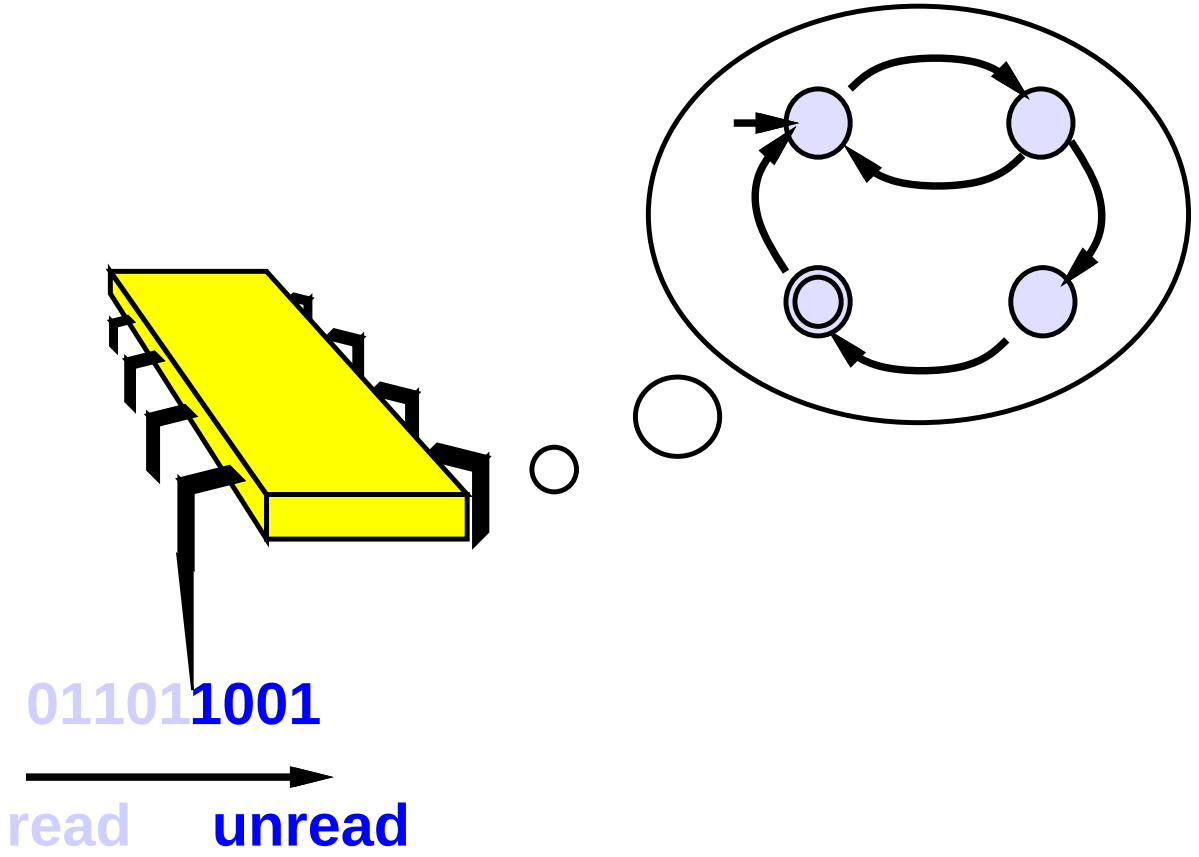
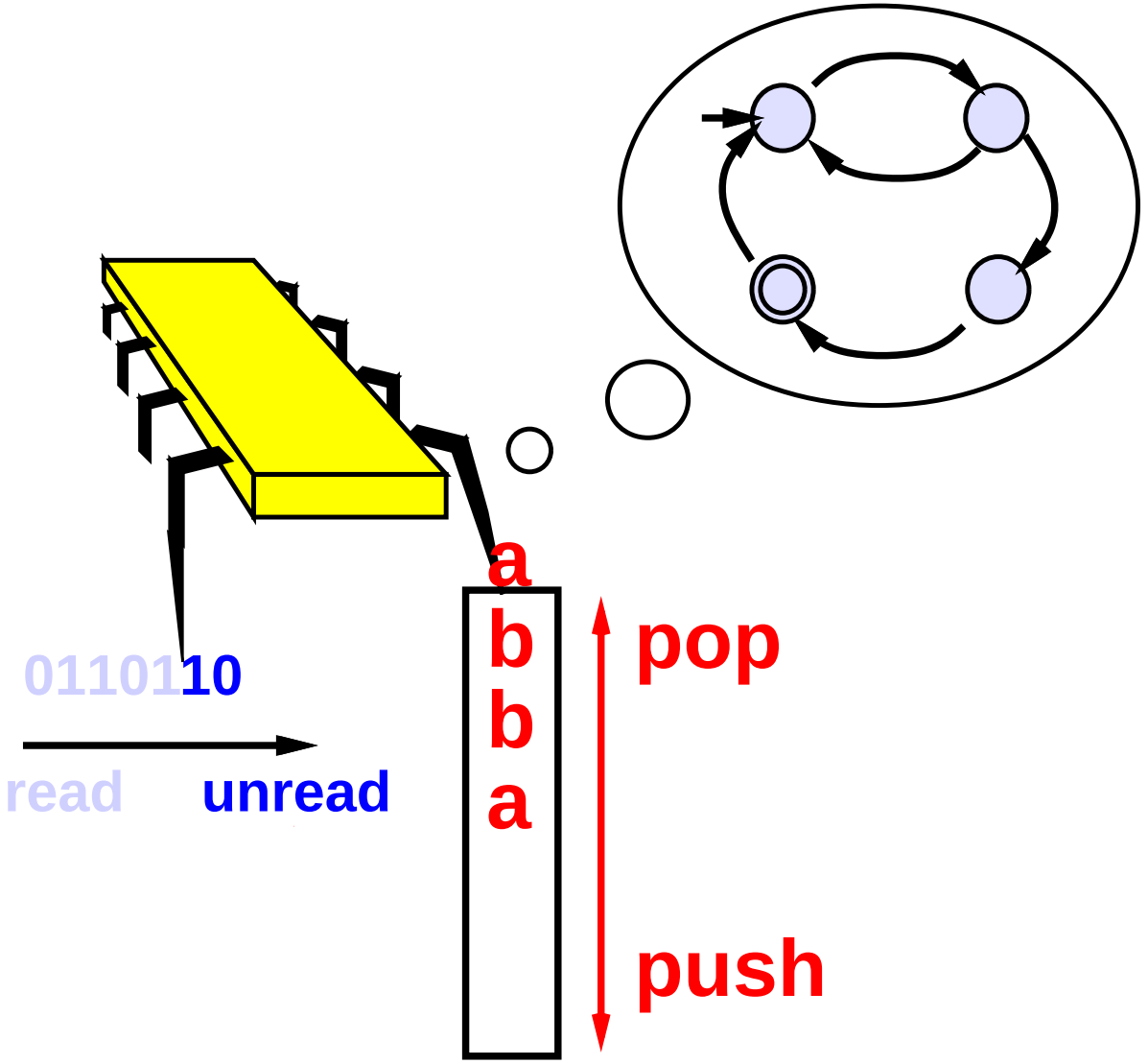
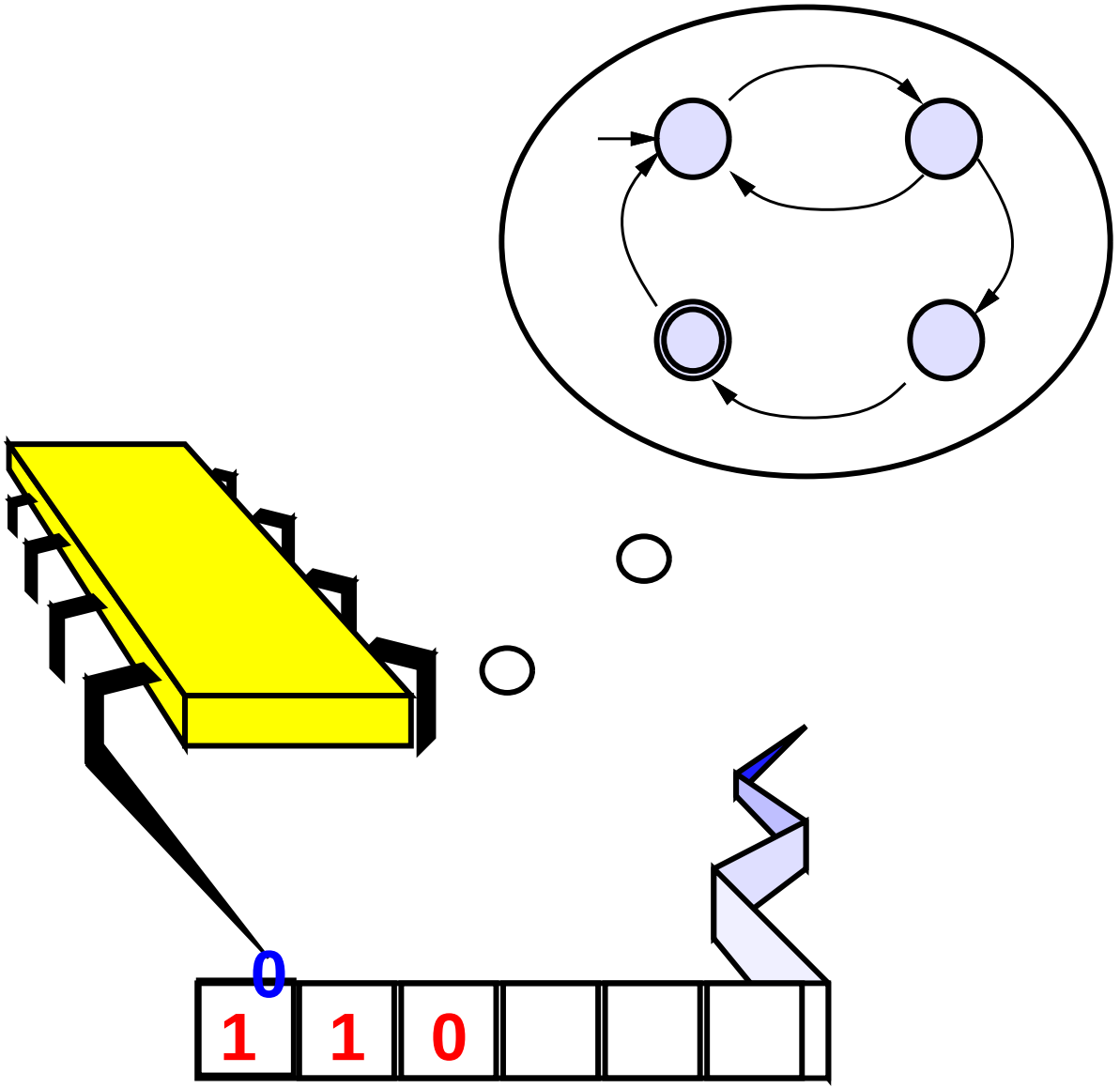
A Finite Automaton



A Pushdown Automaton



A Turing Machine



Alan Turing (1912–1954)

http://www.turing.org.uk/turing/index.html

A Turing Machine (TM)

- uses infinite tape for its memory.

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.
- tape initially contains input string, followed by blanks (␣s).

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.
- tape initially contains input string, followed by blanks (␣s).
- machine can read from and write on tape (storage device).

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.
- tape initially contains input string, followed by blanks (␣s).
- machine can read from and write on tape (storage device).
- machine can move its “head” both left and right on tape.

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.
- tape initially contains input string, followed by blanks (␣s).
- machine can read from and write on tape (storage device).
- machine can move its “head” both left and right on tape.
- machine halts in either accept or reject states.

A Turing Machine (TM)

- uses infinite tape for its memory.
- at any point in a computation, only a finite portion of tape has been accessed.
- tape initially contains input string, followed by blanks (␣s).
- machine can read from and write on tape (storage device).
- machine can move its “head” both left and right on tape.
- machine halts in either accept or reject states.
- machine can also run forever, never halting.

TM vs. DFA: Differences

- A Turing machine can both write on the tape and read from it.

