Theoretical Computer Science

Administrative Information

Lecture 5 - Manuel Mazzara

Mid-term Exam

When : Wednesday, 15 March 2023 10:40-~12:40

Where : 108 (here) and possibly another room or two

What:

Formal Languages, FSA, Pumping Lemma, PDA

Theoretical Computer Science

Complement and FSA

Lecture 5 - Manuel Mazzara

Complement and FSA

- Strings in FSA are **accepted only if the scan reaches a final state**
  - If a final state is not reached the string is not accepted
- If the input string is always scanned (**complete FSA**), then it suffices to “swap yes and no” (F with Q-F)
- If the end of the string cannot be reached (**not complete FSA**), then swapping F with Q-F does not work
- In the case of FSAs there is an easy workaround
  - **Completing the FSA**

General Observation

- Swapping final states means **asking the opposite question** (having an automaton for complement language) **and looking for positive answers** (accepted strings)

- In general, **we cannot consider the negative answer to a question as equivalent to the positive answer to the opposite question!**
  - **We will see what this means for Turing machines**

- In fact, **closure over complement is fundamental when it comes to computability issues**!

Complement and TM (spoiler ahead!)

- TMs are more expressive than FSA
  - More “programs” can be expressed

- TMs and Turing-complete programming languages allows **nonterminating programs** (it is important to express important algorithms)

- A TM accepts a string if it will eventually halt and say "Yes“
- **If it does not halt you cannot know whether it will ever do**
  - Non closure wrt complement

Theoretical Computer Science

Models of computation and memory

Lecture 5 - Manuel Mazzara

Three subtly different problems

- Consider strings over the alphabet {a,b,c}

- Let us consider three problems:

  - determining whether **a string ends in an *a***
  - determining whether **a string is a palindrome**
  - determining whether **a string is of the form $a^n b^n c^n$**

- What problem is easier (if any)?

It is all about the memory model!

- Solving the first problem only **needs to "remember" the same amount of information** no matter how long the input is: the last letter of the word

- The **amount of memory** that the second and third problems require is instead **variable**
  - it depends on the size of the input
  - the first half of the palindrome or can be arbitrarily large

- The difference between the second and third problems is even more subtle
  - the second problem can be solved **moving only one direction in memory**
  - the third problem requires the ability to **"look back" in memory**

A hierarchy of models

- **Some problems are more difficult than others**
  - How to formalize these **notions of difficulty**?

- We build **abstract models of computation**, and we test their ability to decide about these questions

- Each different model of computation has **just enough power** to solve the problem
  - As you may have noticed **(only) the first problem can be solved by an FSA**

Theoretical Computer Science

Regular Languages and closure

Lecture 5 - Manuel Mazzara

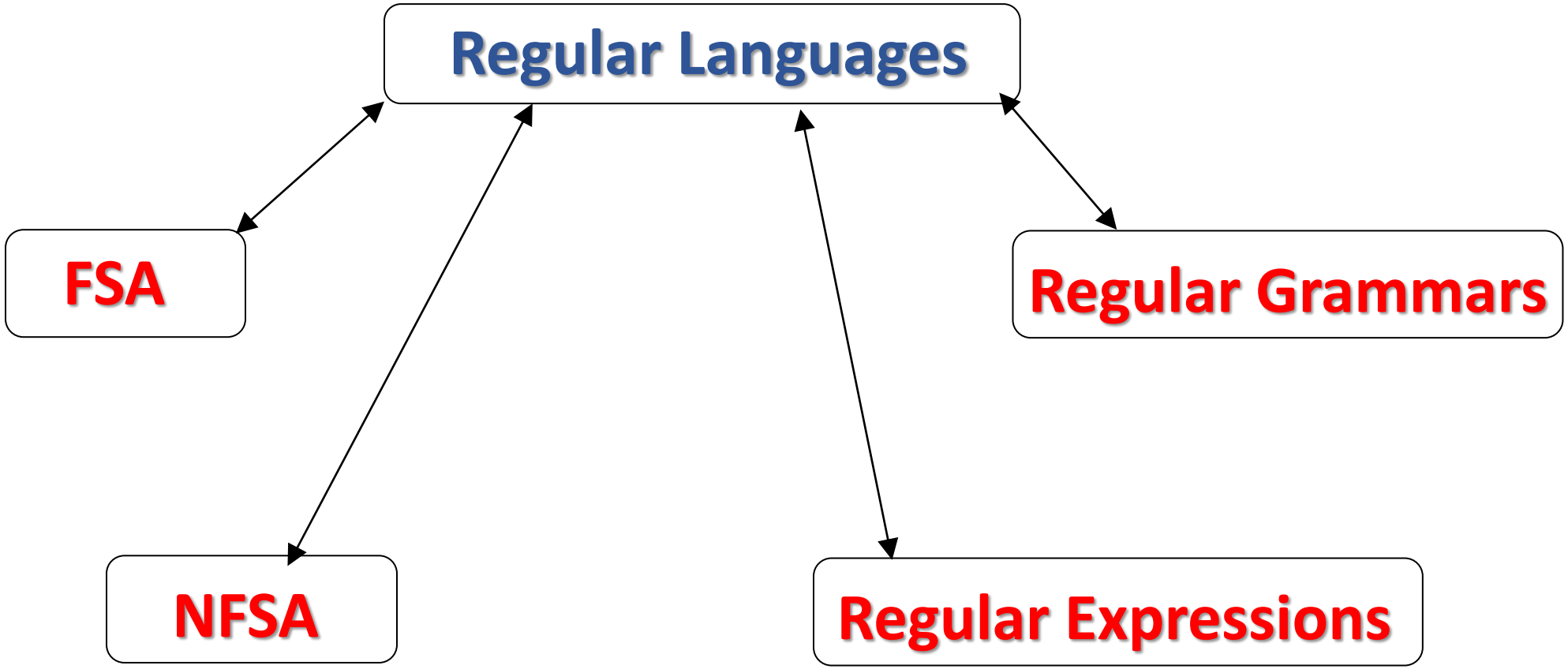
Regular Languages

- A **regular language** is a language recognized by a FSA

- Regular languages are very useful in **input parsing and programming language design**

- We will see models that are equivalent to languages recognized by FSA
  - **Regular expressions**
  - Specific type of **generative grammars**

Representations of Regular Languages



Closure for languages

- $\mathcal{L}$ = $\{L_i\}$: **family** of languages
- $\mathcal{L}$ is **closed w.r.t. operation OP** if and only if, for every L_1 , $L_2 \in \mathcal{L}$, L_1 OP $L_2 \in \mathcal{L}$.
- $\mathcal{R}$: **regular languages** (recognized by FSAs)
- **$\mathcal{R}$ is closed w.r.t. set-theoretic operations, concatenation and "*"**

Theoretical Computer Science

Pumping Lemma for Regular Languages

Lecture 5 - Manuel Mazzara

Representations of Regular Languages

- We are given a **Regular Language** L

- It means that Language L comes in one of the standard representations:
  - **Deterministic Finite State Automata**
  - Nondeterministic Finite State Automata
  - Regular expressions
  - Regular grammars

All finite languages are regular

You can easily build an FSA to recognize a finite language!

Some infinite languages are regular – not all

There are languages that are not regular

What does it mean?

These languages cannot be represented in any of these standard representations

Examples of non regular languages

$$\{a^n b^n : \ n \geq 0\}$$

$$\{ww^R : \ w \in \{a,b\}^*$$

- **How can we prove that a language is not regular?**
- Can we prove that there is no FSA that accepts it?
- This is not easy to prove! How would you prove it?

Proving that a language is not regular

- To prove that a language is **regular**, we can just find an FSA (or another standard representation) for it
- To prove that a language is **not** regular, we need to **prove that there is no possible FSA** (or another standard representation) for it
- This is **possible in principle**. For each specific language, a technical proof with ad-hoc argument could be built
- Here, however, we will show something more powerful: a **generalization**

