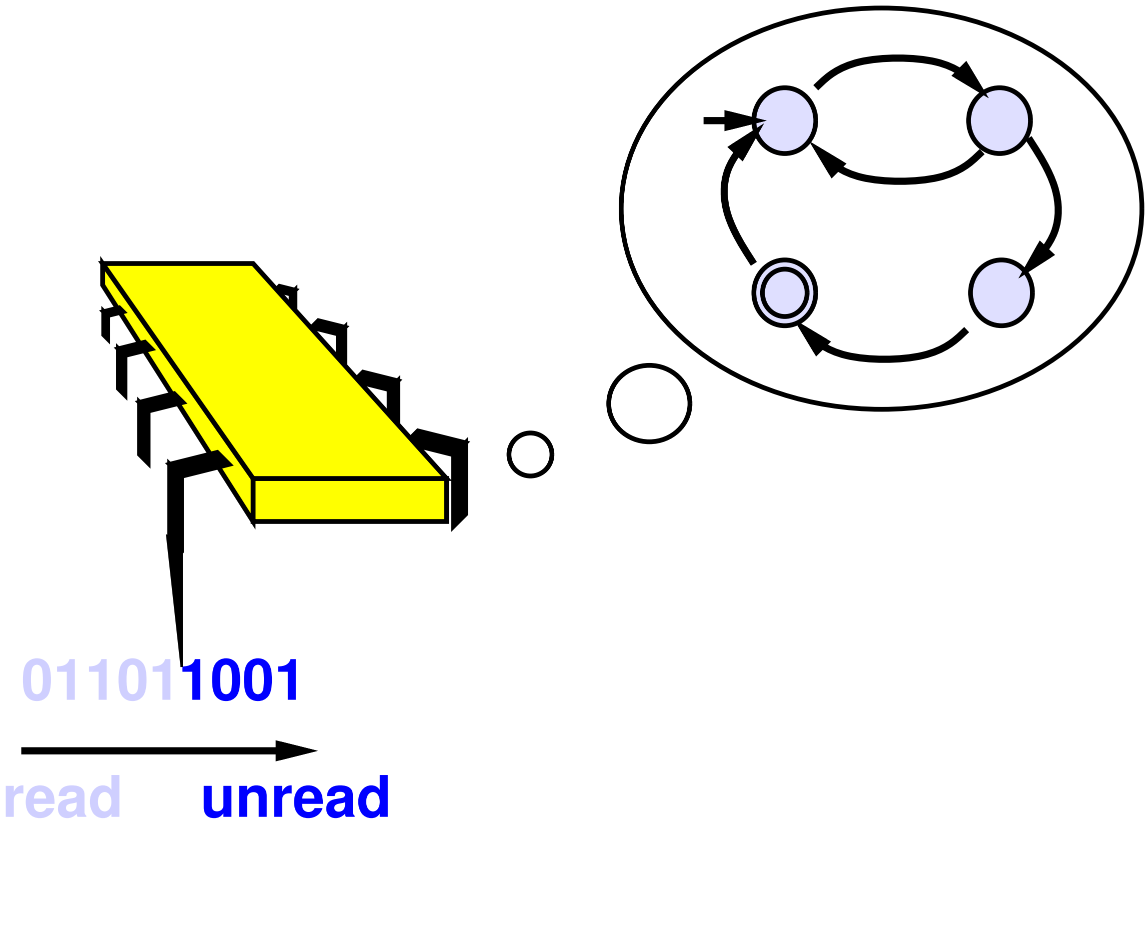
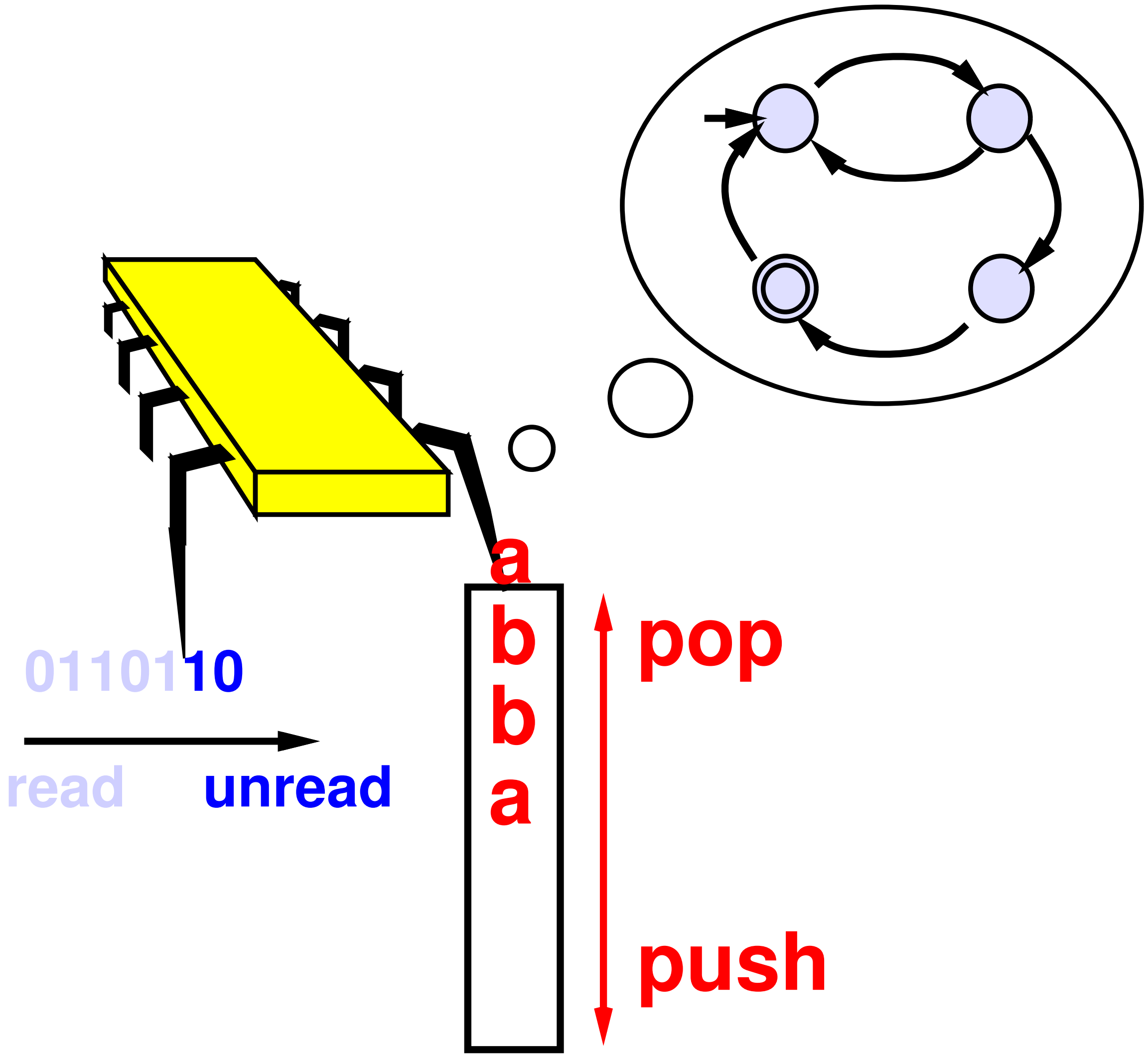
