Universal Turing Machine

Gita Ghasemi
Erfan Mahdavi
Erfan Abbasi

What is a Turing Machine?

- A Turing Machine (TM) is a simple model of a computer
- It was invented to study what computers can and cannot do
- Basic idea: A machine that reads and writes on an infinite tape, following rules

History: Why and When Was It Invented?

- Invented in 1936 by Alan Turing
- Why? To solve a big math problem: Can we decide if a statement is true or false using rules?
- This came after Gödel's work in 1931 showing math has limits
- Turing wanted to define "computation" clearly

Components of a Turing Machine

- Tape: Infinite memory storage
- Head: Reads and writes symbols on the tape
- States (Q): Machine's internal condition
- Alphabet (Σ, Γ): Symbols allowed on the tape
- Transition Rules (δ): How the machine moves and changes symbols

How It Works (Simple View)

1- Read the symbol under the head

2- Check the current state

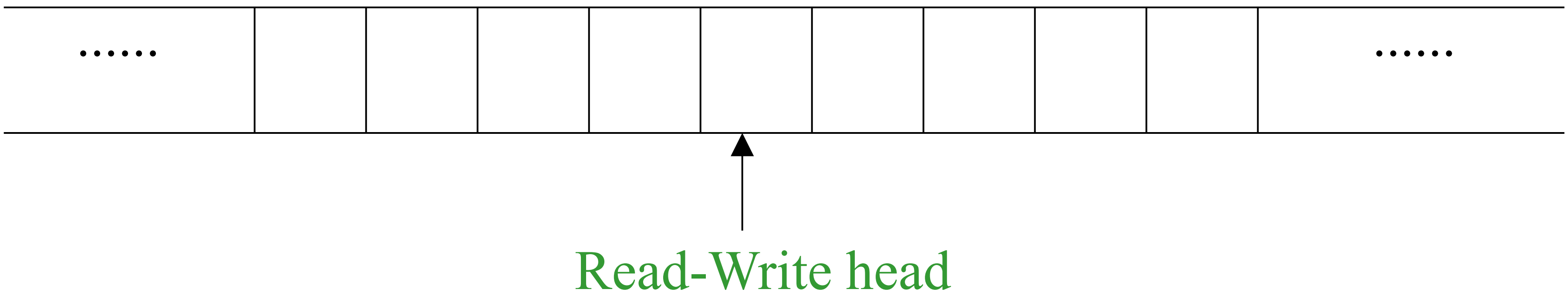
3- Change the symbol if needed

4- Move the head left or right

5- Change state

Repeat until machine halts

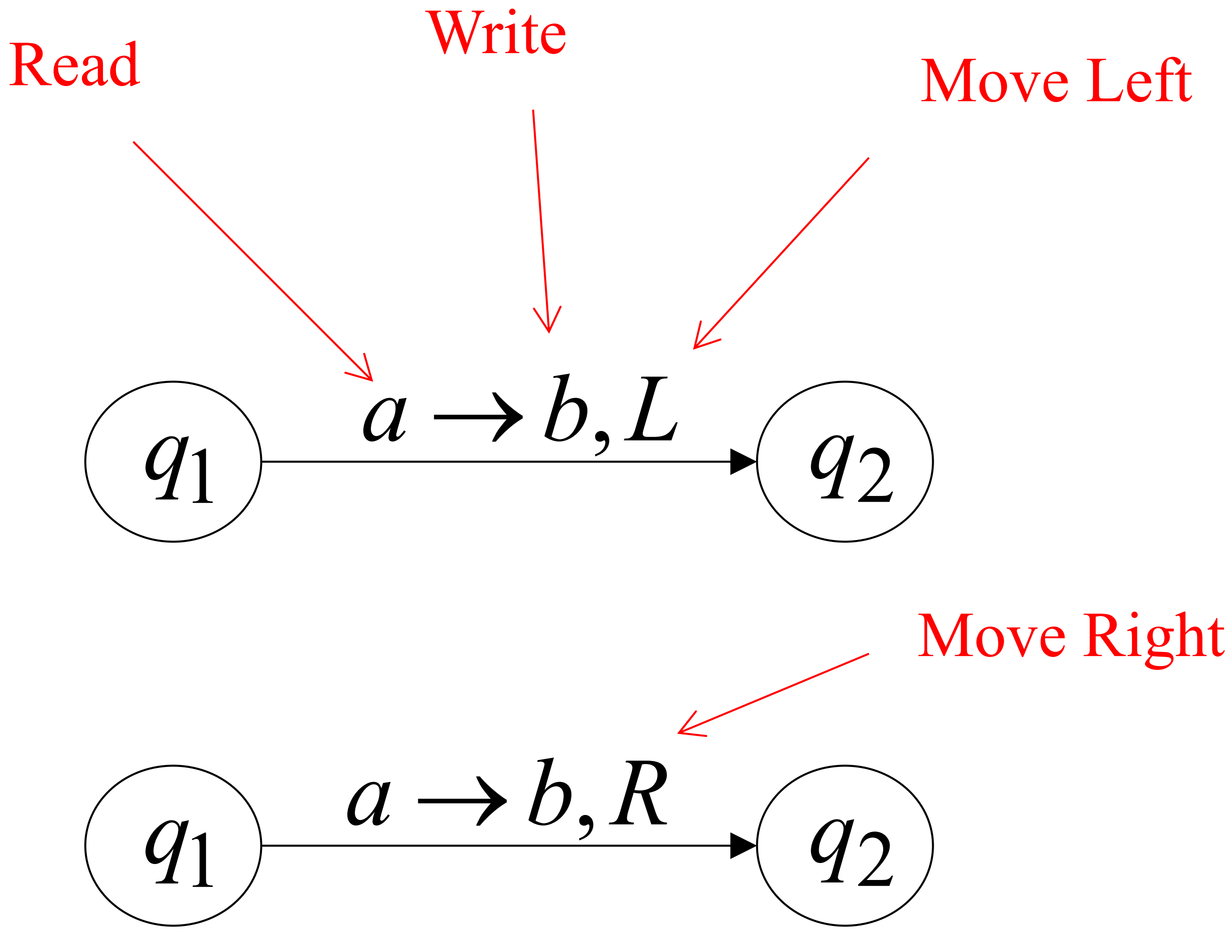
Tape and Head



The head at each time step:

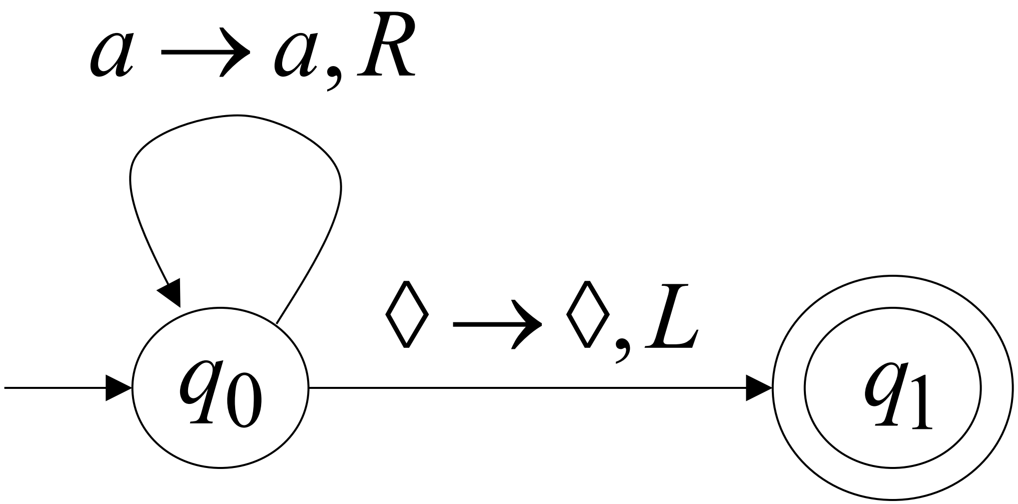
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

