

Griffith University

3515ICT Theory of Computation

Turing Machines

(Based loosely on slides by Harald Søndergaard of
The University of Melbourne)

Overview

- Turing machines: a general model of computation (3.1)
- Variants of Turing machines (3.2)
- Algorithms and the Church-Turing thesis (3.3)
- Decidable problems (4.1)
- The Halting Problem and other undecidable problems (4.2)
- Undecidable problems from language theory (5.1)
- The Post Correspondence Problem and its applications (5.2)
- Rice's Theorem (Problem 5.28)

Alan Turing

- Born London, 1912.
- Studied quantum mechanics, probability, logic at Cambridge.
- Published landmark paper, 1936:
 - Proposed most widely accepted formal model for algorithmic computation.
 - Proved the existence of computationally unsolvable problems.
 - Proved the existence of universal machines.
- Did PhD in logic, algebra, number theory at Princeton, 1936–38.
- Worked on UK cryptography program, 1939-1945.
- Contributed to design and construction of first UK digital computers, 1946–49.

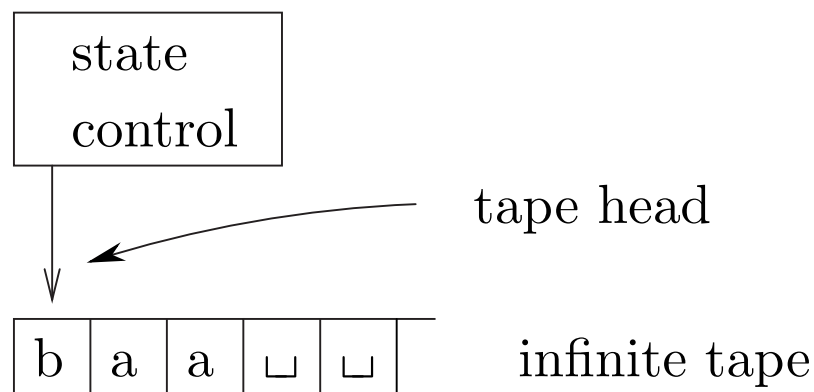
Alan Turing (cont.)

- Wrote numerical, learning and AI software for first computers, 1947–49.
- Proposed Turing Test for artificial intelligence, 1950.
- Studied mathematics of biological growth, elected FRS, 1951.
- Arrested for homosexuality, 1952; died (suicide), 1954.
- See *Alan Turing: The Enigma*, Andrew Hodge (Vintage, 1992).
- See <http://www.turing.org.uk/turing/>.

Turing Machines

So what was Turing's computational model?

A *Turing machine* (*TM*) is a finite state machine with an infinite tape from which it reads its input and on which it records its computations.



The machine has both *accept* and *reject* states.

Comparing a TM and a DFA or DPDA:

- A TM may both read from and write to its input tape.
- A TM may move both left and right over its working storage.
- A TM halts immediately when it reaches an accept or reject state.

Turing Machines Formally

A *Turing machine* is a 7-tuple
 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where

1. Q is a finite set of states,
2. Γ is a finite tape alphabet, which includes the blank character, \sqcup ,
3. $\Sigma \subseteq \Gamma \setminus \{\sqcup\}$ is the input alphabet,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the initial state,
6. $q_a \in Q$ is the accept state, and
7. $q_r \in Q$ ($q_r \neq q_a$) is the reject state.

The Transition Function

A transition $\delta(q_i, x) = (q_j, y, d)$ depends on:

1. the current state q_i , and
2. the current symbol x under the tape head.

It consists of three actions:

1. change state to q_j ,
2. over-write tape symbol x by y , and
3. move the tape head in direction d (L or R).