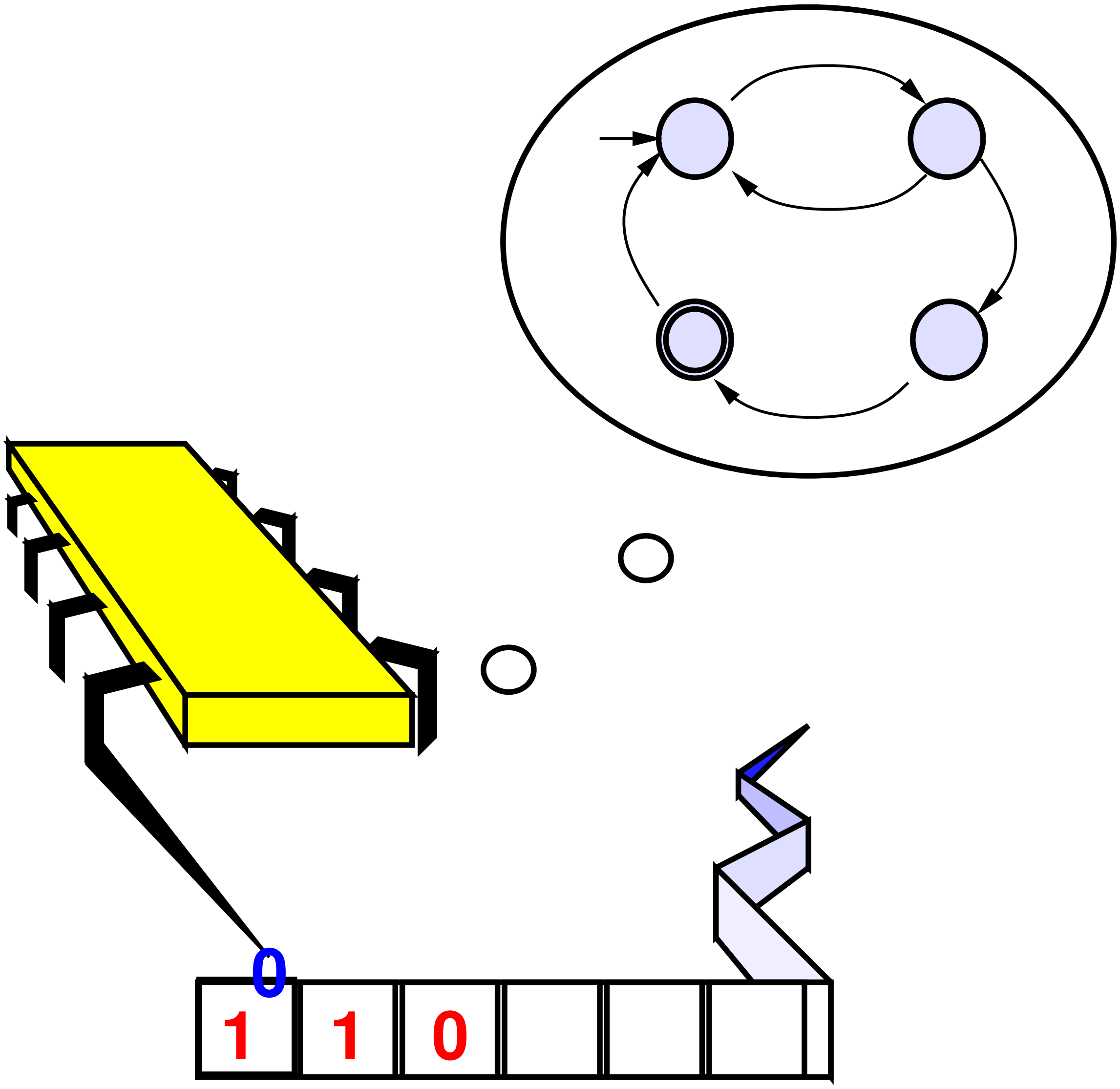
A Finite Automaton