States and Transitions

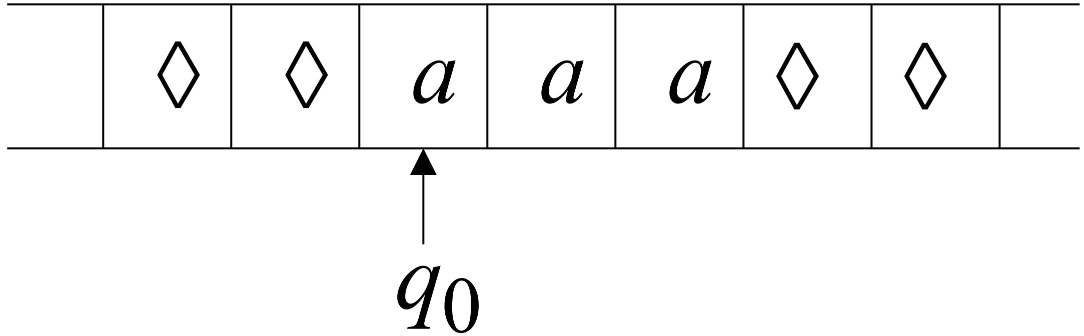


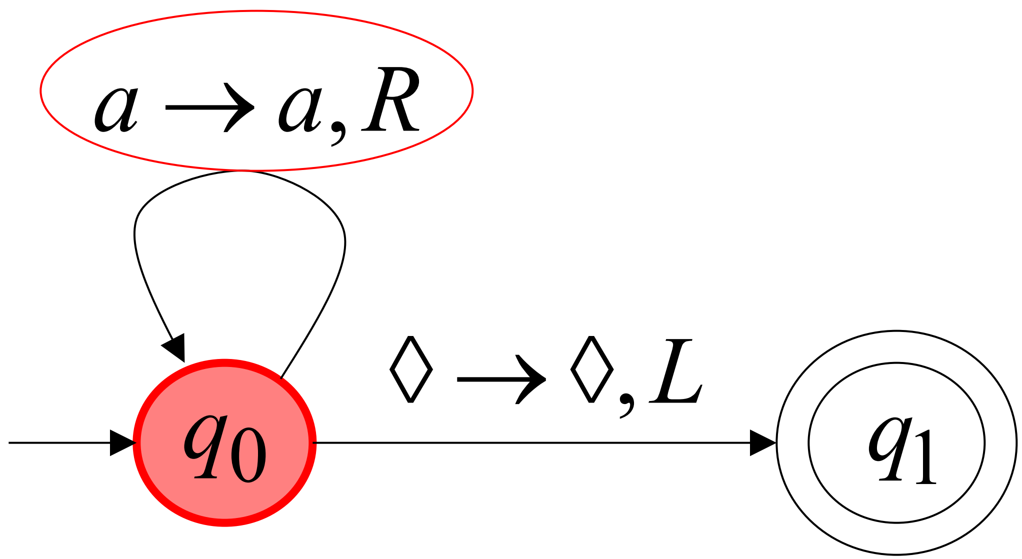
Example

Turing machine for the language aa^*

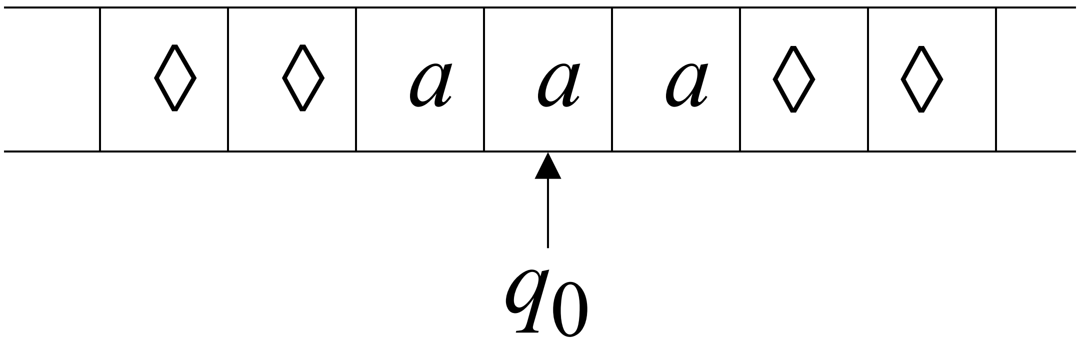


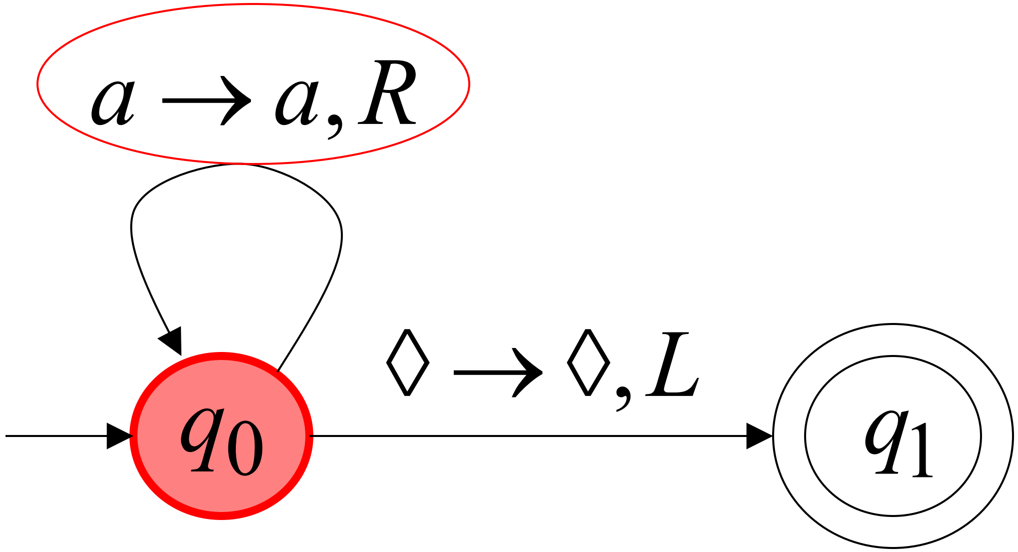
Time 0


$\Diamond$ $\Diamond$ a a a $\Diamond$ $\Diamond$
q_0

