The use of mathematical logic as a descriptive formalism / 1

- Logic is a “universal” formalism (very close to natural language)
- It can be applied to a wide variety of contexts, not only to computer science
- Mathematical logic has several connections with areas of (theoretical) computer science
  - many notations and languages based on Logic (e.g., in programming languages, specification languages)
  - relations with automata, formal languages, grammars
  - there are many important results concerning (various types of) logics:
    - expressive power, decidability of satisfiability/validity problem, complexity of decision procedures
- We assume a (moderate) knowledge from previous courses, otherwise
  - Read very carefully Ch.0 of textbook, §0.2 Basic Elements of Mathematical Logic
  - Read very carefully the slide file: MATHEMATICAL LOGIC–INTRODUCTION in the Lessons document folder on WeBeeP
  - for a more in-depth approach:"Introduction to Mathematical Logic" by Elliott Mendelson (for mathematically oriented students very fond of logics)

The use of mathematical logic as a descriptive formalism / 2

- We provide a partly informal treatment of Logic, used mainly as a descriptive notation to define
  - Languages (with a part on the relation between regular languages and a specific class of mathematical logic formulae)
  - Program properties
  - System timing properties
- Many important features (expressive power, decidability, complexity) of various types of logics depend on
  - assumed Interpretation domain (Universe ***U***): the set of possible values of terms (constants, variables, expressions)
  - assumed alphabet of predicate and function symbols
  - possibility for (quantified) variables to denote only first-order objects (elements of the universe) or second-order objects (sets of such elements and relations among them)
- by including suitable predicate and function symbols a «maximal» generality is soon achieved (examples from arithmetics and the famous Gödel incompleteness theorem)
- Often we «play» the following «game»: define some entity in logic using a given, suitably constrained alphabet

1. Mathematical logic to define languages

language $L = \{a^n b^n \mid n \geq 1\}$ specified by the first-order formula

$$\forall x \left(x \in L \leftrightarrow \exists n \, (n \geq 1 \wedge x = a^n \cdot b^n) \right)$$

Notice the alphabet:

variable ***x*** denotes a string, ***n*** a natural number
constant ***L*** denotes a language (set of strings), (constant $\boldsymbol{\varepsilon}$ the empty string)
set-theoretic predicate '$\in$', equality predicate '='
operations '$\cdot$' (string concatenation) and x^n (power on strings)

power operation on strings, x^n, is defined by the formula

$$\forall n \left(\left(n = 0 \rightarrow x^n = \varepsilon \right) \wedge \left(n > 0 \rightarrow x^n = x^{n-1} \cdot x \right) \right)$$

based on the elementary operation for concatenation "$\cdot$" and the integer '-' operation

Notice: for brevity, outermost universal quantifiers are often left implicit, for instance
$\boldsymbol{x \in L \leftrightarrow \exists n (n \geq 1 \wedge x = a^n \cdot b^n)}$ to be intepreted as $\forall \boldsymbol{x} (\boldsymbol{x \in L \leftrightarrow \exists n (n \geq 1 \wedge x = a^n \cdot b^n)})$

- Define language $L_1 = a^*b^*$ without using power operation x^n

  (otherwise, simply, $x \in L_1 \leftrightarrow \exists m \exists n (x = a^m b^n \wedge n \geq 0 \wedge m \geq 0)$)

$$x \in L_1 \leftrightarrow \Big((x = \varepsilon) \vee \exists y \Big(y \in L_1 \wedge \big(x = ay \vee x = yb \big) \Big) \Big)$$

- Define $L_2 = b^*c^*$ similarly
- $L_3 = a^*b^*c^*$ $(= L_1 \cdot L_2)$ can also be defined (without using x^n) using the same sort of inductive style:

$$x \in L_3 \leftrightarrow (x \in L_1) \vee (x \in L_2) \vee \exists y \, (y \in L_3 \wedge (x = ay \vee x = yc))$$

- Definition of language $L_4 = \{x \mid \#_a x = \#_b x\}$ based on definition of function $\#_a x$ ($\#_b x$ defined similarly) by:

$$\big(x = \varepsilon \rightarrow \#_a x = 0\big) \wedge \big(x = ay \rightarrow \#_a x = \#_a y + 1\big) \wedge \big(x = by \rightarrow \#_a x = \#_a y\big)$$

- Then Def. of L_4 in the customary way: $x \in L_4 \leftrightarrow (\#_a x = \#_b x)$

A very constrained logic to define languages

- Monadic First Order (MFO) logic (monadic ≡ predicates with one argument) with (in addition to monadic predicates) the only ***binary*** order predicate '<'
- boolean connectives ($\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, etc. ...) and quantifiers ($\exists$, $\forall$) defined as usual
- w.r.t. alphabet $\boldsymbol{I}$ consider string $\boldsymbol{w \in I^*}$, with $|\boldsymbol{w}|=\boldsymbol{n}$, i.e., $\boldsymbol{w=w_0\ w_1\ \ldots\ w_{n-1}}$
- Universe $\boldsymbol{U = \{0, \ldots, n\text{-}1\}}$ of string ***positions*** (NB: if $\boldsymbol{x = \varepsilon}$ then $\boldsymbol{U = \varnothing}$)
- for every symbol $\boldsymbol{a \in I}$ monadic predicate $\boldsymbol{a(x)}$, true for string $\boldsymbol{w}$ iff $\boldsymbol{w_x = a}$, i.e., iff symbol $\boldsymbol{a}$ occurs in string $\boldsymbol{w}$ at position $\boldsymbol{x}$
- binary predicate '<' among string positions has the usual meaning

- Derived definitions
  - $x \geq y$ defined as $\neg(x < y)$
  - $x \leq y$ as $y \geq x$
  - $x = y$ as $x \leq y \wedge y \leq x$
  - $x \neq y$ as $\neg(x = y)$
  - $x > y$ as $y < x$
  - immediate successor: $succ(x, y)$ ($S(x, y)$ for short) as $x < y \wedge \neg\exists z\ (x < z \wedge z < y)$

  - constant **0**: $x = 0$ as $\forall y \neg\ (y < x)$
  - all natural integer constants **1**, **2**, **3**, ... : defined as successors of **0**, **1**, **2**, etc.
  - $y = x + 1$ defined as $succ(x, y)$
  - $y = x+k$, for every $k > 1$, as $\exists z_1 \ldots \exists z_{k-1}\ (z_1 = x+1 \wedge \ldots \wedge y = z_{k-1}+1)$
  - $y = x - 1$ as $succ(y, x)$
  - $y = x - k$, for every $k > 1$, as $x = y + k$
  - first position in the string : $first(x)$ as $\neg\exists y\ (y < x)$ (equiv. to $x = 0$)
  - last position in the string : $last(x)$ as $\neg\exists y\ (y > x)$

  - **NB**: terms like $x + y$, with x and y both variables are ***not admitted***

Interpretation of a MFO Logic formula w.r.t. a string

- every string $w \in I^*$ corresponds to (defines) an interpretation structure with alphabet I and universe $U = \{ 0, \ldots, |w|-1 \}$
- ex: for string $w = acbaa$, with $n = |w| = 5$, we have
  - alphabet $I = \{a, b, c\}$
  - f.o. variables denote positions in w; they are interpreted over the Universe $U = [0 .. n-1] = [0 .. 4]$
  - for every alphabet element $i \in I$, predicate $i(x)$ denotes the set of positions x at which $w_x = i$, (NB: $\forall i \neq j \; \forall x \; \neg(i(x) \wedge j(x))$)
    - for $w = acbaa$, $a(0)$, $c(1)$, $b(2)$, $a(3)$, $a(4)$ are ***true***;
      $b(0)$, $c(0)$, $a(1)$, $b(1)$, $a(2)$, $c(2)$, $b(3)$, $c(3)$, $b(4)$, $c(4)$ are ***false***
    - NB: in the atomic formula $a(x)$, a is a predicate ***constant***: it is not quantified
  - less-than relation for $w = acbaa$ is a set of pairs of positions:

    '<' = {(0,1), (0,2), ... (1,2), (1,3), … (3,4)}

Notation: (a ***sentence*** is a formula with all variables quantified)

string w satisfies sentence φ: $w \models \varphi$

string w does not satisfy sentence φ: $w \not\models \varphi$

Any sentence φ defines the language $L(\varphi)$ that includes exactly (all and only) the strings satisfying φ**:**

$$L(\varphi) = \{ w \mid w \models \varphi \}$$

Examples of MFO sentences defining strings and languages

- $\varphi: \exists x(x = 0 \wedge a(x))$ $\quad L(\varphi) = aI^*$ strings starting with an a
- strings where every a is immediately followed by a b :

  $\varphi : \forall x (a(x) \rightarrow \exists y (S(x, y) \wedge b(y)))$

  $aaba \not\models \varphi, \quad babbbab \models \varphi, \quad bba \not\models \varphi$

- (non-empty) strings ending with an 'a' : $\exists x(last(x) \wedge a(x))$
- strings (of length ≥ 3) where second symbol before the last is a

  $\exists x(\exists y (y = x+2 \wedge last(y)) \wedge a(x))$

- Every ***singletone*** (one-element) language easily defined; example $L_{abc} = \{ abc \}$

$\exists x \exists y \exists z \, (x=0 \wedge S(x,y) \wedge S(y,z) \wedge last(z) \wedge a(x) \wedge b(y) \wedge c(z))$

Empty string ε often requires some special care

- for string $\boldsymbol{\varepsilon}$, the set of positions is empty
- convention: when $\boldsymbol{U} = \varnothing$, $\exists \boldsymbol{x}\boldsymbol{\varphi}$ is false and $\forall \boldsymbol{x}\boldsymbol{\varphi}$ true, for any φ

  (see lecture notes, where semantics is precisely defined)

  Lecture Notes on Monadic First- and Second-Order Logic on Strings - Study material on WeBeeP

- a sentence for $\{\varepsilon\}$ must be true for ε but false $\forall \boldsymbol{w} \neq \varepsilon$

$$\neg\exists \boldsymbol{x}\ (\ \boldsymbol{a}(\boldsymbol{x}) \vee \neg \boldsymbol{a}(\boldsymbol{x})\ )$$ (**i.e.,** $\neg\exists x$ (***true***), **or** $\neg\exists x$ **or** $U=\varnothing$)

or, equivalently, $\forall \boldsymbol{x}\ (\ \boldsymbol{a}(\boldsymbol{x}) \wedge \neg \boldsymbol{a}(\boldsymbol{x})\ )$ (**i.e.,** $\forall x$ (***false***))

Properties of MFO (1/2)

- Family of languages expressible in MFO is closed under set-theoretic operations (union, intersection, complement etc.)
  - trivial: use disjunction '$\vee$', conjunction '$\wedge$', negation '$\neg$'
- Every ***finite*** or ***co-finite*** (its complement is finite) language is expressible in MFO
  - from above closure properties under propositional operators and MFO ability to express singletone languages
- Every language over a one-letter alphabet (e.g., $\Sigma=\{a\}$) expressible in MFO is either ***finite*** or ***co-finite***
  - Proof is not trivial, interested students find it in the lecture notes

- Therefore in MFO one cannot express the language $\boldsymbol{L_e = (aa)^*}$ that includes ***exactly*** the even-length strings over $\boldsymbol{I=\{a\}}$
  - because $\boldsymbol{L_e = (aa)^*}$ is neither finite nor co-finite

Properties of MFO (2/2)

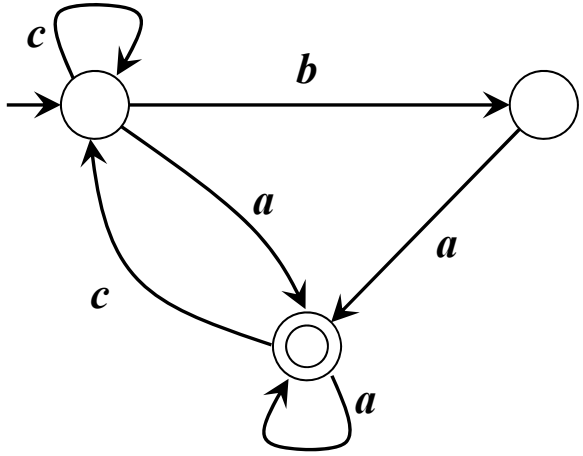
- MFO is strictly less powerful than FA (Finite state Automata)
  - from a MFO formula easy to obtain an equivalent FA (we do not show how...)
  - $L_e = (aa)^*$ is recognized by a simple FA (but not expressible in MFO)

- Languages defined by MFO are ***not*** closed under Kleene star '*' :
  - MFO formula $\exists x \exists y\ (\ x{=}0 \wedge y{=}1 \wedge a(x) \wedge a(y) \wedge last(y))$ defines language $L_{e2} = \{aa\}$, and we have $L_e = L_{e2}^*$

- MFO defines the family of (so-called) ***star-free languages***, that can be obtained starting from any finite language by means of a finite number of union, intersection, complement, and concatenations operations (but not Kleen star '*')

Example of relation between FA and MFO logic formula

$$\varphi = \left(\begin{array}{c} \textbf{no } 'a' \textbf{ is followed by a } 'b' \\ \wedge \\ \textbf{every } 'b' \textbf{ is followed by an } 'a' \\ \wedge \\ \textbf{the string ends with an } 'a' \end{array} \right)$$



$$\varphi = \left(\begin{array}{c} \neg\exists x \exists y (S(x,y) \wedge a(x) \wedge b(y)) \\ \wedge \\ \forall x \left(b(x) \rightarrow \exists y \left(S(x,y) \wedge a(y) \right) \right) \\ \wedge \\ \exists x \, (last(x) \wedge a(x)) \end{array} \right)$$

- Are the two models equivalent?
- Is it always possible to
  - find an FA equiv. to a MFO?
  - find an MFO equiv. to a FA?

No, because MFO logic is strictly less powerful than Finite state Automata

Monadic Second Order (MSO) Logic

- Expressive power of MFO can be increased by introducing variables denoting ***monadic predicates*** i.e., ***sets of numbers*** (remember: for every string w, the universe U includes the numbers representing positions in w)
  - notation: if predicate X represents a set, then $X(3)$ is the same as $3 \in X$
- For clarity we use uppercase identifiers for second order variables denoting predicates: e.g., a possible formula is of the type

$$\exists X \, \varphi(X)$$

  - lowercase identifiers still used for first-order variables, denoting positions in the strings
- Example: the language $L_e = (aa)^*$ is defined by the following MSO formula, where predicate E identifies the even ***positions*** (odd indices)

$$\begin{aligned} \exists E \, \forall x \, (\quad & a(x) \wedge \\ & (x = 0 \rightarrow \neg E(x)) \wedge \\ & \forall y \, (y = x+1 \rightarrow \, (\neg E(x) \leftrightarrow E(y))) \, \wedge \\ & (last(x) \rightarrow E(x)) \qquad) \end{aligned}$$

Another example of MSO language specification

Words over $\boldsymbol{I = \{a, b\}}$ where every two occurrences of $\boldsymbol{b}$, having no other $\boldsymbol{b}$ in between, are separated by an odd number of $\boldsymbol{a}$'s

$$\varphi = \forall x \forall y \left(\begin{array}{c} b(x) \wedge x < y \wedge b(y) \wedge \forall z \big(x < z \wedge z < y \rightarrow \neg b(z) \big) \\ \rightarrow \\ \exists X \Big(X(x) \wedge X(y) \wedge \forall u \forall v \ \big(S(u,v) \rightarrow \big(X(u) \leftrightarrow \neg X(v) \big) \big) \Big) \end{array} \right)$$

second order quantification **:**
X is a predicate that alternates true and false values

$$a\ b\ a\ a\ a\ b\ a\ b \models \varphi$$

$\neg X\ X\ \neg X\ X\ \neg X\ X\ \neg X\ X$

OK **OK**

$$a\ b\ a\ a\ b\ a\ b \not\models \varphi$$

$\neg X\ X$ **????** $X\ \neg X\ X$

KO **OK**

Important result: the family of languages defined by MSO sentences is equal to RL (regular languages)

Büchi's theorem (1960)

- A language (of finite-length words) is recognizable by a finite automaton iff it is definable by a **MSO** sentence
- The two ***conversions*** (automaton $\Leftrightarrow$ formula) are both ***effective***

We provide a (sketchy) proof of the «automaton $\Rightarrow$ formula» part

We only exemplify (provide an intuition for) the «formula $\Rightarrow$ automaton» part

