Introduction to Computability

<u>Question</u>: Using a counting argument, we have seen that there are functions that cannot be computed (or, in other words, problems that cannot be solved by any algorithm).
How can we exhibit a specific problem of this form?

<u>Solution</u>: we need a formal definition of algorithm

Let us start with something we know: Java
Can we show that there is no Java program that solves a specific problem?

<u>Hello-World problem</u>:

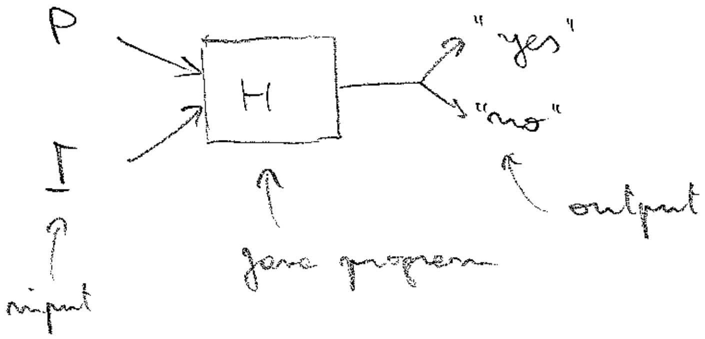
Your first Java program HW:

```
public class HW {
    public static void main (String [] args) {
        System.out.println ("Hello, world");
    }
}
```

The first 12 characters output by HW are "Hello, world".

Hello-world problem (HWP): given an arbitrary Java program P and an input I for P, does P(I) print "Hello, world" as its first 12 characters?

Consider a solution to HWP:



Does such a program H exist?

- we could scan P for println statements
- but, how do we know whether they are executed?

To give you an idea how difficult this can become, consider <u>Fermat's last theorem</u>:

The equation $x^n + y^n = z^n$ has no integer solution for $n \geq 3$.

For $n = 2$: a solution is $x = 3$, $y = 4$, $z = 5$

For $n \geq 3$: mathematicians have believed that the theorem is true, but no proof was found until recently (proof given by Wyles is very complex, and still under verification)

Consider a simple Java program P_1 that:

1) reads input n
2) for all possible x, y, z do
   if $(x^n + y^n = z^n)$
      println ("Hello, world");

Consider input $n = 3$: P_1 prints "Hello, world" only if F.L.T. is false, otherwise P_1 loops forever.

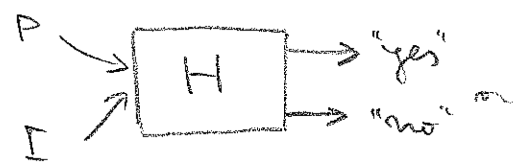
⇒ If we could solve HWP, we would also have proved or disproved F.L.T.

This would be too nice!! Where is the problem?

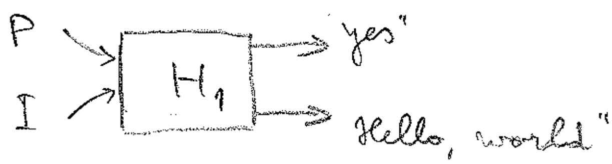
<u>Theorem</u>: There is no Java program H that decides HWP.

<u>Proof</u>: assume H exists and derive a contradiction.

Consider H:

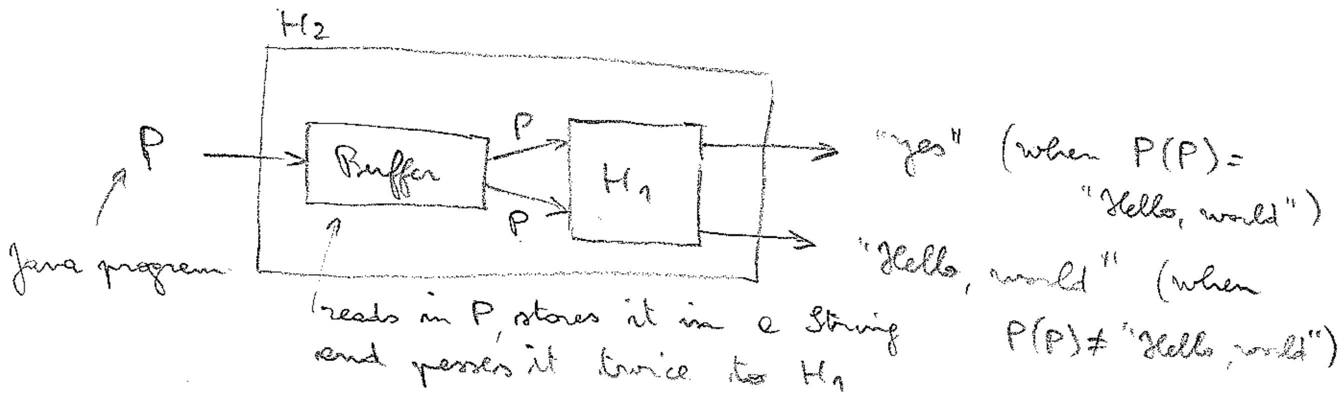


We modify H to H_1 s.t. H_1 prints "Hello, world" instead of "no"



