1

Teoria della computazione

- Quali problemi sappiamo risolvere
  - Con quali macchine
  - In assoluto
- A prima vista la domanda può sembrare troppo generale:
  - Che cosa intendiamo per problema?
    Un calcolo matematico; una decisione di un'assemblea di condominio; il prelievo di contante dal Bancomat; …?
  - Quante e quali macchine astratte dobbiamo considerare?
  - Che significa saper risolvere "in assoluto" un problema:
    Pinco può essere più capace di pallino;
    Se non so risolvere un problema con un mezzo potrei riuscire a risolverlo con un altro.

2

In realtà siamo già in grado di inquadrare per bene il tema pur nella sua generalità

- Ricordiamo in primis che il concetto di linguaggio ci permette di formalizzare qualsiasi "problema informatico":
  - $x \in L$?
  - $y = \tau(x)$
- In realtà anche le due formulazioni di cui sopra possono essere ricondotte l'una all'altra:
  - Se ho una macchina che sa risolvere il problema $y = \tau(x)$ e voglio usarla per risolvere un problema $x \in L$?, mi basta definire $\tau(x) = 1$ se $x \in L$, $\tau(x) = 0$ se $x \notin L$.
  - Viceversa, se ho una macchina che sa risolvere il problema $x \in L$?, posso definire il linguaggio $L_\tau = \{x\$y \mid y = \tau(x)\}$ dopodiché, fissato un certo x, enumero tutte le possibili stringhe y sull'alfabeto di uscita e per ognuna di esse domando alla macchina se $x\$y \in L_\tau$: prima o poi, *se $\tau(x)$ è definita*, troverò quella per cui la macchina risponde positivamente: quella è la y cercata (ricordiamo il gioco di indovinare un oggetto formulando domande e ottenendo risposte "binarie"). Procedimento forse "lunghetto" ma in questo momento la lunghezza del calcolo non ci interessa.

3

- Quanto alla macchina di calcolo … effettivamente ce n'è una miriade, oltre quelle che conosciamo; e ben di più se ne possono inventare ⇒ al massimo potrei ambire a risultati del tipo
  - $\{a^n b^n | n > 0\}$ può essere riconosciuto da un AP e da una MT ma non da un FSA.
- In realtà, a ben pensare, abbiamo già osservato che non è poi così facile "superare" la MT: aggiungere nastri, testine, nondeterminismo, … non produce aumento di potenza (nel senso dei linguaggi riconoscibili); non è poi così difficile far fare alla MT ciò che fa un normale calcolatore: basta simulare la memoria dell'uno con quella dell'altra (o viceversa) (torneremo su questo tema in maniera meno ovvia quando ci interesserà esaminare anche il *costo* delle computazioni)

con una generalizzazione potente e audace:

4

Tesi di Church (e Turing e altri): Anni 30!!

- Non esiste meccanismo di calcolo automatico superiore alla MT o ai formalismi ad essa equivalenti.
  Fin qui potrebbe essere un *Teorema* di Church (da aggiornare ogni volta che qualcuno si sveglia la mattina con un nuovo modello di calcolo)
- Nessun *algoritmo*, indipendentemente dallo strumento utilizzato per implementarlo, può risolvere problemi che non siano risolvibili dalla MT: la MT è il calcolatore più potente che abbiamo e che potremo mai avere!
- Allora è ben posta la domanda: "Quali sono i problemi risolvibili algoritmicamente o automaticamente che dir si voglia?":
  Gli stessi risolvibili dalla semplicissima MT!

5

E allora studiamo per bene le MT

- Esistono problemi che non si possono risolvere?
- Come si fa a capirlo?

- Le risposte che troveremo varranno anche per programmi C, Modula, transputer, ….

6

Primo fatto:

- Le MT sono *algoritmicamente enumerabili*
- Enumerazione di un insieme S:
- E: S ↔ N
- Enumerazione algoritmica: E può essere calcolata mediante un algoritmo … cioè mediante una MT
- Enumerazione algoritmica di $\{a, b\}^*$:

{ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, …}
↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕ ↕
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ….}

7

- Tanto per fissare le idee e per semplicità, senza perdere generalità alcuna:
- Fissiamo un alfabeto (unico) A
  (negli esempi |A| = 2, A = {0, 1} (più il blank '_'))
- MT a nastro singolo
- Ulteriori convenzioni seguiranno
- Lasciando stare le MT a uno stato … osserviamo quelle a due stati:

	0	1
q_0	⊥	⊥
q_1	⊥	⊥

MT_0

	0	1
q_0	⊥	⊥
q_1	⊥	$<q_0, 0, S>$

MT_1 .....