TM vs. DFA: 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

TM vs. DFA: 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

TM vs. DFA: 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

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

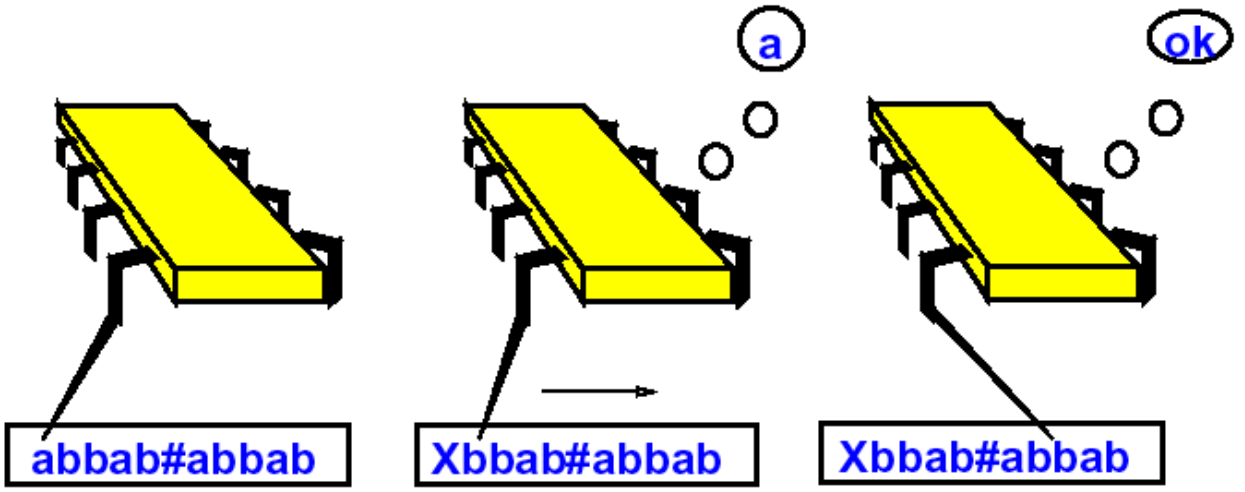
Example

A machine that tests for membership in the language

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

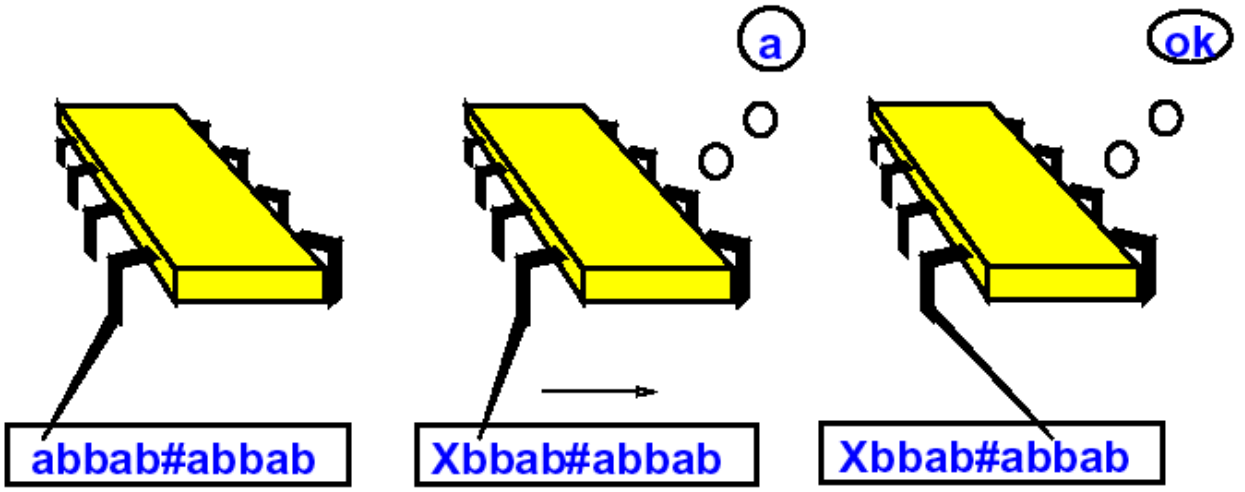
Zig-zags across tape, crossing off matching symbols.

Example (2)



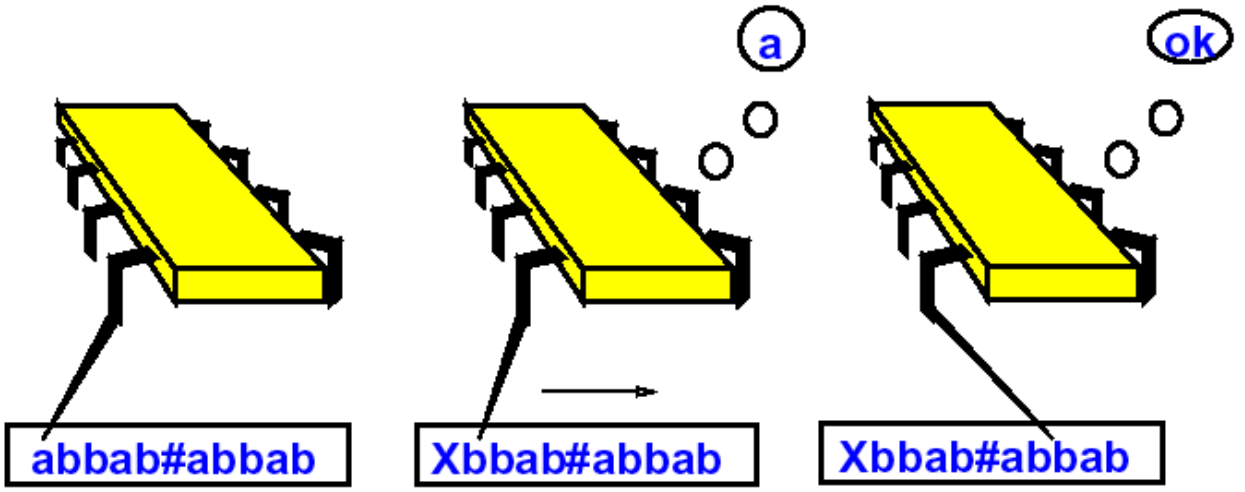
- tape head starts over leftmost symbol

Example (2)



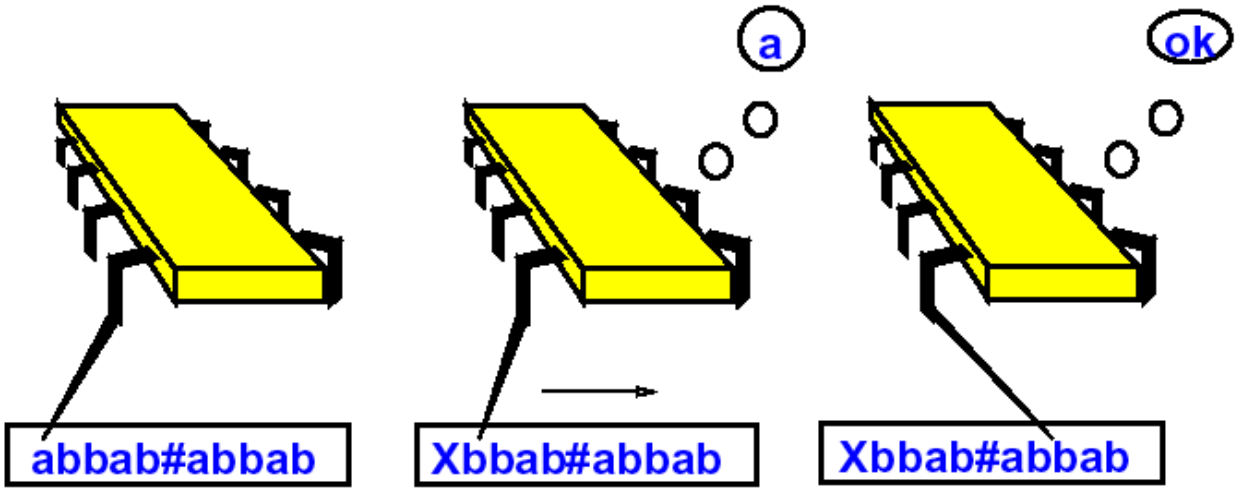
- tape head starts over leftmost symbol
- record symbol in control and overwrite X on tape

Example (2)



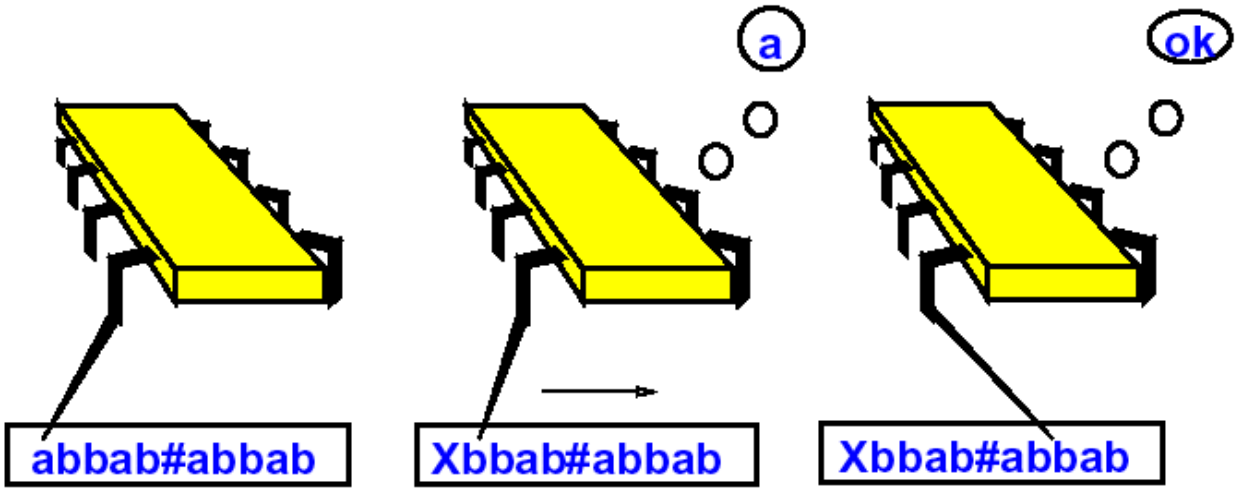
- tape head starts over leftmost symbol
- record symbol in control and overwrite X on tape
- scan right: reject if encounter blank ($\sqcup$) before $\#$

Example (2)



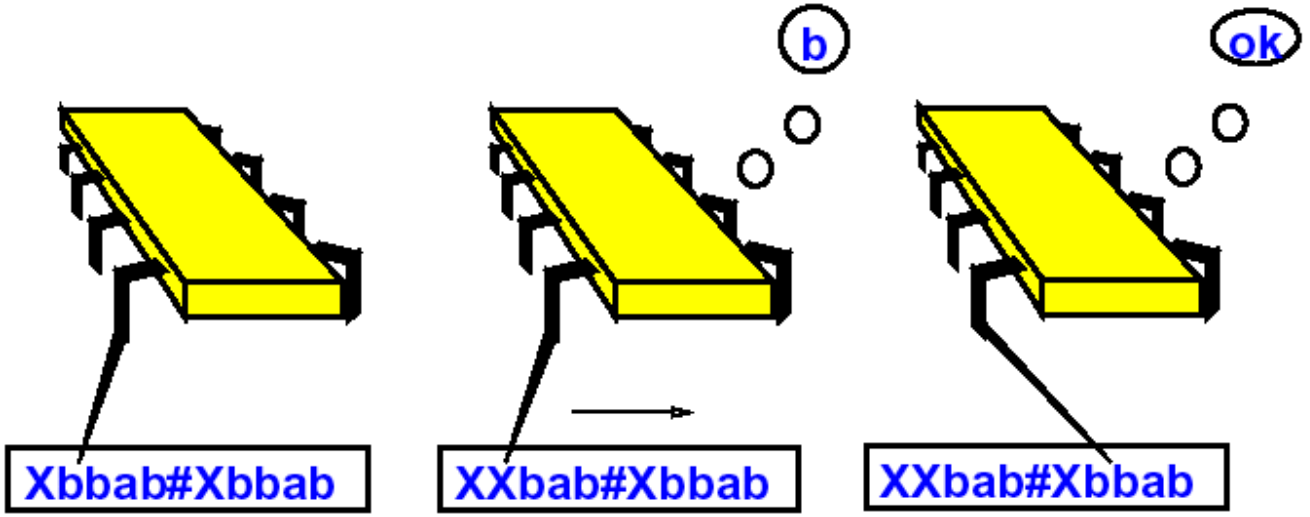
- tape head starts over leftmost symbol
- record symbol in control and overwrite X on tape
- scan right: reject if encounter blank ($\sqcup$) before $\#$
- when $\#$ encountered, move right one space

Example (2)



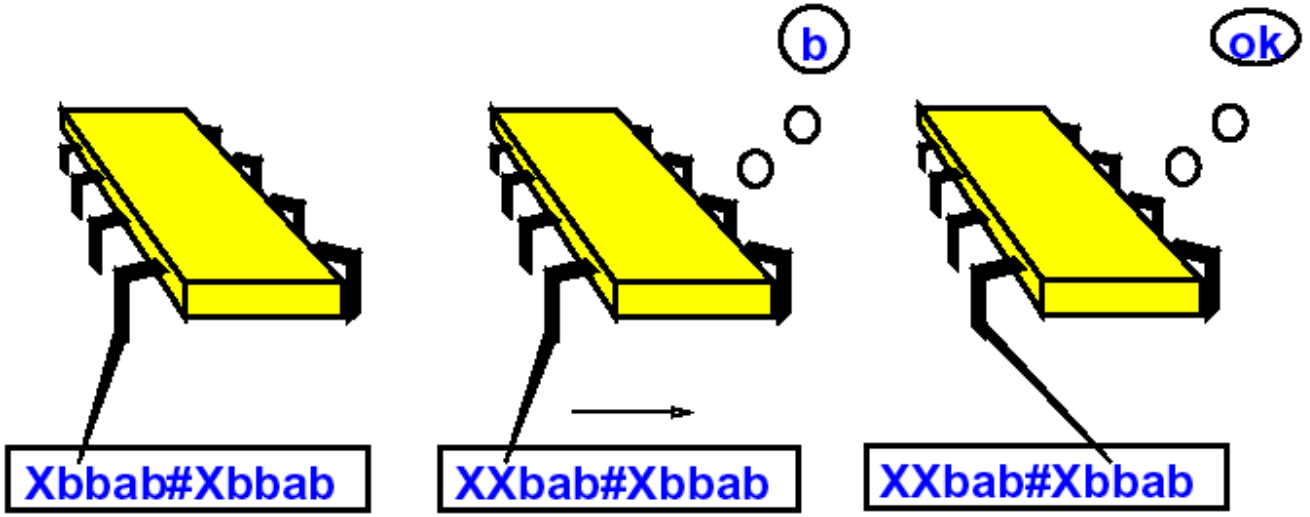
- tape head starts over leftmost symbol
- record symbol in control and overwrite X on tape
- scan right: reject if encounter blank ($\sqcup$) before $\#$
- when $\#$ encountered, move right one space
- if symbols don't match, reject.

