Riepilogo (1)

1. Automi, grammatiche e altri formalismi possono essere considerati dispositivi meccanici per risolvere problemi matematici
   - I problemi matematici sono spesso la formalizzazione di problemi pratici
   - Esempio: il problema di decidere se una stringa appartiene a un linguaggio può essere visto come la formalizzazione matematica del riconoscimento di un'istruzione in un linguaggio di programmazione

Riepilogo (2)

2. Alcuni formalismi sono più potenti di altri
   – Le TM possono accettare linguaggi che non sono riconoscibili da NPDA
   – …
3. Nessun formalismo (tra quelli che abbiamo visto) è più potente delle TM
   – Alcuni formalismi sono potenti quanto le TM (es., automi a due pile)

Alcune domande (1)

- I formalismi introdotti sono adeguati per catturare l'essenza di un solutore meccanico?
  - Equazioni differenziali
  - Trovare un cammino in un grafo
  - …
- La capacità di un meccanismo di risolvere un problema dipende dal modo in cui il problema è formalizzato?

Alcune domande (2)

- Ci sono formalismi computazionali più potenti delle TM?
  - Un qualche (futuro) super computer?
- Una volta che un problema è stato formalizzato adeguatamente, possiamo sempre risolverlo mediante dispositivi meccanici?

…

- Abbiamo le risposte?

Teoria della computabilità

- Vogliamo mostrare che i nostri formalismi sono adeguati per ogni particolare problema da risolvere meccanicamente
  - Che cos'è un problema?
- Dobbiamo investigare sul potere delle TM (e formalismi equivalenti) rispetto a ogni altro dispositivo di calcolo, teorico o pratico
- Studiamo i limiti della risoluzione meccanica di problemi
- Possiamo risolvere meccanicamente tutti i problemi definibili?

Piccola parentesi storica

- L'analisi aristotelica dei modelli inferenziali (**sillogismi**) è stato l'unico paradigma logico per due millenni ed è alla base della logica proposizionale
- L'obiettivo di Aristotele era di sviluppare un metodo universale di ragionamento con il quale si potesse imparare tutto ciò che c'è da sapere sulla realtà
- Sia Leibniz sia Pascal (17º secolo) capirono che l'analisi sillogistica era realizzabile **meccanicamente**
  - Perfino con la tecnologia limitata dell'epoca!
- Leibniz sognava un **calculus ratiocinator**:
  - *Allora, non ci sarà più bisogno fra due filosofi di discussioni più lunghe di quelle tra due matematici, poiché basterà che essi prendano le loro penne, che si siedano al loro tavolo […] e che entrambi dicano: "**Calcoliamo**".*

I nostri formalismi

- Molti problemi possono essere formalizzati come riconoscimento di linguaggi o traduzione
  - $x \in L$?
  - $y = \tau(x)$?
- Più precisamente, ogni problema matematico può essere formalizzato in questi modi

Formalizzazione di un problema matematico

- I nostri formalismi sono adeguati per tutti i problemi con domini numerabili
- Gli elementi di tali domini possono essere messi in corrispondenza biunivoca con gli elementi di $\mathbb{N}$
- Il problema originale è ridotto al calcolo di una funzione f: $\mathbb{N} \rightarrow \mathbb{N}$

Esempio

Problema: trovare una soluzione al sistema di equazioni del tipo

$a_1x_1+a_2x_2=c_1$

$b_1x_1+b_2x_2=c_2$

dove a_1, a_2, b_1, b_2, c_1, c_2 sono interi.

Può essere visto come il calcolo di una funzione $f:\mathbb{Z}^6 \rightarrow \mathbb{Q}^2$ (numeri interi e razionali, rispettivamente), definita come

$$f(a_1, a_2, b_1, b_2, c_1, c_2)=<r_1, r_2>$$

dove r_1, r_2 sono i valori delle incognite che risolvono le equazioni.

Riconoscimento e traduzione

- Il riconoscimento e la traduzione sono due formulazioni di un problema che possono essere ridotte l'una all'altra
- Dalla traduzione al riconoscimento
  - Se ho una macchina che può risolvere tutti i problemi della forma $y = \tau(x)$ e voglio usarla per risolvere il problema $x \in L$?, è sufficiente definire

    $\tau(x)$ = se $x \in L$ allora 1 altrimenti 0
- Dal riconoscimento alla traduzione
  - Se ho una macchina che può risolvere tutti i problemi della forma $x \in L$?, posso definire il linguaggio $L_\tau = \{x\$y \mid y = \tau(x)\}$
  - Per una x fissata, posso enumerare tutte le possibili stringhe y sull'alfabeto di uscita e per ognuna di esse posso chiedere alla macchina se $x\$y \in L_\tau$
  - Prima o poi, se $\tau(x)$ è definita, troverò la stringa per la quale la macchina risponde positivamente
  - La procedura è "lunghetta"…
  - Ma qui non ci interessa la lunghezza della computazione

Note

- Tutti i formalismi esaminati sono discreti
  - Domini matematici numerabili definiti in modo finito

  ma questo è in accordo con la tecnologia digitale
- La classe dei problemi che possono essere risolti da una TM è indipendente dall'alfabeto scelto (sempre che ci siano almeno due simboli)

TM e linguaggi di programmazione

- Data una TM M si può costruire un programma in Java (o C o FORTRAN o…) che simula M
  - Assunzione: il computer esegue il programma con una quantità di memoria arbitrariamente grande
- Dato un qualunque programma Java (o…) si può costruire una TM che calcola la stessa funzione calcolata dal programma

→ le TM hanno lo stesso potere espressivo dei linguaggi di programmazione di alto livello

(tali linguaggi si dicono perciò *Turing completi*)

Tesi di Church (parte I)

Non c'è nessun formalismo per modellare il calcolo meccanico che sia più potente della TM (o formalismi equivalenti)

Non è un teorema (in teoria andrebbe verificato ogni volta che qualcuno inventa un nuovo modello computazionale)

computer quantistici…? che cosa cambia?