8

- Quante MT con due stati?
  $\delta: Q \times A \rightarrow Q \times A \times \{R,L,S\} \ldots \cup \{\perp\}$
- In generale: quante $f: D \rightarrow R$?
  - $|R|^{|D|}$
- Con |Q| = 2, |A| = 2 (+ '_'), $(2*3*3+1)^{(2*3)} = 19^6$ MT a due stati
- Mettiamo in ordine queste MT: $\{M_0, M_1, \ldots M_{196-1}\}$
- Poi mettiamo in ordine alla stessa maniera le $(3*3*3+1)^{(3*3)}$ MT con 3 stati e così via.
- Otteniamo un'enumerazione $E: \{MT\} \leftrightarrow N$
- E è algoritmica o effettiva: *pensiamo* a scrivere un programma C (ovvero una MT) che, dato n, costruisce la n-esima MT (ad esempio una tabella che definisce δ) e viceversa, data una (tabella che descrive una) MT M, dice in che posizione è: E(M).
- E(M): *numero di Göedel* di M, E: göedelizzazione.

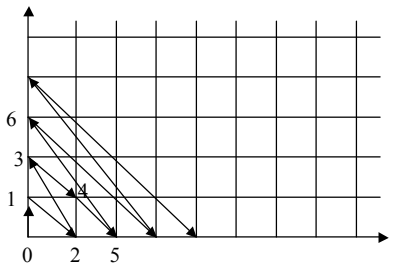
9

- Ulteriore convenzione: visto che stiamo parlando di numeri, d'ora in avanti, e per un po':
- Problema = calcolo di una funzione f: $\mathbb{N} \rightarrow \mathbb{N}$
- f_y = funzione calcolata dalla y-esima MT
- NB: $f_y(x) = \perp$ se M_y non si ferma quando riceve in ingresso x
- Aggiungiamo la convenzione $f_y(x) = \perp$ se *e solo se* M_y non si ferma quando riceve in ingresso x
  - basta far sì che una qualsiasi MT che si fermasse in uno stato che non porta alla definizione di un valore significativo $f_y(x)$ si porti in un nuovo stato che la fa procedere all'infinito, ad esempio spostando la testina sempre a destra, senza più fermarsi (è solo una comodità)

10

Secondo fatto:

- Esiste una *Macchina di Turing Universale* (MTU): la MT che calcola la funzione $g(y,x) = f_y(x)$
- Ad essere precisi la MTU così definita non sembra appartenere alla famiglia $\{M_y\}$ perché f_y è funzione di una variabile, mentre g è funzione di due variabili
- Però noi sappiamo che $\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$: ad esempio:



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

11

- Possiamo codificare g(y,x) come una $\hat{g}(n) = g(d^{-1}(n))$
  NB: d e d^{-1} sono computabili
- Schema di una MTU che calcola $\hat{g}$ (d'ora in poi per semplicità scriveremo direttamente g):
  - Dato n calcolo $d^{-1}(n) = \langle y,x \rangle$
  - Costruisco la tabella di M_y (calcolando $E^{-1}(y)$) e la memorizzo nel nastro della MTU:

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

  - In un'altra porzione di nastro simulo la configurazione di M_y

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

NB: I simboli speciali #, $ e gli stati sono codificati come stringhe di 0 e 1

Alla fine MTU lascia sul nastro $f_y(x)$, se e solo se M_y termina la computazione su x

12

- La MT è un modello molto astratto e semplice di calcolatore
- Approfondendo l'analogia:
- MT: calcolatore a programma cablato.
  Una MT "normale" esegue sempre lo stesso algoritmo, calcola sempre la stessa funzione
- MTU: calcolatore a programma memorizzato:
  y = programma
  x = dati del programma

13

Torniamo alla domanda "quanti e quali problemi sono risolvibili algoritmicamente?"

- Quante e quali sono le funzioni f_y: $\mathbb{N} \rightarrow \mathbb{N}$?
- Cominciamo con il "Quante":
- $\{f: \mathbb{N} \rightarrow \mathbb{N}\} \supseteq \{f: \mathbb{N} \rightarrow \{0,1\}\} \Rightarrow$
  $|\{f: \mathbb{N} \rightarrow \mathbb{N}\}| >= |\{f: \mathbb{N} \rightarrow \{0,1\}\}| = \wp(\mathbb{N}) = 2^{\aleph_0}$
- D'altro canto l'insieme $\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}$ è per definizione numerabile:
  NB: E: $\{M_y\} \leftrightarrow \mathbb{N}$ induce $E^{\wedge}$: $\mathbb{N} \rightarrow \{f_y\}$ non biunivoca
  (in molti casi $f_y = f_z$, con $z \neq y$) ma basta per asserire
- $|\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}| = \aleph_0 < 2^{\aleph_0} \Rightarrow$
- la "gran parte" delle funzioni (problemi) non è calcolabile (risolvibile) algoritmicamente!!

14

È proprio una grave perdita?

- In realtà quanti sono i problemi *definibili*?
- Per definire un problema usiamo una frase (stringa) di un qualche linguaggio:
  - $f(x) = x^2$
  - $f(x) = \int_a^x g(z)\, dz$
  - "Il numero che moltiplicato per se stesso dà y"
  - …
- Ma un linguaggio è un sottoinsieme di A^*, che è un insieme numerabile →
- L'insieme dei problemi *definibili* è pure numerabile, come quello dei problemi *risolvibili* (algoritmicamente).
  - Possiamo ancora sperare che coincidano
- Certamente {Problemi risolvibili} ⊆ {Problemi definibili}
  (Una MT *definisce* una funzione, oltre a calcolarla)

15

Passiamo allora alla domanda:
"Quali sono i problemi risolvibili?"