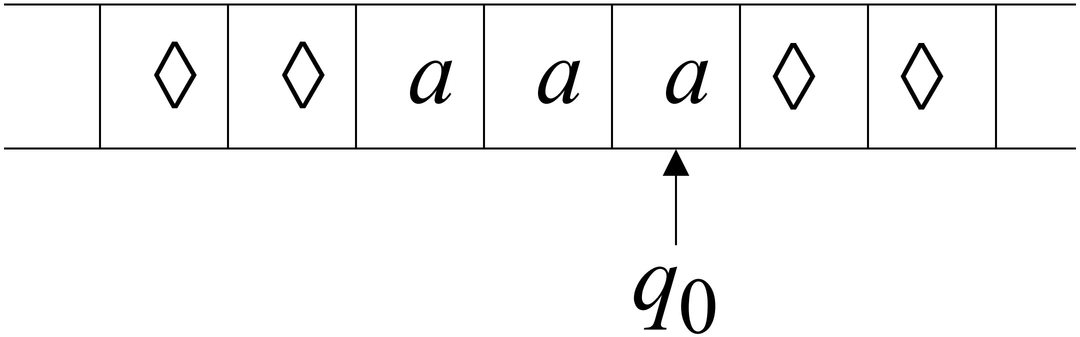

$a \rightarrow a, R$
$\Diamond \rightarrow \Diamond, L$
q_0
q_1

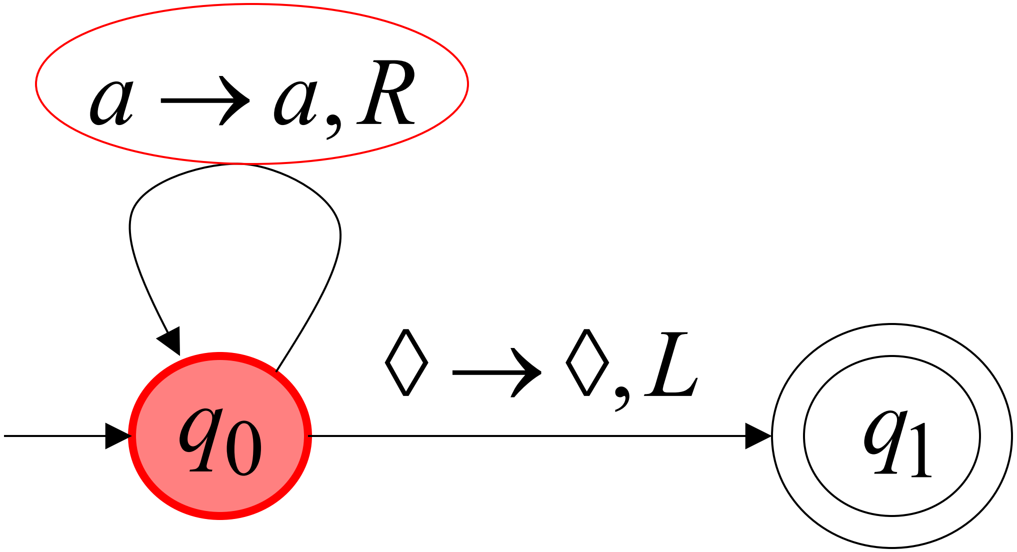
Time 1



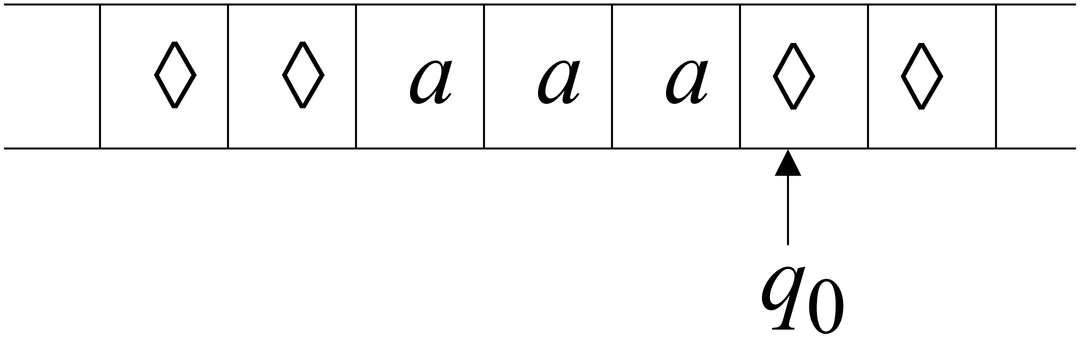


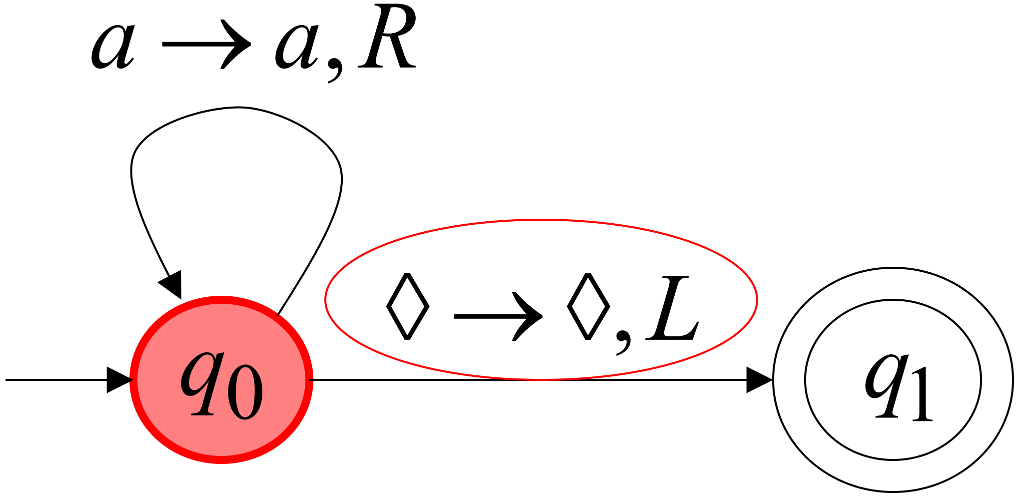
Time 2



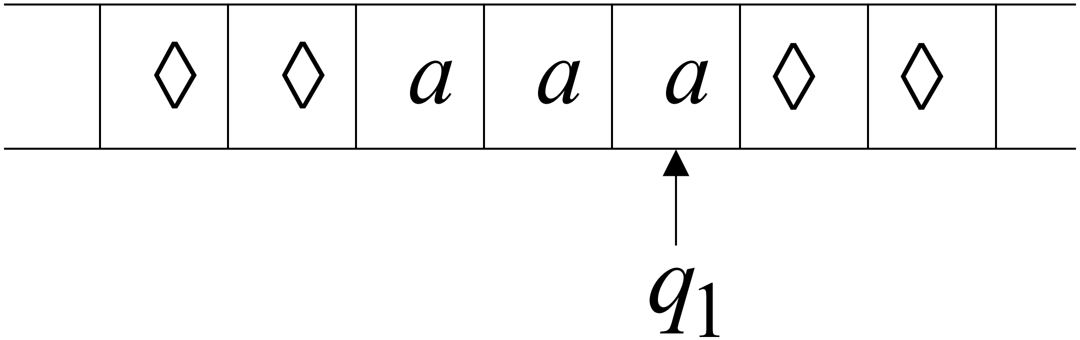