(Note: we have to modify the println statements in H)

We modify H_1 to H_2, which takes only input P and feeds it to H_1 as both P and I:



Let us consider $H_2(P)$ when $P = H_2$.

suppose $H_2(H_2)$ = "yes" ⇒ $P(P)$ = "Hello, world"
suppose $H_2(H_2)$ = "Hello, world" ⇒ $P(P) \neq$ "Hello, world"

But $P = H_2$ ⇒ Contradiction ⇒ H, H_1, H_2 cannot exist! q.e.d.

We have shown HWP to be <u>undecidable</u>, i.e., there cannot be an algorithm (or a program) that solves it.

We can show that other problems are undecidable by "reducing" HWP to them

<u>Reductions</u>

<u>Foo-problem</u>: given a program R and its input z, does R ever call a function named foo while executing on input z.

Idea: we <u>reduce</u> the HWP to the foo-problem, i.e. we show that if it's possible to solve the foo-problem on (R, z), then we can solve HWP on (Q, y), for any program Q with input y.
Since HWP is undecidable, so is the foo-problem.

Suppose there is a program F that takes as input (R, z) and decides the foo-problem for (R, z).
We show how F can be used to construct H that decides HWP on input (Q, y)

Idea: apply modifications to Q

1) rename function foo in Q (if present) to pippo.
$\Rightarrow Q_1$
2) add a dummy function foo to Q_1 $\Rightarrow Q_2$
3) modify Q_2 to store all its output in some array A
$\Rightarrow Q_3$
4) modify Q_3 so that after every println statement it checks array A to see if "Hello, world" has been printed. If yes, then call function foo $\Rightarrow Q_4$

Note: We can write a Java program that takes as input a Java sourcefile and modifies it as specified above.

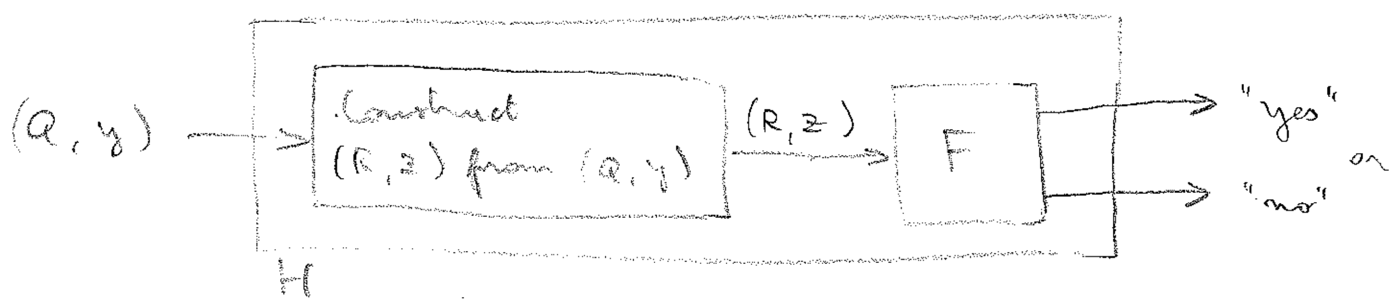
Let $R = Q_4$ and $z = y$

We have by construction:

$Q(y)$ prints "Hello, world" $\Leftrightarrow$
$R(z)$ calls function foo.

Hence, we can use F that solves foo-problem on $R(z)$ to construct H that solves HWP on $Q(y)$.

Schematically:



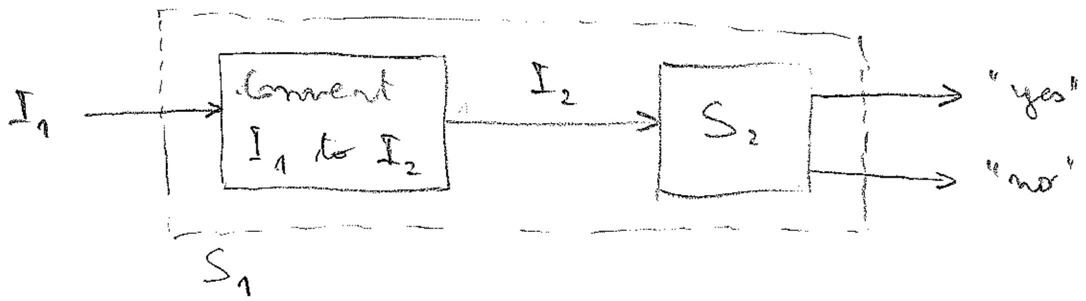
But since H does not exist, also F cannot exist.