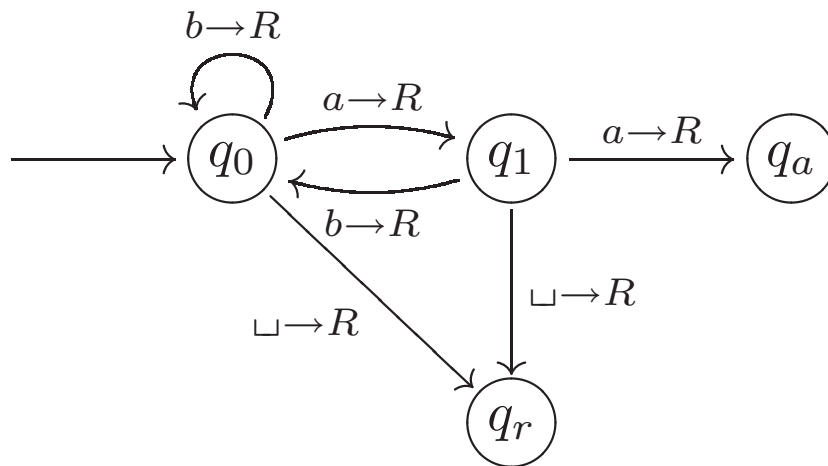
If the tape head is on the leftmost symbol, moving left has no effect.

We have a graphical notation for TMs similar to that for finite automata and PDAs.

On an arrow from q_i to q_j we write:

- $x \rightarrow d$ if $\delta(q_i, x) = (q_j, x, d)$, and
- $x \rightarrow y, d$ if $\delta(q_i, x) = (q_j, y, d)$, $y \neq x$.

Example 1



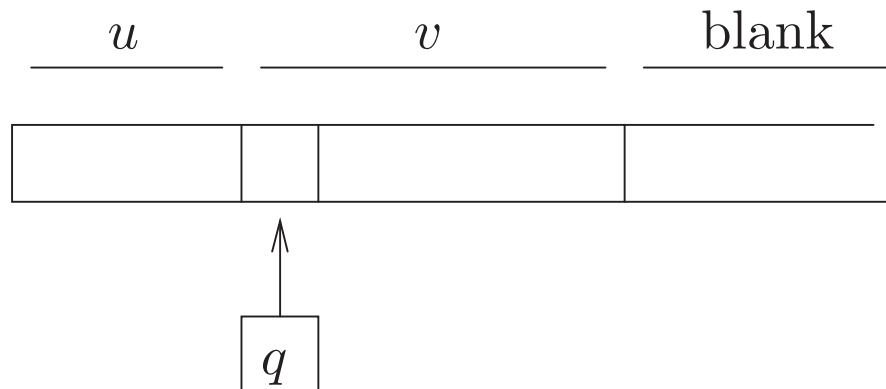
This machine recognises the regular language $(a + b)^* aa(a + b)^* = (b + ab)^* aa(a + b)^*$.

This machine is not very interesting — it always moves right and it never writes to its tape!
(Every regular language can be recognised by such a machine. Why?)

We can omit q_r from TM diagrams with the assumption that transitions that are not specified go to q_r .

Turing Machine Configurations

We write uqv for this configuration:



On input aba , the example machine goes through these configurations:

$$\begin{aligned} & \varepsilon q_0 aba \quad (\text{or just } q_0 aba) \\ \vdash & a q_1 ba \\ \vdash & ab q_0 a \\ \vdash & aba q_1 \sqcup \\ \vdash & aba \sqcup q_r \end{aligned}$$

We read the relation ' \vdash ' as 'yields'.

Computations Formally

For all $q_i, q_j \in Q$, $a, b, c \in \Gamma$, and $u, v \in \Gamma^*$, we have

$$uq_i b v \vdash ucq_j v \quad \text{if } \delta(q_i, b) = (q_j, c, R)$$

$$q_i b v \vdash q_j c v \quad \text{if } \delta(q_i, b) = (q_j, c, L)$$

$$uaq_i b v \vdash uq_j a c v \quad \text{if } \delta(q_i, b) = (q_j, c, L)$$

The *start configuration* of M on input w is $q_0 w$.

M *accepts* w iff there is a sequence of configurations $C_0, C_1, C_2, \dots, C_k$ such that

1. C_0 is the start configuration $q_0 w$,
2. $C_i \vdash C_{i+1}$ for all $i \in \{0, \dots, k-1\}$, and
3. The state of C_k is q_a .

Turing Machines and Languages

The set of strings accepted by a Turing machine M is the language *recognised by M* , $L(M)$.