Such generalization is known with the name of…

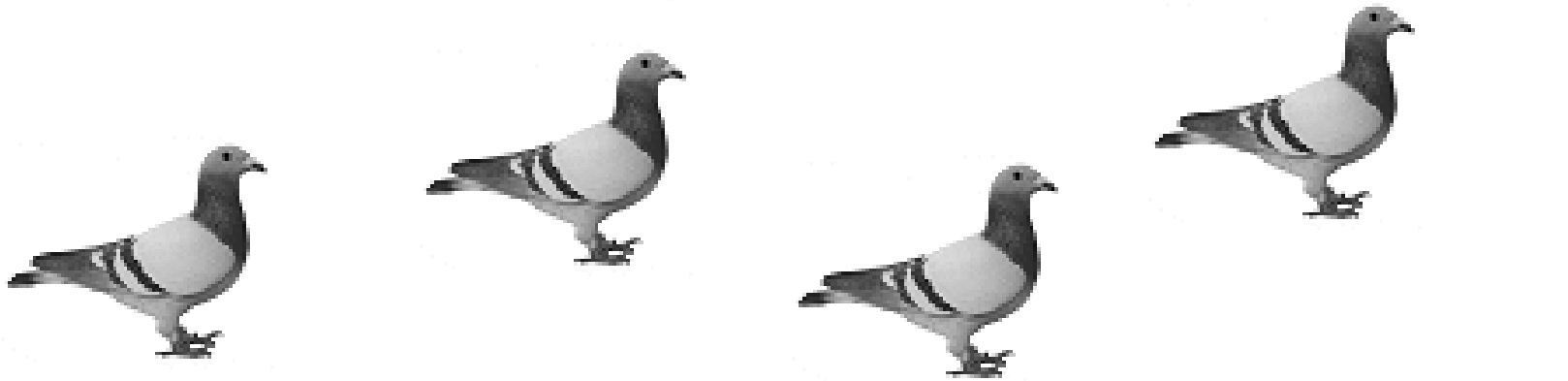
Pumping Lemma

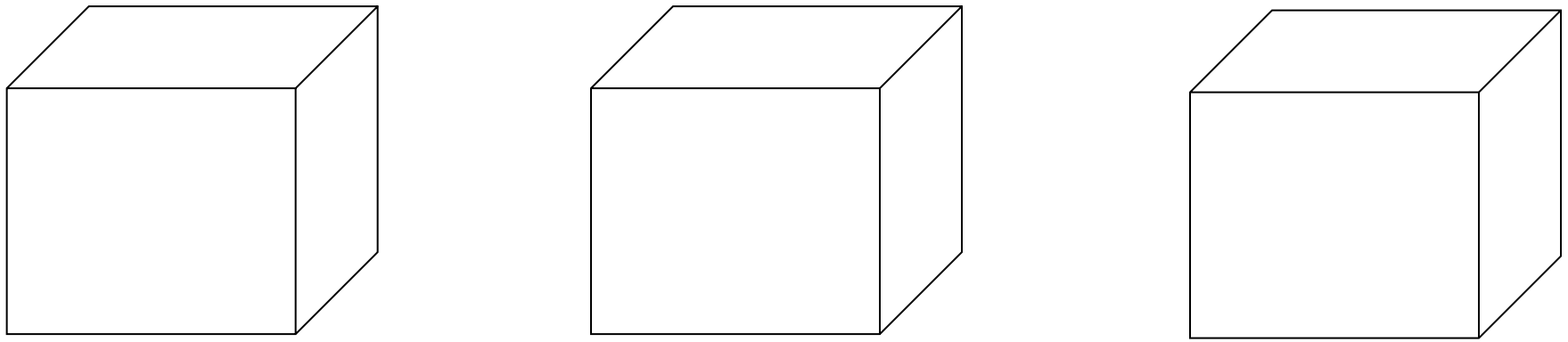

ONE DOES NOT
SIMPLY
UNDERSTAND PUMPING LEMMA
ator.net


KEEP
IT
SIMPLE

The Pigeonhole Principle (1)

4 pigeons



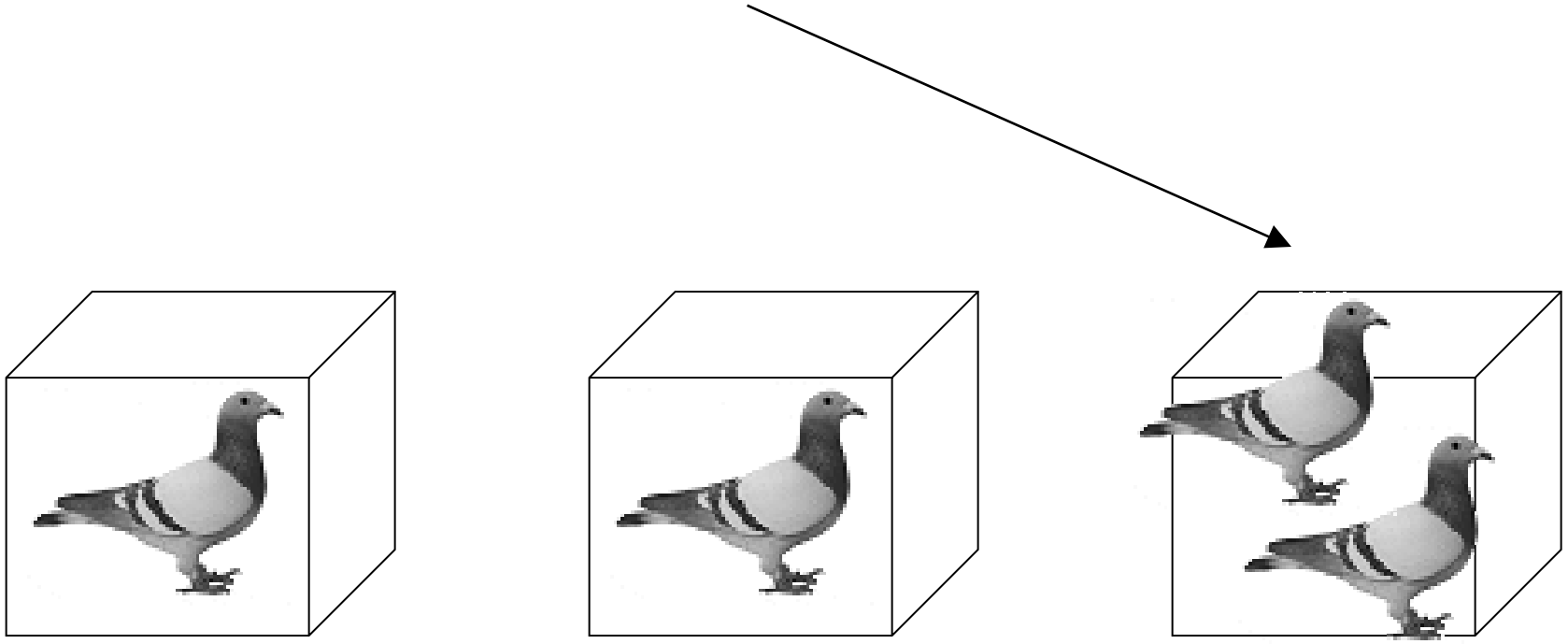


3 pigeonholes

The Pigeonhole Principle (2)

