Computability, Decidability, and Complexity (I)

Prof.Dr. Ferucio Laurenţiu Ţiplea

Department of Computer Science
"Al.I.Cuza" University of Iaşi
Iaşi, Romania
E-mail: fltiplea@mail.dntis.ro

Outline

Introduction to Computability

- Questions and problems
- Computation (What-) problems and decision problems
- Problems and algorithms
- Hilbert's problems
- Hilbert's program
- Algorithms and functions
- The theory of computation

Questions and Problems

Questions:

- For what real number x is $2x^2 + 36x + 25 = 0$?
- What is the smallest prime number greater than 20?
- Is 257 a prime number?

A problem is a class of questions. Each question in the class is an instance of the problem. Examples of problems:

- Find the real roots of the equation $ax^2 + bx + c = 0$, where a, b, and c are real numbers. If we substitute a, b, and c by real values, we get an instance of this problem;
- Does the equation $ax^2 + bx + c = 0$, where a, b, and c are real numbers, have positive (real) roots? If we substitute a, b, and c by real values, we get an instance of this problem.

Computation and Decision Problems

Most problems in mathematical sciences are of two kinds:

- computation (what-) problems - such a problem is to obtain the value of a function for a given argument.

  Example: What is the square root of a given positive integer x to nearest thousandth?

- decision problems (yes/no problems) - these are problems whose answer is yes or no.

  Example: Does a given vector satisfy a given set of linear equations?

Decision problems could be considered computation problems but it is worthwhile to make the distinction.

Problems and Algorithms

An algorithm for a problem is an organized set of commands for answering on demand any question that is an instance of the problem.

A problem is solvable if there exists an algorithm for it. Otherwise it is called unsolvable.

A solvable decision problem is also called decidable; an unsolvable decision problem is also called undecidable.

An algorithm for a decision problem is usually called a decision algorithm (procedure).

Hilbert's Problems

Hilbert's problems:

1. list of 23 (originally) unsolved problems in mathematics
2. 10 (1, 2, 6, 7, 8, 13, 16, 19, 21, and 22 – Derbyshire 2004, p. 377) were presented at the Second International Congress in Paris on August 8, 1900
3. the final list of 23 problems omitted one additional problem on proof theory (Thiele 2001)
4. they were designed to serve as examples for the kinds of problems whose solutions would lead to the furthering of disciplines in mathematics (some of them were areas for investigation and therefore not strictly "problems")

Hilbert's Problems (cont'd)

Hilbert's tenth problem:

> Find a decision procedure to tell whether a Diophantine equation has a solution

Diophantine equation:

$$P(x_1, \ldots, x_n) = 0,$$

where P is a polynomial in the integer unknowns $x_1, \ldots, x_n$ with integer coefficients.

Answer:

- At the time Hilbert posed this problem, mathematicians did not consider the possibility of proving it unsolvable
- When undecidability results began to appear in the 1930s and early 1940s, many researchers began to suspect that the problem was unsolvable
- 1972, Y. Matiyacevic: Hilbert's tenth problem is undecidable

Hilbert's Program

Hilbert's Program (early 1920s):

> Formalize all of mathematics in axiomatic form, together with a proof that this axiomatization of mathematics is consistent.
> The consistency proof itself was to be carried out using only what Hilbert called "finitary" methods.

In September 1930, Kurt Gödel announced his first incompleteness theorem at a conference in Königsberg, showing that Hilbert's program is not realizable.

Algoritms and Functions

Every algorithm computes a function:

$$\text{input } (x) \rightarrow \text{algorithm} \rightarrow \text{output } (y)$$

$$f(x) = y$$

A function computable by some algorithm is called a computable function.

The theory of computation is concerned with the following main question:

> What functions are computable? Equivalently, What problems are solvable?