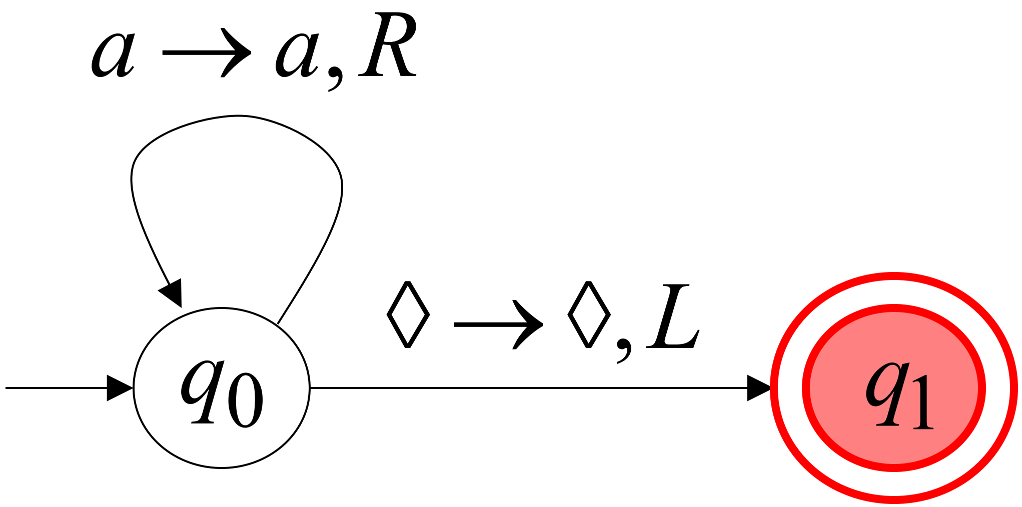
Time 3





Time 4





The Church–Turing Thesis

Idea: Any problem that can be solved by an algorithm can be done by a TM.

This means TMs are as powerful as any computer

Bridge to multi-task: But a basic TM does only one job. What if we want it to do many tasks?