Algoritmi

- Il concetto di “algoritmo” è di fondamentale importanza nell’informatica
- Intuitivamente, con algoritmo intendiamo una procedura per risolvere problemi mediante un dispositivo di calcolo automatico

  … visto anche come un modo astratto per rappresentare programmi per calcolatore
- In sintesi, la nozione di algoritmo cattura la nozione di sequenza astratta di comandi per risolvere un problema in modo meccanico

Proprietà (informali) degli algoritmi (1)

1. La sequenza di istruzioni dev’essere finita
2. Qualunque istruzione dev’essere eseguibile mediante un processore meccanico per il calcolo
3. Il processore è dotato di memoria per immagazzinare risultati intermedi
4. La computazione è discreta
   - L’informazione è codificata digitalmente
   - La computazione procede attraverso passi discreti

Proprietà (informali) degli algoritmi (2)

5. Gli algoritmi sono eseguiti in modo deterministico
6. Non c'è limite sulla quantità di dati in ingresso e in uscita
7. Non c'è limite sulla quantità di memoria richiesta per effettuare una computazione
8. Non c'è limite sul numero di passi discreti richiesti per effettuare una computazione

Tesi di Church (parte II)

Ogni algoritmo può essere codificato mediante una TM (o formalismo equivalente)

Nessun algoritmo può risolvere problemi che non possono essere risolti da una TM: la TM è il più potente calcolatore che abbiamo e che avremo mai!

Domanda importante (e risposta parziale)

Quali sono i problemi che possiamo risolvere algoritmicamente (o, con un termine equivalente, “automaticamente”)?

Risposta:
Sono esattamente i problemi che possono essere risolti dalla semplice TM!

TM: problemi aperti (per ora)

- TM = dispositivo per risolvere un dato problema predefinito
  - Una TM può essere vista come un computer astratto, non programmabile, predisposto per un uso speciale
- Domande:
  - Le TM possono modellare computer programmabili?
  - Le TM calcolano tutte le funzioni da $\mathbb{N}$ a $\mathbb{N}$?

Enumerazione algoritmica

- Un insieme S può essere enumerato algoritmicamente (E) se possiamo trovare una biiezione tra S e $\mathbb{N}$
  - $E : S \leftrightarrow \mathbb{N}$
  - E può essere calcolato con un algoritmo (cioè una TM grazie alla tesi di Church)
- Esempio: enumerazione algoritmica di $\{a,b\}^*$

ε	a	b	aa	ab	ba	bb	aaa	aab	…
0	1	2	3	4	5	6	7	8	…

Fatto importante

Le TM possono essere enumerate algoritmicamente

- Dobbiamo mostrare l'algoritmo che lo fa
- Alcune ipotesi (senza perdita di generalità) :
  - TM a nastro singolo
  - Alfabeto fissato A (es., |A| = 3, A = {0, 1, _})

Enumerazione di TM (1)

- Esempio: TM con due stati
- Le possibili funzioni di transizione sono

	0	1	_
q_0	⊥	⊥	⊥
q_1	⊥	⊥	⊥

MT_0

	0	1	_
q_0	⊥	⊥	⊥
q_1	⊥	$<q_0, 0, S>$	⊥

MT_1

......

- NB: le TM differiscono anche per gli stati finali
  – Con 2 stati ci sono 2^2=4 possibili insiemi di stati finali

Enumerazione di TM (2)

- Quante funzioni di transizione per TM con 2 stati? $\delta: Q \times A \rightarrow Q \times A \times \{R,L,S\} \cup \{\perp\}$
- In generale: quante funzioni f: D → R?

$\longrightarrow |R|^{|D|}$ ($\forall x \in D$ abbiamo |R| scelte}

… quindi con |Q| = 2, |A| = 3, $(2 \cdot 3 \cdot 3+1)^{(2 \cdot 3)} = 19^6$ funzioni di transizione di TM con 2 stati

- Considerando le 2^2 scelte per gli stati finali, abbiamo (al più) $19^6 \cdot 2^2$ TM
- Ordiniamo queste TM: $\{M_0, M_1, \ldots M_{19^6 \cdot 2^2 - 1}\}$

Enumerazione di TM (3)

- Analogamente possiamo ordinare le $(3\cdot3\cdot3+1)^{(3\cdot3)}\cdot2^3$ TM con 3 stati e così via.
- Otteniamo un'enumerazione E: {TM} $\leftrightarrow \mathbb{N}$
- L'enumerazione E è algoritmica (o *effettiva*):
  - Possiamo sempre scrivere un programma in C (cioè una TM…) che, dato *n*, produce l'*n*-esima TM
  - e viceversa
- E(M) è detto numero di Gödel di M e E è una Gödelizzazione

Alcune convenzioni

- Poiché parliamo di numeri, le seguenti nozioni saranno considerate equivalenti:
  - “Problema”
  - “Calcolo di una funzione f: $\mathbb{N} \rightarrow \mathbb{N}$”
- Inoltre: f_y = “funzione calcolata dalla y-esima TM”

Prima risposta

Le TM modellano computer programmabili?

→ Sì!

- Consideriamo la Macchina di Turing Universale (UTM)

La UTM computa la funzione $g(y,x)=f_y(x)$

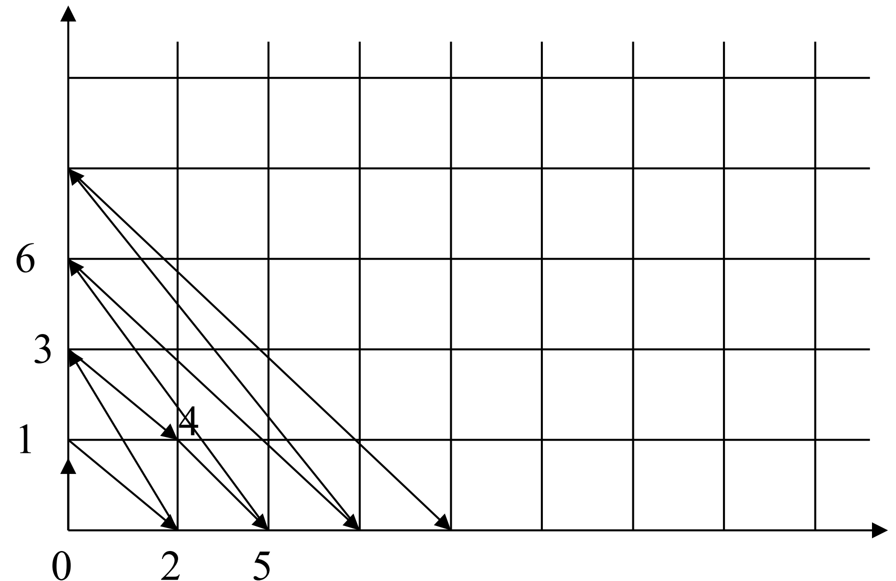
– $f_y(x)$ = funzione calcolata dalla y-esima TM sull'ingresso x

La UTM è una TM

La UTM non sembra appartenere alla famiglia $\{M_y\}$ perché f_y è una funzione di una variabile, mentre g è una funzione di due variabili