A language A is *Turing-recognisable* or *computably enumerable (c.e.)* or *recursively enumerable (r.e.)* (or *semi-decidable*) iff $A = L(M)$ for some Turing machine M .

Three behaviours are possible for M on input w : M may accept w , reject w , or fail to halt.

If language A is recognised by a Turing machine M that *halts on all inputs*, we say that M *decides A* .

A language A is *Turing-decidable*, or just *decidable*, or *computable*, or (even) *recursive* iff some Turing machine decides A .

(The difference between Turing-recognisable and (Turing-)decidable is *very* important.)

Example 2 (Sipser, Example 3.7)

This machine decides the language $\{0^{2^n} \mid n \geq 0\}$.

(Note that this language is *not* context-free.)

We can describe how this machine operates as follows (an implementation level description):

$M_2 =$ ‘On input string w :

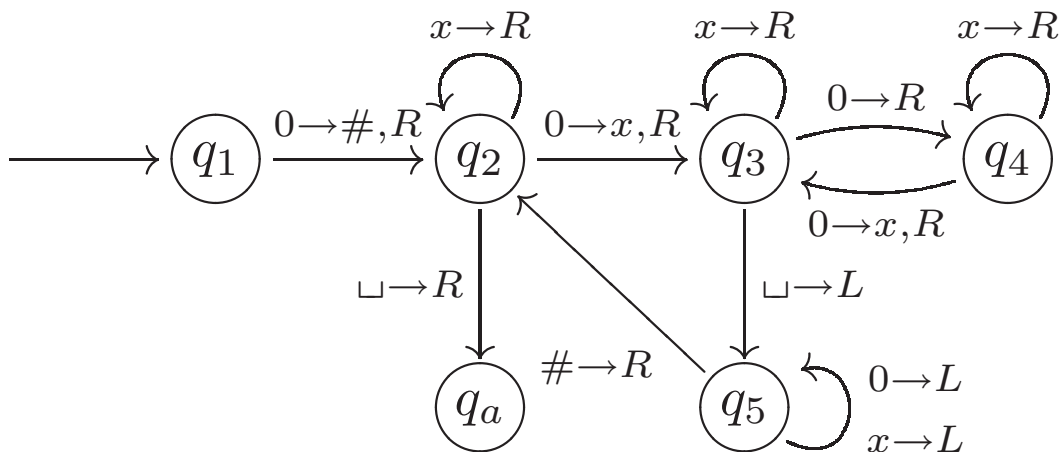
1. Read tape from left to right crossing off every second 0.
2. If the tape contained a single 0, *accept*.
3. Otherwise, if the tape contained an odd number of 0’s, *reject*.
4. Return to the left of the tape.
5. Go to step 1.’

Example 2 (cont.)

This machine decides the language $\{0^{2^n} \mid n \geq 0\}$.

A formal description of the machine follows.

It has input alphabet $\{0\}$, tape alphabet $\{\sqcup, 0, x, \#\}$ and start state q_1 .



Running the machine on input 000:

$q_1 000 \vdash \#q_2 00 \vdash \#xq_3 0 \vdash \#x0q_4 \sqcup \vdash \#x0 \sqcup q_r$

Running the machine on input 0000:

$q_1 0000 \vdash \#q_2 000 \vdash \#xq_3 00 \vdash \#x0q_4 0 \vdash$
 $\#x0xq_3 \sqcup \vdash \#x0q_5 x \sqcup \vdash \#xq_5 0x \sqcup \vdash \#q_5 x0x \sqcup \vdash$
 $q_5 \#x0x \sqcup \vdash \#q_2 x0x \sqcup \vdash \#xq_2 0x \sqcup \vdash \#xxq_3 x \sqcup \vdash$
 $\#xxxq_3 \sqcup \vdash \dots$

Example 3 (Sipser, Example 3.9)

This machine decides the language

$$\{ w\#w \mid w \in \{0,1\}^* \}.$$

(Again, this language is *not* context-free.)

