Algorithms,
Nondeterminism,
Complexity, and
Randomicity

Alberto Pettorossi
Department of Informatics, Systems, and Production,
University of Roma Tor Vergata

4 March 2009

EXISTENCE OF NON-COMPUTABLE FUNCTIONS

$\mathbb{N}$: natural numbers

$\wp(\mathbb{N})$: set of all subsets of natural numbers

$\mathbb{N} \rightarrow \mathbb{N}$: functions from $\mathbb{N}$ to $\mathbb{N}$

Progs: programs in C++ (or Java)

Theorem. $\mid \mathbb{N} \mid \, = \, \mid \mathit{Progs} \mid$

Theorem. $\mid \mathbb{N} \rightarrow \mathbb{N} \mid \, = \, \mid \wp(\mathbb{N}) \mid \, = \, \mid \mathbb{R} \mid$

Cantor Theorem. For any set A, there is no bijection from A to $\wp(A)$.

Corollary. There is a function f from $\mathbb{N}$ to $\mathbb{N}$ such that it has no corresponding C++ program, i.e., f is a non-computable function.

Theorem. $\mid \mathbb{R} - \mathbb{N} \mid \, = \, \mid \mathbb{R} \mid$.

Corollary. (i) There are $\mid \mathbb{N} \mid$ computable functions from $\mathbb{N}$ to $\mathbb{N}$.
(ii) There are $\mid \mathbb{R} \mid$ (computable or non-computable) functions from $\mathbb{N}$ to $\mathbb{N}$.

VARIOUS LEVELS OF COMPUTABILITY

(above) : non-computable functions

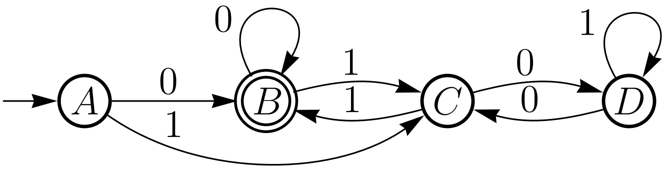
(type 0) (r.e.) : Turing Machines (TM) = FA + 2 stacks (push x, pop, is-empty ?)
computable functions

(rec) : Turing Machines which terminate for all inputs
total computable functions

(type 1) : ...

(type 2) : Pushdown Automata = FA + 1 stack (push x, pop, is-empty?)
$\{a^n b^n \mid n \geq 1\}$

(type 3) : Finite Automata (= FA)



(divisibility by 3 for binary numbers)

1100 (= 12) takes from A to B.

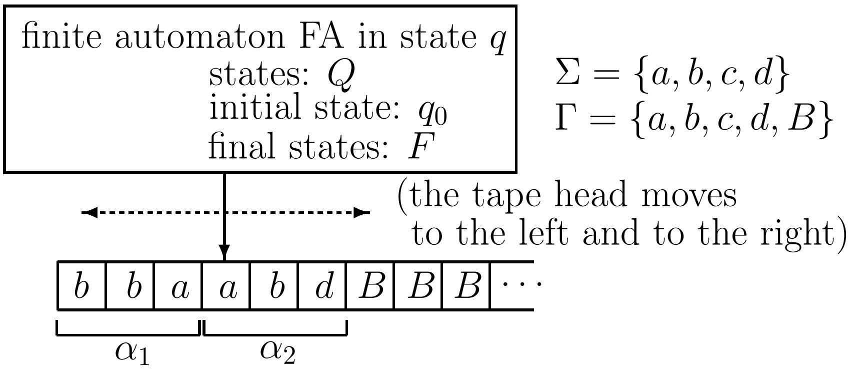
Note 1. FA + 2 stacks (push x, pop, is-empty?)
= FA + 2 counters (+1, −1, is-0 ?)
= FA + 1 tape (read, write, move L, move R)

Note 2. Also the encoding of the input and the decoding of the output should be computable (for instance, the encoding of 12 into 1100).

A TURING MACHINE

A *Turing Machine* is a septuple $\langle Q, \Sigma, \Gamma, q_0, B, F, \delta \rangle$, where:
- Q is the set of *states*,
- Σ is the *input alphabet*,
- Γ is the *tape alphabet*,
- q_0 is the *initial state*,
- B is the *blank symbol*,
- F is the set of the *final states*, and
- δ is a *partial function* from $Q \times \Gamma$ to $Q \times (\Gamma - \{B\}) \times \{L, R\}$, called the *transition function*, which defines the set of *instructions* or *quintuples* of the Turing Machine.