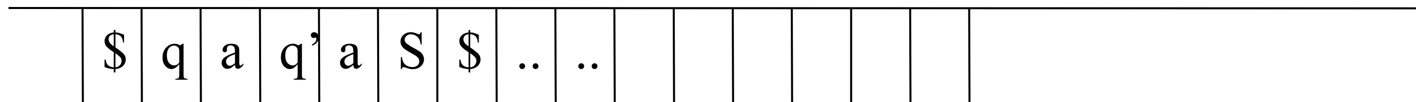
… ma sappiamo che $\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$

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



Come funziona la UTM? (1)

- Possiamo codificare $g(y,x)$ come una $g\hat{}(n) = g(d^{-1}(n))$, con $n=d(y,x)$, $<y,x>=d^{-1}(n)$
  - d e d^{-1} sono entrambe computabili
- Organizzazione di una UTM che calcola g^
  - Dato n calcola $d^{-1}(n) =<y,x>$
  - Poi costruisce la funzione di transizione di M_y (calcolando $E^{-1}(y)$) e la memorizza sul nastro della UTM:



Come funziona la UTM? (2)

- In un'altra porzione del nastro memorizziamo una codifica della configurazione di M_y

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

  - NB: I simboli speciali (#, $, stati, ...) sono usati come separatori e non appaiono altrove
- Alla fine, la UTM lascia sul nastro $f_y(x)$ se e solo se M_y termina la sua computazione su x

Note

- La TM è un modello molto astratto e semplice di calcolatore
- Continuiamo l'analogia:
  - TM: computer con un singolo programma cablato
    Una TM "ordinaria" esegue sempre lo stesso algoritmo, cioè calcola sempre la stessa funzione
  - UTM: computer con un programma salvato in memoria:
    y = programma
    x = ingresso del programma

Seconda risposta

Le TM computano **tutte** le funzioni da $\mathbb{N}$ a $\mathbb{N}$? Più precisamente, la UTM calcola tutte le funzioni da $\mathbb{N}$ a $\mathbb{N}$?

→ No, ci sono funzioni che non possono essere computate dalla UTM

- Cioè, per la tesi di Church: ci sono problemi che non possono essere risolti algoritmicamente
- Non è solo un'intuizione, si può dimostrare

Cardinalità di insiemi di funzioni

- Due domande interessanti:
  - Quante funzioni f: $\mathbb{N} \rightarrow \mathbb{N}$ ci sono?
  - Quali sono le funzioni f_y: $\mathbb{N} \rightarrow \mathbb{N}$?
- Cominciamo con "quante":

  $\{f: \mathbb{N} \rightarrow \mathbb{N}\} \supseteq \{f: \mathbb{N} \rightarrow \{0,1\}\} \Rightarrow$

$|\{f: \mathbb{N} \rightarrow \mathbb{N}\}| \geq |\{f: \mathbb{N} \rightarrow \{0,1\}\}| = |\wp(\mathbb{N})| = 2^{\aleph_0}$

$\aleph_0$= cardinalità dell'insieme $\mathbb{N}$ dei numeri naturali
  - si legge "alef con zero"
- $2^{\aleph_0}$ = cardinalità dell'insieme $\mathbb{R}$ dei numeri reali

Problemi e soluzioni

- Insieme di problemi: $|\{f: \mathbb{N} \rightarrow \mathbb{N}\}| \geq |\{f: \mathbb{N} \rightarrow \{0,1\}\}| = |\wp(\mathbb{N})| = 2^{\aleph_0}$
- L'insieme di funzioni calcolate da TM $\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}$ è per definizione numerabile
  - NB: E: $\{M_y\} \leftrightarrow \mathbb{N}$ induce $E^{\wedge}: \mathbb{N} \rightarrow \{f_y\}$ non uno a uno (in molti casi $f_y = f_z$, con $z \neq y$) ma basta per poter stabilire che $|\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}| = \aleph_0 < 2^{\aleph_0} = |\{f: \mathbb{N} \rightarrow \mathbb{N}\}|$

$\rightarrow$ la "maggior parte" dei problemi non può essere risolta algoritmicamente!

  - Ci sono molti più problemi che programmi!

Problemi definibili

- Per definire un problema ci serve una frase (stringa) di qualche linguaggio:
  - $f(x) = x^2$
  - $f(x) = \int_a^x g(z)dz$
  - "il numero che moltiplicato per se stesso è uguale a y"
  - …
- Ogni linguaggio è un sottoinsieme di A^*, che è un insieme numerabile

→ L'insieme dei problemi che si possono definire è quindi numerabile

Domanda: {Problemi risolvibili} = {Problemi definibili}?

- Certamente {Problemi risolvibili} ⊆ {Problemi definibili}

Esempio

- Il “problema dell’arresto” (*halting problem*)
  - Costruisco un programma
  - Gli do dei dati in ingresso
  - So che in generale il programma potrebbe non terminare la propria esecuzione (potrebbe “andare in loop”)

→ Posso determinare (in anticipo) se succederà?

- Questo problema può essere espresso in termini di TM:
  - Data una funzione

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

→ C’è una TM che calcola *g*?

Quali problemi possono essere risolti?

- Non esiste una TM che calcola g
  - Ecco perché un computer (che è un programma) non può avvertirci se un programma che abbiamo scritto andrà in un loop infinito su certi dati in ingresso
    - Mentre può facilmente indicarci se manca una “}”
- Alcuni esempi:
  - Determinare se un’espressione aritmetica è ben parentetizzata è un problema risolvibile (decidibile)
  - Determinare se un dato programma entrerà in un ciclo infinito su un dato ingresso è un problema algoritmicamente irrisolvibile (indecidibile)

Metodi di dimostrazione

- Dimostrazione per induzione
  - Questa tecnica può essere usata, ad esempio, per mostrare che un FSA è coerente con il linguaggio per il quale è stato definito
- Dimostrazione costruttiva
- Dimostrazione per assurdo

…. Ci sono altri approcci utili

Dimostrazioni per diagonalizzazione (1)

- Il ragionamento originale per diagonalizzazione fu usato da Georg Cantor nel 1891 per dimostrare che $\mathbb{R}$ ha una cardinalità più grande di $\mathbb{N}$
- E' usato per mostrare l'indecidibilità di alcuni "famosi" problemi
- La nozione di **dimostrazione per diagonalizzazione** si riferisce a uno schema comune

Dimostrazioni per diagonalizzazione (2)

Informalmente, listiamo tutte le funzioni calcolabili da $\mathbb{N}$ a $\mathbb{N}$ e i loro valori, disponendo ogni valore $f_y(x)$ in una tabella:

$$\begin{array}{cccccc} & 1 & 2 & 3 & 4 & \cdots \\ f_1 & f_1(1) & f_1(2) & f_1(3) & f_1(4) & \cdots \\ f_2 & f_2(1) & f_2(2) & f_2(3) & f_2(4) & \cdots \\ f_3 & f_3(1) & f_3(2) & f_3(3) & f_3(4) & \cdots \\ f_4 & f_4(1) & f_4(2) & f_4(3) & f_4(4) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

La diagonale dell'array è la lista di valori dati da $f_k(k)$, con k in $\mathbb{N}$.