A pigeonhole must

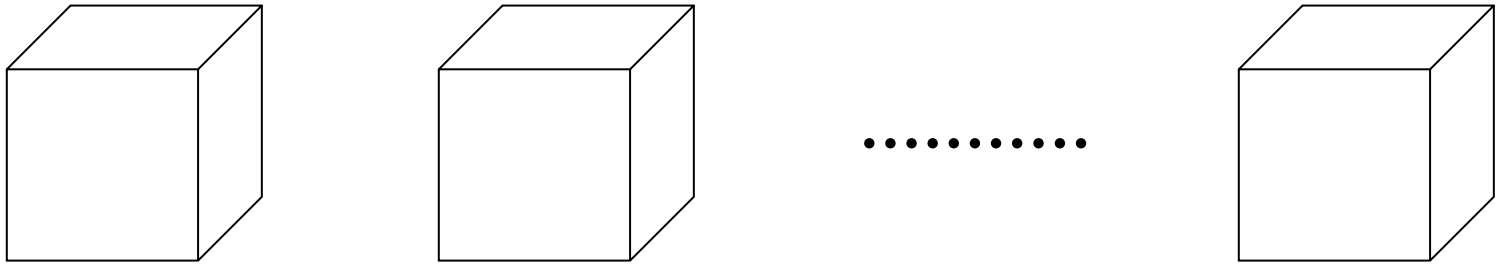
contain at least two pigeons



Generalization

n pigeons

m pigeonholes $n > m$



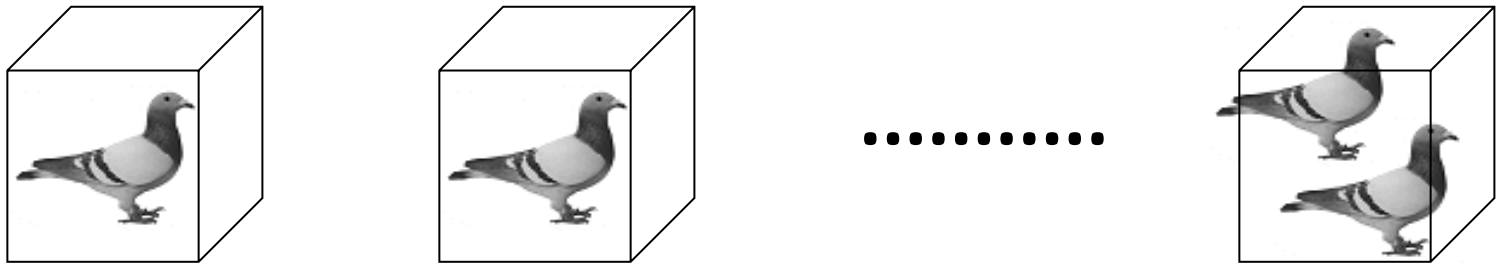
Straightforward results

n pigeons

m pigeonholes

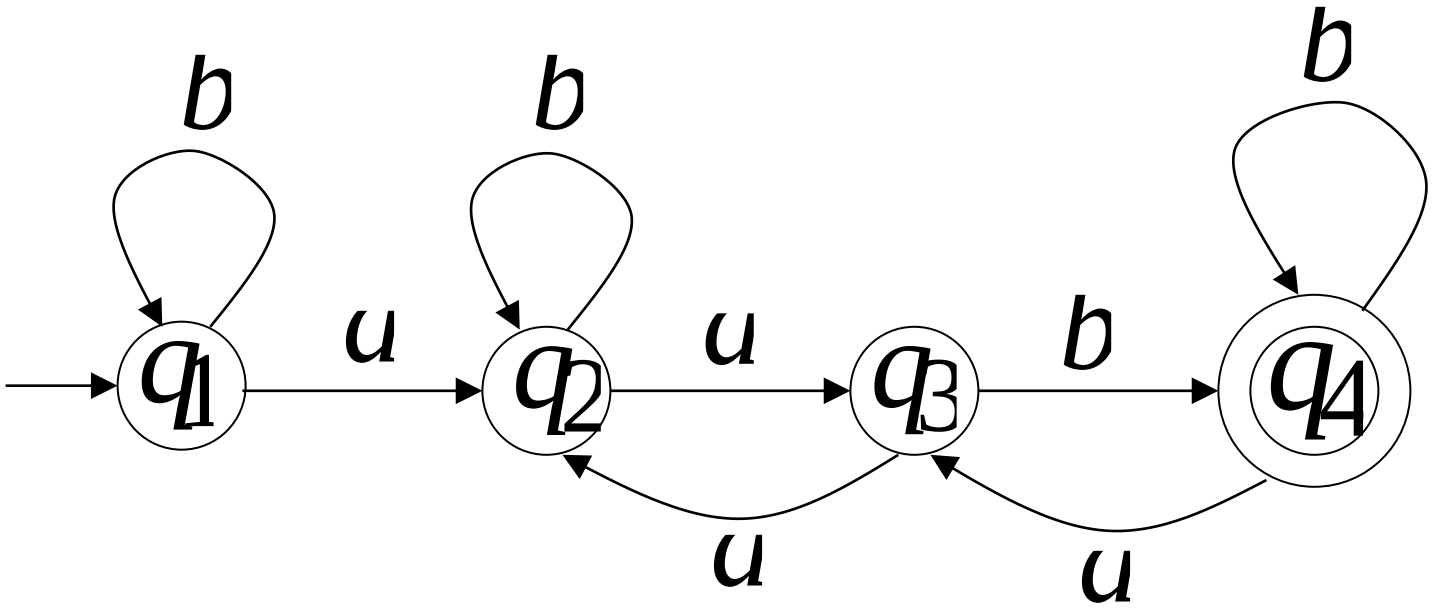
$n > m$

There is a pigeonhole with at least 2 pigeons



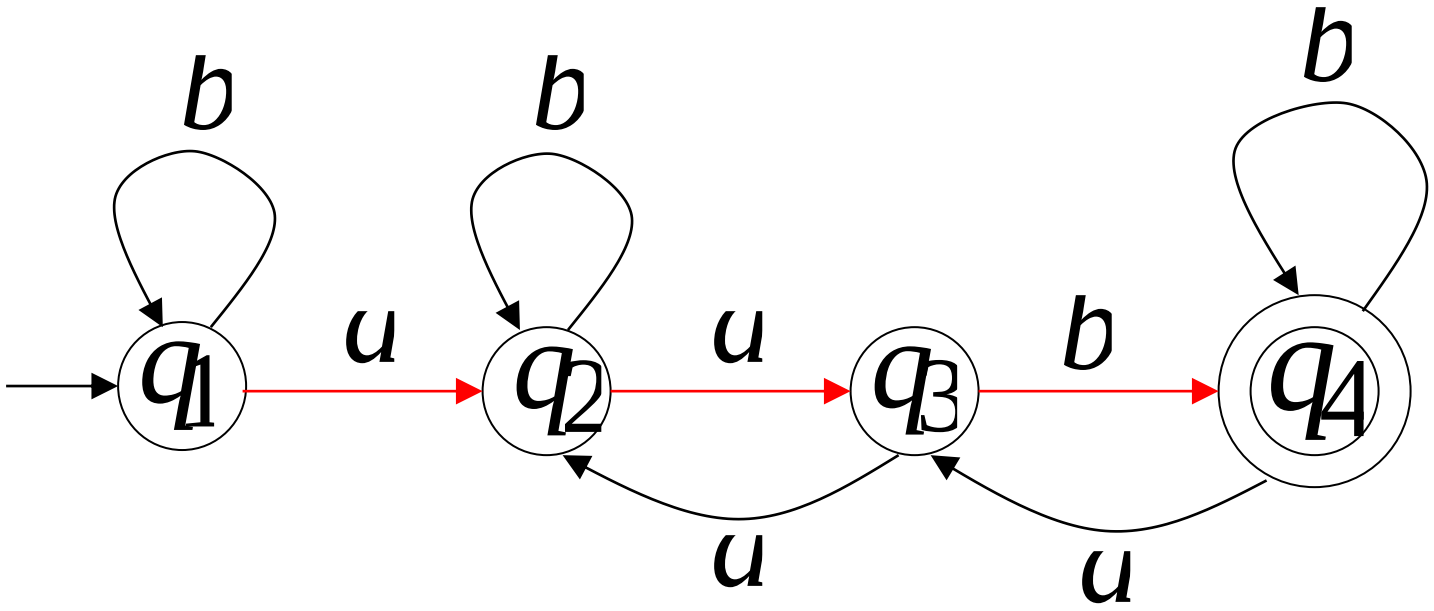
Example (1)

FSA with 4 states



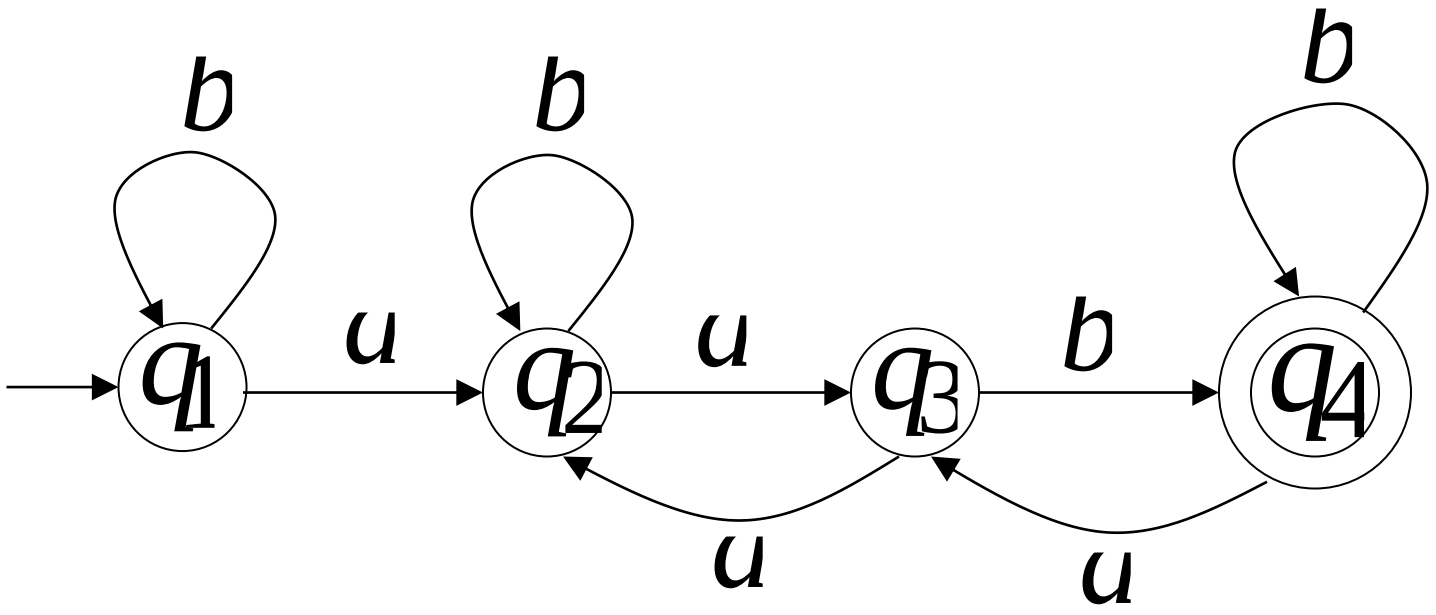
Example (2)