M_3 = ‘On input string w :

1. Zig-zag over the tape to corresponding positions on either side of the $\#$ symbol checking that these positions contain the same symbol. If they do not, or if no $\#$ is found, *reject*.
2. When all symbols to the left of $\#$ have been marked, check for any remaining symbols to the right of $\#$. If there are none, *accept*; otherwise, *reject*.’

See pp.138–139 and Fig. 3.10, p.145, for details.

Example 4 (Sipser, Example 3.11)

This machine decides the language

$$\{ a^i b^j c^k \mid i \times j = k, i, j, k > 0 \}.$$

(This language is also *not* context-free.)

An implementation-level description of the machine follows.

$M_3 =$ ‘On input string w :

1. Cross off the first a , and scan to the right for the first b . Alternately cross off the first b and the first c until all bs are gone.
2. Move left to the last crossed-off a , restoring the crossed-off bs . If there is another a , return to step 1. Otherwise, move right past all crossed-off bs and cs . If no c 's remain, *accept*, else *reject*.’

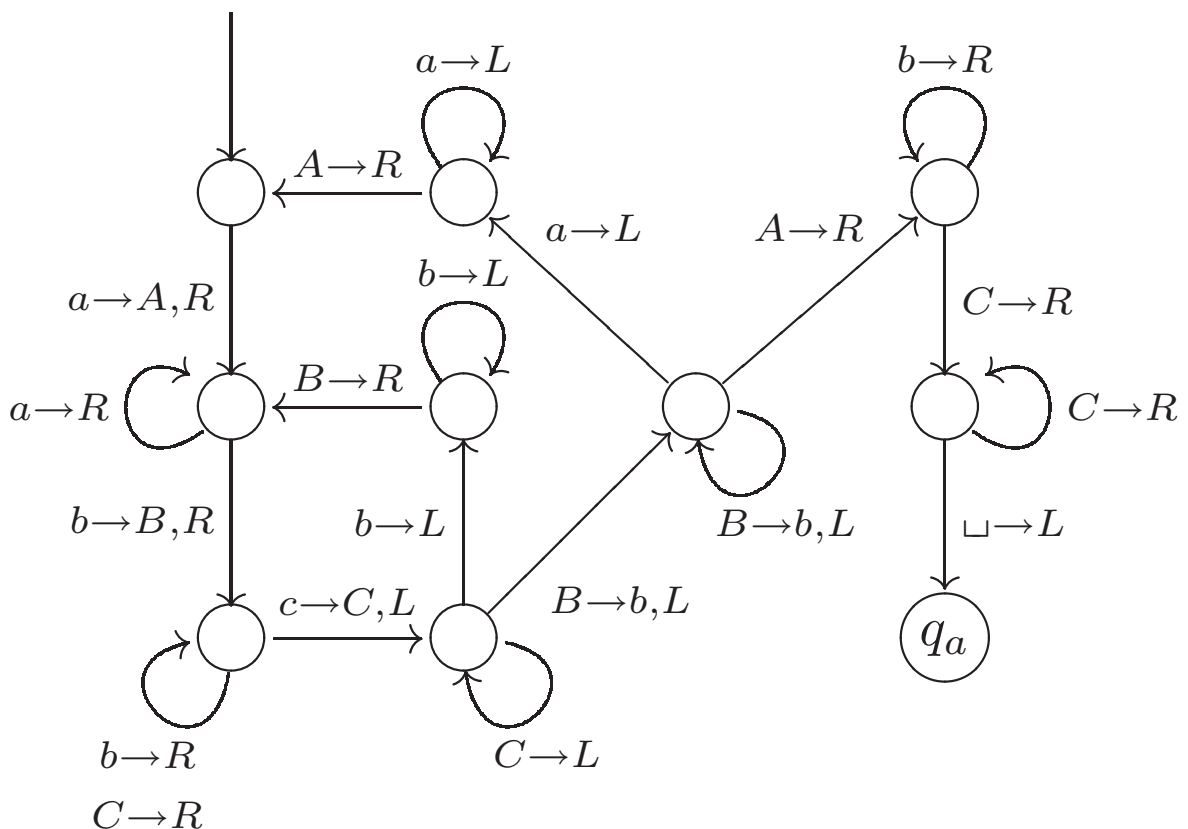
Example 4 (cont.)