Q.e.d

Showing undecidability by reduction from undecidable problem

Problem P_1 taking input I_1 known to be undecidable
 — " — P_2 — " — I_2 to show undecidable.

Reduction: convert I_1 to I_2 such that
$$P_1(I_1) = \text{"yes"} \quad \text{iff} \quad P_2(I_2) = \text{"yes"}$$
given solution program S_2 for P_2, we could obtain
 — " — S_1 for P_1



Since S_1 does exist, we obtain that S_2 cannot exist
$\Rightarrow$ P_2 is undecidable.

Existence of undecidable problems:

While it was tricky to show that a specific problem is undecidable, it is rather easy to show that there are infinitely many undecidable problems.

We use a counting argument:

- a problem P is a language over Σ (for some finite Σ)
  (the strings in the language represent those instances of P for which the answer is "yes")
  $\Rightarrow$ there are uncountably many problems
- an algorithm is a string over Σ' (for some finite Σ')
  $\Rightarrow$ there are countably many algorithms

$\Rightarrow$ There must be (uncountably many) problems for which there is no algorithm.

Turing Machines

Java (or C, Pascal, ...) programs are not well-suited to develop a theory of computation:

- run-time environment and run-time errors
- complex language constructs
- finite memory
- "state" of the computation is complicated to represent
- would need to show that the results for a specific programming language are in fact general

⇒ We resort to an abstract computing device, the Turing Machine (TM)

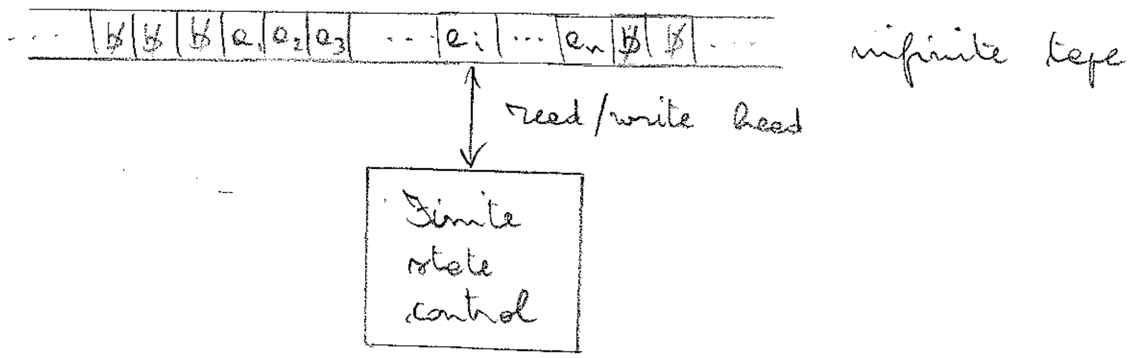
- simple and universal programming language
- state of computation is easy to describe
- unbounded memory
- can simulate any known computing device

Church-Turing hypothesis:

All reasonably powerful computation models are equivalent to TMs (but not more powerful).

⇒ TMs model anything we can compute.

The TM:



Programmed by specifying transitions

move depends on
- current state (finitely many)
- symbol under the tape head

effects of a move:
- new state
- write new symbol on tape cell under the head
- move head left/right/stay

Observations:

relationship to real computers: CPU ⟷ finite state control
memory ⟷ tape

"differences" (features lost in the abstraction)
- no random access memory
- limited instruction set

However: a TM can simulate a computer (with a cubic increase in running time — see book 8.6)

<u>Definition</u> a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \not b, F)$

Q ... set of states (finite) $q_0 \in Q$ ... initial state
Σ .. input alphabet (finite) Γ ... tape alphabet (finite)
$F \subseteq Q$ ... final states $\not b \in \Gamma$ ... blank symbol

Conditions: $\Sigma \subseteq \Gamma$, since input is written initially on tape
$\not b \in \Gamma - \Sigma$, since the rest of the tape is blank

Initially: - state q_0
- tape contains w surrounded by $\not b$
- tape head is at the leftmost cell of the input

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

$\delta(q, x) = (p, y, d)$ means that
if M is in state q and tape head is over symbol x,
then M - changes state to p
- replaces x by y on the tape
- moves tape head by one cell in direction d
(left for L, right for R, S for stay in place)

The TM is deterministic:
for each $\delta(q, x)$ we have at most one move
($\delta(q, x)$ could also be undefined)

<u>Acceptance</u>: w is accepted by TM M if M, when started with w on the tape, eventually enters a final state

We can assume that all final states are halting, i.e. no transition is defined for them

<u>Rejection</u>: - halts in non final state (i.e., no transition defined)
- never halts (infinite loop)

Difference between FA / PDA and TM:

FA/PDA scans over w and accepts/rejects when it has reached its end

TM can move back and forth over w and accepts/rejects when it halts or rejects if it loops forever

<u>Example:</u> $L = \{ w \#^+ w t \mid w \in \{0,1\}^+ , t \in \{0,1,\#\}^* \}$

initially

$\not{b}$	$\not{b}$	$\not{b}$	... w ...	#	...	#	... w ...	... t ...	$\not{b}$	$\not{b}$	...

