Informatica Teorica

Massimo Perego, Omar Masri

Contents

Introduzione

Si "contrappone" all'informatica applicata, ovvero qualsiasi applicazione dell'informatica atta a raggiungere uno scopo, dove l'informatica è solamente lo strumento per raggiungere in maniera efficace un obiettivo. Con "*informatica teorica*" l'oggetto è l'informatica stessa, si studiano i fondamenti della disciplina in modo rigoroso e scientifico. Può essere fatto ponendosi delle questioni fondamentali: il *cosa* e il *come* dell'informatica, ovvero cosa è in grado di fare l'informatica e come è in grado di farlo.

Cosa: L'informatica è "la disciplina che studia l'informazione e la sua elaborazione automatica", quindi l'oggetto sono l'informazione e i dispositivi di calcolo per gestirla; scienza dell'informazione. Diventa lo studio come risolvere automaticamente un problema. Ma tutti i problemi sono risolvibili in maniera automatica? Cosa è in grado di fare l'informatica?

La branca dell'informatica teorica che studia cosa è risolvibile si chiama **Teoria della Calcolabilità**, studia cosa è calcolabile per via automatica. Spoiler: non tutti i problemi sono risolvibili per via automatica, e non potranno mai esserlo per limiti dell'informatica stessa. Cerchiamo una caratterizzazione generale di cosa è calcolabile e cosa no, si vogliono fornire strumenti per capire ciò che è calcolabile. La caratterizzazione deve essere fatta matematicamente, in quanto il rigore e la tecnica matematica permettono di trarre conclusioni sull'informatica.

Come: Una volta individuati i problemi calcolabili, come possiamo calcolarli? Il dominio della **Teoria della Complessità** vuole descrivere le risoluzione dei problemi tramite mezzi automatici in termini di risorse computazionali necessarie. Una "risorsa computazionale" è qualsiasi cosa che viene consumata durante l'esecuzione per risolvere il problema, come possono essere elettricità o numero di processori, generalmente i parametri più importanti considerati sono tempo e spazio di memoria. Bisognerà definire in modo preciso cosa si intende con "tempo" e "spazio". Una volta fissati i parametri bisogna definire anche cosa si intende con "risolvere efficientemente" un problema, in termini di tempo e spazio. La teoria della calcolabilità dice quali problemi sono calcolabili, la teoria della complessità dice, all'interno dei problemi calcolabili, quali sono risolvibili efficientemente.

Capitolo 1

Teoria della Calcolabilità

1.1 Notazione

1.1.1 Funzioni

Funzione: Una funzione f dall'insieme A all'insieme B è una LEGGE che spiega come associare a ogni elemento di A un elemento di B. Si scrive

$$f : A \to B$$

chiamiamo A dominio e B codominio.

Per dire come agisce su un elemento si usa $f(a) = b$, b è l'immagine di a secondo f (di conseguenza a è la controimmagine). Per definizione di funzione, è possibile che elementi del codominio siano raggiungibili da più elementi del dominio, ma non il contrario. Possiamo classificare le funzioni in base a questa caratteristica:

- **Iniettiva:** $f : A \to B$ è iniettiva se e solo se $\forall a, b \in A,\ a \neq b \implies f(a) \neq f(b)$
- **Suriettiva:** $f : A \to B$ è suriettiva se e solo se $\forall b \in B,\ \exists a \in A : f(a) = b$: un altro modo per definirla è tramite l'insieme immagine di f, definito come

  $$\mathrm{Im}_f = \{b \in B : \exists a, f(a) = b\} = \{f(a) : a \in A\}$$

  Solitamente $\mathrm{Im}_f \subseteq B$, ma f è suriettiva se e solo se $\mathrm{Im}_f = B$;
- **Biettiva:** $f : A \to B$ è biettiva se e solo se è sia iniettiva che suriettiva, ovvero

  $$\begin{array}{ll} \forall a, b \in A, a \neq b : & f(a) \neq f(b) \\ \forall b \in B, \exists a \in A : & f(a) = b \end{array} \implies \forall b \in B, \exists! a \in A : f(a) = b$$

Inversa: Per le funzioni biettive si può naturalmente associare il concetto di "inversa": dato $f : A \to B$ biettiva, si definisce inversa la funzione $f^{-1} : B \to A$ tale che $f^{-1}(b) = a \Leftrightarrow f(a) = b$.