Dimostrazioni per diagonalizzazione (3)

- Definiamo una funzione d:$\mathbb{N} \rightarrow \mathbb{N}$ che differisce dalla diagonale in ogni valore
- d è una funzione $\mathbb{N} \rightarrow \mathbb{N}$, quindi deve apparire nella lista
  - ma differisce anche da ogni membro della lista sulla diagonale
  - da cui l'assurdo

Informalmente

Nessuna TM può decidere se, data una generica TM M e un generico ingresso x, M si arresta con l'ingresso x

- Qui con "arresto" intendiamo "arresto in uno stato finale"
  - NB: Si può sempre costruire una TM che alla fine termina se e solo se raggiunge uno stato finale (emulazione)

Formalmente

Nessuna TM può calcolare la funzione totale g: $\mathbb{N} \times \mathbb{N} \to \{0,1\}$ definita come

g(y,x)= se $f_y(x) \neq \bot$ allora 1

altrimenti 0

- $f_y(x) \neq \bot$ significa che M_y si arresta in uno stato finale leggendo x cosicché $f_y(x)$ è definito

Dimostrazione (1)

- Si usa una tipica tecnica diagonale
- **Assumiamo** (per assurdo) che

$$g(y,x) = \text{se } f_y(x) \neq \bot \text{ allora } 1 \text{ altrimenti } 0$$

  sia computabile
- Da g, definiamo una funzione h tale che

$$h(x) = \text{se } g(x,x) = 0 \text{ allora } 1 \text{ altrimenti } \bot$$

  - $g(x,x) = 0$ corrisponde a $f_x(x) = \bot$
- Se g fosse computabile allora anche h sarebbe computabile

Dimostrazione (2)

- Notare che se h è computabile allora $h = f_i$ per qualche i.
- Calcoliamo h in i
  - Se $h(i) = f_i(i) = 1$ allora $g(i,i) = 0$,
    cioè $f_i(i)=\perp$ ASSURDO
  - Se $h(i) = f_i(i) =\perp$ allora $g(i,i) = 1$
    cioè, $f_i(i) \neq \perp$ ASSURDO

Un lemma importante (1)

- La funzione

$$h'(x) = \text{se } f_x(x) \neq \perp \text{ allora } 1 \text{ altrimenti } 0$$

  non è computabile

  – Non è un corollario, bensì un lemma dell'enunciato precedente (halting problem)

  – Questo risultato non deriva dal precedente

Un lemma importante (2)

- Notare che h'(x) è un caso speciale della funzione g(y,x)

  $$g(y,x) = \text{se } f_y(x) \neq \perp \text{ allora 1 altrimenti 0}$$

  perché h'(x)=g(x,x) e abbiamo appena mostrato che g non è computabile
  - La non calcolabilità di h'(x) non è una conseguenza immediata della non calcolabilità di g(y,x):
- Se un problema non è risolvibile, allora un suo caso speciale potrebbe essere risolvibile
  - per esempio, alcune proprietà che non possono essere decise per i linguaggi ricorsivamente enumerabili possono essere decise per i linguaggi regolari
- Invece una generalizzazione di un problema non risolvibile è necessariamente non risolvibile
- Se un problema è risolvibile, una sua generalizzazione potrebbe non essere risolvibile
  - Ma qualunque sua specializzazione è certamente risolvibile

Altro risultato importante (1)

- La funzione $k(y) = 1$ se f_y è totale, cioè

  $$k(y) = \text{se } (\forall x \in \mathbb{N}\ f_y(x) \neq \bot) \text{ allora } 1 \text{ altrimenti } 0$$

  non è computabile
- Questo problema presenta una quantificazione su tutti i possibili dati in ingresso
- In qualche caso potremmo essere in grado di stabilire, per uno specifico valore di x, se $f_y(x) \neq \bot$
  - anche se potessimo farlo per ogni singolo valore specifico di x non saremmo necessariamente in grado di rispondere alla domanda "f_y è una funzione totale?"
    - Se trovo x tale che $f_y(x) = \bot$ allora f_y non è totale, ma se non lo trovo?

Altro risultato importante (2)

- Da un punto di vista pratico, questo problema è forse ancora più importante del precedente
- Dato un (generico) programma, vogliamo sapere se terminerà l'esecuzione per *qualsiasi dato* in ingresso o se potrà, per *qualche dato*, andare in loop infinito
- Nel problema dell'arresto, invece, eravamo interessati a sapere, dato un (generico) programma e un (generico) ingresso, se il programma dato terminerà l'esecuzione con l'ingresso dato

Dimostrazione (1)

- Tecnica standard: diagonale + assurdo, più tecnicalità varie.
- Ipotesi: $k(y)$ = se ($\forall x \in \mathbb{N}\ f_y(x) \neq \perp$) allora 1 altrimenti 0
  è computabile e (per definizione) totale
- Allora definisco $g(x) = w$ = indice (numero di Gödel) della x-esima TM (in E) che calcola una funzione totale
- Se k è computabile e totale, allora lo è anche g:
  - computa k(0), k(1), …,
  - sia w_0 il primo valore tale che $k(w_0) = 1$, allora sia $g(0) = w_0$;
  - allora sia $g(1) = w_1$, dove w_1 è il secondo valore tale che $k(w_1) = 1$; …
  - la procedura è algoritmica; inoltre, poiché ci sono infinite funzioni totali, g(x) è certamente definita per ogni x, quindi è totale.

Dimostrazione (2)

- g è anche strettamente monotona: $w_{x+1} > w_x$;
  quindi anche g^{-1} è una funzione, strettamente monotona,
  … ma non totale: $g^{-1}(w)$ è definita solo se w è il numero di Gödel di una funzione totale
- Allora:
  1. Definiamo $h(x) = f_{g(x)}(x) + 1$
     $f_{g(x)}$ è computabile e totale, quindi anche h è computabile e totale
  2. Pertanto abbiamo $h = f_i$ per qualche i
     Poiché h è totale, $g^{-1}(i) \neq \perp$. Sia allora $g^{-1}(i) = j$ e cioè $g(j)=i$
- Qual è il valore di $h(j)$?
  - $h(j) = f_{g(j)}(j) + 1 = f_i(j) + 1$ (da (1))
  - $h = f_i \rightarrow h(j) = f_i(j)$ (da (2))

$\rightarrow$ Assurdo!