TM idea: - remember (in the state) leftmost symbol, and erase it
- move to leftmost symbol after #'s
- if the two don't match, then reject
- otherwise replace the symbol by #, move left and start again

$M = (Q, \Sigma, \Gamma, \delta, q_0, \not{b}, F)$

$Q = \{q_0, q_1, \dots, q_7\}$ $\qquad F = \{q_7\}$

$\Sigma = \{0, 1, \#\}$ $\qquad \Gamma = \{0, 1, \#, \not{b}\}$

$\delta(q_0, 0) = (q_1, \not{b}, R)$ } Erase 0 and look for matching 0
$\delta(q_0, 1) = (q_2, \not{b}, R)$ } – " – 1 – " – 1

$\left.\begin{array}{l} \delta(q_1, 0) = (q_1, 0, R) \\ \delta(q_1, 1) = (q_1, 1, R) \\ \delta(q_1, \#) = (q_3, \#, R) \end{array}\right\}$ Skip over 0's and 1's, till # is found (remembering 0)

$\left.\begin{array}{l} \delta(q_2, 0) = (q_2, 0, R) \\ \delta(q_2, 1) = (q_2, 1, R) \\ \delta(q_2, \#) = (q_4, \#, R) \end{array}\right\}$ – " – (remembering 1)

$\delta(q_3, \#) = (q_3, \#, R)$
$\delta(q_3, 0) = (q_5, \#, L)$

Skip over #'s, look for 0, and replace it by #.
Note: if after #'s a 1 or a $\not b$ is found, M halts and rejects

$\delta(q_4, \#) = (q_4, \#, R)$
$\delta(q_4, 1) = (q_5, \#, L)$

as previous ones, replacing 0/1 with 1/0.

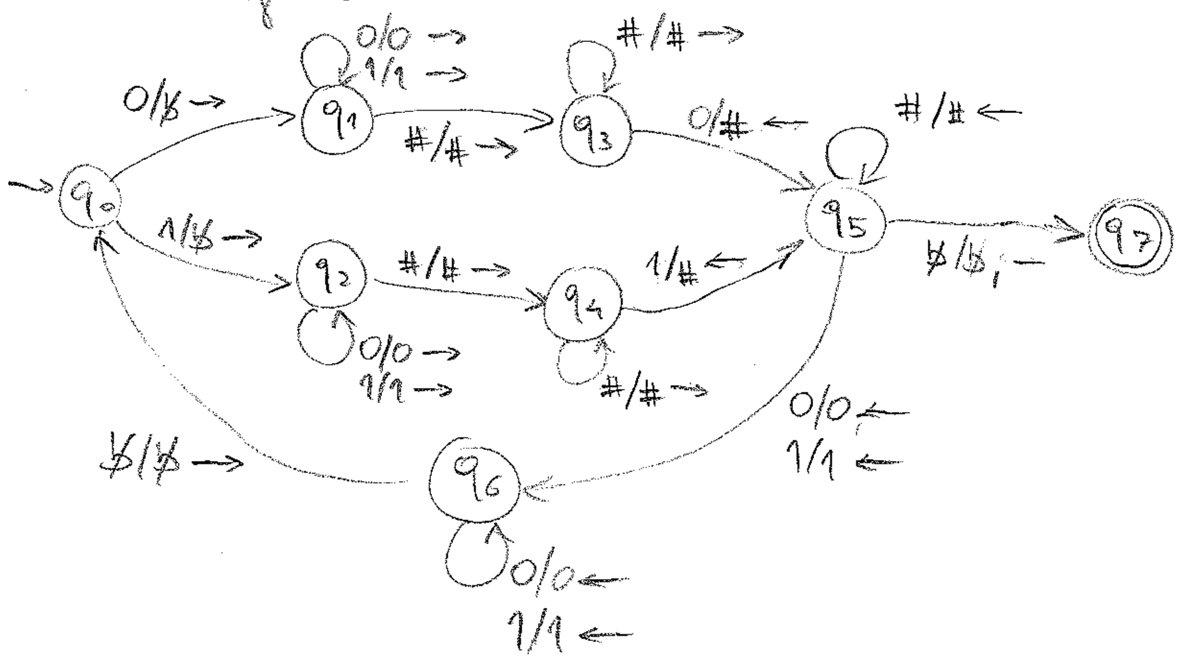
$\delta(q_5, \#) = (q_5, \#, L)$
$\delta(q_5, 0) = (q_6, 0, L)$
$\delta(q_5, 1) = (q_6, 1, L)$
$\delta(q_5, \not b) = (q_7, \not b, S)$

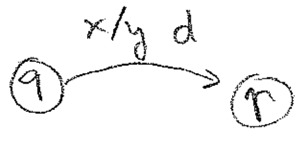
Move left skipping #'s.
If to the left of the #'s a 0 or 1 is found, move to q_6 to skip them also. If $\not b$ is found, accept

$\delta(q_6, 0) = (q_6, 0, L)$
$\delta(q_6, 1) = (q_6, 1, L)$
$\delta(q_6, \not b) = (q_0, \not b, R)$

Move left, skipping 0's and 1's, and restart again.