This machine decides the language

$$\{ a^i b^j c^k \mid i \times j = k, i, j, k > 0 \}.$$

A formal description of the machine follows.

The tape alphabet is $\{\sqcup, a, b, c, A, B, C\}$.



Example 5 (Sipser, Example 3.12)

The *element distinctness problem*:

$E = \{ \#x_1\#x_2\#\dots\#x_l \mid \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j \}$

A machine to recognise (the non-CFL) E :

$M =$ ‘On input w :

1. Mark the leftmost symbol. If it was blank, *accept*. If it was not $\#$, *reject*.
2. Find the next $\#$ and mark it. If no $\#$ is found, *accept*.
3. By zig-zagging, compare the strings to the right of the two marked $\#$'s; if they are equal, *reject*.
4. Move the rightmost of the two marks to the next $\#$, if any. Otherwise, move the leftmost mark to the next $\#$, if any. Otherwise, *accept*.
5. Go to step 3.

Variants of Turing Machines

Other textbooks have definitions of Turing machines that differ slightly from Sipser's.

E.g., we may allow a 'stay put' move in addition to the left and right moves, *i.e.*, we may define the transition function as follows:

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

This clearly adds no expressive power, because any such TM can be transformed to a 'standard' TM by replacing each 'stay put' transition by two transitions, one that moves right to a temporary state and a second that immediately moves back to the left.

I.e., we can simulate 'stay put' TMs by 'standard' TMs.

But the robustness of Turing machines goes far beyond that...

Variants of Turing Machines (cont.)

More significant generalisations of Turing machines have also been proposed:

- Let the TM have a *doubly-infinite tape* (*i.e.*, infinite to left and right), with the tape head initially on the leftmost symbol of the (finite) input, and blanks to left and right of the input. (This is actually the most common definition!)
- Let the TM have *several tapes*, each with its own independent tape head, which may move left or right or stay put.
- Let the TM be *nondeterministic*.

It turns out that none of these variants increase a Turing machine's capabilities as a recogniser!

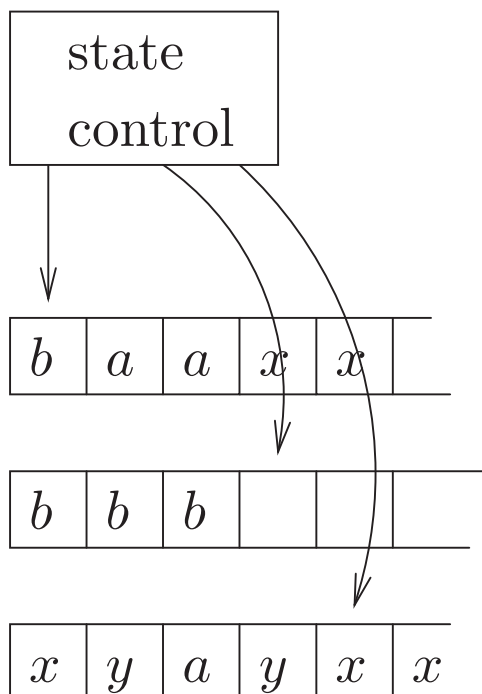
Multitape Machines

A multitape Turing machine has k tapes. Input is on tape 1, the other tapes are initially blank. The transition function now has the form

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

It specifies how the k tape heads behave when the machine is in state q_i , reading a_1, \dots, a_k :

$$\delta(q_i, a_1, \dots, a_k) = (q_j, (b_1, \dots, b_k), (d_1, \dots, d_k))$$



Multitape Machines (cont.)

Theorem. A language is Turing-recognisable iff some multitape Turing machine recognises it.

Proof outline. We show how to simulate a multitape Turing machine M by a standard Turing machine S .

S has tape alphabet $\{\#\} \cup \Gamma \cup \Gamma'$ where $\#$ is a separator, not in $\Gamma \cup \Gamma'$, and there is a one-to-one correspondence between symbols in Γ and 'marked' symbols in Γ' .