Example (3)



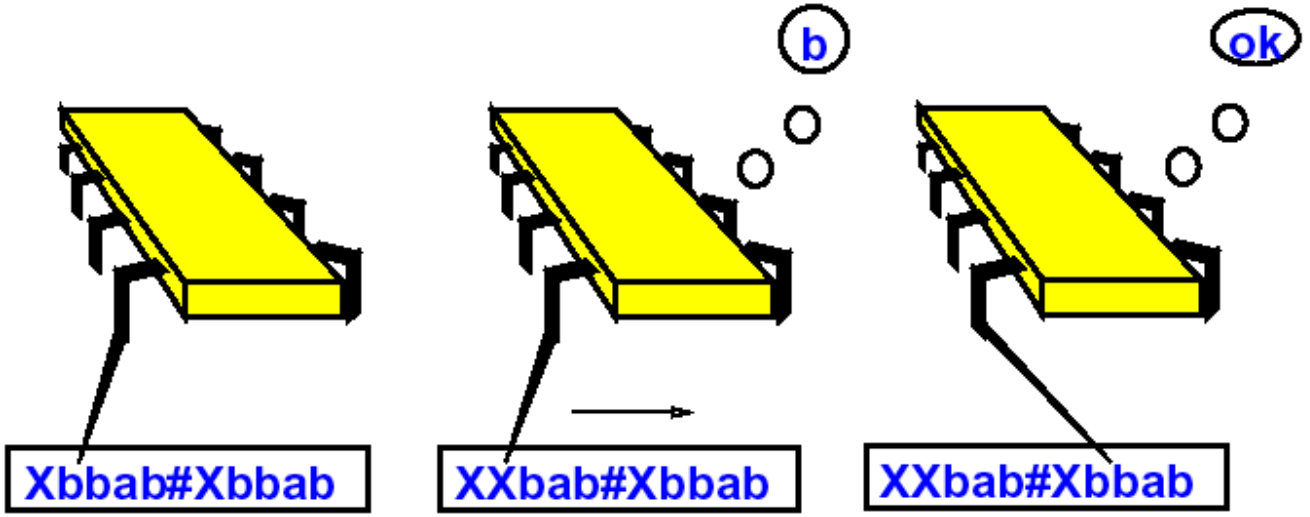
- write X, replacing current symbol (0 or 1)

Example (3)



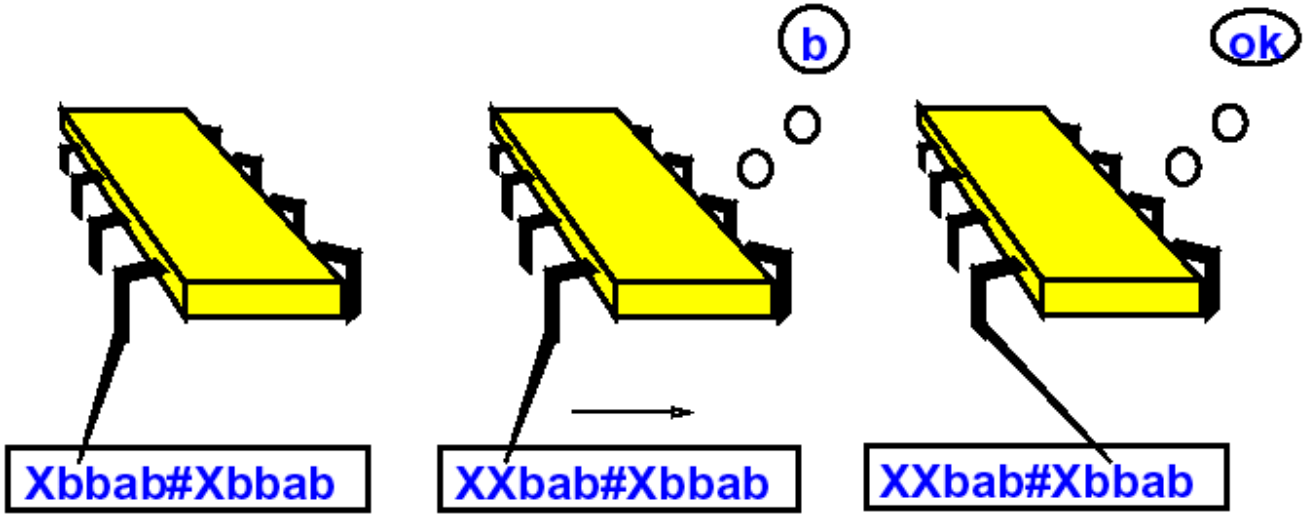
- write X, replacing current symbol (0 or 1)
- scan left, past $\#$ to X

Example (3)



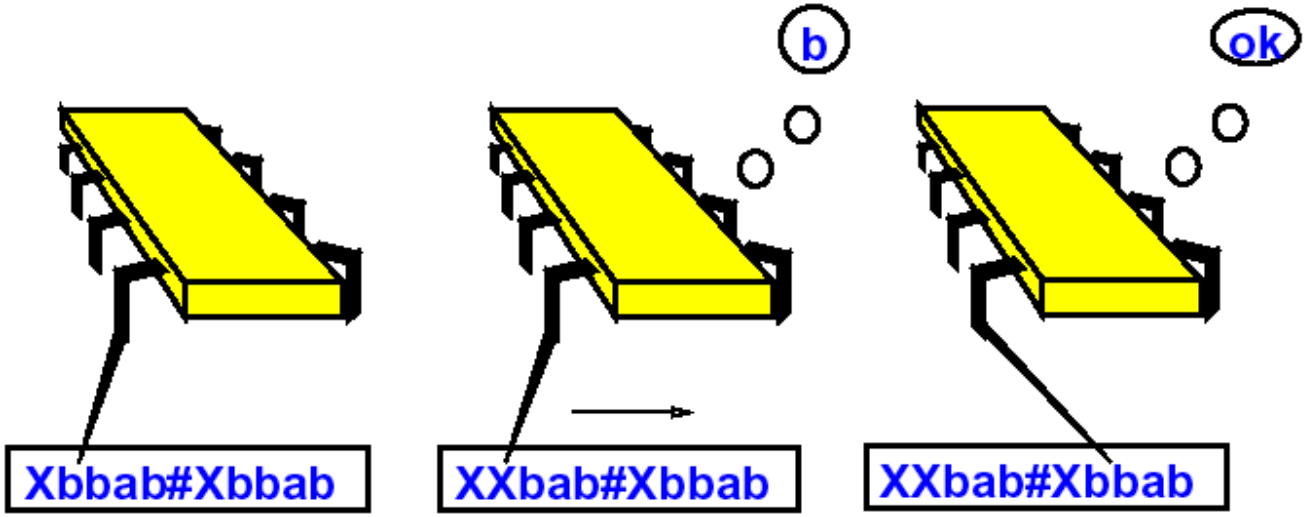
- write X, replacing current symbol (0 or 1)
- scan left, past $\#$ to X
- move one space right

Example (3)



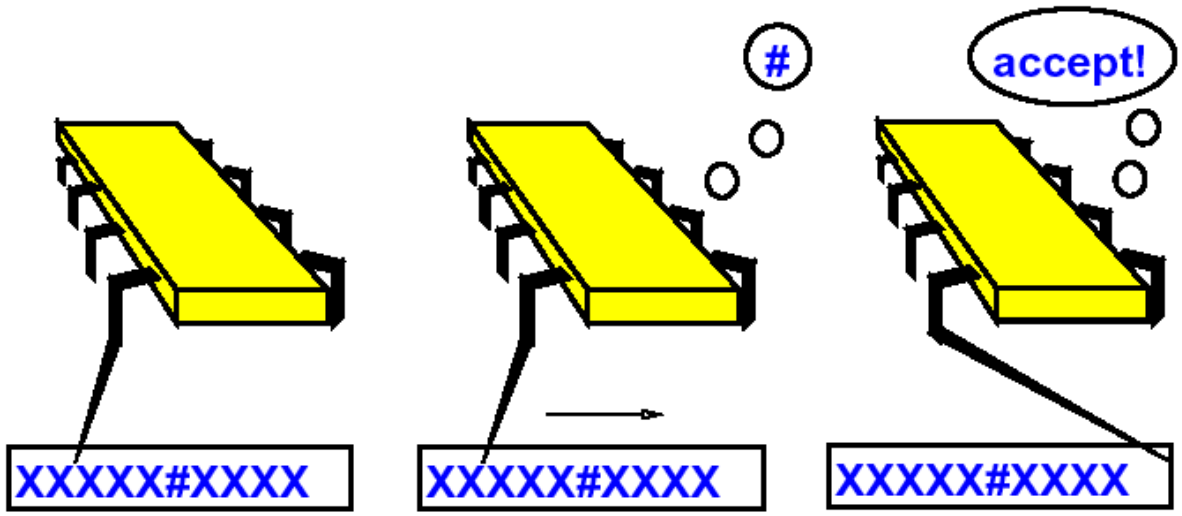
- write X, replacing current symbol (0 or 1)
- scan left, past $\#$ to X
- move one space right
- record symbol and writes X in its place

Example (3)



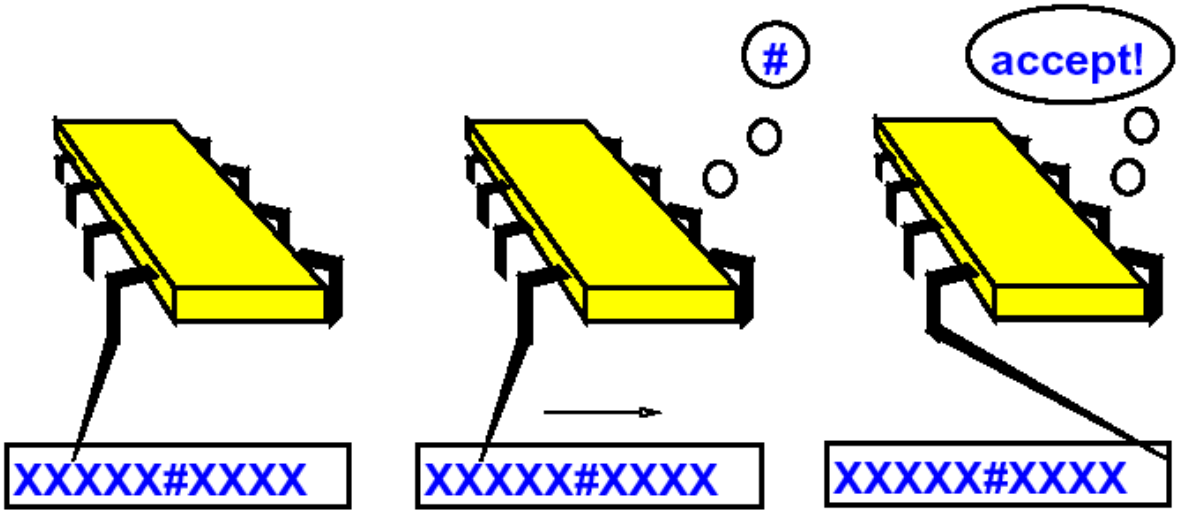
- write X, replacing current symbol (0 or 1)
- scan left, past $\#$ to X
- move one space right
- record symbol and writes X in its place
- scan right past $\#$ to X …

Example (4)



