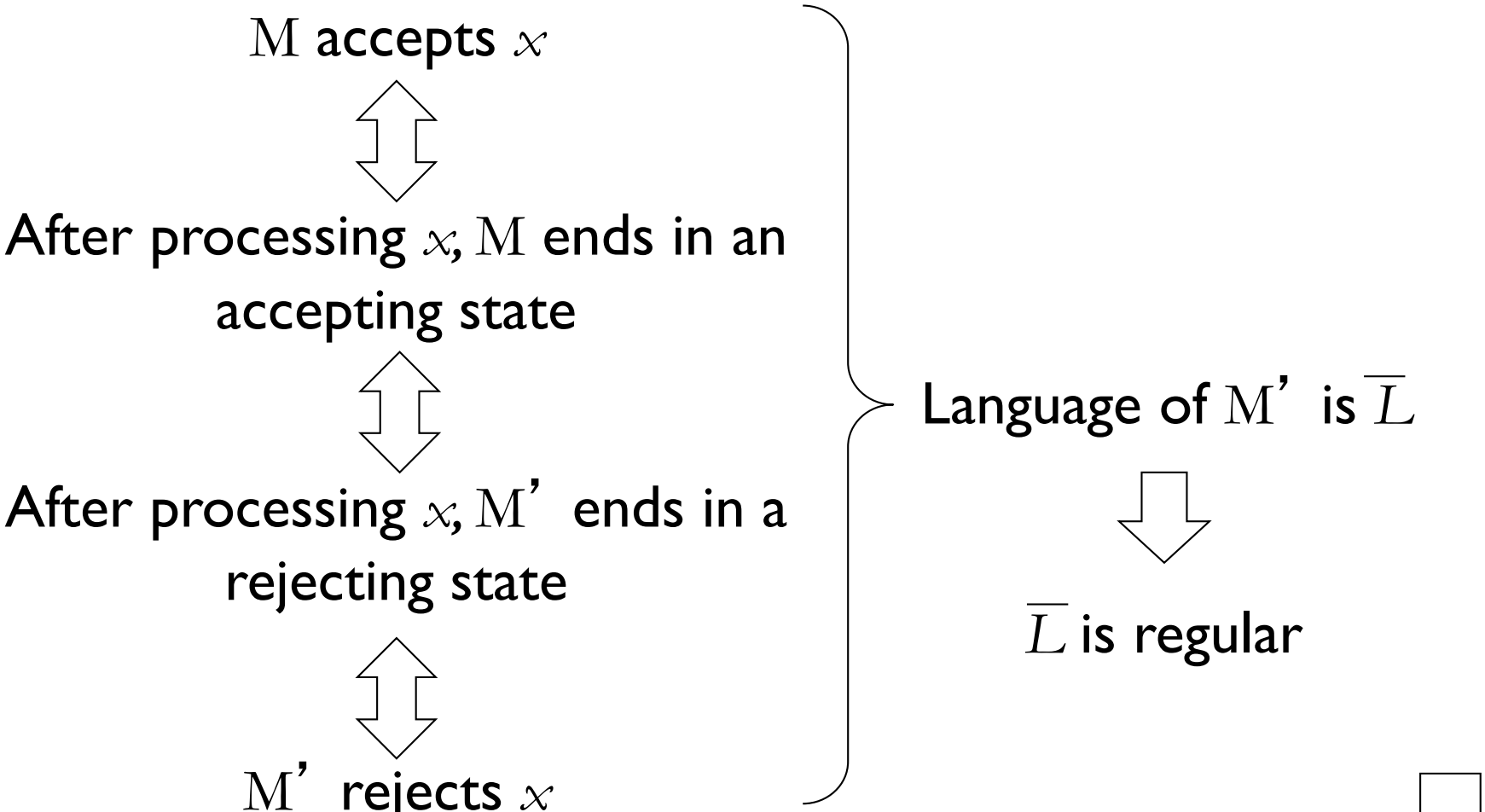


accepts strings not in L

this is exactly $\overline{L}$

Proof of closure under complement

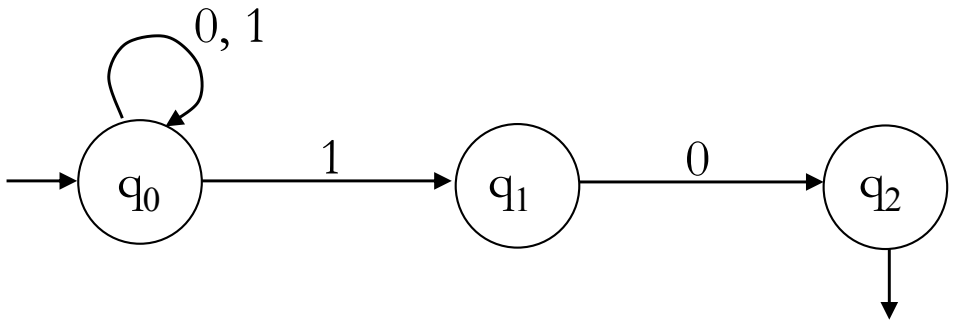
Now for every input $x \in \Sigma^*$:



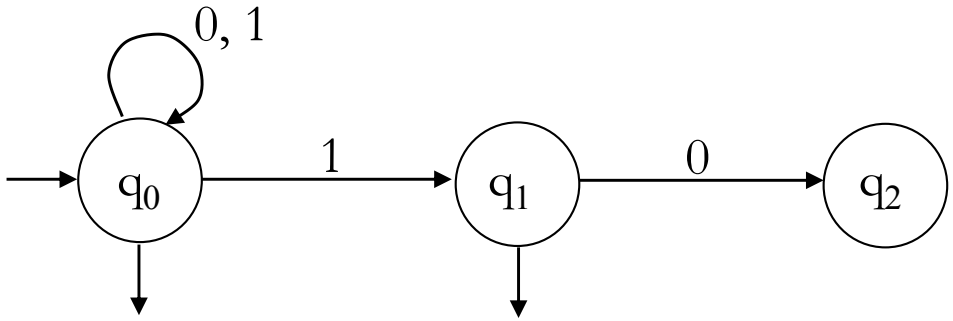
□

Food for thought

- Can we do the same thing with an NFA?

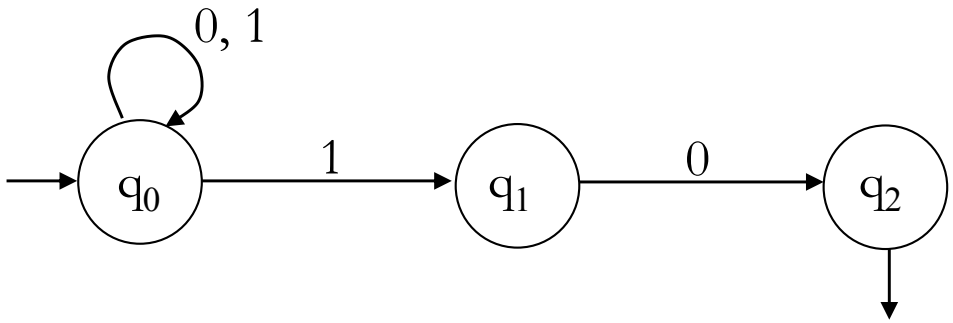


(0+1)*10

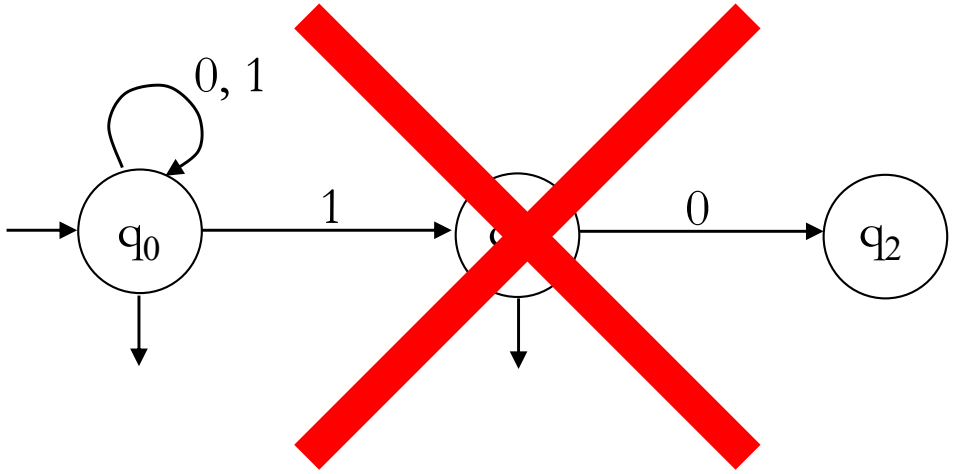


Food for thought

- Can we do the same thing with an NFA?



(0+1)*10



(0+1)* **NO!**

Intersection

- The intersection $L \cap L'$ is the set of strings that are in both L and L'

- Examples:

$L = (0 + 1)^*11$ $\quad$ $L' = 1^*$ $\quad$ $L \cap L' = 1^*11$

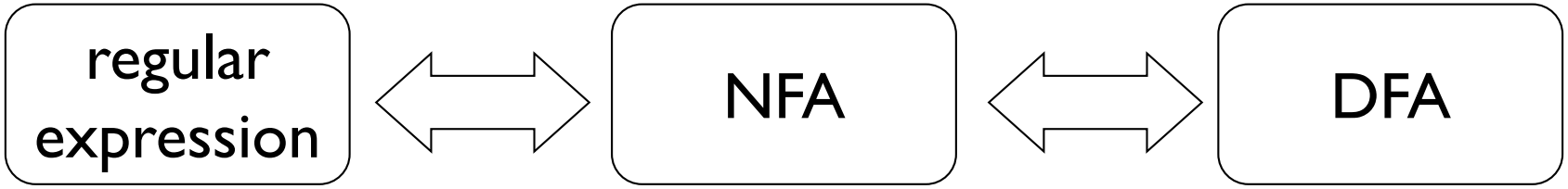
$L = (0 + 1)^*10$ $\quad$ $L' = 1^*$ $\quad$ $L \cap L' = \emptyset$

- If L, L' are regular, is $L \cap L'$ also regular?

Closure under intersection

If L and L' are regular languages, so is $L \cap L'$.

- To argue this, we can use any of the equivalent definitions for regular languages:

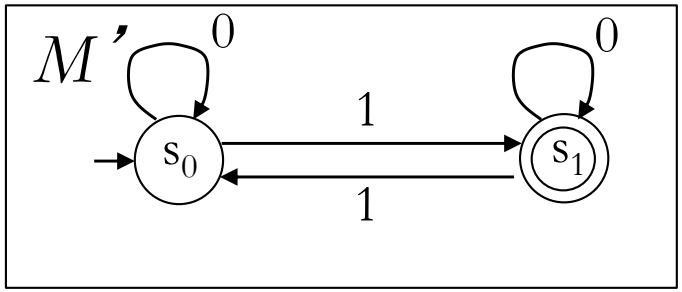


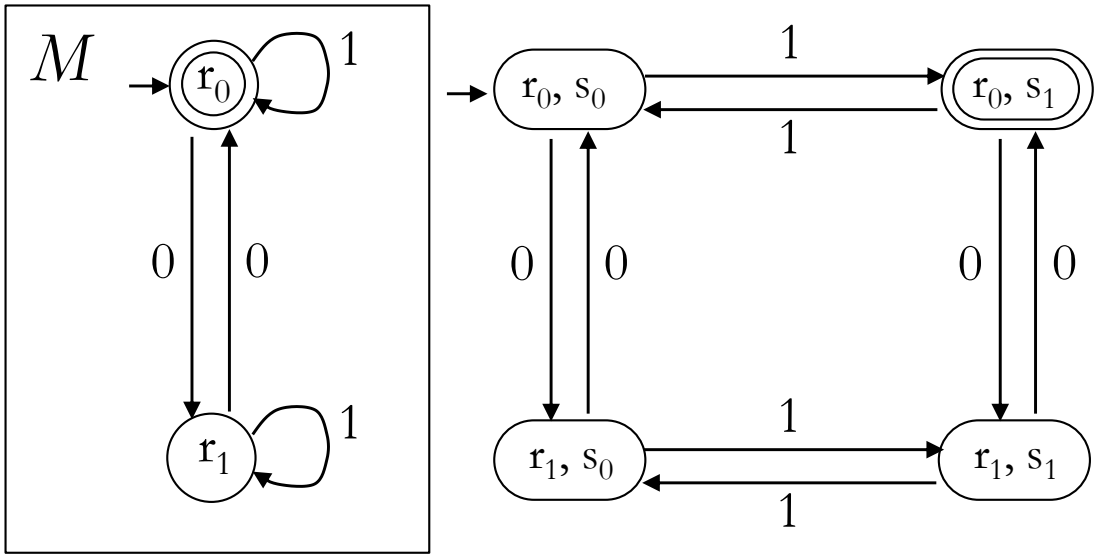
Suppose L and L' have DFAs, call them M and M'

Goal: Construct a DFA (or NFA) for $L \cap L'$

An example

L' = odd number of 1s





L = even number of 0s

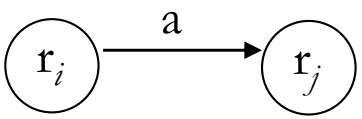
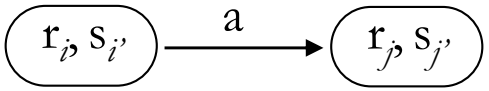
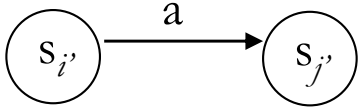
$L \cap L'$ = even number of 0s and odd number of 1s

Closure under intersection

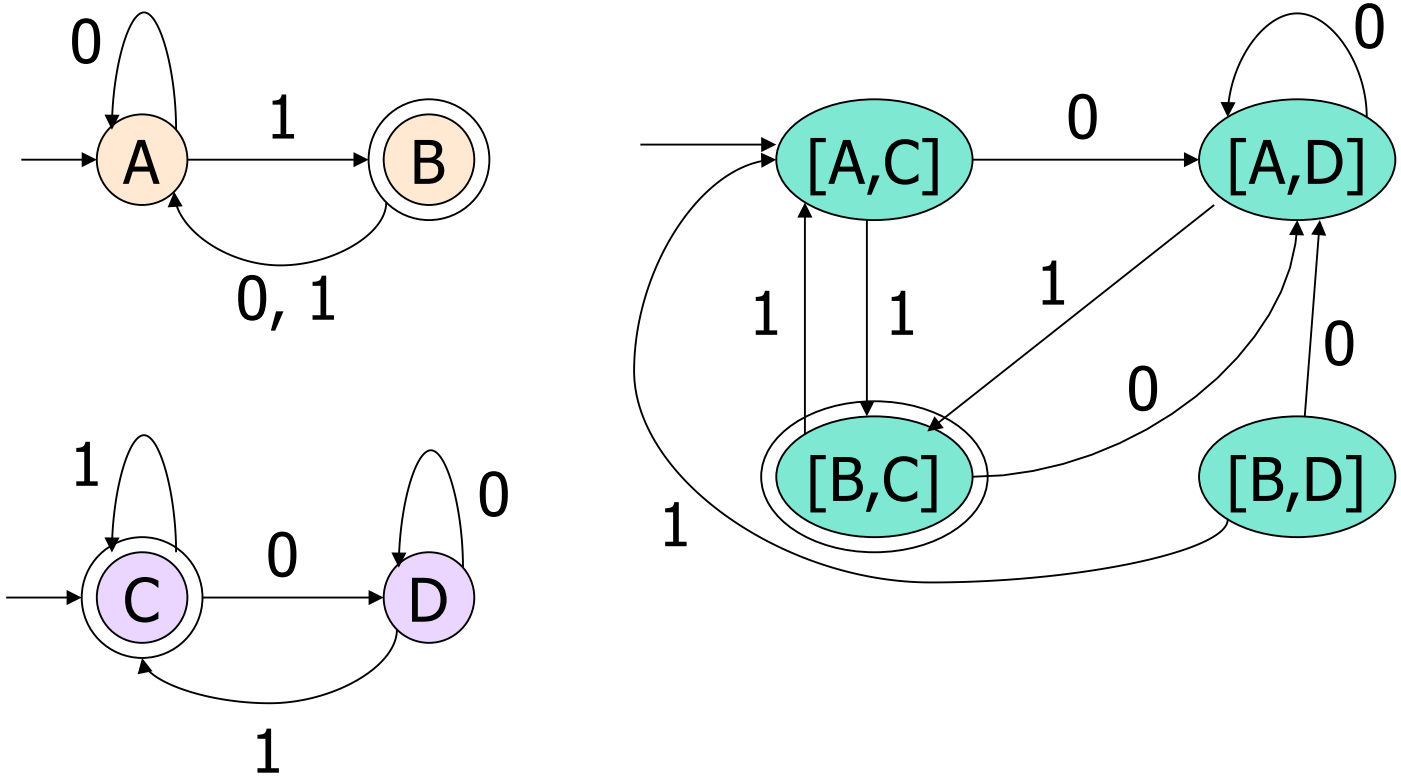
	M and M'	DFA for $L \cap L'$
states	$Q = \{r_1, ..., r_n\}$ $Q' = \{s_1, ..., s_{n'}\}$	$Q \times Q' = \{(r_1, s_1), (r_1, s_2), ..., (r_2, s_1), ..., (r_n, s_{n'})\}$
start state	r_i for M s_j for M'	(r_i, s_j)
accepting states	F for M F' for M'	$F \times F' = \{(r_i, s_j): r_i \in F, s_j \in F'\}$