A Pushdown Automaton



A Turing Machine



A Turing Machine

- uses infinite tape for memory
- tape initially contains input string followed by blanks
- tape can be used as scratch storage
- machine has *accept* and *reject* states
- machine can also run forever

Differences

- A Turing machine can both write on the tape and read from it.
- The read-write head can move both to the left and to the right
- the tape is infinite
- special accepting/rejecting states take immediate effect

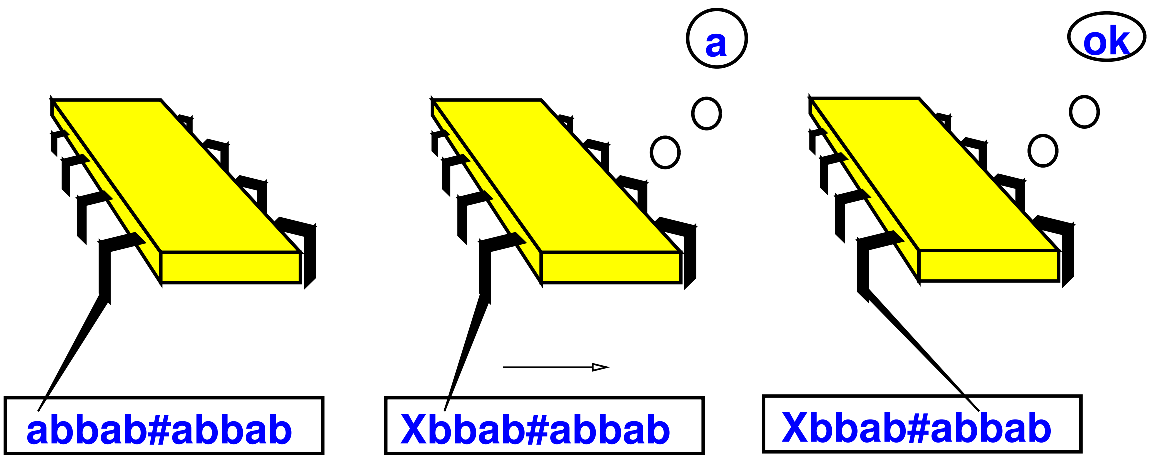
Example

A machine that tests for membership in the language

$$A = \{w\#w | w \in \{0,1\}^*\}$$

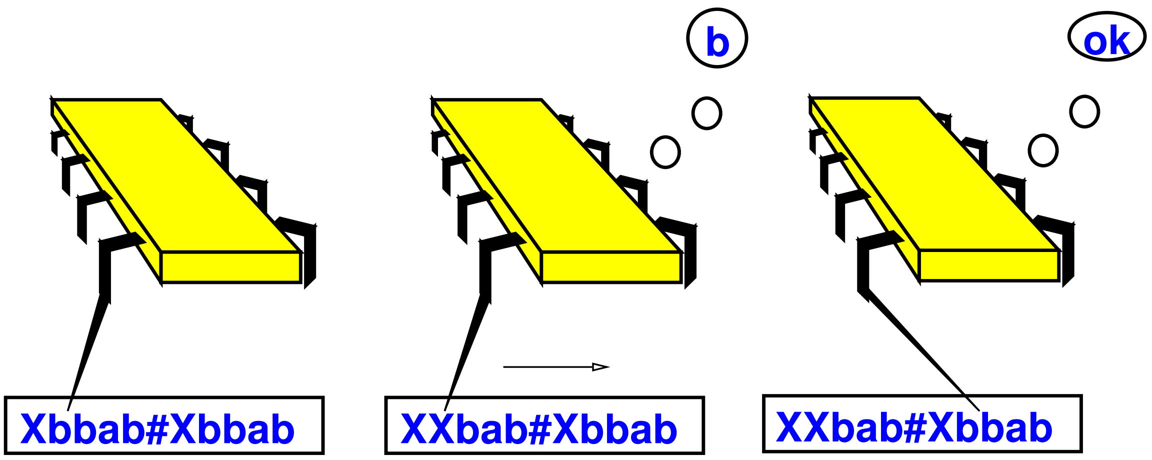
Zig-zag across tape, crossing of matching symbols.

Example



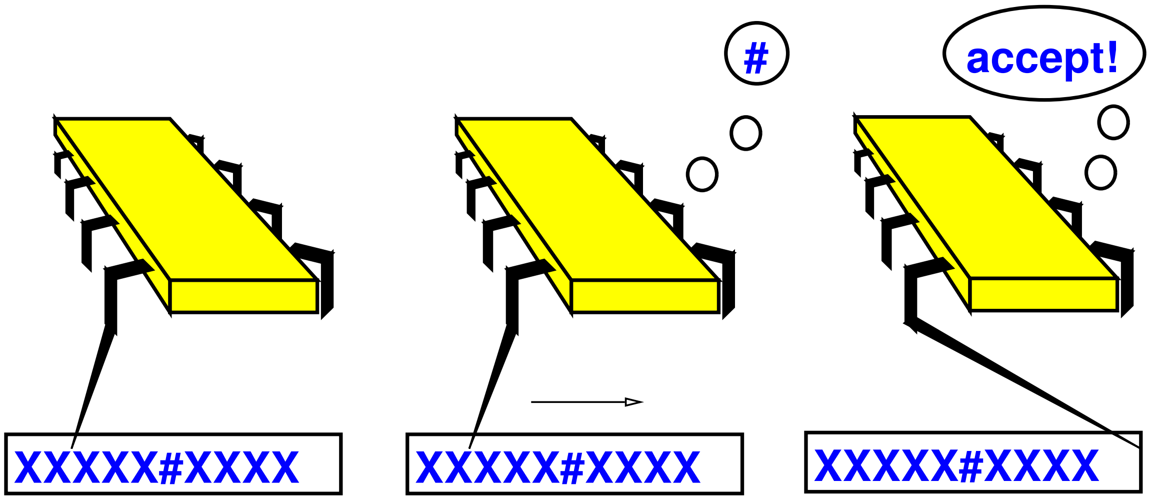
- tape head starts over leftmost symbol
- record symbol in control and overwrite X
- scan right: *reject*if blank encountered before #
- when # encountered, move right one space
- if symbols don't match, *reject*.

Example



- overwrite X
- scan left, past $\#$ to X
- move one space right
- record symbol and overwrite X
- scan right past $\#$ to X ...

Example



- finally, scan left
- if 0 or 1 encountered, *reject*
- when blank encountered, *accept*

Formal Definition

Focus on transition function.

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

(Sipser uses this definition, Papadimitriou doesn't.)

$\delta(q, a) = (r, b, L)$ means:

- in state q where head reads tape symbol a,
- the machine overwrites a with b,
- enters state r,
- moves the head left

Formal Definition

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

- Q is a finite set of *states*,
- Σ is the *input alphabet* not containing the blank symbol $\sqcup$
- Γ is *tape alphabet* containing $\sqcup$ and Σ
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $q_a \in Q$ is the *accept state*, and
- $q_r \in Q$ is the *reject state*.

Formal Definition

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ computes as follows

- input $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on leftmost n tape squares
- rest of tape is blank $\sqcup$
- head is on leftmost square of tape
- because $\sqcup \notin \Sigma$, blank is end of tape

Formal Definition

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$.

When computation starts,

- proceeds according to transition function δ
- if M tries to move head beyond left-hand-end of tape, it doesn't move.
- computation continues until q_a or q_r is reached
- otherwise runs forever

Configurations

Computation changes

- current state
- current head position
- tape contents

Configuration:

$$1011q_7 0111$$

means:

- state is q_7
- LHS of tape is 1011
- RHS of tape is 0111
- head is on RHS 0

Configurations

uaq_ibv *yields* uq_jacv if

$$\delta(q_i, b) = (q_j, c, L)$$

uaq_ibv yields $uacq_jv$ if

$$\delta(q_i, b) = (q_j, c, R)$$

Special case: q_ibv yields q_jcv if

$$\delta(q_i, b) = (q_j, c, L)$$

wq_i is the same as $wq_i\sqcup$.

