

Turing Machines

None of the automata which we have seen so far are general models of computation. *Turing machines* provide such a general model.

Turing machines extend the idea of linear bounded automata by having an infinite length of tape. The symbol $<$ is used to mark the leftmost bound of the tape beyond which the tape head cannot move and which cannot be overwritten. There is no rightmost bound on the tape.

As for linear bounded automata, Turing machines will perform one of the actions $\mathcal{A} \in \{Y, N, L, R\}$ at each step, where:

- Y denotes “Yes”, accept the input string
- N denotes “No”, do not accept the input string
- L denotes “Left”, move the read-write head one space to the left
- R denotes “Right”, move the read-write head one space to the right

Formal Definition of Turing Machines

A Turing machine is a 5-tuple $M = (Q, \Sigma, \Gamma, q_0, \delta)$, where:

- Q is a finite set of states.
- Σ is an alphabet (*input symbols*).
- Γ is an alphabet (*store symbols*).
- $q_0 \in Q$ is the *initial state*.
- δ , the *transition function*, is from $Q \times (\Gamma \cup \{<\})$ to $Q \times (\Gamma \cup \{<\}) \times \mathcal{A}$.

If $((q, a), (q', b, action)) \in \delta$, then when in state q with a at the current read position on the tape, M may replace a with b on the tape, perform the specified *action*, and enter state q' .

The symbol “#” is used to denote a blank tape square.

M accepts $w \in \Sigma^*$ iff it starts with configuration $(q_0, w\underline{\#})$ and the action Y is taken.

Turing Machine Example

$M_1 = (Q, \Sigma, \Gamma, q_0, \delta)$ where:

- $Q = \{s_0, s_1, s_2, s_3, s_4\}$
- $\Sigma = \{a\}$
- $\Gamma = \{a, x, \#\}$
- $q_0 = s_0$
- $((s_0, a), (s_1, \#, R))$
- $((s_0, x), (s_0, x, N))$
- $((s_0, \#), (s_0, \#, N))$
- $((s_1, a), (s_2, x, R))$
- $((s_1, x), (s_1, x, R))$
- $((s_1, \#), (s_1, \#, Y))$
- $((s_2, a), (s_3, a, R))$
- $\delta = ((s_2, x), (s_2, x, R))$
- $((s_2, \#), (s_4, \#, L))$
- $((s_3, a), (s_2, x, R))$
- $((s_3, x), (s_3, x, R))$
- $((s_3, \#), (s_3, \#, N))$
- $((s_4, a), (s_4, a, L))$
- $((s_4, x), (s_4, x, L))$
- $((s_4, \#), (s_1, \#, R))$

This will accept all strings of a 's whose length is a power of 2.

Another Turing Machine Example

$M_2 = (Q, \Sigma, \Gamma, q_0, \delta)$ where:

- $Q = \{s_0\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, b, \#\}$
- $q_0 = s_0$
- $((s_0, a), (s_0, a, L))$
- $((s_0, b), (s_0, b, R))$
- $\delta = ((s_0, \#), (s_0, \#, Y))$
- $((s_0, <), (s_0, <, R))$

This is an example of a Turing machine which may not halt.

Notation

Configuration: $(s, aab\underline{a}a)$

Initial configuration: $(q_0, \underline{w}\#)$

Halted configuration: Configuration when action Y or N is performed.

Hanging configuration: No transition for the current state and current input symbol.

A *computation* is a sequence of configurations for some $n \geq 0$. Such a computation is of length n .

$$\begin{array}{l} (s_0, \underline{a}a\#) \vdash_{M_1} (s_1, \# \underline{a}\#) \\ \vdash_{M_1} (s_2, \# \underline{x}\#) \\ \vdash_{M_1} (s_4, \# \underline{x}\#) \\ \vdash_{M_1} (s_4, \# \underline{x}\#) \\ \vdash_{M_1} (s_1, \# \underline{x}\#) \\ \vdash_{M_1} (s_1, \# \underline{x}\#) \end{array}$$

The final configuration is a halting configuration.

Therefore $(s_0, \underline{a}a\#) \vdash_{M_1}^* (s_1, \# \underline{x}\#)$

Computations

M is said to *halt* on input w iff (q_0, \underline{w}) yields some halted configuration.

M is said to *hang* on input w if (q_0, \underline{w}) yields some hanging configuration.

Turing machines compute functions from strings to strings.

