UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Dipartimento di Ingegneria

Capitolo 1
Linguaggi e Macchine Astratte

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Perché parlare di macchine?

- I programmi scritti in un linguaggio dovranno essere eseguiti su una macchina
- Quali caratteristiche deve avere questa macchina?
- quali sono le caratteristiche base che deve avere
- ha senso considerare più macchine diverse?
- le diverse macchine hanno diversa potenza espressiva?
- Per capire quali problemi si possono risolvere con una certa macchina, dovremo scrivere dei teoremi
  - abbiamo quindi bisogno di avere un modello semplice delle varie macchine: saranno *macchine astratte*, in cui astraiamo da tutti i dettagli possibili

ALCUNE DOMANDE FONDAMENTALI

- **Quali istruzioni** esegue un elaboratore?
- Un elaboratore può risolvere **qualunque problema** ci venga in mente?
  - **in particolare:** ***esistono problemi che un elaboratore NON PUÒ risolvere?***
  - **se sì, vorremmo saperlo:** non avrebbe senso insistere ad andare in una direzione impossibile
- Quanto conta il linguaggio di programmazione in tutto questo?

ALGORITMI e PROGRAMMI

- **ALGORITMO:** sequenza *finita* di mosse che risolve *in un tempo finito* una classe di problemi.
- **CODIFICA**: descrizione dell'algoritmo tramite un *insieme ordinato* di *frasi (istruzioni)* di un *linguaggio di programmazione,* che specificano le *azioni* da svolgere.
- **PROGRAMMA:** testo scritto in accordo alla *sintassi* e alla *semantica* di un linguaggio di programmazione.

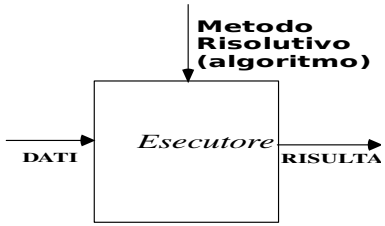
ATTENZIONE **Un programma può *non essere* un algoritmo!**

- Basta che il programma *non termini* (sequenza di mosse infinita)
- Un programma che non sia un algoritmo può comunque essere *molto utile* (sistemi operativi, controllo di semafori, etc…) – in tutti quei casi in cui l'obiettivo sia *agire* più che *calcolare un risultato.*

ESECUZIONE

- L'algoritmo esprime la soluzione a un problema.
- Un programma è la formulazione testuale di un algoritmo in un dato linguaggio di programmazione
- **L'esecuzione delle azioni** specificate dall'algoritmo, ***nell'ordine da esso specificato,*** porta a ottenere, a partire dai dati di ingresso, la soluzione del problema.

Ciò presuppone l'esistenza di un **AUTOMA ESECUTORE,** ossia di una ***macchina astratta*** capace di ***eseguire*** le azioni dell'algoritmo



L'AUTOMA ESECUTORE

Che caratteristiche deve avere tale automa?

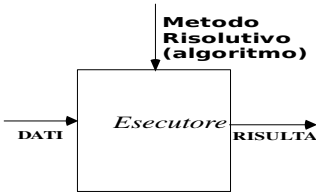
- Deve **ricevere dall'esterno una descrizione** dell'algoritmo

Ma poiché una descrizione è una *frase di un linguaggio..*

- Dev'essere capace di ***interpretare un linguaggio***
  - Lo chiameremo *linguaggio macchina*

Vincoli di realizzabilità sica

- se l'automa è fatto di parti, esse sono in *numero nito*
- ingresso e uscita devono essere denotabili attraverso un *insieme nito* di *simboli*
  - si prescinde da aspetti sici o tecnologici
  - si astrae da speci ci casi concreti
  - ci si concentra sulle proprietà essenziali



SISTEMI FORMALI

Formalizzando cosa sia l'*automa esecutore* si arriva a de nire il concetto stesso di *computabilità*

- Approccio a ***gerarchia di macchine astratte***
  - dagli *Automi a Stati Finiti* alla *Macchina di Turing*
  - "macchina" = macchina astratta con una data *struttura* e un dato insieme di *mosse elementari*
- Approccio funzionale (Hilbert, Church, Kleene)
  - fondato sul concetto di funzione matematica
  - mira a caratterizzare il concetto di *funzione computabile*
- Sistemi di riscrittura (Thue, Post, Markov)
  - descrivono l'automa come insieme di *regole di riscrittura (o di inferenza)* che trasformano frasi in altre frasi

GERARCHIA DI MACCHINE ASTRATTE

- macchina combinatoria
- automi a stati niti
- *... [altre macchine interessanti]*
- **macchina di Turing**

Perché de nire una gerarchia ?

- Macchine diverse potrebbero avere (e in e etti hanno..) ***diversa capacità di risolvere problemi***
  e noi vogliamo scegliere quella più adatta
- Se neanche la macchina più "potente" riesce a risolvere un dato problema, esso potrebbe essere ***non risolubile***
  e se è così, vogliamo saperlo!