S initially reorganises its input $x_1x_2 \cdots x_n$ into

$$\#x'_1x_2 \cdots x_n \underbrace{\#\sqcup'\#\cdots\#\sqcup'\#}_{k-1 \text{ times}}$$

Symbols of Γ' represent marked symbols from Γ , and denote the positions of the tape heads on the k tapes of M .

Multitape Machines (cont.)

To simulate a move of M , S scans its tape to determine the marked symbols. S then scans the tape again, updating it according to M 's transition function. (Note that this requires adding a large but finite number of states—and transitions—to S .)

If a 'virtual head' of M moves to a $\#$ (*i.e.*, passes the input on that tape), S shifts that symbol, and every symbol after it, one cell to the right. In the vacant cell it writes \sqcup' . It then continues to apply M 's transition function.

If M *accepts* or *rejects*, then S *accepts* or *rejects*, respectively.

I.e., adding tapes to Turing machines do not enable them to recognise any more languages.

Nondeterministic Turing Machines

A *nondeterministic Turing machine (NTM)* has a transition function of the form

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

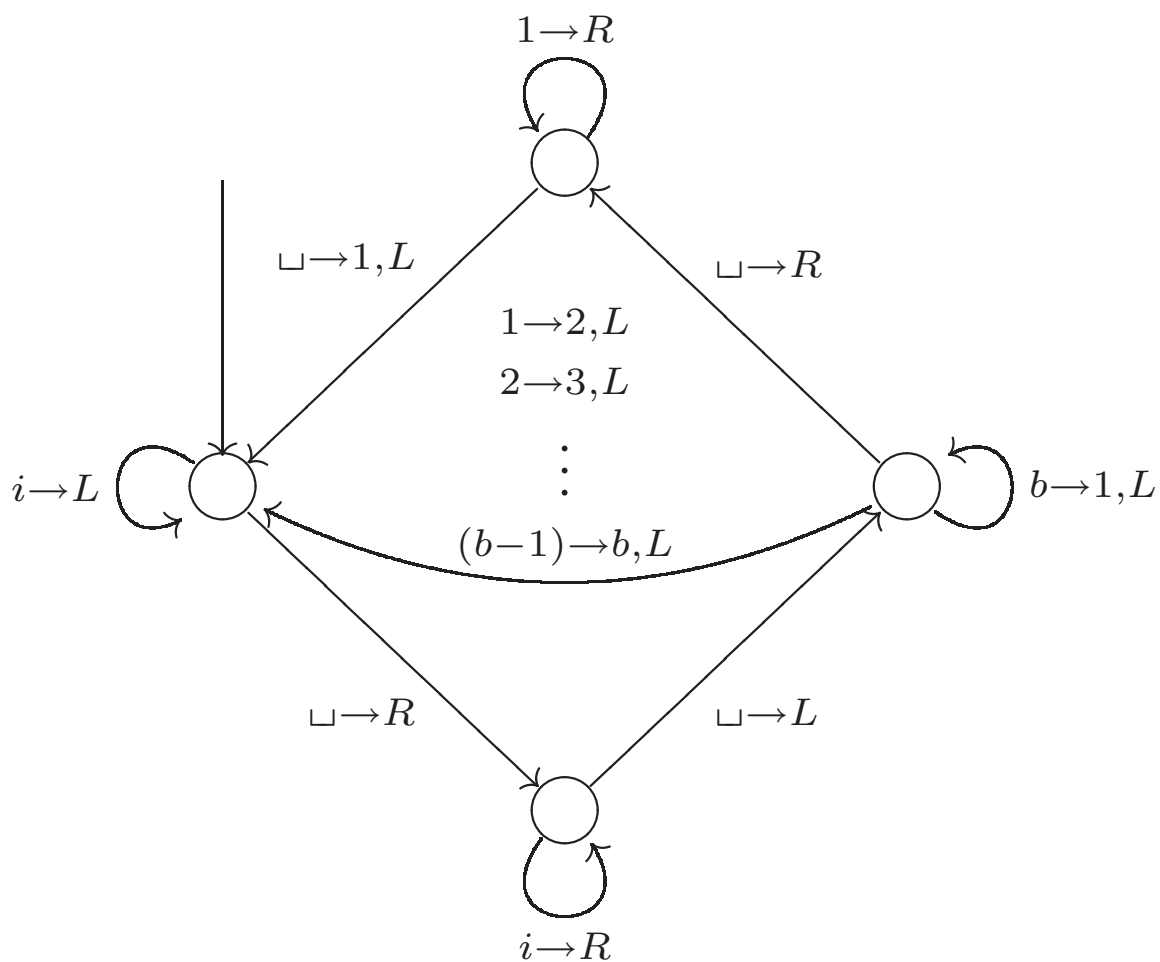
I.e., from each configuration, there may be several possible transitions, generating a ‘tree’ of computations.