We assume that Q and Γ are disjoint sets, $\Sigma \subseteq \Gamma - \{B\}$, $q_0 \in Q$, $B \in \Gamma$, and $F \subseteq Q$.



A Turing Machine in the configuration $b\,b\,a\,q\,a\,b\,d$. The cells of the tape are $c_1, c_2, \ldots$ The head scans the cell c_4 and reads the symbol a.

A Turing Machine accepts an input word w when starting from the configuration $q_0\,w$ reaches a configuration $w_1\,q\,w_2$ where $q \in F$.

For any expression e, by $\ulcorner e \urcorner$ we denote the encoding of e.

A Turing Machine computes the function $f : \mathbb{N} \rightarrow \mathbb{N}$ iff for all $n \in \mathbb{N}$, starting from the configuration $q_0 \ulcorner n \urcorner$ reaches a configuration $q \ulcorner f(n) \urcorner$ where $q \in F$.

PROBLEMS AS SETS OF WORDS

non-halting: $\{\, \ulcorner x \urcorner \,\$\, \ulcorner M \urcorner \mid \text{Turing Machine } M \text{ does not halt for input } x\}$

$\subseteq\ (0+1)^*\,\$\,(0+1)^*$

---

halting: $\{\, \ulcorner x \urcorner \,\$\, \ulcorner M \urcorner \mid \text{Turing Machine } M \text{ halts for input } x\}$

$\subseteq\ (0+1)^*\,\$\,(0+1)^*$

parsing: $\{\, \ulcorner w \urcorner \,\$\, \ulcorner G \urcorner \mid \text{grammar } G \text{ generates } w\}$

$\subseteq\ (0+1)^*\,\$\,(0+1)^*$

. . .

primes: $\{01^n \mid \mathit{prime}(n)\}$ $\subseteq\ 01^*$

sorting: $\{x_1\#\ldots\#x_n\,\$\,x_{i_1}\#\ldots\#x_{i_n} \mid x_{i_1}\leq\ldots\leq x_{i_n}\}$

$\subseteq\ (\mathbb{N}\#)^{n-1}\mathbb{N}\,\$\,(\mathbb{N}\#)^{n-1}\mathbb{N}$

sum: $\{01^n01^m01^{m+n}0 \mid n,m\geq 0\}$ $\subseteq\ 01^*01^*01^*0$

. . .

RECURSIVELY ENUMERABLE SETS AND RECURSIVE SETS

- *Non-recursively enumerable sets* (subsets of L).
  Non-semi-solvable (or non-semi-decidable) problems:
  non-halting, ...

- *Recursively enumerable sets* (subsets of L).

A set $A \subseteq L$ is *r.e.* iff there exists a Turing Machine M such that
for all $a \in A$, M stops in a final state for the input a.

Semi-solvable (or *semi-decidable*) *problems*:

halting,
parsing for type 0 grammars, ...

- *Recursive sets* (subsets of L).

A set $A \subseteq L$ is *rec* iff there exists a Turing Machine M such that
- for all $x \in L$, M stops and
- for all $a \in A$, M stops in a final state for the input a.

Solvable (or decidable) problems:

parsing for type i grammars (with $i = 1, 2, 3$),
primes,
sorting,
sum, ...

ALGORITHMS DENOTE R.E. SETS

Theorem. Given a Turing-complete programming language L, the interpreter I for the language L (maybe written in L) should allow non terminating executions. □

Thus, there exist (i) a program P written in L, and (ii) an input x for P such that the interpreter I, running on P and x, does not terminate.

Thus, if we want to consider I as an algorithm, then the class of algorithms should denote r.e. sets.

Note. An algorithm is a Turing Machine.
The execution of an algorithm is *not a finite sequence of well-defined steps*, as indicated in some books (see below), because: (i) the execution may not terminate, (ii) the execution may be nondeterministic, and (iii) the notion of *step* is unclear.

...

Input - L'algoritmo deve avere un input contenuto in un insieme definito I.
Output - Da ogni insieme di valori in input, l'algoritmo produce un insieme di valori in uscita che comprende la soluzione.
Determinatezza - I passi dell'algoritmo devono essere definiti precisamente.
Finitezza - Un algoritmo deve produrre la soluzione in un numero di passi finito (eventualmente molto grande) per ogni possibile input definito su I.
Efficacia - Deve essere possibile effettuare ogni passo del l'algoritmo esattamente ed in un tempo finito.
Generalità - L'algoritmo deve essere valido per ogni insieme di dati contenuti in I e non solo per alcuni.

...