LA MACCHINA BASE

La *macchina base* è formalmente de nita dalla tripla:

$\langle I, O, mfn \rangle$

dove

- I = insieme nito dei **simboli di ingresso**
- O = insieme nito dei **simboli di uscita**
- $mfn: I \rightarrow O$ *(funzione di macchina)*

Esempio: le *porte logiche* e le *funzioni* in genere (full adder)
Per una funzione booleana a due ingressi e una uscita,
$I = \{\{0,1\} \times \{0,1\}\}$ e $O = \{0,1\}$

Esercizio

- Definire una macchina combinatoria che prende in input 2 lettere (che possono essere `'a'`, `'b'` o `'c'`) e che fornisce le 2 lettere in ordine alfabetico

$\langle I, O, mfn \rangle$

I = {aa, ab, ac, ba, bb, bc, ca, cb, cc}

O = {aa, ab, ac, bb, bc, cc}

mfn

input	output
aa	aa
ab	ab
ac	ac
ba	ab
bb	bb
bc	bc
ca	ac
cb	bc
cc	cc

LIMITI

- Risolvere problemi con questa macchina comporta di *enumerare una ad una tutte le possibili configurazioni d'ingresso,* indicando il valore di uscita corrispondente.
- Limite computazionale: è un dispositivo ***puramente combinatorio*** e come tale *inadatto a problemi che richiedano di "ricordare" qualcosa del passato*
  - Esempio: riconoscimento di sequenze o stringhe
  - Esempio: somme di numeri forniti in successione
  - …

  in quanto è privo di qualunque memoria interna.

L'AUTOMA A STATI FINITI (ASF)

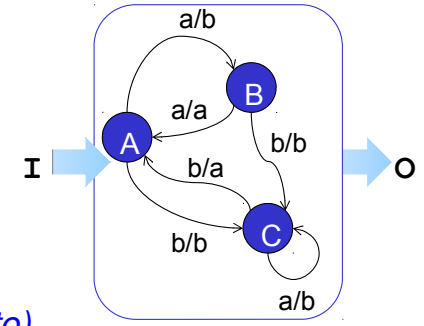
Il modo più semplice per introdurre un'idea di "memoria" è de nire un automa con un numero *nito* di *stati interni*.

Un *automa a stati niti* è de nito dalla quintupla:

$\langle I, O, S, mfn, sfn \rangle$

dove

- I = insieme nito dei **simboli di ingresso**
- O = insieme nito dei **simboli di uscita**
- S = insieme nito degli **stati**
- $mfn: I \times S \rightarrow O$ *(funzione di macchina)*
- $sfn: I \times S \rightarrow S$ *(funzione di stato)*



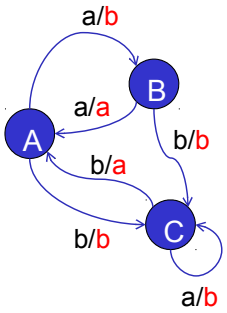
Esempio ASF

- **I** = {a,b} insieme finito dei simboli di ingresso
- **O** = {a,b} insieme finito dei simboli di uscita
- **S** = {A,B,C} insieme finito degli stati
- **mfn:** $I \times S \rightarrow O$ *(funzione di macchina)*

	A	B	C
a	b	a	b
b	b	b	a

- **sfn:** $I \times S \rightarrow S$ *(funzione di stato)*

	A	B	C
a	B	A	C
b	C	C	A



LIMITI

- Poiché lo stato funge da memoria interna, le risposte possono essere diverse *anche a parità di dati d'ingresso.*
- Esistono varie sotto-categorie e rappresentazioni:
  - automi di Mealy, automi di Moore
  - con / senza riferimento temporale esterno: sincroni, asincroni
  - automi equivalenti, automa con numero minimo di stati
  - tabella delle transizioni (specifica mfn e sfn insieme)
- Limite computazionale: è un dispositivo ***a memoria finita*** e come tale *inadatto a problemi che non consentano di limitare a priori la lunghezza delle sequenze da ricordare*
  - Esempio: bilanciamento delle parentesi

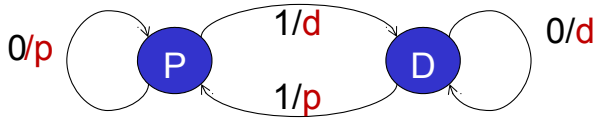
Esercizio

Definire un automa a stati finiti che riceve una sequenza di simboli '1' e '0' e che fornisce in output

- il simbolo 'p' se il numero di 1 e` pari
- il simbolo 'd' se il numero di 1 e` dispari

⟨I, O, S, mfn, sfn⟩

I = {0, 1}
O = {p, d}
S = {P, D}



mfn

	P	D
0	p	d
1	d	p

sfn

	P	D
0	P	D
1	D	P

Esercizio

Si consideri una sequenza di simboli '(' e ')'. Si vuole fornire in output 'ok' se le parentesi sono bilanciate correttamente e 'no' se non sono bilanciate correttamente.

Si definisca un automa a stati finiti che verifica il bilanciamento delle parentesi, supponendo che si arrivi fino ad un livello di annidamento pari a 4.

LA MACCHINA DI TURING

Per superare il limite della memoria nita, **si introduce un "*nastro*" come *supporto di memorizzazione esterno***.