Composizione di funzioni: Date $f : A \to B$ e $g : B \to C$, f composto g è la funzione $g \circ f : A \to C$ definita come $g \circ f(a) = g(f(a))$. Generalmente non commutativo, $f \circ g \neq g \circ f$, ma è associativo $(f \circ g) \circ h = f \circ (g \circ h)$.

Funzione identità: Dato l'insieme A, la funzione identità su A è la funzione $i_A : A \to A$ tale che $i_A(a) = a$, $\forall a \in A$.

Un'altra possibile definizione per funzione inversa diventa:

$$f^{-1} \circ f = i_A \wedge f \circ f^{-1} = i_B$$

Funzioni Parziali: Se una funzione $f : A \to B$ è definita per $a \in A$ si indica con $f(a) \downarrow$ e da questo proviene la categorizzazione: una funzione è **totale** se definita $\forall a \in A$, **parziale** altrimenti (definita solo per qualche elemento di A).

$f(a) \downarrow$ significa che $\exists b$ tale che $f(a) = b$.

Insieme Dominio: Chiamiamo **dominio** (o campo di esistenza) di f l'insieme

$$\mathrm{Dom}_f = \{a \in A | f(a) \downarrow\} \subseteq A$$

Quindi se $\mathrm{Dom}_f = A$ la funzione è totale, se $\mathrm{Dom}_f \subsetneq A$ allora è una funzione parziale. L'insieme di valori per cui la funzione è definita.

Totalizzazione: Si può **totalizzare una funzione parziale** f definendo una funzione a tratti $\bar{f} : A \to B \cup \{\perp\}$ tale che

$$\bar{f}(a) = \begin{cases} f(a) & a \in \mathrm{Dom}_f(a) \\ \perp & \text{altrimenti} \end{cases}$$

Dove $\perp$ è il **simbolo di indefinito**, per tutti i valori per cui la funzione di partenza f non è definita. Da qui in poi $B_\perp$ significa $B \cup \{\perp\}$.

Insieme delle funzioni: L'insieme di tutte le funzioni che vanno da A a B si denota con

$$B^A = \{f : A \to B\}$$

La notazione viene usata in quanto la cardinalità di B^A è esattamente $|B|^{|A|}$, con A e B insiemi finiti.

Volendo includere anche tutte le funzioni parziali:

$$B_\perp^A = \{f : A \to B_\perp\}$$

Ovviamente $B^A \subset B_\perp^A$

1.1.2 Prodotto Cartesiano

Chiamiamo **prodotto cartesiano** l'insieme

$$A \times B = \{(a, b) | a \in A \wedge b \in B\}$$

Rappresenta l'insieme di tutte le coppie ordinate di valori in A e B. In generale non è commutativo, a meno che $A = B$. Può essere esteso a n-uple di valori:

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) | a_i \in A_i\}$$

Il prodotto di n volte lo stesso insieme verrà, per comodità, indicato come

$$A \times \cdots \times A = A^n$$

Proiettore: Operazione "opposta", il proiettore i-esimo è una funzione che estrae l'i-esimo elemento di una tupla, quindi è una funzione

$$\pi_i : A_1 \times \cdots \times A_n \to A_i \;\text{ t.c. }\; \pi_i(a_1, \ldots, a_n) = a_i$$

La proiezione sull'asse in cui sono presenti i valori dell'insieme a_i.

1.1.3 Funzione di Valutazione

Dati A, B e quindi $B_\perp^A$ si definisce **funzione di valutazione** la funzione:

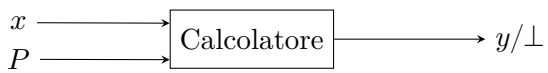
$$\omega : B_\perp^A \times A \to B \quad \text{t.c.} \quad \omega(f, a) = f(a)$$

Prende una funzione f e la valuta su un elemento a del dominio. Si possono fare due tipi di analisi su questa funzione:

- Fisso a e provo tutte le f, ottenendo un *benchmark* di tutte le funzioni su a
- Fisso f e provo tutte le a del dominio, ottenendo il *grafico* di f

1.2 Sistemi di Calcolo

Vogliamo modellare teoricamente un **sistema di calcolo**; quest'ultimo può essere visto come una black box che prende in input un programma P, dei dati x e calcola il risultato y di P su input x. La macchina restituisce y se è riuscita a calcolare un risultato, $\perp$ (indefinito) se è entrata in un loop infinito.