Whenever M is in state r_i and M' is in state s_j, the DFA for $L \cap L'$ will be in state (r_i, s_j)

Closure under intersection

	M and M'	DFA for $L \cap L'$
transitions	 in M	
	 in M'	

Example: Product DFA for Intersection



Reversal

- The reversal w^R of a string w is w written backwards

$w = \texttt{dog}$ $\qquad\qquad$ $w^R = \texttt{god}$

- The reversal L^R of a language L is the language obtained by reversing all its strings

$L = \{\texttt{cat}, \texttt{dog}\}$ $\qquad\qquad$ $L^R = \{\texttt{tac}, \texttt{god}\}$

Reversal of regular languages

- $L =$ all strings that end in 101 is regular

$$(0+1)^*101$$

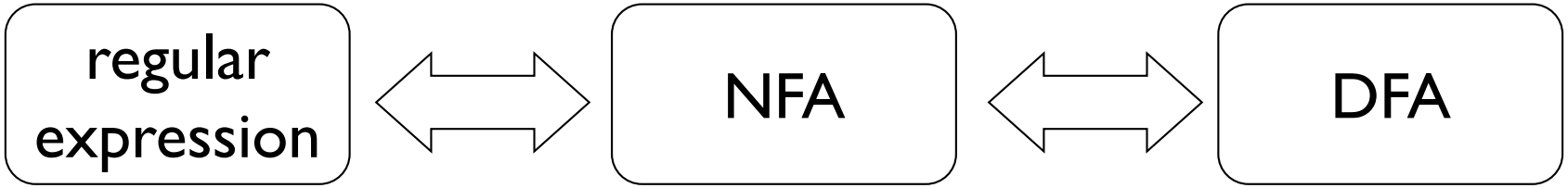
- How about L^R?
- This is the language of all strings beginning in 101
- Yes, because it is represented by

$$101(0+1)^*$$

Closure under reversal

If L is a regular language, so is L^R.

- How do we argue?



Arguing closure under reversal

- Take a regular expression E for L
- We will show how to reverse E
- A regular expression can be of the following types:
  - The special symbols $\emptyset$ and ε
  - Alphabet symbols like $\mathtt{a}, \mathtt{b}$
  - The union, concatenation, or star of simpler expressions

Proof of closure under reversal

regular expression E	⇨	reversal E^R
$\emptyset$		$\emptyset$
ε		ε
a (alphabet symbol)		a
$E_1 + E_2$		${E_1}^R + {E_2}^R$
$E_1 E_2$		${E_2}^R {E_1}^R$
${E_1}*$		$({E_1}^R)*$

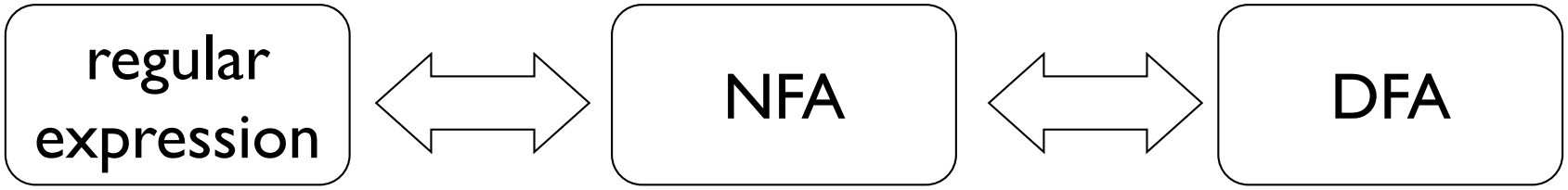
A question

$L^{DUP} = \{ww\text{: } w \in L\}$

$L = \{\texttt{cat}, \texttt{dog}\}$

$L^{DUP} = \{\texttt{catcat}, \texttt{dogdog}\}$

If L is regular, is L^{DUP} also regular?



A question

- Let's try with regular expression:

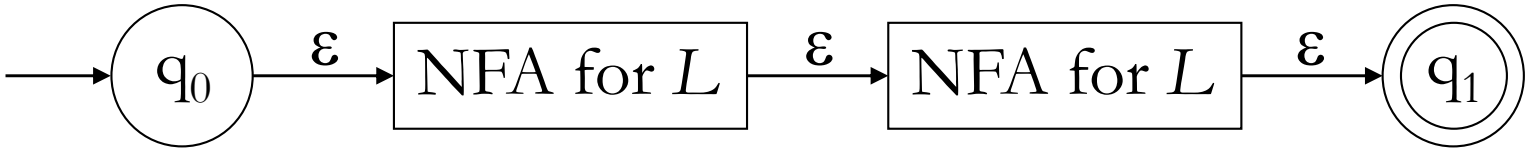
$$L^{DUP} \neq L^2$$

$$L = \{\mathtt{a}, \mathtt{b}\}$$

$$L^{DUP} = \{\mathtt{aa}, \mathtt{bb}\}$$

$$LL = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$$

- Let's try with NFA:



A question

- Let's try with regular expression:

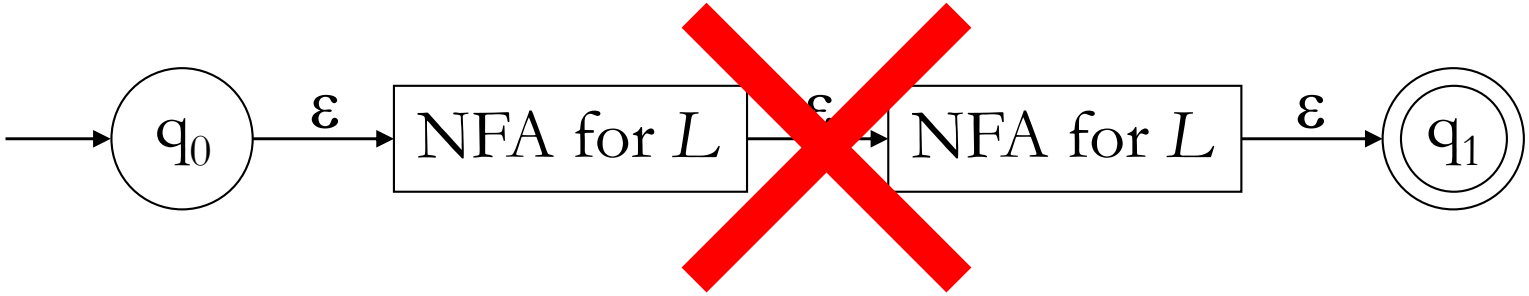
$$L^{DUP} \neq L^2$$

$$L = \{\mathtt{a}, \mathtt{b}\}$$

$$L^{DUP} = \{\mathtt{aa}, \mathtt{bb}\}$$

$$LL = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$$

- Let's try with NFA:



An example

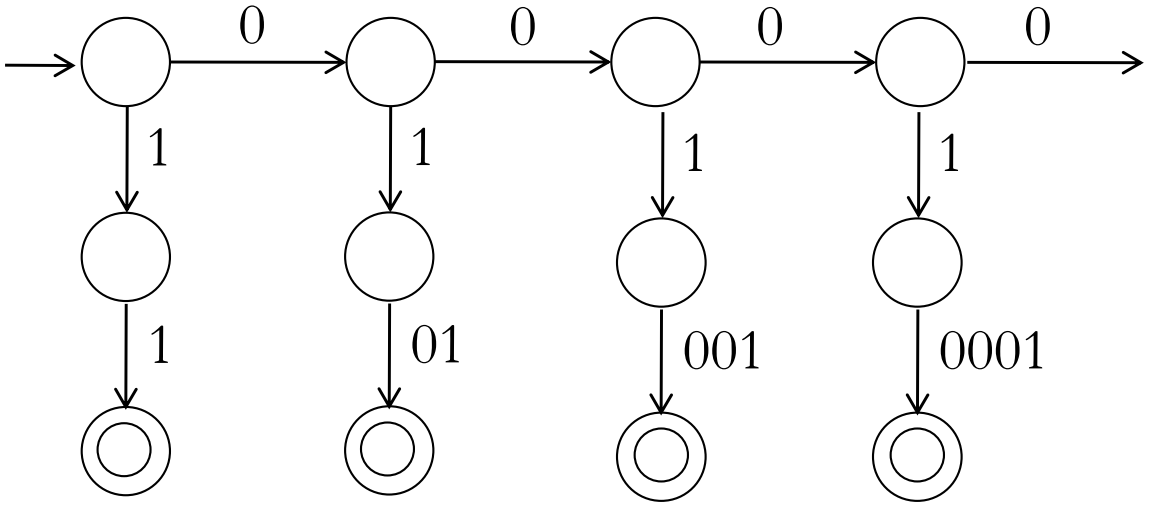
$L = 0^*1$ is regular

$$L = \{1, 01, 001, 0001, ...\}$$

$$L^{DUP} = \{11, 0101, 001001, 00010001, ...\}$$
$$= \{0^n 1 0^n 1 : n \geq 0\}$$

- Let's try to design an NFA for L^{DUP}

An example



$$L^{DUP} = \{11, 0101, 001001, 00010001, ...\}$$

$$= \{0^n 1 0^n 1 : n \geq 0\}$$

The Emptiness Problem

- Given a regular language, does the language contain any string at all.
- Assume representation is DFA.
- Construct the transition graph.
- Compute the set of states reachable from the start state.
- If any final state is reachable, then yes, else no.

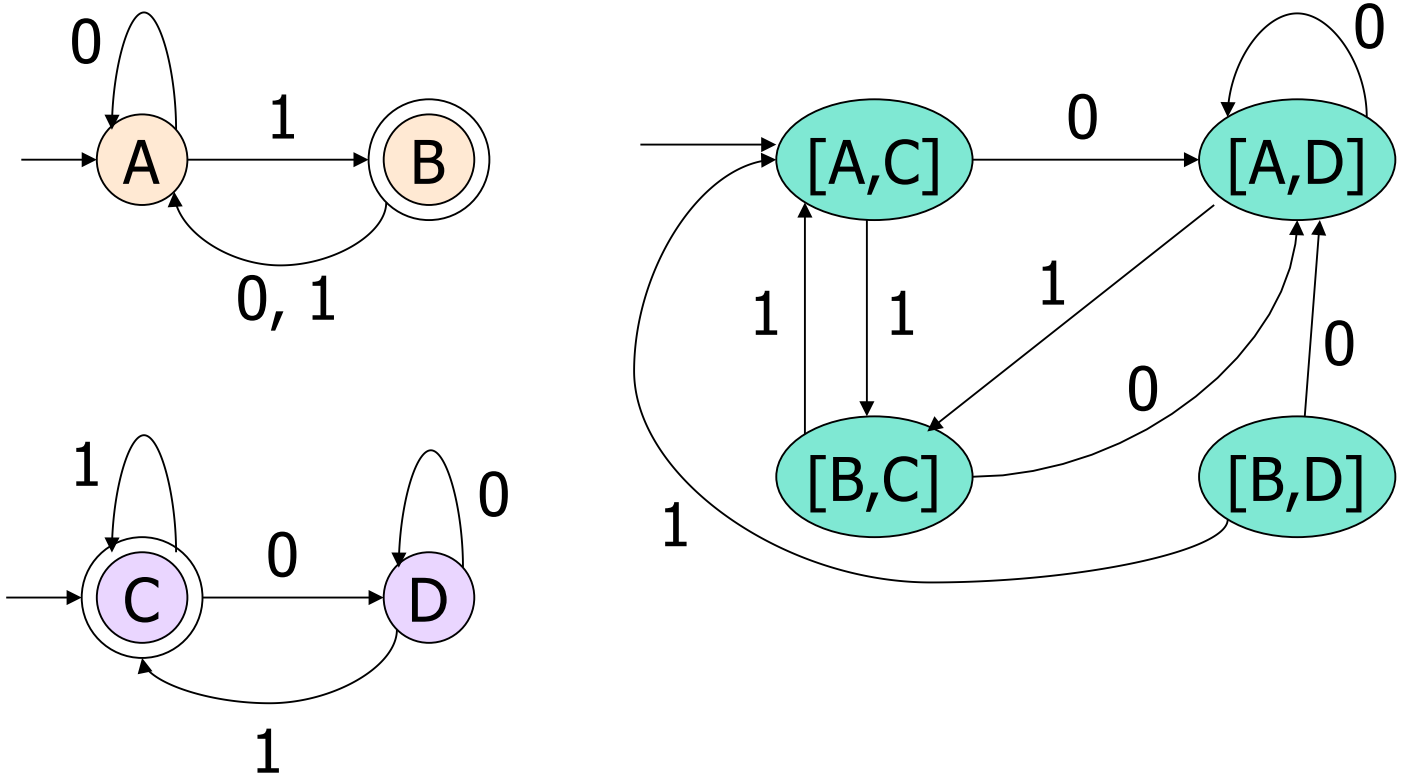
Decision Property: Equivalence

- Given regular languages L and M, is L = M?
- Algorithm involves constructing the *product DFA* from DFA's for L and M.
- Let these DFA's have sets of states Q and R, respectively.
- Product DFA has set of states Q × R.
  - I.e., pairs [q, r] with q in Q, r in R.

Product DFA – Continued

- Start state = $[q_0, r_0]$ (the start states of the DFA's for L, M).
- Transitions: $\delta([q,r], a) = [\delta_L(q,a), \delta_M(r,a)]$
  - δ_L, δ_M are the transition functions for the DFA's of L, M.
  - That is, we simulate the two DFA's in the two state components of the product DFA.

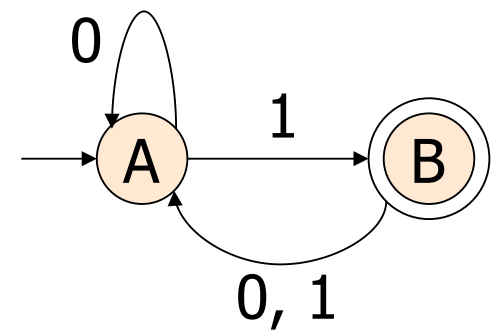
Example: Product DFA

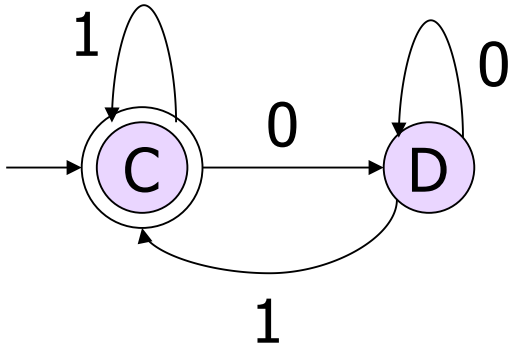


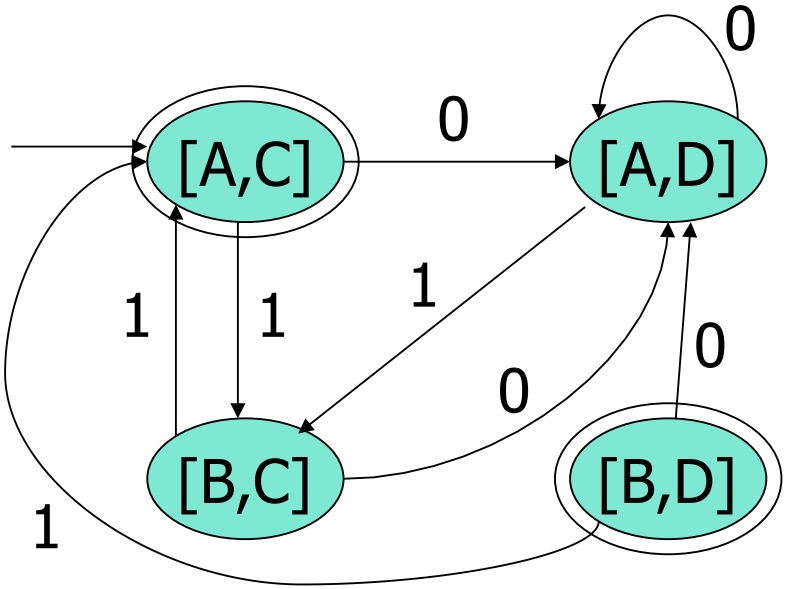
Equivalence Algorithm

- Make the final states of the product DFA be those states [q, r] such that exactly one of q and r is a final state of its own DFA.
- Thus, the product accepts w iff w is in exactly one of L and M.