La *macchina di Turing* è quindi de nita dalla quintupla:

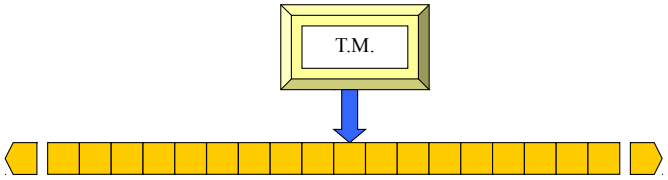
$$\langle A,\ S,\ mfn,\ sfn,\ dfn \rangle$$

dove

- A = insieme nito dei **simboli di ingresso e uscita**
- S = insieme nito degli **stati** *(uno dei quali è **HALT**)*
- $mfn: A \times S \to A$ *(funzione di macchina)*
- $sfn: A \times S \to S$ *(funzione di stato)*
- $dfn: A \times S \to D = \{Left, Right, None\}$ *(funz. di direzione)*

LA MACCHINA DI TURING

Un *nastro* illimitatamente espandibile rappresenta il deposito dei dati (*memoria*)



La macchina è munita di una testina di lettura / scrittura e può:

- *leggere un simbolo* dal nastro
- *scrivere sul nastro il simbolo* speci cato da `mfn()`
- *transitare in un nuovo stato interno* speci cato da `sfn()`
- *spostarsi sul nastro* di una posizione nella direzione indicata da `dfn()`

IPOTESI: quando raggiunge lo stato HALT, la macchina si ferma.

ESEMPIO: RICONOSCIMENTO DI PALINDROMI

Un esempio: riconoscere le *frasi palindrome*

- Si definisce «palindroma» una frase che si possa leggere nei due sensi (da sinistra a destra, da destra a sinistra)

Esempi facili:

"Anna", "Aerea", "Ai lati d'Italia", "Alla bisogna tango si balla", "E le tazzine igienizzatele!",…

Esempi famosi: racconto (in italiano) 6538 caratteri

NB: per convenzione si prescinde da spazi, punteggiatura e accenti

ESEMPIO:
ANALISI DEL PROBLEMA

Per semplicità ci limitiamo a *palindromi di bit*

Esempio di frase palindroma:

♦ **1 0 1 1 0 1** ♦

Approccio

a) Definire un algoritmo appropriato

b) Programmarlo sulla Macchina di Turing

ESEMPIO:
UN POSSIBILE ALGORITMO (1/2)

Situazione iniziale:

♦ **1 0 1 1 0 1** ♦

1. leggere il primo simbolo a sinistra:

   ♦ **1** **0 1 1 0 1** ♦

2. ricordare se è 1 o 0 e marcare tale casella con ♦

   ♦ ♦ **0 1 1 0 1** ♦

3. ora, spostarsi sull'ultimo simbolo a destra:

   ♦ ♦ **0 1 1 0** **1** ♦

ESEMPIO:
UN POSSIBILE ALGORITMO (2/2)

4. se tale simbolo *non* coincide con quello ricordato, scrivere un simbolo d'errore (ad esempio, ☹) e terminare
   (ma non è il caso in esame)
   altrimenti, marcare tale casella con ♦

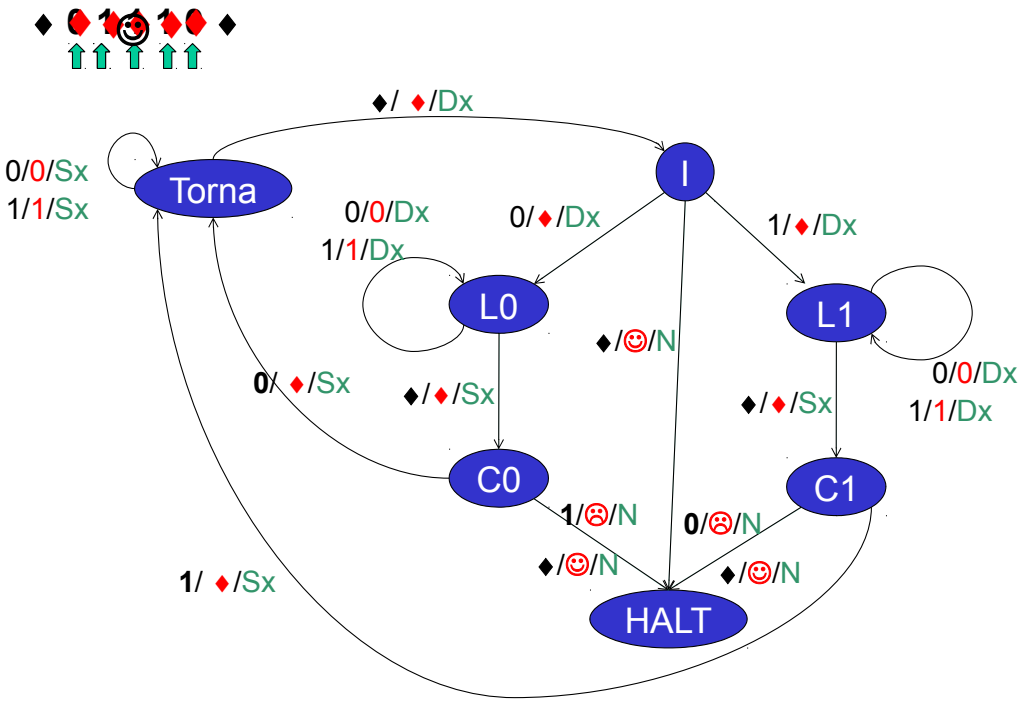
   ♦ ♦ **0 1 1 0** ♦ ♦

   tornare a capo e iterare sulla stringa rimasta:

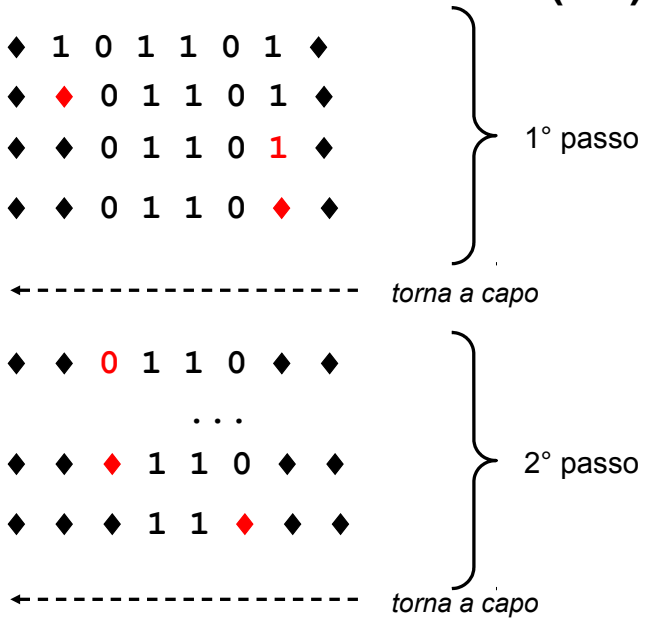
   ♦ ♦ **0 1 1 0** ♦ ♦

   [ottimizzazione possibile: operare anche all'indietro]

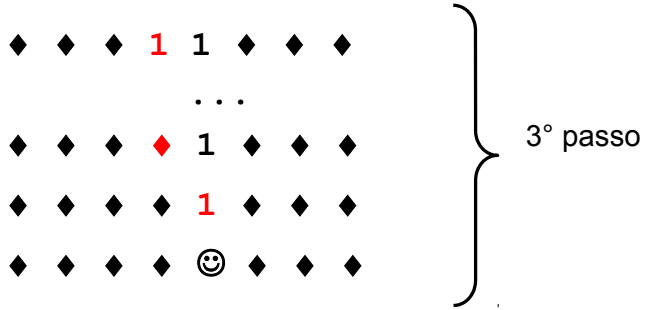
5. **Se si sono consumati tutti i bit (ci sono solo ♦), la frase è palindroma → scrivere un simbolo di esito positivo (ad esempio, ☺) e fermarsi.**



ESEMPIO: EVOLUZIONE (1/2)



ESEMPIO: EVOLUZIONE (2/2)



ESEMPIO: PROGRAMMARE LA MACCHINA

Programmare l'algoritmo sulla macchina di Turing significa **definire** le tre funzioni **mfn**, **sfn**, **dfn.**

Poiché hanno ***lo stesso dominio A x S***

- A = insieme **finito** dei simboli di ingresso e uscita
- S = insieme **finito** degli stati (di cui uno è *HALT*)

dove **A** e **S** sono insiemi ***finiti***, possono essere facilmente definite ***in forma tabellare***.

ESEMPIO: IL PROGRAMMA PER LA MDT

mfn

Stato iniziale

	s0	s1	s2	s3	s4	s5
0	0	♦	0	0	♦	☹
1	1	♦	1	1	☹	♦
♦	♦	☺	♦	♦	☺	☺

sfn

	s0	s1	s2	s3	s4	s5
0	s0	s2	s2	s3	s0	HALT
1	s0	s3	s2	s3	HALT	s0
♦	s1	HALT	s4	s5	HALT	HALT

dfn

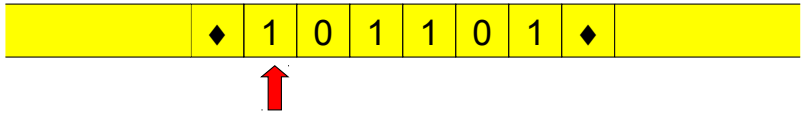
	s0	s1	s2	s3	s4	s5
0	L	R	R	R	L	N
1	L	R	R	R	L	L
♦	R	N	L	L	N	N

ESEMPIO: EVOLUZIONE

Nel nostro caso:

- A = { 0, 1, ♦, ☹, ☺ }
- S = { HALT, s0, *s1 (iniziale)*, s2, s3, s4, s5 }

Configurazione iniziale del nastro:



Configurazione finale del nastro:

LA MACCHINA DI TURING (3/4)

Risolvere un problema con questa macchina richiede di:

- **de nire** una opportuna ***rappresentazione dei dati di partenza*** da porre inizialmente sul nastro e dei ***dati di uscita*** che ci si aspetta di trovare alla ne sul nastro
- **de nire il *comportamento***, ossia le tre funzioni:

`mfn()` `sfn()` `dfn()`

concepite in modo da ***lasciare sul nastro la soluzione quando alla ne la macchina si ferma.***

LA MACCHINA DI TURING (4/4)

- La definizione dice che ***quando alla fine la macchina si ferma,*** **sul nastro c'è la soluzione del problema**
  - altrimenti, c'è stato un errore di programmazione
- **MA... è certo *che prima o poi si fermi?***
  - **in realtà, NO**: come vedremo, è possibile che il comportamento definito dalle tre funzioni causi un *ciclo infinito*
- Non fermarsi, entrando in loop, implica che la macchina potrebbe *non rispondere mai più*

MACCHINA DI TURING: DOMANDE FONDAMENTALI

Due domande:

1. **Esiste sempre una opportuna Macchina di Turing *capace di risolvere qualunque problema?***
   - oppure esistono problemi che nessuna MdT può risolvere?
2. **Esistono macchine "*più potenti*" della MdT?**
   - se esistono, dobbiamo trovarle!
   - se non esistono, dobbiamo sapere " n dove ci si può spingere" con ciò di cui disponiamo

TESI DI CHURCH-TURING

Rimandando per ora la risposta alla Domanda 1,
alla Domanda 2 risponde la

TESI DI CHURCH-TURING
Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella risolta dalla Macchina di Turing.