Risolvibilità e soluzioni (1)

- Osservazione: sapere che un problema è risolvibile non vuol dire che sappiamo come risolverlo (e quale sia la soluzione)
- In matematica abbiamo spesso dimostrazioni non costruttive: si mostra che un oggetto matematico esiste, ma non lo si calcola
- Nel nostro caso:
  - un problema è risolvibile se esiste una TM che lo risolve
  - per alcuni problemi si può concludere che *esiste* una TM che li risolve, senza però essere in grado di costruirla
- Caso banale: il "problema" consiste nel rispondere a una domanda booleana (una cosiddetta *domanda chiusa*):
  - E' vero che il numero di atomi nell'universo è $10^{10^{10^{10^{10}}}}$?
  - E' vero che una "partita perfetta a scacchi" finisce patta?
  - (prima del 1995...) E' vero che $\neg(\exists x,y,z,w \in \mathbb{N} \mid x^y+z^y=w^y \wedge y>2)$?
  - ….

Risolvibilità e soluzioni (2)

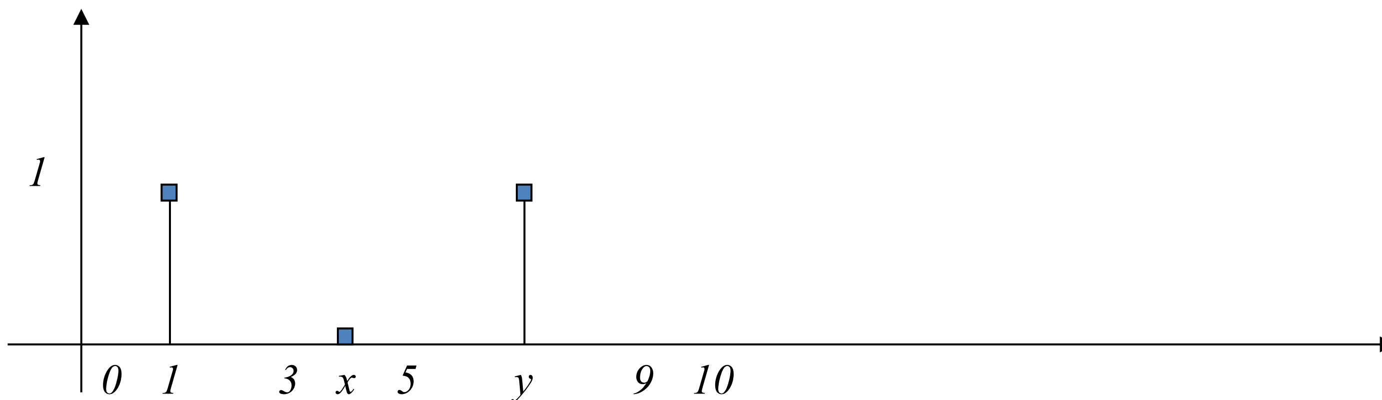
- Negli esempi precedenti, si sa *a priori* che la risposta è o Sì o No
  - eppure non si sa (o sapeva) quale fosse
- Problema = funzione; risolvere un problema = calcolare una funzione
  Quale funzione si può associare ai problemi precedenti?
  Se si codifica VERO=1; FALSO=0, *tutti* i problemi di cui sopra sono espressi da una delle due funzioni seguenti:
  - $f'(x) = 1, \forall x,$
  - $f''(x) = 0, \forall x$

  Entrambe le funzioni cono computabili
- Più in astratto:
  $g(10,20) = 1$ 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 $f_7(28) = \perp$
  ....
- Come problemi indipendenti, sono tutti risolvibili
- Tuttavia, non ne conosciamo necessariamente la soluzione

Esempi non banali (1)

- Consideriamo casi meno banali:
  - f(x) = x-esima cifra nell'espansione decimale di π
    f è computabile (conosciamo algoritmi (TM) per calcolarla)
  - Investighiamo sulla computabilità della seguente funzione
    g(x) = se in π $\exists$ **esattamente x cifre** "5" consecutive allora 1 altrimenti 0

    Calcolando la sequenza (π =3.14159…)
    {f(0) = 3, f(1) = 1, f(2) = 4, f(3) = 1, f(4) = 5, f(5) = 9, …}
    Otteniamo g(1) = 1
    In generale il grafico della funzione g sarà di questo tipo:



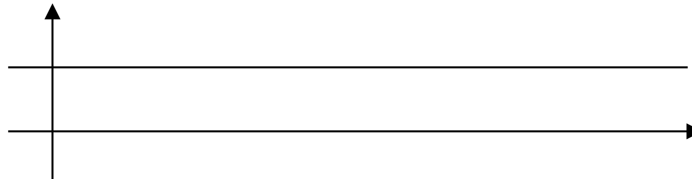
Esempi non banali (2)

- Per qualche valore di x potremmo trovare che g(x) = 1
- Inoltre, se g(x)=1, prima o poi lo scopriremo, se siamo pazienti…
- Ma potremmo concludere che, per esempio, $g(10^{10}) = 0$ se dopo aver calcolato $f(10^{100})$ non abbiamo (ancora) trovato 10^{10} cifre 5 consecutive?

- Possiamo escludere la congettura seguente?
  "Per ogni x, generando una sequenza sufficientemente lunga di π, prima o poi troveremo x 5 consecutivi"
  - Se questa congettura fosse vera allora seguirebbe che g è la funzione costante g(x) = 1 (per ogni x), e potremmo concludere che possiamo calcolare g.
  - In conclusione, *allo stato delle conoscenze attuali*, non possiamo concludere né che g sia computabile, né che non lo sia.

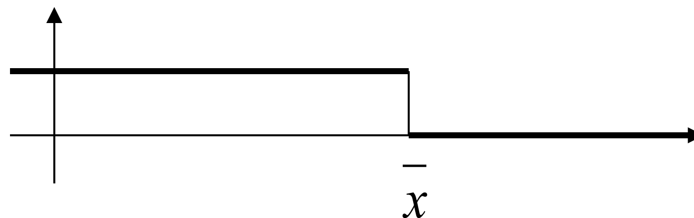
Esempi non banali (3)

- Consideriamo ora la seguente “leggera” modifica di g:
  h(x) =se in π $\exists$ **almeno** x 5 consecutivi allora 1 altrimenti 0
  - se g(x) = 1 allora anche h(x) = 1;
    Notare tuttavia che se, per qualche x, h(x) = 1, allora h(y)=1 $\forall$y≤ x.
  - Il grafico di h è come uno dei due seguenti:
  - 1)



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

  - 2)



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

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

Esempi non banali (4)

- Allora h è certamente nel seguente insieme di funzioni

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

  Ogni funzione in questo insieme è banalmente computabile
  - Per ogni $\bar{x}$ fissato è immediato costruire una TM che calcoli $h_{\bar{x}}$; idem per $\bar{h}$
- Perciò h è certamente computabile
  - esiste una TM che la calcola
- Possiamo calcolare h?
  (più precisamente: sappiamo qual è h tra le funzioni dell'insieme di cui sopra?)

