

# L10 – Turing Machines

## The Formal Model of Computation

Theory of Computation

## From Language Recognition to Computation

Up to this point in the course, our abstract machines have been designed to recognize languages. Finite automata and pushdown automata read an input string and decide whether it belongs to a certain language. While these models differ in expressive power, they share an important limitation: they are fundamentally recognizers. Turing machines represent a conceptual shift. They are not merely devices for recognizing patterns, but formal models intended to capture the very notion of algorithmic computation. With Turing machines, we move from classifying languages to defining what it means for a problem to be computable.

## Historical Motivation

In the early twentieth century, David Hilbert posed the Entscheidungsproblem, asking whether there exists a mechanical procedure that can decide the truth of any mathematical statement. To answer this question, mathematicians first needed a precise definition of what counts as a mechanical procedure.

Alan Turing approached this problem by analyzing how a human computes using paper and pencil. He abstracted this process into a simple mathematical machine that operates by following a fixed set of rules. The resulting model, now known as the Turing machine, provided a precise and rigorous definition of algorithm.

## Why Earlier Models Are Insufficient

Finite automata have no unbounded memory and therefore cannot perform tasks such as counting or comparison beyond a fixed bound. Pushdown automata extend this power by introducing a stack, enabling the recognition of nested structures. However, a stack enforces a strict last-in-first-out discipline and allows only one unbounded dependency to be tracked at a time.

There exist simple-looking problems, such as checking whether three different symbols occur the same number of times, that lie beyond the power of pushdown automata.

Turing machines were introduced to overcome these limitations by providing unrestricted access to memory.

## Informal Description of a Turing Machine

A Turing machine consists of a finite control, an infinite tape divided into cells, and a read–write head that scans the tape one cell at a time. Initially, the input string is written on the tape, and all remaining cells contain a special blank symbol.

At each step, the machine reads the symbol under the head, writes a symbol in the same cell, moves the head one cell to the left or to the right, and changes state. This process continues until the machine halts or runs forever.

## Why Writing on the Tape Matters

A crucial difference between Turing machines and earlier models is the ability to write on the tape. Finite automata and pushdown automata can only read their input; they cannot modify it. Turing machines, by contrast, can erase symbols, mark them, and reuse tape cells.

This ability effectively turns the tape into general-purpose memory. Using the tape, a Turing machine can simulate stacks, counters, arrays, and even other machines. This is the key reason why Turing machines can express general algorithms.

## Formal Definition

Formally, a Turing machine is a seven-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}).$$

Here,  $Q$  is a finite set of states,  $\Sigma$  is the input alphabet, and  $\Gamma$  is the tape alphabet, which includes the blank symbol. The transition function  $\delta$  specifies how the machine reads a symbol, writes a symbol, moves the head, and changes state. The machine starts in state  $q_0$  and halts only in the accepting or rejecting state.

## The Transition Function

The transition function has the form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

Given the current state and the symbol under the head, the function specifies exactly one action: a new state, a symbol to write, and a direction in which to move the head. This function constitutes the entire program of the Turing machine.

## Configurations

To reason precisely about Turing machines, we use the notion of a configuration. A configuration represents the complete instantaneous description of the machine, including the current state, the tape contents, and the position of the head.

Configurations are written in the form  $uqv$ , where  $u$  represents the portion of the tape to the left of the head,  $q$  is the current state, and  $v$  represents the symbol under the head together with the rest of the tape.

## Computations

A computation is a sequence of configurations, where each configuration follows from the previous one by applying the transition function. This formalizes the idea of executing an algorithm step by step.

Acceptance and rejection are defined in terms of reaching configurations that contain the accepting or rejecting state.

## Acceptance, Rejection, and Looping

A Turing machine accepts an input if it eventually halts in the accepting state and rejects it if it halts in the rejecting state. Importantly, the machine is also allowed to run forever on some inputs.

Nontermination is not a flaw of the model. It reflects the fact that some problems cannot be decided by any algorithm.

## Recognizers and Deciders

A Turing machine is said to recognize a language  $L$  if it accepts exactly the strings in  $L$  but may fail to halt on strings not in  $L$ . Such languages are called recursively enumerable.