Dunque, la MdT è l'arma più potente del nostro arsenale.

Ma non sempre serve l'arma nale…

ALTRE MACCHINE…?

Non sempre è necessaria una Macchina di Turing

- **per alcune categorie di problemi *di grande interesse pratico* può bastare un *modello di memoria più limitato***
- il ***Push Down Automaton (PDA)*, ad esempio, prevede un "nastro" *espandibile da una parte sola*, col vincolo di poter accedere *solo alla cella "al top"…***

GERARCHIA DI MACCHINE
- macchine base (combinatorie)
- macchine (automi) a stati niti
- **macchina a stack (PDA)**
- macchina di Turing

MACCHINE SPECIFICHE vs MACCHINE UNIVERSALI

- Una volta de nita la parte di controllo, **la MdT è capace di risolvere uno speci co problema**
- ma così facendo, **essa è *speci ca di quel problema.***
  Siamo circondati da macchine speci che:calcolatrici, lavastoviglie, videoregistratori, videogiochi, orologi, …
- ***Conviene* fare macchine speci che?**
  - sì, per usi particolari e mercati di massa…
  - **no, se vogliamo una macchina di uso generale con cui risolvere ogni problema** (se ci riusciamo..)

Sarebbe possibile costruire una Macchina di Turing Universale?

VERSO MACCHINE UNIVERSALI

- Finora, l'algoritmo realizzato da una data Macchina di Turing era ***cablato nella macchina*** *(le tre funzioni)*
- **E se invece fosse *sul nastro*, e la macchina *se lo andasse a prendere?***

Si otterrebbe una
Macchina di Turing Universale (UTM)

- **Una macchina *il cui "programma" non cambia* quando cambia il problema da risolvere…**
- **.. perché l'algoritmo "cablato" si limita a leggere *dal nastro* una *descrizione dell'algoritmo che serve "un LOADER"***

MACCHINA DI TURING UNIVERSALE

Cosa richiede questo?

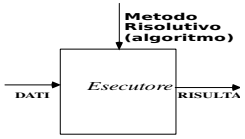
- **saper descrivere l'algoritmo richiesto.**

E per descrivere l'algoritmo?

- **occorre un linguaggio...**
- ***… e una macchina che lo interpreti.***

Conclusione:
la UTM è l'interprete di un linguaggio

È l'automa esecutore:



UTM e MACCHINE REALI

La Macchina di Turing Universale *modella il concetto di elaboratore di uso generale:*

- una macchina che *va a cercare le "istruzioni" da svolgere…* — **fetch**
- .. le interpreta... — **decode**
- .. e le esegue. — **execute**

Ha quindi senso confrontare la UTM con la *Macchina di Von Neumann,* che esprime architettura base di un elaboratore di uso generale.

MACCHINA DI TURING UNIVERSALE e MACCHINA DI VON NEUMANN (1/2)

In pratica:	corrisponde a:
♦ leggere / scrivere un simbolo dal / sul nastro	♦ lettura / scrittura dalla / sulla memoria RAM / ROM
♦ transitare in un nuovo stato interno	♦ nuova configurazione dei registri della CPU
♦ spostarsi sul nastro di una (o più) posizioni	♦ scelta della cella di memoria su cui operare (indirizzo contenuto nell'*Address Register*)

MACCHINA DI TURING UNIVERSALE e MACCHINA DI VON NEUMANN (2/2)

La UTM, però, NON cattura tutti gli aspetti dell'architettura di Von Neumann.

- Una UTM, infatti, elabora prendendo ***dal nastro*** dati e algoritmo e scrivendo ***sul nastro*** i risultati..
- .. ossia ***opera solo sulla memoria interna (RAM):***
  non ha il concetto di "*mondo esterno*",
  né, ovviamente, alcuna istruzione di I/O.

La UTM è *pura computazione:*
non modella la dimensione dell'*interazione,*
che invece esiste nella macchina di Von Neumann

La macchina di Von Neumann possiede istruzioni di I/O, la UTM no.

Introduzione alla Teoria della computabilità

PROBLEMI (IR)RISOLUBILI

- Secondo la Tesi di Church-Turing, non esiste un formalismo capace di risolvere una classe più ampia di problemi (ossia, più "potente") della MdT
- Perciò,

se neanche la macchina di Turing riesce a risolvere un problema, **quel problema è irrisolubile.**

- Cosa signi ca "non riesce a risolvere"?
  - una MdT correttamente programmata non può dare risposte scorrette
  - quindi, "non riuscire" signi ca che *non risponde proprio* e questo può signi care solo che ***non si ferma***

PROBLEMI RISOLUBILI

- Per capire, dobbiamo formalizzare meglio cosa intendiamo per ***problema (ir)risolubile***
- Dato che l'arma "più potente" di cui disponiamo per elaborare è la MdT, è ragionevole de nire:

PROBLEMA RISOLUBILE
un problema la cui soluzione può essere espressa da una MdT (o formalismo equivalente)

Però, la macchina di Turing computa *funzioni,* non "problemi": perché tutto torni, occorre quindi un modo (semplice) per associare a un problema una funzione.

FUNZIONE CARATTERISTICA

- Per instaurare questo ponte tra concetti, de niamo la ***funzione caratteristica*** **di un** ***problema***

Dato un **problema P** e detti
- **l'insieme X dei suoi dati di ingresso**
- **l'insieme Y delle risposte corrette**

si dice ***funzione caratteristica*** **del problema P**

$$f_P: X \rightarrow Y$$

la funzione f_P che associa a ogni dato d'ingresso $x \in X$ la corrispondente risposta corretta $y \in Y$.