  No (al momento): tra le infinite TM che calcolano le funzioni nell'insieme di cui sopra non sappiamo quale sia quella giusta!

Problema di decisione

- Un <u>problema di decisione</u> è una domanda che ha due possibili risposte: sì o no.
  La domanda riguarda un qualche ingresso
- Esempi:
  - Dato un grafo G e un insieme di vertici K, K è una cricca (*clique*)?
  - Dato un grafo G e un insieme di lati M, M è un albero ricoprente (*spanning tree*)?
  - Dato un insieme di assiomi, un insieme di regole e una formula, la formula è dimostrabile a partire da questi assiomi e regole?

Semidecidibilità

- Un problema è <u>semidecidibile</u> se c'è un algoritmo che dice sì se la risposta è sì
  - tuttavia può andare in loop se la risposta è no.
- Consideriamo ancora il problema dell'arresto
  - E' indecidibile,

    … ma è semidecidibile:

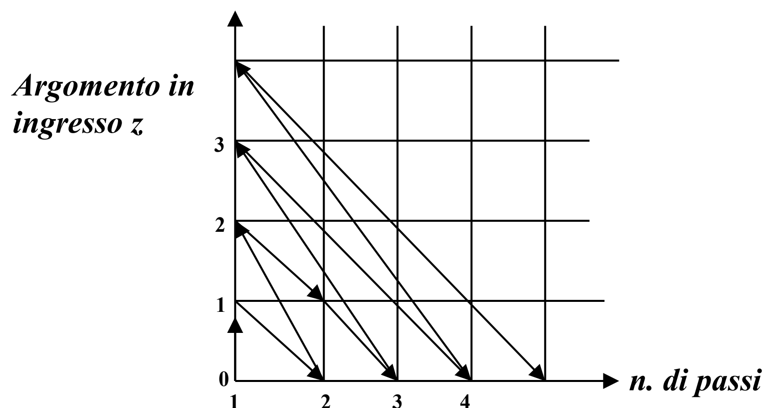
    se la TM si ferma, prima o poi ce ne accorgiamo …
- Come ottenere un risultato di semidecidibilità?

Esempio: una TM si arresta per qualche valore?

- Il problema di determinare se $\exists\, z \mid f_x(z) \neq \perp$ è semidecidibile
- Schema di dimostrazione
  - Se calcoliamo $f_x(0)$ e troviamo che $f_x(0) \neq \perp$ allora la risposta è sì;
  - Che succede se la computazione di $f_x(0)$ non termina e $f_x(1) \neq \perp$: come lo scopriamo?

Schema di dimostrazione

– Usiamo qui un trucco di tipo diagonale:
  - Simuliamo 1 passo di esecuzione di $f_x(0)$: se ci fermiamo, allora la risposta è sì;
  - Altrimenti simuliamo un passo di computazione di $f_x(1)$;
  - Ancora una volta, se non si ferma simuliamo 2 passi di computazione di $f_x(0)$; poi, 1 passo di $f_x(2)$; 2 passi di $f_x(1)$; 3 di $f_x(0)$; e così via, secondo lo schema della figura:



Conclusione

- C’è un gran numero di problemi indecidibili che sono semidecidibili
  - Tipico esempio: errori a runtime nei programmi
- Notiamo che il problema semidecidibile è:
  - La presenza dell’errore (se c’è un errore, prima o poi lo scopro)
  - Non la sua assenza!
- Importante implicazione sulla verifica basata sul testing
  - Famosa asserzione di Dijkstra: il testing può dimostrare la presenza di errori, non la loro assenza

Insiemi ricorsivi

- Concentriamoci sui problemi a risposta sia binaria: Problema = x appartiene all'insieme S? (dove $S \subseteq \mathbb{N}$)
  - Tutti i problemi binari possono essere (ri)formulati in questo modo
- Funzione caratteristica di un insieme S:

$$c_S(x) = \text{se } x \in S \text{ allora 1 altrimenti 0}$$

- Un insieme S è ricorsivo (o decidibile) se e solo se la sua funzione caratteristica è computabile
  - c_S è totale per definizione

Insiemi ricorsivamente enumerabili

- S è <u>ricorsivamente enumerabile</u> (RE) (o semidecidibile) se e solo se:
  - S è l'insieme vuoto, o
  - S è l'immagine di una funzione g_S totale e computabile (detta *generatrice*)

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

    $\Rightarrow$

    $S=\{g_S(0), g_S(1), g_S(2),\ldots\}$
- Il termine "ricorsivamente enumerabile" proviene da questa "enumerazione" e il termine "semidecidibile" può essere spiegato intuitivamente:
  - se $x\in S$ allora, enumerando gli elementi di S, prima o poi si trova x e si riceve una risposta corretta (sì) alla domanda; ma se $x \notin S$?

Teorema (½ + ½ = 1)

(A) Se S è ricorsivo, allora è anche RE (cioè, la decidibilità è più forte della semidecidibilità)

(B) S è ricorsivo se e solo se sia S sia il suo complemento S^ = $\mathbb{N}$ - S sono RE

- Due "semidecidibilità" fanno una "decidibilità"
- Qui, rispondere NO a un problema è esattamente difficile quanto rispondere SÌ al suo complemento

Corollario: la classe di insiemi decidibili (linguaggi, problemi, …) è chiusa rispetto al complemento

Dimostrazione (A)

Enunciato: Ricorsivo -> RE

- Se S è vuoto, è RE per definizione
- Se $S \neq \emptyset$, sia c_s la sua funzione caratteristica
  - poiché $S \neq \emptyset$, $\exists k \in S$, allora $\exists k\ c_S(k) = 1$

  Definiamo la funzione generatrice g_s come segue:

  $$g_S(x) = \text{se } c_S(x) = 1 \text{ allora } x \text{ altrimenti } k$$

  g_S è totale e computabile e $I_{g_S} = S$

  $\rightarrow$ S è RE
- Questa è una dimostrazione non costruttiva:
  - Non sappiamo se $S \neq \emptyset$
  - Non richiediamo un algoritmo per trovare un k specifico
  - Sappiamo solo che g_S esiste se $S \neq \emptyset$: questo ci basta!