Büchi's theorem - «automaton ⇒ formula» part

- automaton $A = (Q, I, q_0, \delta, F)$, $Q = \{q_0, \dots q_k\}$, reads a string w
- use $|Q|$ distinct predicates X_i = {positions of w where A reaches state q_i }
  = { x | A is in state q_i when it reads w_x }
- formula φ equivalent to A :

$$\varphi = \exists X_0 \dots \exists X_k \left(\begin{array}{l} \wedge_{i \neq j} \forall x \neg (X_i(x) \wedge X_j(x)) \\ \wedge \forall x \, (first(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left(S(x,y) \rightarrow \bigvee_{q_j \in \delta(q_i, a)} (X_i(x) \wedge a(x) \wedge X_j(y)) \right) \\ \wedge \forall x \left(last(x) \rightarrow \bigvee_{\exists q_j \in F:\, q_j \in \delta(q_i, a)} (X_i(x) \wedge a(x)) \right) \end{array} \right)$$

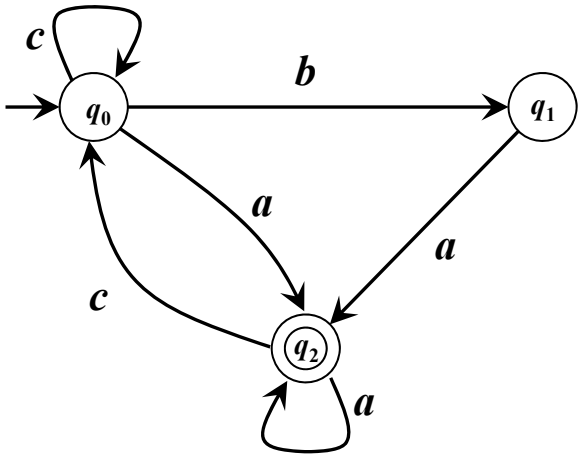
state positions pairwise disjointed

A in the **initial state** right before reading the first symbol

final state: reached after reading the last symbol of the string

transitions: reading a at position x, A goes from state q_i to state q_j

Example of Büchi's theorem - «automaton ⇒ formula»



$$\exists X_0 \exists X_1 \exists X_2 \left(\begin{array}{c} \forall x \big(\neg(X_0(x) \wedge X_1(x)) \wedge \neg(X_0(x) \wedge X_2(x)) \wedge \neg(X_1(x) \wedge X_2(x)) \big) \\ \wedge \forall x \, (first(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left(S(x,y) \rightarrow \begin{pmatrix} X_0(x) \wedge c(x) \wedge X_0(y) \vee X_0(x) \wedge b(x) \wedge X_1(y) \\ \vee X_0(x) \wedge a(x) \wedge X_2(y) \vee X_1(x) \wedge a(x) \wedge X_2(y) \\ \vee X_2(x) \wedge a(x) \wedge X_2(y) \vee X_2(x) \wedge c(x) \wedge X_0(y) \end{pmatrix} \right) \\ \wedge \forall x \big(last(x) \rightarrow (X_0(x) \wedge a(x) \vee X_1(x) \wedge a(x) \vee X_2(x) \wedge a(x)) \big) \end{array} \right)$$

Büchi's theorem - «formula ⇒ automaton», part ¼
A VERY SKETCHY ILLUSTRATION
(Interested students refer to lecture notes)

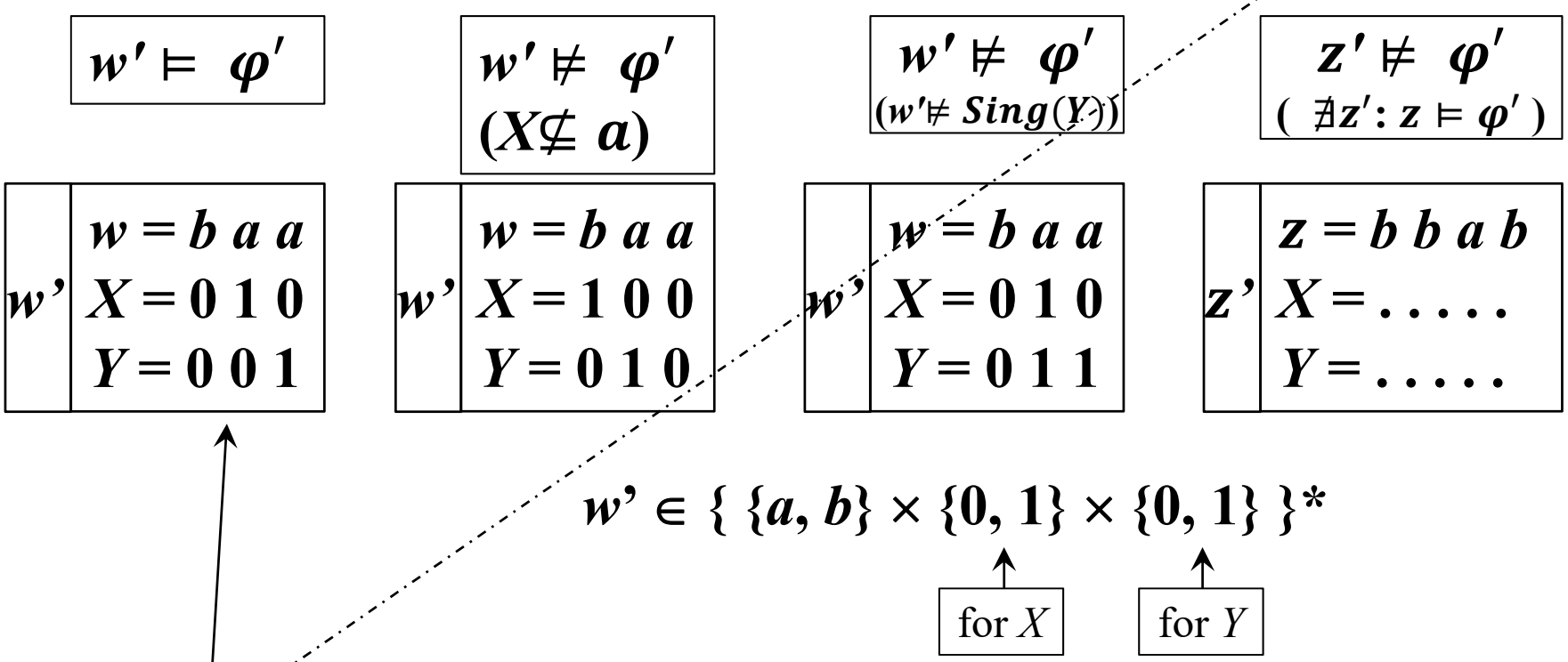
- an illustrative example: (regular) language L of strings $w \in \{a, b\}^*$ that include 2 consecutive a's (i.e., $L = \{a, b\}^* \, aa \, \{a, b\}^*$)
  - costruction of FA recognizing L in [Thomas96] and [Wehr07]: a (technically complex) proof by induction on the structure of φ
- *MSO* formula: $\varphi = \exists x \, \exists y \, S(x, y) \wedge a(x) \wedge a(y)$ **procedure in 4 steps**

1. f.o. variables x, y turned into second order: $X, Y \subseteq U = [\, 0 \,..\, |w|\text{-}1 \,]$
   - X, Y represent variables: we introduce in the s.o. logic the ***Sing*** predicate (singleton, a derived predicate) to state that they are true for only one value
   - s.o. logic includes (as a derived predicate) '$\subseteq$' with the usual meaning
   - formula φ becomes (*NB*: in the formula below a denotes a predicate!)

$$\varphi' = \exists X \, \exists Y \, S(X, Y) \wedge Sing(X) \wedge Sing(Y) \wedge X \subseteq a \wedge Y \subseteq a$$

Büchi's theorem - «formula ⇒ automaton», part 2/4

2. string $w \in \{a, b\}^*$ enriched with 2 components for X and Y, obtain an «enriched string» w' that includes elements for evaluating X and Y, hence it can be used to evaluate φ'



***e.g.*, $Y = 0\ 0\ 1$ encodes «$2 \in Y$» *and* it means: «the value of y is 2»**

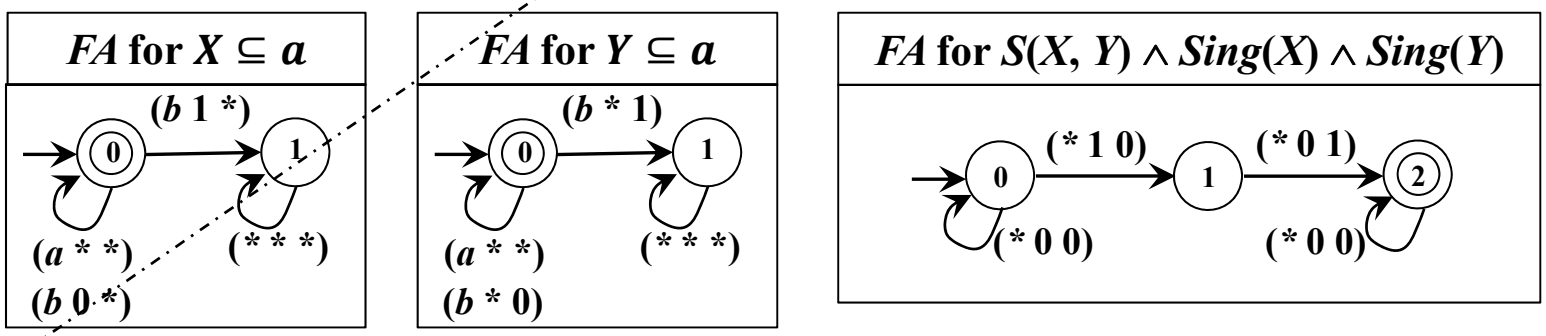
Büchi's theorem - «formula ⇒ automaton», part 3/4

3. build automaton ***A'***
   - with input alphabet $I' = \{a, b\} \times \{0, 1\} \times \{0, 1\}$

     for X for Y
   - ***A'*** accepts ***w'*** iff $w' \vDash \varphi'$
   - ***A'*** built from automata for subformulas

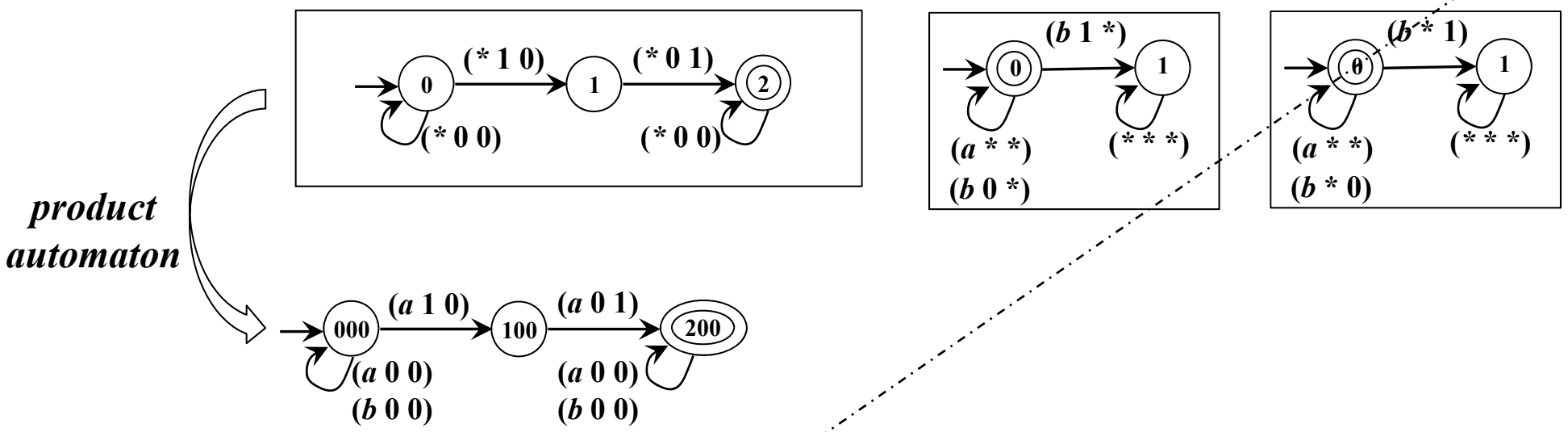
     $$X \subseteq a, \quad Y \subseteq a, \quad S(X, Y) \wedge Sing(X) \wedge Sing(Y)$$

     exploiting closure properties of FA under set-theoretic operations

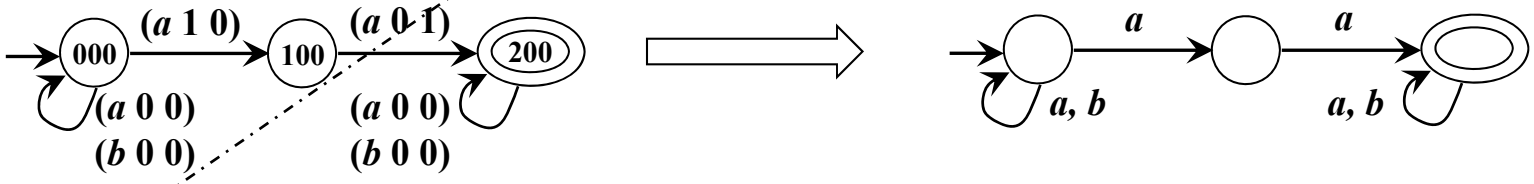


NB: in the above FA's, '*' means «any value», e.g. «(b 1 *)» stands for «(b 1 0) or (b 1 1)»

Büchi's theorem - «formula ⇒ automaton», part 4/4



4. obtain, by «projection», an automaton (in general, nondeterministic) ***A*** that accepts ***w*** if and only if ***A'*** accepts ***w'***



2. Logic to define program properties

- The alphabet includes sets, relations and operators that are in the basic repertoire of programming languages
- Specification of a search algorithm:
  The logical variable ***found*** must be true if and only if there exists an element of the array ***a***, having ***n*** elements, equal to the searched element ***x***:

$$found \leftrightarrow \exists i(1 \le i \le n \wedge a[i] = x)$$

- Specification of an algorithm to reverse an array
  - (output ***b*** contains the same elements as input ***a***, in the reverse order):

$$\forall i\left(1 \le i \le n \;\rightarrow\; b[i] = a[n - i + 1]\right)$$

In more general terms

{Precondition: *Pre*}
Program - or program fragment - *P*
{Postcondition: *Post*}

P must be such that: ***if*** *Pre* holds before the execution of *P* ***then*** *Post* holds after its execution:

- Search in an ordered array:

$$\{\forall i(1 \le i < n \rightarrow a[i] \le a[i+1])\}$$

$$P$$

$$\{found \leftrightarrow \exists i(1 \le i \le n \wedge a[i] = x)\}$$

NB: this does not at all mean that *P* must be a binary search algorithm (or any other one exploiting the ordering in the array). It only means that the ***implementer*** of *P* ***may*** exploit the fact that before the execution of *P* the array is ordered. A sequential search algorithm would be correct w.r.t. this specification. Instead, a binary search algorithm would not be correct w.r.t. a specification having as a precondition simply ***true*** (i.e. with no assumption on the input array).

- Sorting an array of ***n*** elements with no repetitions:

  $\{\neg \exists i, j (1 \le i \le n \wedge 1 \le j \le n \wedge i \ne j \wedge a[i] = a[j])\}$

  $SORT$

  $\{\forall i \left(1 \le i < n \rightarrow a[i] \le a[i+1]\right)\}$

  Is this an adequate ***specification***?
  (Let us think of the analogy: "specification = contract")

- One should not give anything for granted: what about the following ***implementation***:

```
for (k=1; k<=n; k++) a[k]=k;
```

  does it "***satisfy the contract***"?