In tal modo, "**problema (ir-)risolubile**" diventa sinonimo di "**funzione caratteristica (non) computabile**", concetto mappabile sul formalismo della Macchina di Turing.

FUNZIONE COMPUTABILE

- Si tratta ora di formalizzare quando una ***funzione caratteristica*** sia ***computabile***
- Di nuovo, poiché l'arma più potente è la MdT, è logico appoggiare ad essa la definizione:

FUNZIONE COMPUTABILE

Una funzione f: A→B è *computabile* se esiste una Macchina di Turing che

- data sul nastro una rappresentazione di **x∈A**

dopo un numero nito di passi

- produce sul nastro una rappresentazione di **f(x)∈B**

Perché f ***non*** sia computabile occorre dunque che la MdT non riesca a produrre un risultato *in un numero nito di passi,* ossia che *non si fermi* (ovvero, che entri in un ciclo in nito).

FUNZIONI DEFINIBILI vs. FUNZIONI COMPUTABILI

- Vogliamo ora capire ***se tutte le funzioni siano computabili*** o se esistano invece funzioni definibili ma non computabili
- Per rispondere, occorre confrontare:
  - le funzioni che possiamo *definire*

con

  - le funzioni che possiamo *computare* con una MdT.

FUNZIONE COMPUTABILE su $\mathbb{N}$

- Per semplificare la trattazione, nel seguito considereremo solo funzioni ***sui numeri naturali***

$$\mathbf{f: \mathbb{N} \to \mathbb{N}}$$

- Questo *non è limitativo*, perché:
  - ogni informazione è necessariamente finita
  - come tale, può essere codificata con una *collezione di numeri naturali*
  - la quale a sua volta può essere espressa con *un unico numero naturale* mediante il ***procedimento di Gödel***

IL PROCEDIMENTO DI GÖDEL

- **Obiettivo:** data una ***collezione di numeri naturali***, esprimerla con ***un unico numero naturale***.
- **Procedimento**
  - siano $N_1, N_2, \ldots N_k$ i numeri naturali dati
  - siano $P_1, P_2, \ldots P_k$ i primi *k* numeri **primi**
  - sia ***R*** un ***nuovo numero naturale*** così definito:

$$R ::= P_1^{N1} \cdot P_2^{N2} \cdot \ldots \cdot P_k^{Nk}$$

Grazie all'unicità della scomposizione in fattori primi, *R* rappresenta univocamente, in modo compatto, la collezione originale $N_1, N_2, \ldots N_{k.}$

Quante sono le possibili MdT?

- Una Macchina di Turing è definita dalla quintupla

  $\langle A, S, mfn, sfn, dfn \rangle$

- A e` un insieme finito di simboli

  potrei associarli ai naturali 1, 2, 3... $|A|$

- S e` un insieme finito di stati

  li associo ai naturali 1, 2, … $|S|$

- *mfn, sfn* e *dfn* sono delle tabelle $|A| \times |S|$

  il valore di ogni cella della tabella può essere associato a un numero naturale

Quindi ogni MdT può essere associata ad una sequenza finita di naturali che col procedimento di Göedel può essere associato ad un unico naturale!

Quante sono le funzioni $\mathbb{N} \mapsto \mathbb{N}$?

Teorema: l'insieme delle funzioni da $\mathbb{N}$ a $\mathbb{N}$ non è numerabile

Dim: per assurdo, sia f_0, f_1, f_2 … una possibile sequenza che contiene tutte le funzioni da $\mathbb{N}$ a $\mathbb{N}$

		0	1	2	3	4	5	…
0	f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$	$f_0(5)$	…
1	f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	…
2	f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	…
3	f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	…
4	f_4	$f_4(0)$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	$f_4(5)$	…
…								

- Costruiamo una funzione g:

g =	$f_0(0)+1$	$f_1(1)+1$	$f_2(2)+1$	$f_3(3)+1$	$f_4(4)+1$	$f_5(5)+1$	…

- La funzione g in quale posizione si trova nella sequenza?

Quante sono le possibili MdT?

- Quindi l'insieme delle possibili Macchine di Turing è numerabile
- Ogni MdT è associata ad un numero naturale!
- Quindi ogni algoritmo può essere associato ad un naturale
- In effetti, anche i programmi possono essere associati ad un numero naturale:

```
main()
{ printf("hello");
}
```

109 97 105 110 40 41 10 123 32 112 114 105 110 116 102 40 34 104 101 108 108 111 34 41 59 10 125

la codifica ASCII di un programma è una sequenza di interi da 0 a 127, che possono essere pensati come le cifre di un numero in base 128

FUNZIONI DEFINIBILI vs. FUNZIONI COMPUTABILI

- La situazione però è meno grigia di quel che sembra
- Se è vero che la gran parte delle funzioni ***definibili*** non è computabile, è anche vero che ***molte di esse non ci interesseranno mai***
  - le sole funzioni che ci interessano davvero sono infatti quelle **definibili con un linguaggio basato su un *alfabeto finito* di simboli,** che costituiscono un insieme ***enumerabile***
- Ergo, potenzialmente, tutte le funzioni che ci interessano *potrebbero* essere calcolabili… **ma *non è così***
  - sfortunatamente, i due insiemi *non coincidono*
  - esistono funzioni perfettamente definibili con un alfabeto finito, ma *non computabili*

FUNZIONI NON COMPUTABILI

- **Dire che esistono funzioni definibili ma non computabili equivale, purtroppo, a dire che *esistono problemi irrisolubili***
- Per dimostrarlo, basta trovarne uno..