More Configurations

We have

- starting configuration $q_0 w$
- accepting configuration $w_0 q_a w_1$
- rejecting configuration $w_0 q_r w_1$
- halting configurations $w_0 q_a w_1$ and $w_0 q_r w_1$

Accepting a Language

Turing machine M *accepts* input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exist

- C_1 is start configuration of M on w
- each C_i yields C_{i+1}
- C_k is an accepting configuration.

The collection of strings that M accepts is the *language* of M, denoted $L(M)$.

Enumerable Languages

Definition: A language is *(recursively) enumerable* if some Turing machine accepts it.

Enumerable Languages

On an input a TM may

- accept
- reject
- loop (run forever)

Not very practical: never know if TM will halt.

Enumerable Languages

Definition: A TM *decides* a language if it always halts in q_a or q_r.

Definition: A language is *decidable* if some Turing machine decides it.

Example

Here is a TM that accepts

$$\left\{a^i b^j c^k | i \times j = k \text{ where } i, j, k \geq 1\right\}$$

- scan from left to right to check that input is $a^*b^*c^*$
- return to start of tape
- cross off an a and scan right until b occurs. Shuttle between b's and c's until all b's are gone.
- Restore crossed-off b's and repeat previous step if more a's exist. If all a's crossed off, check if all c's crossed off. If yes, *accept*, otherwise *reject*.

Example

Question: How to tell when a TM is at the left end of the tape?

Answer: Mark it with a special symbol.

Or else,

- remember current symbol
- overwrite with special symbol
- move left
- if special symbol still there, head is at start
- otherwise restore previous symbol and move left

Example

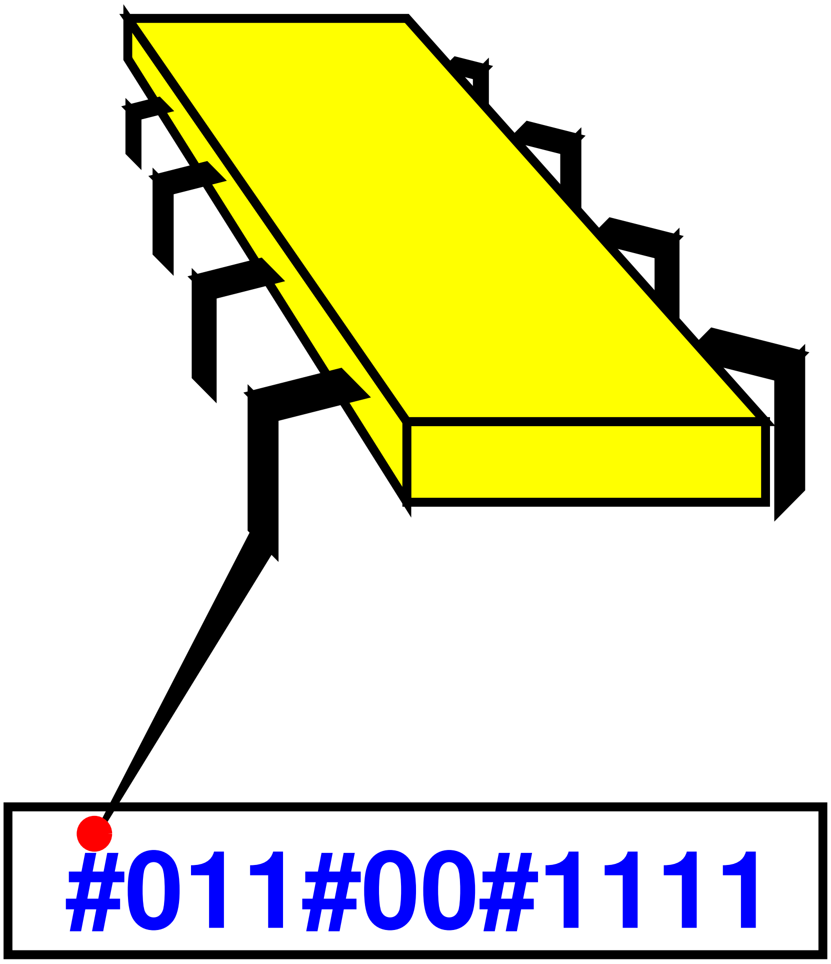
This machine solves *element distinctness* problem.

- List of strings in $\{0,1\}^*$ separated by #'s.
- accept if strings are different

Element Distinctness

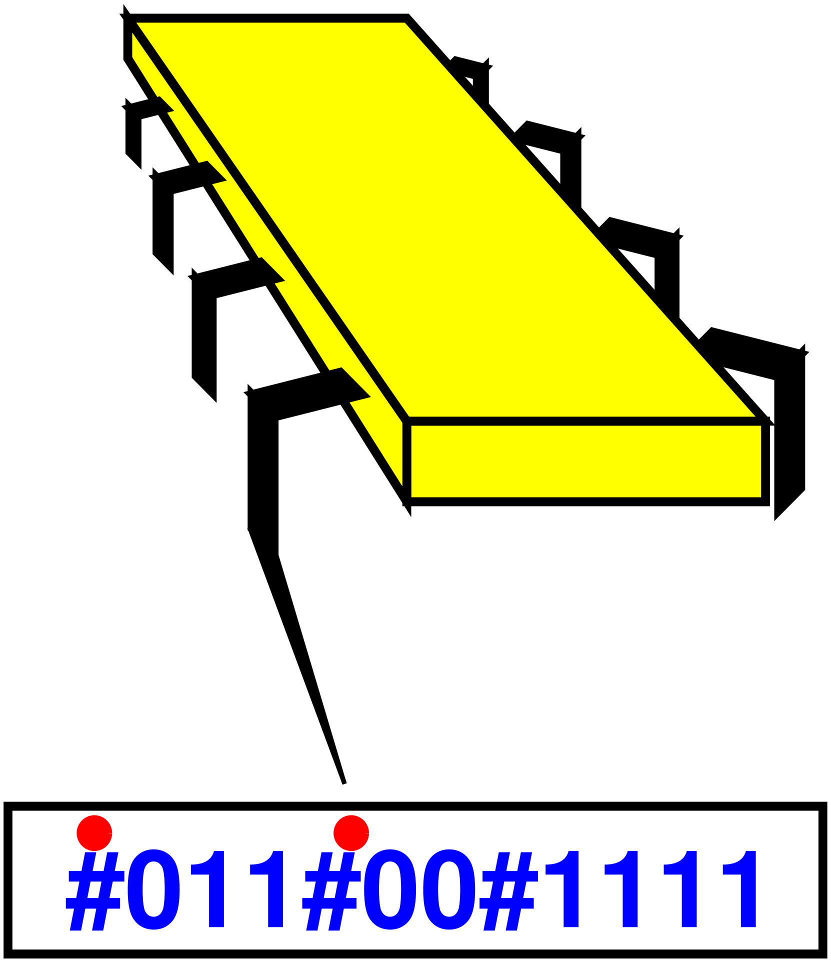
On input w

- place a mark on leftmost tape symbol. If symbol not #, reject.