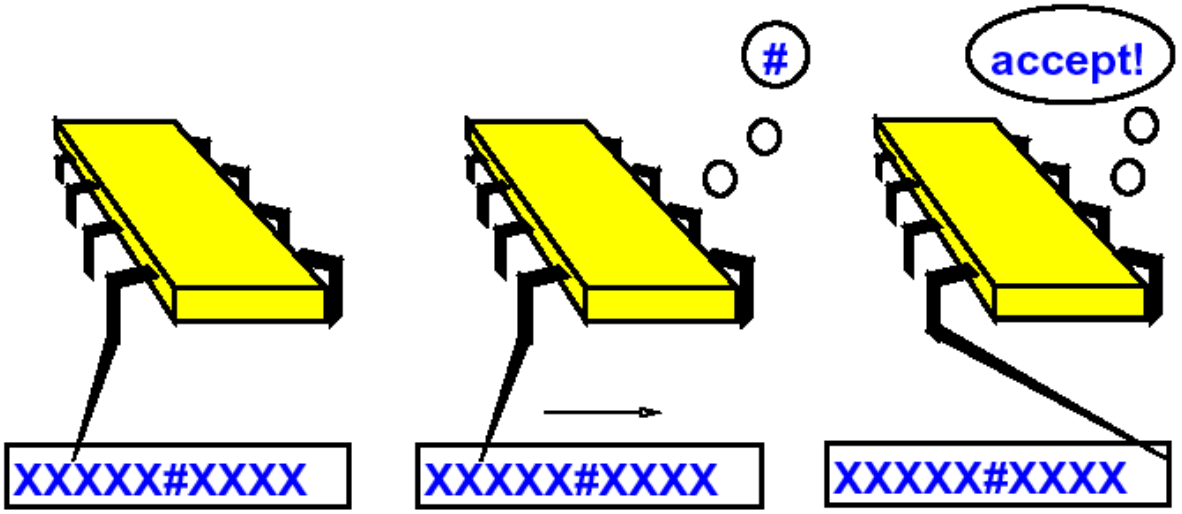
- finally, scan left

Example (4)



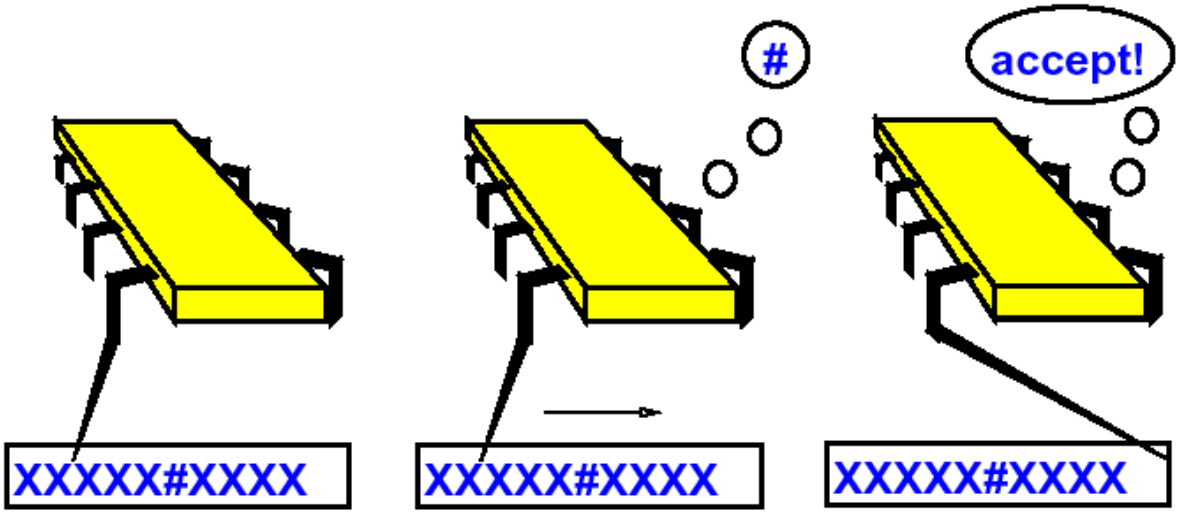
- finally, scan left
- if X encountered, keep going left

Example (4)



- finally, scan left
- if X encountered, keep going left
- if 0 or 1 encountered, reject

Example (4)



- finally, scan left
- if X encountered, keep going left
- if 0 or 1 encountered, reject
- when blank ($\sqcup$) encountered, accept

Formal Definition

We focus on the transition function

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

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

- in state q where head reads tape symbol a,

Formal Definition

We focus on the transition function

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

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

- in state q where head reads tape symbol a,
- the machine writes b, replacing the a,

Formal Definition

We focus on the transition function

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

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

- in state q where head reads tape symbol a,
- the machine writes b, replacing the a,
- enters state r,

Formal Definition

We focus on the transition function

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

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

- in state q where head reads tape symbol a,
- the machine writes b, replacing the a,
- enters state r,
- and moves the head left
  (this is what the L stands for).

Formal Definition (2)

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,

Formal Definition (2)

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, $\sqcup$

Formal Definition (2)

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 the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.

Formal Definition (2)

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 the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,

Formal Definition (2)

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 the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,

Formal Definition (2)

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 the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_a \in Q$ is the accept state, and

Formal Definition (2)

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 the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_a \in Q$ is the accept state, and
- $q_r \in Q$ is the reject state.

Formal Definition (3)

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

- an input of length n, $w = w_1 w_2 \ldots w_n \in \Sigma^*$

Formal Definition (3)

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

- an input of length n, $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on n leftmost tape squares

Formal Definition (3)

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

- an input of length n, $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on n leftmost tape squares
- rest of tape contains blanks $\sqcup$

Formal Definition (3)

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

- an input of length n, $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on n leftmost tape squares
- rest of tape contains blanks $\sqcup$
- read/write head is on leftmost square of tape

Formal Definition (3)

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

- an input of length n, $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on n leftmost tape squares
- rest of tape contains blanks $\sqcup$
- read/write head is on leftmost square of tape
- since $\sqcup \notin \Sigma$, leftmost blank indicates end of input.

Formal Definition (4)

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

When computation starts,

- M proceeds according to transition function δ.

Formal Definition (4)

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

When computation starts,

- M proceeds according to transition function δ.
- If M tries to move head beyond left-hand-end of tape, it doesn't move.

Formal Definition (4)

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

When computation starts,

- M 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,

Formal Definition (4)

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

When computation starts,

- M 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 M runs forever.

Configurations

One step of computation changes

- current state,

Configurations

One step of computation changes

- current state,
- current head position,

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

- For example, configuration $1011q_70111$ means:

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

- For example, configuration $1011q_70111$ means:
- Current state is q_7,

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

- For example, configuration $1011q_70111$ means:
- Current state is q_7,
- left hand side of tape is 1011,

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

- For example, configuration $1011q_70111$ means:
- Current state is q_7,
- left hand side of tape is 1011,
- right hand side of tape is 0111,

Configurations

One step of computation changes

- current state,
- current head position,
- and tape contents.

- For example, configuration $1011q_70111$ means:
- Current state is q_7,
- left hand side of tape is 1011,
- right hand side of tape is 0111,
- and head is on right hand side 0.

Configurations (2)

- If $\delta(q_i, b) = (q_j, c, L)$ then configuration uaq_ibv yields configuration uq_jacv.

Configurations (2)

- If $\delta(q_i, b) = (q_j, c, L)$ then configuration uaq_ibv yields configuration uq_jacv.
- If $\delta(q_i, b) = (q_j, c, R)$, then configuration uaq_ibv yields configuration $uacq_jv$.

Configurations (2)

- If $\delta(q_i, b) = (q_j, c, L)$ then configuration uaq_ibv yields configuration uq_jacv.
- If $\delta(q_i, b) = (q_j, c, R)$, then configuration uaq_ibv yields configuration $uacq_jv$.
- Special case (1): When head is at left end and tries to move left, it changes state and writes on tape but does not move, so if $\delta(q_i, b) = (q_j, c, L)$, configuration q_ibv yields q_jcv.

Configurations (2)

- If $\delta(q_i, b) = (q_j, c, L)$ then configuration uaq_ibv yields configuration uq_jacv.
- If $\delta(q_i, b) = (q_j, c, R)$, then configuration uaq_ibv yields configuration $uacq_jv$.
- Special case (1): When head is at left end and tries to move left, it changes state and writes on tape but does not move, so if $\delta(q_i, b) = (q_j, c, L)$, configuration q_ibv yields q_jcv.
- Special case (2): What happens when head is at right end? We let wq_i and $wq_i\sqcup$ denotes the same configuration, so moves to the right can now be accomodated.

More Configurations

We have

- starting configuration $q_0 w$

More Configurations

We have

- starting configuration $q_0 w$
- accepting configuration $w_0 q_a w_1$

More Configurations

We have

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

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

A Turing machine M accepts input an w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

- C_1 is start configuration of M on w,

Accepting a Language

A Turing machine M accepts input an w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

- C_1 is start configuration of M on w,
- each C_i yields C_{i+1},

Accepting a Language

A Turing machine M accepts input an w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

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

Accepting a Language

A Turing machine M accepts input an w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

- 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 accepted by M is called the language of M, and is denoted $L(M)$.

Enumerable Languages

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

Enumerable Languages (2)

On an input, w, a TM may

- accept

Enumerable Languages (2)

On an input, w, a TM may

- accept
- reject

Enumerable Languages (2)

On an input, w, a TM may

- accept
- reject
- loop (run forever)

Enumerable Languages (2)

On an input, w, a TM may

- accept
- reject
- loop (run forever)

Major concern: In general, we never know if TM will halt.

Decidable Languages

Definition: A TM decides a language if for every input $w \in \Sigma^*$, the TM halts.

Decidable Languages

Definition: A TM decides a language if for every input $w \in \Sigma^*$, the TM halts.

Namely the TM either reaches state q_a (in case $w \in L(M)$) or it reaches state q_r (in case $w \notin L(M)$), but it does not loop.

Decidable Languages

Definition: A TM decides a language if for every input $w \in \Sigma^*$, the TM halts.

Namely the TM either reaches state q_a (in case $w \in L(M)$) or it reaches state q_r (in case $w \notin L(M)$), but it does not loop.

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

Example

Here is a TM that accepts

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

- scan from left to right to check that input is $a^* b^* c^*$

Example

Here is a TM that accepts

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

- scan from left to right to check that input is $a^* b^* c^*$
- return to start of tape

Example

Here is a TM that accepts

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

- scan from left to right to check that input is $a^* b^* c^*$
- return to start of tape
- cross off one a and scan right until b occurs. Shuttle between b's and c's, crossing off one of each, until all b's are gone.

Example

Here is a TM that accepts

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

- scan from left to right to check that input is $a^* b^* c^*$
- return to start of tape
- cross off one a and scan right until b occurs. Shuttle between b's and c's, crossing off one of each, until all b's are gone.
- Restore the 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 (cont.)

Question: To implement algorithm, should be able to tell when a TM is at the left end of the tape.

Answer: Mark it with a special symbol.

Example (cont.)

Question: To implement algorithm, should be able to tell when a TM is at the left end of the tape.

Answer: Mark it with a special symbol.

An alternative, trickier way:

- remember current symbol
- replace current symbol with special symbol

Example (cont.)

Question: To implement algorithm, should be able to tell when a TM is at the left end of the tape.

Answer: Mark it with a special symbol.

An alternative, trickier way:

- remember current symbol
- replace current symbol with special symbol
- move left

Example (cont.)

Question: To implement algorithm, should be able to tell when a TM is at the left end of the tape.

Answer: Mark it with a special symbol.

An alternative, trickier way:

- remember current symbol
- replace current symbol with special symbol
- move left
- if special symbol still there, head is at start

Example (cont.)

Question: To implement algorithm, should be able to tell when a TM is at the left end of the tape.

Answer: Mark it with a special symbol.

An alternative, trickier way:

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

Example

Consider the element distinctness problem

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

Example

Consider the element distinctness problem

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

Verbally,

Example

Consider the element distinctness problem

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

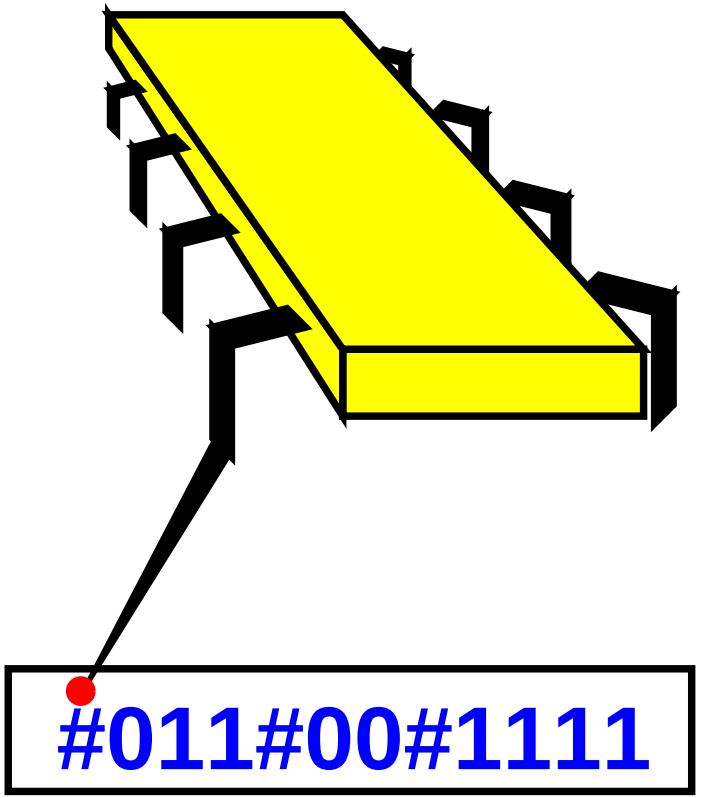
Verbally,

- List of strings in $\{0,1\}^*$ separated by $\#$'s.
- List is in language (& machine should accept) if all strings are different.