- Il problema della terminazione del calcolo (molto "pratico"):
  - Costruisco un programma
  - Fornisco dei dati in ingresso
  - So che in generale il programma potrebbe non terminare l'esecuzione (in gergo: "andare in loop")
  - Posso determinare se questo fatto si verificherà?
- In termini -assolutamente equivalenti- di MT:
  - Data la funzione $g(y,x) = 1$ se $f_y(x) \neq \perp$, $g(y,x) = 0$ se $f_y(x) = \perp$
  - Esiste una MT che calcola g?

16

Risposta: NO!

- Ecco perché il compilatore (un programma) non può segnalarci nella sua diagnostica che il programma che abbiamo scritto andrà in loop con certi dati (mentre può segnalarci se abbiamo dimenticato un **end**):
- Stabilire se un'espressione aritmetica è ben parentetizzata è un problema risolvibile (decidibile);
- Stabilire se un dato programma con un dato in ingresso andrà in loop è un problema irrisolvibile (indecidibile) algoritmicamente
  - ne vedremo molti altri: sono "molte" (anche in termini qualitativi) le cose che il calcolatore non sa fare

Dimostrazione

- Sfrutta una tipica *tecnica diagonale* utilizzata anche nel teorema di Cantor per dimostrare che $\aleph_0 < 2^{\aleph 0}$
- Ipotesi assurda:
  - $g(y,x) = 1$ se $f_y(x) \neq \perp$, $g(y,x) = 0$ se $f_y(x) = \perp$ computabile
- Allora anche
  - $h(x) = 1$ se $g(x,x) = 0$ ($f_x(x) = \perp$), $\perp$ altrimenti ($f_x(x) \neq \perp$) è computabile

    NB: ci siamo posti sulla diagonale y = x e abbiamo scambiato il si col no, poi abbiamo fatto in modo che il no diventasse una nonterminazione (sempre fattibile)
- Se h è computabile, $h = f_{x_0}$ per qualche x_0.
- Domanda $h(x_0) = 1$ o $h(x_0) = \perp$?

$h(x) = 1$ se $g(x,x) = 0$ ($f_x(x) = \perp$),
$h(x) = \perp$ se $g(x,x) = 1$ ($f_x(x) \neq \perp$),
$h = f_{x_0}$

- Supponiamo $h(x_0) = f_{x_0}(x_0) = 1$
  Ciò significa $g(x_0,x_0) = 0$ *ovvero* $f_{x_0}(x_0) = \perp$:
  Contraddizione!

- Allora supponiamo il contrario: $h(x_0) = f_{x_0}(x_0) = \perp$
  Ciò significa $g(x_0,x_0) = 1$ *ovvero* $f_{x_0}(x_0) \neq \perp$:
  Contraddizione!
- Contraddizione in ambo i casi ⇒ ipotesi di partenza (g(y,x) computabile) è falsa, quindi g(y,x) non è computabile

  QED

- Vedremo che "in conseguenza" di questa irrisolvibilità molti altri problemi risultano irrisolvibili.
- Per il momento:

19

- Un altro problema irrisolvibile (corollario?)
  - La funzione $h(x) = 1$ se $f_x(x) \neq \perp$; $= 0$ se $f_x(x) = \perp$ non è calcolabile
- Si tratta di un *lemma*, al limite, non di un corollario
  - Infatti esso costituisce il "cuore" della dimostrazione del teorema precedente
  - Di per se stesso l'enunciato non significa molto
  - E' però importante menzionarlo per sottolineare che il suo enunciato *non* può essere ricavato come conseguenza immediata del teorema precedente (che è *più generale*):
- *NB: in generale, se un problema è irrisolvibile, può darsi che un suo caso particolare diventi risolvibile (vedremo esempi irrisolvibili per linguaggi qualsiasi, risolvibili per linguaggi regolari); invece una sua generalizzazione è necessariamente pure irrisolvibile.*
- *Al contrario, se un problema è risolvibile, può darsi che una sua generalizzazione diventi irrisolvibile mentre una sua particolarizzazione rimane certamente risolvibile.*

20

Un altro importante problema indecidibile

- La funzione
  $k(y) = 1$ se f_y è totale, ossia $f_y(x) \neq \perp \ \forall x \in \mathbb{N}$; $= 0$ altrimenti
  non è calcolabile
- E' un problema *simile ma diverso* dal precedente.
  - Qui abbiamo una quantificazione rispetto a tutti i possibili dati in ingresso.
  - In certi casi potrei essere in grado di stabilire, $\forall x$, se $f_y(x) \neq \perp$, senza per questo poter rispondere alla domanda "f_y è una funzione totale?"
    (Ovviamente se trovo un x tale che $f_y(x) = \perp$, posso concludere che f_y non è totale, ma se non lo trovo?)
  - Viceversa, potrei essere in grado di concludere che f_y non è totale eppure potrei non poter decidere se $f_y(x) \neq \perp$ per un singolo x. (Certo, se invece concludessi che è totale f_y …)
- Dal punto di vista dell'impatto pratico questo teorema è forse ancor più significativo del precedente:
  - dato *un certo programma*, voglio sapere se esso terminerà l'esecuzione per *qualsiasi dato* in ingresso o corre il rischio, per *qualche dato*, di andare in loop.
  - Nel caso precedente, invece, mi interessava sapere se *un certo programma con certi dati* avrebbe terminato o meno l'esecuzione.