Example: Equivalence







Equivalence Algorithm – (2)

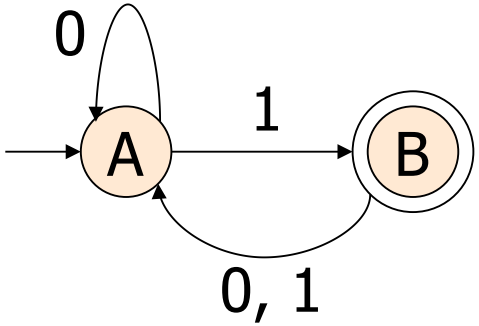
- The product DFA's language is empty iff L = M.
- But we already have an algorithm to test whether the language of a DFA is empty.

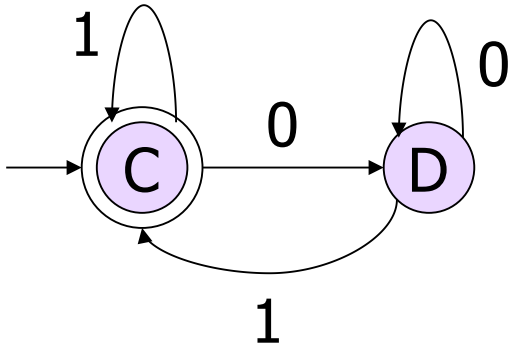
Decision Property: Containment

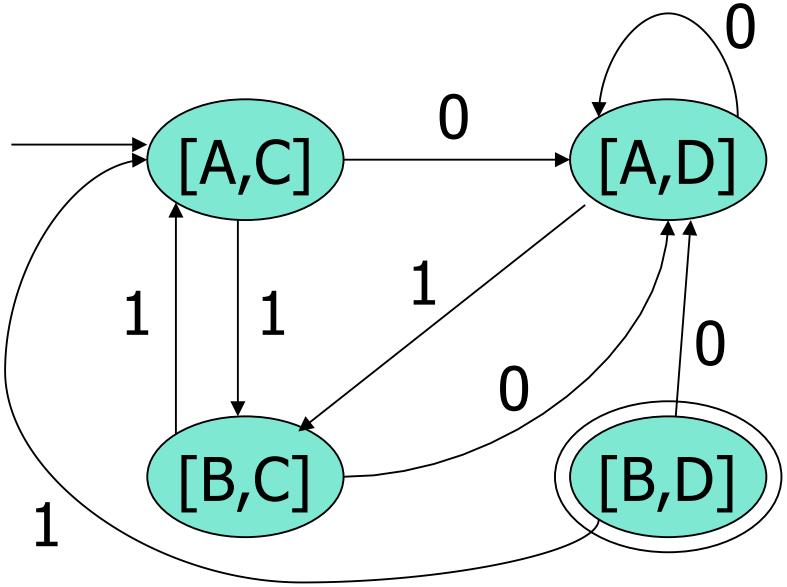
- Given regular languages L and M, is L $\subseteq$ M?
- Algorithm also uses the product automaton.
- How do you define the final states [q, r] of the product so its language is empty iff L $\subseteq$ M?

Answer: q is final; r is not.

Example: Containment







Note: the only final state is unreachable, so containment holds.

A famous open problem for regular languages

A generalised regular expression is a regular expression that allows for the complement operation, i.e. $\overline{R}$ for any regular expression R.

Can all regular languages be expressed using a generalised regular expressions without nesting of the Kleene star (R^*) operation?

This problem is open for more than 50+ years (1960s)!

Non-regular languages

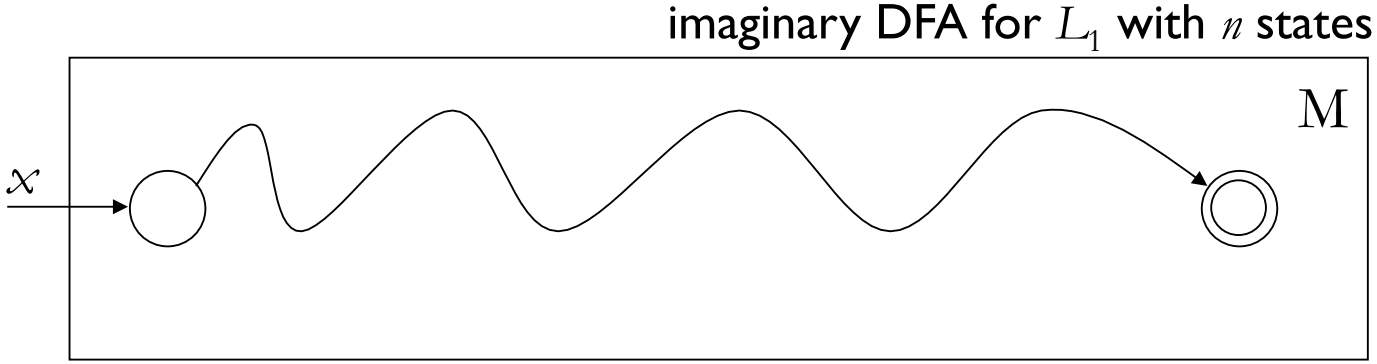
A non-regular language

- An example

  $L_0 = \{0^n 1^n \mid n \geq 0\}$ is not regular.

- We reason by contradiction:
  - Suppose we have managed to construct a DFA M for L_1 with n states
  - We argue something must be wrong with this DFA
  - In particular, M must accept some strings outside L_1

A non-regular language

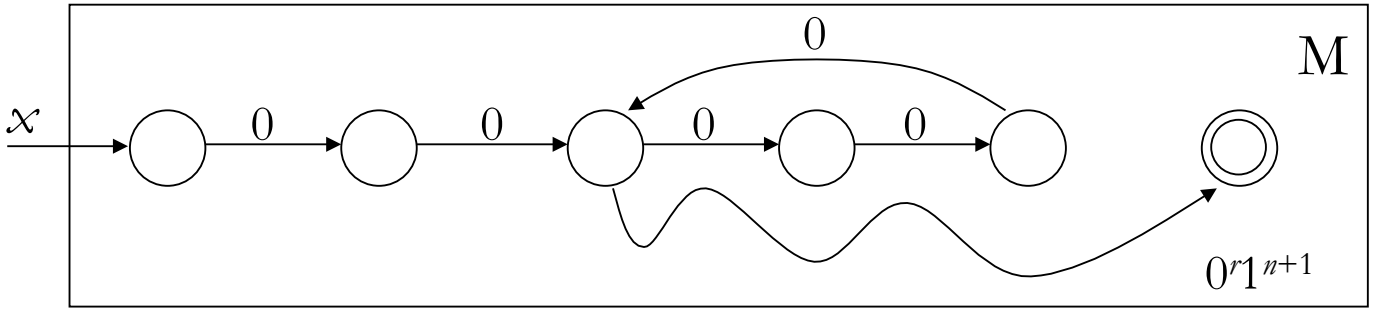


- What happens when we run M on input

$$x = 0^{n+1}1^{n+1}?$$

  - M better accept, because $x \in L_0$

A non-regular language



- What happens when we run M on input

$$x = 0^{n+1}1^{n+1}?$$

  - M better accept, because $x \in L_0$
  - But since M has n states, it must revisit at least one of its states while reading 0^{n+1}

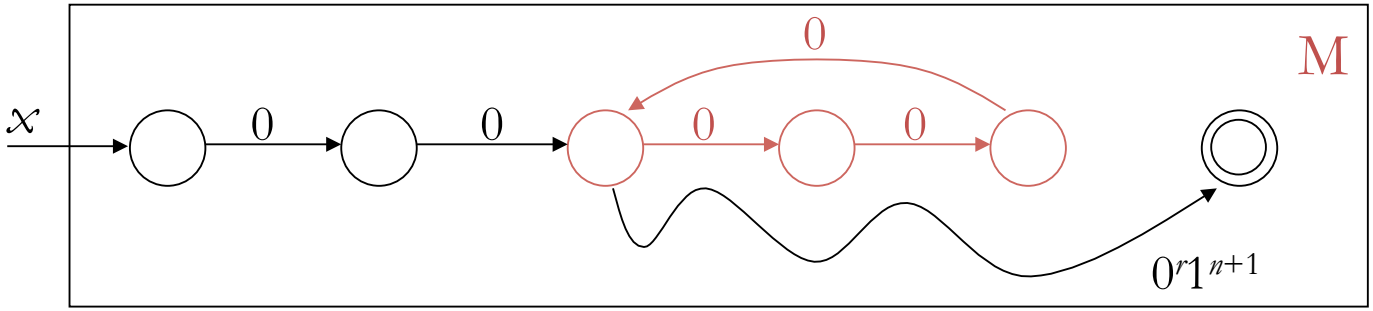
Pigeonhole principle

Suppose you are tossing $n + 1$ balls into n bins.
Then two balls end up in the same bin.

- Here, balls are letters, bins are states:

If you have a DFA with n states and it reads $n + 1$ consecutive letters, then it must end up in the same state twice.

A non-regular language

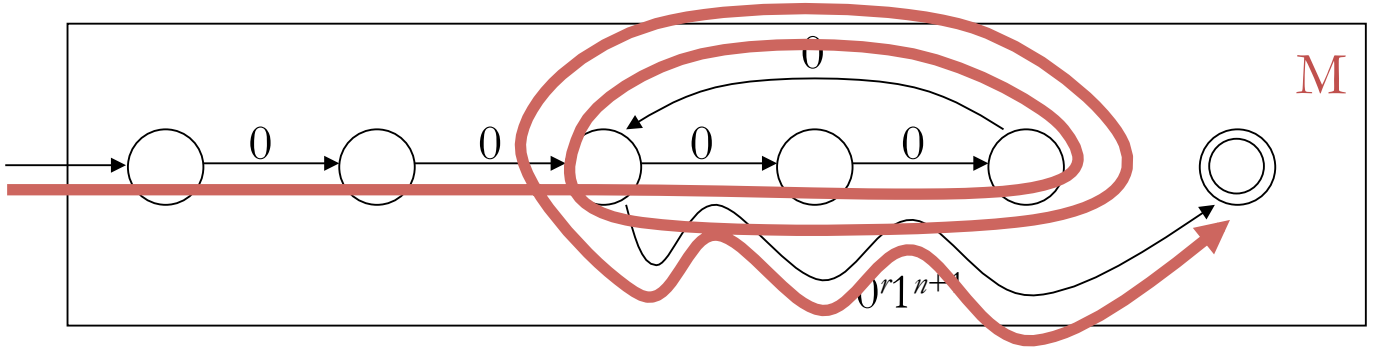


- What happens when we run M on input

$$x = 0^{n+1}1^{n+1}?$$

  - M better accept, because $x \in L_0$
  - But since M has n states, it must revisit at least one of its states while reading 0^{n+1}
  - But then the DFA must contain a loop with 0s

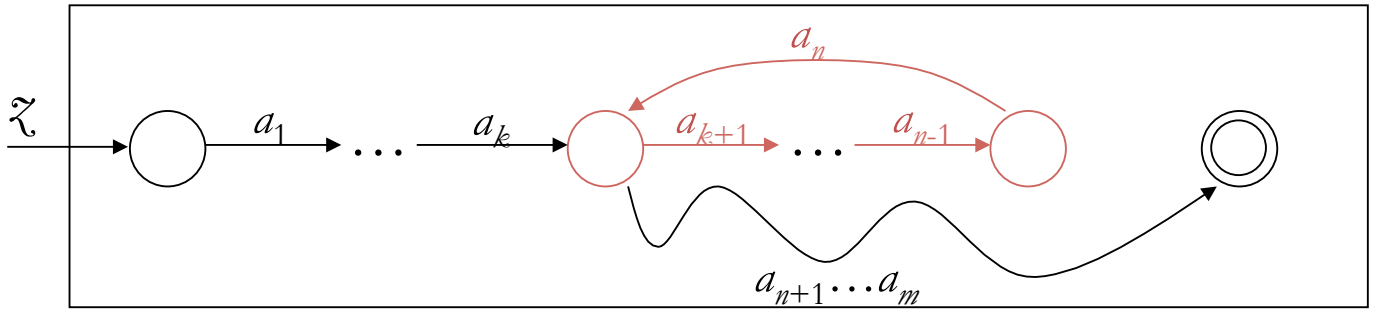
A non-regular language



- The DFA will then also accept strings that go around the loop multiple times
- But such strings have more 0s than 1s, so they are not in L_0!

General method for showing non-regularity

- Every regular language L has a property:



- For every sufficiently long input z in L, there is a "middle part" in z that, even if repeated any number of times, keeps the input inside L

Pumping lemma for regular languages

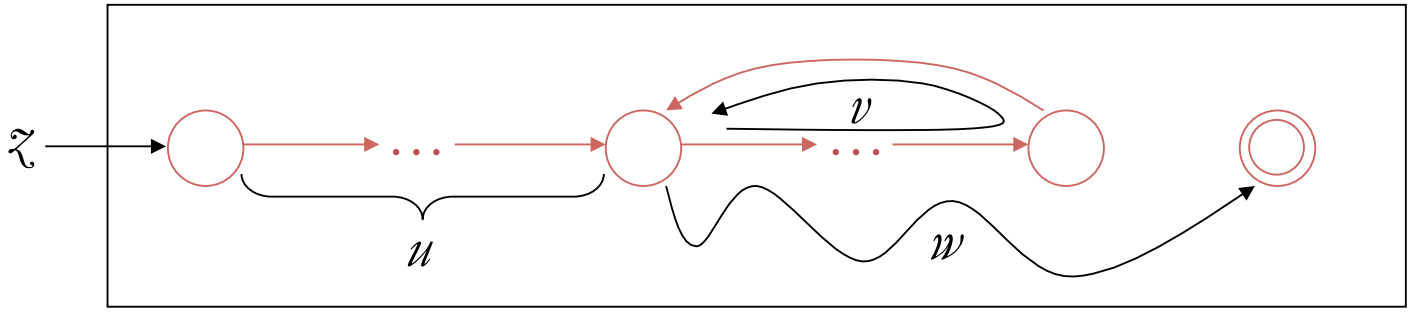
- Pumping lemma: For every regular language L

There exists a number n such that for all strings z in L longer than n, we can write $z = u\,v\,w$ where

① $|uv| \leq n$

② $|v| \geq 1$

③ For all $i \geq 0$, the string $u\,v^i\,w$ is in L.



Arguing non-regularity

- If L is regular, then:

> There exists n such that for all z in L longer than n, we can write $z = u\ v\ w$ where ① $|uv| \leq n$, ② $|v| \geq 1$ and ③ For all $i \geq 0$, the string $u\ v^i\ w$ is in L.

- So to prove L is not regular, it is enough to show:

> For all n there exists z in L longer than n, such that for all ways of writing $z = u\ v\ w$ where ① $|uv| \leq n$ and ② $|v| \geq 1$, the string $u\ v^i\ w$ is not in L for some $i \geq 0$.

Proving non-regularity

For all n there exists z in L longer than n, such that for all ways of writing $z = u\,v\,w$ where ① $|uv| \leq n$ and ② $|v| \geq 1$, there exists some $i \geq 0$ such that the string $u\,v^i\,w$ is not in L.