- An alternative specification
- NB: use variable ***b*** to denote the array before the sorting operation

$$\{\neg\exists i, j(1 \le i \le n \ \wedge\ 1 \le j \le n \ \wedge\ i \ne j \ \wedge\ a[i] = a[j]) \quad \wedge \quad \%\ no\ duplicates\ in\ a$$

$$\forall i\left(1 \le i \le n \rightarrow a[i] = b[i]\right)\}$$

$$SORT$$

$$\{\forall i\left(1 \le i < n \ \rightarrow\ a[i] \le a[i+1]\right) \quad \wedge$$

$$\forall i\left(1 \le i \le n \ \rightarrow\ \exists j\left(1 \le j \le n \ \wedge\ a[i] = b[j]\right)\right) \quad \wedge\ \%\ all\ a[i]\ are\ in\ some\ b[j]$$

$$\forall j\left(1 \le j \le n \ \rightarrow\ \exists i\left(1 \le i \le n \ \wedge\ b[j] = a[i]\right)\right)\} \quad \%\ all\ b[j]\ are\ in\ some\ a[i]$$

- If we eliminate the first line in the precondition, is the specification still satisfactory?
  - (consider, in case of duplicates, their number in the array before and after the sorting)
- In fact, even a well-known, intuitive notion like sorting may be subject to misunderstandings
- In case of critical applications (having important implications on safety, or economic issues) the precision of formal notations is essential for writing good specifications

3. Mathematical logic for specifying system (timing) properties

- distinguished (integer- or real-valued) variables ***t***, ***t'***, t_1, etc. denote time points
- predicates with a time variable as argument denote «facts» that may or may not hold at given time instants
- "If I push the button the light goes on within Δ time units (t.u.)":
  - ***P_B***(***t***): a predicate denoting Push Button at time ***t***
  - ***L_On***(***t***): a predicate denoting that the Light is On at time ***t***

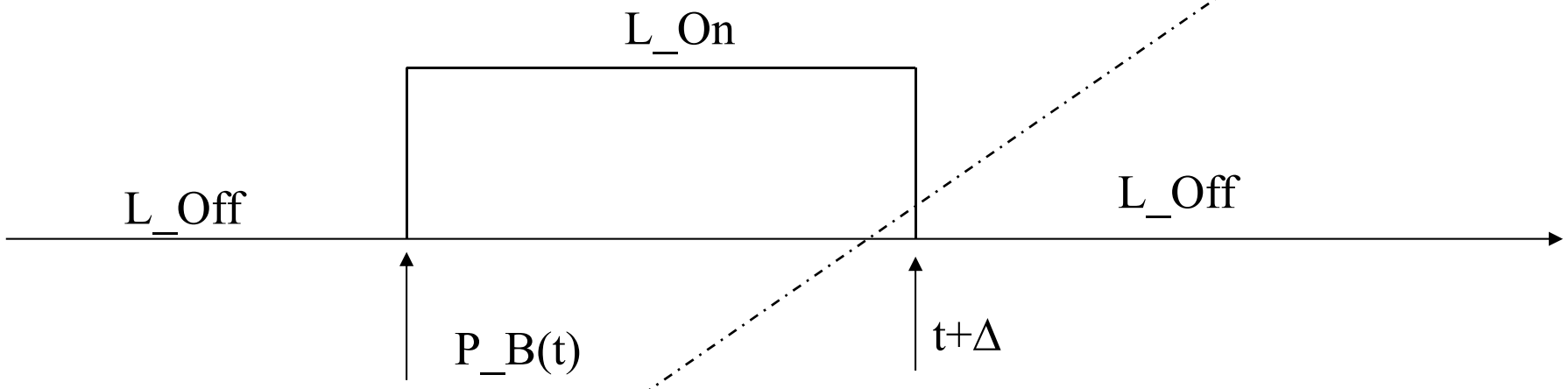
$$\forall t(P\_B(t) \rightarrow \exists t_1((t \leq t_1 \leq t + \Delta) \wedge L\_On(t_1)))$$

This specification is correct but the system requirement is not realistic.

-nondeterminism

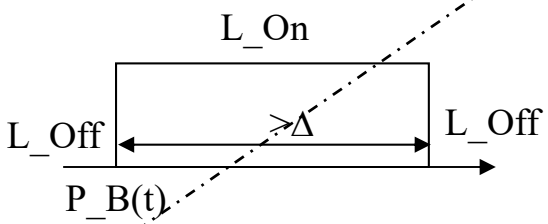
-timed lamps usually stay on for a given time interval

- If the button is pushed the light is on *for* Δ t.u. (similar requirements for an alarm or an electronic safe): here is a typical ***intended interpretation*** (assuming ∀t(L_On(t)↔¬L_Off(t)))

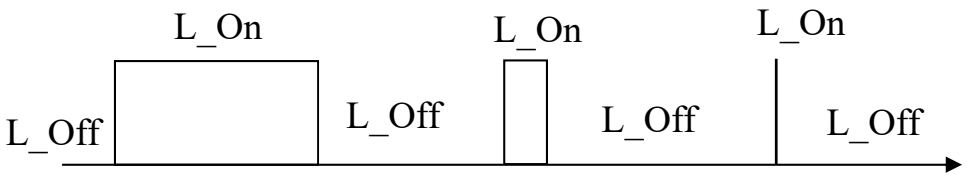


$$\forall t\left(P\_B(t) \rightarrow \forall t_1\left((t \le t_1 \le t+\Delta) \rightarrow L\_On(t_1)\right)\right)$$

- But here are some ***unintended*** interpretations



lamp is on for longer time

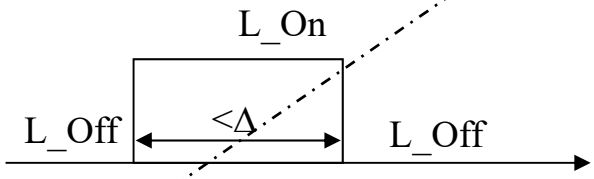


spurious lamp activation (no button press)

- Previous formula provides a ***sufficient*** condition of the lamp to be on. But using only sufficient conditions we get spurious L_On values
- We need a necessary and sufficient condition: try this

$$\forall t\left(P\_B(t) \leftrightarrow \forall t_1\left((t \le t_1 \le t + \Delta) \rightarrow L\_On(t_1)\right)\right)$$

- But this still admits spurious intepretations, because an interval with L_On implies P_B only if it lasts at least Δ t.u.



- A correct necessary condition for the light to be on, to be added to (conjoined with) the sufficient condition in the previous slide):

$$\forall t\left(L\_On(t) \rightarrow \exists d\left((0 < d < \Delta) \wedge P\_B(t-d)\right)\right)$$

Variations on the theme:

- Button for turning off combined with timeout
- Light maintained on by button pressure
- Opening closing of tents/windows/gates, ...
  - keep button pressure or not
  - interrupt movement or not
  - …..

- Logic approach for specification is extremely flexible but needs some method, to be effectively mastered

- Towards specification *languages* and *methods*

Theory of computation

- Which problems can we solve
  - With some (any given) type of machine
  - In the broadest possible meaning
- At first sight the question seems too general:
  - What do we mean by “problem”?
    A mathematical computation; taking a decision in a meeting of people; cash withdrawal from an ATM …?
  - Which abstract machines should we consider?
  - What does it mean to be able to solve a problem:
    If I am not able to solve the problem by some means I might be able to solve it by some other one.

In fact we can formalize the problem in its broader generality

- The notion of language allows us to formalize any "computer science problem" :
  $x \in L$? (Language recognition problem)
  $y = \tau(x)$? (Function computation problem)
  The two above formulations can be reduced to each other:
  - If I can find a machine to solve the problem of computing any function $y = \tau(x)$ and I wish to use this to solve the problem $x \in$ L, it suffices to define the *predicate* function $\tau(x) = 1$ if $x \in L$, $\tau(x)=0$ if $x \notin L$.
  - Viceversa, if I wish to compute function $y = \tau(x)$ I could define the language

$$L_\tau = \{x\$y \mid y = \tau(x)\}$$

  - Assuming that I can recognize L_τ using some machine, then, for a fixed x, I could enumerate all possible strings y over the output alphabet and for each of them ask the machine if $x\$y \in L_\tau$: soon or late, ***if $\tau(x)$ is defined***, I will find the string for which the machine answers positively: this is a way to compute $y=\tau(x)$.
    The procedure is "a bit long" but at the moment we are not concerned about the length of computations

- Concerning the machine … in fact there exist many, besides the ones we know; and many more can be invented, so we might be able to get results of the kind

  $\{a^n b^n | n > 0\}$ is accepted by a PDA and a TM but not by a FA.

- However we noticed that it is not so easy to overcome the computing power of the TM: adding tapes, heads, nondeterminism, … does not increase the power, (i.e., the class of accepted languages);
  It is not so difficult to have the TM do what a normal computer does: It suffices to simulate the memory of one of the two by the other one

HENCE

there is a ultimate generalization:

Church Thesis (back to 1930's!)

- There does not exist a computational device more powerful than the TM or any other formalism that is equivalento to it
  It is not a theorem (in principle, it should be checked every time anyone comes up with a new computational model)
- No *algorithm*, independently from the tool used to implement it, can solve problems that cannot be solved by a TM: the TM is the most powerful computer that we have and will ever have!
- Then the question "Which are the problems that can be solved algorithmically (or, with an equivalent term, "automatically")?" can be answered:
  These are the problems that can be solved by the (relatively simple) TM

Then let us focus on the TM

- Are there problems that cannot be solved by a TM?
- How can we find them?

The answers that we find hold also for C programs, Java programs, supercomputers ….

First, relevant fact:

- We can *algorithmically enumerate* TM's
- Enumeration of a set S:
- $\mathcal{E}$: S $\longleftrightarrow$ $\mathcal{N}$
- Algorithmic Enumeration: $\mathcal{E}$ can be computed by an algorithm, hence (church Thesis…) by a TM
- Algorithmic Enumeration of $\{a, b\}^*$:
- {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, …}
  ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕
- {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ….}
  - Usually called the "lexicographical ordering"

- Now we define an algorithmic enumeration of TM's
- For simplicity, without loss of generality:
- Let us fix a unique alphabet A
  (in the examples |A| = 2, A = {0, 1})
- Single tape TM
- Let us ignore one state TM's … and consider those with two states:

	0	1
q_0	$\perp$	$\perp$
q_1	$\perp$	$\perp$

TM_0

	0	1
q_0	$\perp$	$\perp$
q_1	$\perp$	$<q_0, 0, S>$

TM_1

.....

- How many two-states TM's there exist ?
  $\delta: Q \times A \to Q \times A \times \{R,L,S\} \cup \{\perp\}$
- In general: how many functions of type f: $D \to R$ are there?
- $|R|^{|D|}$ (for each $x \in D$ we have $|R|$ choices}
- With $|Q| = 2$, $|A| = 2$, $(2 \cdot 2 \cdot 3+1)^{(2 \cdot 2)} = 13^4$ two-state TM's
- Let us sort these TM's: $\{M_0, M_1, \ldots M_{134-1}\}$
- Then let us sort similarly the $(3 \cdot 2 \cdot 3+1)^{(3 \cdot 2)}$ 3-state TM's and so on.
- We obtain an enumeration $\mathcal{E}$: {TM} $\longleftrightarrow$ $\mathcal{N}$
- $\mathcal{E}$ is algorithmic (or *effective*): we can write a C program (i.e., a TM…) which, given *n*, produces the *n*-th TM (for instance by providing a table defining δ) and vice versa, given a (table describing a) TM M, tells the position $\mathcal{E}(M)$ of M in the enumeration.
- $\mathcal{E}(M)$ is called the ***Goedel number*** of M, $\mathcal{E}$ a goedelization

- A further convention: since we are speaking of numbers from now on we identify (for what concerns computability)
- Solving a problem = computing a function $f: \mathcal{N} \rightarrow \mathcal{N}$
- f_y = function computed by the y-th TM
- NB: $f_y(x) = \perp$ by definition ***if*** M_y does not stop when it takes x as input
- Conversely we stipulate (with no loss of generality) the reverse
  - if $f_y(x) = \perp$ then it means that the y-th TM does not stop on x
  - it suffices to stipulate that any TM M_y that, on input x, stops in a **non final** state (that does not define any significant output value $f_y(x)$), is by convention equivalent to a TM that enters a new state and continues forever, for instance by moving indefinitely the head right
- Therefore, in conclusion, $f_y(x) = \perp$ ***if and only if*** M_y does not stop when it takes x as input

A brief digression on terminology (to avoid misunderstandings)

- for function f, being total/partial or computable/uncomputable are distinct (orthogonal) matters
- total (resp., partial) = defined for every (resp., undefined for some) value of its domain
- computable = there is a TM that computes it
- All combinations are possible
  - $\forall x \in \mathbb{N}, f_x$ is computable by definition

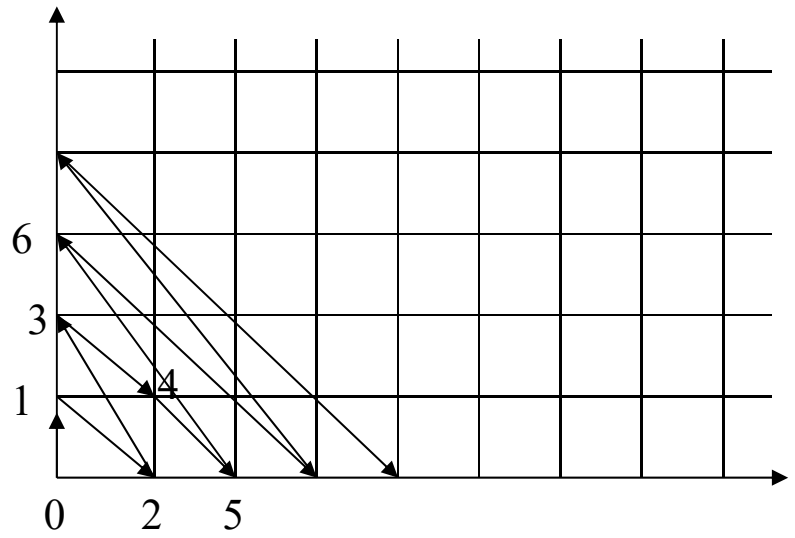
	total	partial
computable	1	2
not computable	3	4

1) function f(x)=x+1 is total and computable
2) the everywhere undefined function $f(x)=\perp$ for all x is partial and computable
   function $f(x)$ = if (x is even) then x else $\perp$ is partial and computable
3) the (predicate) function f(x)=if $f_x(y)=2\cdot y$ then 1 else 0 is total, not computable
4) $f(x)$= if (x is even and $f_{x/2}$ is total) then 1 elsif (x is even and $f_{x/2}$ is partial) then 0 else $\perp$
   is partial and not computable (this will be an easy exercise …)

  - 3) and 4) to be explained later… as well as many other nontrivial, interesting cases

Second relevant fact:

- There exists a ***Universal Turing Machine*** (UTM): the TM that computes the function $g(y, x) = f_y(x)$
- The UTM seems not to belong to the family $\{M_y\}$ because f_y is a function of one variable, while g is a function of two variables
- But we know that integer pairs can be effectively enumerated, i.e., $\mathcal{N} \times \mathcal{N} \leftrightarrow \mathcal{N}$: an example enumeration:



$$d(x,y) = \frac{(x+y)(x+y+1)}{2} + x$$

- So we can devise a suitable $g\hat{}(n)$ such that $g\hat{}(n) = g(d^{-1}(n))=g(y, x)$; $g(y,x)$ is encoded as $g\hat{}(n)$, with $n=d(y,x)$, i.e., $\langle y,x\rangle=d^{-1}(n)$ Notice that d and d^{-1} are both computable
- Sketch of the operation of the UTM that computes $g\hat{}$ (NB from now on for simplicity we will simply write g instead of $g\hat{}$):
  - Given n, the UTM computes $d^{-1}(n) =\langle y,x\rangle$
  - Then it builds the transition function of M_y (by computing $\mathcal{E}^{-1}(y)$) and stores it on some portion of a tape:

	$	q	a	q'	a'	S	$	..	..							

- In another tape portion is stores an encoding of the configuration of M_y

	#	0	1		0			..	1	q	1		1		0	#

NB: The special symbols #, $ and other ones are coded as binary strings

At the end the UTM leaves on the tape only $f_y(x)$ if and only if M_y terminates its computation on x

- The TM is a very abstract and simple model of a computer
- Let us pursue further the analogy:
- TM: computer with a single, built-in program
  An "ordinary" TM always executes the same algorithm,
  i.e., it always computes the same function
- UTM: computer with memory-stored program:
  y = program
  x = input to the program

Back to the question "which problems can be solved algorithmically?

Read very carefully textbook p.4-5!!

- How many and which are the computable functions $f_y: \mathcal{N} \to \mathcal{N}$?
- First, "how many" functions (not necessarily computable) are there?
- $\{f: \mathcal{N} \to \mathcal{N}\} \supseteq \{f: \mathcal{N} \to \{0,1\}\} \Rightarrow$
  $|\{f: \mathcal{N} \to \mathcal{N}\}| \geq |\{f: \mathcal{N} \to \{0,1\}\}| = |\wp(\mathcal{N})| = 2^{\aleph 0}$
- On the other hand, the set $\{f_y: \mathcal{N} \to \mathcal{N}\}$ of ***computable*** functions is by definition denumerable:
  NB: $\mathcal{E}: \{M_y\} \leftrightarrow \mathcal{N}$ induces $\mathcal{E}^{\wedge}: \mathcal{N} \to \{f_y\}$ not one-to-one (in many cases $f_y = f_z$, with $z \neq y$) but (*a fortiori*) it allows us to state that
- $|\{f_y: \mathcal{N} \to \mathcal{N}\}| = \aleph_0 < 2^{\aleph 0} \Rightarrow$
- "most" of the functions (problems) cannot be solved algorithmically!
- There are (very many) more problems than programs!

Is this such a bad shame?