The path *aab* has no repetition of state



States can be repeated

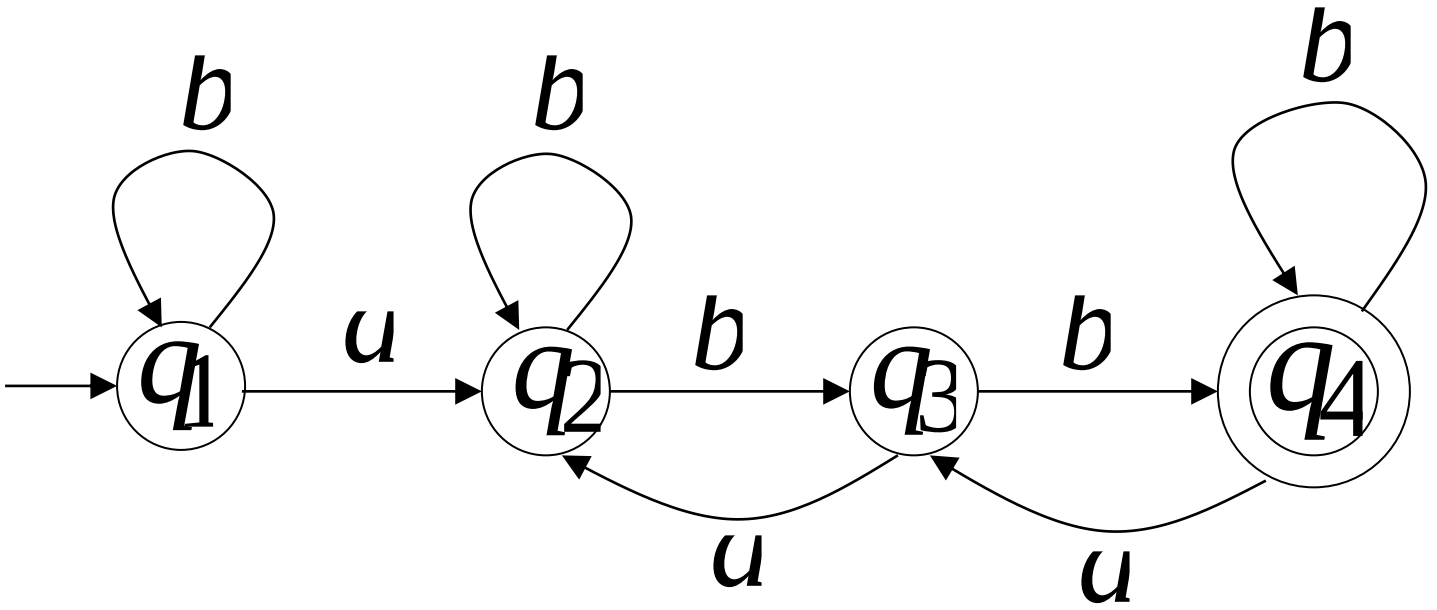
$aabb$
$bbaa$
$abbabb$
$abbbabbab$



Simple fact

If the path for string X is greater than 4

then a state <u>must be repeated</u>



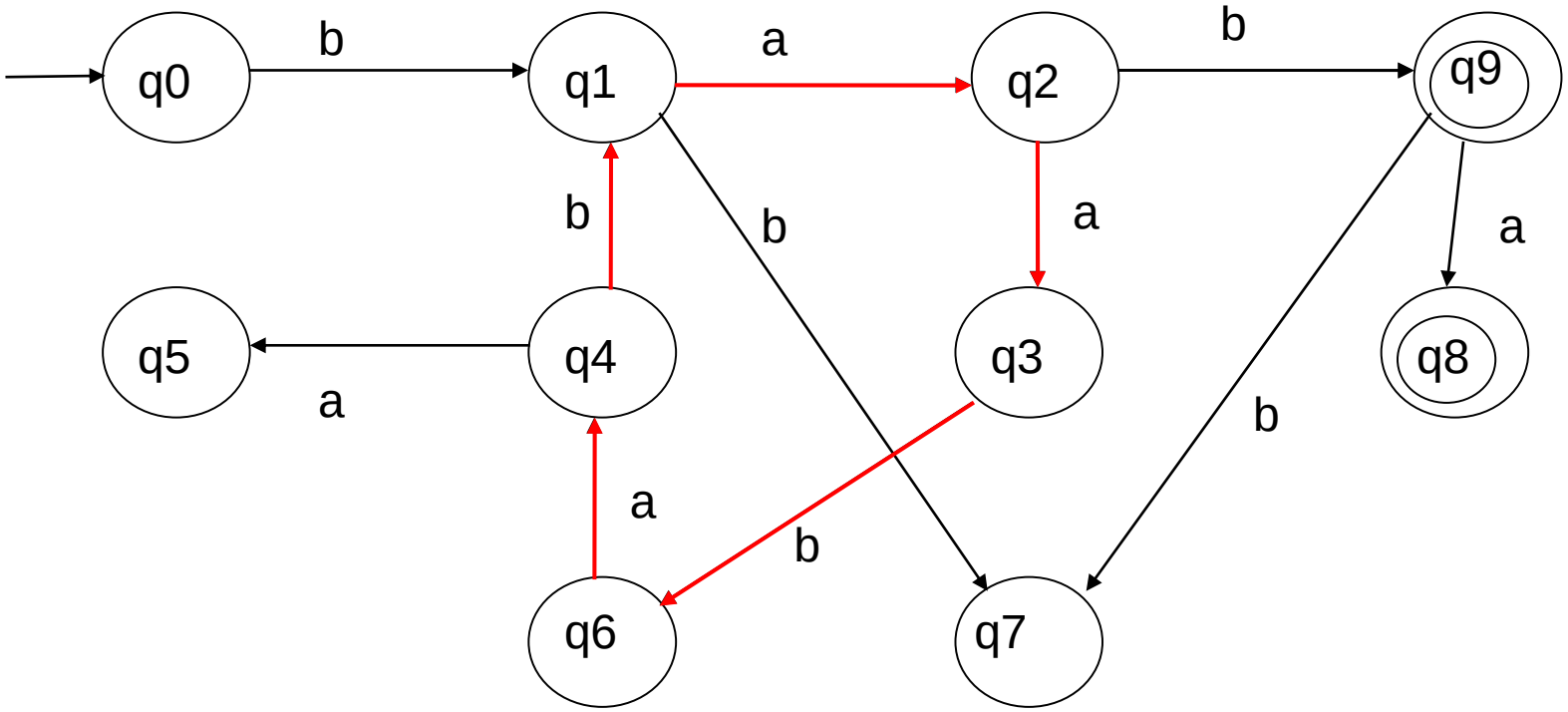
Anyone called me?

FSA and Pigeonhole Principle

- If in a path transitions $\geq$ states then a state is **repeated**
- *Transitions are pigeons*
- *States are pigeonholes*
- **If a string has length greater than the number of states, there has to be a repeated state in the visit**

Cycles

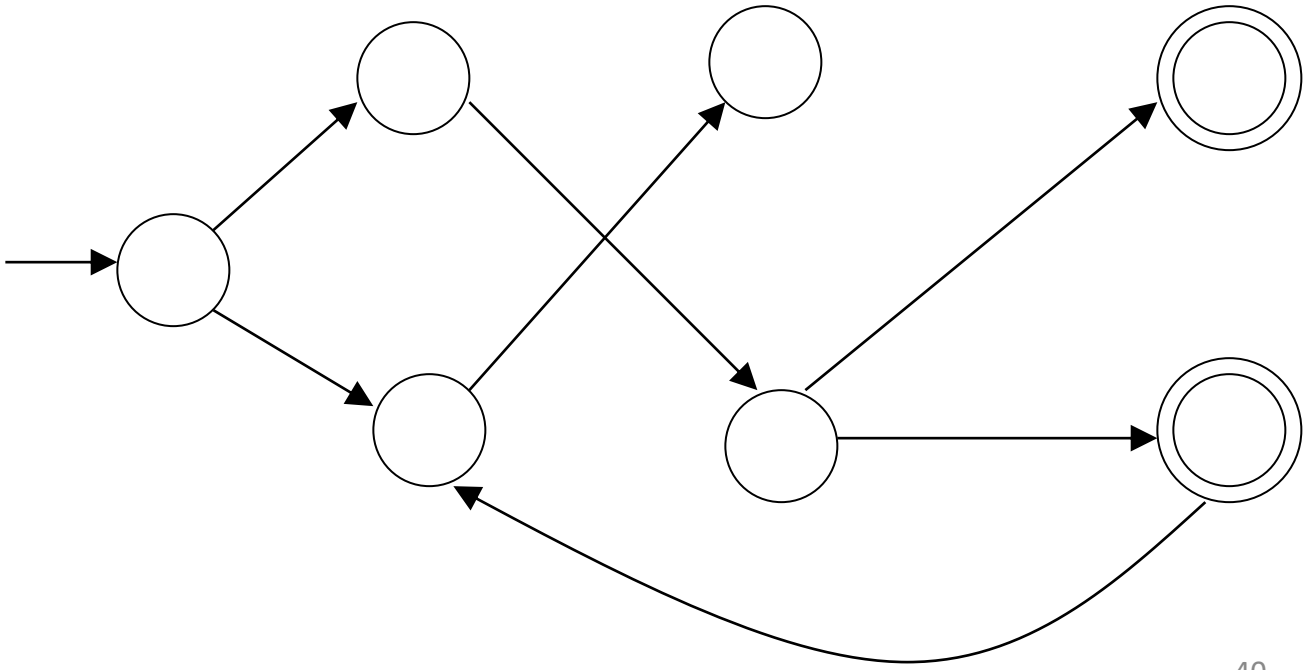
There is a cycle: q1 ----aabab---> q1



If one goes trough the cycle once, then one can also go through it 2,3, …, n times

Languages and Cycles (1)