Quindi, formalmente, possiamo definire un sistema di calcolo come una funzione

$$\mathcal{C} : \text{PROG} \times \text{DATI} \to \text{DATI}_\perp$$

Possiamo vedere un sistema di calcolo come una funzione di valutazione:

- i dati x corrispondono all'input a
- il programma P corrisponde alla funzione f

Formalmente, un programma $P \in \text{PROG}$ è una sequenza di regole che trasformano un dato input in uno di output, ovvero l'espressione di una funzione secondo una sintassi

$$P : \text{DATI} \to \text{DATI}_\perp$$

e di conseguenza $P \in \text{DATI}_\perp^{\text{DATI}}$. In questo modo abbiamo mappato l'insieme PROG sull'insieme delle funzioni, il che ci permette di definire il sistema di calcolo come la funzione

$$\mathcal{C} : \text{DATI}_\perp^{\text{DATI}} \times \text{DATI} \to \text{DATI}_\perp$$

Analoga alla funzione di valutazione. Con $\mathcal{C}(P, x)$ indichiamo la funzione calcolata da P su x dal sistema di calcolo $\mathcal{C}$, che viene detta **semantica** di P, ovvero il suo "significato" su input x.

Il modello solitamente considerato quando si parla di calcolatori è quello di **Von Neumann**.

Un esempio: calcolo della semantica (significato) di un programma:

Programma:

P $\equiv$
```
input(x)
if ⟨x⟩_2 == 0
  return x
else
  while (1 < 2)
  return 1
```

Semantica:

$$\mathcal{C}(P, \_) : \mathbb{N} \to \mathbb{N}_\perp$$

$$\mathcal{C}(P, n) = \begin{cases} n & \text{se } n \text{ pari} \\ \perp & \text{altrimenti} \end{cases}$$

1.3 Potenza Computazionale

Indicando con

$$\mathcal{C}(P, \_) : \text{DATI} \to \text{DATI}_\perp$$

la funzione che viene calcolata dal programma P (semantica di P).

La **potenza computazionale** di un calcolatore è definita come l'insieme di **tutte le funzioni che quel sistema di calcolo è in grado di calcolare**, ovvero

$$F(\mathcal{C}) = \{\mathcal{C}(P, \_) | P \in \text{PROG}\} \subseteq \text{DATI}_\perp^{\text{DATI}}$$

Ovvero, l'insieme di tutte le possibili semantiche di funzioni calcolabili con il sistema $\mathcal{C}$. Stabilire il carattere di quest'ultima inclusione equivale a stabilire *cosa può fare l'informatica*:

- se $F(\mathcal{C}) \subsetneq \text{DATI}_\perp^{\text{DATI}}$ allora esistono compiti **non automatizzabili**
- se $F(\mathcal{C}) = \text{DATI}_\perp^{\text{DATI}}$ allora l'informatica *può fare tutto*

Calcolare funzioni vuol dire risolvere problemi *in generale*, a ogni problema è possibile associare una funzione soluzione programmando la quale posso risolvere automaticamente il problema

Un possibile approccio per risolvere l'inclusione è tramite la **cardinalità** (funzione che associa ogni insieme al numero di elementi che contiene) dei due insiemi. Potrebbe però presentare dei problemi: è efficace solo quando si parla di insiemi finiti. Ad esempio $|\mathbb{N}| = |\mathbb{R}| = \infty$.

Serve una diversa definizione di cardinalità che considera l'esistenza di infiniti *più densi di altri*.

1.4 Relazioni di Equivalenza

Dati due insiemi A, B, una *relazione binaria* R è un sottoinsieme $R \subseteq A \times B$ di coppie ordinate. Due elementi sono in relazione se e solo se $(a, b) \in R$. Indichiamo la relazione tra due elementi anche con la notazione infissa aRb.