- In fact, how many problems can be ***defined***?
- To define a problem we typically use a phrase (a string) of some language:
  - $f(x) = x^2$
  - $f(x) = \int_a^x g(z)dz$
  - "the number that multiplied by itself is equal to y"
  - …
- But any language (over some alphabet A) is a subset of A^*, which also is a denumerable set ⇒
- Hence the set of the problems that can be *defined* is also denumerable, just like the set of the problems that can be (algorithmically) solved
- Therefore we can still hope that they are the same
  Certainly {Solvable Problems} ⊆ {Definable Problems}

(BTW, a TM *defines* a function, besides computing it)

Next we turn again to the question:
"Which problems can be solved?"

- **<u>The problem of termination</u>**
  (it has quite "practical" implications):
  - One can build a program
  - One can provide input data for it
  - One knows that in general the program might not terminate its execution (in jargon: "run into a loop")
  - Can one determine if this will occur?
    - NB: determine **«in advance»**, not by running the program on the input and ...
- Stated in –completely equivalent– terms of TM:
  - Given (predicate) function $\boldsymbol{g(y, x) = 1}$ **if** $\boldsymbol{f_y(x) \neq \perp}$, $\boldsymbol{g(y,x) = 0}$ **if** $\boldsymbol{f_y(x) = \perp}$
  - Does there exist a TM that computes g? (i.e., is g computable?)

Answer: **NO**

- That's why a computer (which is a program) cannot warn us that the program we just wrote will run into an endless execution ***on a given input datum*** (while it easily signals a missing "}"):
- Determining if an arithmetic expression is well parenthesized is a solvable (decidable) problem;
- Determining if a given program will run into an endless execution on a given input is an algorithmically unsolvable (undecidable) problem [we will see many more ones: there are many things that a computer cannot do]

Proof

- It employs a typical *diagonal technique* (adopted also in the Cantor theorem to show that $\aleph_0 < 2^{\aleph 0}$)
- Let us ***assume*** (by contradiction) that the total function :

  $g(y,x) = 1$ if $f_y(x) \neq \perp$, $\quad g(y,x) = 0$ if $f_y(x) = \perp$

  is computable
- Then also the partial function

  $h(x) =$ 1 if $g(x,x) = 0$ (i.e., if $f_x(x) = \perp$),

  $\perp$ if $g(x,x) = 1$ (i.e., if $f_x(x) \neq \perp$)

  is computable

  NB: we went on the *diagonal* $y=x$, we changed the no answer ($g(x,x) = 0$) into a yes answer ($h(x)=1$), and we turned the yes ($g(x,x) = 1$) into a nontermination ($h(x)=\perp$), which can always be easily done by modifying the TM that (supposedly) computes g
- If h is computable then $h = f_{xh}$ for some xh.
- Question: $h(xh) = 1$ or $h(xh) = \perp$?

RECALL $h(x) = 1$ if $g(x,x) = 0$ (i.e., if $f_x(x) = \perp$), $h(x) = \perp$ if $g(x,x) = 1$ (i.e., if $f_x(x) \neq \perp$)

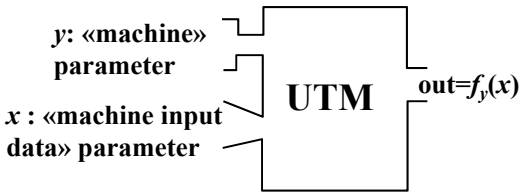
- Let us assume that $h(xh) = f_{xh}(xh) = 1$
- Then $g(xh,xh) = 0$, that is, $f_{xh}(xh) = \perp$:
- A contradiction

- Then let us assume the opposite: $h(xh) = f_{xh}(xh) = \perp$
- Then $g(xh,xh) = 1$, that is, $f_{xh}(xh) \neq \perp$:
- Another contradiction

then the assumption was wrong

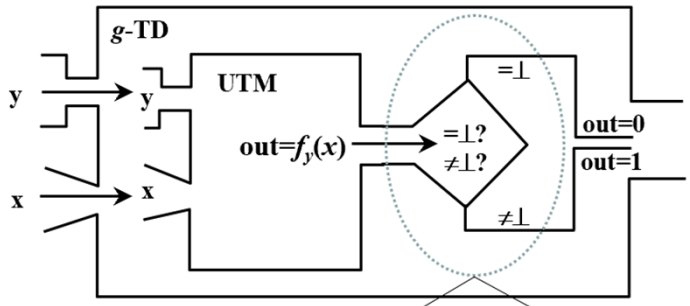
QED

PICTORIAL REPRESENTATION OF THE UNSOLVABILITY OF HALTING



$g(y,x)= 1$ if $f_y(x)\neq\perp$, $g(y,x)=0$ if $f_y(x)=\perp$

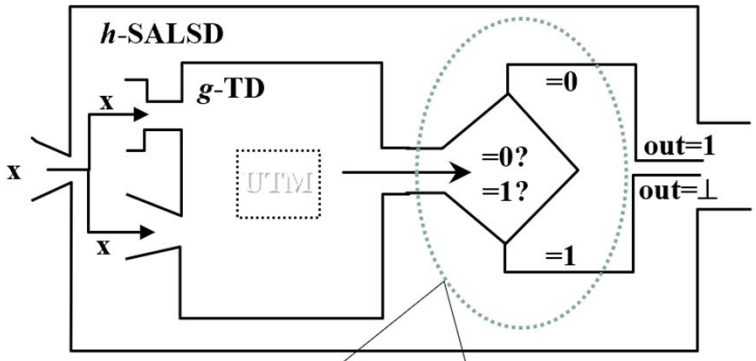
g is a "Termination Detecting" function (*g*-TD)



NB: unclear how this test would be possible: can't just launch the computation of $f_y(x)$ and wait to see what happens

We introduce function $h(x) = 1$ if $g(x, x)=0$, $h(x) = \perp$ if $g(x, x)=1$

h detects if a function, applied to itself, loops forever (the opposite of terminating), but if the function terminates *h* is undefined (the TM that computes *h* goes into a loop); we say that it "semidetects" loops: it answers only in case a loop occurs; therefore we call *h* "Self Application Loop Semi Detector" function (*h*-SALSD)



NB: this test is obviously, easily feasible

out=1 means: answer "YES", i.e., *x* applied to itself does not terminate

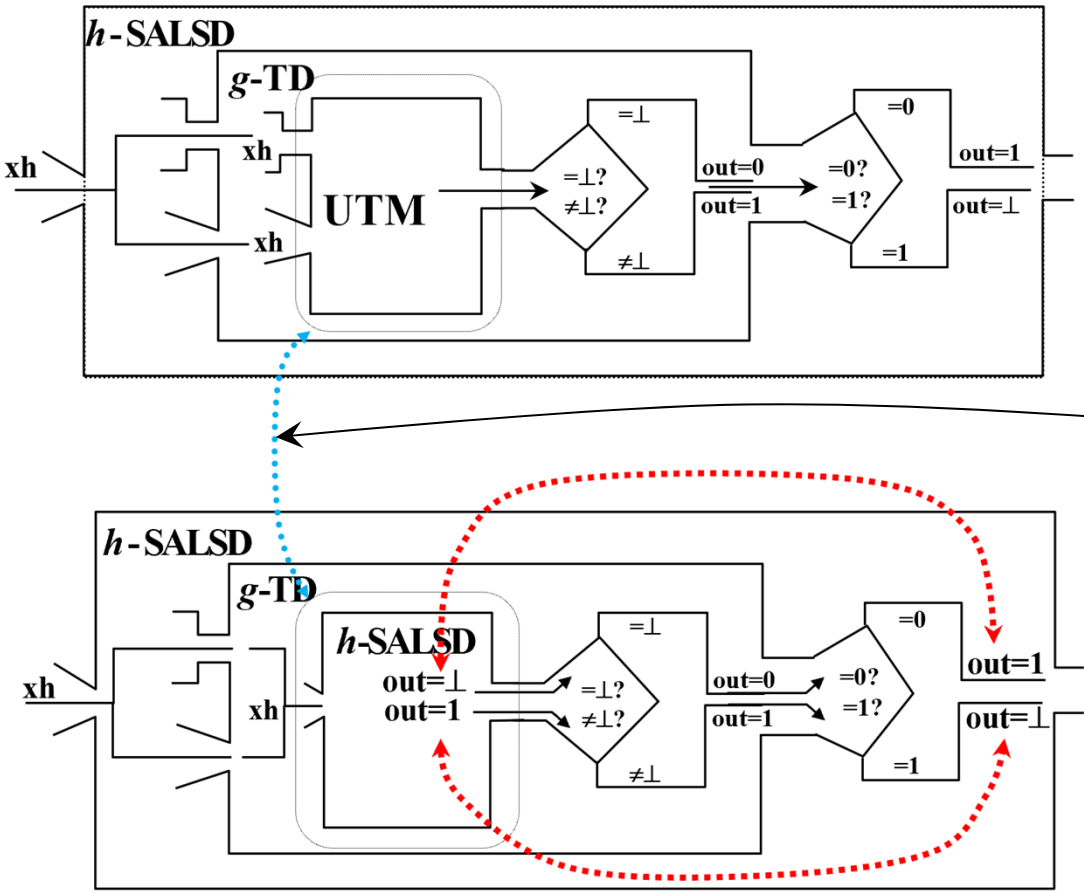
out=⊥ means: *h*-SALSD TM loops forever (provides no answer)

h-SALSD is a variant of *g*-TD that is very easy to compute ***if TD is computable***

assume *h*-SALSD = f_{xh} , i.e., *xh* is the Goedel # of *h*-SALSD

PICTORIAL REPRESENTATION OF THE UNSOLVABILITY OF HALTING

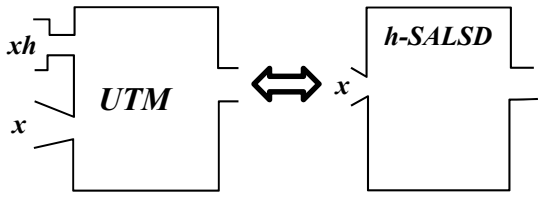
now let us apply *h-SALSD* to itself



applied to itself, *h*-SALSD:

- returns 1 iff TD says that SALSD applied to itself does not terminate
- returns ⊥ (i.e., its TM loops forever) iff TD says that SALSD applied to itself terminates

but the UTM applied to *xh* computes function *h*: $f_{xh}(xh)=h(xh)$, i.e., function *h*-SALSD



therefore a contradiction follows : **if $h(xh)=1$ then $h(xh)=\perp$**
if $h(xh)=\perp$ then $h(xh)=1$

- A first corollary of the unsolvability of the halting problem for the TM
  - The predicate (hence total function) characterizing the set of computable functions that are defined when applied to themselves : $\boldsymbol{h(x)=1}$ **if** $\boldsymbol{f_x(x) \neq \perp}$**;** $\boldsymbol{h(x)=0}$ **if** $\boldsymbol{f_x(x) = \perp}$ is not computable (proof on textbook, Corollary 2.7 p.168)
  - By itself the assertion is not very meaningful (but it will be so later, see slide 193)
  - Notice that $h(x)$ is a special case of function $g(y, x)$ (the function of slide 172)

    $g(y,x) = 1$ if $f_y(x) \neq \perp$, $g(y,x) = 0$ if $f_y(x) = \perp$

    [because $h(x)=g(x,x)$], and g was just proved to be uncomputable
  - Notice that the uncomputability of $h(x)$ is ***not*** a necessary consequence of the uncomputability of $g(y,x)$:
  - *NB: in general, if a problem is unsolvable, then a special case of it might be solvable (e.g., some properties that cannot be decided for any language can be decided for regular languages); instead a more general case of an unsolvable problem is necessarily unsolvable.*
    On the contrary if a problem is solvable, a generalization of it ***might*** *be unsolvable, while any specialization of it is certainly solvable.*

Another important unsolvable problem

- The (predicate, hence total) function
  $k(y) = 1$ if f_y is total, i.e., $f_y(x) \neq \perp \; \forall \; x \in \mathcal{N}$; $k(y) = 0$ otherwise
  is not computable (NB: this result is a trivial consequence of the coming Rice Theorem, because function k is a predicate characterizing total computable functions)
- NB: it is a problem *similar to but different* from the previous one. Here we have a quantification w.r.t. all possible input data.
  For this problem, **testing is useless**: In some cases one could be able to establish, for a large set of values of variabile x, that $f_y(x) \neq \perp$, without however being able to answer the question "is f_y a total function?" (Obviously, if one finds an x such that $f_y(x) = \perp$, one can conclude that f_y is not total, but what if one does not find it?).
  Vice versa, one could be able to conclude that f_y is not total and however be unable to decide whether $f_y(x) \neq \perp$ for a given single x. (However, if one was able to conclude that f_y is total there would be no doubt on whether $f_y(x) \neq \perp$ for a given x)
- From a practical viewpoint, this problem is perhaps even more relevant than the halting problem: given a program, one wants to know if it will terminate the execution *for every input* datum or if it may, *for some* datum, run into an endless execution. In the *problem of termination*, instead, one was interested to know if *a given program with some given input datum* would terminate.

Proof

- Standard technique: diagonal + contradiction, with some more technical detail.
- Hypothesis: $k(y) = 1$ if f_y *is total*, i.e., if $f_y(x) \neq \perp \; \forall \; x \in \mathcal{N}$; otherwise $k(y) = 0$ is computable and obviously, by definition, total
- Then define $g(x) = w$ = index (Goedel number) of the x-th TM (in $\mathcal{E}$) that computes a total function.
- If k is computable and total, then so is g:
  - compute $k(0), k(1), \ldots,$ let w_0 the first value such that $k(w_0) = 1$, then let $g(0) = w_0$;
  - then let $g(1) = w_1$, w_1 being the second value such that $k(w_1) = 1$; …
  - the procedure is algorithmic; furthermore, being total functions infinite in number, $g(x)$ is certainly defined for each x, hence it is total.
- g is also strictly monotonic: $w_{x+1} > w_x$;
- hence g^{-1} is also a function, strictly monotonic too, though not total: $g^{-1}(w)$ is defined only if w is the Goedel number of a total function.
- next define
  - (α) $h(x) = f_{g(x)}(x) + 1 = f_w(x) + 1$: f_w is computable and total hence so is $h \Rightarrow$
  - (β) $h = f_{w0}$ for some w_0; since h is total, $g^{-1}(w_0) \neq \perp$, let $g^{-1}(w_0) = x_0$ (hence $w_0 = g(x_0)$)

RECALL

- (α) $h(x) = f_{g(x)}(x) + 1 = f_w(x) + 1$: f_w is computable and total hence so is $h \Rightarrow$
- (β) $h = f_{w0}$ for some w_0; since h is total, $g^{-1}(w_0) \neq \perp$, let $g^{-1}(w_0) = x_0$ (hence $w_0 = g(x_0)$)

w_0 is the Goedel number of h

- What is the value of $h(x_0)$?
  - $h(x_0) = f_{g(x0)}(x_0) + 1 = f_{w0}(x_0) + 1$ (from (α))
  - $h = f_{w0}$ hence $h(x_0) = f_{w0}(x_0)$ (from (β))
- *Contradiction!*

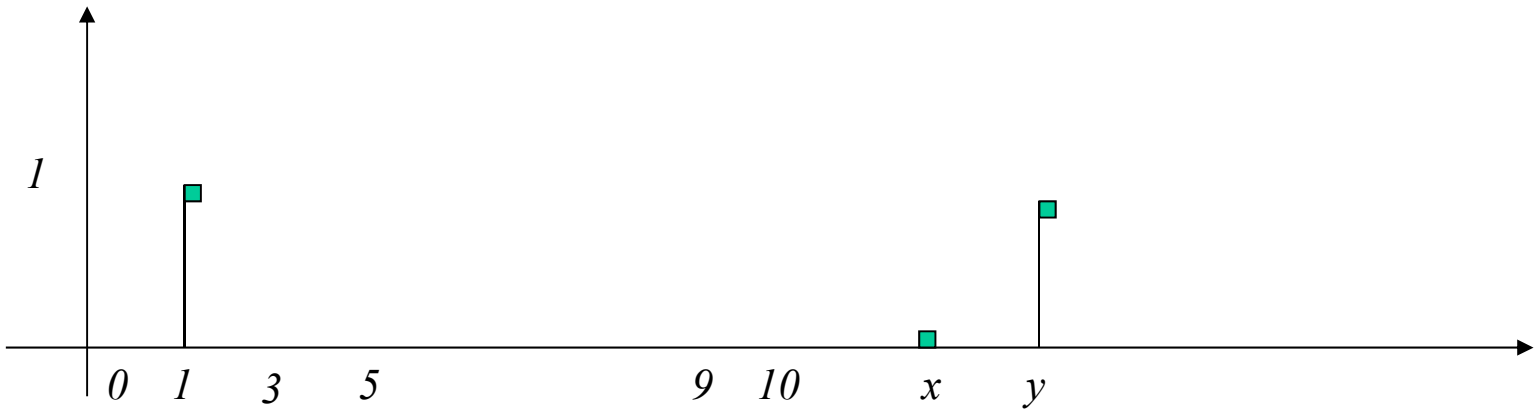
A crucial remark: knowing that a problem is solvable does not mean being able to solve it!