21

Dimostrazione (non indispensabile ma utile)

- Meccanismo standard: assurdo + diagonale, con qualche dettaglio tecnico in più.
- Ipotesi assurda: $k(y) = 1$ se f_y è totale, ossia $f_y(x) \neq \perp \; \forall \; x \in \mathbb{N}$; altrimenti $= 0$ calcolabile, e, ovviamente, totale per definizione
- Definisco allora $g(x) = w$ = indice (numero di Göedel) della x-esima MT (in $\mathbb{E}$) che calcola una funzione totale.
- Se k è calcolabile e totale, allora lo è anche g:
  - calcolo $k(0), k(1), \ldots$, sia w_0 il primo valore tale che $k(w_0) = 1$, allora pongo $g(0) = w_0$;
  - procedendo pongo $g(1) = w_1$, essendo w_1 il secondo valore tale che $k(w_1) = 1$; …
  - il procedimento è algoritmico; inoltre, essendo le funzioni totali infinite, $g(x)$ è certo definita per ogni x, ergo è totale.
- g è anche strettamente monotona: passando da x a $x+1$, w_{x+1} è certo > di w_x;
- quindi g^{-1} è una funzione, pure strettamente monotona, anche se non totale: $g^{-1}(w)$ è definita solo se w è il numero di Göedel di una funzione totale.
- Definisco ora

  (α) $h(x) = f_{g(x)}(x) + 1 = f_w(x) + 1$: f_w è calcolabile e totale e quindi anche h lo è $\Rightarrow$

  (β) $h = f_{w_h}$ per qualche w_h; siccome h è totale, $g^{-1}(w_h) \neq \perp$, poniamo $g^{-1}(w_h) = x_h$

22

(α) $h(x) = f_{g(x)}(x) + 1 = f_w(x) + 1$
(β) $h = f_{w_h}$ per qualche w_h; $g^{-1}(w_h) = x_h$

- Quanto vale $h(x_h)$?
  - $h(x_h) = f_{g(x_h)}(x_h) + 1 = f_{w_h}(x_h) + 1$ (da (α))
  - $h = f_{w_h} \Rightarrow h(x_h) = f_{w_h}(x_h)$ (da (β))
- *Contraddizione!*

23

NB (*molto critico*): sapere che un problema è risolvibile non vuol dire saperlo risolvere!

- In matematica spesso ottengo dimostrazioni non costruttive: dimostro che la soluzione di un problema esiste (ed è unica) ma non per questo fornisco un modo di calcolarla
- Nel nostro caso:
  - un problema è risolvibile se *esiste* una MT che lo risolve
  - per certi problemi posso arrivare alla conclusione che *esiste* una MT che li risolve ma *non per questo essere in grado di fornirla*
- Cominciamo con un caso banale
  - Il "problema" consiste nel rispondere a una domanda la cui risposta è necessariamente Sì o No:
    - E' vero che il numero di molecole dell'universo è $10^{10^{10^{10^{10}}}}$?
    - E' vero che la "partita a scacchi perfetta" termina in parità?
    - (20 anni fa ...) E' vero che $\neg(\exists x, y, z, w \in \mathbb{N} \mid x^y + z^y = w^y \wedge y>2)$?
    - ….

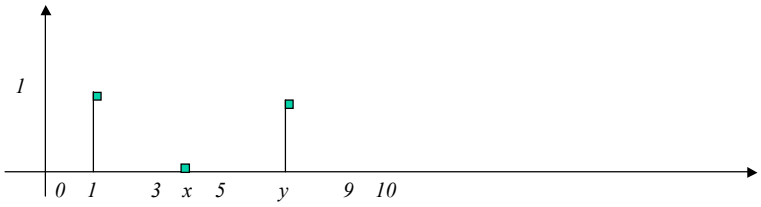
24

- In questi casi so a priori che la risposta è Si o No anche se non so (sapevo) quale sia (fosse).
- La cosa stupisce un po' meno se ricordiamo che:
  - Problema = funzione
  - risolvere un problema = calcolare una funzione
- Che funzione associamo ai problemi di cui sopra?
  - Codificando TRUE = 1; FALSE = 0, *tutti* i problemi sono espressi da una delle due funzioni:
    - $f_{true}(x) = 1, \forall x$
    - $f_{false}(x) = 0, \forall x$
  - Entrambe le funzioni sono banalmente calcolabili →

    Qualsiasi sia la risposta al problema essa è *calcolabile* ma *non necessariamente nota*.
- In maniera un po' più astratta:
  $g(10,20) = 1$ se $f_{10}(20) \neq \perp$, $g(10,20) = 0$ se $f_{10}(20) = \perp$
  $g(100,200) = 1$ se $f_{100}(200) \neq \perp$, $g(100,200) = 0$ se $f_{100}(200) = \perp$
  $g(7,28) = 1$ se $f_{7}(28) \neq \perp$, $g(7,28) = 0$ se $f_{7}(28) = \perp$
  ….
  sono tutti problemi risolvibili, anche se non è detto che noi ne conosciamo la soluzione.