DETERMINISM AND NONDETERMINISM

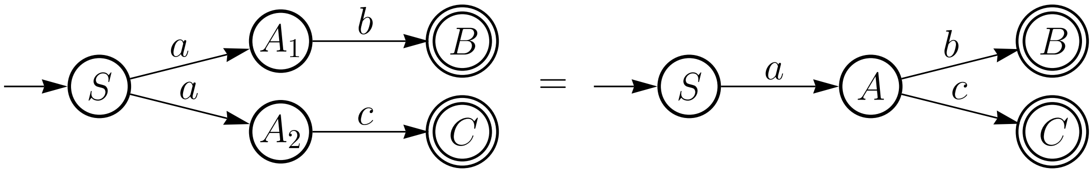
Deterministic acceptance:
starting from the initial state, the computation of the automaton stops in a final state.

Nondeterministic acceptance:
there exists a computation starting from the initial state such that the automaton stops in a final state.

(type 0) : Nondeterministic Turing Machines = Deterministic Turing Machines

(type 2) : Nondeterministic Pushdown Automata
$\supset$ Deterministic Pushdown Automata

(type 3) : Nondeterministic Finite Automata = Deterministic Finite Automata



TIME COMPLEXITY

- Polynomial relationship between the time complexity of Random Access Machines (or C++ programs) and deterministic Turing Machines.

There exists $k \geq 0$ such that for every algorithm A from $\mathbb{N}$ to $\mathbb{N}$,

there exists a Random Access Machine (without multiplication or division) which takes $O(n)$ time units for executing A iff
there exists a deterministic Turing Machine which takes $O(n^k)$ time units (that is, polynomial time w.r.t. n) for executing A.

The same for *space*, instead of *time*.

- **P** : Deterministic Polynomial Class.

A predicate $\lambda d.\, p(d) : D \rightarrow \{\mathit{true}, \mathit{false}\}$ is in P iff there exists a deterministic Turing Machine M such that for all $d \in D$,

M evaluates $p(d)$ in polynomial time w.r.t. $\mathit{size}(d)$.

One can show that the class **P** is closed w.r.t. negation, that is, **P** = **co-P**. Thus, in the above definition of the class **P**, actually, it does not matter whether the evaluation of $p(d)$ returns *true* or *false*.

THE **NP** CLASS

- **NP** : Nondeterministic Polynomial Class.

A predicate $\lambda d.\, p(d) : D \rightarrow \{true, false\}$ is in **NP** iff

(i) there exists a deterministic Turing Machine M,

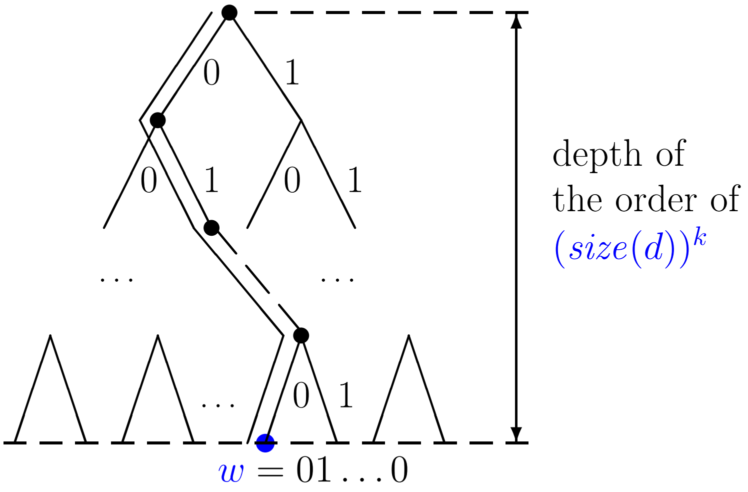
(ii) there exists a finite alphabet Σ,

(iii) there exists a predicate $\pi : D \times \Sigma^* \rightarrow \{true, false\}$, and

(iv) there exists $k \geq 0$ such that

- for all $d \in D$, M evaluates $p(d)$ by returning the value: $\exists w \in W.\ \pi(d, w)$ where $W = \{w \in \Sigma^* \mid size(w) = (size(d))^k\}$, and
- for all $d \in D$, $w \in \Sigma^*$, M evaluates $\pi(d, w)$ in polynomial time w.r.t. $size(d) \times size(w)$.

The evaluation of $p(d)$ in **NP** is a search, in a polynomially deep tree, of a leaf, if any, associated with a word w such that $\pi(d, w)$ holds.



(In the picture we have assumed $\Sigma = \{0, 1\}$.)