The Theory of Computation

- The theory of computation originated in the 1930s, in the work of the logicians Church, Gödel, Kleene, Post, and Turing
- The theory of computation provides models of computation together with their basic properties
- The concept of computation is related to that of an algorithm
- Computation: numerical or non-numerical

Turing Machines

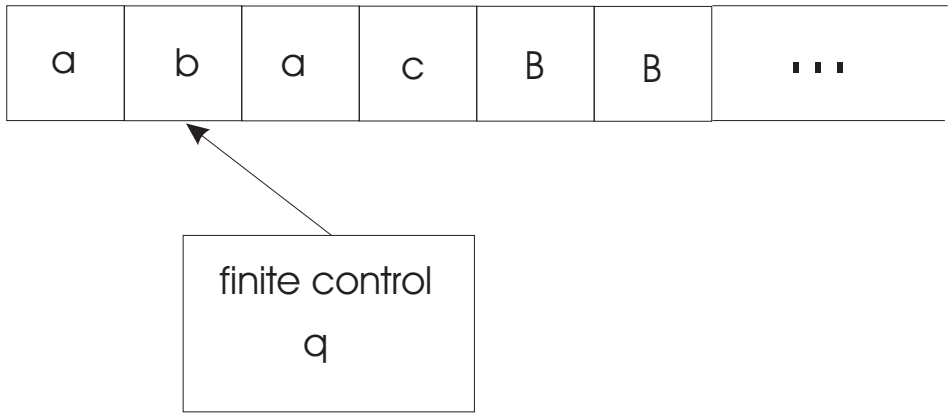
A formal model for an effective procedure should possess certain properties:

- it should be finitely describable;
- the procedure should consist of discrete steps, each of which can be carried out mechanically.

In 1936, Alan Turing proposed such a model. His model, called today Turing machine, has become the accepted formalization of an effective procedure.

Turing Machines

The basic model has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time.



The tape has a leftmost cell but it is infinite to the right. Each cell may hold exactly one of a finite number of tape symbols. Initially, the n leftmost cells, for some $n \geq 0$, hold the input; the remaining infinity of cells each hold the blank, which is a special tape symbol that is not an input symbol. The finite control can be in one of a finite number of internal states.

Turing Machines

In one move, the machine, depending upon the symbol scanned by the tape head and the state of the finite control, does the followings:

- changes state,
- replaces the current symbol on the tape cell by a new symbol, and
- moves its tape head one cell to the left or to the right.

Turing Machines

Definition 1

A deterministic Turing machine, abbreviated DTM, is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite non-empty set of states;
- Σ is a finite non-empty set of input symbols, called the input alphabet of M;
- Γ is a finite non-empty set of tape symbols, called the tape alphabet of M. It is assumed that $\Sigma \subseteq \Gamma$;
- $\delta : Q \times \Gamma \rightsquigarrow Q \times \Gamma \times \{L, R\}$ is the transition relation/function of M;
- $q_0 \in Q$ is the initial/start state;
- $B \in \Gamma - \Sigma$ is the blank symbol;
- $F \subseteq Q$ is the set of final states.

Turing Machines: Configurations

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a DTM.

- A configuration (instantaneous description) of M is any pair $(q, u|av) \in Q \times \Gamma^*\{|\}\Gamma^+$
- A configuration $(q, u|av)$ for which $q = q_0$, $u = \lambda$, and $v = \lambda$ whenever $a = B$, is called an initial configuration
- $Config(M)$ stands for the set of all configurations of M
- $C_0(w)$ stands for $(q_0, |w)$, if $w \neq \lambda$, and $(q_0, |B)$, otherwise

Example 2

- $(q, aaBa|aBBBaab)$
- $(q, |BaBBab)$

Turing Machines: Transition Relation

Let M be a DTM. Define the transition relation of M,

$$\vdash_M \subseteq Config(M) \times Config(M),$$

as follows:

$$(q, u|av) \vdash_M (q', u'|a'v')$$

iff one of the following two properties holds:

- (move/step to the right)
  $\delta(q,a) = (q', \overline{a}, R)$, $u' = u\overline{a}$, and

  $$\texttt{if } v = \lambda \texttt{ then } a' = B \textit{ and } v' = \lambda \texttt{ else } v = a'v';$$

- (move/step to the left)
  $\delta(q,a) = (q', \overline{a}, L)$, $u \neq \lambda$, $u = u'a'$, $v' = \overline{a}v$.

Turing Machines: Computation

Let M be a DTM.

- (computation step/move/transition)

$$(q, u|av) \vdash_M (q', u'|a'v')$$

- (computation)

$$(q_0, u_0|a_0v_0) \vdash_M \cdots \vdash_M (q_n, u_n|a_nv_n),$$

  where $(q_0, u_0|a_0v_0)$ is an initial configuration.