Bridge to Multi-Task Machines

- Basic TM is for one task (e.g., just adding).
- To do more: Combine TMs like building blocks
- This leads to modular TMs: Use sub-machines for different parts

Suggestion: Use Modular Turing Machines

- Modular means: Break big task into small TMs.
- Example: Main TM calls "sub-TMs" like functions in code
- Benefits: Reuse parts, easier to fix errors, build step by step
- Like Lego: Connect small pieces for big things

Example: Check Condition and Add

- Task: If x >= y, add them; else, subtract y from x
- Modular steps
  - Sub-TM 1: Compare x and y (check lengths on tape)
  - If true: Call add TM (like Slide 6)
  - If false: Call subtract TM (erase y symbols from x)
- Example: x=3 (111), y=2 (11). Compare says yes, add to 11111 (5)

Problems with Modularity

- Too many tasks? Need to define many sub-TMs, which is time-consuming
- Changing tasks requires rebuilding the whole machine
- Tape management: Sub-TMs might mess up the tape for others
- Not flexible for new jobs without redesign

Need: A better tool that can handle any task without changes

Introduction to Universal Turing Machine (UTM)

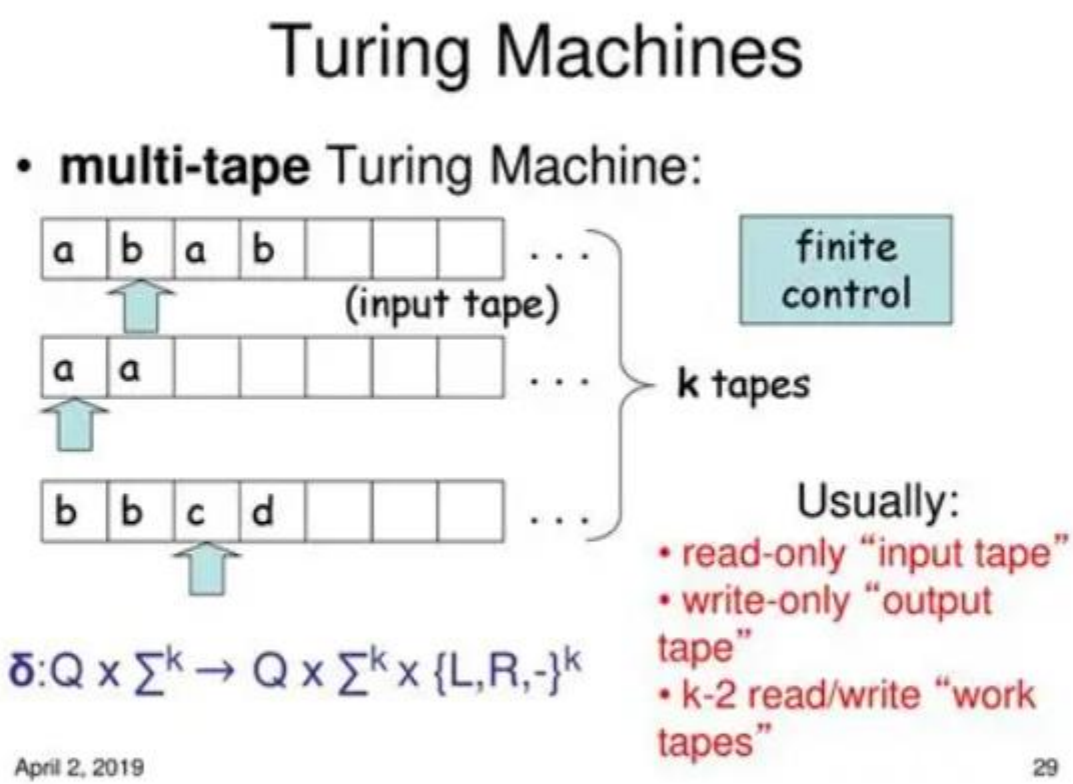
- UTM fixes modularity problems
- It's a "general-purpose" TM: Can simulate any other TM
- How? Give it a description (code) of another TM + input
- Resolves: No need to rebuild; just change the input code
- Like a modern computer running any program

How UTM Works and Its Components

- Works: Reads the code of the target TM, simulates each step
- Components (similar to TM, but special):
  - Multiple tapes: One for TM code, one for input, one for work
  - δ: Rules to decode and copy states/symbols
  - Simulates: Step by step, like running software
- Key: Encoding – Turn any TM into a string of 0s and 1s

Turing Machine Variants:

- turing machine with (left,right) + stay
- left resert TM : (left,right) + reset to left cell
- muti tape TM :



Problems:

1- only have 1 tape head

2- what if one of the tapes allocates a new cell

3- what if one of the tapes moves L of it's "LEFT END"

Universal Turing machine

- A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w, can simulate the computation of M on w.