- This is a (Ehrenfeucht–Fraïssé) game between two players, say Adam *(for all)* and Eve *(exists)*

	Adam	Eve
1	chooses n	chooses $z \in L$ $(\|z\| > n)$
2	writes $z = uvw$ $(\|uv\| \leq n, \|v\| \geq 1)$	chooses i
		Eve wins if $uv^iw \notin L$
		(otherwise Adam wins)

Arguing non-regularity

- Eve needs to give a strategy that, regardless of what Adam does, she always wins the game

	Adam	Eve
1	chooses n	chooses $z \in L \ (\|z\| > n)$
2	writes $z = uvw \ (\|uv\| \leq n, \|v\| \geq 1)$	chooses i
		Eve wins if $uv^i w \notin L$

Example

	Adam	Eve
1	chooses n	chooses $z \in L$ $(\|z\| > n)$
2	writes $z = uvw$ $(\|uv\| \leq n, \|v\| \geq 1)$	chooses i
		Eve wins if $uv^i w \notin L$

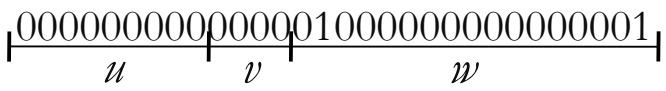
$$L_0 = \{0^n 1^n \mid n \geq 0\}$$

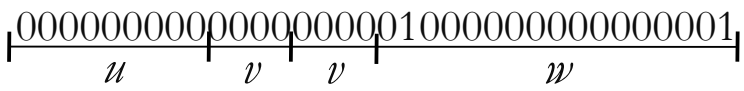
	Adam	Eve
1	chooses n	$z = 0^n 1^n \in L_0$
2	writes $z = uvw$	$i = 2$
	$\underbrace{000000000}_{u}\underbrace{000}_{v}\underbrace{0011111111111111}_{w}$	$uv^2w = 0^{n+k}1^n \notin L_0$
	$\underbrace{000000000}_{u}\underbrace{000}_{v}\underbrace{000}_{v}\underbrace{0011111111111111}_{w}$	*(where $k = \|v\|$)*

Example

$L^{DUP} = \{ww \mid w \in \Sigma^*\}$ $\quad\quad$ $\Sigma = \{0, 1\}$

	Adam	Eve
1	chooses n	$z = 0^n 1 0^n 1$
2	writes $z = uvw$	$i = 2$
		$uv^2w = 0^{n+k}10^n1 \notin L^{DUP}$
		(where $k = \lvert v \rvert$)





Which of these are regular?

$L_0 = \{x \mid x = 0^n 1^n, n \geq 0\}$ ✘ $\Sigma = \{0, 1\}$

$L^{DUP} = \{xx \mid x \in \Sigma^* \}$ ✘

$L_1 = \{x \mid x$ has the same number of 0s and 1s$\}$

$L_2 = \{x \mid x = 0^n 1^m, n > m \geq 0\}$

$L_3 = \{x \mid x$ has the same number of patterns 01 and 11$\}$

$L_4 = \{x \mid x$ has the same number of patterns 01 and 10$\}$

$L_5 = \{x \mid x$ has a different number of 0s and 1s$\}$

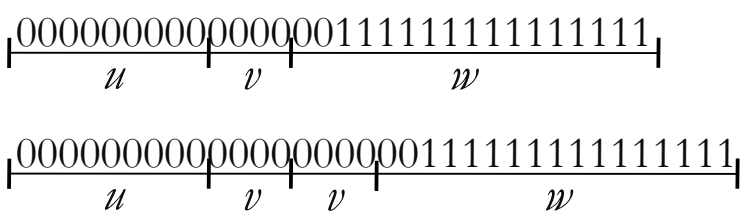
$L_6 = \{1^p \mid p$ is prime$\}$

$L_7 = \{x \mid x$ is a palindrome, i.e., $x = x^R\}$

Example

$L_1 = \{x \mid x \text{ has the same number of 0s and 1s}\}$

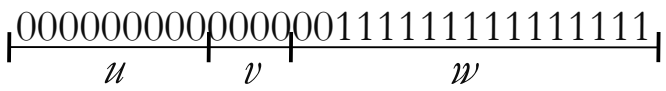
	Adam	Eve
1	chooses n	$z = 0^n1^n$
2	writes $z = uvw$	$i = 2$
		$uv^2w = 0^{n+k}1^n \notin L_1$
		(where $k = \lvert v \rvert$)

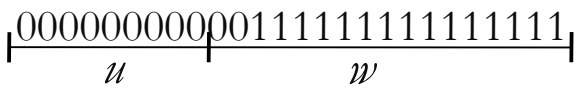


Example

$$L_2 = \{x \mid x = 0^m 1^n, m > n \geq 0\}$$

	Adam	Eve
1	chooses n	$z = 0^{n+1}1^n$
2	write $z = uvw$	$i = 0$
		$uv^0w = 0^{n+1-k}1^n \notin L_2$
		(where $k = \lvert v \rvert$)





Example

$$L_3 = \{x \mid x \text{ has the same number of } 01\text{s and } 11\text{s}\}$$

	Adam	Eve
1	chooses n	$z = (01)^n(11)^n$
	$n = 1$	$z = 0111 \notin L_3$

has too many 11s

What Eve had in mind:

$n = 1$	$z = 011$
$n = 2$	$z = 010111$
$n = 3$	$z = 010101111$

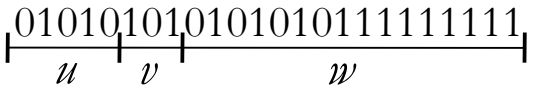
$$z = (01)^n 1^n$$

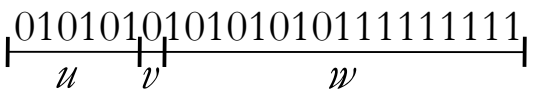
has n 01s and n 11s

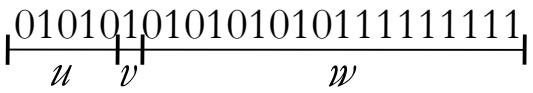
Example

$L_3 = \{x \mid x \text{ has the same number of } 01\text{s and } 11\text{s}\}$

	Adam	Eve wins!
1	chooses n	$z = (01)^n 1^n$
2	writes $z = uvw$	$i = 0$



or 

or 

Taking out v will remove
at least one 01,
but it does not remove any 11s

so $uv^0w \notin L_3$

Example

$L_4 = \{x \mid x \text{ has the same number of 01s and 10s}\}$

	Adam	Eve
1	chooses n	$z = (01)^n(10)^n$

$n = 1$ $z = 0110$

$n = 2$ $z = 01011010$

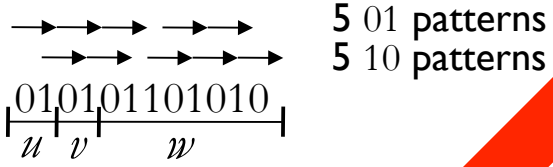
$n = 3$ $z = 010101101010$

Example

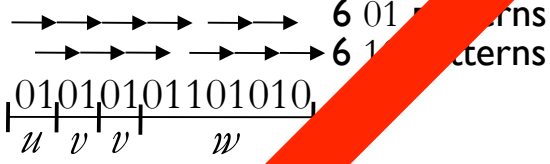
$L_4 = \{x \mid x \text{ has the same number of 01s and 10s}\}$

is regular!

	Adam	Eve
1	chooses n	$(01)^n(10)^n$
2	writes $z = uvw$	$i = 0$



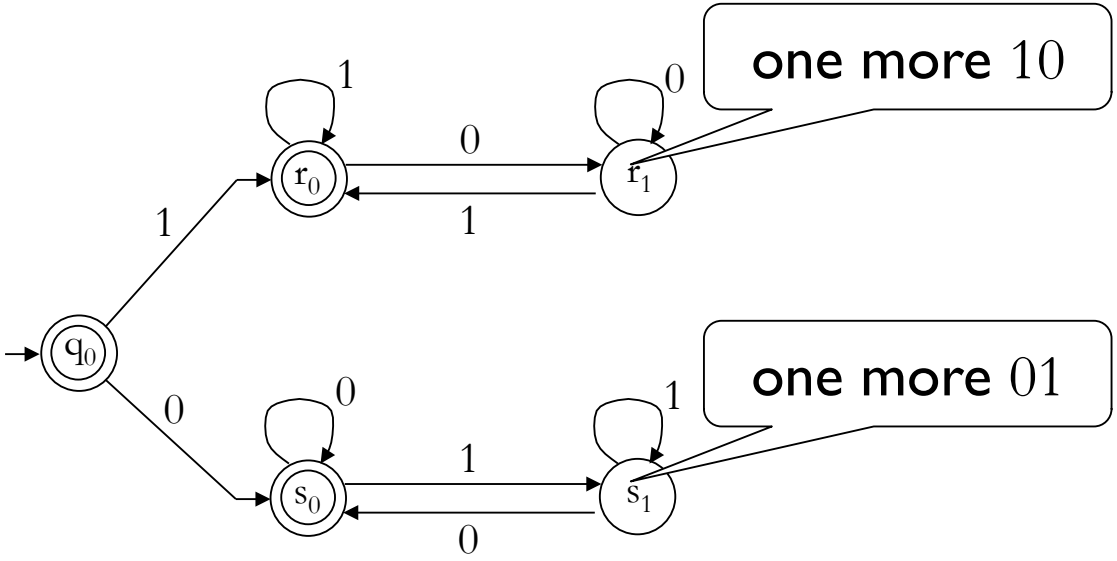
5 01 patterns
5 10 patterns



6 01 [illegible]rns
6 1[illegible]tterns

4 01 patterns
4 10 patterns

Example



$L_4 = \{x \mid x \text{ has the same number of 01s and 10s}\}$

Example

$L_5 = \{x \mid x \text{ has a different number of 0s than 1s}\}$

	Adam	Eve
1	chooses n	$z = ?$

there is an easier way!

$L_1 = \{x\text{: } x \text{ has the same number of 0s and 1s}\} = \overline{L_5}$

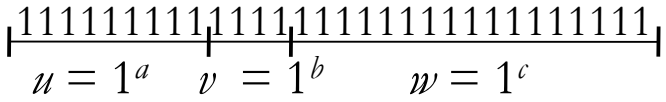
If L_5 is regular, then $L_1 = \overline{L_5}$ is also regular

But L_1 is not regular, so L_5 cannot be regular

Example

$L_6 = \{1^p \mid p \text{ is prime}\}$

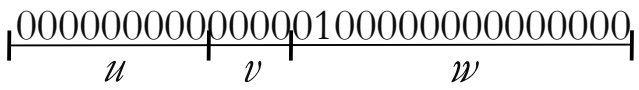
	Adam	Eve
1	chooses n	$z = 1^p$ where $p > n$ is prime
2	writes $z = uvw = 1^a1^b1^c$	$i = ?\ a + c$
	(where $n = a+b+c$)	$uv^iw = 1^a1^{ib}1^c$
		$= 1^{(a+c)+ib}$
		$= 1^{(a+c)+(a+c)b}$
		$= 1^{(a+c)(b+1)}$
		$= 1^{\text{composite}} \notin L_6$

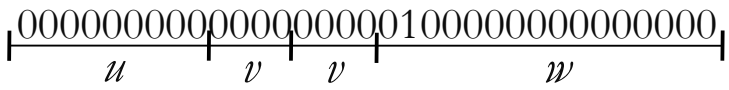


Example

$L_7 = \{x \mid x \text{ is a palindrome, i.e., } x = x^R\}$

	Adam	Eve
1	chooses n	$z = 0^n 1 0^n$
2	writes $z = uvw$	$i = 2$
		$uv^2w = 0^{n+k}10^n \notin L_7$
		(where $k = \lvert v \rvert$)





Which of these are regular?

$\Sigma = \{0, 1\}$

$L_0 = \{x \mid x = 0^n 1^n, n \geq 0\}$ ✘

$L^{DUP} = \{xx \mid x \in \Sigma^* \}$ ✘

$L_1 = \{x \mid x$ has the same number of 0s and 1s$\}$ ✘

$L_2 = \{x \mid x = 0^n 1^m, n > m \geq 0\}$ ✘

$L_3 = \{x \mid x$ has the same number of patterns 01 and 11$\}$ ✘

$L_4 = \{x \mid x$ has the same number of patterns 01 and 10$\}$ ✔

$L_5 = \{x \mid x$ has a different number of 0s and 1s$\}$ ✘

$L_6 = \{1^p \mid p$ is prime$\}$ ✘

$L_7 = \{x \mid x$ is a palindrome, i.e., $x = x^R\}$ ✘

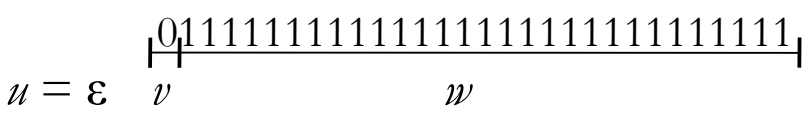
Important points

- Pumping lemma is a necessary but not sufficient condition for a language to be regular!
- Even if there does not exist a winning strategy for *Eve* in the game played on *L*, the language *L* can still be non-regular.
- The only way to prove that a language is regular is to construct either a finite state machine or a regular expression for this language!

A very special example

$$L_8 = \{01^k \mid k = m^2 \text{ for some } m>0\} \cup 000^*1^* \cup 1^*$$

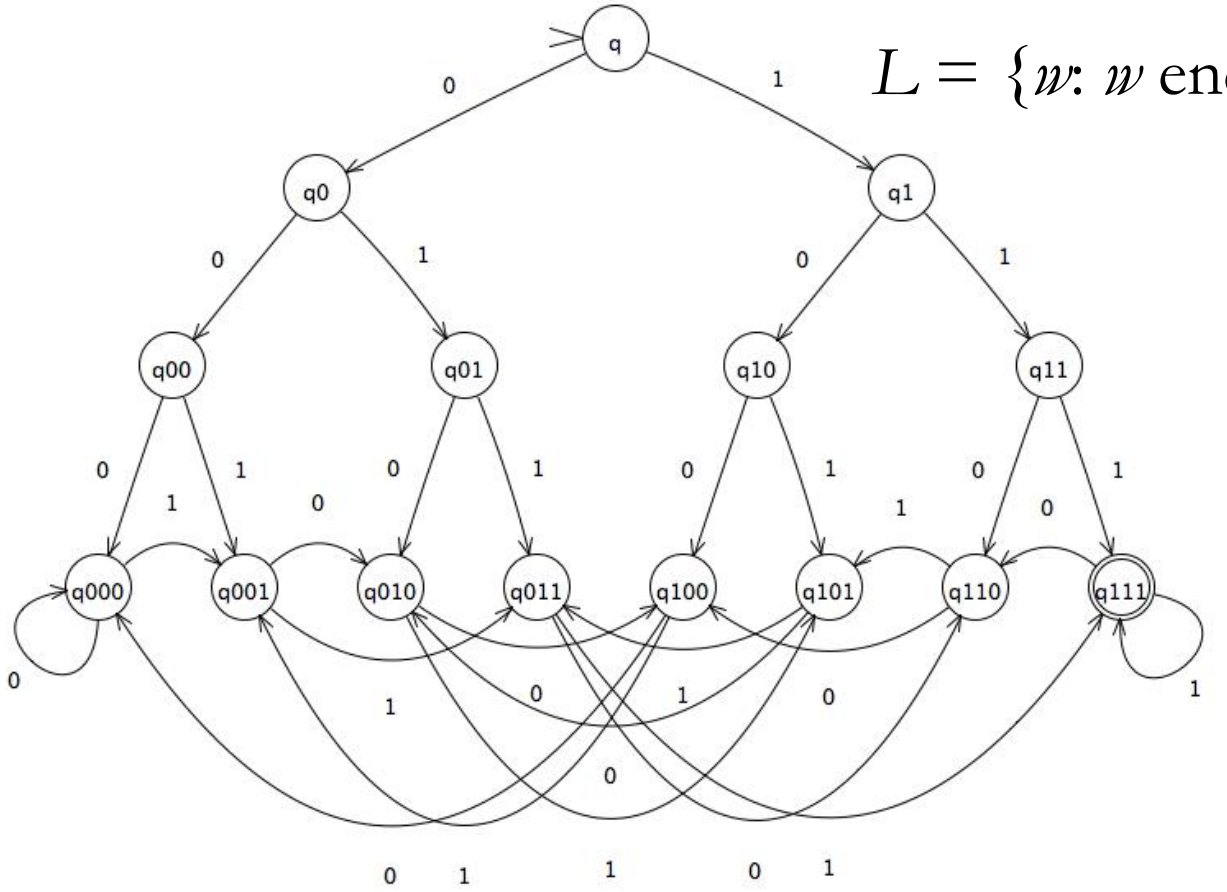
	Adam	Eve
1	chooses n	$z = 01^l$ where $l = n^2$
2	writes $z = uvw$	$i = ?$
		uv^iw



To prove non-regularity consider L_8^R !