Transition diagram





represents $\delta(q, x) = (r, y, d)$

Instantaneous description (I.D.) or configuration of a TM

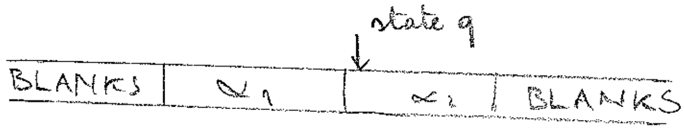
describes the current situation of TM and tape.

I.D. = $\alpha_1 q \alpha_2$ with $q \in Q$

$\alpha_1, \alpha_2 \in \Gamma^*$

means: - non-blank portion of tape contains $\alpha_1 \alpha_2$
- head is on leftmost symbol of α_2
- machine is in state q

corresponds to



Let $ID = \Gamma^* \times Q \times \Gamma^*$ be the set of instantaneous descriptions. We use a relation $\vdash \subseteq ID \times ID$ to describe the transitions of the TM.

Example: $q_0 01\#01 \vdash q_1 1\#01 \vdash 1 q_1 \#01 \vdash$
$\vdash 1\# q_3 01 \vdash 1 q_5 \#\#1 \vdash$
$\vdash q_5 1\#\#1 \vdash q_6 \not b 1\#\#1 \vdash$
$\vdash q_0 1\#\#1 \vdash \ldots \vdash$
$\vdash q_5 \not b \#\#\# \vdash q_7 \#\#\#$ ← accepts

Note: we can define $\vdash$ formally, making use of δ. [Exercise]

Making use of the closure $\vdash^*$ of $\vdash$ we can define the language accepted by a TM

Definition: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \not b, F)$ be a TM. Then the language $\mathcal{L}(M)$ accepted by M is

$\mathcal{L}(M) = \{ w \in \Sigma^* \mid q_0 w \vdash^* \alpha_1 q \alpha_2$ with $q \in F$ and $\alpha_1, \alpha_2 \in \Gamma^* \}$

Notes:

1) We have used TMs for language recognition, which in turn corresponds to solving decision problems

- We can, however, consider also TMs as computing functions:
- the output (result of the function) is left on the tape

2) The class of languages accepted by TMs are called recursively enumerable
   - for a string w in the language
      - the TM halts on input w in a final state
   - for a string w not in the language
      - the TM may halt in a non-final state, or
      - it may loop forever

Those languages for which the TM always halts (regardless of whether it accepts or not) are called recursive:
- these languages correspond to recursive functions
- TMs that always halt are a good model of algorithms and they correspond to decidable problems

Programming techniques for TMs

We present some notational conveniences that make it easier to write TM programs

Idea: use structured states and tape symbols

1) Storage in the state: ("CPU register")

Idea: state names are a tuple of the form

$[q, D_1, \ldots, D_k]$

D_i .. acts as stored symbol

q ... control portion of the state

Example: TM $M = (Q, \Sigma, \Gamma, \delta, q_s, \not b, F)$ for $L = 01^* + 10^*$

Idea: M remembers the first symbol and checks that it does not reappear

$Q = \{ [q_i, a] \mid i \in \{0, 1\}, a \in \{0, 1, \_\} \} =$

$\{ [q_0, \_], [q_0, 0], [q_0, 1], [q_1, \_], [q_1, 0], [q_1, 1] \}$

$\Sigma = \{0, 1\}$ $\qquad \Gamma = \{0, 1, \not b\}$

$q_s = [q_0, \not b]$ $\qquad F = \{[q_1, \_]\}$

Meaning of $[q_i, a]$

- control portion q_i:
  - q_0 ... M has not yet read its first symbol
  - q_1 ... M has read its first symbol
- data portion a: a is the first symbol read

transitions:

$$\delta([q_0, -], e) = ([q_1, e], e, R) \quad , \text{ for } e \in \{0, 1\}$$

... M remembers in $[q_1, e]$ that it has read e

$$\left.\begin{array}{l} \delta([q_1, 0], 1) = ([q_1, 0], 1, R) \\ \delta([q_1, 1], 0) = ([q_1, 1], 0, R) \end{array}\right\}$$ M moves right as long as it does not see the first symbol

$$\delta([q_1, e], \not b) = ([q_1, -], \not b, R) \quad , \text{ for } e \in \{0, 1\}$$

... M accepts when it reaches the first $\not b$

2) Multiple tracks:

Idea: view tape as having multiple tracks, i.e. each symbol in Γ has multiple components

	0	*	$\not b$	
...	1	0	0	..
	a	a	c	

the symbols on the tape are $\begin{bmatrix} 0 \\ 1 \\ a \end{bmatrix}, \begin{bmatrix} * \\ 0 \\ a \end{bmatrix}, \begin{bmatrix} \not b \\ 0 \\ c \end{bmatrix}$

Example: $L = \{ww \mid w \in \{0,1\}^+\}$

We first need to find midpoint, and then we can match corresponding symbols.