- It is an open problem whether or not **NP** is closed w.r.t. negation, that is, whether or not **NP** = **co-NP**.
- **NP** $\cap$ **co-NP** $\neq \emptyset$.
  PRIME and CO-PRIME both belong to **NP** and **co-NP** (see below).
- If **P** = **NP** then **NP** = **co-NP** (because **P** = **co-P**).
  The reversed implication, if **NP** = **co-NP** then **P** = **NP**, does *not* hold.

CO-PRIME IS IN **NP**

CO-PRIME

Input: a positive integer p in binary notation, using $k = \lceil \log_2 p \rceil$ bits.

Output: ‘yes’ iff p is *not* a prime number.

An instance of the CO-PRIME problem can be solved as follows:

Step 1. We construct the set $W \in \{0, 1\}^k$ of the encodings of all positive numbers i, for $1 < i < p$.

Step 2. Then we return ‘yes’ iff *there exists* an encoding $w \in W$ such that w represents a number which divides p. $\square$

(Obviously, to test whether or not a given number divides p requires polynomial time with respect to k.)

As recently shown, actually the CO-PRIME problem is in **P**, because the PRIME problem (see below) is in **P**.

PRIME IS IN **NP** (1)

PRIME

Input: a positive integer p in binary notation, using $k = \lceil \log_2 p \rceil$ bits.

Output: ‘yes’ iff p is a prime number.

- The proof that a problem is in **NP** is, in general, *not* a proof that the negated problem is in **NP**.

Theorem 1. [Fermat Theorem] A number $p\,(>2)$ is prime iff there exists x with $1<x<p$, such that
(α): $x^{p-1} = 1 \pmod{p}$ and
(β): for all i, with $1<i<p-1$, $x^i \neq 1 \pmod{p}$.

Example. We have that $p=7$ is a prime number because there exists x, which is 3, such that $1<x<7$ and
(α) holds because $3^{7-1} = 729 = 1 \pmod{7}$, and
(β) holds because

$$\begin{array}{lll}
3^2 = 2 & \pmod{7} & \text{(indeed, } 3 \times 3 = 9 = 7+2) \\
3^3 = 6 & \pmod{7} & \text{(indeed, } 2 \times 3 = 6) \\
3^4 = 4 & \pmod{7} & \text{(indeed, } 6 \times 3 = 18 = (2 \times 7)+4) \\
3^5 = 5 & \pmod{7} & \text{(indeed, } 4 \times 3 = 12 = 7+5)
\end{array}$$

□

Theorem 2. Assume that: (i) $p>2$, (ii) $1<x<p$, and (iii) $x^{p-1}=1 \pmod{p}$.
If for all $p_j \in \mathit{primefactors}(p-1)$, $x^{(p-1)/p_j} \neq 1 \pmod{p}$
then for all i, with $1<i<p-1$, we have that $x^i \neq 1 \pmod{p}$.

PRIME IS IN **NP** (2)

Theorem 2. Assume that: (i) $p>2$, (ii) $1<x<p$, and (iii) $x^{p-1}=1 \pmod p$.
If for all $p_j \in \mathit{primefactors}(p-1)$, $x^{(p-1)/p_j} \neq 1 \pmod p$
then for all i, with $1<i<p-1$, we have that $x^i \neq 1 \pmod p$.

- The size of the input p is $\lceil \log_2 p \rceil$.
- $|\{i \mid 1<i<p-1\}|$ $(=p-3)$ tests are too many, because $p-3 = O(2^{\log_2 p})$.
- $|\mathit{primefactors}(p-1)|$ tests are *not* too many, because $|\mathit{primefactors}(p-1)| \leq \lfloor \log_2 p \rfloor$.

Example (continued). In order to test Condition (β) we need to test that:

$3^2 \neq 1 \pmod 7$ $\quad (*)$
$3^3 \neq 1 \pmod 7$ $\quad (*)$
$3^4 \neq 1 \pmod 7$
$3^5 \neq 1 \pmod 7$.

Indeed, all these inequalities hold, because

$3^2 = 9 = 2 \pmod 7$
$3^3 = 27 = 6 \pmod 7$
$3^4 = 81 = 4 \pmod 7$
$3^5 = 243 = 5 \pmod 7$.

By Theorem 2, since $\mathit{primefactors}(7-1) = \{2, 3\}$,
in order to test (β), it is enough to test only the disequations marked with $(*)$.
Indeed, we have that:

$3^{6/2} \neq 1 \pmod 7$ and
$3^{6/3} \neq 1 \pmod 7$. □

FROM **P** TO **EXPTIME**

EXPTIME	Simplex Algorithm for linear programming $min\ z = c^{\mathrm{T}}x$ with $Ax = b, x \geq 0$
$\cup\!|$	($\|$? open problem)
PSPACE	(= **NSPACE**) membership of context-sensitive languages
$\cup\!|$	($\|$? open problem)
NP	satisfiability of propositional formulas in CNF
$\cup\!|$	($\|$? open problem)
P	emptiness of context-free languages

One of the $\cup\!|$'s is $\cup$, because **P** $\subset$ **EXPTIME**.