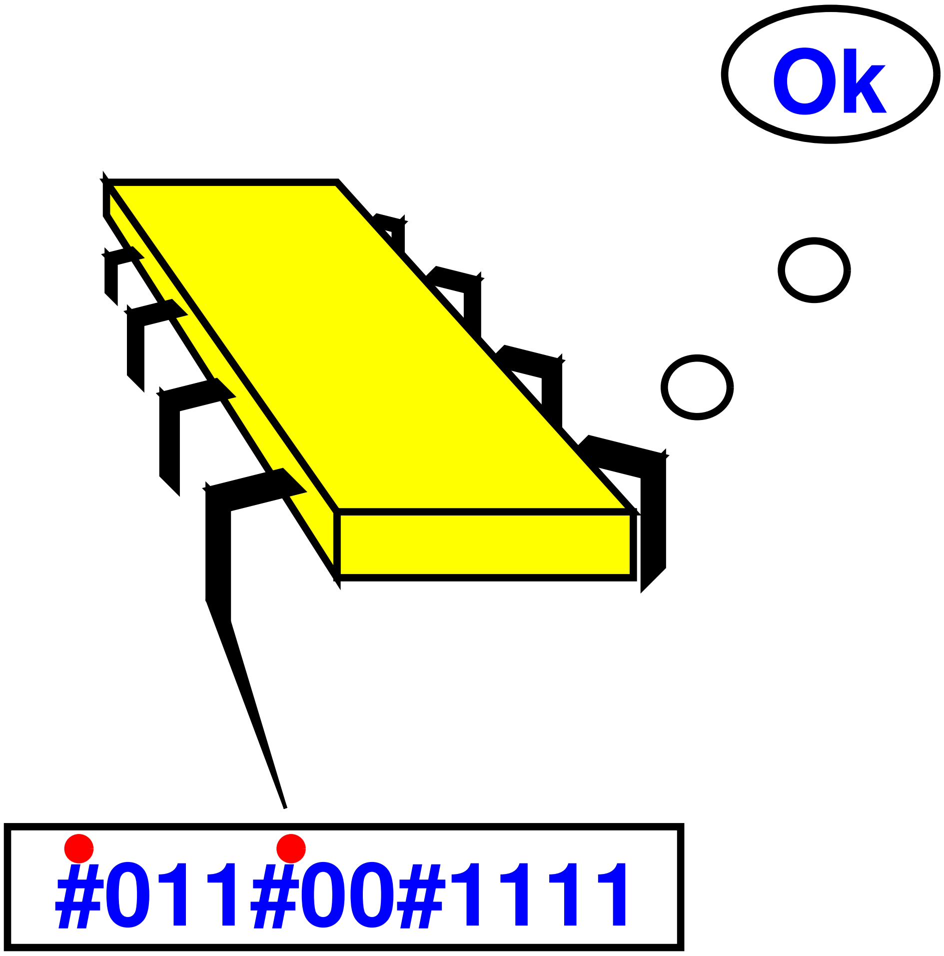
Element Distinctness

- scan right to next # and place mark on top. If none encountered, reject



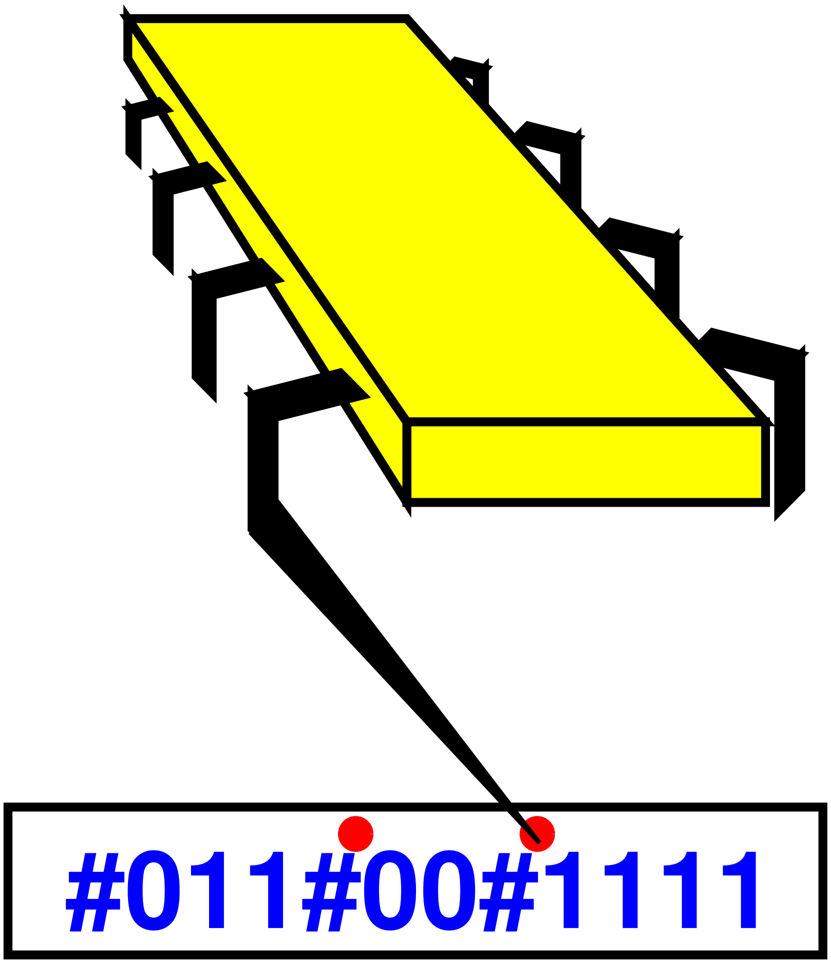
Element Distinctness

- By zig-zagging, compare the two strings to the left of the marked #'s. If equal, reject.



Element Distinctness

- Move rightmost mark to next #, if any.
- otherwise move leftmost mark to next # on right and rightmost mark to # after that.
- if not possible, accept.



Element Distinctness

Question: How do we "mark" a symbol?