To find midpoint: we view tape as 2 tracks

			*		
0	1	1	0	1	1

← used to put markers on symbols

Hence: $\Gamma = \left\{ \begin{bmatrix} \not b \\ \not b \end{bmatrix}, \begin{bmatrix} \not b \\ 0 \end{bmatrix}, \begin{bmatrix} \not b \\ 1 \end{bmatrix}, \begin{bmatrix} * \\ 0 \end{bmatrix}, \begin{bmatrix} * \\ 1 \end{bmatrix} \right\}$

(note: we need no * over $\not b$)

We put markers on two outermost symbols and move them inwards:

$$\delta\left(q_0, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right]\right) = \left(q_1, \left[\begin{smallmatrix} * \\ i \end{smallmatrix}\right], R\right)$$
$$\delta\left(q_1, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right]\right) = \left(q_1, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right], R\right)$$

move right till end or first marked symbol

$$\delta\left(q_1, \left[\begin{smallmatrix} \not b \\ \not b \end{smallmatrix}\right]\right) = \left(q_2, \left[\begin{smallmatrix} \not b \\ \not b \end{smallmatrix}\right], L\right)$$
$$\delta\left(q_1, \left[\begin{smallmatrix} * \\ i \end{smallmatrix}\right]\right) = \left(q_2, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right], L\right)$$
$$\delta\left(q_2, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right]\right) = \left(q_3, \left[\begin{smallmatrix} * \\ i \end{smallmatrix}\right], L\right)$$

move rightmost mark one symbol to the left

$$\delta\left(q_3, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right]\right) = \left(q_3, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right], L\right)$$
$$\delta\left(q_3, \left[\begin{smallmatrix} * \\ i \end{smallmatrix}\right]\right) = \left(q_0, \left[\begin{smallmatrix} \not b \\ i \end{smallmatrix}\right], R\right)$$

move left till end or first marked symbol

Note: we have each of the above for $i \in \{0, 1\}$

At the end: head is over first symbol of second w, with a $*$ above it, in state q_0.

3) Subroutines / procedure calls

6/12/2006

Example: shifting over

given: $ID_1 = \alpha\, q_i\, x\, \beta$

want: $ID_2 = \alpha\, \Box\, q_i\, x\, \beta$

for $x \in \Gamma$
$\alpha, \beta \in \Gamma^*$
$\Box \in \Gamma$

Subroutine for shifting over can be used repeatedly to create space in the middle of the tape

E.g. to implement a counter

$\$0\$ \to \$1\$ \to \$\Box 1\$ \to \$01\$ \to \$10\$ \to$
$\to \$11\$ \to \$\Box 11\$ \to \$011\$ \to \dots$

Procedure call: $\delta(q_i, x) = ([p, x], [\overset{j}{\square}], R) \quad , \forall x \in \Gamma$

- remember return state q_j and erased symbol x
- state p calls procedure

Procedure p for shifting

1) shift 1 cell to the right

$$\delta([p, x], y) = ([p, y], x, R) \qquad \forall x, y \in \Gamma \text{ with } y \neq \not{b}$$

2) till we have reached end of β

$$\delta([p, y], \not{b}) = (r, y, L) \qquad \forall y \in \Gamma$$

3) return to calling point by moving left

$$\delta(r, y) = (r, y, L) \qquad \forall y \neq [\overset{j}{\square}]$$

4) exit and return to state q_j

$$\delta(r, [\overset{j}{\square}]) = (q_j, \square, R)$$

In fact, we can implement arbitrary complex procedures, with any kind of parameter passing

$\boxed{\text{Exercise:}}$ redesign the TMs you have seen so far to take advantage of storage in the state, multiple tracks, and subroutines