Formally: Let f be a function from Σ_0^* to Σ_1^* . Turing machine M is said to compute f if for any $w \in \Sigma_0^*$, if $f(w) = u$ then:

$$(q_0, \underline{w}\#) \vdash_M^* (q, u\#)$$

where $(q, u\#)$ is a halted configuration.

f is said to be a *Turing-computable* function.

Multiple parameters: $f(w_1, \dots, w_k) = u$:

$$(q_0, \underline{w_1}\#w_2\#\dots\#w_k\#) \vdash_M^* (q, u\#).$$

where $(q, u\#)$ is a halted configuration.

Functions on Natural Numbers

Represent numbers in unary notation using only the symbol 1 (zero is represented by the empty string).

$f : \mathbb{N} \rightarrow \mathbb{N}$ is computed by M if M computes $f' : \{1\}^* \rightarrow \{1\}^*$ where $f'(1^n) = 1^{f(n)}$ for each $n \in \mathbb{N}$.

Example: $f(n) = n + 1$ for each $n \in \mathbb{N}$.

State	Symbol	$\delta(\text{State}, \text{Symbol})$
q_0	1	$(q_0, 1, R)$
q_0	#	$(q_0, 1, Y)$

$(q_0, \underline{11}\#) \vdash_M (q_0, \underline{11}\#) \vdash_M (q_0, \underline{11}\#) \vdash_M (q_0, \underline{111})$

In general, $(q_0, 1^n\#) \vdash_M^* (q_0, 1^{n+1}\#)$.

What about $n = 0$?

Functions on Natural Numbers

As another example, the following Turing machine calculates the addition of two unary numbers. It is assumed that the initial configuration of the machine is $(a, \underline{m}\#n\#)$, where m and n are the two numbers to be added.

State	Symbol	$\delta(\text{State}, \text{Symbol})$
a	1	$(b, \#, R)$
a	#	$(a, \#, Y)$
b	1	$(b, 1, R)$
b	#	$(c, \#, R)$
c	1	$(c, 1, R)$
c	#	$(d, 1, L)$
d	1	$(d, 1, L)$
d	#	$(e, \#, L)$
e	1	$(e, 1, L)$
e	#	$(a, \#, R)$

Recursive Languages

A language $L \subseteq \Sigma_0^*$ is *recursive* (also called *Turing-decidable*) iff the characteristic function $\chi_L : \Sigma_0^* \rightarrow \{Y, N\}$ is Turing-computable, where for each $w \in \Sigma_0^*$,

$$\chi_L(w) = \begin{cases} Y, & \text{if } w \in L \\ N, & \text{otherwise} \end{cases}$$

Example: Let $\Sigma_0 = \{a\}$, and let $L = \{w \in \Sigma_0^* : |w| \text{ is even}\}$.

M erases the marks from left to right, with the current parity encoded by the state. Once a blank at the right is reached, mark Y or N as appropriate.

Recursively Enumerable Languages

M *accepts* a string w if M halts on input w .

- M accepts a language iff M halts on w iff $w \in L$.
- A language is *recursively enumerable* (also called *Turing-acceptable* or *semi-decidable*) if there is some Turing machine that accepts it.

Example: $\Sigma_0 = \{a, b\}$, $L = \{w \in \Sigma_0^* : w \text{ contains at least one } a\}$.

State	Symbol	$\delta(\text{State}, \text{Symbol})$
q_0	a	(q_0, a, Y)
q_0	b	(q_0, b, R)
q_0	$\#$	$(q_0, \#, R)$

Every recursive language is recursively enumerable.

Combining Turing Machines

Two Turing machine computations M_1 and M_2 can be combined into larger machines:

- M_1 prepares string as input to M_2 .
- M_1 passes control to M_2 with I/O head at end of input.
- M_1 retrieves control when M_2 has completed.

Some Simple Machines

Basic machines:

- *Symbol-writing* machines M_a (one for each symbol $a \in \Sigma$).
- *Head-moving* machines R and L move the head appropriately.

These can be combined to create more complex machines:

- If the current symbol $a = b$, then do M_1 , else do M_2 ($IF(a = b, M_1, M_2)$)
- Move head to the right until a blank is found ($R_\#$).
- Find first blank square to left ($L_\#$).
- Copy Machine: Transform $w\#$ into $w\#w\#$
- $C = IF(a = \#, R_\#, M_\#R_\#M_aL_\#L_\#M_aRC)$
- Shift Machine: Transform $\#w\#$ into $w\#$
- $S = IF(a = \#, L, M_\#LM_aRRS)$

A Non-Context-Free Language

Construct a Turing machine to recognize the following language:

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

If the start cell is empty, then HALT.