Universal Turing machine

- The language

  A_{TM} = {<M, w> | M is a Turing Machine and accepts w}

  is Turing Recognizable (Not decidable).

Standard Way of Describing TMs

- Assume that

  $Q = \{q1, q2, ..., qn\}$,

  with q1 the initial state, q2 the single final state,
- And

  $\Gamma = \{a1, a2, ..., am\}$, where a1 represents the blank.

Standard Way of Describing TMs

- We then select an encoding in which q1 is represented by 1, q2 is represented by 11, and so on.

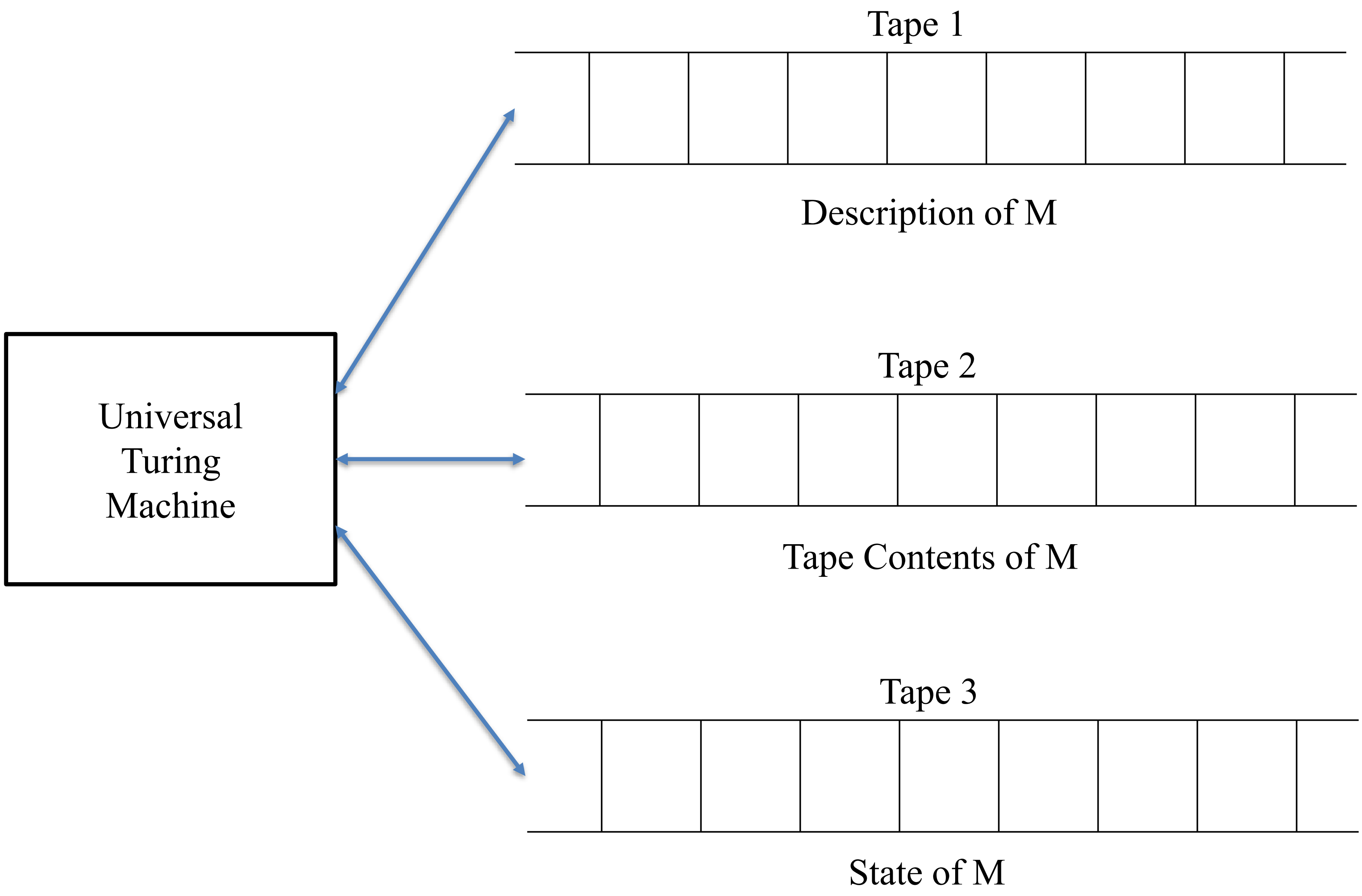
- Similarly, a1 is encoded as 1, a2 as 11, etc. The symbol 0 will be used as a separator between the 1’s.

Standard Way of Describing TMs

- It follows from this that any Turing machine has a finite encoding as a string on $\{0, 1\}^+$ and that, given any encoding of M, we can decode it uniquely.