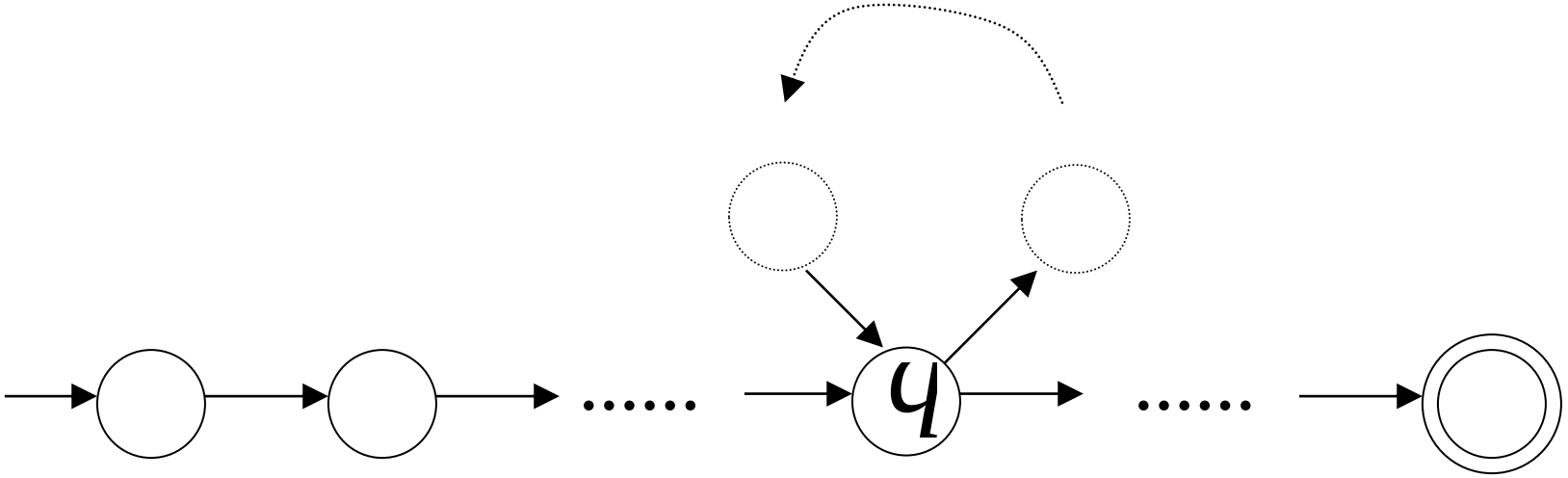
- Consider an **infinite** regular language
- The FSA recognizing it has *m* states



Languages and Cycles (2)

Take string X belonging to L and with length greater than *m*

A state has to be repeated in the walk



Pumping Lemma for Regular Languages

- It is an essential property of **all regular languages**

- All **sufficiently long words** in a regular language may be *pumped*, i.e. have a middle section of the word repeated an arbitrary number of times, **to produce a new word that also belongs to the same language**

- **The pumping lemma is used to prove that a particular language is non-regular**
  - You will see a few examples in the tutorial and lab sessions
  - Here we see **the most notable**!

Pumping Lemma: formal statement

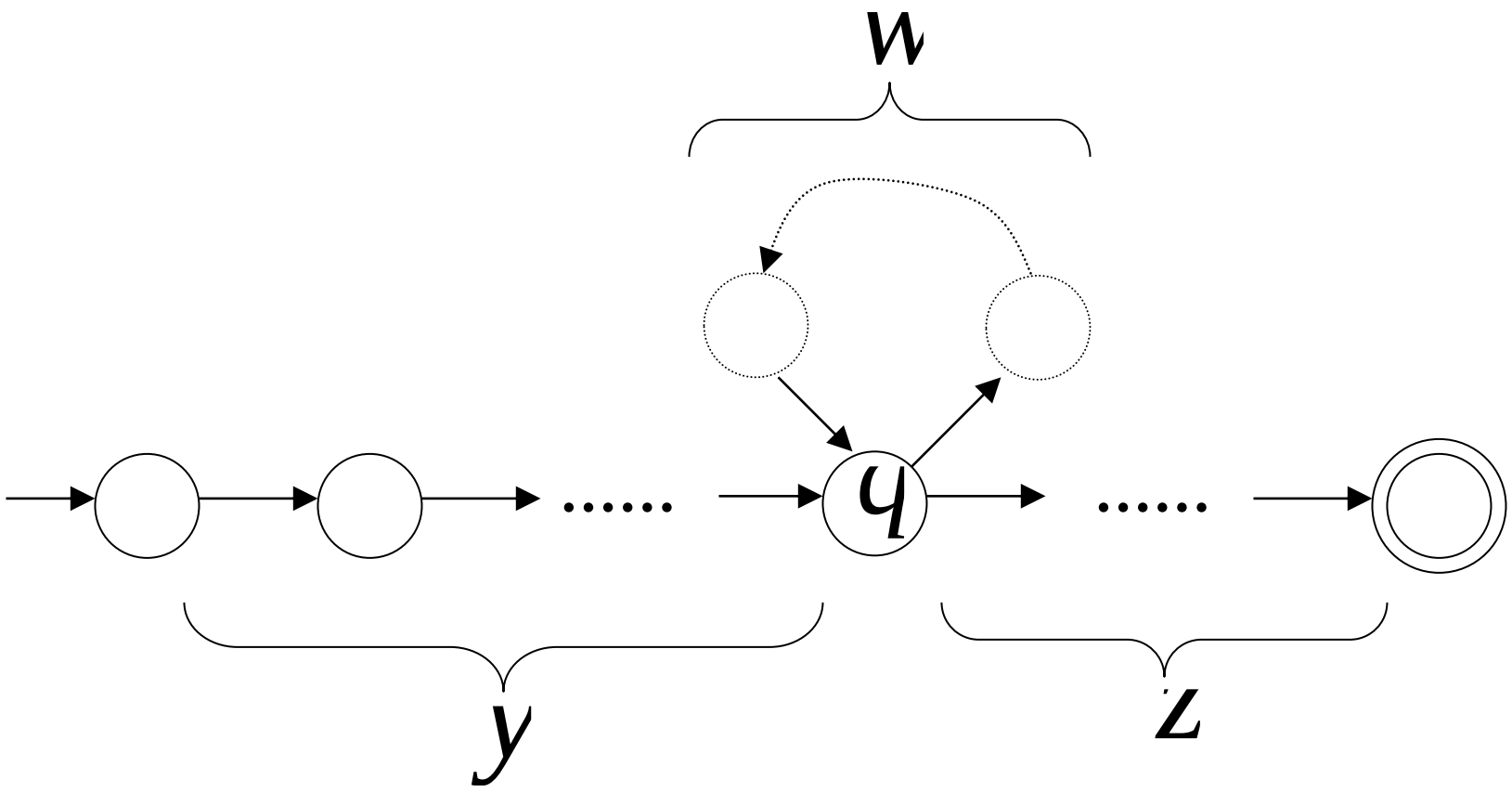
- Given a *regular language* **L**, If **x** $\in$ **L** and $|\mathbf{x}| \geq |\mathbf{Q}|$, then there exists a q $\in$ Q and a **w** $\in$ I^+ such that:
  - x = ywz
  - $\delta^*(q_0, y) = q$
  - $\delta^*(q, z) = q' \in F$
  - $\delta^*(q, w) = q$
  - $|yw| \leq |\mathbf{Q}|$
  - $yw^n z \in L, \forall n \geq 0$

For languages that are not regular these conditions does NOT have to apply. However, some at time may apply.

- This is the *Pumping Lemma* (one can "pump" **w**)

The pumping lemma is a necessary but not sufficient condition for a language to be regular

Pumping Lemma: graphical **proof sketch**



The “Pigeonhole” principle

- If *p* pigeons are placed into fewer than *p* holes, some holes must hold more than one pigeon

- The sequence of states traversed during the recognition of a string **must have a repeated state** (in our formal definition we called it **q**)

The three stages of comprehension

Intuition : the idea under discussion

Formalization: a rigorous mathematical statement

Examples: instances of the statement

A negative consequence of pumping lemma

- Is the language **L = {$a^n b^n$ | n > 0}** recognized by some FSA?
- *Let us suppose it is*, then:
- Consider x = $a^m b^m$, m > |Q| and let us apply the Pumping Lemma
- Possible cases:
  - x = ywz, w = a^k , k > 0 ====> $a^{m-k} a^{(k*r)} b^m \in L$, $\forall r$: **NO**
  - x = ywz, w = b^k , k > 0 ====> $a^m b^{(k*r)} b^{m-k} \in L$, $\forall r$: **NO**
  - x = ywz, w = $a^k b^s$, k, s > 0 ====> $a^{m-k} (a^k b^s)^r b^{m-s} \in L$ $\forall r$: **NO**