If some computation branch leads to *accept*, then the machine *accepts* its input.

This is exactly the same form of nondeterminism as in NFAs and PDAs (except that some branches of an NTM may have infinite length).

Nondet. Turing Machines (cont.)

First, here is a deterministic machine, *NEXT*, to generate $\{1, \dots, b\}^*$, in increasing length, in lexicographic order within each length:



Try running this for $b = 3$, starting with a blank tape.

Simulating Nondeterminism

Theorem. A language is Turing-recognisable iff some nondeterministic Turing machine recognises it.

Proof outline. We need to show that every nondeterministic Turing machine N can be simulated by a (deterministic) Turing machine D .

We show how N can be simulated by a 3-tape Turing machine.

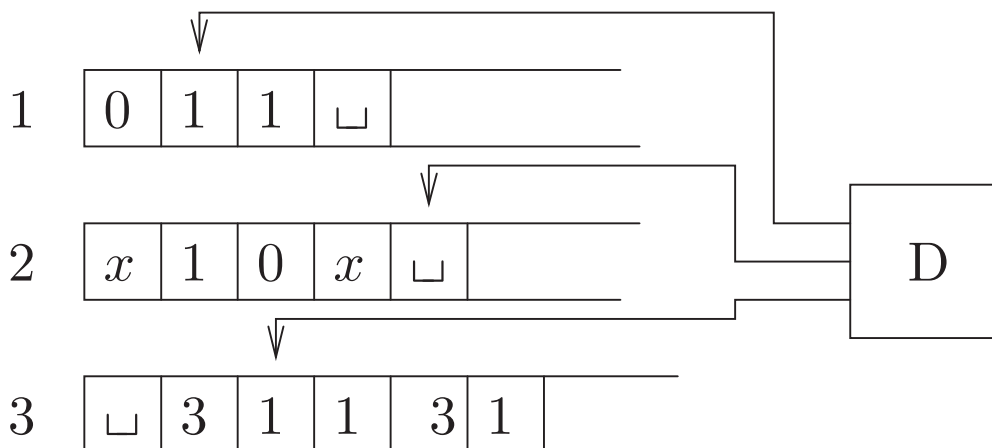
Let b be the largest number of choices, according to N 's transition function, for any state/symbol combination.

Tape 1 contains the input.

Tape 2 is used to simulate N 's behaviour for each sequence of choices given by tape 3.

Tape 3 holds successive sequences from $\{1, \dots, b\}^*$, in increasing length.

Simulating Nondeterminism (cont.)



1. Initially tape 1 contains input w and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N on one branch of the nondeterministic computation. Tape 3 determines how N makes each successive choice. If the end of tape 3 is reached, or if N rejects, go to step 4. If N accepts, *accept*.
4. Replace the string on tape 3 by the next string generated by the machine $NEXT$. If N *rejects* on *all* branches to this depth, *reject*. Otherwise, go to step 2.

Nondet. Turing Machines (cont.)

I.e., adding nondeterminism to Turing machines does not allow them to recognise any more languages.

A nondeterministic Turing machine that halts on every branch of its computation on all inputs is called a *decider*.

Corollary. A language is decidable iff some nondeterministic Turing machine decides it.

Enumerators

The Turing machine *NEXT* we built to generate all strings in $\{1, \dots, b\}^*$ is an example of an *enumerator*.