Standard Way of Describing TMs

- A universal Turing machine M_u then has an input alphabet that includes {0, 1} and the structure of a multitape machine as shown in the next slide:


Tape 1
Description of M
Universal
Turing
Machine
Tape 2
Tape Contents of M
Tape 3
State of M

Tapes in UTM

- For any input M and w:
- Tape 1 will keep an encoded definition of M.
- Tape 2 will contain the tape contents of M,
- Tape 3 the internal state of M.

Tapes in UTM

- M_u looks first at the contents of tapes 2 and 3 to determine the configuration of M. It then consults tape 1 to see what M would do in this configuration. Finally, tapes 2 and 3 will be modified to reflect the result of the move.

Enumeration Procedure.

- We can prove that a set is countable if we can produce a method by which its elements can be written in some sequence. We call such a method an enumeration procedure.

- Enumeration procedure: روال بر شمارش

DEFINITION 10.4

Let S be a set of strings on some alphabet Σ. Then an enumeration procedure for S is a Turing machine that can carry out the sequence of

steps

$$q_0\square \overset{*}{\vdash} q_s x_1 \# s_1 \overset{*}{\vdash} q_s x_2 \# s_2 \ldots,$$

with $x_i \in \Gamma^* - \{\#\}$, $s_i \in S$, in such a way that any s in S is produced in a finite number of steps. The state q_s is a state signifying membership in S; that is, whenever q_s is entered, the string following # must be in S.

Proper Order

- We can use a modified order, in which we take the length of the string as the first criterion, followed by an alphabetic ordering of all equal-length strings. This is an enumeration procedure.
- We call this ordering, the **proper order**.

Theorem 10.3

- The set of all Turing machines, although infinite, is countable

Proof of Theorem 10.3

- We can encode each Turing machine using 0 and 1.
- With this encoding, we then construct the following enumeration procedure:

Proof of Theorem 10.3

- 1. Generate the next string in $\{0, 1\}^+$ in proper order.
- 2. Check the generated string to see if it defines a Turing machine. If so, write it on the tape in the form required by Definition 10.4. If not, ignore the string.
- 3. Return to Step 1.

Proof of Theorem 10.3

- Since every Turing machine has a finite description, any specific machine will eventually be generated by this process.

Machine M_inc : unary increment : w -> w1

Q = {q1, q2} (q1=start, q_2=halt)
Tape alphabet = {a1, a2} (a1= blank, a2= 1)
Transitions:

- rule A: T(q1, a2) -> (q1, a2, R)   means: if 1, scan right
- rule B: T(q1, a1) -> (q2, a2, L) means: if blank, write 1 and halt

Encodings: q1 -> 1   q2 -> 11   a1 -> 1   a2-> 11   R->1   L-> 11

rule A: 1 0 11 0 1 0 11 0 1   rule B: 1 0 1 0 11 0 11 0 11

Full <M_inc> = 10110101101 **00** 101011011011

Consequence 1: Countability of Turing Machines

Definition 10.4 (Enumeration Procedure):

- A TM E is an enumerator for a set S if E starts on a blank tape and, over time, prints out every string s in S, separated by a # symbol.
- E must produce every s in S in a finite number of steps
- A set is **countable** if an enumeration procedure exists for it.

Theorem 1**0.3:** The set of all Turing Machines, S_{TM}, is **countable**

- We can generate all possible binary strings in "proper order“

Like : 0 1 00 01 10 11 000

We can build an "Enumerator" TM E that does the following forever:

Generate the next string s in proper order.

Check if s is a valid encoding of a TM ('000' is invalid).

If s is a valid <M>, print it to the output tape.

Consequence 2: The Halting Problem

The Problem: Can we create a TM H that decides if any machine M will halt on any input w?
Input to H: <M,w> (the encoding of M and its input)
Output of H (must always halt):
Accept -> if M halts on w.
Reject -> if M loops forever on w.

Proof by contradiction: we use H to build a machine D
D logic:
- if H accepts(halts) : D loops
- if H rejects(loops) : D halts

What happens if we run D on <D>. SO H is asked about (<D>,<D>)
2 CASES ARE POSSIBLE: either D loops or halts on <D>

What the UTM proves: The UTM is the formal construction of a programmable, general-purpose computer. It proves that one fixed machine can run any computable program. This is the CPU + RAM model.
The Church-Turing Thesis mentioned, this is the broader philosophical claim:"Any function that is effectively calculable (by any intuitive, mechanical method) can be computed by a Turing Machine."The UTM gives this thesis incredible strength: one single TM can do it all.Key Takeaways:Programs are Data: We can encode any TM M as a string <M>.Universality: A single UTM U can simulate any M on any w.
Limits: Because we can encode TMs, we can prove they are countable, and more importantly, that fundamental problems like the Halting Problem are undecidable.