If a Turing machine halts on every input and accepts exactly the strings in  $L$ , then it is said to decide  $L$ . Such languages are called decidable. This distinction is central to computability theory.

## A Language Beyond PDA Power

Consider the language  $\{a^n b^n c^n \mid n \geq 0\}$ . This language is not context-free and cannot be recognized by a pushdown automaton.

A Turing machine can recognize this language by repeatedly crossing off one  $a$ , one  $b$ , and one  $c$  in each iteration. The machine uses the tape to store intermediate markings and to move back and forth across the input.

## Why Turing Machines Are Strictly More Powerful

Every finite automaton and pushdown automaton can be simulated by a Turing machine. However, there exist languages, such as  $\{a^n b^n c^n\}$ , that can be decided by Turing machines but not by pushdown automata.

This establishes that Turing machines strictly extend the power of all previous automaton models.

## Robustness and the Church–Turing Thesis

Many variants of Turing machines exist, including multi-tape machines and nondeterministic machines. Despite superficial differences, all these models are equivalent in computational power.

This robustness supports the Church–Turing Thesis, which asserts that any function that can be computed by an effective mechanical procedure can be computed by a Turing machine. Although not a theorem, it is foundational to the theory of computation.

beamer []beamerthemeMadrid tikz

# Turing Machines

## Proofs, Examples, and Diagrams

Theory of Computation

## Theorem: Turing Machines Simulate Pushdown Automata

**Theorem.** Every language recognized by a pushdown automaton is also recognized by a Turing machine.

**Idea of Proof.** A Turing machine can simulate the stack of a PDA using its tape and finite control.

## Proof Sketch: TM Simulates PDA

Let  $P$  be a pushdown automaton.

We construct a Turing machine  $M$  such that:

$$L(M) = L(P)$$

The tape of  $M$  is divided conceptually into:

- ▶ an input region
- ▶ a stack region

Stack operations of  $P$  (push, pop, read top) are simulated by writing and erasing symbols in the stack region of the tape.

Each transition of  $P$  is simulated step-by-step by  $M$ .

## Conclusion of the Simulation Proof

Since every PDA transition can be simulated, the Turing machine recognizes exactly the same language.

Therefore:

$$\text{CFL} \subseteq \text{TM-recognizable}$$

This shows that Turing machines are *at least* as powerful as pushdown automata.

## Theorem: Turing Machines Are Strictly More Powerful

**Theorem.** There exists a language recognized by a Turing machine that cannot be recognized by any pushdown automaton.

**Proof Strategy.** Exhibit a specific language and compare capabilities.

## Proof via Language $\{a^n b^n c^n\}$

Consider:

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

Facts:

- ▶  $L$  is not context-free (proved using pumping lemma)
- ▶ Therefore, no PDA recognizes  $L$

However, a Turing machine *can* decide  $L$ .

Thus:

$$\text{PDA} \subsetneq \text{TM}$$

## Example 1: TM for $\{a^n b^n\}$

We describe a Turing machine that decides:

$$L = \{a^n b^n \mid n \geq 0\}$$

The machine repeatedly:

1. crosses off the leftmost unmarked  $a$
2. moves right to find and cross off a  $b$
3. returns to the beginning

If a matching  $b$  is missing, the machine rejects. If all symbols are crossed off, it accepts.

## Tape Evolution for $\{a^n b^n\}$

Initial tape:

aabb

After one iteration:

XaYb

After second iteration:

XXYY

All symbols matched  $\Rightarrow$  accept.

## Example 2: TM for $\{a^n b^n c^n\}$

We now extend the idea to:

$$L = \{a^n b^n c^n\}$$

Algorithm:

1. Mark the leftmost  $a$
2. Find and mark a  $b$
3. Find and mark a  $c$
4. Return to the start

If at any step a symbol is missing or out of order, the machine rejects.