Element Distinctness – Solution

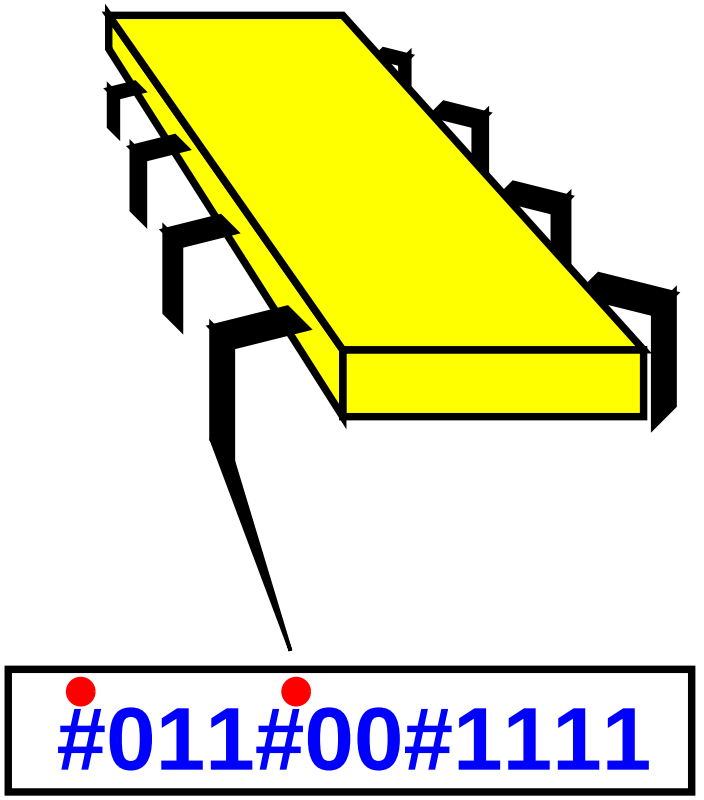
On input w

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



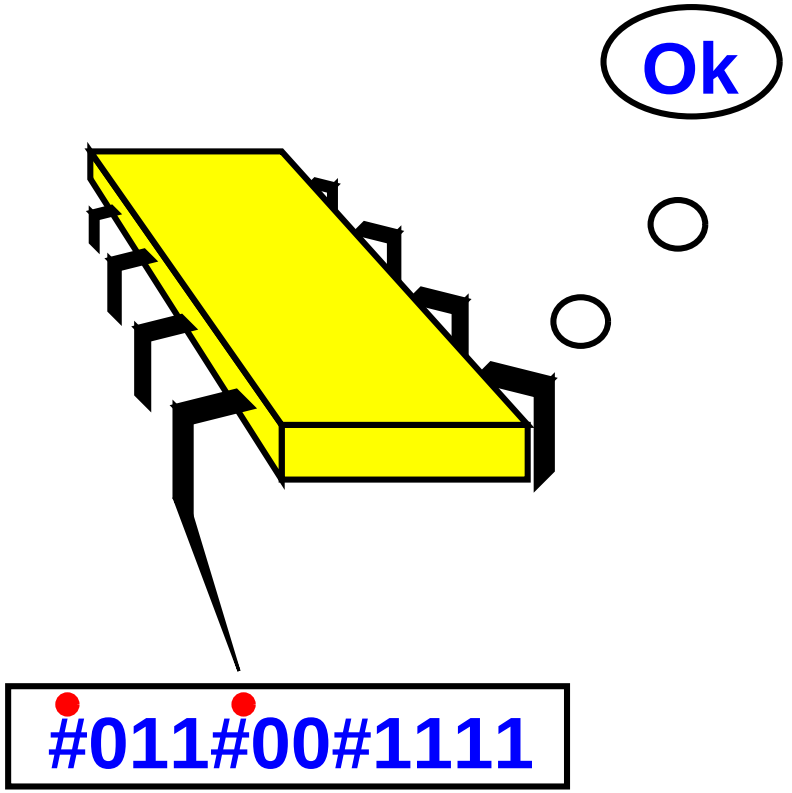
Element Distinctness – Solution (2)

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



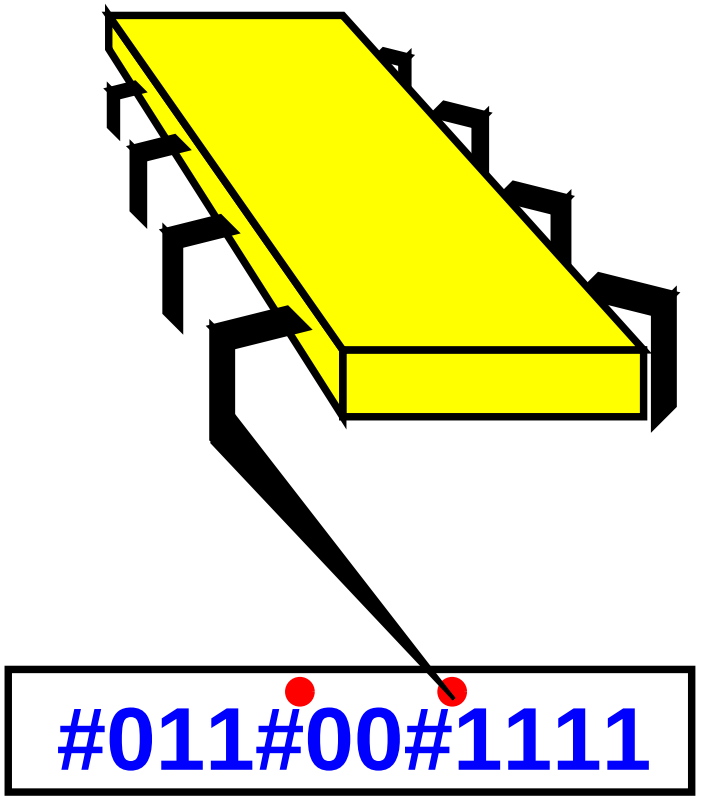
Element Distinctness – Solution (3)

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



Element Distinctness – Solution (4)

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



Element Distinctness – Solution (end)

Question: How do we "mark" a symbol?

Answer: For each tape symbol $\#$, add tape symbol $\bullet$ $\#$ to the tape alphabet Γ.

TMs Variants

Alternative Turing machine definitions abound.

For example, suppose the Turing machine head is allowed to stay put.

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

TMs Variants

Alternative Turing machine definitions abound.

For example, 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?)

TMs Variants

Alternative Turing machine definitions abound.

For example, 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?)

Important notion here: Two-way simulation (model A capable of simulating model B; model B capable of simulating model A).

Multitape Turing Machines

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

For the transition function:

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

Multitape Turing Machines

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

For the transition function:
$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, the expression
$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means

Multitape Turing Machines

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

For the transition function:
$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, the expression
$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means

- machine starts in state q_i
- if heads 1 through k reading $a_1, \ldots, a_k$,
- then machine goes to state q_j,

Multitape Turing Machines

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

For the transition function:
$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, the expression
$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means

- machine starts in state q_i
- if heads 1 through k reading $a_1, \ldots, a_k$,
- then machine goes to state q_j,
- heads 1 thought k write $b_1, \ldots, b_k$,

Multitape Turing Machines

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

For the transition function:
$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, the expression
$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means

- machine starts in state q_i
- if heads 1 through k reading $a_1, \ldots, a_k$,
- then machine goes to state 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 there is some multitape Turing machine that accepts it.

Equivalence

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

One direction is trivial.

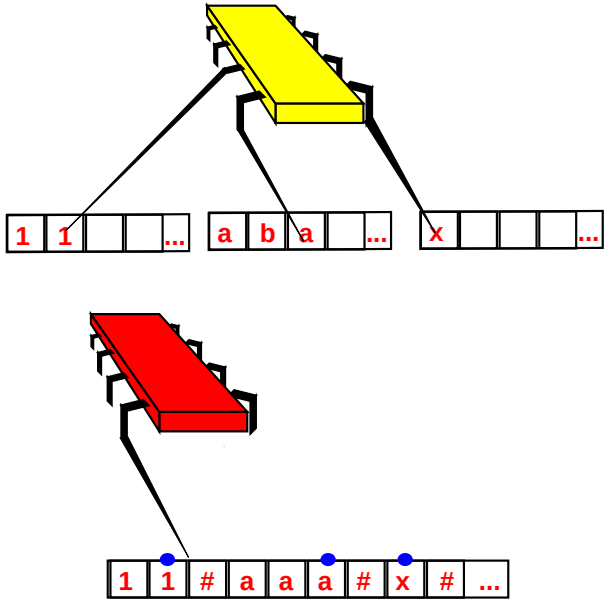
Equivalence

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

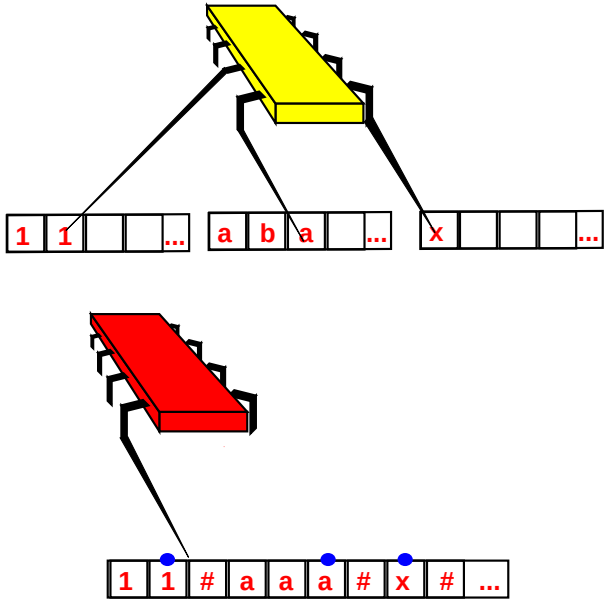
One direction is trivial.

To prove the other direction, we will show how to convert a multitape TM, M, into an equivalent single-tape TM, S.

Simulation

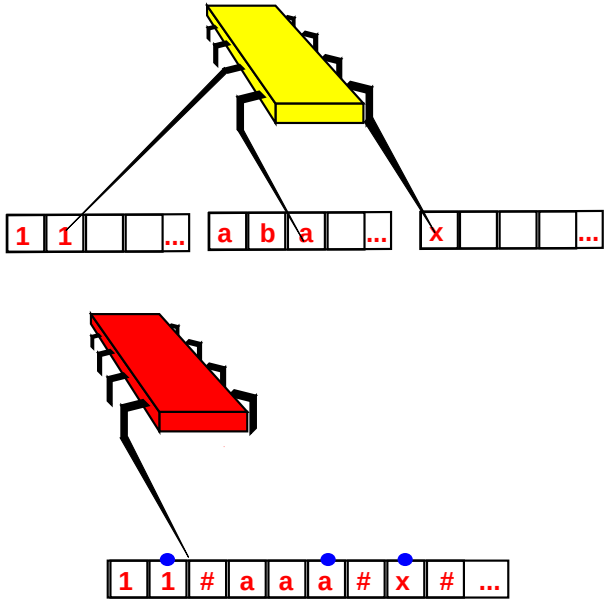


Simulation



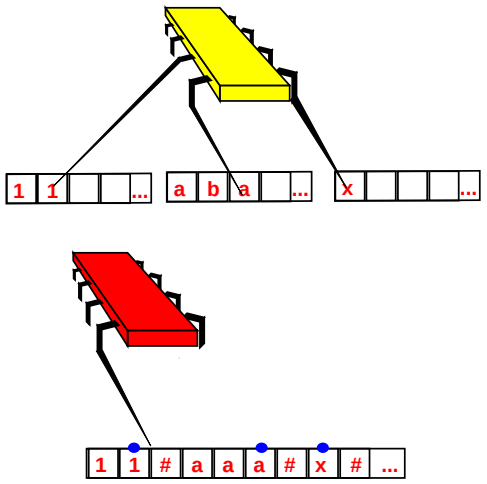
- S simulates k tapes of M by storing them all on a single tape with delimiter $\#$.