- In mathematics we often have non-constructive proofs: one shows that a mathematical object exists without providing a way to actually find (and exhibit) it
- In our case:
  - a problem is solvable if *there exists* a TM that solves it
  - for some problems we can reach the conclusion that *there exists* a TM that solves them, but, despite our knowledge of this, we are unable to find (build) it or we do not know which one it is in a set of TM that certainly includes the "right one"
- Let us start with a trivial case
  - The "problem" consists of answering a question with a yes/no answer (a so-called ***closed question***, whose answer does not depend on an input value/parameter):
    - Is it true that the number of atoms in the universe is $10^{10^{10^{10^{10}}}}$?
    - Is it true that the "perfect chess game" will end in parity?
    - (30 years ago ...) Is it true that $\neg\exists x, y, z, w \in \mathcal{N} (x^y + z^y = w^y \wedge y > 2)$? (proved in 1994)
    - ….

- In such cases one knows *a priori* that the answer is either ***Yes*** or ***No***, though one does not know (or did not until some point in time) which one it is
- This fact is less surprising if we consider that
  Problem = function; solve a problem = compute a function
  What function can one associate to the above problems?
  If one encodes TRUE=1; FALSE=0, *all* the above problems are expressed by one of the following two functions: $f1(x) = 1 \ \forall x$, or $f0(x) = 0 \ \forall x$
  Both functions are trivially computable (all constant functions are…), hence
  Whatever the answer is, it is ***computable***, though ***not necessarily known***.
- More abstractly, considering for instance function $g(y, x)$ of the halting problem:
  $g(10,20) = 1$ if $f_{10}(20) \neq \perp$, $g(10,20) = 0$ if $f_{10}(20) = \perp$
  $g(100,200) = 1$ if $f_{100}(200) \neq \perp$, $g(100,200) = 0$ if $f_{100}(200) = \perp$
  $g(7,28) = 1$ if $f_7(28) \neq \perp$, $g(7,28) = 0$ if $f_7(28) = \perp$
  ….
  Hence $g(10,20)$ (i.e.: does TM M_{10} stop on input 20?), $g(100,200)$, $g(7,28)$ *etc.* are all solvable problems, though we do not necessarily know the solution (i.e., the value of g for those arguments).

- Let us now consider less trivial and more instructive cases:
  - $f(x)$ = x-th digit of the decimal expansion of π.
    f is certainly computable (we know algorithms (TM's) to compute it)
  - Instead, *g(x) = 1 if somewhere in* π *there exist* ***exactly x*** *consecutive digits* 5,
    0 otherwise
    might be computable or not (NB: it is total). How can we try to compute ***g***?
  - Using our ability to compute f (we *currently* have no other knowledge)
  - By computing the sequence (notice that π=3.14159…)
    {f(0) = 3, f(1) = 1, f(2) = 4, f(3) = 1, f(4) = 5, f(5) = 9, …}

We get $g(1) = 1$ (i.e., there is a sequence of exactly 1 consecutive digit 5)
In general the plot of function g will be something like:



- For some value of x we might find that $g(x) = 1$;
- better, if $g(x)=1$, soon or late we will find it, if we are patient
- but what if $g(x)=0$?
  - Computing $f(y)$ for all $y \in [1..1000]$ or for all $y \in [1..10^{1000}]$ or … is useless
    - A side remark (forward pointing): the set $\{ x \mid g(x) = 1 \}$ is ***semidecidable***
- If the following conjecture
  "For every x, by producing a sufficiently long sequence of π, soon or late we will find exactly x consecutive 5's"

  was true, then g would be the constant function $g(x) = 1 \ \forall \ x$

  hence g would be computable
- Otherwise g could be some possibly very irregular function, maybe computable maybe not …
- In conclusion, *at the current state of the art*, we cannot conclude that ***g*** is computable, nor that it is not

- Now consider the following "slight" modification of g:
  $h(x) = 1$ *if in* π $\exists$ ***at least*** x *consecutive 5's,* 0 *otherwise*

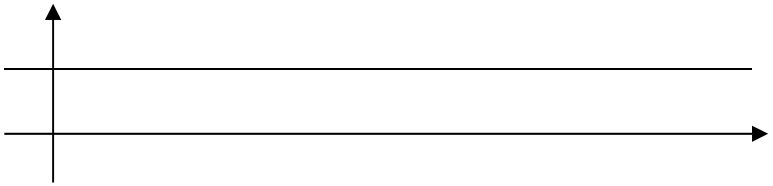
  (obviously, if $g(x) = 1$ then also $h(x) = 1$)

  Notice that, for any x,

if $h(x) = 1$, then $h(y)=1$ $\forall y \leq x$ (h is "downward closed for value 1"), and

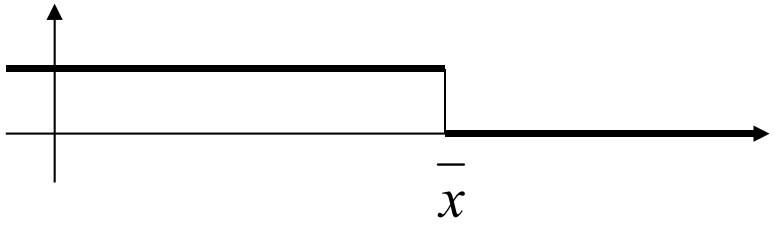
if $h(x)=0$ then $h(y)=0$ $\forall y > x$ (h is "upward closed for value 0")

  - Hence the plot of h is like one the following two:
  - 1)



($h(x) = 1 \ \forall \ x$)

  - 2)



$h(x) = 1 \ \forall \ x \leq \bar{x}$

$h(x) = 0 \ \forall \ x > \bar{x}$

- Then h certainly is in the following set of functions

$$\{h_{\bar{x}} \mid h_{\bar{x}}(x) = 1 \;\forall\; x \leq \bar{x} \wedge h_{\bar{x}}(x) = 0 \;\forall\; x > \bar{x}\} \cup \{\bar{h} \mid \bar{h}(x) = 1 \;\forall\; x\}$$

  Notice that each one of the functions in the above set is trivially computable (for any fixed $\bar{x}$ it is immediate to construct a TM that computes $h_{\bar{x}}$; similarly for $\bar{h}$)
- Hence h is certainly computable: there *exists* a TM that computes it
- Do we know which of the functions in the above set is h?

  No (at the moment): among the infinite TM's that compute the functions of the above set we do not know which is the right one

Decidability and semidecidability

Or: 1/2 + 1/2 = 1

- Let us focus on the problems stated in such a way that the answer is binary:
  Problem = " given set $S \subseteq \mathcal{N}$ and $x \in \mathcal{N}$, $x \in S$? "

  (NB: *all* problems can be (re)phrased in such a way,

  because they can all be viewed as a language - see slide 158)
- **characteristic function** or **characteristic *predicate*** of a set ***S***: it is the predicate characterizing the set

  $$c_S(x) = 1 \text{ if } x \in S,\ c_S(x) = 0 \text{ if } x \notin S$$

  (NB: c_S is total by definition)
- A set ***S*** is ***recursive*** (R) *or* ***decidable*** if and only if its characteristic function is computable
  - Note on terminology: It is also customary to say that solvable problems are ***decidable***

- ***S*** is ***recursively enumerable*** (RE) (or ***semidecidable***) if and only if:
  - S is the empty set, *or*
  - S is the ***image*** of a ***total, computable*** function g_S, the ***generating function*** of ***S***:

    $$S = I_{g_S} = \{x \mid x = g_S(y), y \in N\}$$

    notice that this implies

    $$S = \{g_S(0), g_S(1), g_S(2), g_S(3), \ldots\}$$

    the term "recursively (i.e., algorithmically) enumerable" comes from this "enumeration"
  - The term "semidecidable" can also be explained intuitively:
    given the question "$x \in S$?", if $x \in S$ then, *by enumerating the elements of S*, **soon or late** one finds x and is able to get a correct (yes) answer to the question;
    but what if $x \notin S$? In this case the above procedure does not work: using function g_S one continues indefinitely to generate elements of S without coming to a conclusion.
  - A formal characterization of this matter comes from the following ...

Theorem

- A) If S is recursive, it is also RE
  (i.e., decidable is more than –not less than- semidecidable)
- B) S is recursive if and only if both S itself and its complement $S^{\wedge} = \mathcal{N} - S$ are RE
  (two "semidecidabilities" make a "decidability"; or, when answering NO is equivalent to (i.e., it is equally difficult as) answering Yes
- (Corollary: the class of decidable sets (languages, problems, …) is closed under complement)
- Proof:

A): *S* recursive implies *S* RE

- If S is empty it is RE by definition
- Let us then assume $S \neq \emptyset$ and call c_s its characteristic predicate: note that, since $S \neq \emptyset$,

  $\exists\, k \in S$, that is $\exists\, k\, c_s(k) = 1$
- Let us define the generating function g_s as follows:
  $g_s(x) = x$ *if* $c_s(x) = 1$, otherwise $g_s(x) = k$ (the k above)
- g_s is total, computable (because so is c_s), and $I_{gs} = S$
- $\rightarrow S$ is RE
- NB: it is a *non-constructive* proof:
  do we know if $S \neq \emptyset$? not necessarily…
  We only know that if $S \neq \emptyset$ *there exists* g_s : this is enough for us!

B) *S* is recursive if and only if both *S* and *S*^ = $\mathcal{N}$-*S* are RE

B) equivalent to: (B.1 *S* recursive → both *S* and *S*^ RE) **and**
(B.2 both *S* and *S*^ RE → *S* recursive)

- B.1.1) *S* recursive → *S* RE (already proved in part A)
- B.1.2) *S* recursive → $c_S(x)$ (= 1 *if* $x \in S$; = 0 *if* $x \notin S$) computable
  → $c_{S^\wedge}(x)$ (= 0 *if* $x \in S$; = 1 *if* $x \notin S$) computable
  → *S*^ recursive → *S*^ RE
- B.2) *S* RE → construct the enumeration $S = \{g_S(0), g_S(1), g_S(2), g_S(3),...\}$
  S^ RE → construct $S^\wedge = \{g_{S^\wedge}(0), g_{S^\wedge}(1), g_{S^\wedge}(2), g_{S^\wedge}(3),...\}$
  But $S \cup S^\wedge = \mathcal{N}$, $S \cap S^\wedge = \varnothing$

hence $\forall x \in \mathcal{N}$, *x* belongs to exactly one of the two enumerations →
The following enumeration

$$\{g_S(0), g_{S^\wedge}(0), g_S(1), g_{S^\wedge}(1), g_S(2), g_{S^\wedge}(2), g_S(3), g_{S^\wedge}(3),...\}$$

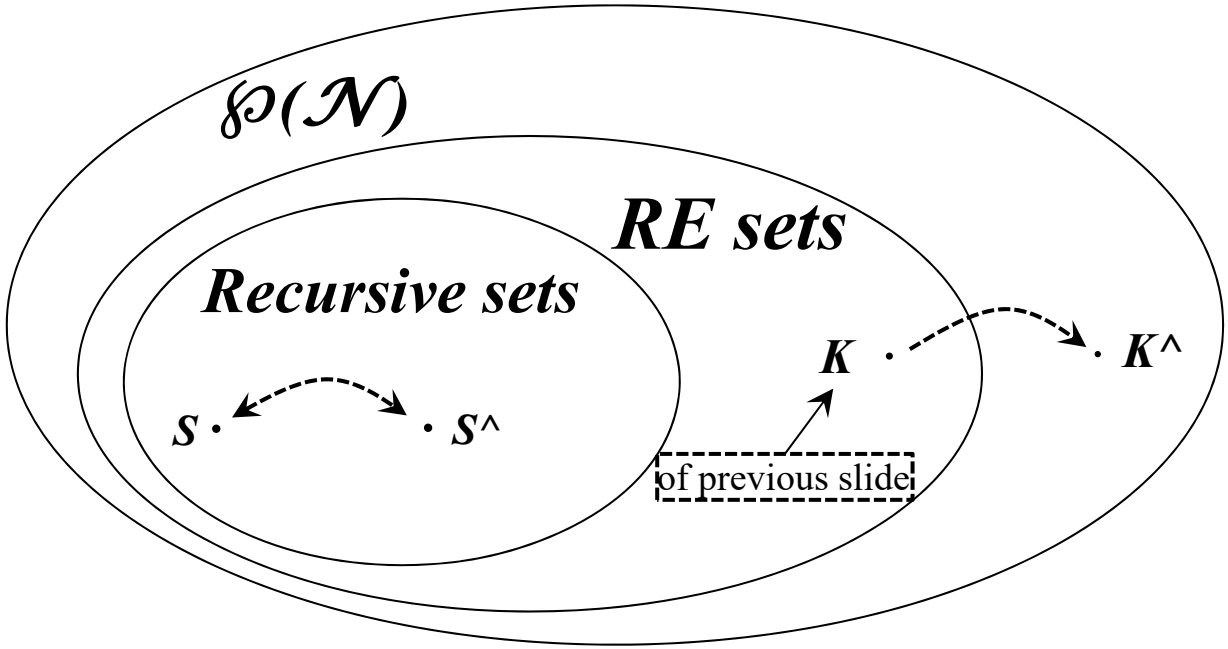
certainly includes any *x* in exactly one position: if *x* is at an odd position, then $x \in S$, if it is at an even position then $x \in S^\wedge$. Hence c_S can be computed.

Other very important results

- $\boldsymbol{S}$ is RE $\leftrightarrow$ $\boldsymbol{S} = \boldsymbol{D_h}$, with $\boldsymbol{h}$ computable and partial: $\boldsymbol{S} = \boldsymbol{D_h} = \{x |\ h(x) \neq \perp\}$
  and

  $\boldsymbol{S}$ is RE $\leftrightarrow$ $\boldsymbol{S} = \boldsymbol{I_g}$, with $\boldsymbol{g}$ computable and partial: $\boldsymbol{S} = \boldsymbol{I_g} = \{x |\ \exists y \in \mathcal{N} :\ x = g(y)\}$
- Proof is omitted here: it uses a quite useful and significant technique
- The above theorem allows us to view RE sets as characterizing precisely
  the languages ***recognized/accepted*** by the Turing Machines
  (NB not ***decided*** **: decide** and **recognize/accept** differ slightly)
- It can also serve as a Lemma to prove that:
- There exist semidecidable sets that are not decidable:

  $K = \{x |\ f_x(x) \neq \perp\}$ is semidecidable because $K = D_h$ with $h(x) = f_x(x)$.
  We know however that the characteristic function of K,
  ($c_K(x) = 1$ *if* $f_x(x) \neq \perp$, 0 otherwise) is not computable (see function $h(x)$, slide 178)
  $\Rightarrow K$ is not decidable
- Conclusion:



BTW: notice that the relation «is the complement of» among sets is symmetric

Inclusions are all strict

Corollary: the class of RE sets (i.e., languages recognized by TM's) is *not* closed under complement

Why? Because if RE sets were closed under complement then all RE sets would also be recursive (also their complement would be RE…), but we know (from the previous slide) that this is not the case

[a brief digression]

Undecidability of the Halting problem as a key to $\mathbf{R \neq RE}$

- Undecidability of halting is a fundamental fact (not just a bizarre anomaly)
  - It is a key to the distinction between **R** and **RE**
- Theorem: if Halting was decidable, then $\mathbf{R = RE}$
  - Suppose $\boldsymbol{S}$ is semidecidable: $\Rightarrow$ (slide 189) $\exists$ a TM $\boldsymbol{M_S}$ that ***semidecides*** or ***recognizes*** it (slide 193), that is, for any $\boldsymbol{x}$, $\boldsymbol{M_S}$ executed with input $\boldsymbol{x}$
    - terminates and accepts if $\boldsymbol{x \in S}$
    - does not terminate if $\boldsymbol{x \notin S}$
  - if Halting decidable $\Rightarrow$ possible to tell whether $\boldsymbol{M_S}$ would halt on $\boldsymbol{x}$ or not ***in advance*** (without actually executing $\boldsymbol{M_S}$)
    - if (we know that) $\boldsymbol{M_S}$ halts on $\boldsymbol{x} \Rightarrow$ accept $\boldsymbol{x}$ (determine that $\boldsymbol{x \in S}$)
    - if (we know that) $\boldsymbol{M_S}$ does not halt on $\boldsymbol{x} \Rightarrow$ reject $\boldsymbol{x}$ (determine that $\boldsymbol{x \notin S}$)
  - therefore (under the assumption that halting is decidable) $\boldsymbol{S}$ would be not only RE but also R

The mighty Rice theorem

- Let $\boldsymbol{F}$ be a set of ***computable*** functions
  The set $\boldsymbol{S}$ of (the indices of) TM's that compute the functions of $\boldsymbol{F}$

  $$\boldsymbol{S} = \{ \boldsymbol{x} \mid \boldsymbol{f_x} \in \boldsymbol{F} \}$$

  is decidable if and only if $\boldsymbol{F} = \varnothing$ or $\boldsymbol{F}$ is the set of all computable functions
- $\Rightarrow$ (alas!) in *all non trivial cases* ($\exists f$ s.t. $f \in F$ and $\exists f$ s.t. $f \notin F$) S is not decidable!
- $\Rightarrow$ e.g., the following very interesting problems are unsolvable
  - Program correctness: does P solve a given problem (identified by a computable function f)? i.e., $F=\{f\}$, $P \approx M_p$ does $\boldsymbol{p}$ belong to the set $\boldsymbol{S} = \{ \boldsymbol{x} \mid \boldsymbol{f_x} \in \{ \boldsymbol{f} \} \}$?
  - Program equivalence (given $\boldsymbol{M_y}$, same problem as above, with the set $\boldsymbol{F}=\{\boldsymbol{f_y}\}$)
  - Does a program have any specified property concerning the function it computes (function with only even values in the image, function with a limited image, …)?
  - …
- There is an endless list of interesting problems whose unsolvability follows trivially from the Rice theorem

How can we, in practice, determine that a problem/set is (semi)decidable or not?