## Why PDA Cannot Do This

The algorithm requires:

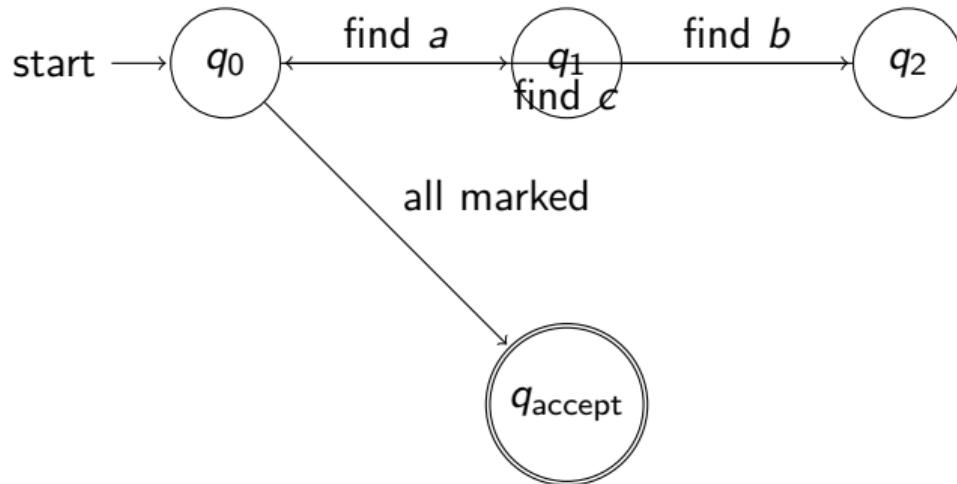
- ▶ tracking three quantities
- ▶ revisiting earlier input

A PDA has:

- ▶ only one stack
- ▶ no ability to move backward

Thus this task is impossible for PDAs but trivial for TMs.

## State Diagram: High-Level TM



This abstract diagram represents the logical phases of the TM for  $\{a^n b^n c^n\}$ .

## Reading the Diagram

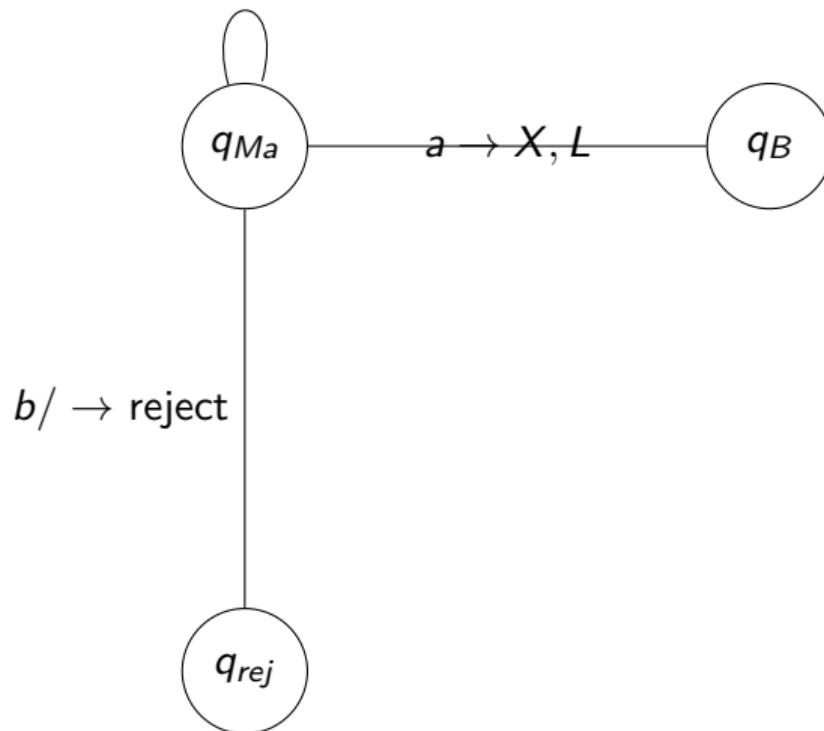
Each state corresponds to a phase of the algorithm, not a single tape action.

Edges represent scanning and marking procedures, each of which may take many tape steps.