Una classe importante di relazioni è quella delle **relazioni di equivalenza**: una relazione $R \subseteq A^2$ è una relazione di equivalenza se e solo se rispetta le proprietà di

- **Riflessività**: $\forall a \in A,\ (a, a) \in R$
- **Simmetria**: $\forall a, b \in A,\ (a, b) \in R \Leftrightarrow (b, a) \in R$
- **Transitività**: $\forall a, b, c \in A,\ (a, b) \in R \land (b, c) \in R \implies (a, c) \in R$

1.4.1 Partizione indotta dalla relazione di equivalenza

A ogni relazione di equivalenza $R \subseteq A^2$ si può associare una **partizione**, ovvero un insieme di sottoinsiemi $A_1, A_2, \ldots, A_i, \ldots \subseteq A$ tali che

- $A_i \neq \varnothing$
- Se $i \neq j$ allora $A_i \cap A_j = \varnothing$
- $\bigcup_{i \geqslant 1} A_i = A$

La relazione R definita su A^2 *induce* una partizione $\{A_1, A_2, \ldots\}$ su A.

1.4.2 Classi di equivalenza e Insieme quoziente

Dato un elemento $a \in A$, chiamiamo **classe di equivalenza** di a l'insieme

$$[a]_R = \{b \in A | (a, b) \in R\}$$

Ovvero, tutti gli elementi in relazione con a, chiamato **rappresentante** della classe.

Si può dimostrare facilmente che

- non esistono classi di equivalenza vuote, per riflessività
- dati $a, b \in A$, allora $[a]_R \cap [b]_R = \varnothing$, oppure $[a]_R = [b]_R$, i due elementi o sono in relazione o non lo sono
- $\bigcup_{a \in A} [a]_R = A$

L'insieme delle classi di equivalenza, per definizione, è una partizione indotta da R su A, detta **insieme quoziente** di A rispetto ad R, denotato con A/R.

1.5 Cardinalità

1.5.1 Isomorfismi

Due insiemi A e B sono **isomorfi** (*equi-numerosi*) se esiste una biezione tra essi, denotato come $A \sim B$.

Chiamando $\mathcal{U}$ l'insieme di tutti gli insiemi, la relazione $\sim$ è $\sim \subseteq \mathcal{U}^2$ (tecnicamente $\mathcal{U}$ non è un insieme per ZFC ma una classe propria).

Dimostriamo che $\sim$ è una relazione di equivalenza:

- Riflessività: $A \sim A$, la biezione è data dalla funzione identità i_A
- Simmetria: $A \sim B \Leftrightarrow B \sim A$, la biezione è data dalla funzione inversa
- Transitività: $A \sim B \wedge B \sim C \implies A \sim C$, la biezione è data dalla composizione delle funzioni usate per $A \sim B$ e $B \sim C$

Dato che $\sim$ è una relazione di equivalenza, permette di partizionare l'insieme $\mathcal{U}$, risultando in classi di equivalenza contenenti insiemi isomorfi, ovvero con la stessa cardinalità. Possiamo quindi definire la **cardinalità** come l'insieme quoziente di $\mathcal{U}$ rispetto alla relazione $\sim$.

Questo approccio permette il *confronto delle cardinalità di insiemi infiniti*, basta trovare una funzione biettiva tra i due insiemi per poter affermare che sono isomorfi.

1.5.2 Cardinalità finita

La prima classe di cardinalità è quella delle cardinalità finite. Definiamo la seguente famiglia di insiemi:

$$J_n = \begin{cases} \varnothing & \text{se } n = 0 \\ \{1, \dots, n\} & \text{se } n > 0 \end{cases}$$

Un insieme A ha **cardinalità finita** se e solo se $A \sim J_n$ per qualche $n \in \mathbb{N}$; in tal caso possiamo scrivere $|A| = n$. La classe di equivalenza $[J_n]_\sim$ identifica tutti gli insiemi di $\mathcal{U}$ contenenti n elementi.

1.5.3 Cardinalità infinita

L'altra classe di cardinalità è quella delle **cardinalità infinite**, ovvero gli insiemi non in relazione con J_n. Si possono dividere in **numerabili** e **non numerabili**.

Insiemi numerabili

Un insieme A è numerabile se e solo se $A \sim \mathbb{N}$, ovvero $A \in [\mathbb{N}]_\sim$. Vengono anche detti **listabili**, in quanto è possibile elencare tutti gli elementi dell'insieme A tramite una funzione f biettiva tra $\mathbb{N}$ e A;

grazie ad f possiamo elencare gli elementi di A, formando l'insieme

$$A = \{f(0), f(1), \dots\}$$

Ed è esaustivo, in quanto elenca tutti gli elementi di A. Questi insiemi hanno cardinalità $\aleph_0$ (*aleph*).