- a configuration C is called a blocking configuration if there is no configuration C' such that $C \vdash_M C'$;

- a configuration C' is called reachable from C if $C \vdash_M^* C'$;

Turing Machines

Remark 1

Let M be a DTM. Then:

1. *In any configuration, the machine can perform at most one move (because the machine is deterministic);*
2. *Blocking configurations are not necessarily final configurations, and vice-versa;*
3. *A configuration $(q,u|av)$ is a blocking configuration if either $\delta(q,a)\uparrow$ or $u=\lambda$ and $\delta(q,a)=(q',\overline{a},L)$.*

Turing Machines: Accept, Reject, Loop

Let M be a DTM and w and input for M.

- M halts on w if there exists a blocking or final configuration C such that $(q_0, |w) \overset{*}{\vdash} C$. Two cases are to be considered:
  - M accepts w if C is a final configuration. Denote by $accept(M)$ the set of all w accepted by M (also called the language accepted by M, denoted $L(M)$);
  - M rejects w if C is a blocking configuration. Denote by $reject(M)$ the set of all w rejected by M;
- M loops on w if it does not halt on w.

$$accept(M) \cup reject(M) \cup loop(M) = \Sigma^*$$

Turing Machines

Example 3

The following Turing machine accepts $L = \{0^n 1^n | n \geq 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
q_4					

q_4 *is the only final state.*

Turing Machines: Computing Functions

Turing machines may be viewed as computers of arithmetic functions. The traditional approach is to represent integers in unary. The integer $x \geq 0$ is represented by 0^x.

$$(x_1, \ldots, x_k) \rightsquigarrow 0^{x_1} 1 \cdots 10^{x_k}$$

Example 4

- $(1,2,1) \rightsquigarrow 010010$
- $(2,0,0,3) \rightsquigarrow 00111000$

Remark 2

- *Three tape symbols are needed:* 0, 1, *and* B;
- 0 *and* 1 *may be replaced by any two distinct symbols* a *and* b.

Turing Machines: Computing Functions

Let M be a DTM and $f : \mathbf{N}^k \to \mathbf{N}$. M computes f if, for any $x_1, \ldots, x_k \in \mathbf{N}$, one of the following two properties holds true:

- if $f(x_1, \ldots, x_k)\downarrow$ then M halts on $(q_0, |0^{x_1}1 \cdots 10^{x_k})$ with a tape consisting of $0^{f(x_1,\ldots,x_k)}$;
- if $f(x_1, \ldots, x_k)\uparrow$ then M does not halt on $(q_0, |0^{x_1}1 \cdots 10^{x_k})$.

An arithmetic function is Turing-computable if there exists a Turing machine which computes it.

We denote by TCF the set of all Turing-computable functions.

Turing Machines: Computing Functions

Remark 3

- *All natural numbers are Turing-computable;*
- *If the unary notation for natural numbers is adopted as being*

$$x \rightsquigarrow 0^{x+1}$$

then Turing machines with only two tape symbols are sufficient to compute arithmetic functions (prove it!).

Turing Machines: Computing Functions

Example 5

The following Turing machine computes $x \dot{-} y$.

	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6			

The machine has no final state.

Storage in the finite control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other one storing a symbol.

Example 6

Design a Turing machine that looks at the first input symbol, records it in its finite control, accepts if the first symbol does not appear elsewhere on its input, and rejects, otherwise.

Shifting over

A Turing machine can make space on its tape by shifting blocks of symbols a finite number k of cells to the right. To do so,

- the machine uses its finite control to store k symbols;
- initially, the state is $(q, B, \ldots, B)$;
- if the current state is $(q, X_1, \ldots, X_k)$ and the symbol scanned on the tape is X, the machine prints X_1, and enters the state $(q, X_2, \ldots, X_k, X)$. This process is repeated until the block in question is exhausted.

Remark 4

- *If space is available, it can push blocks of symbols to the left;*
- *If k is large, the number of states could became very large. A different solution will be proposed by the "checking off symbols" technique;*
- *by this technique, blocks of symbols can be surrounded by two special symbols, \$ and #, for instance.*

Multiple tracks

We can view the tape of the Turing machine as being divided into k tracks, for any $k \geq 2$. To do so, the tape symbols are considered k-tuples, one component for each track.