This abstraction is typical in TM design:

- ▶ low-level details are suppressed
- ▶ correctness is argued at the algorithmic level

$$X/Y \rightarrow X/Y, R$$



# Turing Machine Construction Example

## Proofs, Examples, and Diagrams

We construct a single-tape Turing machine that decides

$$L = \{ w \in \{a, b\}^* \mid \#a(w) = \#b(w) \}.$$

That is, the machine accepts exactly those strings over  $\{a, b\}$  containing an equal number of  $a$ 's and  $b$ 's.

Tape alphabet:

$$\Gamma = \{a, b, X, Y, \}\text{.}$$

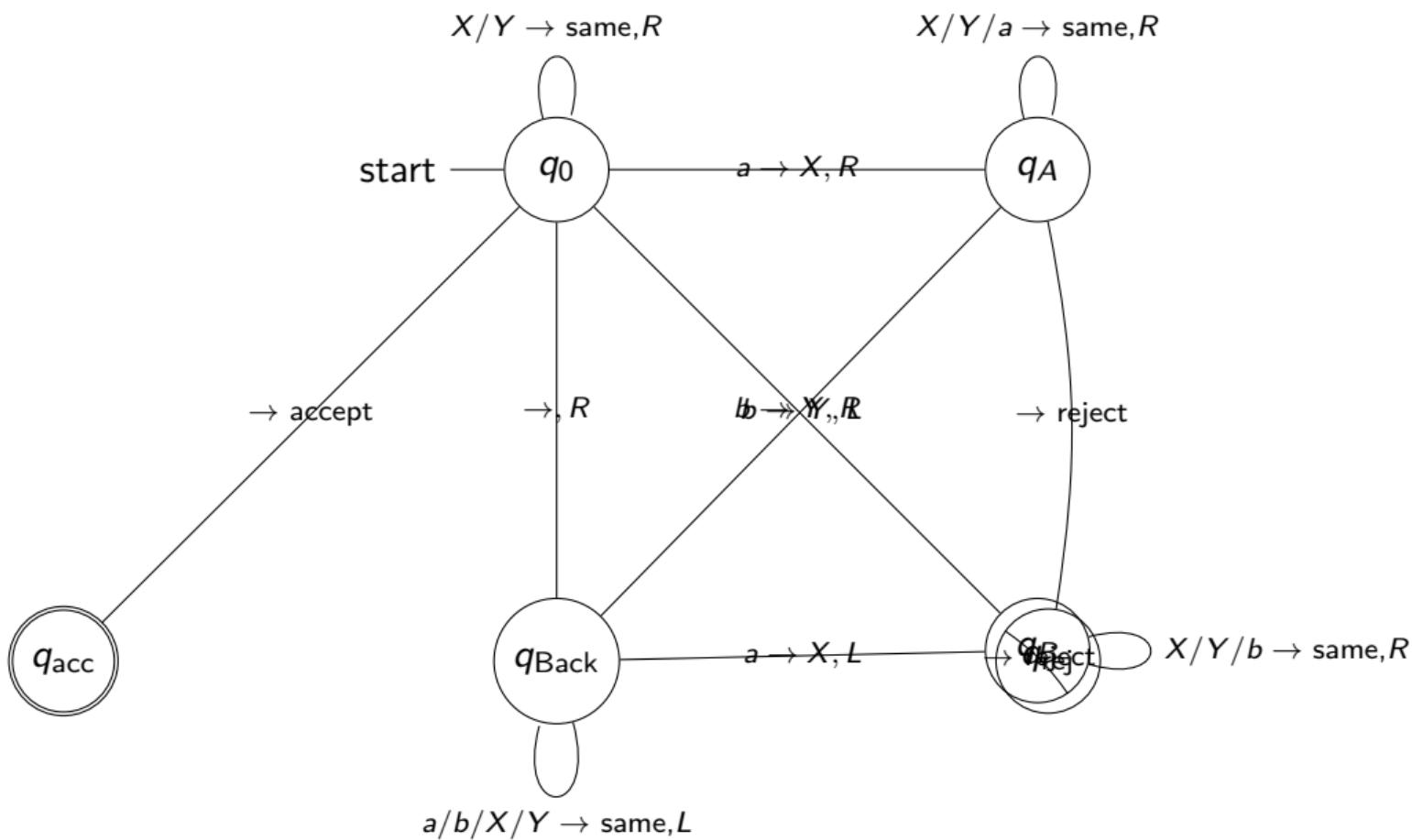
Markers:

$$X = \text{marked } a, \quad Y = \text{marked } b.$$

The machine repeatedly performs pair-cancellation: it finds the leftmost unmarked symbol. If it is  $a$ , it marks it  $X$  and then scans right to find an unmarked  $b$  to mark as  $Y$ . If it is  $b$ , it marks it  $Y$  and then scans right to find an unmarked  $a$  to mark as  $X$ . After marking a matching opposite symbol, it returns to the left end of the tape and repeats. If the machine ever fails to find a required opposite symbol before hitting the blank, it rejects. If eventually all symbols are marked and the head reaches blank while searching for a new unmarked symbol, it accepts.

Because each successful iteration marks at least two previously unmarked symbols, the machine must halt on all inputs, so it is a decider.

The diagram below is compressed: labels such as " $X/Y \rightarrow \text{same}, R$ " stand for multiple  $\delta$  transitions (one for each listed symbol). This keeps the graph readable while remaining faithful to the intended transition function.



We trace the machine on input abba. The head position is underlined.

$q_0 : \underline{a} b b a$

In  $q_0$  the first unmarked symbol is  $a$ . Mark it  $X$  and go to  $q_A$ .

$q_A : \underline{X} b b a$

In  $q_A$  scan right to the first unmarked  $b$ , mark it  $Y$ , then go left (to return).

$q_{\text{Back}} : X \underline{Y} b a$

Move left until reaching blank, then move right into  $q_0$ .

$q_0 : \underline{X} Y b a$

In  $q_0$  skip markers. First unmarked is  $b$ . Mark it  $Y$  and go to  $q_B$ .

$q_B : X Y \underline{Y} a$

In  $q_B$  scan right to find an unmarked  $a$ , mark it  $X$ , then return.

$q_{\text{Back}} : X Y Y \underline{X}$

Return to the left end again:

$q_0 : \underline{X} Y Y X$

Now  $q_0$  sees only markers and then blank, so it accepts.

Trace input aab. This should reject since there are more  $a$ 's than  $b$ 's.

$q_0 : \underline{a} a b$

Mark the leftmost  $a$ :

$q_A : \underline{X} a b$

In  $q_A$  scan right and find  $b$ , mark it  $Y$ , return:

$q_{\text{Back}} : X a \underline{Y}$

Return to start:

$q_0 : \underline{X} a Y$

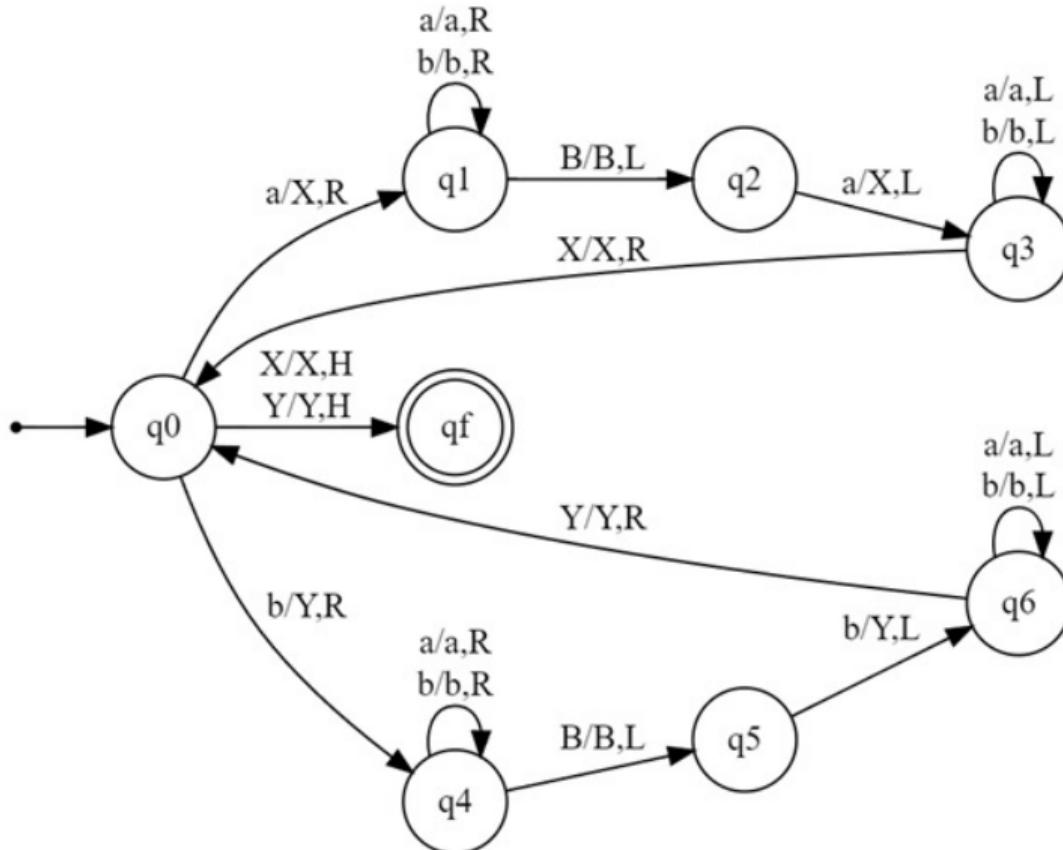
In  $q_0$ , the next unmarked symbol is  $a$ . Mark it  $X$  and go to  $q_A$ :

$q_A : X \underline{X} Y$

Now  $q_A$  scans right looking for an unmarked  $b$ , but reaches blank without finding one, so the machine rejects.

**Conclusion:** The machine decides  $L$  by repeatedly canceling one  $a$  with one  $b$ .

# Turing Machine Example



The following figure shows a Turing machine  $M$  (the diagram is provided separately).

1. What problem does this Turing machine solve?
2. Write the language  $L(M)$  accepted by this machine.

**Idea.** The machine decides whether the input string over  $\{a, b\}$  is a palindrome by repeatedly matching the leftmost unmarked symbol with the rightmost unmarked symbol.

- ▶ In state  $q_0$ , it finds the leftmost unmarked symbol.
  - ▶ If it sees  $a$ , it marks it as  $X$  and moves right (transition  $a/X, R$ ) to  $q_1$ .
  - ▶ If it sees  $b$ , it marks it as  $Y$  and moves right (transition  $b/Y, R$ ) to  $q_4$ .
- ▶ In  $q_1$  (respectively  $q_4$ ), it scans right over  $as$  and  $bs$  until it reaches the blank  $B$ , then moves left (transition  $B/B, L$ ) to  $q_2$  (respectively  $q_5$ ).
- ▶ In  $q_2$ , it must find an  $a$  at the right end to match the leftmost marked  $X$ : it changes that  $a$  to  $X$  and moves left to  $q_3$  (transition  $a/X, L$ ). Similarly, in  $q_5$ , it must find a  $b$  at the right end: it changes that  $b$  to  $Y$  and moves left to  $q_6$  (transition  $b/Y, L$ ).
- ▶ In  $q_3$  and  $q_6$ , it moves left over  $as$  and  $bs$  until it finds the left marker ( $X$  or  $Y$ ), then goes right back to  $q_0$  (transition  $X/X, R$  from  $q_3$  or  $Y/Y, R$  from  $q_6$ ) to repeat the process.

- If in  $q_0$  the machine sees only markers  $X$  and  $Y$  (i.e., the input is fully matched), it halts and accepts in  $q_f$  (transitions  $X/X, H$  and  $Y/Y, H$ ).

If at any matching step the required symbol is not found (e.g.,  $q_2$  expects  $a$  but sees  $b$  or  $B$ ), no transition applies and the machine rejects.

**Therefore,**

$$L(M) = \{ w \in \{a, b\}^* \mid w = w^R \},$$

i.e., the set of all palindromes over  $\{a, b\}$  (including  $\varepsilon$ ).