DFA minimisation

Example

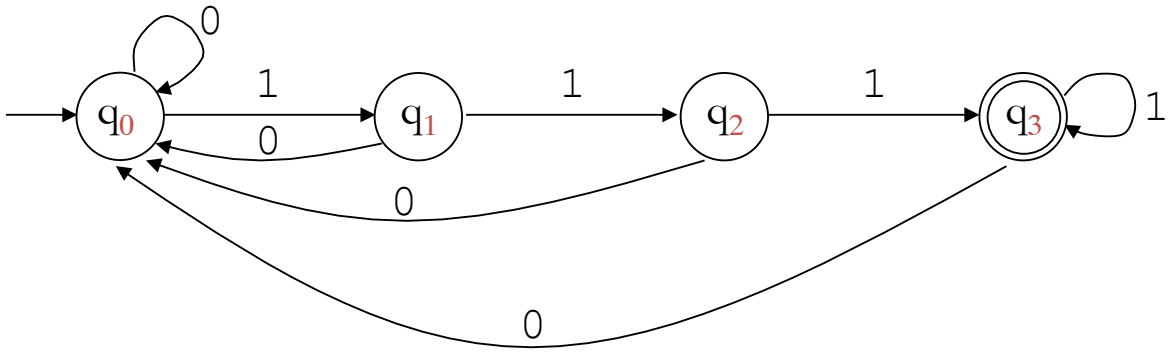


$L = \{w: w \text{ ends in } 111\}$

Isn't there a smaller one?

Smaller DFA

$L = \{w: w \text{ ends in } 111\}$

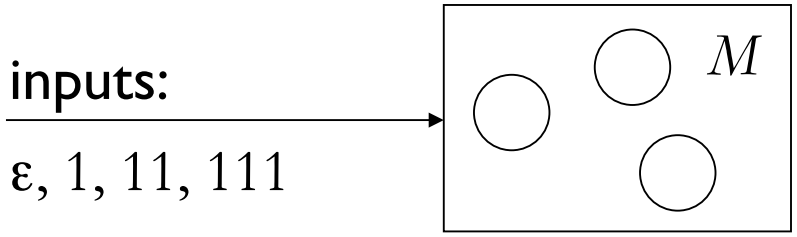


Can we do it with 3 states?

Even smaller DFA?

$$L = \{w: w \text{ ends in } 111\}$$

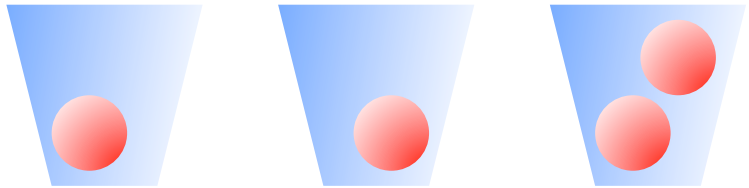
- Suppose we had a 3 state DFA for L



- There are 4 inputs but 3 states
  On two of these inputs M ends in the same state

Pigeonhole principle

Suppose you are tossing 4 balls into 3 bins.
Then two balls end up in the same bin.

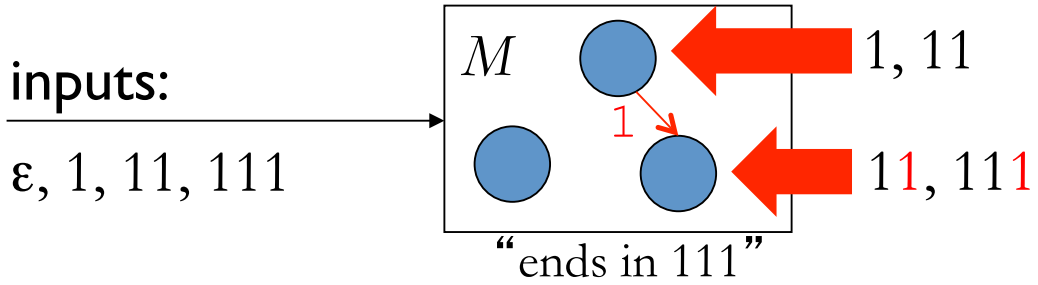


Take 4 inputs and feed them into a 3 state DFA.
Then two inputs end up in the same state.

A smaller DFA

$$L = \{w: w \text{ ends in } 111\}$$

- Suppose inputs $x = 1, y = 11$ lead to the same state

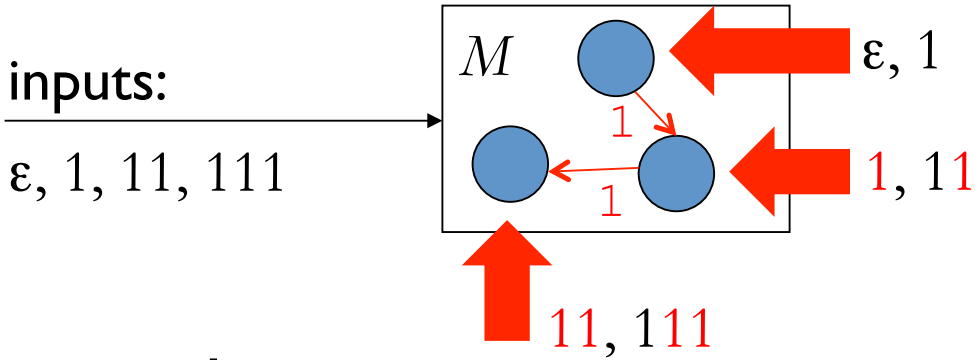


- Then after reading one more 1
  - The state of $x1 = 11$ should be rejecting
  - The state of $y1 = 111$ should be accepting

A smaller DFA

$$L = \{w: w \text{ ends in } 111\}$$

- Suppose inputs $x = \varepsilon, y = 1$ lead to the same state



- Then after reading 11
  - The state of $x11 = 11$ should be rejecting
  - The state of $y11 = 111$ should be accepting

No smaller DFA!

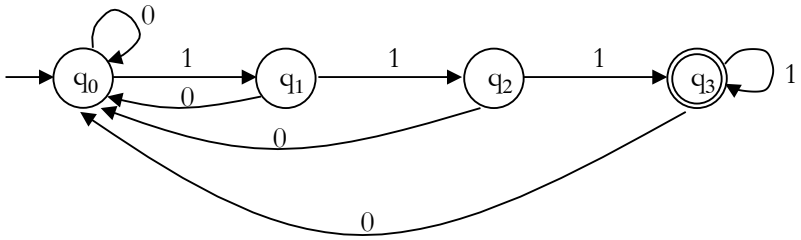
- After looking at all possible pairs (x, y)

  $(\varepsilon, 1)$ $(\varepsilon, 11)$ $(\varepsilon, 111)$ $(1, 11)$ $(1, 111)$ $(11, 111)$

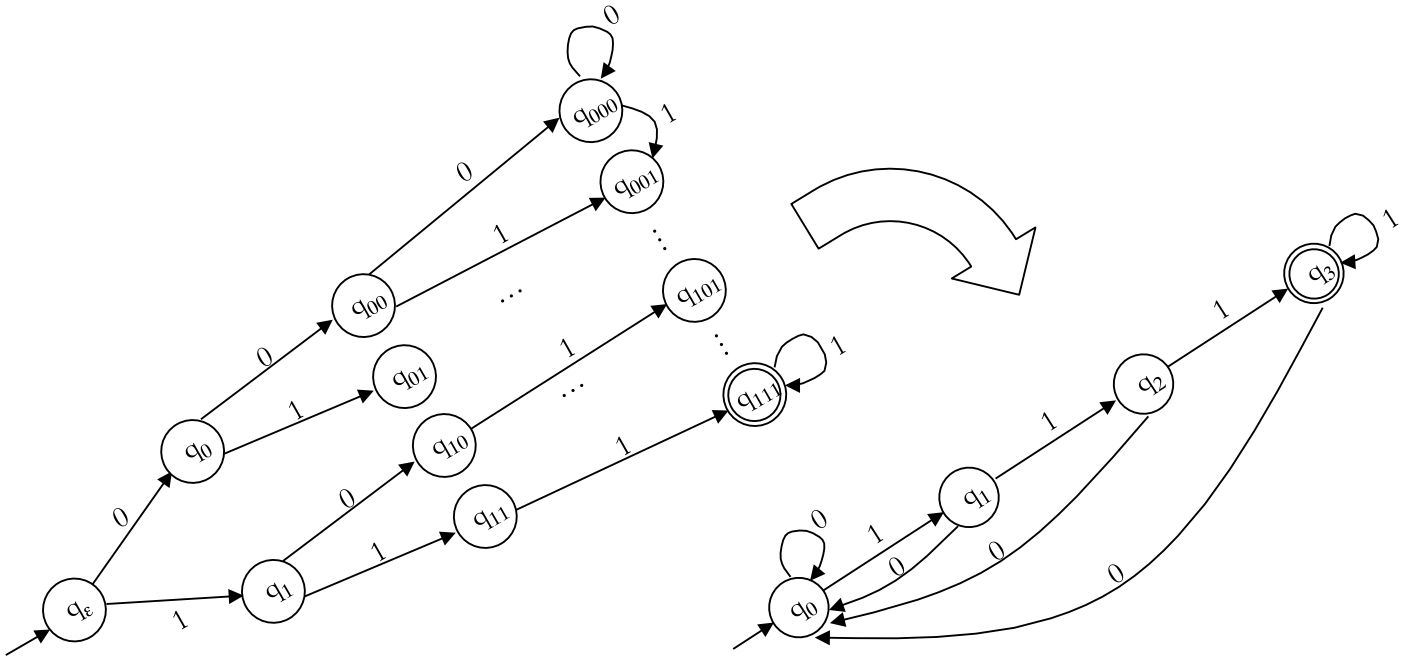
  we conclude that

  There is no DFA with 3 states for L

- So, this DFA is minimal



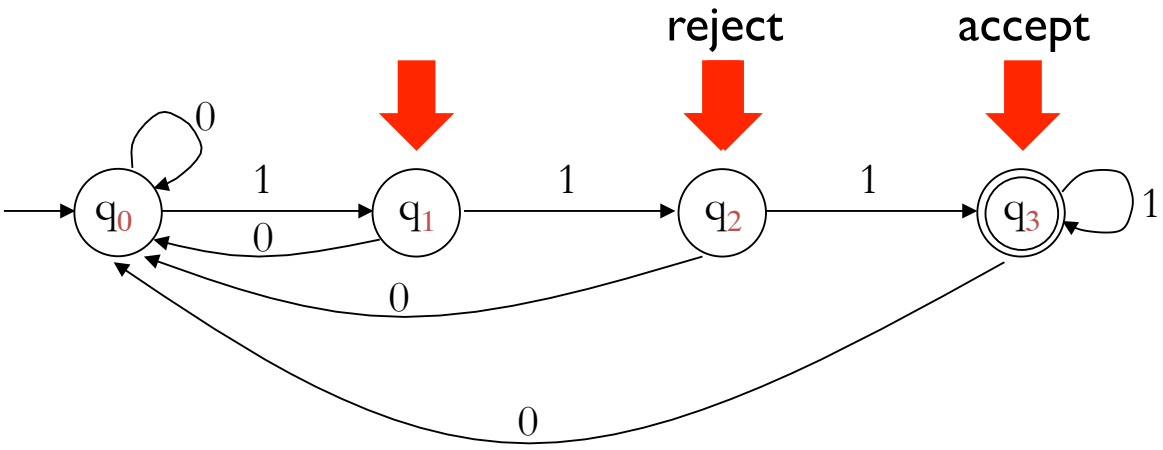
DFA minimisation



We now show how to turn any DFA for L into the minimal DFA for L

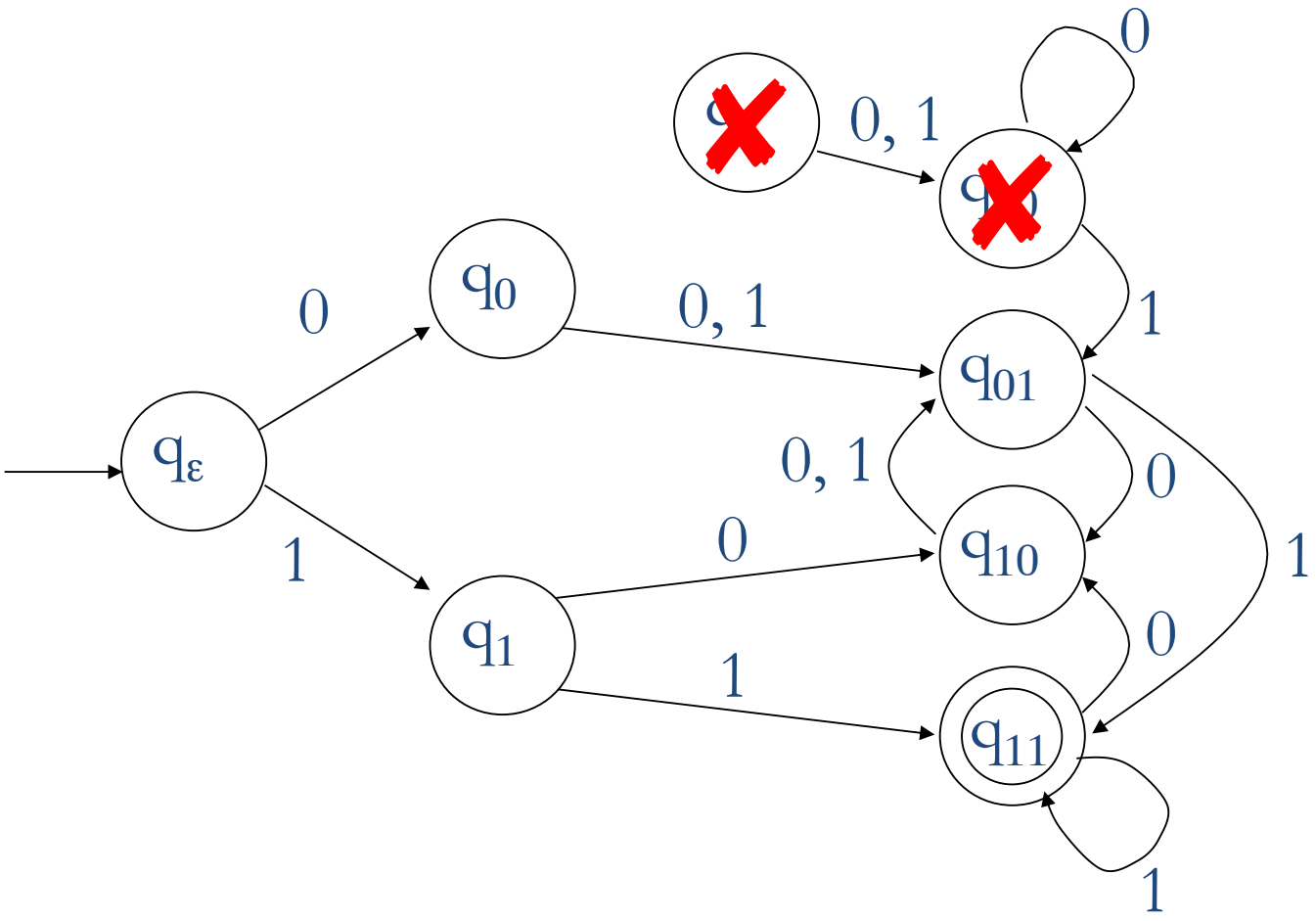
Minimal DFA and distinguishable states

- First, we have to understand minimal DFAs:

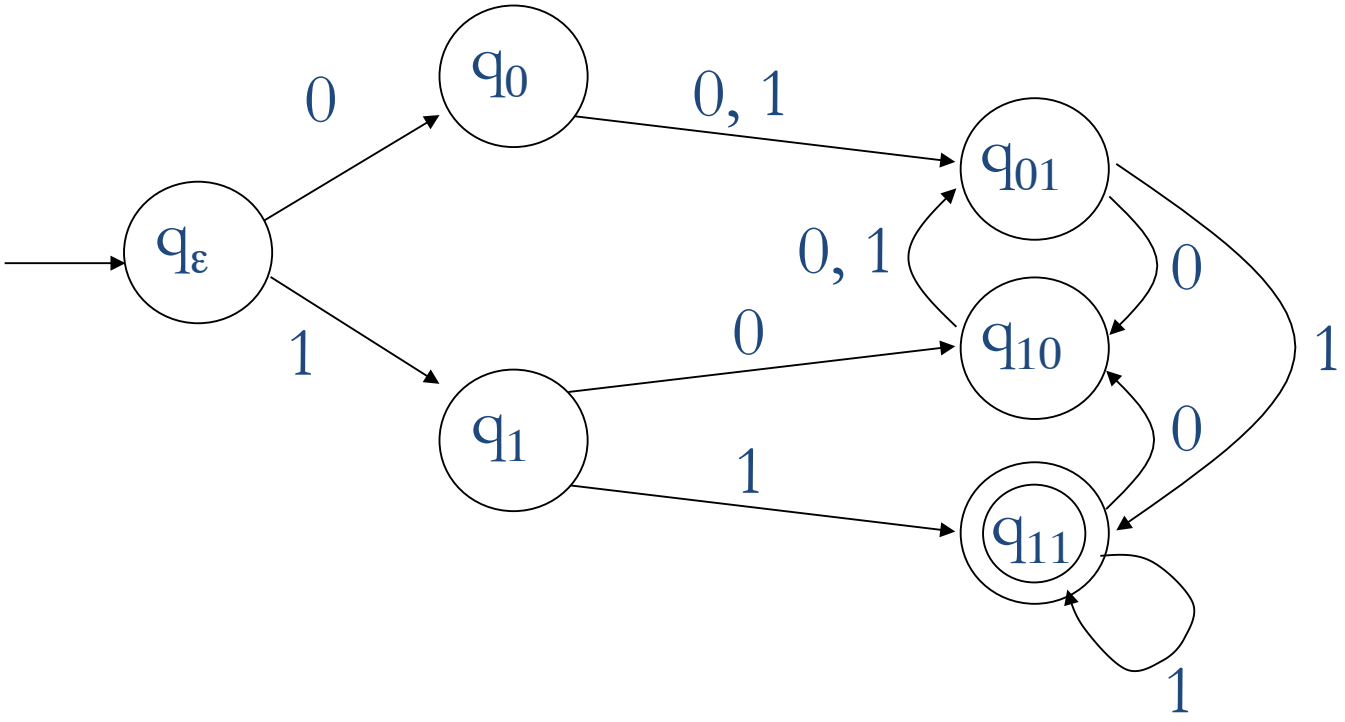


minimal DFA $\Leftrightarrow$ every pair of states is distinguishable and every state is accessible

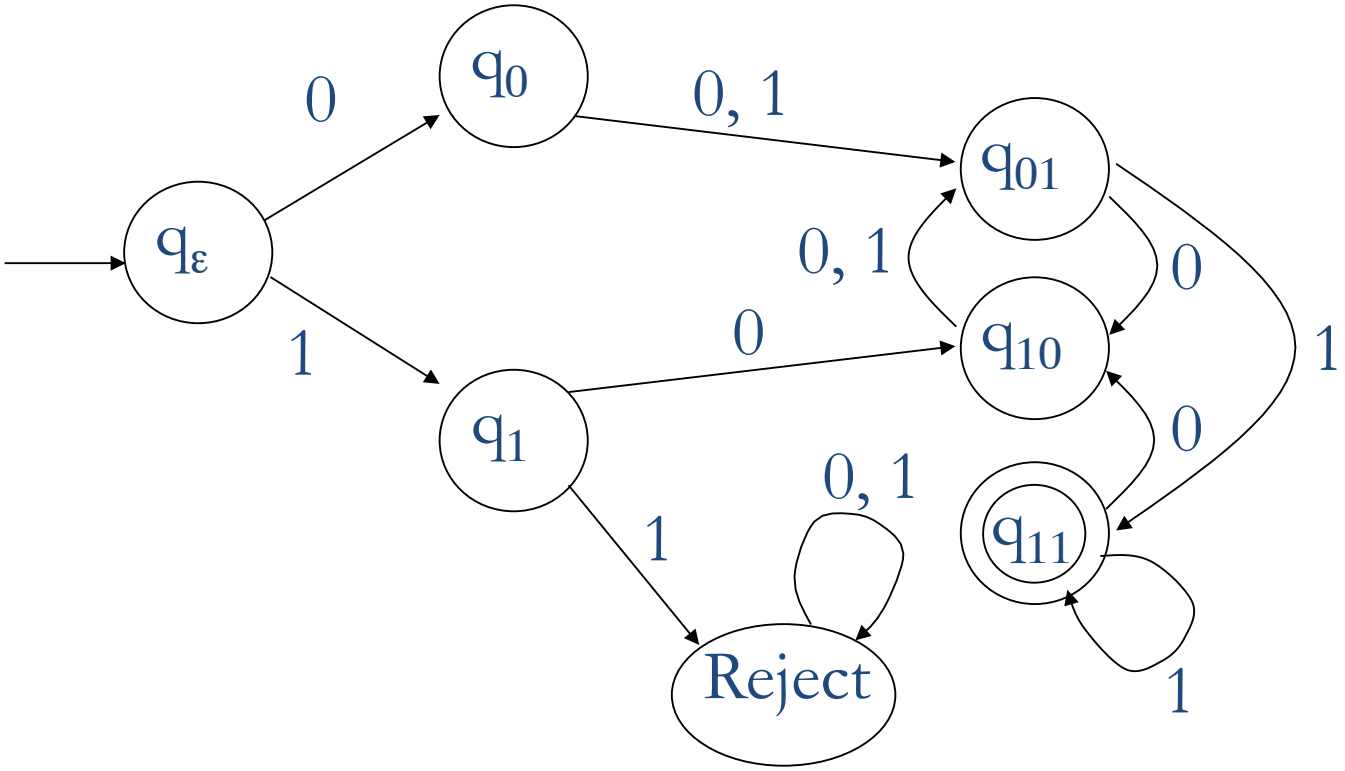
Accessible states (example 1)



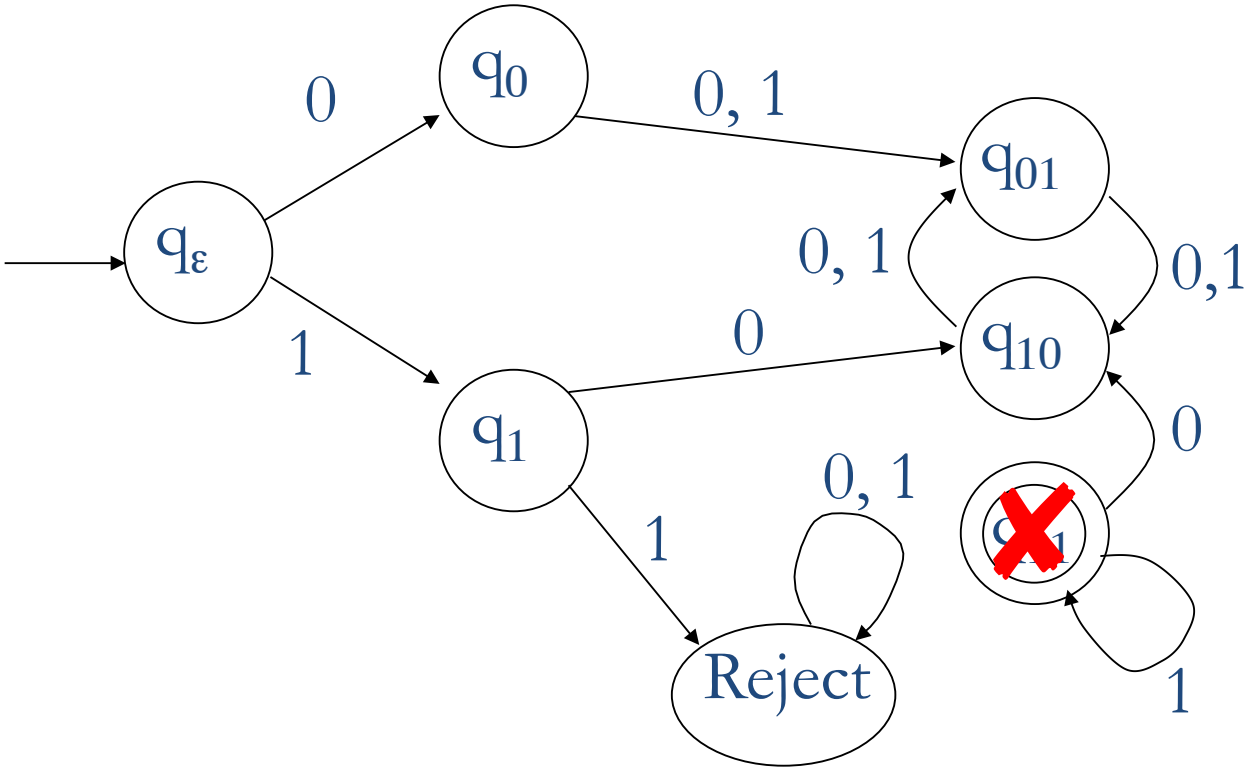
Accessible states (example 1)



Accessible states (example 2)

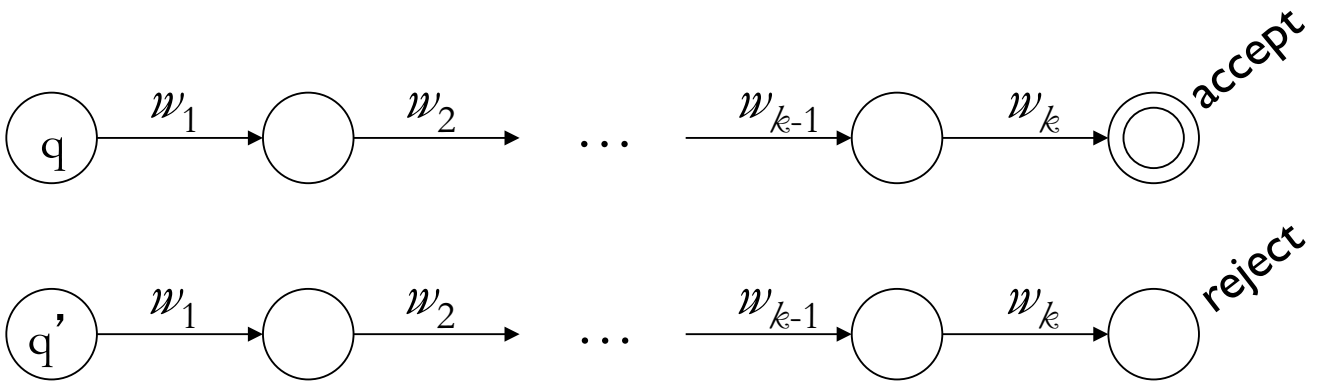


Accessible states (example 3)



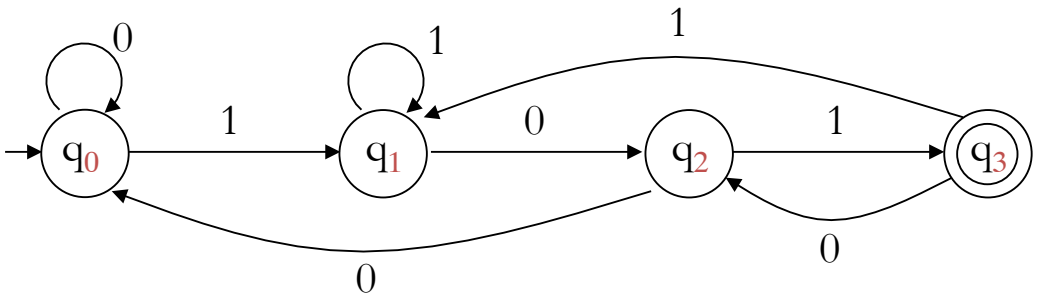
Distinguishable states

- Two states q and q' are distinguishable if



on the same continuation string $w_1 w_2 ... w_k$, one accepts, but the other rejects

Examples of distinguishable states



(q_0, q_1) distinguishable by 01

(q_0, q_2) distinguishable by 1

(q_0, q_3) distinguishable by ε

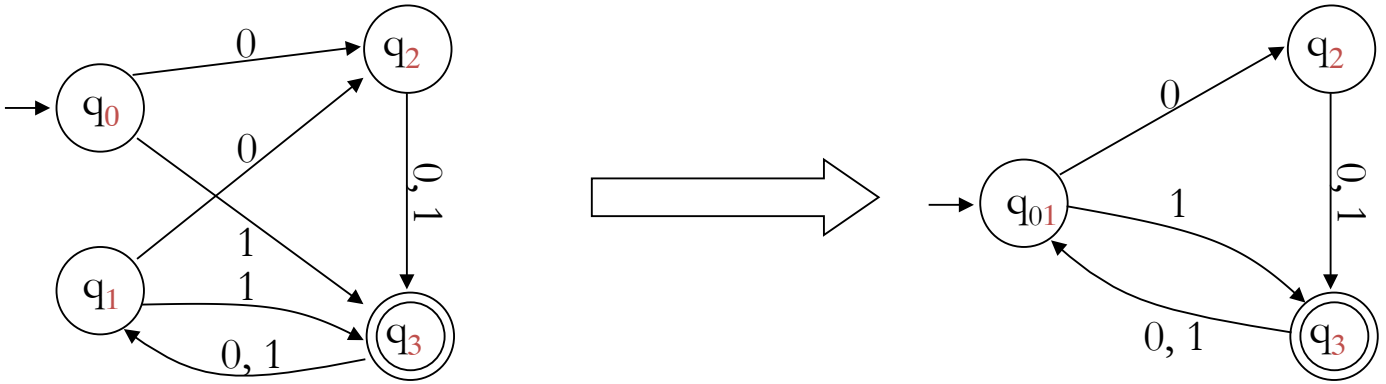
(q_1, q_2) distinguishable by 1

(q_1, q_3) distinguishable by ε

(q_2, q_3) distinguishable by ε

DFA is minimal

Examples of distinguishable states



(q_0, q_3) distinguishable by ε

(q_1, q_3) distinguishable by ε

(q_2, q_3) distinguishable by ε

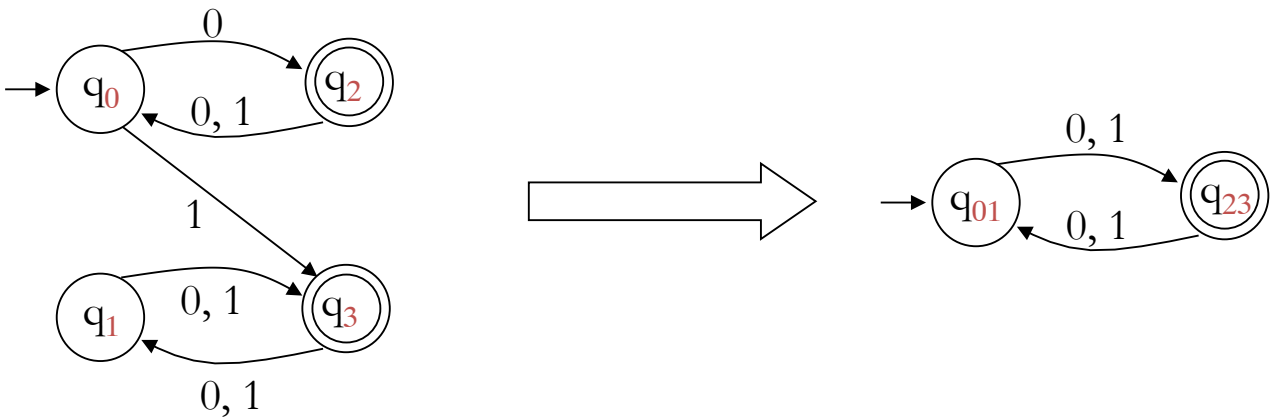
(q_1, q_2) distinguishable by 0

(q_0, q_2) distinguishable by 0

(q_0, q_1) indistinguishable

indistinguishable pairs can be merged

Examples of distinguishable states



(q_0, q_2) distinguishable by ε

(q_1, q_2) distinguishable by ε

(q_0, q_3) distinguishable by ε

(q_1, q_3) distinguishable by ε

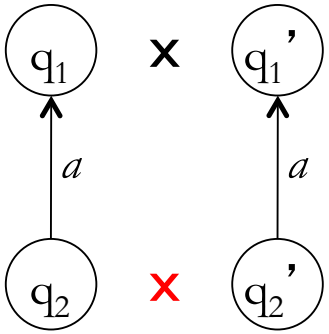
(q_0, q_1) indistinguishable

(q_2, q_3) indistinguishable

Finding (in)distinguishable states

Rule 1: 