Example 7

Design a Turing machine that takes a binary input greater than 2 and determines whether it is a prime.

Checking off symbols

Turing machines can check off symbols on their tapes. To do so, the tape is divided into 2 tracks, one holding the input and the other one being used to check symbols.

Example 8

Design a Turing machine that accepts the language

$$L = \{wcw | w \in \{a,b\}^*\},$$

where $c \notin \{a,b\}$.

Checking off symbols

Using this technique, a Turing machine can shift blocks of symbols a finite number k of cells to the right, as follows:

- check (if it is necessarily) the first symbol of the block;
- repeatedly do
  - store and erase the rightmost symbol of the block;
  - move right k cells and print the symbol;

  until the checked symbol is encounter (and shift it too).

If k is large, this solution might be more convenient than the previous one.

Subroutines

As with programs, Turing machine can be designed in a “to-down” manner, by using “subroutines”. The general idea is to write part of a Turing machine program to serve as a subroutine, with an initial state and several return states. The call of this subroutine is effected by entering in its initial state, and the return is effected by the move from a return state.

Example 9

Design a Turing machine which computes the multiplication function.

Universal Turing Machines

Recall that, to compute functions, Turing machines can be considered in the form $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$, where:

- $Q = \{q_1, \ldots, q_n\}$, $n \geq 1$;
- $\Sigma = \{0, 1\}$;
- $\Gamma = \{0, 1, B\}$;
- $F = \emptyset$.

Moreover,

- denote $0 = X_1$, $1 = X_2$, $B = X_3$, and
- denote $L = D_1$, $R = D_2$, and
- assume that for any state q there exists at least one transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ such that $q = q_i$ or $q = q_k$.

Universal Turing Machines

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ a Turing machine as given above. We encode M as follows:

- encode each transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ by

$$0^i 1 0^j 1 0^k 1 0^l 1 0^m$$

- assume that M has r transitions and their codes are $code_1, \ldots, code_r$;
- encode M by:

$$111 code_1 11 code_2 11 \cdots 11 code_{r-1} 11 code_r 111$$

Universal Turing Machines

Example 10

Let $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \emptyset)$, *where:*

- $\delta(q_1, 1) = (q_3, 0, R)$,
- $\delta(q_3, 0) = (q_1, 1, R)$,
- $\delta(q_3, 1) = (q_2, 0, R)$, *and*
- $\delta(q_3, B) = (q_3, 1, L)$.

The following binary sequence is a code of this machine:

$$111010010001010011000101010010011$$

$$000100100101001100010001000100101111$$

Universal Turing Machines

Remark 5

1. *Except for the totally undefined Turing machine or Turing machines with exactly one transition, all the other Turing machines have more than one code, but finitely many.*

2. *If we order lexicographically the transitions of Turing machines, then each Turing machine has exactly one code. The lexicographic order is given by:*

   - *order first by states,* $q_1 < q_2 < \cdots < q_n$,
   - *order then by input symbols,* $0 < 1 < B$.

   The unique code such obtained is denoted by $\langle M \rangle$.

3. *Distinct Turing machines have distinct codes.*

Universal Turing Machines

- The code of a Turing machine is a binary sequence. If we order lexicographically these binary sequences, we get the standard enumeration of Turing machines

$$M_0, M_1, \ldots$$

- The standard enumeration of Turing machines induces a standard enumeration of all n-ary recursive functions:

$$f_0^{(n)}, f_1^{(n)}, \ldots$$

  where $f_i^{(n)}$ is the n-ary function computed by M_i;

- As there are infinitely many Turing machines computing the same n-ary function, each function $f_i^{(n)}$ appears infinitely many times in the standard enumeration of all n-ary recursive functions.

Universal Turing Machines

Given M_i and $x_1, \ldots, x_n \in \mathbf{N}$, we denote by $\langle M, x_1, \ldots, x_n \rangle$ the binary string

$$\langle M_i \rangle 0^{x_1} 1 \cdots 10^{x_n}$$

Theorem 11

There exists a Turing machine M_u that, on input $\langle M_i, x_1, \ldots, x_n \rangle$, computes $f_i^{(n)}(x_1, \ldots, x_n)$, for all i and $x_1, \ldots, x_n$.