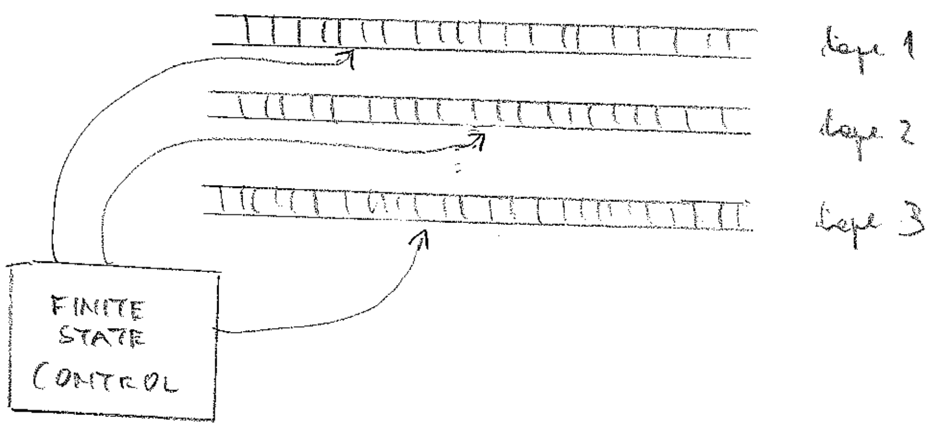
Extensions to the basic TM

Note: if the TM seen so far can compute all that can be computed, then it should not become more expressive by extending it

We consider two extensions: - multiple tapes
- non determinism

and show that both can be captured by the basic T.M.

1) Multi-tape T.M.



Initially: input w is on tape 1 with tape-head on the leftmost symbol. Other tapes are all blank.

Transitions: specify behaviour of each head independently

$$\delta(q, x_1, \ldots, x_k) = (p, (y_1, d_1), \ldots, (y_k, d_k))$$

x_i ... symbol under head i

y_i ... new symbol written to head i

d_i ... direction in which head i moves

To simulate k-tape TM M_k with a 1-tape TM M_1, we use $2k$ tracks in M_1: for each tape of M_k

- one track of M_1 to store tape content
- one track of M_1 to mark head position with $*$

A	B	A	C	B	A		tape 1
			*				head 1
0	0	1	1	1	0		tape 2
	*						head 2
b	b	a	b	a	b		tape 3
					*		head 3

Each transition of M_k is simulated by a series of transitions of M_1: $\delta(q, x_1, \dots, x_k) = (p, (y_1, d_1), \dots, (y_k, d_k))$

- start at leftmost head position marker
- sweep right and remember in appropriate "CPU registers" the symbols x_i under each head (note: these are exactly k, and hence finitely many)
- knowing all x_i's, sweep left, change each x_i to y_i, and move the marker for tape i according to d_i

Note: M_1 needs to remember always how many of the k heads are to its left (uses an additional CPU-register)

The final states of M_1 are those that have in the state-component a final state of M_k.

We can verify that we can construct M_1 so that $\mathcal{L}(M_1) = \mathcal{L}(M_k)$

(details are straightforward, but cumbersome)

Simulation speed:

Note: - enhancements do not affect the expressive power of a TM
- they do affect its efficiency

Definition: a TM is said to have running time $T(n)$ if it halts within $T(n)$ steps on all inputs of length n.

Note: $T(n)$ could be infinite

Theorem: If M_k has running time $T(n)$, then M_1 will simulate it with running time $O(T(n)^2)$.

Proof: Consider input w of length n.
- M_k runs at most $T(n)$ time on it.
- At each step, leftmost and rightmost heads can drift apart by at most 2 additional cells.
- It follows that after $T(n)$ steps the k heads cannot be more than $2 \cdot T(n)$ apart, and M_k uses $\leq 2 \cdot T(n)$ tape cells

Consider M_1:
- makes two sweeps for each transition of M_k
- each sweep takes at most $O(T(n))$
- number of transitions of M_k is $\leq T(n)$

It follows that the total running time is $O(T(n)^2)$.

2) Nondeterministic TMs (NTM)

In a (deterministic) TM, $\delta(q, x)$ is unique or undefined.

In a NTM, $\delta(q, x)$ is a finite set of triples

$$\delta(q, x) = \{(p_1, y_1, d_1), \ldots, (p_k, y_k, d_k)\}$$

At each step, the NTM can non-deterministically choose which transition to make.

As for other ND devices: a string w is accepted if the NTM has at least one execution leading to a final state.

__Example:__ $\Sigma = \{0, 1, \ldots, 9\}$

$L = \{ w \in \Sigma^* \mid$ a 0 appears i positions to the left of some i in w, with $0 < i \leq 9 \} =$

$= \{ w \in \Sigma^* \mid \exists j > 0$ s.t. $w_{j - w_j} = 0 \}$

(w_i indicates the i-th character of w)

Ex. $02146 \notin L$

$58108554421 \ldots$

$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \ldots$

$w_7 = 4$

$w_3 = w_{7-4} = 0$

NTM N s.t. $L(M) = L$

$Q = \{q_0, f, [p, 0], [p, 1], \ldots, [p, 9]\}$

$F = \{f\}$

$\Gamma = \{0, 1, \ldots, 9, \not b\}$

Idea for N: - scan w from left to right,
- guess at some $w_j = i$,
- store i in CPU register, and
- move i steps left to find 0

Transitions:

- $\delta(q_0, 0) = \{(q_0, 0, R)\}$ (since $w_j > 1$)
- $\forall i > 0 : \delta(q_0, i) = \{(q_0, i, R), \underset{\text{guess}}{([q, i], i, L)}\}$
- $\forall i \geq 2, \forall x \in \Gamma : \delta([q, i], x) = \{[q, i-1], x, L)$
- accepting: $\delta([q, 1], 0] = \{(f, 0, R\})$

Execution traces on input $w = 103332$

$q_0 103332 \vdash 1 q_0 03332 \vdash 10 q_0 3332 \vdash 103 q_0 332 \vdash$
$\vdash 10 [q, 3] 3332 \vdash 1 [q, 2] 03332 \vdash [q, 1] 103332$

$\Rightarrow$ reject

$q_0 103332 \vdash^* 1033 q_0 32 \vdash 103 [q, 3] 332 \vdash$
$\vdash 10 [q, 2] 3332 \vdash 1 [q, 1] 03332 \vdash 10 f 3332$

$\Rightarrow$ accept

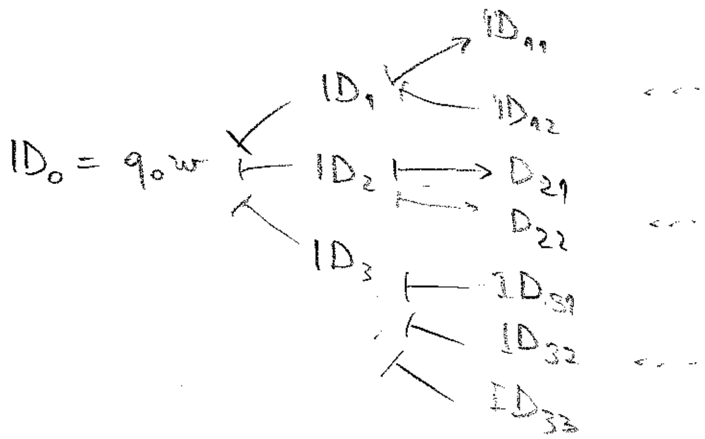
11/12/2006

__Theorem__: Let N be a NTM. Then there exists a DTM D s.t. $\mathcal{L}(D) = \mathcal{L}(N)$

__Proof__: Given N and w, we show how a multi-tape DTM can simulate the execution of N on input w.
We can then convert the multi-tape DTM to a single-tape DTM

Idea for the simulation:

Consider the execution tree of N on w



DTM D will perform a breadth-first search of the execution tree, systematically enumerating the IDs, until it finds an accepting one.

We use <u>two tapes</u>:

tape 2: is for working

tape 1: contains a sequence of ID's of N in BFS order

- $*$ used to separate two ID's
- $\hat{*}$ marks next ID to be explored
  - ID's to the left of $\hat{*}$ have been explored
  - ID's to the right of $\hat{*}$ are still to be explored
- initially, only $ID_0 = q_0 w$ is on the tape
- we can use multiple tracks for convenience

Algorithm: repeat the following steps

step 0: examine current ID_c (the one after $\hat{*}$) and read q, a from it

if $q \in F$, then accept and halt

step 1: let $\delta(q, a)$ have k possible transitions

- copy ID_c onto tape 2
- make k new copies of ID_c and place them at the end of tape 1

step 2: modify the k copies of ID_c on tape 1 to become the k possible outcomes of $\delta(q, a)$ on ID_c

step 3: move $\hat{*}$ right past ID_c.
clean up tape 2
return to step 0

It is possible to verify:
- the above steps can all be implemented in a DTM.
- the construction is correct, i.e. $w \in \mathcal{L}(D)$ iff $w \in \mathcal{L}(N)$

Evolution of tape 1:

1) $\hat{*}\ ID_0 *$
2) $\hat{*}\ ID_0 * ID_0 * ID_0 * ID_0 *$
3) $\hat{*}\ ID_0 * ID_1 * ID_2 * ID_3 *$
4) $*\ ID_0\ \hat{*}\ ID_1 * ID_2 * ID_3 *$
5) $*\ ID_0\ \hat{*}\ ID_1 * ID_2 * ID_3 * ID_1 * ID_1 *$
6) $*$ — " — $* ID_{11} * ID_{12} *$
7) $*\ ID_0 * ID_1\ \hat{*}\ ID_2 * ID_3 * ID_{11} * ID_{12} *$
⋮

Simulation time:

Let NTM N have running time $T(n)$.
What is the running time of D?

Let m be the maximum number of non-det. choices for each transition (i.e., the maximum size of $\delta(q,x)$)

Consider execution tree of N on w.

let $t = T(|w|)$ $\Rightarrow$ exec. tree has at most t levels

size of the tree is $\leq 1 + m + m^2 + \dots + m^t =$

$$= \frac{m^{t+1} - 1}{m-1} = O(m^t)$$

Thus D has at most $O(m^t)$ iterations of steps 0-3.
Each iteration requires at most $O(m^t)$ steps

$\Rightarrow$ Total running time is $m^{O(t)}$, i.e. exponential