- If we find an algorithm that always terminates → decidable
- If we find an algorithm that may not terminate, but it always terminates when the answer is positive → semidecidable
- If we think that the problem/set is not (semi)decidable, how can we prove it?
- Do we have to build a new diagonal proof every time? … no way!
- There are easier means:
- A first, very powerful tool is the Rice theorem
  - Applies to many «problems» concerning computation, e.g., software features
- Though implicitly, we have already used another very natural and general technique:

Problem reduction

- If one has an algorithm to solve problem P one can use it to solve problem P':
  - TRIVIAL EXAMPLE: One can compute the *product* of two numbers *a* and *b* if one can compute the operations: *sum*, *difference*, *division by 2*, *square*. It suffices to use the formula $a \times b = ((a+b)^2 - a^2 - b^2)/2$. Hence *multiplication* is reduced to {*sum*, *difference*, *division by 2*, *square*}
  - In general if there is an algorithm that, given a instance of a problem P' builds its solution by producing (algorithmically) an instance of another problem P that is solvable, and such that from the solution of P one can obtain algorithmically that of P', then **P'** has been ***reduced to*** **P**.
  - Using the ***set inclusion formulation*** of a problem:
    - I want to solve $x \in S$'
    - I can solve $y \in S$ for all possible y
    - If I have a computable, total function t such that $x \in S' \leftrightarrow t(x) \in S$ then I can answer algorithmically the question $x \in S$'
    - i.e., I have reduced the problem $x \in S$' to the problem $y \in S$

- The method can work also in the opposite way:
  - I want to know ***if*** I can solve $x \in S$
  - I know I cannot solve $y \in S'$ (S' is *not* decidable)
  - If I reduce S' to S, that is,
    I find a computable total function t such that $y \in S' \leftrightarrow t(y) \in S$
    then I can conclude that $x \in S$ is not solvable (otherwise I could show that $x \in S'$ is solvable by reducing it to $x \in S$)
- In fact, we already used implicitly this way of reasoning several times:
  - From the undecidability of the halting problem for the TM we derived in general the undecidability of the problem of termination of any computation on any computer; for instance, concerning the termination of C programs:
    - Consider a TM M_y and an integer x
    - I can build a C program, P, that simulates M_y and I can store x in an input file f
    - program P terminates its computation on file f if and only if $f_y(x) \neq \perp$
    - If I could decide if P terminates its computation on f then I could solve also the halting problem for the TM.

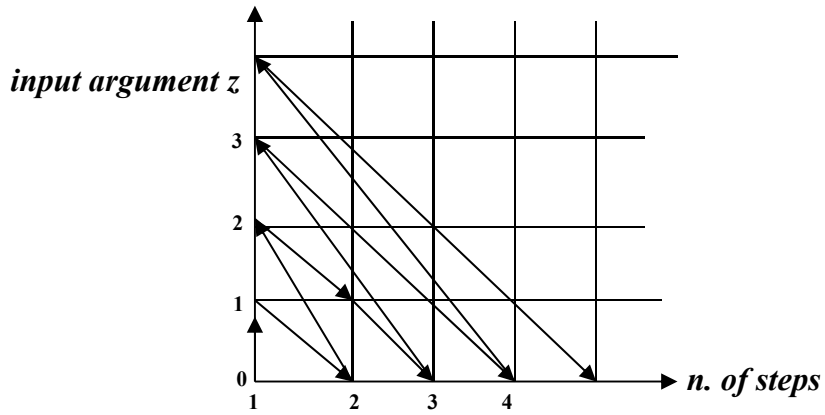
Reduction is a general, powerful technique

- Is the problem "does a generic program P access an uninitialized variable?" decidable?
  - Let us assume (by contradiction) that it is decidable and let us show by reduction that in that case the halting problem would be decidable
  - We can consider and instance P of the **halting** problem and **reduce** it **to** the following "**uninitialized variable access**" problem P^:
    - P^:
      begin var x, y: …
      P;
      y := x
      end
      making sure that identifiers *x* and *y* are "fresh variables", not used in P
    - It is clear that the assignment statement y := x results in accessing an uninitialized variable, because *x* and *y* do not occur in P
    - Hence the uninitialized variable *x* is accessed in P^ if and only if P terminates.
  - Then if I could solve the problem of "diagnosis of uninitialized variable access" then I could solve also the termination problem, which cannot be.

- The same technique can be applied to prove the undecidability of many other typical properties of program execution:
  - Array indices out of bounds
  - Division by 0
  - Dynamic type compatibility
  - …
  - Typical ***run time errors***: concerning this issue ...

- Let us consider again the previous examples
  - halt of the TM
  - Division by 0 and other run-time errors, …
- The related sets are undecidable, but they are semidecidable:
  a. if the TM stops, soon or late I find it;
  b. if there exists any datum x, input of a program P, such that P, executed on input x, eventually executes a division by 0, then soon or late I can find it …
- Let us make a digression on the latter problem:
  - How can I ensure the above result (b.):
    - if I start to execute program P on x and P does not stop on x, how can I find that P, executed on $y \neq x$, will execute a division by 0?

- In general: Theorem (abstract formulation of the various concrete cases above):
  - The set of values x for which $\exists\, z$ such that $f_x(z) \neq \perp$ is semidecidable
    (NB: here the "input parameter" of the problem is the TM index x)
  - Sketch of the proof
    - If I compute $f_x(0)$ and find it $\neq \perp$ then I can answer;
    - But what if the computation of $f_x(0)$ does not terminate and $f_x(1) \neq \perp$: how can I find it?
    - Then I use the following trick, known as ***dovetailing***:
      - I simulate 1 execution step of $f_x(0)$: if it stops, then I have answered positively the question;
      - Otherwise I can simulate one computation step of $f_x(1)$;
      - Again if it does not stop I can simulate 2 steps of $f_x(0)$; next 1 step of $f_x(2)$; 2 steps of $f_x(1)$; 3 of $f_x(0)$; and so on, according to the scheme in the figure:



This way, if $\exists\, z$ s.t. $f_x(z) \neq \perp$, eventually I find it because eventually I will simulate enough computation steps of $f_x(z)$ to terminate

- Concluding the digression:
- we have therefore a significant number of problems/sets (typically, related to run time errors in programs) that are not decidable, but are semidecidable.
- We must however pay attention to which is precisely the ***semi***decidable problem:
  - detecting the ***presence*** of the error (i.e., if there is one I can find it)
    - NB: the semidecidable set is the set of erroneous programs
  - Not its absence! The set in question is that of error-free programs
- Notice however that, since the complement $\neg\mathbf{s}$ of a set $\mathbf{s} \in$ (RE – R) is not even RE (otherwise they would both be decidable),

  The absence of errors (i.e., the ***correctness*** of a program with respect to an error) is not only not decidable, but it is not even semidecidable!
- Important implications on verification by ***testing***
  - Famous statement by Dijkstra: testing can prove the ***presence*** of errors, ***not*** their ***absence***

- Hence, as an additional result, we obtain a systematic technique to prove that a (unsolvable) problem is not RE: by proving that its complement is RE.

The complexity of computing

- We do not analyze individual algorithms (refer to courses on data structures and algorithms)
- We do not study advanced algorithms (refer to successive courses)
- Rather:
  - A critical examination of the problem and an approach to its solution
  - We strive for general principles
  - To be able to set individual problems in the correct framework

Complexity as a refinement of the notion of computability

- We are not satisfied to know that we can (algorithmically) solve a problem: we want to know how much it costs to solve it
- A critical analysis of the notion of costs (and benefits):
  - Cost of execution (necessary physical resources), in turn consisting of:
    - Time
      - compilation
      - execution
    - Space
  - Development cost
  - …
  - Objective and subjective evaluations, trade-off among contrasting goals …
  - … towards problems and approaches typical of Software Engineering
- We limit ourselves to notions of cost that are objective and can be formalized in a quantitative fashion: typical resources are ***memory*** and ***execution time***

We would like a hypothesis similar to the one on computability:

- The questions we ask and the answers we find do ***not*** depend on how we state the problem nor on the model we use to analyze it (Church Thesis).
- But:
  - Computing the sum in unary notation is quite different than in base k
  - Reducing the problem of computing a function $\tau(x)$ to that of deciding the language problem $x \in L_\tau = \{x\$y \mid y=\tau(x)\}$ could change completely the complexity: I might have to decide the language inclusion problem an unbound number of times to solve the original function-computing problem
  - Is it likely that, when the computer (or TM) changes, the execution times does not change? Obviously not, but...

- We will be able, by paying sufficient care, to obtain results that have a very broad validity ….
- … a sort of “Church Thesis for complexity”
- But for the moment …

… let us start with a complexity analysis for the TM

- Time complexity: let a computation be represented by a sequence of configurations (relation $\vdash$)
  $c = c_0 \vdash c_1 \vdash c_2 \vdash c_3 \ldots \vdash c_r$
  $T_M(x) = r$ if the computation stops at c_r, ∞ otherwise
- Space complexity:
  $c = c_0 \vdash c_1 \vdash c_2 \vdash c_3 \ldots \vdash c_r$

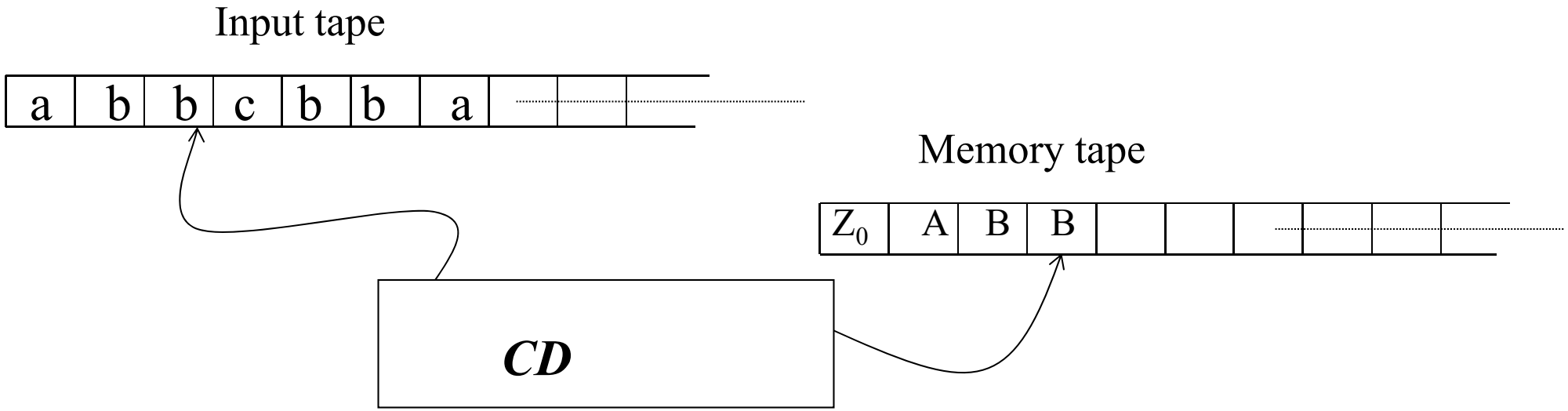
$$S_M(x) = \sum_{j=1}^{k} \max\{|\alpha_{ij}|, i = 1,\ldots,r\} \quad \alpha_{ij} = \text{content of tape j at i - th step}$$

NB: k is the number of tapes; sum of maximal occupation of each tape, possibly at different times

- NB, very relevant fact: $\forall\, x \;\; S_M(x) \leq k \cdot T_M(x)$

because $T_M(x)$ is also the distance the head can reach

A first example: accepting $\{wcw^R\}$, $w \in \{a,b\}^*$



$T_M(x) = |x| + 1$ if $x \in L$

$T_M(x) = |w|+1$ if $x = \overline{saucu^Rb}\ p$ (=> the TM rejects x upon reading the b after u^R)

and $w = saucu^R$, with $u,s,p \in \{a,b\}^*$

$T_M(x) = |x| + 1$ if $x \in \{a,b\}^*$ i.e., x does not contain any c

…

$S_M(x) = |x|$ if $x \in \{a,b\}^*$, $\lfloor |x|/2 \rfloor$ if $x \in L$, ...

- Too many details …
- useful/necessary?

- Let us try to simplify and focus on the essentials:

- From complexity as $\boldsymbol{f(x)}$ (x input string) to complexity as $\boldsymbol{f(n)}$ (n "size" of the input datum x):
  $\boldsymbol{n = |x|}$ (string length), or rows/columns of a matrix, or number of records in a file, …
  But in general $|x_1| = |x_2|$ does not imply $T_M(x_1) = T_M(x_2)$ (idem for S_M), therefore ...
- if $|x_1| = |x_2|$ but $T_M(x_1) \neq T_M(x_2)$ which one do we choose?