If q is accepting and q' is rejecting
Mark (q, q') as distinguishable (x)

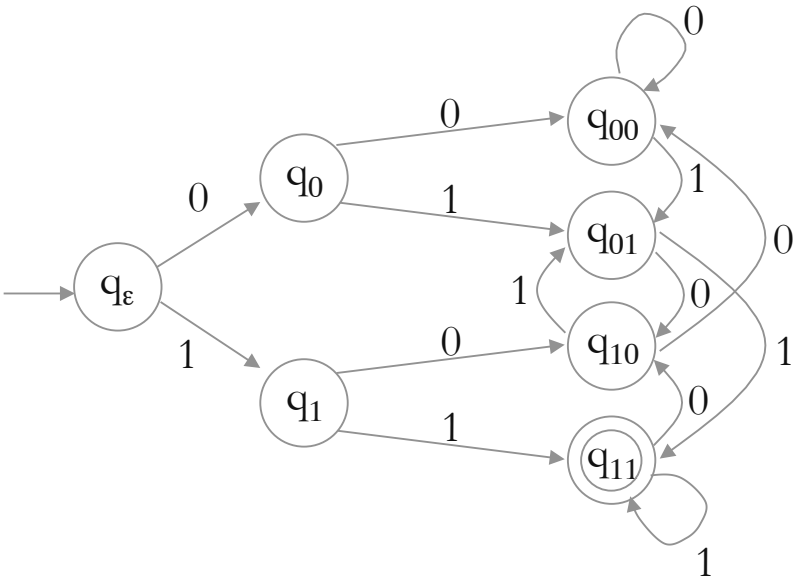
Rule 2: 

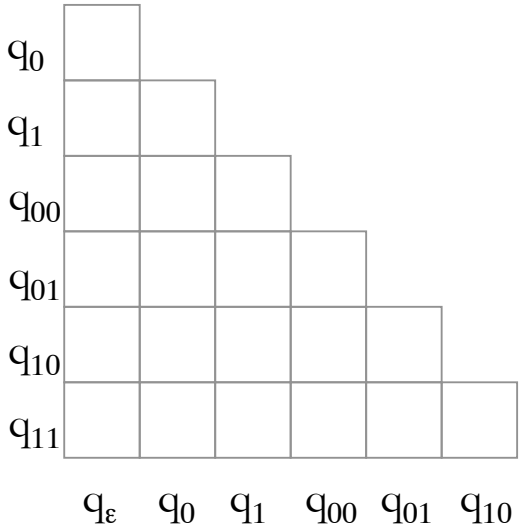
If (q_1, q_1') are marked,
Mark (q_2, q_2') as distinguishable (x)

Rule 3:

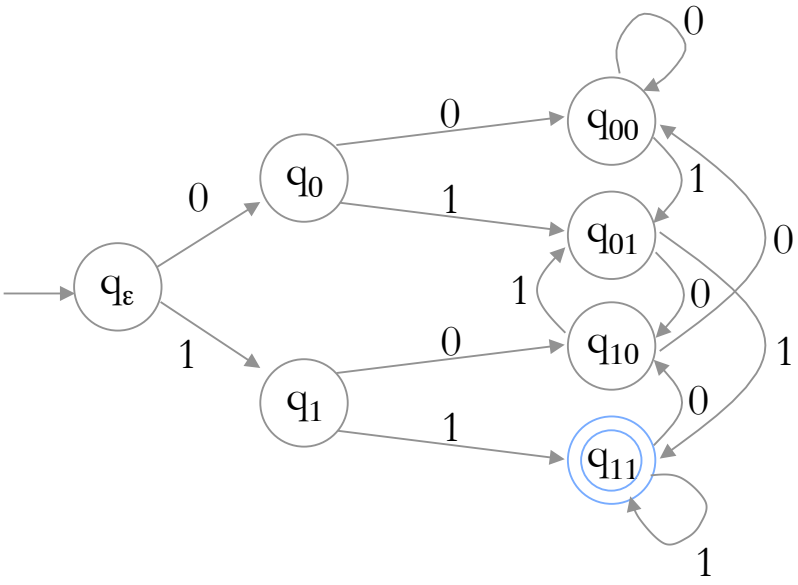
Unmarked pairs are indistinguishable
Merge them into groups

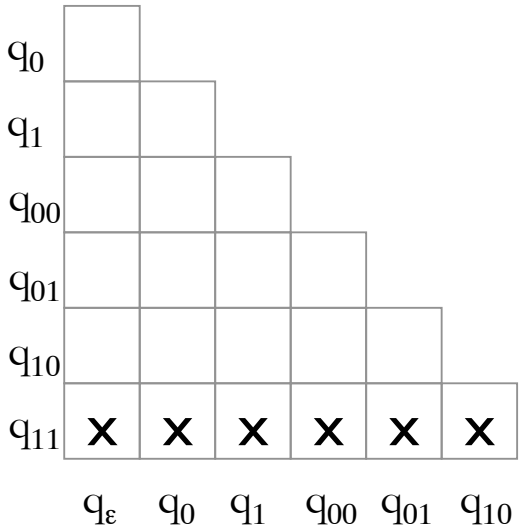
Example of DFA minimisation





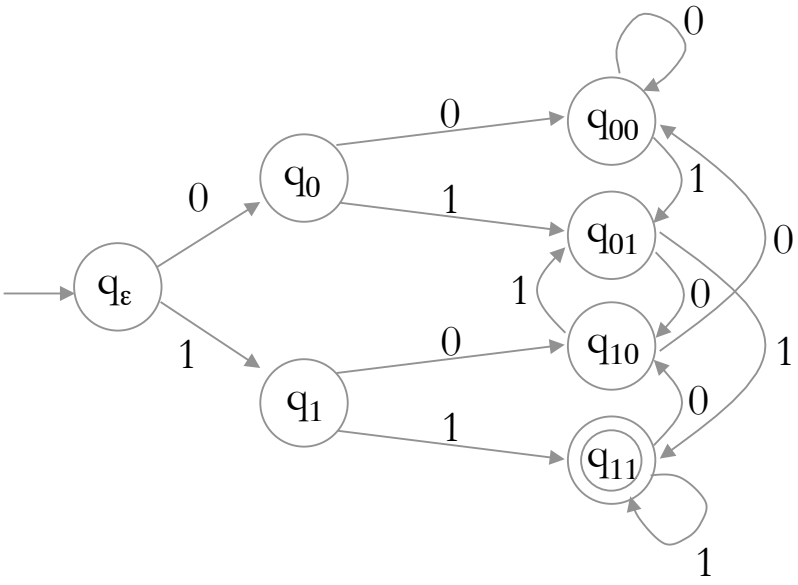
Example of DFA minimisation

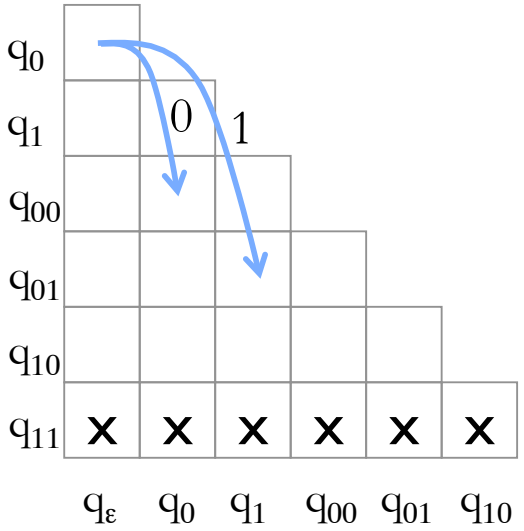




① q_{11} is distinguishable from all other states

Example of DFA minimisation

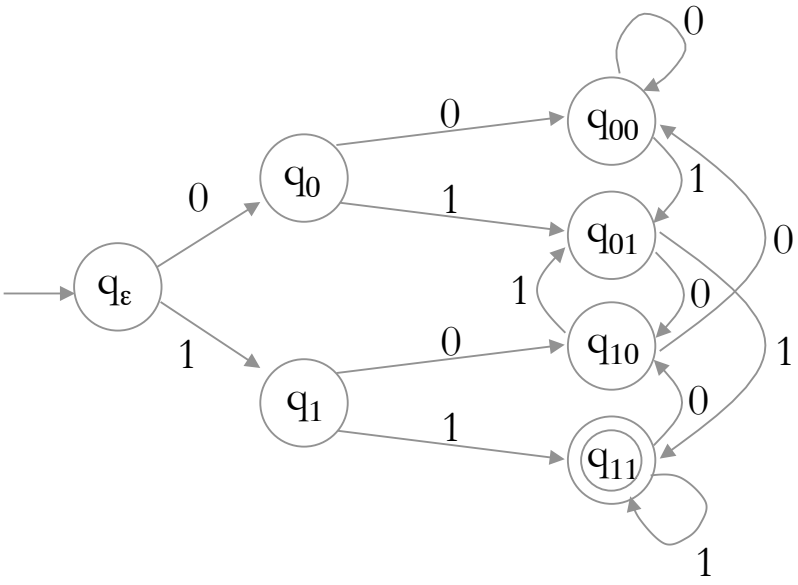


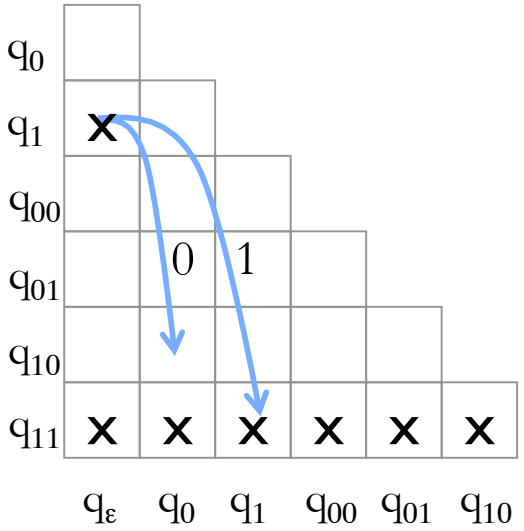


② Look at pair q_ε, q_0

Neither (q_0, q_{00}) nor (q_1, q_{01}) are distinguishable

Example of DFA minimisation

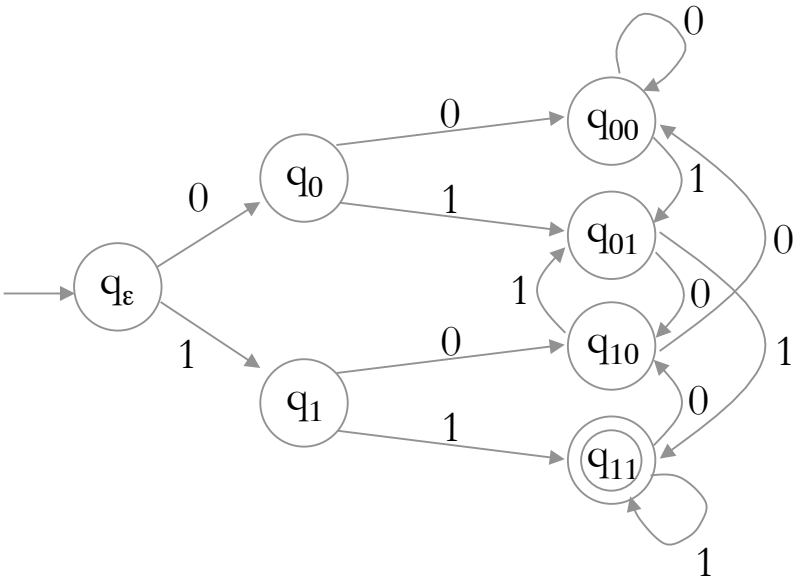




② Look at pair q_ε, q_1

(q_1, q_{11}) is distinguishable

Example of DFA minimization

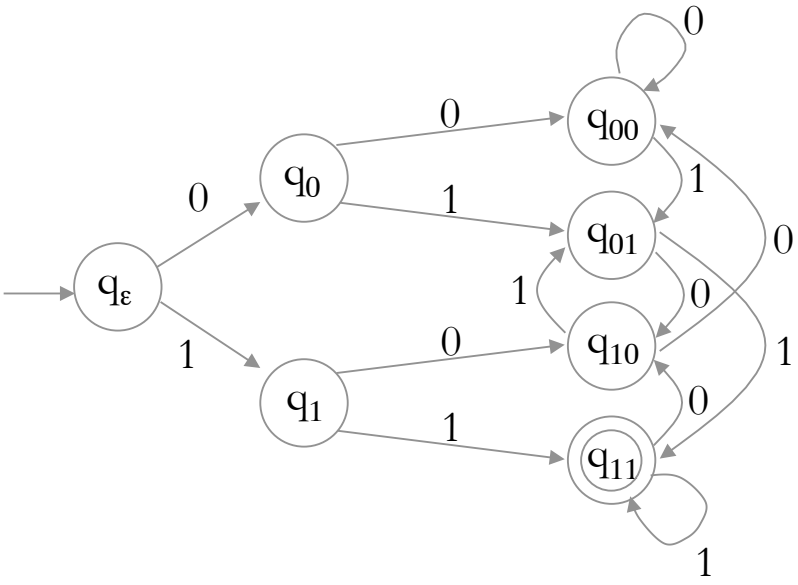


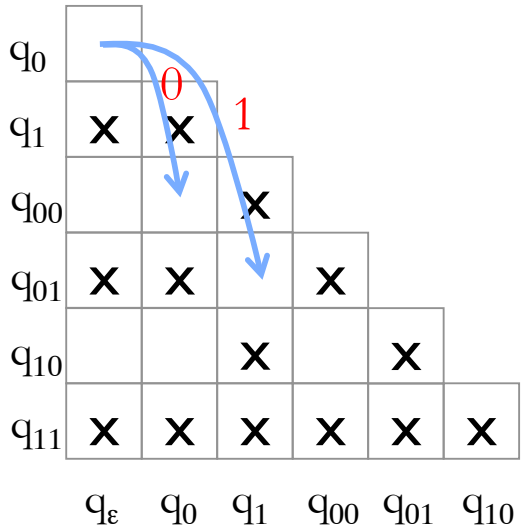
	qε	q0	q1	q00	q01	q10
q0						
q1	X	X				
q00			X			
q01	X	X		X		
q10			X		X	
q11	X	X	X	X	X	X