Answer: For each tape symbol $\#$, add tape symbol $\overset{\bullet}{\#}$.

Variants

Alternative Turing machine definitions abound.

Suppose the Turing machine head is allowed to stay put.

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

Question: Does this add any power?

Answer: No. Replace each S transition with two transitions: R then L. (Why not vice-versa?)

Notion of simulation important.

Multitape Turing Machines

- each tape has its own head
- initially, input string on tape 1 and rest blank

Transition function:

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

Expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

- machine starts in q_i
- if heads 1 through k reading $a_1, \ldots, a_k$,
- then machine goes to q_j,
- heads 1 thought k write $b_1, \ldots, b_k$,
- and moves each head right or left as specified.

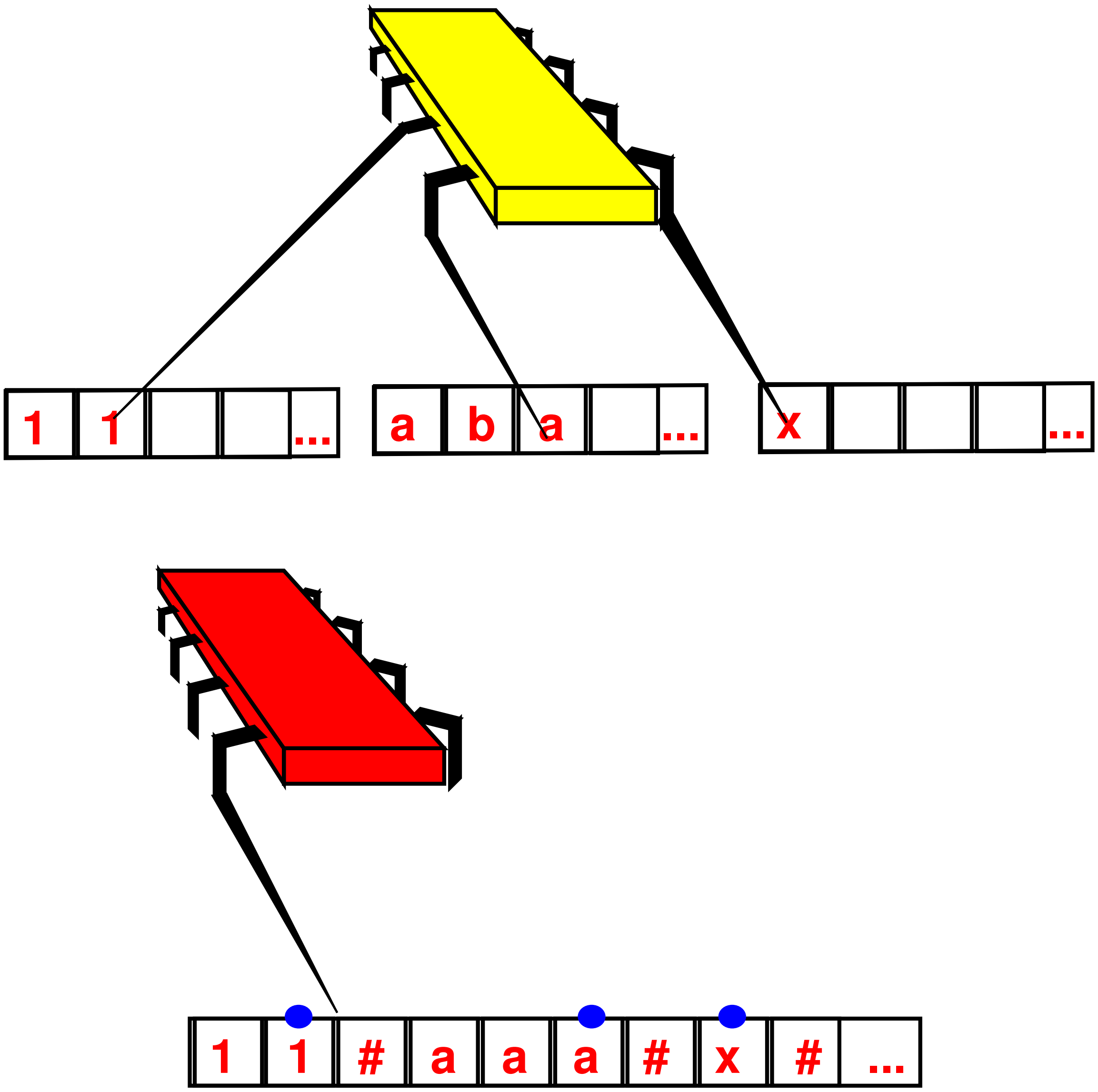
Equivalence

Theorem: A language is enumerable if and only if some multitape Turing machine accepts it.

One direction is trivial.

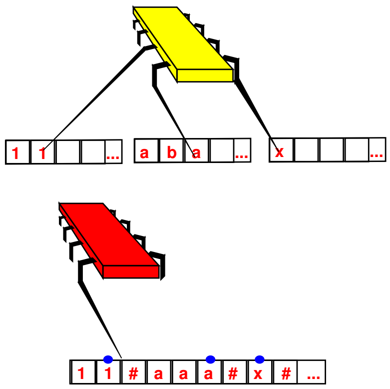
Show how to convert a multitape TM M into a single-tape TM S.

Simulation



- M has k tapes.
- S simulates k tapes by storing them all on a single tape with delimiter $\#$.
- S "marks" current head position

Simulation



On input $w = w_1 \cdots w_n$, S:

- puts on its tape:

$$\# \overset{\bullet}{w_1}\, w_2 \cdot w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

- scans its tape from first $\#$ to $k+1$-st $\#$ to read symbols under "virtual" heads.
- rescans to write new symbols and move heads
- if S tries to move virtual head onto $\#$, then M is trying to move head onto unused blank square. S writes blank symbol on tape, and shifts the rest of the tape right one square.

Non-Deterministic Turing Machines

Transition function:

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

- computation is a tree
- accepts if any branch accepts

Equivalence

Theorem: A language is enumerable if and only if some non-deterministic Turing machine accepts it.

One direction is trivial.

Show how to convert a non-deterministic TM N into a deterministic TM D.