PROBLEMA DELL' *HALT* DELLA MACCHINA DI TURING

Stabilire se una data macchina di Turing *T*, con un generico ingresso *X*, si ferma oppure no.

La funzione `halts()`

- Supponiamo che esista una funzione
  `int halts(char filename[]);`
  scritta, ad es. in linguaggio C, che prende in ingresso un programma .c (dato come file di testo) e che fornisce
- 1, se il programma termina
- 0, se il programma va in loop

Utilissima!!

Congettura di Goldbach

Ogni numero *n* pari e >2 e` la somma di due numeri primi

file `goldbach.c`

```
int sommaDiPrimi(int n);

main()
{ int n=4;
  while (sommaDiPrimi(n))
     n=n+2;
}
```

```
main()
{printf(halts("goldbach.c"));
}
```

Che cosa fa questo programma?

- **`paradox()`** termina o no?

file `paradox.c`

```
int halts(char filename[]);

int paradox()
{  if (halts("paradox.c"))
       while (1)
           ;
   else return 1;
}
main()
{  paradox();
}
```

HALT della MACCHINA DI TURING (1/7)

Questo problema è **perfettamente de nito,**
ma, nel caso generale, ***non è computabile***

Dimostrazione

- Siano:
  - **M** l'insieme di tutte le Macchine di Turing
  - **X** l'insieme di tutti i possibili ingressi
- Siano inoltre:
  - **x** $\in$ **X** un generico dato di ingresso, rappresentato come intero (Gödel)
  - $\mathbf{T_m} \in \mathbf{M}$ una generica Macchina di Turing, rappresentata con il suo indice **m** (Gödel)
- Indichiamo in ne con:
  - $\perp$ un *risultato inde nito*, dovuto a una Macchina di Turing che non risponde (entra in un ciclo in nito)

HALT della MACCHINA DI TURING (2/7)

La funzione caratteristica f_{HALT} di questo problema può essere così de nita:

$$f_{HALT}(m,x) = \begin{cases} \textbf{1, se la macchina di indice m con ingresso x si ferma} \\ \textbf{0, se la macchina di indice m con ingresso x } \textbf{\textit{non}} \textbf{ si ferma} \end{cases}$$

Questa funzione è ***ben de nita*** ma, *nel caso generale,* ***non computabile***, perché tentare di calcolarla conduce a un assurdo.

HALT della MACCHINA DI TURING (3/7)

Se f_{HALT} è computabile, deve esistere una Macchina di Turing T_{HALT} capace di calcolarla.

Data la macchina T_{HALT}, è facile scriverne un'altra T_g che abbia la seguente funzione caratteristica f_g, che è *funzione soltanto di n*:

$$f_g(n) = \begin{cases} \textbf{1, se } f_{HALT}(n,n) = 0 \\ \perp\textbf{, se } f_{HALT}(n,n) = 1 \end{cases}$$

La MdT **n** con ingresso **n** *non* si ferma

La MdT **n** con ingresso **n** si ferma

HALT della MACCHINA DI TURING (5/7)

Come caso particolare, sia ora n = g

- prendiamo cioè come ingresso ***proprio quel particolare numero g*** *che rappresenta la Macchina di Turing* T_g che calcola $\mathbf{g_{HALT}}$
- ciò equivale a dare come ingresso a TG *la sua stessa descrizione*

L'e etto è quello di generare un assurdo. Infatti..

HALT della MACCHINA DI TURING (6/7)

Sostituendo **g** al posto di **n** si ottiene:

$$f_g(g) = \begin{cases} 1, \text{ se } f_{HALT}(g,g) = 0 \\ \perp, \text{ se } f_{HALT}(g,g) = 1 \end{cases}$$

che è ***palesemente assurdo***, dato che:

- se T_g si ferma (f_g dà come risultato 1), f_{HALT} dice che T_g ***non*** si ferma ($f_{HALT}(g,g)$ vale 0)
- se T_g ***non*** si ferma (f_g è inde nita), f_{HALT} dice che T_g si ferma ($f_{HALT}(g,g)$ vale 1)

Generabilità e Decidibilità di insiemi

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per la prima versione dei lucidi

HALT della MACCHINA DI TURING (7/7)

Conclusione:

La funzione caratteristica di questo problema ***non è computabile*** nel caso generale.

Decidere se una data Macchina di Turing T, con un generico ingresso, si fermi oppure no *è un problema INDECIDIBILE*

Il prossimo passo è indagare la decidibilità dei problemi che ci interessano – in particolare, *la decidibilità di un insieme.*

GENERABILITÀ e DECIDIBILITÀ DI INSIEMI

- Poiché un linguaggio è – come vedremo – un insieme di frasi, ci interessa indagare in generale il problema della *generabilità* e della *decidibilità* di un insieme.
- L'analisi matematica introduce il concetto di ***insieme numerabile*** (i cui elementi possono essere contati)
  - Esiste una funzione di corrispondenza coi naturali
- A noi questo non basta: vogliamo che **tale funzione sia computabile,** affinché l'insieme sia ***effettivamente generabile*** da una Macchina di Turing
  - Concetto di **insieme *ricorsivamente enumerabile***

Si ringrazia il Prof. Enrico Denti per la prima versione dei lucidi
Vietata la diffusione e la riproduzione non autorizzata

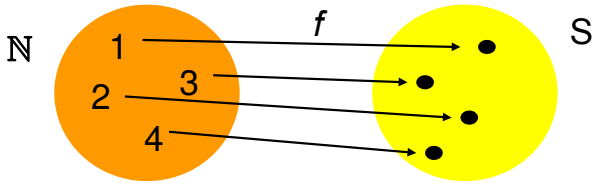
Dagli INSIEMI NUMERABILI...