Dimostrazione (B)

(B) è equivalente a:

(B.1) S ricorsivo → sia S sia S^ RE **e**
(B.2) sia S sia S^ RE → S ricorsivo

(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 $x \notin S$) computabile →

$c_{S^\wedge}(x)$ (= 0 se $x \in S$, $c_S(x) = 1$ se $x \notin S$) computabile →

S^ ricorsivo →

S^ RE

(B.2) S RE → costruisco l'enumerazione $S=\{g_S(0), g_S(1), g_S(2),\ldots\}$
S^ RE → costruisco $S^\wedge=\{g_{S^\wedge}(0), g_{S^\wedge}(1), g_{S^\wedge}(2),\ldots\}$
… ma $S \cup S^\wedge = \mathbb{N}$, $S \cap S^\wedge = \emptyset$
quindi $\forall x \in \mathbb{N}, \exists y | x=g_{S^\wedge}(y) \vee x=g_S(y) \wedge \neg(\exists z | x=g_{S^\wedge}(z) \wedge x=g_S(z))$

- x appartiene a una e una sola delle due enumerazioni →

Se si costruisce l'enumerazione

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

si può certamente trovare in essa qualunque x:

se x è in una posizione dispari, allora $x \in S$; se in una pari allora $x \in S^\wedge$.
Quindi c_S può essere computata

Osservazioni di interesse pratico

- Consideriamo l'insieme S con le seguenti caratteristiche:
  - $i \in S \rightarrow f_i$ totale (cioè S contiene *solo* indici di funzioni totali e computabili)
  - f totale e computabile $\rightarrow \exists\, i \in S \mid f_i = f$ (cioè S le contiene *tutte*)

  S è l'insieme di indici di funzioni totali e computabili

  allora S **non** è RE
- Dimostrabile per diagonalizzazione
- Questo significa che non esiste un formalismo RE (automi, grammatiche, funzioni ricorsive, …) che possa definire tutte le funzioni computabili e totali e solo quelle:
  - Gli FSA definiscono funzioni totali, ma non tutte;
  - Le TM definiscono tutte le funzioni computabili, ma anche quelle non totali
  - Analogamente, il linguaggio C permette di codificare qualunque algoritmo, ma anche quelli che non terminano
    - Non c'è nessun sottoinsieme del C che definisca esattamente tutti i programmi che terminano!
    - L'insieme di programmi C in cui i cicli **for** soddisfano certi vincoli che garantiscono la terminazione include solo programmi che terminano, ma non tutti i programmi che terminano!

Sbarazzarsi delle funzioni parziali…

- Estendiamo una funzione, ad esempio arricchendo $\mathbb{N}$ con un nuovo valore $\{\perp\}$, o semplicemente attribuendo a f un valore convenzionale quando f è indefinita
- Operazione matematicamente sensata (infatti in matematica si presta poca attenzione alle funzioni parziali)
- Il trucco non funziona a causa del risultato seguente:
  - Non esiste una funzione totale e computabile che sia un'estensione della funzione seguente (computabile ma non totale)

    $$g(x) = \text{se } f_x(x) \neq \perp \text{ allora } f_x(x) + 1, \text{ altrimenti } \perp$$

  - Ancora una volta, dimostrabile per diagonalizzazione
- Si può quindi prendere una funzione parziale e farla diventare totale, ma così facendo si può perdere la computabilità

Altro risultato utile

- Teorema
  - S è RE $\leftrightarrow$ S = D_h, con h computabile e parziale: S = {x| h(x) $\neq\perp$}
    e
  - S è RE $\leftrightarrow$ S = I_g, con g computabile e parziale: S = {x| x = g(y), y $\in \mathbb{N}$}
- Possiamo ora mostrare che esistono insiemi semidecidibili che non sono decidibili

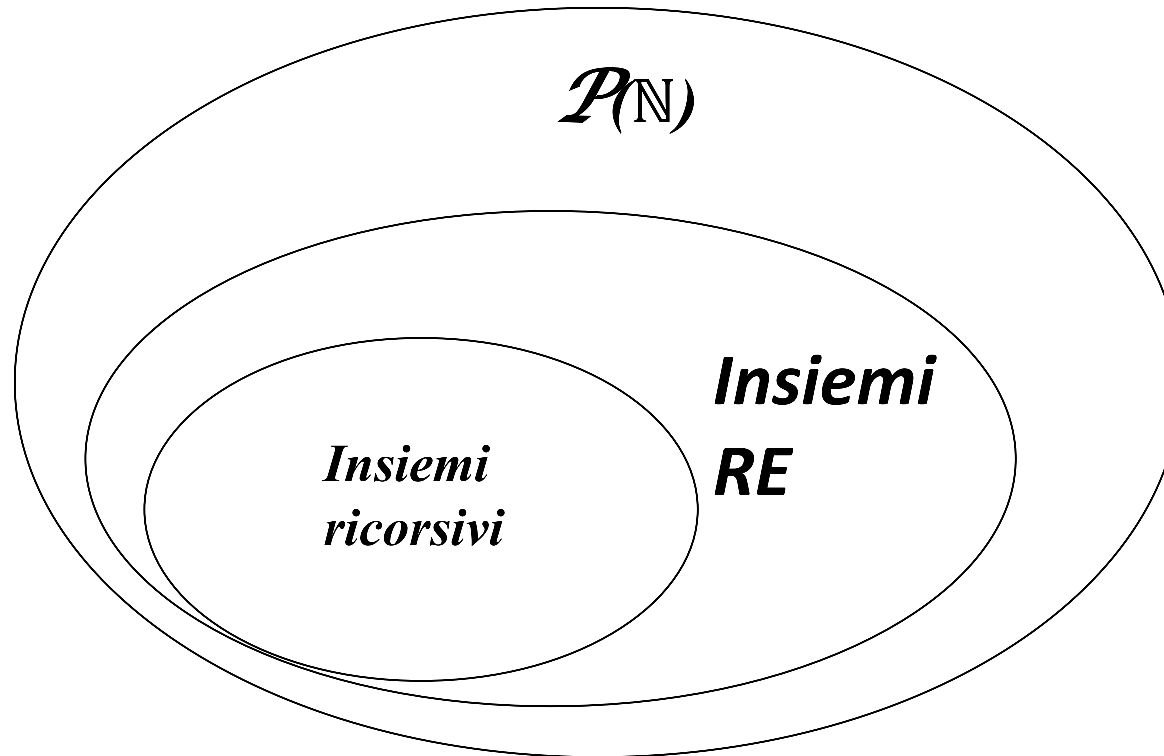
  K = {x| $f_x(x) \neq\perp$} è semidecidibile perché K = D_h con h(x) = $f_x(x)$. Sappiamo tuttavia che la funzione caratteristica di K,

  $c_K(x)$ = se $f_x(x) \neq\perp$ allora 1 altrimenti 0

  non è computabile (è la funzione h'(x) del lemma a pagina 45)