Simulation

Basic idea:

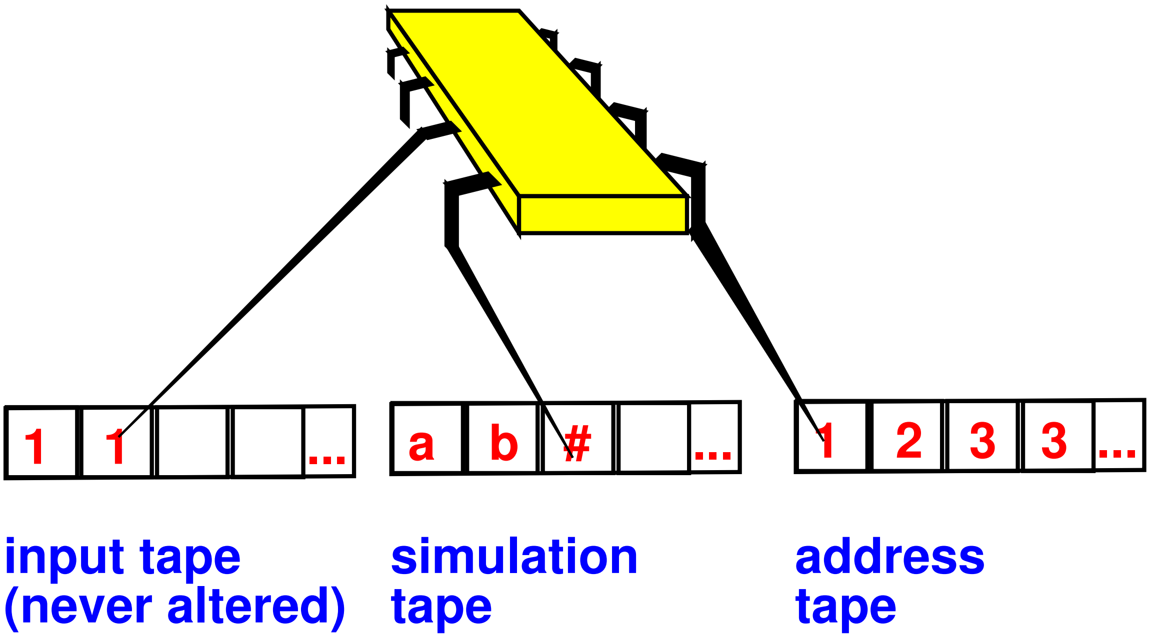
- D tries all possible branches
- If D finds any accepting state, it accepts.
- If all branches reject, D rejects.
- If all branches reject or loop, D loops.

Simulation

N's computation is a tree.

- each branch is branch of non-determinism
- each node is a configuration of N
- root is starting configuration
- depth-first search doesn't work (why?)
- breadth-first search works.

Simulation



D has three tapes

- the *input* tape is never altered
- the *simulation* tape is a copy of N's tape
- the *address* tape keeps track of D's location in N's computation tree.

Simulation

The address tape:

- every node in the tree has at most b children
- every node in the tree has an address that is a string over the alphabet

$$\Sigma_b = 1, 2, \ldots, b$$

- to get to node with address 123
  - start at root
  - take first branch
  - take second branch
- ignore meaningless addresses

Simulation

The simulation

- Initially, the input tape contains w, and the others are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on one non-deterministic branch. On each choice, consult the next symbol on address tape. Accept if accepting configuration reached. Skip to next step if
  - symbols on address tape exhausted
  - non-deterministic choice invalid
  - rejecting configuration reached
- Replace the string on the address tape with the lexicographically next string. Simulate this branch of N's computation by jumping to Step 2.

Deciders

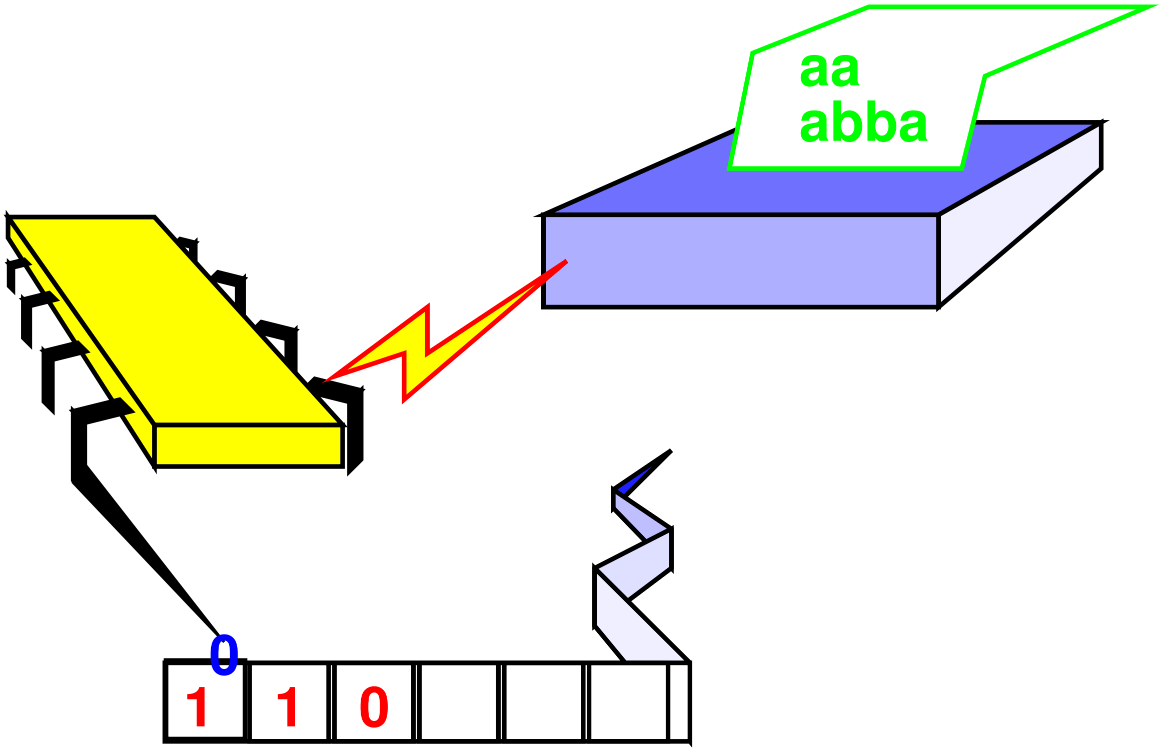
Definition: A non-deterministic TM is a *decider* if all branches halt on all inputs.

Theorem: A language is decidable if and only if there is a non-deterministic Turing machine that decides it.

Enumerators

A language is enumerable if some Turing machine accepts it.

But why *enumerable*?



Definition: An *enumerator* is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings
- *enumerates* a language.