25

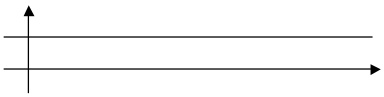
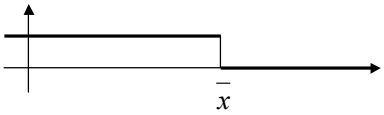
- Esaminiamo ora casi un po' meno banali ma molto istruttivi.
- *f(x)* = *x*-esima cifra dell'espansione decimale di π.
  - *f* è sicuramente calcolabile e conosciamo algoritmi (MT) che la calcolano
- Basandosi sulla capacità di calcolare *f* (allo *stato attuale* non si hanno altre fonti di conoscenza) investighiamo la calcolabilità di
  - *g(x) = 1 se in π esistono esattamente x 5 consecutivi, 0 altrimenti*
    - Calcolando la sequenza:
      {f(0) = 3, f(1) = 1, f(2) = 4, f(3) = 1, f(4) = 5, f(5) ≠ 5, …}
      Otteniamo g(1) = 1
  - In generale il grafico di g sarà del tipo seguente:



26

- Per qualche valore di *x* potremmo scoprire che *g(x) = 1*
  - anzi, se *g(x) = 1*, prima o poi, pur di aver pazienza, lo scopriamo (approfondiremo questo fatto più avanti)
- Che elementi abbiamo per concludere, ad esempio, che *g(100.000.000)=0* se dopo aver calcolato *f(1.000.000.000.000.000)* non abbiamo ancora trovato 100.000.000 5 consecutivi?
  - *Allo stato attuale* di conoscenze (personali), nessuno!
- Possiamo però escludere che sia vera la seguente congettura?
  - "Qualsiasi sia *x*, pur di espandere un numero sufficientemente grande di cifre di π, prima o poi si troveranno esattamente *x* 5 consecutivi"
- Se essa fosse vera, ne ricaveremmo che *g* è la funzione costante *g(x) = 1* ∀ *x* e scopriremmo quindi anche che la sappiamo calcolare.
- In conclusione, *allo stato attuale*, non possiamo concludere né che *g* sia calcolabile, né che non lo sia.

27

- Consideriamo ora la seguente "lieve" modifica di g:
  - *h(x) = 1 se in* π $\exists$ ***almeno*** *x 5 consecutivi, 0 altrimenti*
- Ovviamente, se *g(x) = 1* anche *h(x) = 1*
- Osserviamo però che se per qualche *x h(x) = 1,* allora *h(y) = 1* $\forall\, y \leq x$. Ciò significa che il grafico di *h* è di uno dei due tipi seguenti:
  - 1)  ($h(x) = 1\ \forall x$)
  - 2)  $h(x) = 1\ \forall\ x \leq \bar{x}$; $h(x) = 0\ \forall\ x > \bar{x}$

28

- Quindi *h* appartiene sicuramente all'insieme delle funzioni

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

- Si noti che ognuna delle funzioni di questo insieme é banalmente calcolabile (fissato $\bar{x}$ è immediato costruire una MT che calcola $h_{\bar{x}}$; idem per $\bar{h}$)
- Dunque *h* è sicuramente calcolabile: *esiste* una MT che la calcola
- Sappiamo calcolare *h*?
  Attualmente no: tra le infinite MT che calcolano le funzioni dell'insieme suddetto non sappiamo quale sia quella giusta!

29

Decidibilità e semidecidibilità
Ovvero:
1/2 + 1/2 = 1

- Concentriamoci su problemi formulati in modo tale che la risposta sia di tipo binario:
  - Problema = insieme $S \subseteq \mathsf{N}$: $x \in S$?
- Funzione caratteristica di un insieme S:
  - $c_S(x) = 1$ se $x \in S$, $c_S(x) = 0$ se $x \notin S$
- Un insieme S è *ricorsivo (o decidibile)* se e solo se la sua funzione caratteristica è computabile
  - NB: c_S è totale per definizione

30

- S è *ricorsivamente enumerabile* (RE) (o *semidecidibile*) se e solo se:
  - S è l'insieme vuoto ($S = \varnothing$)

    oppure
  - S è l'insieme immagine di una funzione totale e computabile:

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

    $\Rightarrow$

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

    da qui il termine "ricorsivamente (algoritmicamente) enumerabile"
- possiamo spiegare in termini intuitivi anche l'attributo "semidecidibile":
  - se $x \in S$, *enumerando* gli elementi di S prima o poi trovo x e sono in grado di ottenere la risposta giusta (Sì) alla domanda;
  - ma se $x \notin S$?
- una conferma più formale ci viene dal seguente ...