INTERACTIVE PROOF SYSTEM (1)

Prover:	TM **Alice**	$x \in L\,?$	Verifier:	TM **Bob**
• input tape:	$\ulcorner x \urcorner$ $\leftrightarrow$		• input tape:	$\ulcorner x \urcorner$ $\leftrightarrow$
• random tape:	0101011011 . . . $\rightarrow$		• random tape:	1001010110 . . . $\rightarrow$
• working tape:	. . . $\leftrightarrow$		• working tape:	. . . $\leftrightarrow$
	(no limitations)		total polynomial time w.r.t. *size*(x)	

• interaction tape: $m_0 \mid m_1 \mid m_2 \mid \ldots$
$\rightarrow$

In the interation tape:

- mutual exclusive use by TM **Alice** and TM **Bob** according to a protocol.
- every message m_i uses polynomial space.

Proof in probability:

- if $x \in L$ then **Bob** accepts x with probability $\geq \frac{2}{3}$.
- if $x \notin L$ then **Bob** accepts x with probability $\leq \frac{1}{3}$.

Definition. The class **IP** is the class of the languages L recognized by an interactive proof system.

Note. We can replace the values $\frac{2}{3}$ and $\frac{1}{3}$ by any other values $> \frac{1}{2}$ and $< \frac{1}{2}$, respectively, without changing the definition of the class **IP**.

Indeed, by repeating the protocol we can reach any desired level of probability.

INTERACTIVE PROOF SYSTEM (2)

Definition. The class **IP** is the class of languages recognized by an interactive proof system.

Fact. **NP** $\subseteq$ **IP**.
Proof. Here is the protocol. The TM **Alice** sends to the TM **Bob** the encoding of the TM **Alice** and the list of the choices which are made by the TM **Alice** for accepting the input string x. Then TM **Bob** in polynomial time will act according to that list of choices and will accept x (with probability 1). $\square$

Theorem. **IP** = **PSPACE**.

GRAPH NON-ISOMORPHISM.
Input: two directed or undirected graphs G_1 and G_2.

Output: 'yes' iff the graphs G_1 and G_2 are non-isomorphic (that is, they are *not* equal modulo a permutation of the vertices).

Fact. The GRAPH NON-ISOMORPHISM problem is in **IP**.

Proof. Here is the protocol. G_1 and G_2, are given both to **Alice** and **Bob**. The protocol has two steps.

Step 1.

(1.1) **Bob** randomly chooses an i in $\{1, 2\}$.

(1.2) **Bob** sends to **Alice** a graph H which is isomorphic to G_i and is obtained by **Bob** by randomly permuting the vertices of G_i.

(1.3) **Alice** looks for a j in $\{1, 2\}$ such that graph G_j is isomorphic to H and she sends j back to **Bob**. (Note that **Alice** will always succeed in finding j, but if the two graphs G_1 and G_2 are isomorphic, j may be different from i.)

Step 2. It is equal to Step 1.
When **Bob** receives the second value of j, he announces the end of the protocol.

If the two given graphs are non-isomorphic, in both steps **Alice** returns to **Bob** the same value he sent to her, and **Bob** accepts the given graphs as non-isomorphic with probability 1.

If the two given graphs are isomorphic, since in each step the probability that **Alice** guesses the i chosen by **Bob** is $\frac{1}{2}$, we have that the probability that **Bob** accepts the given graphs as non-isomorphic is at most $\frac{1}{4}$ (which is $\leq \frac{1}{3}$). □

INTERACTIVE PROOF SYSTEM (4)

GRAPH ISOMORPHISM.

Input: two directed or undirected graphs G_1 and G_2.

Output: 'yes' iff the graphs G_1 and G_2 are isomorphic (that is, they are equal modulo a permutation of the names of their vertices).

Fact 1. The GRAPH ISOMORPHISM problem is in **NP**.

Fact 2. The GRAPH NON-ISOMORPHISM problem is in **co-NP**.

It is an open problem whether or not the GRAPH NON-ISOMORPHISM problem is in **NP**.

References

[1] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[2] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, Usa, Monterey, California, Usa, 1987. Third Edition.

[3] A. Pettorossi. *Elements of Computability, Decidability, and Complexity.* Aracne Editrice, Roma, Italy, 2007. Second Edition.