Theorem

Theorem: A language is enumerable if and only if some enumerator enumerates it.

Must show

- If E enumerates language A, then some TM M accepts A.
- If M accepts A, then some enumerator E enumerates it.

Theorem

Claim: If E enumerates language A, then some TM M accepts A.

On input w

- run E. Every time E outputs a string, compare it to w.
- If w is output, accept.

Theorem

Claim: If M accepts A, then some enumerator E enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in Σ^*.

Define E

- repeat the following for $i = 1, 2, 3, \ldots$
- run M for i steps on each input $s_1, s_2, \ldots, s_i$.
- if any computation accepts, print out the corresponding s.

Note that duplicates may appear.

Remarks

Many models have been proposed for general-purpose computation.

Remarkably, all "reasonable" models are equivalent to Turing machines.

All "reasonable" programming languages are equivalent.

The notion of an *algorithm* is model-independent!

We don't really care about Turing machines as such, we care about computation.

What is an Algorithm?

Informally

- a recipe
- a procedure
- a computer program
- Who cares? I know it when I see it.

Historically,

- notion has long history in Mathematics, but
- not precisely defined until 20th century
- informal notions rarely questioned, but
- insufficient

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address.

- 23 mathematical problems
- challenge for the next century
- the 10th problem concerned algorithms

First, some background . . .

Polynomials

A *term* is a product of *variables* and a constant *coefficient*

$$6x^3yz^2$$

A *polynomial* is a sum of terms:

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

A *root* of a polynomial is an assignment of values to variables so that the value of the polynomial is zero.

For example $x = 5, y = 3$, and $z = 0$ is a root of this polynomial.

We are interested in *integral* roots.

Some polynomials have integral roots, some don't.

Hilbert's 10th Problem

The Problem: Devise an algorithm that tests whether a polynomial has an integral root.

Actually, what he said was

> to devise a process according to which it can be determined by a finite number of operations.

Note that

- Hilbert explicitly asks that algorithm be "devised"
- apparently assumes that algorithm exists and one only need find it.

Hilbert's 10th Problem

We now know no such algorithm exists.

Mathematicians of 1900 could not have proved this, because they didn't have a formal notion of an algorithm.

Informal notions work fine for constructing algorithms (we know one when we see it).

Formal notions are required to show that no algorithm exists.

Church-Turing Thesis

Formal notion appeared in 1936:

- λ-calculus of Alonzo Church
- Turing machines of Alan Turing
- these definitions look very different, but are equivalent.

The *Church-Turing Thesis*:

> The intuitive notion of algorithms equals Turing machine algorithms.

In 1970, it was shown that no algorithm exists for testing whether a polynomial has integral roots.

Hilbert's 10th Problem

Consider the language:

$$D = \{p | p \text{ is a polynomial with an integral root}\}$$

Hilbert's tenth problem asks whether this language is decidable.

It is not decidable, but it is enumerable!

Hilbert's 10th Problem

Consider the simpler language:

$$D_1 = \{p | p \text{is a polynomial over } x \text{ with an integral root}\}$$

Here is a Turing machine that accepts D_1.

On input p,

- evaluate p with x set successively to $0, 1, -1, 2, -2, \ldots$.
- if p evaluates to zero, accept.

Remark

$D_1 = \{p | p \text{is a polynomial over } x \text{ with an integral root}\}$

Note that

- If p has an integral root, the machine accepts.
- if not, it loops
- M_1 is an acceptor, but not a decider.

Remark

In fact, D_1 is decidable.

The roots of p lie between

$$\pm k \frac{c_{max}}{c_1}$$

where c_{max} is max coefficient, and c_1 is high-order coefficient.

Doesn't work for multivariable polynomials.

Remark

What about "unreasonable" models of computation.

Consider Digital's unreleased $\aleph$-AXP processor.

Like a Turing machine, except

- Takes first step in 1 second.
- Takes second step in 1/2 second.
- Takes i-th step in 2^{-i} seconds . . .

After 2 seconds, the $\aleph - AXP$ decides any enumerable language!

Question: Does this invalidate the Church-Turing Thesis?

Encoding

Input to a Turing machine is a string of symbols.

But we want algorithms that work on graphs, languages, Turing machines, etc.

Need to choose an *encoding* for objects.

Can often be done in many reasonable ways.

Sometimes distinguish between X the object and $\langle X \rangle$, the encoding.

Encoding

Let A be the language consisting of strings representing undirected graphs.

A graph is *connected* if every node can be reached from any other node by traveling along edges.

The language:

$$A = \{\langle G \rangle | G \text{is a connected undirected graph}\}$$

High-Level Description

High-level description of a machine that decides

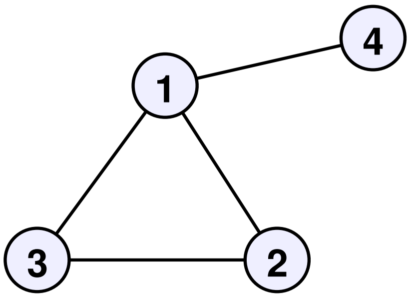
$$A = \{\langle G\rangle | G \text{is a connected undirected graph}\}$$

On input $\langle G\rangle$, encoding of graph G

- select first node of G and mark it.
- repeat until no new nodes marked:
- For each node in G, mark if attached by an edge to a node already marked.
- scan nodes of G to determine whether they are all marked. If so, accept, otherwise reject.

Some Details

Question: How is G encoded?



vertexes edges

(1,2,3,4)((1,2),(2,3),(3,1),(1,4))

G **<G>**

Answer: List of nodes followed by list of edges.

More Details

On input M checks that input is valid graph encoding

- two lists
- first is list of numbers
- second is list of pairs
- first list contains no duplicates (element distinctness subroutine)
- every node in second list appears in first

Now ready to start "step one"

Detailed Algorithm

On input $\langle G \rangle$, encoding of graph G

1 mark first node with a dot on leftmost digit.

2 loop:
- M scans list and "underlines" undotted node n_1.
- M rescans and "underlines" dotted node n_2.
- M scans edges.
- M tests each edge if it is (n_1, n_2).
- If so, dot n_1, remove underlines, and restart Step 2.
- If not, check next edge. When no more edges, move underline to next dotted n_2.
- when no more dotted vertexes, move underlines: new n_1 is next undotted node and new n_1 is first dotted node. Repeat Step 2. When no more undotted nodes, go to Step 3.

3 M scans the list of nodes. If all dotted, accept, otherwise reject.

Even more detailed description possible!