Proof For the sake of simplicity we will only consider the case $n = 1$.
We shall exhibit a 3-track Turing machine M_u:

- the first track holds the input $\langle M_i, x \rangle$;

Universal Turing Machines

- the second track is used to simulate the tape of M_i;
- the third track holds the current state of M_i, with q_i represented by 0^i. Initially, it holds 0^1 (i.e., q_1).

The behavior of M_u is as follows:

- mark the first cell so the head can find its way back;
- M_u checks the input to see that $\langle M_i \rangle$ is the code of some Turing machine;
- copy 0^x on the second track and initialize track 3 to hold 0 (q_1);

Universal Turing Machines

- let 0^i (q_i) be the current content of track 3 and X_j be the current symbol scanned on tracked 2.
  - M_u scans the first track (from left to right) looking for a substring which begins with 110^i10^j1;
  - If no such string is found, M_u halts (in a blocking configuration);
  - If such a code is found, let it be $0^i10^j10^k10^l10^m$. Then, it puts 0^k on track 3, replaces X_j by X_l on the second track, and moves in the direction pointed by D_m;
- When M halts, M_u halts.

□

Universal Turing Machines

A Turing machine as the one in the proof above is called an universal Turing machine.

Corollary 12

There exist n-ary Universal Turing Machines, for any $n \geq 2$.

Proof.

Modify an arbitrary universal Turing machine M_u as follows:

- start off with $0^i 10^{x_1} 1 \ldots 10^{x_n}$ on the tape;
- generate the code of M_i (use a new track if needed);
- call M_u on $\langle M_i, x_1, \ldots, x_n \rangle$.

The machine such obtained computes an universal function $u^{(n+1)}$, for all $n \geq 1$. □

Recursively Enumerable Sets

A set $A \subseteq \mathbf{N}$ is called recursively enumerable, abbreviated r.e., if $A = \emptyset$ or A is the range of an unary total recursive function.

Remark 6

If $A \subseteq \mathbf{N}$ is a non-empty r.e. set and f is an unary total recursive function such that $A = cod(f)$, then we will say that f enumerates A. Why?

- $0 \rightsquigarrow f(0) \in A$;
- $1 \rightsquigarrow f(1) \in A$;
- $\ldots$
- $A = \{f(x) | x \geq 0\}$.

Recursively Enumerable Sets

Remark 7

$A \subseteq \mathbf{N}$ is an r.e. set if $A = \emptyset$ or there exists a Turing machine M such that:

- *M halts on all inputs $x \in \mathbf{N}$;*
- *for each input x, when M halts on x then the output $M(x)$ is in A;*
- *for any $a \in A$ there exists $x \in \mathbf{N}$ such that $a = M(x)$*

(the machine M enumerates A).

We may say:

A is an r.e. set if $A = \emptyset$ or there exists an algorithm $\mathscr{A}$ that enumerates all the elements of A.

Recursively Enumerable Sets

Notation:

- RS = the set of all recursive sets $A \subseteq \mathbf{N}$
- RES = the set of all r.e. sets $A \subseteq \mathbf{N}$

Proposition 1

$$RS \subseteq RES$$

Recursively Enumerable Sets

Theorem 13

1. *A set is r.e. iff it is the range of an unary recursive function.*
2. *A set is r.e. iff it is the domain of an unary recursive function.*

Remark 8

The above theorem can be used to show that certain sets are/are not r.e.:

- *use (1) to show that a given set is r.e.;*
- *use (2) to show that a given set is not r.e.*

Recursively Enumerable Sets

Example 14

1. The set $A = \{x | f_x(x)\downarrow\}$ is r.e.
2. The set $\mathbf{N} - A$, where A is the set from (1.), is not r.e.
3. The set $B = \{x | f_x \text{ is total}\}$ is not r.e.
4. The set $\mathbf{N} - B$, where B is the set from (3.), is not r.e.

Proposition 2

The set RES is closed under union and intersection but not under complement.

From this proposition follows that $RS \subset RES$

Recursively Enumerable Sets

Theorem 15

A set A is recursive iff A and $\mathbf{N} - A$ are r.e.

Proof.

$\Rightarrow$: straightforward.

$\Leftarrow$: If $A = \emptyset$ or $\mathbf{N} - A = \emptyset$, then A is recursive.