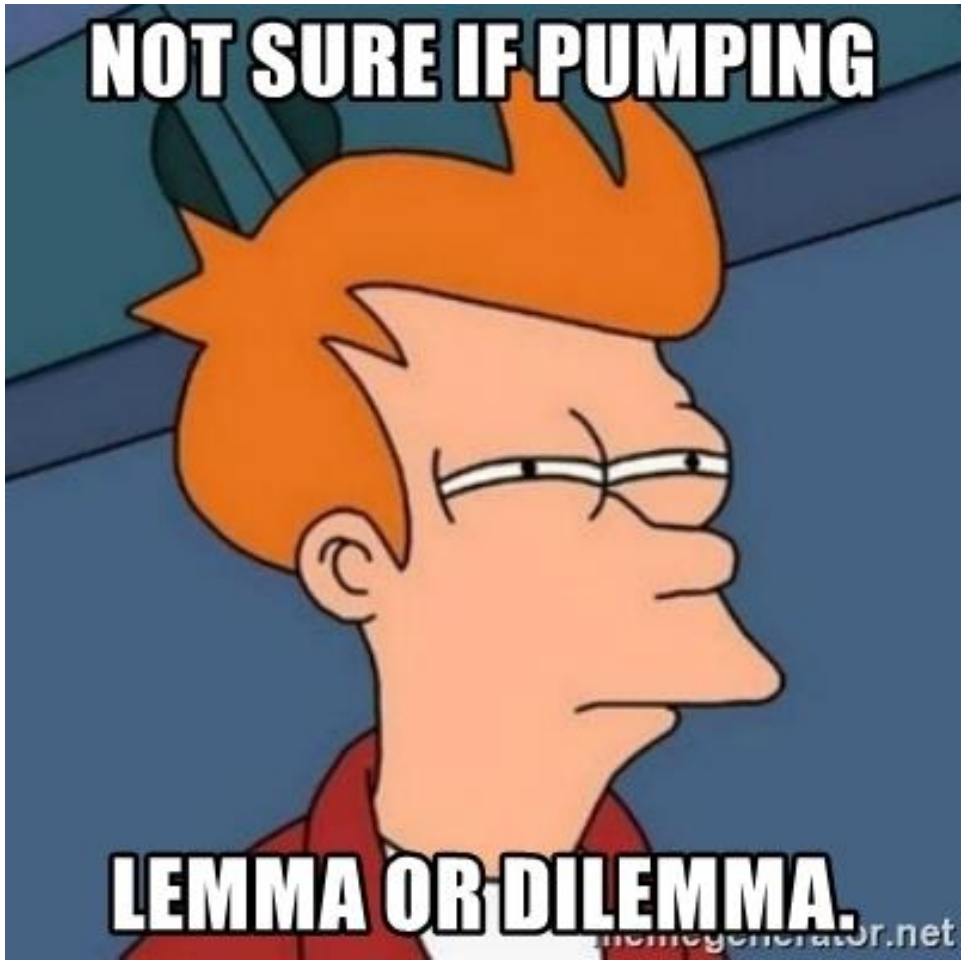
Levels of expressiveness

- In order to *"count"* an arbitrary *n* we need an **infinite memory**!

- **Fixed vs finite**
  - **FSA and regular languages are about fixed memory**

- From the toy example $\{a^n b^n\}$ to more concrete cases:
  - **Checking well-balancing of brackets** (typically used in programming languages) cannot be done with fixed memory

- We therefore need more powerful models (**PDA**)

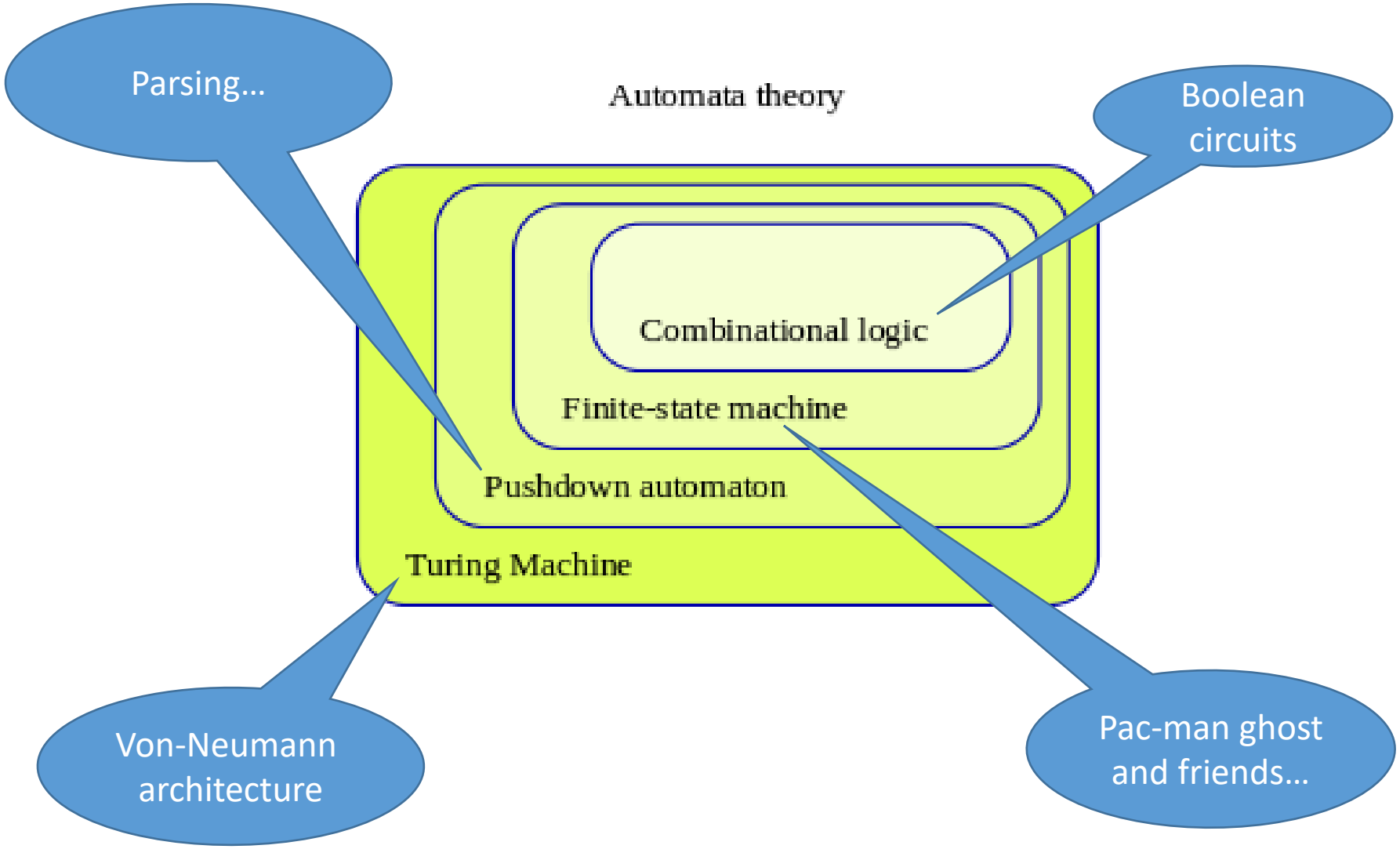

NOT SURE IF PUMPING
LEMMA OR DILEMMA.