31

Teorema

A. Se S è ricorsivo è anche RE
   - *decidibile* è più che (non meno di) *semidecidibile*

B. S è ricorsivo se e solo se S stesso e il suo complemento $\hat{S} = \mathbb{N} - S$ sono RE
   - due semidecidibilità fanno una decidibilità
   - ovvero quando rispondere No equivale a (è ugualmente difficile che) rispondere Sì (e quando invece …)

- Corollario: gli insiemi (linguaggi, problemi, …) decidibili sono chiusi rispetto al complemento.
- Dimostrazione:

32

A. S ricorsivo ⇒ S RE

- Se S è vuoto esso è RE per definizione!
- Assumiamo allora $S \neq \emptyset$ e indichiamo con c_s la sua funzione caratteristica:
  $\Rightarrow \exists k \in S$, cioè tale che $c_s(k) = 1$
- Definiamo la funzione g_s:
  $g_s(x) = x$ se $c_s(x) = 1$, altrimenti $g_s(x) = k$
- g_s è totale, computabile e $I_{g_s} = S$
  $\Rightarrow$ S è RE
- NB: dimostrazione *non costruttiva*
  - sappiamo se $S \neq \emptyset$? Sappiamo calcolare g_s?
  - sappiamo solo che *esiste* g_s se $S \neq \emptyset$: è quanto ci basta!

B. S è ricorsivo ⇔ S e $\hat{S}$ = N - S sono RE

- B.1.1) S ricorsivo ⇒ S RE (parte A)
- B.1.2) S ricorsivo ⇒ $c_S(x)$ (= *1* se $x \in S$, $c_S(x) = 0$ se $x \notin S$) calcolabile
  ⇒ $c_{\hat{S}}(x)$ (= *0* se $x \in S$, $c_{\hat{S}}(x) = 1$ se $x \notin S$) calcolabile
  ⇒ $\hat{S}$ ricorsivo ⇒ $\hat{S}$ RE
- B.2) S RE ⇒ $S = \{g_S(0), g_S(1), g_S(2), g_S(3), ...\}$
  $\hat{S}$ RE ⇒ $\hat{S} = \{g_{\hat{S}}(0), g_{\hat{S}}(1), g_{\hat{S}}(2), g_{\hat{S}}(3), ...\}$
  Ma $S \cup \hat{S} = \mathbb{N}$, $S \cap \hat{S} = \varnothing$ perciò
  $\forall x \in \mathbb{N}\ (\exists y \mid x = g_S(y) \vee x = g_{\hat{S}}(y) \wedge \neg(\exists z,w \mid x = g_S(z) \wedge x = g_{\hat{S}}(w)))$
  (x appartiene a una e una sola delle due enumerazioni)
  ⇒ se costruisco l'enumerazione
  $\{g_S(0), g_{\hat{S}}(0), g_S(1), g_{\hat{S}}(1), g_S(2), g_{\hat{S}}(2), g_S(3), g_{\hat{S}}(3), ...\}$
  sono sicuro di trovarvi qualsiasi x:
  - se trovo x in un posto dispari concludo $x \in S$,
  - se lo trovo in posto pari concludo $x \in \hat{S}$.

  ⇒ So calcolare c_S ⇒ S è ricorsivo.

Alcuni enunciati con importanti risvolti pratici

- Se un generico insieme S ha le seguenti caratteristiche:
  - $i \in S \Rightarrow f_i$ totale
  - f totale e computabile ⇒ $\exists i \in S \mid f_i = f$

  Allora S non è RE
- Ciò significa: non esiste un formalismo (RE: automi, grammatiche, …) in grado di definire *tutte e sole* le funzioni calcolabili totali
  - i FSA, definiscono "funzioni totali" ma non tutte
  - le MT definiscono tutte le funzioni calcolabili, ma anche quelle non totali
  - il C permette di programmare qualsiasi algoritmo, ma anche quelli che non terminano: esiste un sottoinsieme RE del C che definisca solo i programmi che terminano? NO!
    - Ad esempio l'insieme dei programmi i cui loop siano solo cicli `for` con precise limitazioni può contenere solo programmi che terminano ma non tutti.

35

- Tentiamo un altro trucco per sbarazzarci delle scomode funzioni non totali (algoritmi che vanno in loop):
  - estendiamo una funzione, ad esempio arricchendo N con il nuovo *valore* $\{\perp\}$, oppure semplicemente attribuendo ad f un valore convenzionale quando f è indefinita.
    - Matematicamente l'operazione è perfettamente sensata (infatti in matematica pura si presta poca attenzione alle funzioni parziali)
- Il trucco non funziona:
  - Non esiste una funzione totale e computabile che sia un'estensione della funzione *computabile ma non totale* $g(x) = \{$se $f_x(x) \neq \perp$ allora $f_x(x) + 1$, *altrimenti* $\perp\}$
- Posso prendere una funzione parziale e farla diventare totale, ma nel farlo potrei perderne la computabilità: coperta corta!