Assume that both A and $\mathbf{N} - A$ are non-empty. Let g_1 and g_2 be two total recursive functions such that $A = cod(g_1)$ and $\mathbf{N} - A = cod(g_2)$.

Let $h(x) = \mu(y)[g_1(y) = x \vee g_2(y) = x]$. h is recursive and

$$\chi_A(x) = \begin{cases} 1, & g_1(h(x)) = x \\ 0, & \text{otherwise} \end{cases}$$

Therefore, A is reursive. □

Recursively Enumerable Sets

Remark 9

Given a subset $A \subseteq \mathbf{N}$, define the language of A as being the set $L_A = \{0^x | x \in A\} \subseteq \{0,1\}^*$. Then:

$$
\begin{aligned}
A \subseteq \mathbf{N} \text{ is recursive} &\Leftrightarrow \chi_A \text{ is total recursive} \\
&\Leftrightarrow (\exists \text{ a Turing machine } M) \\
&\quad (M \text{ computes } \chi_A) \\
&\Leftrightarrow (\exists \text{ a Turing machine } M') \\
&\quad (L_A = accept(M') \wedge \overline{L_A} = reject(M'))
\end{aligned}
$$

The second equivalence is based on:

- *Given M which computes χ_A, design M' which accepts when M halts in a configuration $(q, |0)$, and rejects when M halts in a configuration $(q, |B)$.*

Recursively Enumerable Sets

Remark 10

$$A \subseteq \mathbf{N} \text{ is r.e.} \Leftrightarrow A = dom(f) \text{ for some recursive function}$$
$$\Leftrightarrow (\exists \text{ a Turing machine } M)$$
$$(L_A = \{0^x | M \text{ halts on } 0^x\})$$
$$\Leftrightarrow (\exists \text{ a Turing machine } M')$$
$$(L_A = accept(M'))$$

The second equivalence is based on:

- *Given M which computes f, design M' which accepts 0^x if and only if M halts on 0^x.*

Recursively Enumerable Sets

Extension to languages

Definition 16

Let Σ be an alphabet and $L \subseteq \Sigma^*$. L is recursive if there exists a Turing machine M such that $L = accept(M)$ and $\overline{L} = reject(M)$.
We say that L is recursively enumerable, abbreviated r.e., if there exists a Turing machine M such that $L = accept(M)$.

Denote:

- $RS(\Sigma)$ = the set of all recursive sets $L \subseteq \Sigma^*$;
- $RES(\Sigma)$ = the set of all r.e. sets $L \subseteq \Sigma^*$;
- $\mathscr{L}_{rec}$ = the class of all recursive languages;
- $\mathscr{L}_{re}$ = the class of all r.e. languages.

Recursively Enumerable Sets

Remark 11

The main difference between recursive and r.e. sets:

- *If* L *is recursive* $\Rightarrow (\exists M)(L = acccept(M) \wedge \overline{L} = reject(M))$
  - *given* $w \in \Sigma^*$, *if* $w \in L$ *then* M *accepts* w *and, therefore, we know that* w *is a member of* L;
  - *given* $w \in \Sigma^*$, *if* $w \notin L$ *then* M *rejects* w *and, therefore, we know that* w *is not a member of* L;
- *If* L *is r.e.* $\Rightarrow (\exists M)(L = acccept(M))$
  - *given* $w \in \Sigma^*$, *if* $w \in L$ *then* M *accepts* w *and, therefore, we know that* w *is a member of* L;
  - *given* $w \in \Sigma^*$, *if* $w \notin L$ *then* M *either rejects* w *or loops. In this case we might not know that* w *is not a member of* L.

Recursively Enumerable Sets

Theorem 17

1. $RS(\Sigma)$ *is closed under union, intersection, and complement;*
2. $RES(\Sigma)$ *is closed under union and intersection, but not under complement;*
3. $RS(\Sigma) \subseteq RES(\Sigma)$*;*
4. $L \subseteq \Sigma^*$ *is recursive iff* L *and* $\Sigma^* - L$ *are r.e.*
5. $\mathscr{L}_{rec} \subset \mathscr{L}_{re} = \mathscr{L}_0$ *(*$\mathscr{L}_0$ *is the class of type 0 Chomsky languages).*