Simulation



- S simulates k tapes of M by storing them all on a single tape with delimiter $\#$.
- S marks the current positions of the k heads by placing $\bullet$ "above" the letters in current positions. It "knows" which tape the mark belongs to by counting (up to k) from the $\#$'s to the left.

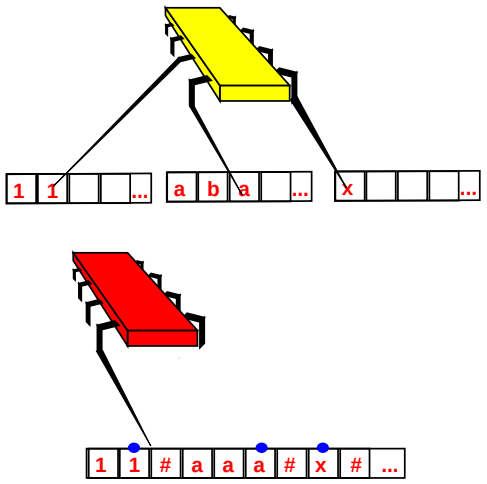
Simulation (2)



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

- writes on its tape $\# \dot{w}_1 \; w_2 \cdot w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$

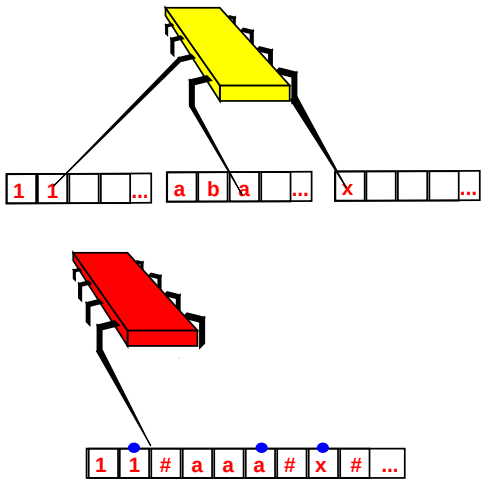
Simulation (2)



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

- writes on its tape $\# \dot{w}_1 \; w_2 \cdot w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$
- scans its tape from first $\#$ to $k+1$-st $\#$ to read symbols under "virtual" heads.

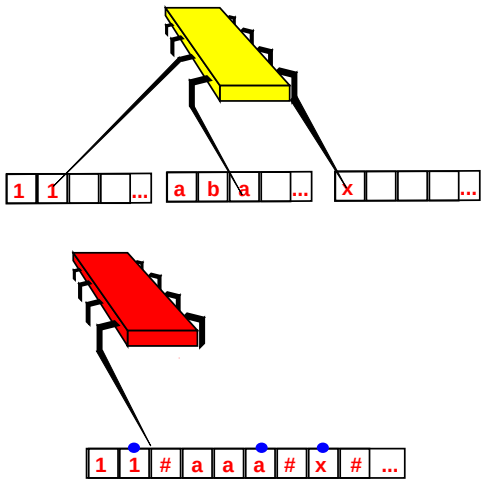
Simulation (2)



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

- writes on its tape $\# \dot{w}_1 \; w_2 \cdot w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$
- scans its tape from first $\#$ to $k+1$-st $\#$ to read symbols under "virtual" heads.
- rescans to write new symbols and move heads

Simulation (2)



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

- writes on its tape $\# \dot{w}_1 \; w_2 \cdot w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$
- scans its tape from first $\#$ to $k+1$-st $\#$ to read symbols under "virtual" heads.
- rescans to write new symbols and move heads
- S tries to move virtual head onto $\#$ when M is trying to move head onto unused blank square. S writes blank $\sqcup$ on tape, and shifts rest of the tape one square to the right.

Non-Deterministic Turing Machines

Transition function:

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

Non-Deterministic Turing Machines

Transition function:

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

- Computation is a tree.

Non-Deterministic Turing Machines

Transition function:

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

- Computation is a tree.
- Accepts if there is ($\exists$) an accepting branch.

Equivalence

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

Equivalence

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

One direction is trivial.

Equivalence

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

One direction is trivial.

To prove the other direction, we will show how to convert a non-deterministic TM, N, into an equivalent deterministic TM, D.

Simulating Non-Determinism

Basic idea:

- D tries all possible branches

Simulating Non-Determinism

Basic idea:

- D tries all possible branches
- If D finds any accepting branch, it accepts.

Simulating Non-Determinism

Basic idea:

- D tries all possible branches
- If D finds any accepting branch, it accepts.
- If all branches reject, D rejects.

Simulating Non-Determinism

Basic idea:

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

Simulating Non-Determinism (2)

N's computation is a tree.

- each tree branch is branch of N's non-deterministic computation

Simulating Non-Determinism (2)

N's computation is a tree.

- each tree branch is branch of N's non-deterministic computation
- each tree node is a configuration of N

Simulating Non-Determinism (2)

N's computation is a tree.

- each tree branch is branch of N's non-deterministic computation
- each tree node is a configuration of N
- root is starting configuration

Simulating Non-Determinism (2)

N's computation is a tree.

- each tree branch is branch of N's non-deterministic computation
- each tree node is a configuration of N
- root is starting configuration
- the number of children of each node is at most the number of N's states, denoted by b.

Simulating Non-Determinism (2)

N's computation is a tree.

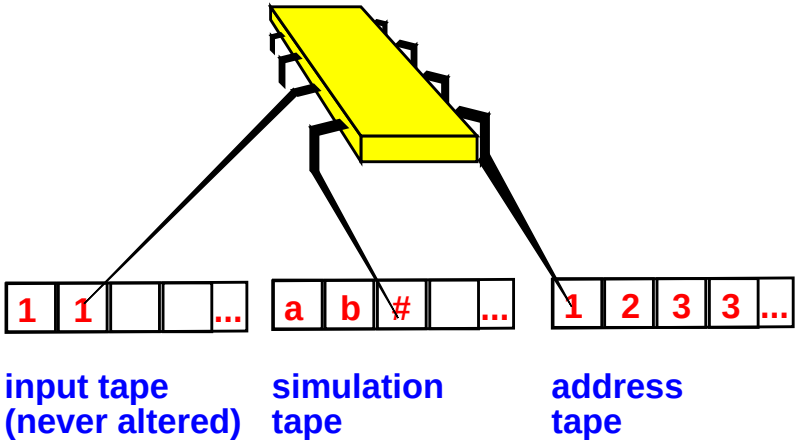
- each tree branch is branch of N's non-deterministic computation
- each tree node is a configuration of N
- root is starting configuration
- the number of children of each node is at most the number of N's states, denoted by b.
- depth-first search doesn't work (why?)

Simulating Non-Determinism (2)

N's computation is a tree.

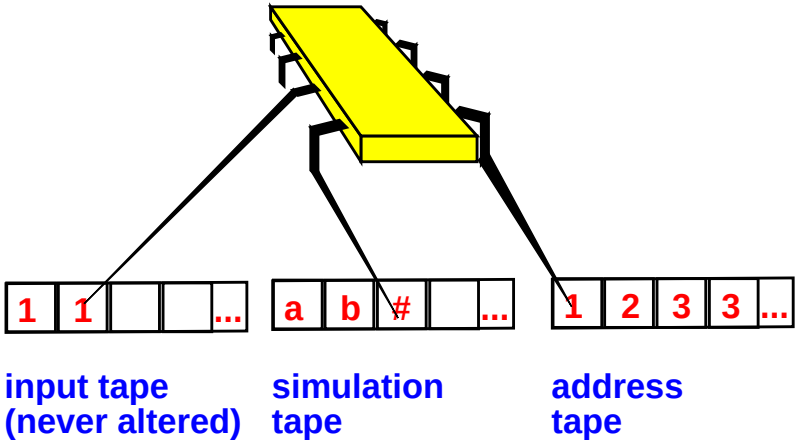
- each tree branch is branch of N's non-deterministic computation
- each tree node is a configuration of N
- root is starting configuration
- the number of children of each node is at most the number of N's states, denoted by b.
- depth-first search doesn't work (why?)
- breadth-first search does work, as we'll show.

Simulating Non-Determinism (3)



D has three tapes

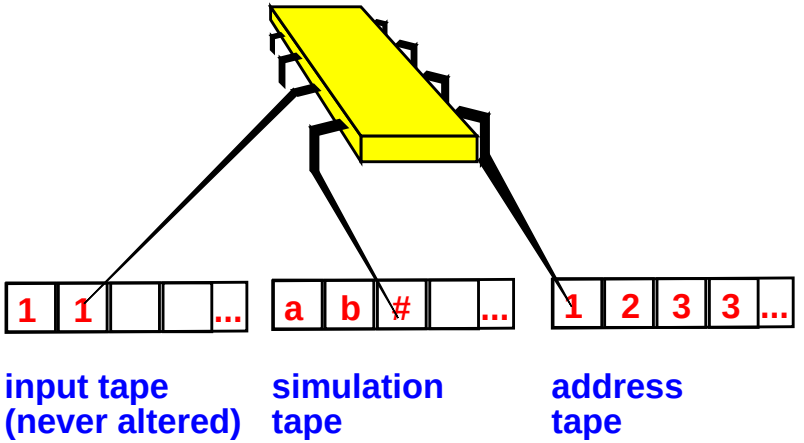
Simulating Non-Determinism (3)



D has three tapes

- the input tape is never altered

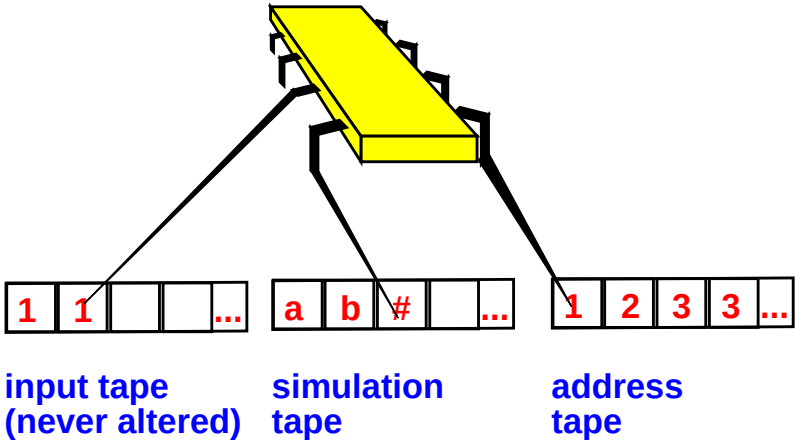
Simulating Non-Determinism (3)



D has three tapes

- the input tape is never altered
- the simulation tape is a copy of N's tape

Simulating Non-Determinism (3)



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.

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231
  - start at root

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231
  - start at root
  - take second child of root

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231
  - start at root
  - take second child of root
  - take third child of current node

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231
  - start at root
  - take second child of root
  - take third child of current node
  - take first child of current node

Simulating Non-Determinism (4)

The address tape:

- every node in the tree has at most b children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231
  - start at root
  - take second child of root
  - take third child of current node
  - take first child of current node
- ignore meaningless addresses (choices not available for configuration along branch)

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on a finite portion of one non-deterministic branch. On each choice, consult the next symbol on address tape. Accept if accepting configuration reached. Skip to next step if

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on a finite portion of 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

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on a finite portion of 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

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on a finite portion of 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

Simulating Non-Determinism (5)

- Initially, the input tape contains w, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate N on input w on a finite portion of 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 string on address tape with the lexicographically next string. Jump to Step 2 to simulate this branch of N's computation.

Deciders

Definition: A non-deterministic TM is a decider if on all inputs, all branches halt (in either state q_a or q_r).

Deciders

Definition: A non-deterministic TM is a decider if on all inputs, all branches halt (in either state q_a or q_r).

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

Deciders

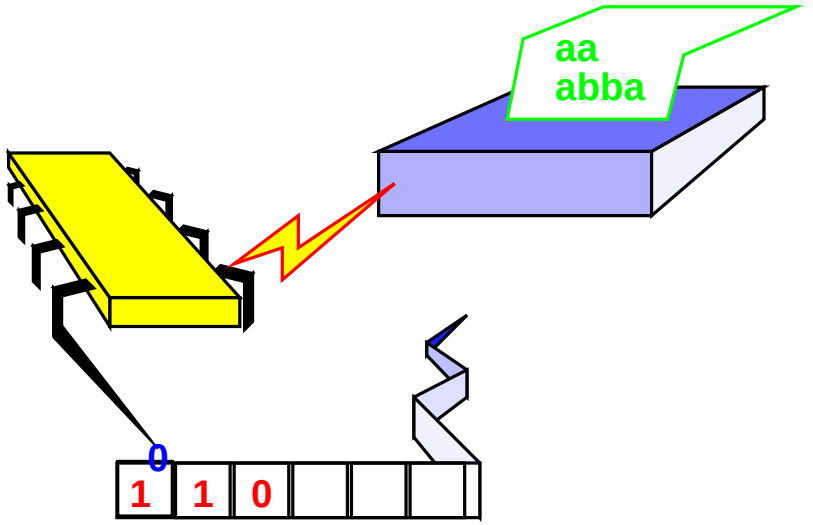
Definition: A non-deterministic TM is a decider if on all inputs, all branches halt (in either state q_a or q_r).