36

- S è RE $\Leftrightarrow$

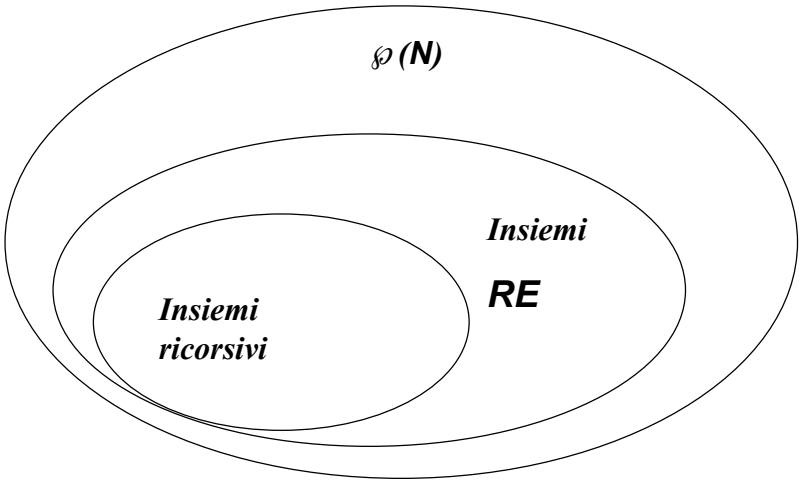
  $S = D_h$, con h computabile e parziale: $S = \{x |\ h(x) \neq \perp\}$

  oppure

  $S = I_g$, con g computabile e parziale: $S = \{x |\ x = g(y),\ y \in \mathbb{N}\}$
- Dimostrazione omessa (qui) ma usa una tecnica particolarmente utile e significativa (v. esercitazioni)
- Serve come Lemma per dimostrare che:
  Esistono insiemi semidecidibili che non sono decidibili:
  - $K = \{x |\ f_x(x) \neq \perp\}$ è semidecidibile
    - $K = D_h$ con $h(x) = f_x(x)$
  - Però sappiamo anche che $c_K(x)$ (= *1* se $f_x(x) \neq \perp$, *0* altrimenti) non è computabile

    $\Rightarrow$ K non è decidibile
- Conclusione:

37



I contenimenti sono tutti stretti

Corollario: gli insiemi RE (i linguaggi riconosciuti dalle MT) non sono chiusi rispetto al complemento

38

Il (potentissimo) Teorema di Rice

- Sia F un insieme di funzioni computabili:
  L'insieme S de(gli indici di) MT che calcolano funzioni di F ($S = \{x | f_x \in F\}$) è decidibile *se e solo se* F = ∅ o F è l'insieme di *tutte* le funzioni computabili

- ⇒ in tutti i casi non banali S non è decidibile!
  - La correttezza di un programma: P risolve il problema specificato? (M_x calcola la funzione costituente l'insieme $\{f\}$?)
  - L'equivalenza tra due programmi (M_x calcola la funzione costituente l'insieme $\{f_y\}$?)
  - Un programma gode di una qualsiasi proprietà riferibile alla funzione da esso calcolata (funzione a valori pari, funzione con insieme immagine limitato, …)?
  - …
- Sono solo alcuni tra gli innumerevoli esempi di problemi la cui indecidibilità discende banalmente dal teorema di Rice.

Come facciamo, in pratica, a stabilire se un problema è (semi)decidibile o no?

(ovviamente questo è a sua volta un problema indecidibile)

- Se troviamo un algoritmo che termina sempre → decidibile
- Se troviamo un algoritmo che potrebbe non terminare ma termina sempre se la risposta al problema è Sì → semidecidibile
- Ma se riteniamo che il problema in questione sia non (semi)decidibile, come facciamo a dimostrarlo?
  - Tentiamo di costruirci una nuova dimostrazione di tipo diagonale ogni volta? … staremmo freschi!
- In realtà abbiamo ora mezzi più comodi:
  - Un primo strumento potentissimo l'abbiamo già visto: il teorema di Rice
- Di fatto implicitamente abbiamo già usato più volte una tecnica naturale e generalissima:

La riduzione di problemi

- Se ho un algoritmo per risolvere il problema P lo posso sfruttare per risolvere il problema P':
  - Se so risolvere il problema della ricerca di un elemento in un insieme posso costruire un algoritmo per risolvere il problema dell'intersezione tra due insiemi
- In generale se trovo un algoritmo che, data un'istanza di P', ne costruisce la soluzione ricavandone un'istanza di P per il quale a sua volta so ricavare la soluzione, ho *ridotto* P' a P.
- Formalmente:
  - Voglio risolvere "$x \in S$?"
  - So risolvere "$y \in S'$?"
  - Se trovo una funzione t calcolabile e totale tale che $x \in S \leftrightarrow t(x) \in S'$ sono in grado di rispondere algoritmicamente alla domanda "$x \in S$?"
    - riduco rispondere a "$x \in S$?" a rispondere a "$y \in S'$?"