Theoretical Computer Science

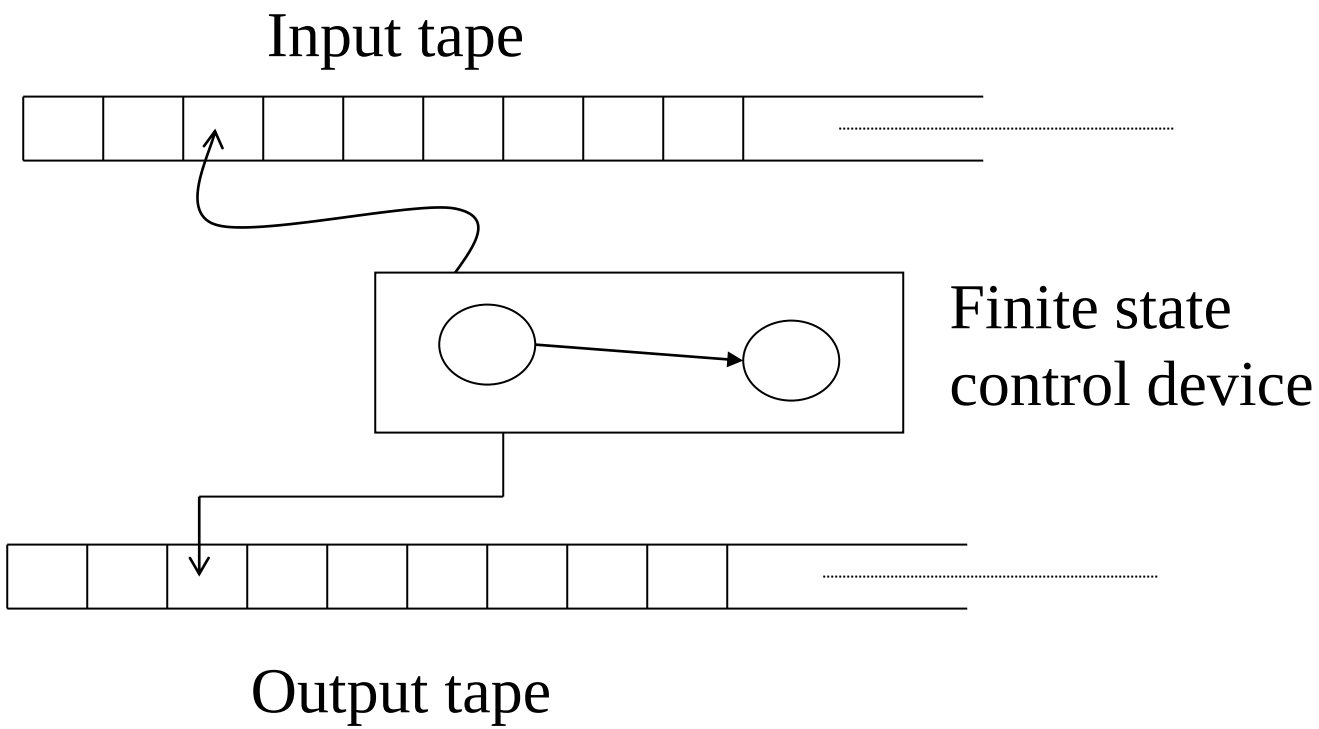
Introduction to Pushdown Automata

Lecture 5 - Manuel Mazzara

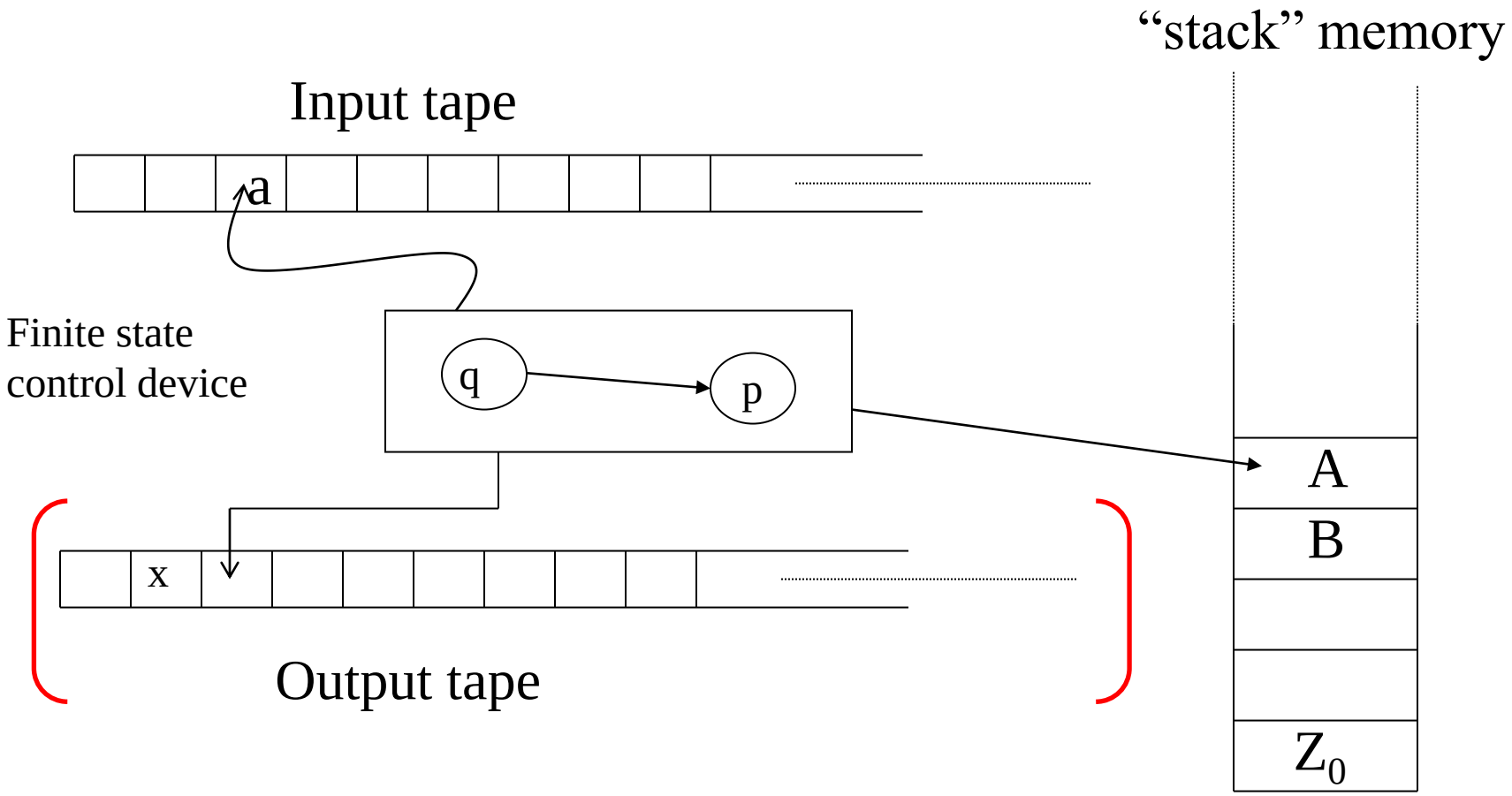
A bit of context



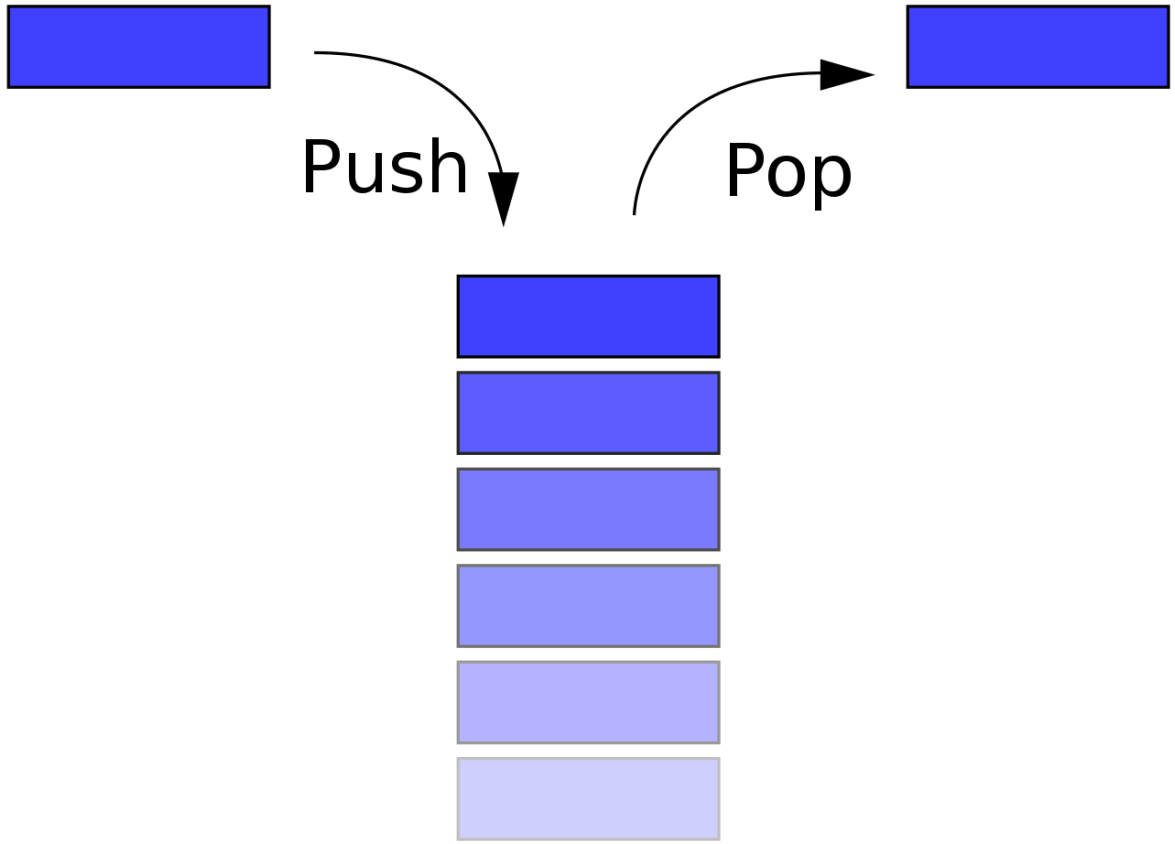
The mechanical view of FSAs

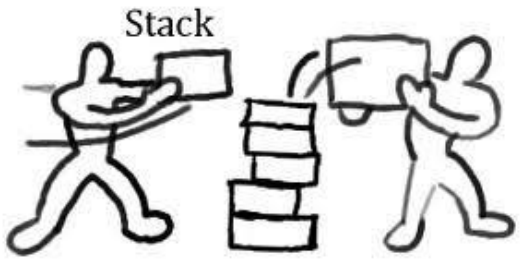


Adding a (destructive) external memory

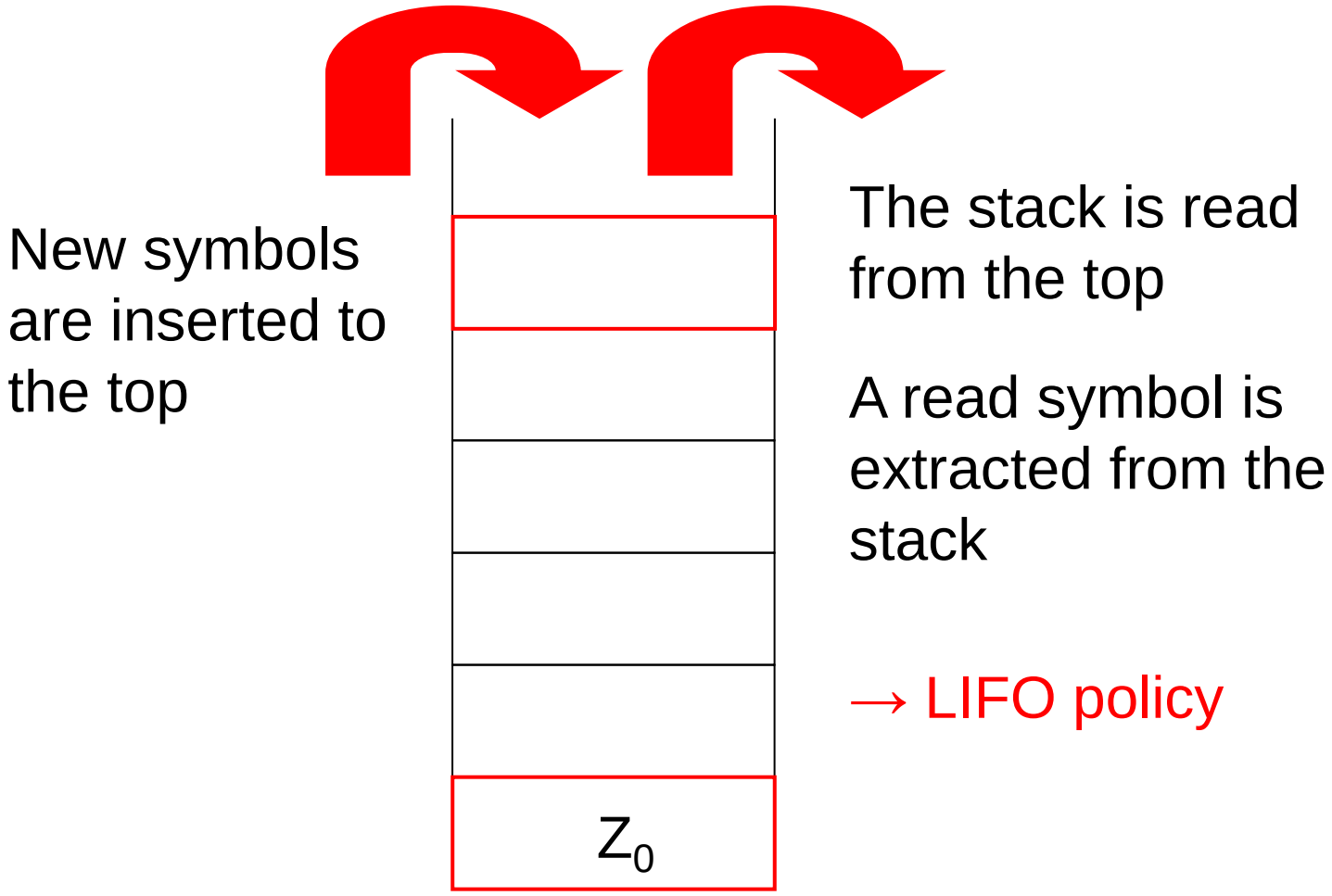


What is a stack?





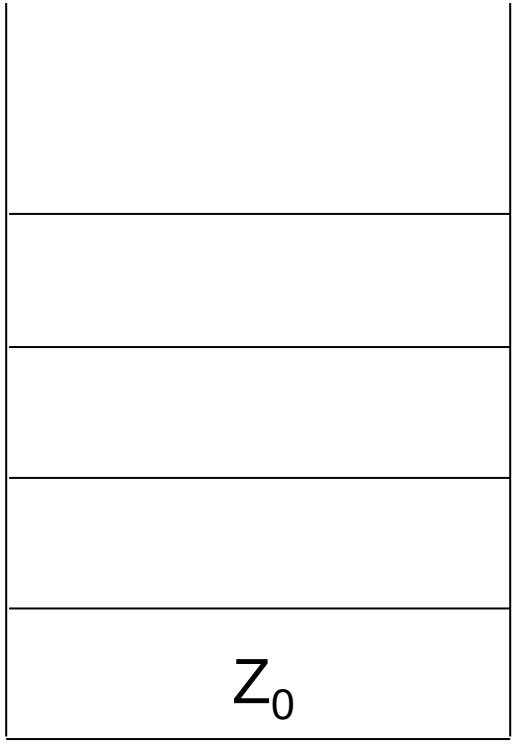
What is a stack (boring way)



Example

Insert in the following order the symbols

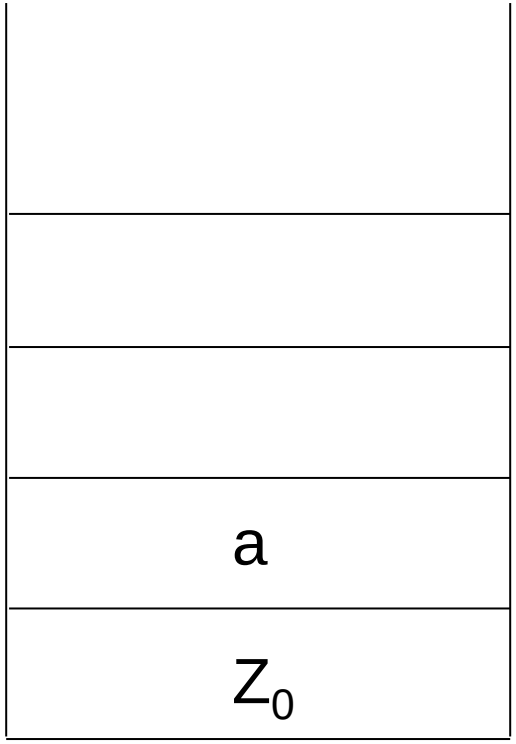
- a
- b
- c



Example

Insert in the following order the symbols

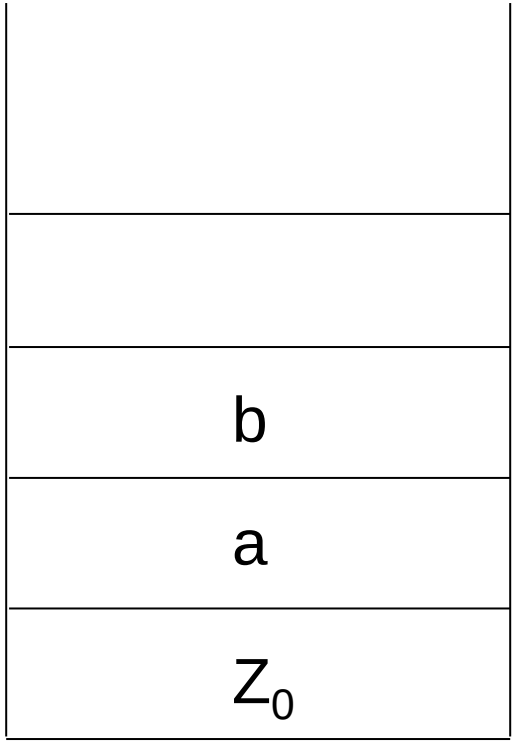
- a
- b
- c



Example

Insert in the following order the symbols

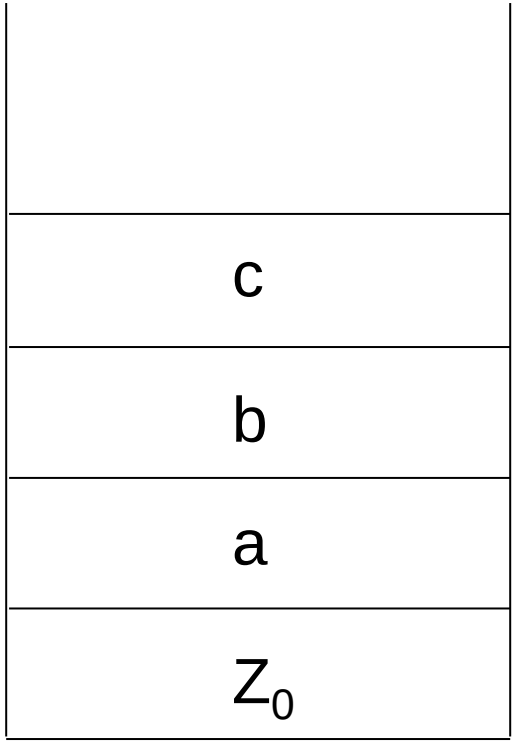
- a
- b
- c



Example

Insert in the following order the symbols

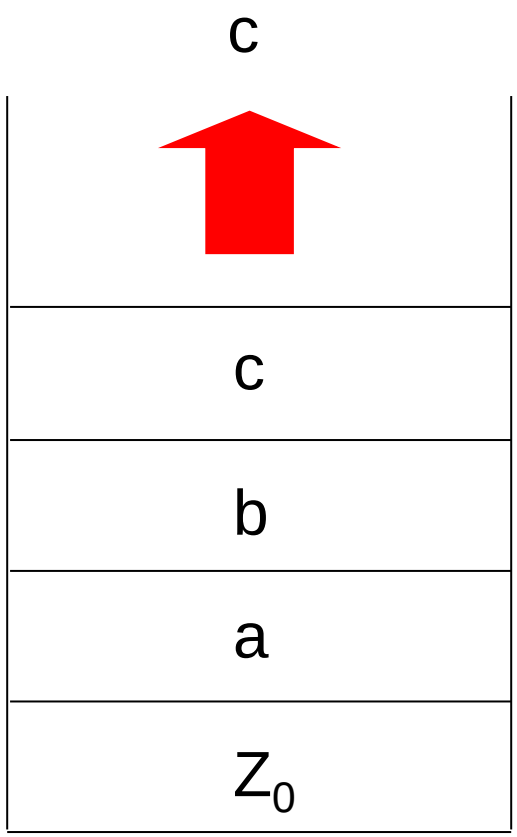
- a
- b
- c



Example

Insert in the following order the symbols

- a
- b
- c



Read from the stack

Trying to tell something new…

- Who invented the stack idea?
- **Alan Turing**'s 1946 paper on the Automatic Computing Engine (ACE), the first electronic computer developed in UK
- "*To arrange for the splitting up of operations into subsidiary operations*"
- Theory of **subroutines**
- Two instructions: BURY and UNBURY

Pushdown automata

- Finite state automata can be enriched with a stack

  → Pushdown Automata (PDA)

- PDAs differ from finite state machines in two ways:
  - They can **use the top of the stack to decide which transition has to be made**
  - They can **manipulate the stack** as part of performing a transition

Moves of a PDA

Depending on

- the symbol read from the input (but it could also read nothing)
- the symbol read from the top of the stack
- the state of the control device

the PDA

- *changes its state*
- *moves ahead the scanning head*
- *changes the symbol read from the stack with a string α (possibly empty)*

Of course, we will formally dissect this kind of automata and understand where they are useful