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

Definition (reminder): A language is decidable if some deterministic Turing machine decides it.

Enumerators

A language is enumerable if it is accepted by some Turing machine.
But why *enumerable*?

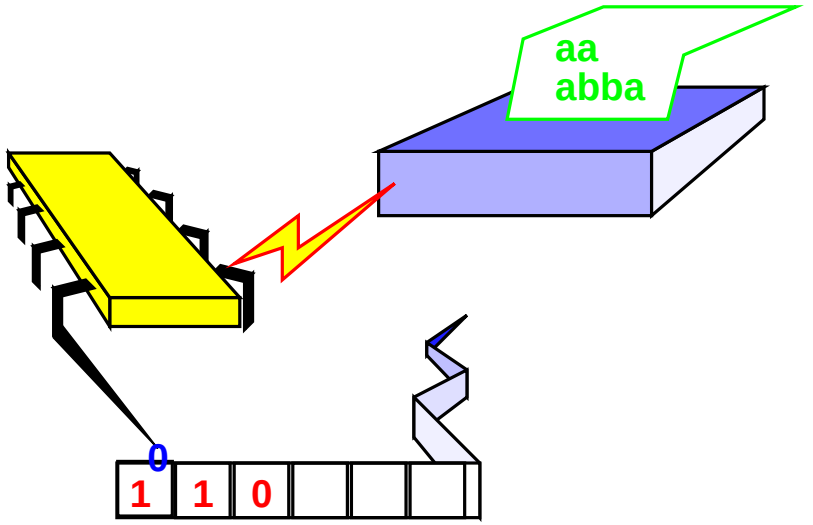


Definition: An enumerator is a TM with a printer.

- TM sends strings to printer

Enumerators

A language is enumerable if it is accepted by some Turing machine.
But why *enumerable*?

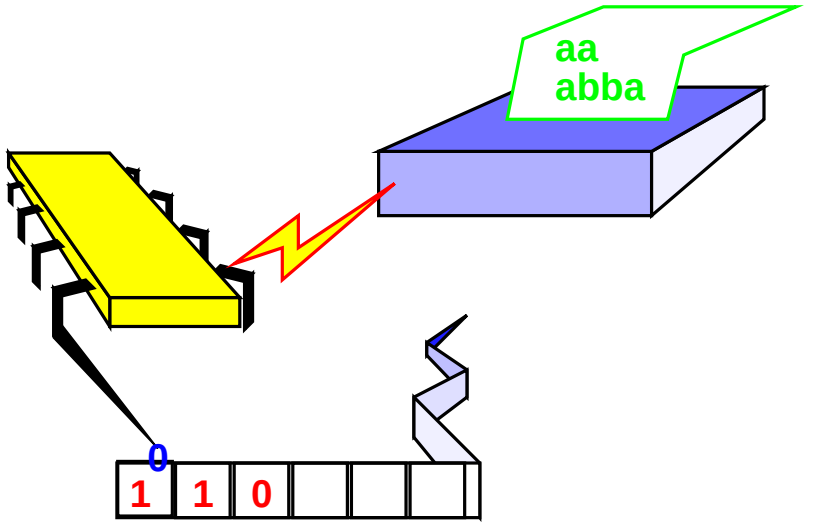


Definition: An enumerator is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings

Enumerators

A language is enumerable if it is accepted by some Turing machine.
But why *enumerable*?



Definition: An enumerator is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings
- the TM enumerates a language – all strings produced

Theorem

Theorem: A language is accepted by some Turing machine if and only if some enumerator enumerates it.

Theorem

Theorem: A language is accepted by some Turing machine if and only if some enumerator enumerates it.

Will 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, TM M

Theorem

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

On input w, TM M

- Runs E. Every time E outputs a string v, M compares it to w.

Theorem

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

On input w, TM M

- Runs E. Every time E outputs a string v, M compares it to w.
- If $v = w$, M accept.

Theorem

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

On input w, TM M

- Runs E. Every time E outputs a string v, M compares it to w.
- If $v = w$, M accept.
- If $v \neq w$, M continues running E.

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^* (*e.g.* strings in lexicographic order).
The enumerator, E

- repeat the following for $i = 1, 2, 3, \ldots$

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^* (*e.g.* strings in lexicographic order).
The enumerator, E

- repeat the following for $i = 1, 2, 3, \ldots$
- run M for i steps on each input $s_1, s_2, \ldots, s_i$.

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^* (*e.g.* strings in lexicographic order).
The enumerator, 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.

♣

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^* (*e.g.* strings in lexicographic order).
The enumerator, 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 with this procedure, each output is duplicated infinitely often.

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^* (*e.g.* strings in lexicographic order).
The enumerator, 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 with this procedure, each output is duplicated infinitely often.

How can this duplication be avoided?

Remarks

- Many models have been proposed for general-purpose computation.

Remarks

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.

Remarks

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (*e.g.* Java, Pascal, C, Scheme, Mathematica, Maple, Cobol,. . .) are equivalent.

Remarks

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (*e.g.* Java, Pascal, C, Scheme, Mathematica, Maple, Cobol,. . .) are equivalent.
- The notion of an algorithm is model-independent!

Remarks

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (*e.g.* Java, Pascal, C, Scheme, Mathematica, Maple, Cobol,. . .) are equivalent.
- The notion of an algorithm is model-independent!
- We don’t really care about Turing machines *per se*, we care about understanding computation.

What is an Algorithm?

- Informally

What is an Algorithm?

- Informally
  - a recipe

What is an Algorithm?

- Informally
  - a recipe
  - a procedure

What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program

What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it ⌒

What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it ⌒
- Historically,

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 (starting with Euclid's gcd algorithm), but

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 (starting with Euclid's gcd algorithm), but
  - not precisely defined until 20th century

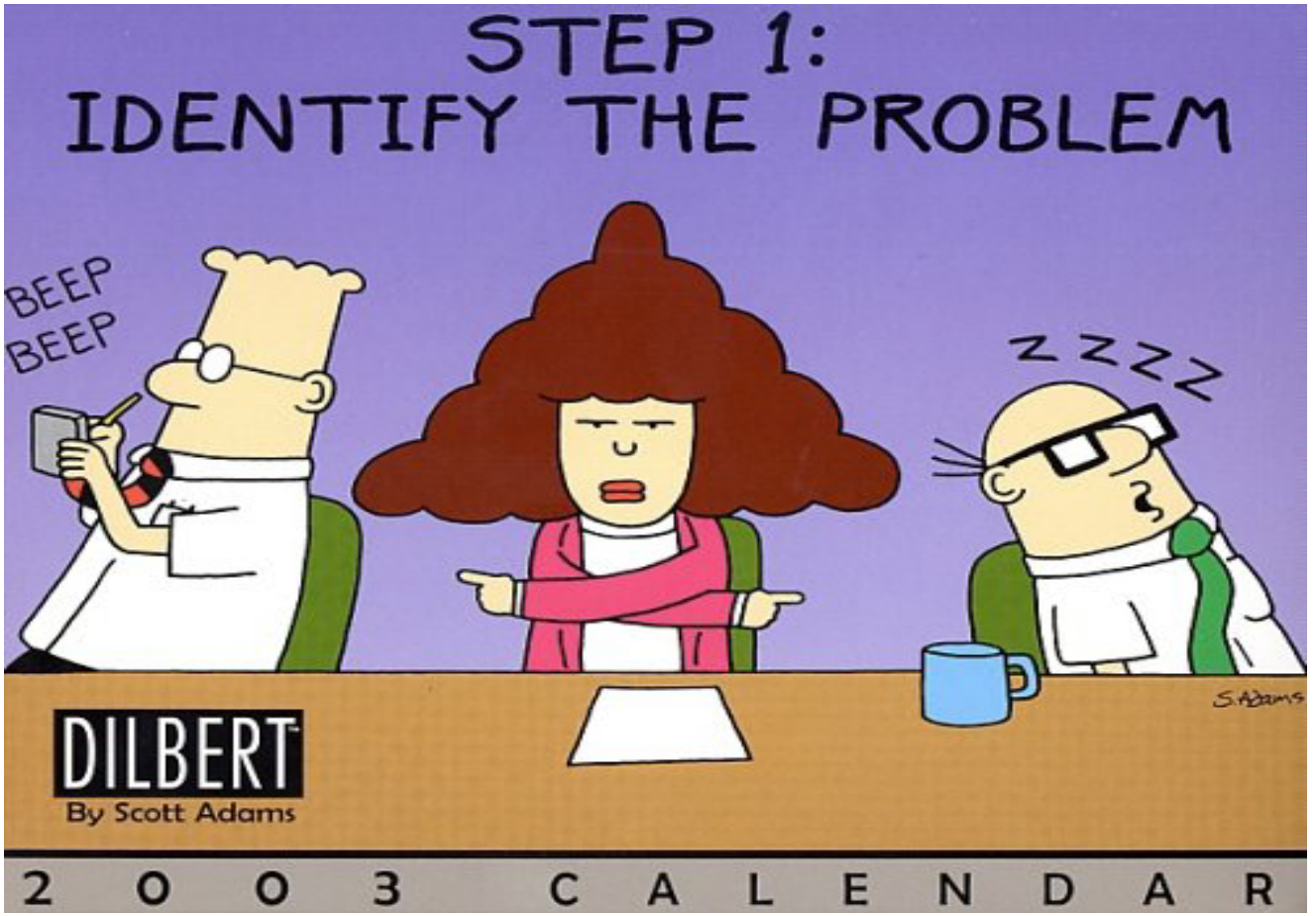
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it ⌒
- Historically,
  - notion has long history in Mathematics (starting with Euclid's gcd algorithm), but
  - not precisely defined until 20th century
  - informal notions rarely questioned,

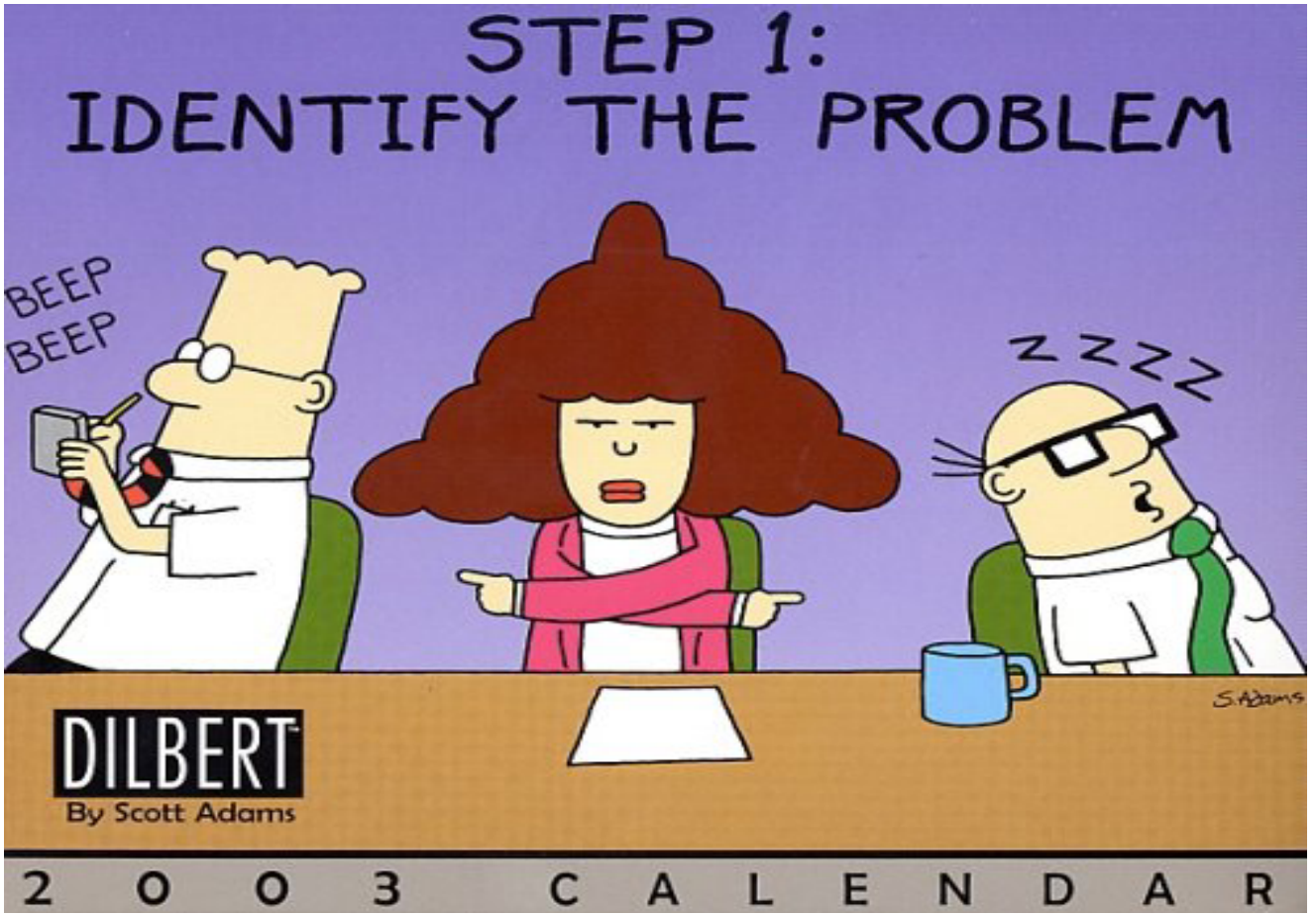
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it ⌒
- Historically,
  - notion has long history in Mathematics (starting with Euclid's gcd algorithm), but
  - not precisely defined until 20th century
  - informal notions rarely questioned,
  - still, they were insufficient

Dilbert's Problems



Dilbert's Problems



Too much beer last night? These are D. Hilbert's problems, not Dilbert's problems, we are supposed to talk about...