② After going through the whole table once

we make another pass

Example of DFA minimisation

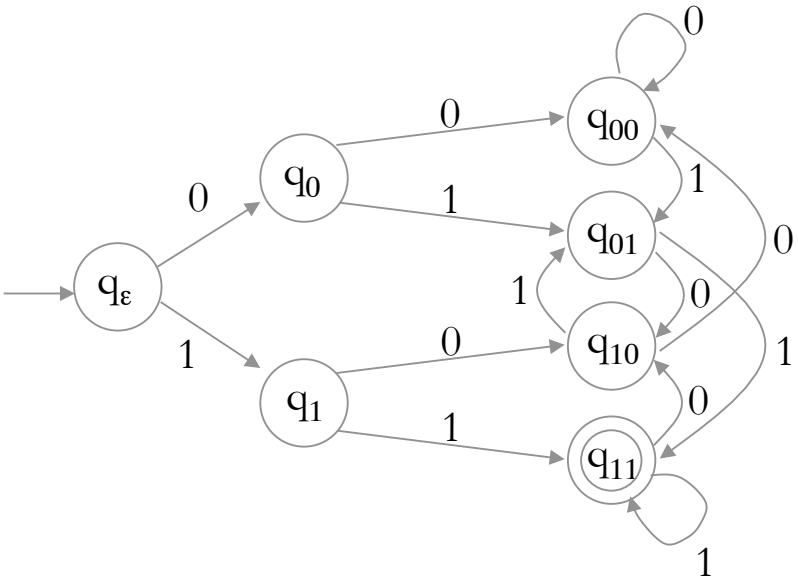


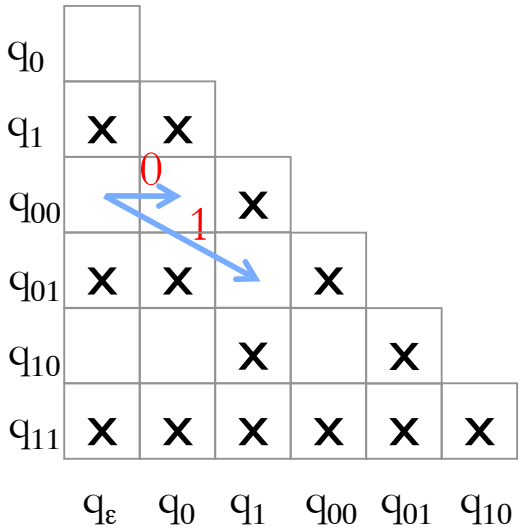


② Look at pair q_ε, q_0

Neither (q_1, q_{00}) nor (q_1, q_{01}) are distinguishable

Example of DFA minimisation

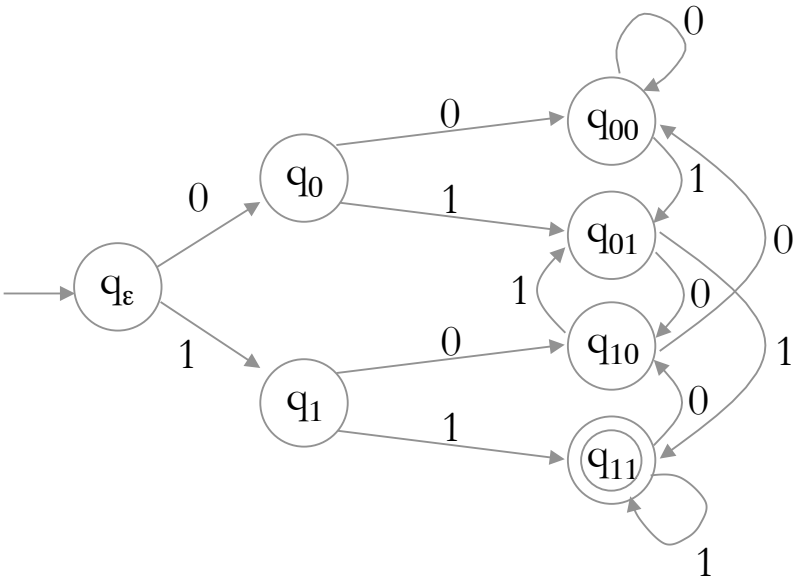




② Look at pair q_ε, q_{00}

Neither (q_0, q_{00}) nor (q_1, q_{01}) are distinguishable

Example of DFA minimisation

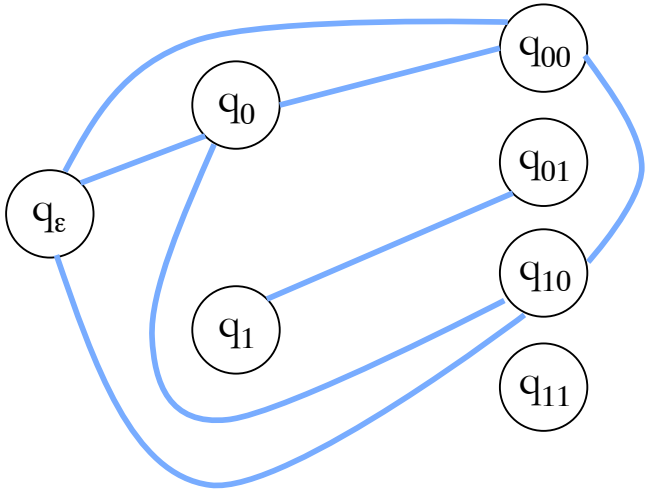


	qε	q0	q1	q00	q01	q10
q0						
q1	X	X				
q00			X			
q01	X	X		X		
q10			X		X	
q11	X	X	X	X	X	X

② In the second pass, nothing changes

So we are ready to apply Rule 3

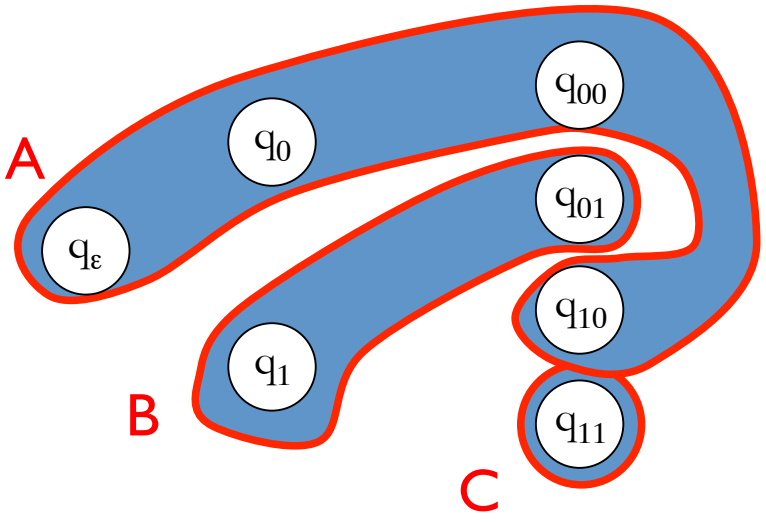
Example of DFA minimisation



	q_ε	q_0	q_1	q_{00}	q_{01}	q_{10}
q_0						
q_1	x	x				
q_{00}			x			
q_{01}	x	x		x		
q_{10}			x		x	
q_{11}	x	x	x	x	x	x

③ Merge unmarked pairs into groups

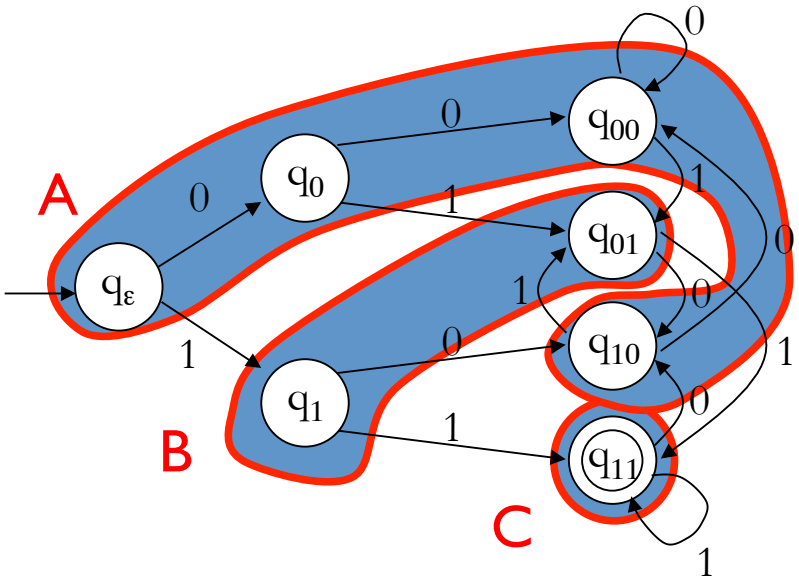
Example of DFA minimisation



	q_ε	q_0	q_1	q_{00}	q_{01}	q_{10}
q_0	A					
q_1	x	x				
q_{00}	A	A	x			
q_{01}	x	x	B	x		
q_{10}	A	A	x	A	x	
q_{11}	x	x	x	x	x	x

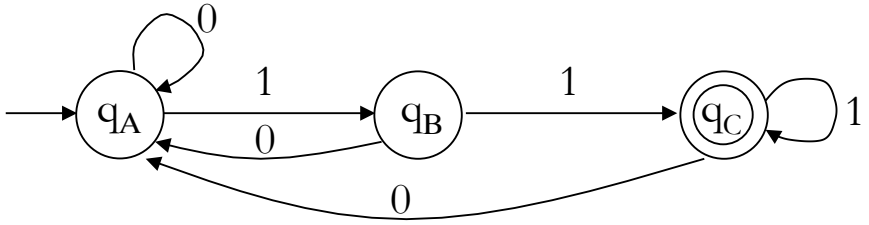
③ Merge unmarked pairs into groups

Example of DFA minimization

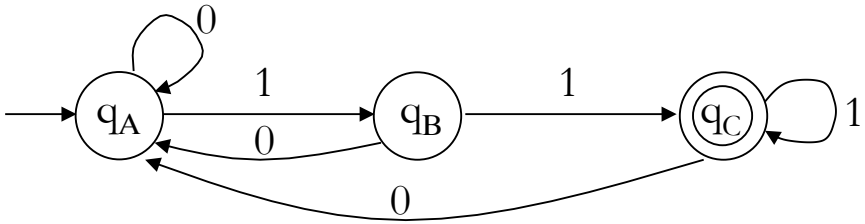


	q_ε	q_0	q_1	q_{00}	q_{01}	q_{10}
q_0	A					
q_1	x	x				
q_{00}	A	A	x			
q_{01}	x	x	B	x		
q_{10}	A	A	x	A	x	
q_{11}	x	x	x	x	x	x

minimised DFA:



Example of DFA minimisation



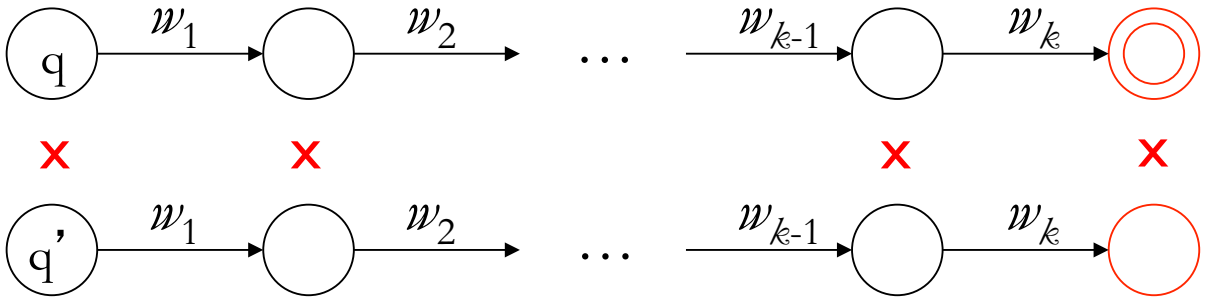
How do we know this DFA is minimal?

Answer: All pairs are distinguishable

q_B	1	
q_C	ε	ε
	q_A	q_B

Why it works

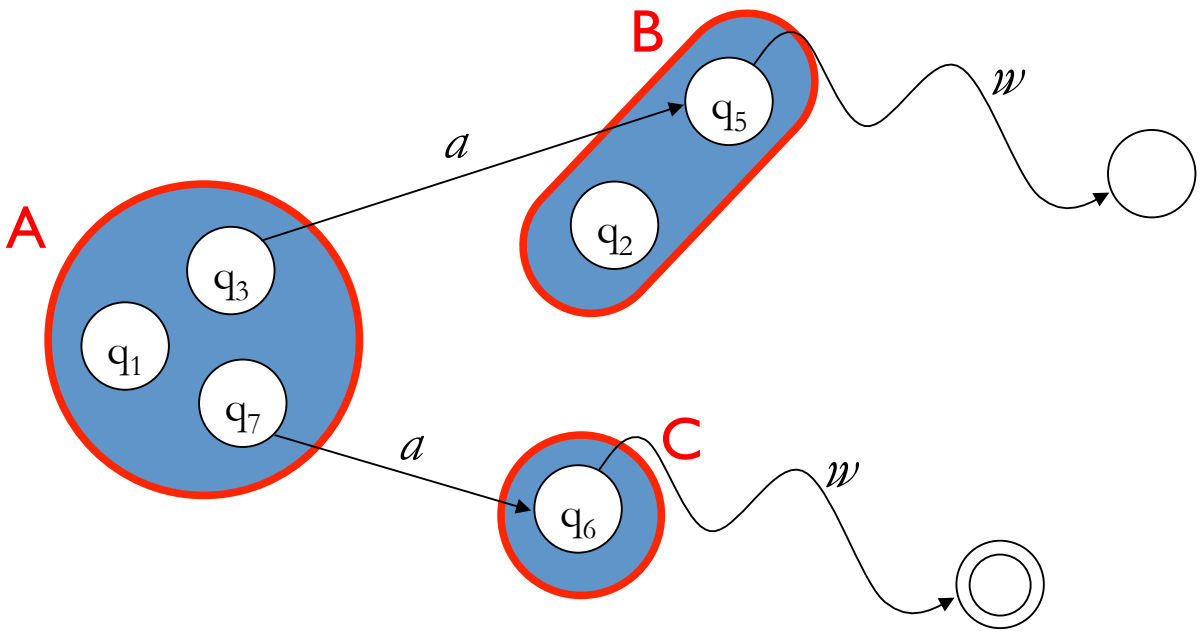
- Why do we end up finding all distinguishable pairs?



Because we work backwards

Why it works

- Why are there no inconsistencies when we merge?



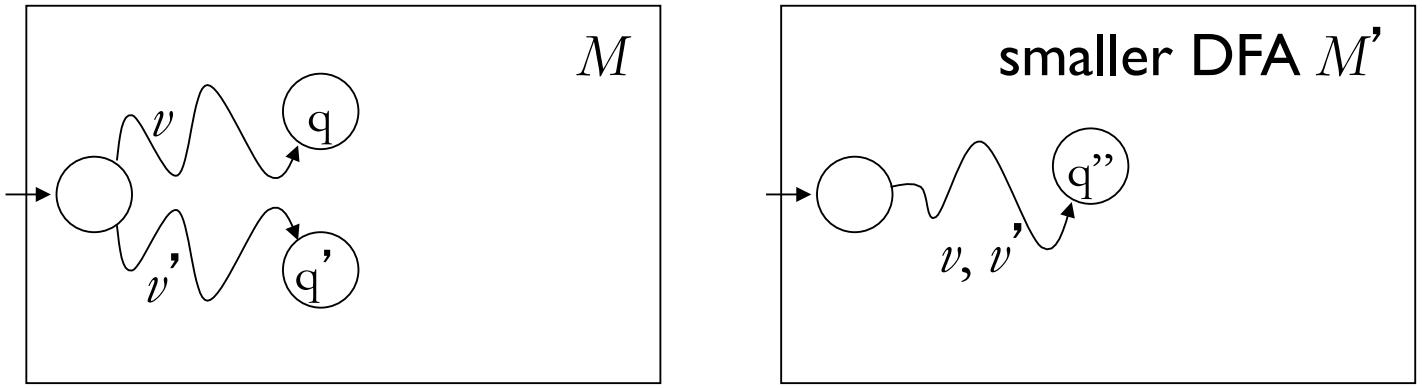
Because we only merge indistinguishable states

Why it works

- Why is there no smaller DFA?

  Suppose there is

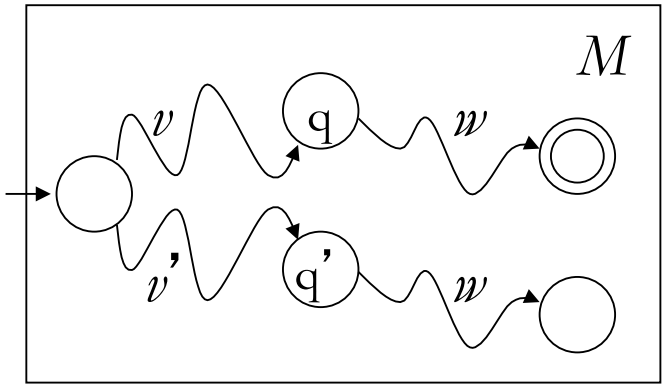
  By the pigeonhole principle this must happen:



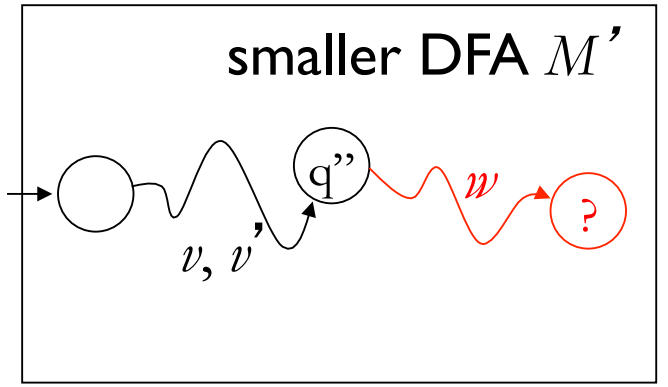
Why it works

- Why is there no smaller DFA?

But then



Every pair of states
is distinguishable



q" cannot exist!

More formally

Theorem: If a DFA has no inaccessible states, and no indistinguishable states, then it has a minimal number of states.

- **Proof**: Suppose M has n states that are accessible and distinguishable.
- Suppose M' is equivalent to M but has only n − 1 states. We can find n words that reach each of the n states of M (use accessibility)
- Two of these words must reach the same state of M'.
- We can find a suffix for these two words such that M would accept one but not the other (use distinguishability)
- But both words, with any suffix attached, must reach the same state in M'. So M' cannot accept the same set of words.