- Il procedimento può funzionare anche al contrario:
  - Voglio sapere se posso risolvere "$x \in S$?"
  - So di non saper risolvere "$y \in S'$?" (S' non è decidibile)
  - Se trovo una funzione t calcolabile e totale tale che $y \in S' \leftrightarrow t(y) \in S$ (cioè se riesco a ridurre "$y \in S'$?" a "$x \in S$?") ne concludo che "$x \in S$?" è non decidibile, altrimenti ne ricaverei la conseguenza assurda che anche S' è decidibile
- In realtà abbiamo usato questo meccanismo già varie volte in forma implicita:
  - Dall'indecidibilità del problema dell'halt della MT abbiamo concluso in generale l'indecidibilità del problema della terminazione del calcolo:
    - Ho una MT M_y e un numero intero x in ingresso
    - Costruisco un programma P (per esempio scritto in C) che simula M_y, e memorizzo x in un file di ingresso f
    - P termina la sua computazione su f se e solo se $g(y,x) \neq \perp$
    - Se sapessi decidere se P termina la sua computazione su f saprei risolvere anche il problema dell'halt della MT $\Rightarrow$ IMPOSSIBILE!
  - NB: avremmo potuto ridimostrare in modo diretto l'indecidibilità della terminazione dei programmi C enumerando i programmi e applicando la stessa tecnica diagonale … con un po' più di dettagli notazionali.

Un meccanismo abbastanza generale

- E' decidibile dire se durante l'esecuzione di un generico programma P si accede ad una variabile non inizializzata?
  - Supponiamo per assurdo che sia decidibile
  - Allora consideriamo il problema dell'halt e riduciamolo al problema nuovo come segue:
    - Dato un generico P' (che riceve in ingresso generici dati D), costruisco un P cosiffatto:

```
begin var x, y: …
      P'
      y := x;
end
```

      avendo cura di usare identificatori x e y che *non sono usati* in P'
  - E' chiaro che l'assegnamento y := x produce un accesso ad x, la quale *non è inizializzata* perché x non compare in P'
  - Quindi l'accesso ad una variabile non inizializzata accade in P se e solo se P' termina.
  - Allora, se sapessi decidere il problema dell'accesso ad una variabile non inizializzata, saprei decidere anche il problema della terminazione del calcolo, ciò che è assurdo.

43

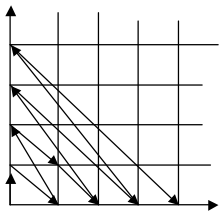
- La stessa tecnica può essere applicata per dimostrare l'indecidibilità di molte altre tipiche proprietà dei programmi durante la loro esecuzione:
  - Indici di array fuori dai limiti
  - Divisione per 0
  - Compatibilità dinamica tra tipi
  - …
  - Tipici errori a run-time: a questo proposito ...

44

- Riprendiamo in considerazione gli esempi precedenti
  - l'halt della MT
  - la divisione per 0 e gli altri errori a run-time, …
- Sono non decidibili ma semidecidibili:
  - se la MT si ferma, prima o poi lo scopro;
  - se esiste un dato x di un programma P tale per cui P tenti a un certo punto di eseguire una divisione per 0 prima o poi lo scopro …
- Fermiamoci un momento e apriamo una parentesi:
  - come scopro il fatto precedente: se io comincio ad eseguire P sul dato x e P non si ferma su x, come faccio a scoprire che eseguendo P su y, P eseguirà una divisione per 0?

45

- In generale: Teorema (formulazione astratta dei vari casi concreti precedenti):
  Il problema di stabilire se $\exists\, z |\, f_x(z) \neq \perp$ è *semidecidibile*
- Schema di dimostrazione
  - E' chiaro che se cerco di calcolare $f_x(0)$ e trovo che è $\neq \perp$ sono a posto;
  - Però se la computazione di $f_x(0)$ non termina e $f_x(1)$ è $\neq \perp$ come posso scoprirlo?
  - Uso allora il seguente trucco (ancora una volta di sapore diagonale):
    - simulo 1 passo di computazione di $f_x(0)$: se mi fermo, ho chiuso positivamente;
    - in caso contrario simulo un passo di computazione di $f_x(1)$;
    - se ancora non mi sono fermato simulo 2 passi del calcolo di $f_x(0)$; successivamente 1 passo di $f_x(2)$; 2 di $f_x(1)$; 3 di $f_x(0)$; e così via secondo lo schema già adottato di figura:



  - In questa maniera se $\exists\, z |\, f_x(z) \neq \perp$, prima o poi lo trovo, perché prima o poi simulerò abbastanza passi della computazione di $f_x(z)$ per farla arrestare.

46

- Chiudendo la parentesi:
  abbiamo dunque un notevole numero di problemi (tipicamente gli errori a run-time dei programmi) non decidibili ma *semi*decidibili.
- Attenzione però: qual'è esattamente il problema *semi*decidibile:
  - la *presenza* dell'errore (se c'è lo trovo)
  - *non* l'assenza!
- Ma il complemento di un problema in RE - R non è neanche RE (altrimenti sarebbe anche decidibile),
  ⇒ L'assenza di errori (ovvero la *correttezza* di un programma rispetto ad un errore) non solo non è decidibile, ma *non è neanche semidecidibile*!

- Otteniamo un modo sistematico per dimostrare che un problema *non decidibile* non è *neanche RE*:
  dimostrare che il suo complemento lo è.