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
- challenge for the next (20th) century

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
- challenge for the next (20th) century
- the 10th problem directly concerned algorithms

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
- challenge for the next (20th) century
- the 10th problem directly concerned algorithms

November 2003: significant progress on the 6th problem.

Hilbert's 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
- challenge for the next (20th) century
- the 10th problem directly concerned algorithms

November 2003: significant progress on the 6th problem.

But for us, start with some background on the 10th ...

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.
- A polynomial is a sum of terms, *e.g.* $6x^3yz^2 + 3xy^2 - x^3 - 10$.

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.
- A polynomial is a sum of terms, *e.g.* $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.
- A polynomial is a sum of terms, *e.g.* $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.
- For example, $x = 5, y = 3$, and $z = 0$ is a root of the polynomial above.

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.
- A polynomial is a sum of terms, *e.g.* $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.
- For example, $x = 5, y = 3$, and $z = 0$ is a root of the polynomial above.
- Here, we are interested in integral roots, namely an assignment of integers to all variables.

Polynomials

- A term is a product of variables and a constant coefficient, *e.g.* $6x^3yz^2$.
- A polynomial is a sum of terms, *e.g.* $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.
- For example, $x = 5, y = 3$, and $z = 0$ is a root of the polynomial above.
- Here, we are interested in integral roots, namely an assignment of integers to all variables.
- Some polynomials have integral roots, some don't (*e.g.* $x^2 - 2$).

Hilbert's Tenth Problem

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

Actually, what he said (translated from German) was

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

Hilbert's Tenth Problem

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

Actually, what he said (translated from German) 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 Hilbert assumes that such an algorithm must exist, and someone "only" need find it.

Hilbert's Tenth Problem

- We now know no algorithm exists for this task.

Hilbert's Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn't have a formal notion of an algorithm.

Hilbert's Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn't have a formal notion of an algorithm.
- Intuitive notions work fine for constructing algorithms (we know one when we see it).

Hilbert's Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn't have a formal notion of an algorithm.
- Intuitive 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 notions appeared in 1936:

- λ-calculus of Alonzo Church

Church-Turing Thesis

Formal notions appeared in 1936:

- λ-calculus of Alonzo Church
- Turing machines of Alan Turing

Church-Turing Thesis

Formal notions appeared in 1936:

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

Church-Turing Thesis

Formal notions appeared in 1936:

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

The Church-Turing Thesis:

> "The intuitive notion of algorithms equals Turing machine algorithms".

Hilbert's Tenth Problem

In 1970, 23 years old Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, proved that no algorithm exists for testing whether a polynomial has integral roots

(a survey of the proof)

Reformulating Hilbert's Tenth Problem

Consider the language:

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

Reformulating Hilbert's Tenth Problem

Consider the language:

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

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

Reformulating Hilbert's Tenth Problem

Consider the language:

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

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

We now know it is not decidable, but it is enumerable!

Univariate Polynomials

Consider the simpler language:

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

Univariate Polynomials

Consider the simpler language:

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

Here is a Turing machine that accepts D_1.

Univariate Polynomials

Consider the simpler language:

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

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.

Univariate Polynomials (2)

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

Note that

Univariate Polynomials (2)

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

Note that

- If p has an integral root, the machine accepts.

Univariate Polynomials (2)

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

Note that

- If p has an integral root, the machine accepts.
- If not, M_1 loops.

Univariate Polynomials (2)

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

Note that

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

Univariate Polynomials (3)

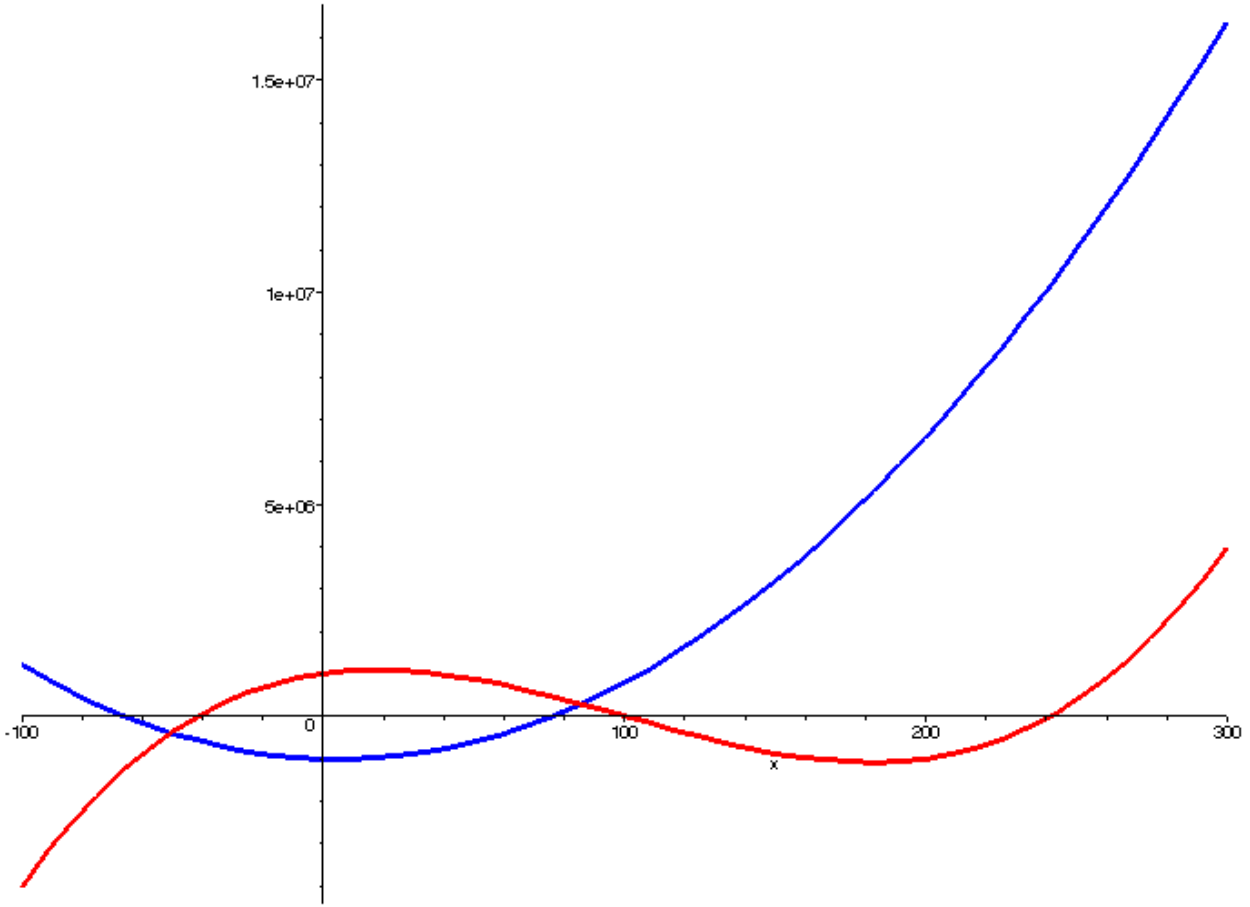
```
> f:=x->x^3-300*x^2+10000*x+1000000;
```

$$f := x \to x^3 - 300\,x^2 + 10000\,x + 1000000$$

```
> g:=x->200*x^2-2000*x-1000000;
```

$$g := x \to 200\,x^2 - 2000\,x - 1000000$$

```
> plot([f(x),g(x)],x=-100..300,color=[red,blue],thickness=3);
```



Univariate Polynomials (4)

In fact, D_1 is decidable.

Can show that all real roots of $p[x]$ lie inside interval

$$(-|kc_{max}/c_1|, |kc_{max}/c_1|) \ ,$$

where k is number of terms, c_{max} is max coefficient, and c_1 is high-order coefficient.

Univariate Polynomials (4)

In fact, D_1 is decidable.

Can show that all real roots of $p[x]$ lie inside interval

$$(-|kc_{max}/c_1|, |kc_{max}/c_1|) \ ,$$

where k is number of terms, c_{max} is max coefficient, and c_1 is high-order coefficient.

By Matijasevič theorem, such effective bounds on range of real roots cannot be computed for multivariable polynomials.

Wild Models

What about “unreasonable” models of computation?

Consider MUntel’s ℵ-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except

Wild Models

What about "unreasonable" models of computation?

Consider MUntel's ℵ-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.

Wild Models

What about "unreasonable" models of computation?

Consider MUntel's ℵ-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in $1/2$ second.

Wild Models

What about "unreasonable" models of computation?

Consider MUntel's ℵ-AXP© processor (to be released XMAS 2003).

- 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 ...

Wild Models

What about "unreasonable" models of computation?

Consider MUntel's $\aleph$-AXP$^{©}$ processor (to be released XMAS 2003).

- 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!

Wild Models

What about "unreasonable" models of computation?

Consider MUntel's $\aleph$-AXP$^{\copyright}$ processor (to be released XMAS 2003).

- 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$^{\copyright}$ decides any enumerable language!

Question: Does the $\aleph$-AXP$^{\copyright}$ invalidate the Church-Turing Thesis?

Encoding

- Input to a Turing machine is a string of symbols.

Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.

Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
- Need to choose an encoding for objects.

Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
- Need to choose an encoding for objects.
- Can often be done in many reasonable ways.

Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, 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 $\langle X \rangle$, its encoding.

Encoding

Consider strings representing undirected graphs.

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

Define the language:

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

High-Level Description

High-level description of a machine that decides

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

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

- select first node of G and mark it.

High-Level Description

High-level description of a machine that decides

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

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

- select first node of G and mark it.
- repeat until no new nodes marked:

High-Level Description

High-level description of a machine that decides

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

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.

High-Level Description

High-level description of a machine that decides

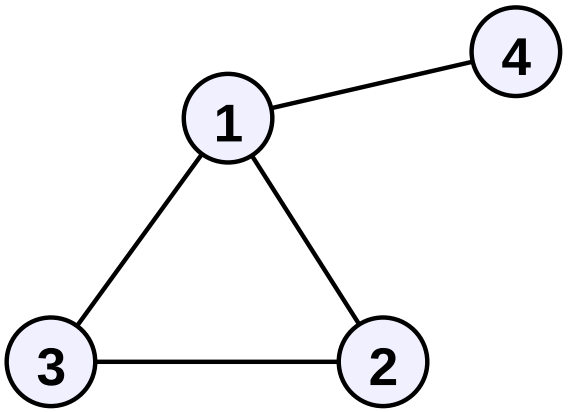
$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$$

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

More Details

On input M checks that input is valid graph encoding

- two lists
- first is list of numbers

More Details

On input M checks that input is valid graph encoding

- two lists
- first is list of numbers
- second is list of pairs

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)

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

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:
   - Scans list and "underlines" undotted node n_1.

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

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and "underlines" undotted node n_1.
   - M rescans and "underlines" dotted node n_2.

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

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and "underlines" undotted node n_1.
   - M rescans and "underlines" dotted node n_2.
   - M scans edges.
3. M scans the list of nodes. If all dotted, accept, otherwise reject.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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).
3. M scans the list of nodes. If all dotted, accept, otherwise reject.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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, goto Step 2.
3. M scans the list of nodes. If all dotted, accept, otherwise reject.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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, goto Step 2.
   - If not, check next edge. When no more edges, move underline to next dotted n_2.
3. M scans the list of nodes. If all dotted, accept, otherwise reject.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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, goto 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.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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, goto 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.

Detailed Algorithm

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

1. mark first node with a dot on leftmost digit.
2. loop:
   - 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, goto 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.