[Analisi matematica]

Un insieme ***numerabile*** è un ***insieme i cui elementi possono essere "contati",*** ossia che possiede una ***funzione biettiva***

$$f: \mathbb{N} \rightarrow S$$

che mette in corrispondenza *i numeri naturali* con gli *elementi dell'insieme.*



..agli INSIEMI RICORSIVAMENTE ENUMERABILI (SEMI-DECIDIBILI)

- Un insieme è ***ricorsivamente enumerabile*** (***semi-decidibile***) se tale funzione è ***computabile.***
  - In tal caso una MdT, computando la funzione, può *generare uno ad uno* gli elementi dell'insieme.
- L'insieme è quindi *e ettivamente generabile*, calcolando elemento per elemento la funzione f.

INSIEMI RICORSIVAMENTE ENUMERABILI (SEMI-DECIDIBILI)

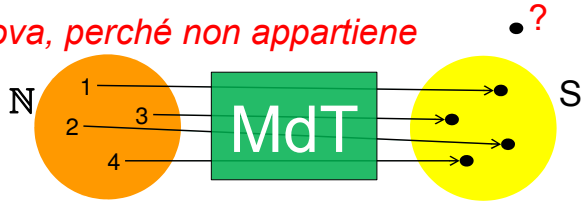
DOMANDA:

Se **l'insieme *S* è ricorsivamente enumerabile** posso capire se un dato elemento *x* appartiene all'insieme?

IL PROBLEMA DI DECIDERE

Come potrebbe una MdT decidere se un dato elemento appartiene a un insieme?

- In un insieme ricorsivamente enumerabile, potrebbe generare uno ad uno gli elementi dell'insieme, *fermandosi se e quando lo trova.*
- Già, ma.. *se **non** lo trova, perché non appartiene all'insieme?*
- La MdT va in loop: non risponderà mai.



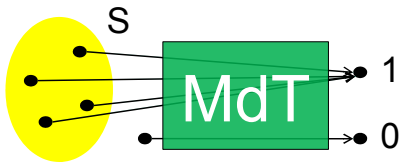
- **Semi-decidibile** indica che si riesce a decidere se appartiene (in positivo), ma non se «non appartiene»

INSIEMI DECIDIBILI (o RICORSIVI)

Occorre dunque un concetto più forte e completo della semi-decidibilità.

Un insieme S è **decidibile** (o ricorsivo) **se la sua funzione caratteristica**:

$$f(x) = \begin{cases} 1, \text{ se } x \in S \\ 0, \text{ se } x \notin S \end{cases}$$



è *computabile*

ossia, **se esiste una Macchina di Turing capace di rispondere sì o no, *senza entrare mai in un ciclo in nito*, alla domanda se un certo elemento appartiene all'insieme.**

INSIEMI DECIDIBILI (o RICORSIVI)

TEOREMA 1

Se un insieme è *decidibile* è anche *semidecidibile*

Ovvio: se si può decidere sia se appartiene sia se non appartiene, si può chiaramente decidere se appartiene.
Non è detto che valga il viceversa

TEOREMA 2

Un insieme *S* è *decidibile* *se e solo se*

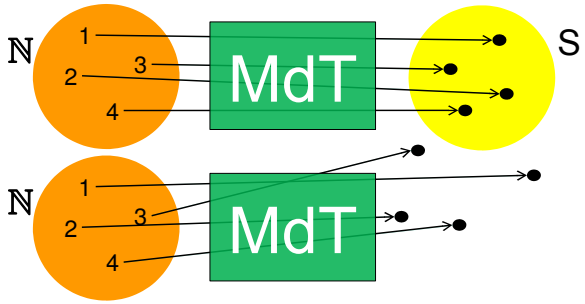
- **sia *S***
- **sia il suo complemento *N-S***

sono *semidecidibili.*

IL PROBLEMA DI DECIDERE (2)

Il Teorema 2 è la chiave della questione:

- in un insieme ricorsivamente enumerabile si va in loop se l'elemento non è presente perché si continua a generare elementi, sperando (inutilmente) di trovarlo
- Richiedendo che *anche il complemento* sia semi-decidibile, l'elemento dato prima o poi sarà sicuramente generato da una delle due MdT, evitando il loop.



INSIEMI & LINGUAGGI

Perché questo è importante?

- I linguaggi di programmazione sono costruiti a partire da un **alfabeto finito**
- Ogni linguaggio è caratterizzato dall'**insieme** (infinito) **delle sue *frasi lecite.***

- **Non basta** che tale insieme possa essere **generato (semi-decidibilità)**
  - ossia, che si possano generare le frasi "lecite" previste
- **è indispensabile poter decidere se una frase è giusta o sbagliata *senza entrare in ciclo infinito***
  ⇒ ***decidibilità***

INSIEMI & LINGUAGGI

- Se così non fosse, il compilatore o interprete ***potrebbe non rispondere,*** entrando in un ciclo infinito), **davanti a una frase errata**

mentre ovviamente *noi vogliamo che si fermi* e segnali l'errore.