If the current cell contains a ,

then write X and scan right.

Look for b , replace with Y .

Look for c , replace with Z .

Now, scan left to an a to right of X .

Repeat process.

Make sure nothing to right of c 's.

Extensions

The following extensions do not increase the power of Turing Machines.

- 2-way infinite tape
- Multiple tapes
- Multiple heads on one tape
- Two-dimensional “tape”
- Non-determinism

Grammars

The languages which can be recognised by Turing machines are those which can be described by an *unrestricted* or *free* grammar, also known as a *rewriting system*. This is a grammar in which every production has the form:

$\alpha \rightarrow \beta$, where α contains at least one non-terminal

Example: $L = \{w \in \{a, b, c\}^* \mid w \text{ has equal numbers of } a\text{'s, } b\text{'s and } c\text{'s}\}$.

S	\rightarrow	ϵ
S	\rightarrow	$ABC'S$
AB	\rightarrow	BA
AC	\rightarrow	CA
BC	\rightarrow	CB
BA	\rightarrow	AB
CA	\rightarrow	AC
CB	\rightarrow	BC
A	\rightarrow	a
B	\rightarrow	b
C	\rightarrow	c

Computable Functions

We wish to characterise those functions from \mathbb{N} to \mathbb{N} which can be computed.

We first of all define the following *basic* functions:

- $\text{zero}_k(n_1, \dots, n_k) = 0$.
- $\text{succ}(n) = n + 1$ for all $n \in \mathbb{N}$.
- $p_i(n_1, \dots, n_k) = n_i$ for $1 \leq i \leq k$.

We now define a way of *composing* existing functions to define new ones:

$$f(x) = h(g_1(x), \dots, g_m(x))$$

Primitive Recursion

We can construct a *primitive recursive* function f from existing functions h and g as follows:

$$\begin{aligned} f(x, 0) &= h(x) \\ f(x, \text{succ}(y)) &= g(x, y, f(x, y)) \end{aligned}$$

For example:

$$\begin{aligned} \text{plus}(m, 0) &= m \\ \text{plus}(m, \text{succ}(n)) &= \text{succ}(\text{plus}(m, n)) \\ \text{mult}(m, 0) &= \text{zero}(m) \\ \text{mult}(m, \text{succ}(n)) &= \text{plus}(m, \text{mult}(m, n)) \end{aligned}$$

All primitive recursive functions are *total* recursive functions.

There are computable functions from \mathbb{N} to \mathbb{N} which are not primitive recursive. For example, the Ackerman function:

$$\begin{aligned} \text{Ack}(0, n) &= \text{succ}(n) \\ \text{Ack}(\text{succ}(m), 0) &= \text{Ack}(m, 1) \\ \text{Ack}(\text{succ}(m), \text{succ}(n)) &= \text{Ack}(m, \text{Ack}(\text{succ}(m), n)) \end{aligned}$$

μ -Recursion

Unbounded minimisation: for a predicate P , the unbounded minimisation of P is the function f defined as follows:

$$f(x) = \min\{y \mid P(x, y) \text{ is true}\}$$

that is, the least value of y that satisfies $P(x, y)$. This is also denoted as follows:

$$f(x) = \mu y [P(x, y) \text{ is true}]$$

Functions defined in this way are called μ -recursive functions. For example:

$$\text{minus}(x, y) = \mu z [y + z = x]$$

All *partial* recursive functions are μ -recursive functions.

Theorem: A function is μ -recursive iff it is computable.

Gödelization

We can convert numbers to strings by using unary notation.

We can convert strings to unique numbers as well.

Assign each character in Σ a unique number - the i^{th} character has value i .

Denote the i^{th} prime number as p_i .

Represent string $S = s_{i_1}s_{i_2}\dots s_{i_k}$ of length k as:

$$g(s_{i_1}s_{i_2}\dots s_{i_k}) = p_1^{i_1}p_2^{i_2}\dots p_k^{i_k}$$

For example CAT is $2^33^15^{20}$.

Now we can use μ -recursive functions to compute functions from strings to strings.