We could imagine it being attached to a printer, and it would print all the strings in $\{1, \dots, b\}^*$, one after the other, never terminating.

Or, we could imagine a 2-tape Turing machine that wrote—and never erased—the sequence of strings on its second tape.

For an enumerator to generate a language L , it must eventually generate (or print, or write) each string $w \in L$. It is allowed to generate any string more than once, and it is allowed to generate the strings in any order.

The reason why we also call Turing-recognisable languages *computably enumerable* (or recursively enumerable) is the following theorem.

Enumerators (cont.)

Theorem. A language L is Turing-recognisable iff some enumerator generates L .

Proof. Let E enumerate L . Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E . For each string s output by E , if $s = w$, accept.

Conversely, let M recognise L . Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate Σ^* , producing s_1, s_2, \dots . Here is what E does:

1. Let $i = 1$.
2. Simulate M for i steps on each of s_1, \dots, s_i .
3. For each accepting computation, print that s .
4. Increment i and go to step 2.

Enumerators (cont.)

Corollary. A language L is Turing-decidable iff some enumerator generates each string of L exactly once, in order of nondecreasing length.

Proof. Exercise for the reader. (Actually, a simplification of the previous proof.)

Restricted TMs

Adding capabilities to TMs does not give more power. What happens if we remove capabilities?

Theorem. Every TM can be simulated by a TM with a binary tape alphabet $\Gamma = \{1, B\}$.

Theorem. Not every TM can be simulated by a TM whose head always remains on the cells containing the initial input.

Equivalence with other models (cont.)

(Deterministic) Turing machines have been proved to have the same expressive power as each of the following computational models:

- Doubly-infinite Turing machines
- Multitape Turing machines
- Nondeterministic Turing machines
- Binary-tape-alphabet Turing machines
- Two-stack pushdown automata
- Queue machines
- Lambda-calculus expressions
- Recursively defined (numeric) functions
- Lisp-definable (symbolic) functions
- Register machines (effectively, stored program computers, with infinite storage)
- Many other models!

The Church-Turing Thesis

Church and Turing (in effect) proposed the following thesis:

A problem (or language) is (algorithmically) decidable if and only if some Turing machine can decide it.

Equivalently, as Turing machines can also compute functions, if we consider the final tape state as the function value:

A function is (algorithmically) computable if and only if some Turing machine can compute it.

We cannot *prove* this thesis because it concerns the informal concepts ‘algorithmically decidable’ and ‘algorithmically computable’.

But the evidence for this thesis is overwhelming and the thesis is universally accepted.

An example

Hilbert's tenth problem (1900). Find an algorithm that determines the integral roots of an arbitrary polynomial (e.g., $5x^3y^2z - 4x^2y^3 + 3y^2z^5 - 60 = 0$).

As it turns (Matijasevič 1970), no such algorithm exists! *I.e.*, Matijasevič proved that the problem

“Does polynomial p have integral roots?”

is not algorithmically *decidable*.

This fact, however, could only be proved after mathematicians had agreed on what an algorithm *is*.

An example (cont.)

Note that the corresponding language

$$P = \{ p \mid p \text{ is a polynomial with integral roots} \}$$

is Turing-recognisable.

To see this, here is how we can build a Turing machine M to recognise P .

Suppose the variables in p are x , y and z .

M can enumerate all integer triples (i, j, k) .

So M can evaluate p on each triple (i, j, k) in turn.

If any of these evaluations returns 0, M accepts.

(As there are infinitely many integer triples, M might never halt.)

Descriptions of TMs

Turing machines may be described at different levels:

1. *Formal description* (states, tape and transition function)
2. *Implementation description* (data representation on tape, head movement, procedural)
3. *Abstract* (high-level) *description* (English prose, ignoring data representation, control details)

Uses of Turing Machines

Problems from many domains may be *encoded* as strings for Turing machines:

1. Arithmetic
2. Polynomials
3. Context-free grammars
4. Graphs, *e.g.*, see Example 3.23, pp.157–159.

We can also regard a TM as defining a function, with input as the argument, and final tape state as the value.

These two observations mean that TMs can be used as general computing machines, not just as language recognisers.