  $\rightarrow$ K non è decidibile

Conclusione



Notare che tutte le inclusioni sono strette

Gli insiemi RE (cioè i linguaggi riconosciuti dalle TM) *non* sono chiusi rispetto al complemento

Proprietà di chiusura

Famiglia di linguaggi	Formalismo	∪	∩	c	\	*	·
		Unione	Intersezione	Complemento	Differenza	Stella di Kleene	Concatenazione
Star-free	MFO	√	√	√	√	X	√
Regolari	FSA, NFA Espressioni regolari Grammatiche regolari MSO	√	√	√	√	√	√
Non contestuali deterministici	DPDA	X	X	√	X	X	X
Non contestuali	NPDA Grammatiche non contestuali	√	X	X	X	√	√
Dipendenti dal contesto	Grammatiche dip. dal contesto (Automa lineare limitato)	√	√	√	√	√	√
Ricorsivi	(Decider)	√	√	√	√	√	√
Ricorsivamente enumerabili	TM, NTM Grammatiche generali	√	√	X	X	√	√

Teorema del punto fisso di Kleene

- Sia t una funzione totale e computabile. Allora si può sempre trovare un intero p tale che $f_p = f_{t(p)}$
  - La funzione f_p è detta punto fisso di t

Teorema di Rice

Sia F un insieme di funzioni computabili
L'insieme S degli indici delle TM che calcolano le funzioni di F

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

è decidibile se e solo se

- o $F = \emptyset$
- o F è l'insieme di tutte le funzioni computabili

→ in tutti i casi non banali S non è decidibile!

Dimostrazione (1)

- Per assurdo, si supponga che
  - S è ricorsivo,
  - F≠ Ø e
  - F non è l'insieme di tutte le funzioni computabili
- Consideriamo la funzione caratteristica c_S di S
  - $c_S(x)$ = se $f_x \in F$ allora 1 altrimenti 0
- Per ipotesi, c_S è computabile, quindi enumerando ogni TM M_i, troviamo

  (1) il primo $i \in S$ tale che $f_i \in F$ e

  (2) il primo $j \notin S$ tale che $f_j \notin F$

Dimostrazione (2)

- Poiché c_S è computabile, allora lo è anche c'_S:

$$c'_S(x) = \text{ se } f_x \notin F \text{ allora } i \text{ altrimenti } j \qquad (3)$$

- Per il teorema di Kleene, c'è una x' tale che

$$f_{c'_S(x')} = f_{x'} \qquad (4)$$

- Ci sono due possibilità:
  - Supponiamo che $c'_S(x')=i$, allora per la (3) $f_{x'} \notin F$, ma per la (4) $f_{x'} = f_i$ e per la (1) $f_i \in F$: assurdo.
  - Supponiamo invece che $c'_S(x')=j$, allora per la (3) $f_{x'} \in F$, ma per la (4) $f_{x'}=f_j$ e per la (2) $f_j \notin F$: assurdo.

Dal teorema di Rice

- Il teorema di Rice ha forti implicazioni negative:
  - C'è una lista sconfinata di problemi interessanti la cui indecidibilità segue banalmente dal teorema di Rice
- Dato un qualunque insieme F={g}, per il teorema di Rice non è decidibile se una generica TM calcoli g o meno

In pratica

- Correttezza dei programmi
  - P risolve il problema specificato?
  - Cioè, M_x calcola la funzione che costituisce l'insieme {f}?
- Equivalenza tra programmi
  - M_x calcola la funzione che costituisce 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, …)?
- …

Riduzione di problemi (1)

- Un problema P’ è <u>ridotto</u> a un problema P se un algoritmo per risolvere P viene usato per risolvere P’, cioè:
  - P è risolvibile
  - C’è un algoritmo che, per ogni data istanza di P’:
    1. determina una corrispondente istanza di P
    2. costruisce algoritmicamente la soluzione dell’istanza di P’ dalla soluzione dell’istanza di P
- Esempio:
  - Supponiamo che il problema di cercare un elemento in un insieme sia risolvibile
  - Allora si può risolvere anche il problema di calcolare l’intersezione tra due insiemi

Riduzione di problemi (2)

- Formalmente:
  - Vogliamo verificare se $x \in S$
  - Siamo in grado di verificare se $y \in S'$
  - Se c'è una funzione computabile e totale t tale che

$$x \in S \leftrightarrow t(x) \in S'$$

  - allora possiamo rispondere alla domanda "$x \in S$?" in modo algoritmico

Riduzione nell'altro senso

- Questo procedimento funziona anche al contrario:
  - Vogliamo sapere se possiamo risolvere $x \in S$
  - Sappiamo che non si può risolvere $y \in S'$ (S' *non* è decidibile)
  - Se troviamo una funzione computabile e totale t tale che
    $$y \in S' \leftrightarrow t(y) \in S$$
  - allora possiamo concludere che $x \in S$ non è decidibile

Altro esempio di riduzione

- Abbiamo già implicitamente usato questo ragionamento varie volte:
  - Dall'indecidibilità del problema dell'arresto per le TM abbiamo derivato in generale l'indecidibilità del problema della terminazione di qualunque computazione al computer:
    - Consideriamo una TM M_y, un intero x, un programma P scritto in C, un file f in ingresso
    - Possiamo costruire un programma P scritto in C che simuli M_y e memorizzare x in un file f in ingresso
    - P termina la sua computazione su f se e solo se $g(y,x) \neq \perp$
    - Se potessimo decidere se P termina la computazione su f, allora potremmo anche risolvere il problema dell'arresto per la TM

Un meccanismo alquanto generale (1)

- E’ decidibile il problema di stabilire se “un generico programma P acceda a una variabile non inizializzata”?
  - Supponiamo (per assurdo) che sia decidibile
  - Allora possiamo considerare il problema dell’arresto e ridurlo al seguente nuovo problema:
    - Dato un generico programma P^ che riceve in ingresso il generico dato D, possiamo costruire un programma P come segue:
      ```
      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 a x che non è inizializzata perché x non compare in P^
    - Quindi l’accesso alla variabile non inizializzata x avviene in P se e solo se P^ termina.
- Se potessimo risolvere il problema dell’accesso a una variabile non inizializzata allora potremmo anche risolvere il problema della terminazione, il che è assurdo

Un meccanismo alquanto generale (2)

- La stessa tecnica può essere applicata per dimostrare l’indecidibilità di molte altre proprietà tipiche dell’esecuzione dei programmi:
  - Indici di array oltre i limiti
  - Divisione per 0
  - Compatibilità dinamica di tipi
  - …
  - Tipici errori a runtime