- Choice of the **worst case**:
  $T_M(n) = \max\{T_M(x), |x| = n\}$ (idem for $S_M(n)$)
- Choice of the **average case**:

$$T_M(n) = \frac{\sum_{|x|=n} T_M(x)}{k^n}, \quad k = \text{cardinality of the alphabet}$$

- We will mostly adopt the worst case:
  - Most relevant from an engineering viewpoint (for certain applications)
  - Mathematically, the simplest (the average case should consider probabilistic hypotheses on data distribution: e.g., names on a phonebook are not equally likely)

- We want a simple, concise, precise and practical way to indicate how a complexity function $f(n)$ grows with its argument n
- Use of the Θ notation to
  - estimate the ***dominant factor*** in the growth of a (complexity) function and its asymptotic behavior
  - Say when two different functions can be considered equivalent

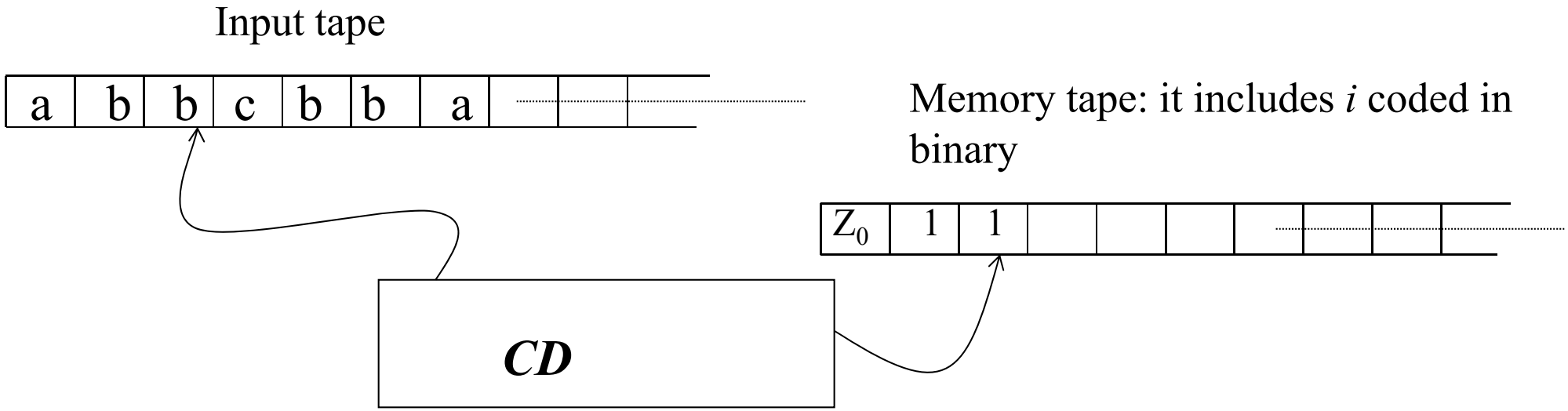
$$f \,\Theta\, g \leftrightarrow \exists c \; such\; that \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty$$

- Θ is an ***equivalence relation*** therefore…
- For any function $f(n)$ we may say that T_M is $\Theta(f)$, or $T_M \in \Theta(f)$

  ($\Theta(f)$ is viewed as an equivalence class) …
- For instance, taking $T_M(n)=f(n)=5n^3+3n^2$, we may say T_M is $\Theta(n^3)$, or $T_M \in \Theta(n^3)$
- (Saying that T_M is $\Theta(n)$ is like saying that $T_M(n)$ is linear?)
- Successively we will see that the Θ notation not only shows the dominant growth factor, but it also, in a way, describes the part that is "independent from the power of the computing device"

Let's return to the example $\{wcw^R\}$

- $T_M(n)$ is $\Theta(n)$, $S_M(n)$ is also $\Theta(n)$
- Can one do anything better?
- Concerning $T_M(n)$ it is difficult (in general one has to at least read all the input string)
- Concerning $S_M(n)$:



Store in the memory tape only the position i of the symbol to examine; then move the scanning head in position i and n-i+1 to compare the two read symbols ===>

- One gets: (let $n=|w|$)
- $S_M(n)$: $\Theta(\log(n))$ (input is ***not*** copied to the memory tape)
  but
- $T_M(n)$: $\Theta(n^2 \cdot \log(n))$: use 2 tapes T_1 (main counter) and T_2 (auxiliary counter),
  - $\forall i$, starting from 0
    - increment i (coded in binary on T_1) ($\Theta(\log(i))$ to implement $i := i+1$);
    - read next input symbol and store in the state
    - copy i on an auxiliary T_2 ($j := i$) ($\Theta(\log(i))$); then go to opposite end of input ($\Theta(i)$)
    - decrement (i times) j by 1 and move by 1 position the scanning head (towards the center) ($\Theta(i \cdot \log(i))$);
    - check for equality current input and symbol stored in the state
  - dominating factor is $\sum_{i=1}^{n} i \cdot \log(i) = n^2 \cdot \log(n)$
- A typical ***time-space trade-off***

- BTW: The example shows us why in the k-tape TM the scanning head can move in the two directions: otherwise one would lose important cases of sub-linear **spatial** complexity

Concerning k-tape TM ...

- Let us consider a change in the computing model:
  - For FA always $S_A(n)$ is $\Theta(k)$ and $T_A(n)$ is $\Theta(n)$, or even more precisely $T_A(n) = n$ (FA are ***real-time*** machines…);
  - For PDA always $S_A(n) \leq \Theta(n)$ and $T_A(n) \in \Theta(n)$ (number of ε-moves bounded *a priori*);
  - For ***single tape*** TM?
    - Accepting $\{wcw^R\}$ requires, at first sight, $\Theta(n^2)$ (head goes up and back n times)
    - Space complexity will never be $< \Theta(n)$ (which provides an additional explanation of choosing k-tape TM as the principal model)
    - Can one do better than $\Theta(n^2)$? NO: the proof is technically complex as it is often the case with non-trivial complexity lower bounds.
    - (NB: a PDA accepts $\{wcw^R\}$ in $\Theta(n)$)

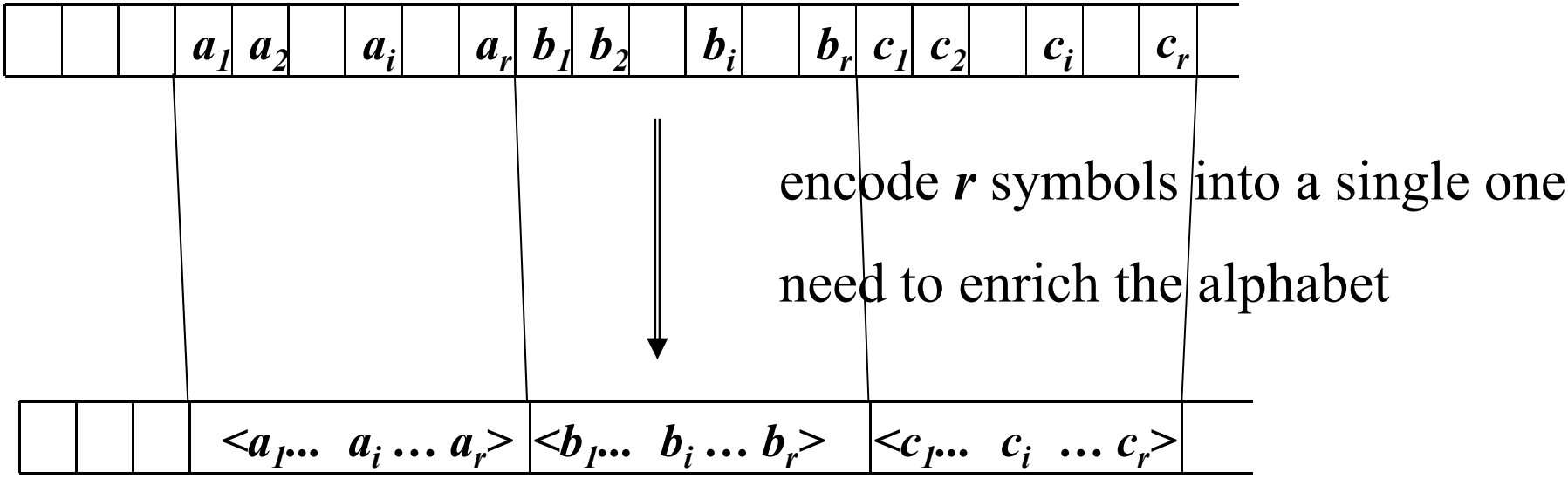
$\Rightarrow$

  - Hence, single tape **TM more powerful than PDA but sometimes less efficient**
  - What about von Neumann computers? We'll see in a while ...

Linear "speed-up" theorems

- If L is accepted by a k-tape TM M with complexity $S_M(n)$, then for each c > 0 one can build a k-tape TM M' with complexity $\mathbf{S_{M'}(n) < c \cdot S_M(n)}$

  (=> one can "reduce space")



$r \cdot c \geq 2$ **e.g. to reach half the complexity (c=1/2) one must take** $r \geq 4$

- If L is accepted by a **k-tape** TM M with space complexity $S_M(n)$, one can build a **1-tape** (NB: not a single tape) TM M' with complexity $S_{M'}(n) = S_M(n)$.

  (=> one can "reduce #tapes")

- If L is accepted by a k-tape TM M with space complexity $S_M(n)$, then for each $c > 0$ one can build a 1-tape TM M' with complexity $S_{M'}(n) < c \cdot S_M(n)$.

  (=> one can "reduce space + #tapes")

- If L is accepted by a k-tape TM M with ***time*** complexity $T_M(n)$, then for every $c > 0$ one can devise a (k+1-tape) TM M' with complexity
  $T_{M'}(n) = \max\{n+1, c \cdot T_M(n)\}$
  (=> one can "reduce time")
- Proof schema is the same as the one for spatial complexity (collapse *r* adjacent cells of M into one cell of M'), with some additional technical detail:
  - First, one must read and translate all the input (which requires *n* moves)
  - (This will create some problems within the class $\Theta(n)$)
  - Then M' can use a fixed number of moves (i.e., 6) to simulate groups of *r* moves of M
  - then by suitably choosing *r* one can reduce the complexity by an arbitrary (linear) factor

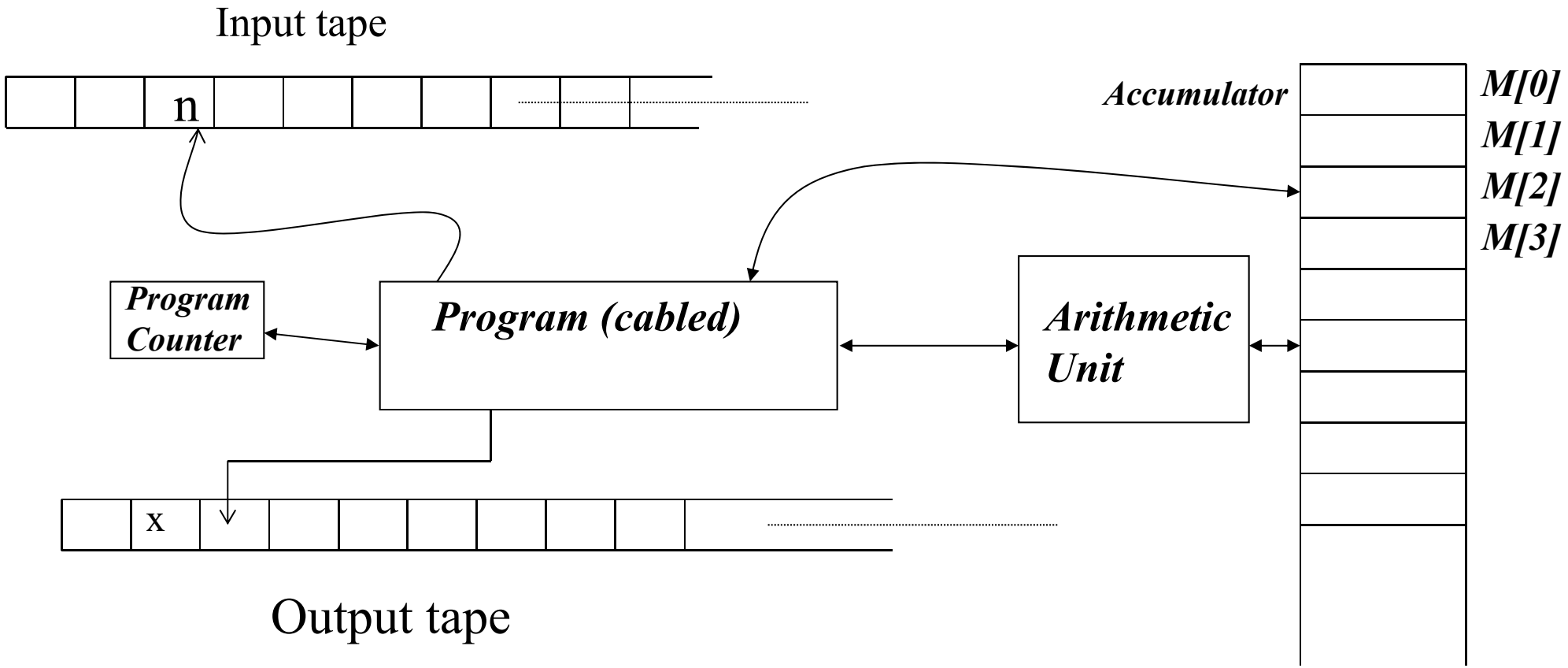
Practical implications of the linear speed-up theorems

- The proof scheme is valid for every computation model: also for real computers:
- This amounts to increasing the **physical parallelism** (from 16 bit to 32, to 64…)
- So, as long as one can increase the computing power in terms of available resources, one can increase *ad libitum* the execution speed
- This performance increase is however limited to improvements that are at most ***linear***: one cannot change the **order of magnitude** (i.e., the Θ class)
- Improvements in the order of magnitude can be obtained only by changing the algorithm and not in a mechanical/automatic way:
- E.g., for sufficiently large values of n sorting a sequence of n elements with merge sort will always be more efficient than sorting it through insertion sort or bubble sort, even if the more efficient algorithm is executed on a computer of modest power and the less efficient one on a supercomputer:
- Moral of the story: Brains can still overcome brute force!

Let us go back to comparing TM and real computers

- At first sight the comparison is unfair …:
  - To compute the sum of two numbers a TM needs $\Theta(n)$ (n is the length of –the string that encodes– the two numbers) while a computer provides this operation as an elementary operation (executed in a single machine step)
  - A computer can directly access any memory cell, while the TM has only a sequential access:
    - For instance, if we try to implement binary search through a TM we even get a complexity worse than linear, that is $\Theta(n.\log(n)) > \Theta(n)$
- We therefore cannot get along with complexity estimates that are bound only to the TM

A very abstract computer model: the RAM machine



Every cell contains an integer, not a symbol (NB!)

- The RAM instruction repertoir:

– LOAD [=, *] X	M[0] := M[X], X, M[M[X]]
– i.e., LOAD X	M[0] := M[X]
LOAD =X	M[0] := X
LOAD * X	M[0] := M[M[X]]
– STORE [*] X	M[X] := M[0], M[M[X]] := M[0]
– ADD [=, *] X	M[0] := M[0]+M[X], …
– SUB, MULT, DIV	…
– READ [*] X	
– WRITE [=, *] X	
– JUMP lab	PC := b(lab)
– JZ, JGZ, ... lab	
– HALT	

A RAM program computing the function
is_prime(n) = ***if*** *n is prime* ***then*** *1* ***else*** *0*

Consider in sequence all values from 1 to *n* and check whether they are divisors of *n*


```
        READ   1        The input value n is stored in cell  M[1]
        LOAD=  1        If n = 1, it is trivially prime ...
        SUB    1
        JZ     YES
        LOAD=  2        the counter in M[2] is initialized to  2
        STORE  2
LOOP:   LOAD   1        If M [1] = M[2]  (i.e., counter = n)  then n is prime
        SUB    2
        JZ     YES
        LOAD   1        If M [1] = (M[1] DIV M[2] ) * M[2] then
        DIV    2        M[2] is a divisor of M[1];
        MULT   2        hence M[1] is not prime
        SUB    1
        JZ     NO
        LOAD   2        the counter in M[2] is incremented by 1 and the loop is repeated
        ADD=   1
        STORE  2
        JUMP   LOOP
YES     WRITE=          1
        HALT
NO      WRITE=          0
        HALT
```

How much does it cost to execute the above RAM program?

- Obviously:
  - $S_R(n)$ is $\Theta(2)$
  - $T_R(n)$ is $\Theta(n)$
  - (Attention, however: what is n?? It is ***not*** the length of the input string!
    BTW, the input size is constant, independent of the value of n …
    Pay attention to the "data size" parameter!)
- Also obviously:
  - Accepting wcw^R:
    - $S_R(n)$ is $\Theta(n)$
    - $T_R(n)$ is $\Theta(n)$
  - Binary search: $T_R(n)$ is $\Theta(\log(n))$ (…assuming the set to be searched is in memory)
  - Sorting: …
  - … these complexity figures seem reasonable …
- However ...

Let us compute $2^{(2^n)}$ using a RAM (or any similar) machine

- read n;
- x := 2;
- for i :=1 to n do x := x*x;
- write x

- One gets $((2)^2)^2$) … n times, i.e., 2^{2^n}

- What is the time complexity?
- $\Theta(n)$
- Are we sure?!
- How can it be? As a matter of fact just to ***write*** the result in binary one needs at least 2^n bits (hence time 2^n) !

- The above analysis is definitely not realistic!