Insiemi non numerabili

Gli insiemi non numerabili sono insiemi a cardinalità infinita ma non listabili, sono "*più fitti*" di $\mathbb{N}$; ogni lista generata non può essere esaustiva. Il più noto tra gli insiemi non numerabili è l'insieme $\mathbb{R}$ dei numeri reali.

Teorema 1.5.1. *L'insieme* $\mathbb{R}$ *non è numerabile* ($\mathbb{R} \not\sim \mathbb{N}$)

Proof. Suddividiamo la dimostrazione in 3 punti:

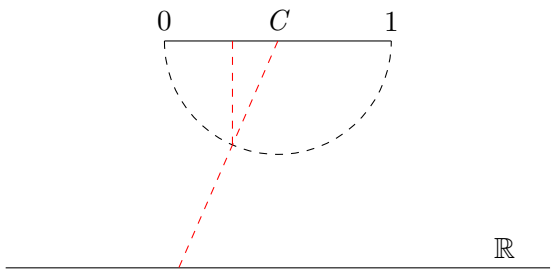
1. dimostriamo che $\mathbb{R} \sim (0, 1)$
2. dimostriamo che $\mathbb{N} \not\sim (0, 1)$
3. $\mathbb{N} \not\sim (0, 1) \sim \mathbb{R} \implies \mathbb{N} \not\sim \mathbb{R}$ (per transitività di $\sim$)

Per dimostrare che $\mathbb{R} \sim (0, 1)$ serve trovare una biezione tra $\mathbb{R}$ e $(0, 1)$.

Usiamo una rappresentazione grafica:

- disegnare una semicirconferenza di raggio 1/2, centrata in 1/2, quindi con diametro 1
- disegnare la perpendicolare al punto da mappare che interseca la circonferenza
- disegnare la semiretta passante per il centro C e l'intersezione precedente

L'intersezione tra asse reale (parallela al diametro) e semiretta finale è il punto mappato.



Questo approccio permette di dire che $\mathbb{R}$ è isomorfo a qualsiasi segmento di lunghezza maggiore di 0. La stessa biezione vale anche sull'intervallo chiuso $[0, 1]$ (e di conseguenza qualsiasi intervallo chiuso), usando la "compattificazione" $\mathbb{R} = \mathbb{R} \cup \{\pm\infty\}$ e mappando 0 su $-\infty$ e 1 su $+\infty$.

Continuiamo dimostrando che $\mathbb{N} \not\sim (0, 1)$: serve dimostrare che l'intervallo $(0, 1)$ non è listabile, quindi che ogni lista manca di almeno un elemento. Proviamo a "costruire" un elemento che andrà a mancare.

Per assurdo, sia $\mathbb{N} \sim (0, 1)$, allora possiamo listare gli elementi di $(0, 1)$ come

$$\begin{array}{llllll} 0. & a_{00} & a_{01} & a_{02} & \dots \\ 0. & a_{10} & a_{11} & a_{12} & \dots \\ 0. & a_{20} & a_{21} & a_{22} & \dots \\ & & \vdots & & \end{array}$$

dove con a_{ij} indichiamo la cifra di posto j dell'i-esimo elemento della lista.

Costruiamo il numero $c = 0.c_0c_1\ldots$ tale che

$$c_i = \begin{cases} 2 & \text{se } a_{ii} \neq 2 \\ 3 & \text{se } a_{ii} = 2 \end{cases}$$

Viene costruito "guardando" le cifre sulla diagonale principale, apparterrà sicuramente a $(0,1)$ ma differirà per almeno una posizione (quella sulla diagonale principale) da ogni numero presente all'interno della lista. Questo è assurdo sotto l'assunzione che $(0,1)$ è numerabile, quindi abbiamo provato che $\mathbb{N} \not\sim (0,1)$.

Il terzo punto $\mathbb{R} \not\sim \mathbb{N}$ si dimostra per transitività. Più in generale, non si riesce a listare nessun segmento di lunghezza maggiore di 0.

□

Questa dimostrazione (punto 2 in particolare) è detta **dimostrazione per diagonalizzazione**. L'insieme $\mathbb{R}$ viene detto **insieme continuo** e tutti gli insiemi isomorfi a $\mathbb{R}$ si dicono continui a loro volta.

1.5.4 Insieme delle Parti

L'**insieme delle parti** di $\mathbb{N}$ (anche detto *power set*), è definito come

$$P(\mathbb{N}) = 2^{\mathbb{N}} = \{S \mid S \text{ è sottoinsieme di } \mathbb{N}\}$$

Teorema 1.5.2. $P(\mathbb{N}) \not\sim \mathbb{N}$.