What's the problem? The RAM (and also the von Neumann machine) is a bit too...
abstract

- Can a memory cell (=memory unit) store an arbitrary number?
- Can an arithmetic operation (on ***any*** value) be a unit-cost elementary operation?
- This is correct only as long as the abstract machine is an adequate model of the actual machine (manipulated values are small compared to the numbers that can be encoded into 16, 32, 64, … bits)
- Otherwise … double precision etc. ---> the operations are not elementary any more and must be ***programmed***
- ---->
- Should we re-do all the algorithms and complexity analysis based on the adopted level of precision (i.e., number of bits)?
- Conceptually, yes, but in practice, and more easily:

- Logarithmic cost criterion: based on a "microscopic" analysis (see the "microcode") of the HW operations:

- What is the cost of copying the number *i* from a store cell to another one?
  As many elementary micro-operations as the number of bits necessary to encode *i* :
  log(*i*)
- What is the cost of accessing the store cell in the *i*-th position?:
  the opening of log(*i*) access "gates" to the same number of memory blocks
- What is the cost of executing the operation
  LOAD i ?
- …
- By means of a simple, systematic analysis we get the following ...
  for brevity *l*(*x*) stands for *log*(*x*) (but *l*(*x*) may as well be read as "***length***" of ***x***)

Table of logarithmic RAM costs

•	LOAD=	x	l(x)
•	LOAD	x	l(x) + l(M[x])
•	LOAD*	x	l(x) + l(M[x]) + l(M[M[x]])
•	STORE	x	l(x) + l(M[0])
•	STORE *	x	l(x) + l(M[x]) + l(M[0])
•	ADD=	x	l(M[0]) + l(x)
•	ADD	x	l(M[0]) + l(x) + l(M[x])
•	ADD *	x	l(M[0]) + l(x) + l(M[x]) + l(M[M[x]])
•	…		
•	READ	x	l(value of current input) + l(x)
•	READ*	x	l(value of current input) + l(x) + l(M[x])
•	WRITE=	x	l(x)
•	WRITE	x	l(x) + l(M[x])
•	WRITE *	x	l(x) + l(M[x]) + l(M[M[x]])
•	JUMP	lab	1
•	JGZ	lab	l(M[0])
•	JZ	lab	l(M[0])
•	HALT		1

Let us apply the new cost criterion

- To the computation of ***is-prime*(*n*)** (only main points)

M[1] stores *n*
M[2] stores the counter

LOOP:	LOAD	1	1+l(n)	
	SUB	2	l(n) +2 + l(M[2])	
	JZ	YES	l(M[0])	
	LOAD	1	1+l(n)	
	DIV	2	l(n) +2 + l(M[2])	
	MULT	2	l(n/M[2]) +2 + l(M[2])	($< 2 \cdot\ l$(n))
	SUB	1	l(M[0]) +1 + l(n)	($< 2 \cdot l$(n) +1)
	JZ	NO	… $\leq l$(n)	
	LOAD	2	… $\leq l$(n) + k	
	ADD=	1	...	
	STORE	2		
	JUMP	LOOP		

- In conclusion, one can easily put an upper bound on the cost of the individual loop iteration as $\Theta(\log(n))$
- Hence the overall time complexity is $\Theta(n \cdot \log(n))$

- Similarly we obtain:

- $\Theta(n.\log(n))$ for accepting wcw^R (NB: greater than the TM! Is it possible to do better?)
- $\Theta(\log^2(n))$ for binary search
- …
- ? overall (logarithmic criterion) cost = overall (constant criterion) cost * log(n)?
- ? overall (logarithmic criterion) cost = overall (constant criterion) cost * log(overall constant criterion cost)?
- Often, but not always:
- For the case of computing 2^{2^n} overall logarithmic cost $\geq 2^n$ (time complexity $\geq$ space complexity)
- Is there a criterion to choose which criterion to adopt?
  - Common sense (!):
  - If the computation does not change the order of magnitude of the input data, the initially (statically?) allocated memory may not change at run time ---> it does not depend on data ---> an individual cell can be considered elementary and with it all the relative operations ---> constant cost criterion is OK
  - Otherwise (computing a factorial, 2^{2^n}, "heavy" recursion, …) one needs the logarithmic criterion, the only one that is certainly correct.

Relations among the complexity measures obtained w.r.t. different computation models

- The same problem, solved with different kinds of machines can have different complexity
- It may happen that for P1 model M1 is better than model M2 but for P2 the opposite holds (binary search, direct access, sequential access and storing, accepting wcw^R)
- There is no model that is superior in all cases
- There is no result similar to the Church Thesis for complexity …
- However:
- It is possible to establish ***a priori*** at least a relation among the complexity of various computational models - an upperbound on the ratio of figures
- Polynomial Correlation Theorem (thesis) (analogy with the Church Thesis):
  - Under "reasonable" cost criterion hypotheses (the constant criterion for the RAM is not "reasonable" in all cases!) if a problem can be solved by a computation model M_1 with (space/time) complexity $\mathbf{C_1(}\boldsymbol{n}\mathbf{)}$ then there exists a a suitable ***polynomial*** $\mathbf{P_2}$ such that the same problem can be solved by ***any*** other computation model M_2 with complexity $\mathbf{C_2(}\boldsymbol{n}\mathbf{) \leq P_2(C_1(}\boldsymbol{n}\mathbf{))}$

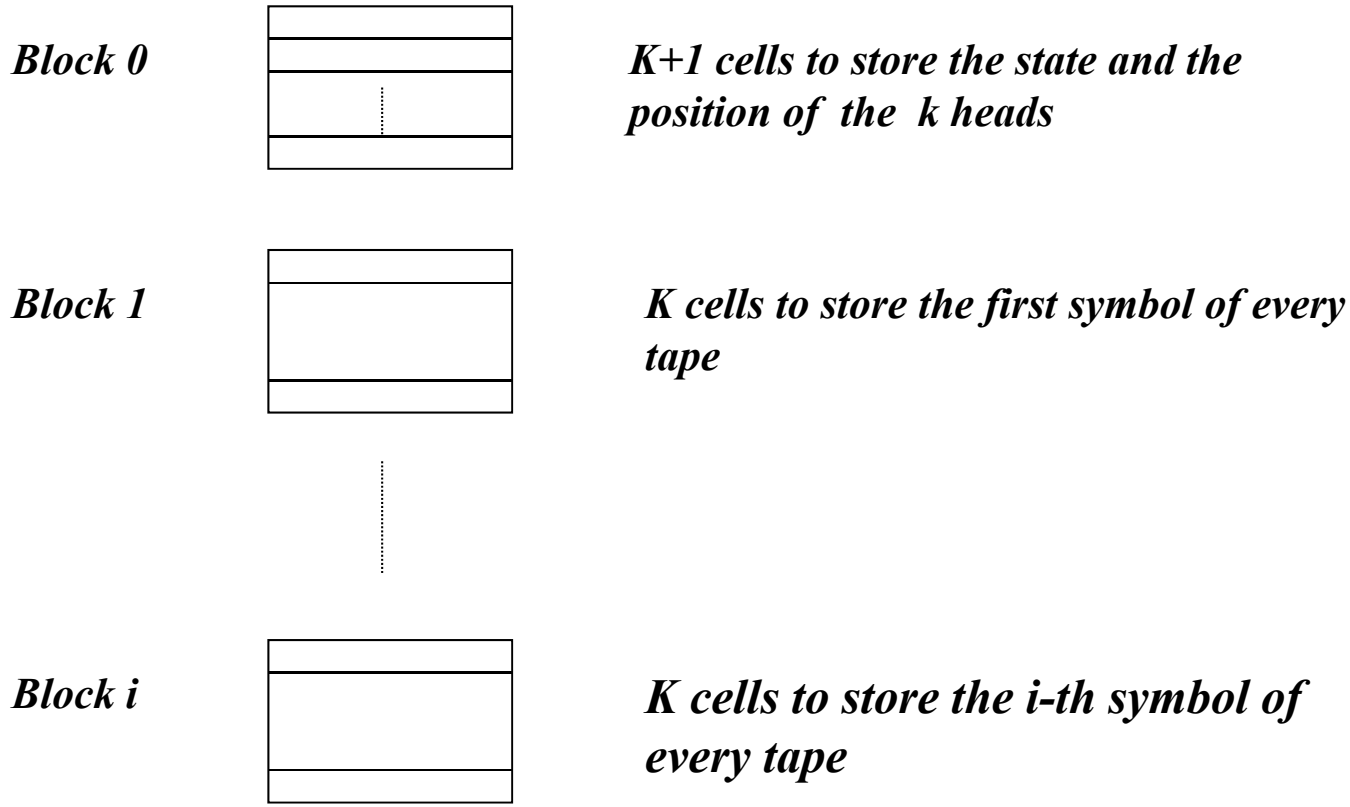
Before proving the theorem (not a thesis any more!) for the TM-RAM case, let us evaluate its impact:

- Though a polynomial can be as large as n^{1000}, it is always better than the exponential "abyss" ($\boldsymbol{n^k}$ against $\boldsymbol{2^n}$)
- Thanks to the polynomial correlation theorem we can speak of the class of problems that can be solved in polynomial time/space (not of the quadratic ones!): this class is not affected by the adopted model (it is ***invariant*** w.r.t. the model)
- Thanks to this result –and other important theoretical facts– the following analogy has been adopted:
  - Class $\mathcal{P}$ of practically "tractable" problems = class of problems solvable in polynomial time
  - $\mathcal{P}$ includes also the problems with complexity n^{1000} (better, at any rate, than the exponential ones), but practical experience shows that the problems with any practical applications (searches, paths, optimizations, …) that are in $\mathcal{P}$ also admit solutions with an acceptable degree (exponent) in the polynomial
  - (similarly we will see shortly that the complexity relation between TM and RAM is "close")

(Time) correlation between TM and RAM:

1: How a RAM can simulate a (k-tape) TM

- The RAM memory simulates the TM memory:
  1 RAM cell for each cell of the TM tape
  However, instead of using blocks of adjacent RAM memory cells to simulate each tape, we associate a -k cells- block to each k-tuple of cells taken for each position of the tape, + a "base" block:



- A move of the TM is simulated by the RAM:

Block 0

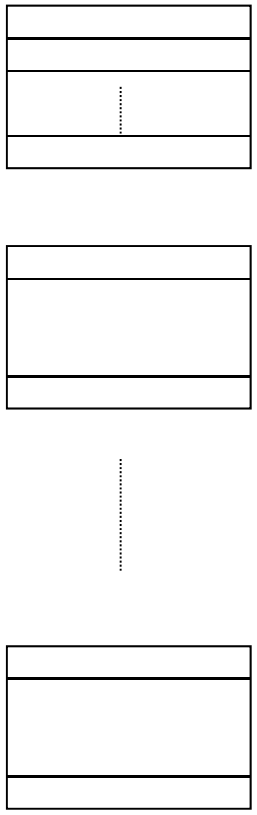
state and heads' positions

Block 1

k cells for first symbol on each tape

Block i

k cells for i-th symbol on each tape



- ***Reading:***
- The content of the 0 block is checked
  (a packet of k+1 accesses, c*(k+1) moves)
- k indirect accesses to k blocks to examine the content of the cells corresponding to the heads
- ***Writing:***
- The state is changed (in block 0)
- The content of the cells corresponding to the head positions are updated through indirect STORE operations
- The values of the k heads positions are updated (in block 0)

A move of the TM requires h*k RAM moves:

With a constant cost criterion: $T_R \in \Theta(T_M)$

With a logarithmic cost criterion [the "serious" one]: $T_R \in \Theta(T_M.log(T_M))$ (an indirect access to position *i* costs log(i), and the value of the position is bound by the time complexity, i.e., $S_M \leq T_M$)

Time correlation between TM and RAM: (Theorem 3.17 on the textbook)
2: How a TM can simulate a RAM
(in a simple but central case: accepting languages (hence no output tape) without using MULT nor DIV: the generalization is trivial)

- The TM has 3 tapes: the first tape encodes the content of the RAM memory:

.....	$	i_j	#	$M[i_j]$	$	.....	$	i_k	#	$M[i_k]$	$	.....

- NB:
  - The various RAM cells are kept ordered (i.e., $i_j<i_{j+1}$)
  - Initially the tape is empty ---> at a generic time it includes only the cells that have been assigned a value (through a STORE)
  - i_j and $M[i_j]$ are represented in binary encoding
- 2 further tapes:
  - A tape includes the "accumulator register" M[0] (in binary)
  - A "service" tape

- How a RAM step is simulated by the TM:

.....	$	i_j	#	$M[i_j]$	$	.....	$	i_k	#	$M[i_k]$	$	.....

- Let us consider a few representative examples:
- LOAD h:
  - Look for the value of h on the main tape (if it is not found: error)
  - The part next to h, M[h], it is copied into (the tape that stores) M[0]
- STORE h:
  - Look for h. If it is not found, "make a hole" using the service tape
  - Store h and copy M[0] in the part next to it (M[h]); copy the successive part from the service tape
  - If h already exists copy M[0] in the part next to it (M[h]); this can require the use of the service tape if the number of cells already occupied is not equal to that of M[0].
- ADD* h:
  - Look for h; look for M[h]; …
- With an easy generalization:
- ***Simulating a RAM move can take the TM a number of moves with an upper bound c*(length of the tape that stores the content of the RAM memory)***.

- Now a Lemma (Lemma 3.18 on the textbook):
  the length of the main tape has an upper bound equal to a function not greater than $\Theta(T_R)$

.....	$	i_j	#	$M[i_j]$	$	.....	$	i_k	#	$M[i_k]$	$	.....

Each "i_j-th cell" of the RAM requires in the TM tape $l(i_j) + l(M[i_j])$ (+2) tape cells.

Each "i_j-th cell" exists in the tape if and only if the RAM has executed at least a STORE on it.

The STORE costs for the RAM is $l(i_j) + l(M[i_j])$ ---->

To fill r cells, of total length

$$\sum_{j=1,r} l(i_j) + l(M[i_J])$$

the <u>RAM</u> needs a time (w.r.t.* logarithmic *cost criterion) that is <u>at least</u> proportional to the same value.

Hence to simulate a RAM move, the TM needs a time ***at most*** $\Theta(T_R)$;

a RAM move costs *at least* 1;

if the RAM has complexity T_R, the TM executes *at most* T_R moves ---> the complete simulation of the RAM by the TM costs *at most* $\Theta(T^2_R)$.

Some remarks and concluding warnings

- Pay attention to the "data dimension" parameter:
  - Input string length (absolute value)
  - Value of the datum (n)
  - Number of elements of a table, of nodes in a graph, of rows in a matrix, …
  - …
  - There are relations among these parameters, but they are not always linear (the number n requires an input string of length $\log(n)$).
- Does binary search implemented on a TM (whose complexity is $\Theta(n.\log(n))$) violate the polynomial correlation theorem?
- Pay attention to the hypothesis: accepting a language ---> data not already in memory ---> complexity at least linear.
- *in practice* the average case is often used (Quicksort, compilation, …) but not for critical, real-time applications

A brief digression on some advanced –but extremely relevant– aspects of computational complexity

- Some important questions:
  - Are there any lower bounds on complexity?
  - By increasing the complexity class does the class of solvable problems (always) increase?
    (i.e., do I get more by spending more?)
  - Does there exist a sort of "universal complexity class" (all solvable problems belong to that class)?
  - Does it make any sense to consider the complexity of nondeterministic computing models? If yes, how can this complexity be defined?
  - Does the introduction of nondeterminism change the complexity of solving problems?
  - ...

Let us focus on the complexity of nondeterministic computations

- *First of all*: how can it be defined?
  - w.r.t. the fastest computation?
  - w.r.t. the slowest?
  - What if some computations terminate and others do not?
  - w.r.t. only the accepting computations?
  - The fastest computation among the accepting ones, … if there is any !
- What is the practical meaning of nondeterministic computations as long as real computations are deterministic?
- To answer, let us consider in general nondeterministic computations as a model for parallelism: “blind” search among various ways, …
- The great practical impact of this theme derives from the very fact that ….

- … that many problems of great practical impact have a simple, natural and "efficient" solution in nondeterministic terms:
  - Hamiltonian path on a graph (a path that touches all nodes exactly once)
  - Satisfiability of logical propositional formulas (→requirements verification of finite state systems)
  - ….
- What is common to the solution of all such problems is the ***difficulty of finding*** the solution, as opposed to the ***ease of checking*** that a possible (candidate) solution is in fact correct:
  - If a "little devil" would tell me "try this: see if this solution is correct", then checking if the advice is right or not is not difficult ---->
  - Typical problems of this kind are solved exhaustively (i.e., by "trying all possibilities") in a nondeterministic fashion: choose (ND) a possible solution (in polynomial, often linear complexity), then check if it is so (also polynomial complexity).
  - Obviously when one goes to the deterministic version, trying all ways is very onerous (combinatorial effects ⇒ exponential complexity or worse)
- Based on this approach one can derive a substantial class of problems (including the above examples and thousands of other ones):

- $\mathcal{NP}$: the class of problems that can be solved ***non***deterministically in polynomial time
- $\mathcal{P}$: the class of problems that can be solved deterministically in polynomial time (the ***tractable*** problems)

- A momentous question : $\mathcal{P} = \mathcal{NP}$?
- ***Most likely*** not. However, surprisingly … this crucial question has not been answered!
- If $\mathcal{P} = \mathcal{NP}$ we might solve efficiently an enormous quantity of problems that now are intractable, and must be addressed by means of heuristics, by special cases, etc.
- The notion of ($\mathcal{NP}$) ***completeness***: a "representative" of a class incorporates the essence of all problems of the class: if we find the solution for it we have it for all!
  - $\mathcal{NP}$ completeness is based on the existence of ***polynomial-time reductions***: a problem is reduced to another one by means of a polynomial-time transformation (total computable function)
- It is interesting to notice that in the enormous set of $\mathcal{NP}$ problems, a great number is also $\mathcal{NP}$-complete: it would suffice to solve one of then in (deterministic) polynomial time and we would have $\mathcal{P} = \mathcal{NP}$; it would suffice to prove that one of them is intractable and all other ones would also be so!

- Finally: nondeterministic is not a synonym for random, but … random computations are very effective and promising.