Proof. Possiamo dimostrare questo teorema tramite diagonalizzazione. Il vettore caratteristico di un sottoinsieme è un vettore che nella posizione p_i ha 1 se $i \in A$, 0 altrimenti (tipo vettore di incidenza).

Rappresentiamo $A \subseteq \mathbb{N}$ sfruttando il suo vettore caratteristico

$$\begin{array}{lcccccccc} \mathbb{N}: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\ A: & 0 & 1 & 1 & 0 & 1 & 1 & 0 & \ldots \end{array}$$

Supponiamo, per assurdo, che $P(\mathbb{N})$ sia numerabile. Vista questa proprietà, possiamo listare tutti i vettori caratteristiche che appartengono a $P(\mathbb{N})$ come

$$\begin{array}{lcccc} b_0 & = & b_{00} & b_{01} & b_{02} & \ldots \\ b_1 & = & b_{10} & b_{11} & b_{12} & \ldots \\ b_2 & = & b_{20} & b_{21} & b_{22} & \ldots \end{array}$$

Vogliamo quindi costruire un vettore che appartiene a $P(\mathbb{N})$ ma non presente nella lista precedente. Definiamo

$$c = \bar{b_{00}}\,\bar{b_{11}}\,\bar{b_{22}}\ldots$$

ovvero il vettore che contiene in posizione c_i il complemento di b_{ii}.

Questo vettore appartiene a $P(\mathbb{N})$, in quanto sicuramente sottoinsieme di $\mathbb{N}$, ma non è presente nella lista precedente perché diverso da ogni elemento almeno di una cifra (quella sulla diagonale principale).

Questo è assurdo per l'assunzione che $P(\mathbb{N})$ è numerabile, quindi $P(\mathbb{N}) \not\sim \mathbb{N}$.

□

Insieme delle funzioni

L'**insieme delle funzioni** da $\mathbb{N}$ a $\mathbb{N}$ è definito come

$$\mathbb{N}_{\perp}^{\mathbb{N}} = \{f : \mathbb{N} \to \mathbb{N}_{\perp}\}$$

Teorema 1.5.3. $\mathbb{N}_{\perp}^{\mathbb{N}} \not\sim \mathbb{N}$.

Proof. Diagonalizzazione strikes again. Assumiamo, per assurdo, che $\mathbb{N}_{\perp}^{\mathbb{N}}$ sia numerabile. Possiamo quindi listare $\mathbb{N}_{\perp}^{\mathbb{N}}$ come $\{f_0, f_1, f_2, \dots\}$

	0	1	2	3	...	$\mathbb{N}$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	...	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	...	...
...	...	...	...	...	...	...

Costruiamo una funzione $\varphi : \mathbb{N} \to \mathbb{N}_{\perp}$ per dimostrare l'assurdo. Un'idea potrebbe essere $\varphi(n) = f_n(n) + 1$, "spostando" la diagonale, ma non tiene in considerazione il caso $f_n(n) = \perp$ in quanto non sapremmo dare un valore a $\varphi(n) = \perp + 1$.

Definiamo quindi

$$\varphi(n) = \begin{cases} 1 & \text{se } f_n(n) = \perp \\ f_n(n) + 1 & \text{se } f_n(n) \downarrow \end{cases}$$

(la stessa cosa, ma gestendo il caso di $f_n(n) = \perp$).

Questa funzione appartiene a $\mathbb{N}_{\perp}^{\mathbb{N}}$, ma non è presente nella lista precedente, infatti $\forall k \in \mathbb{N}$ si ottiene

$$\varphi(k) = \begin{cases} 1 \neq f_k(k) = \perp & \text{se } f_k(k) = \perp \\ f_k(k) + 1 \neq f_k(k) & \text{se } f_k(k) \downarrow \end{cases}$$

Questo è assurdo sotto l'assunzione che $\mathbb{N}_{\perp}^{\mathbb{N}}$ è numerabile, quindi $\mathbb{N}_{\perp}^{\mathbb{N}} \not\sim \mathbb{N}$.

□

1.6 Potenza Computazionale di un sistema di calcolo

1.6.1 Validità dell'inclusione $F(\mathcal{C}) \subseteq \mathbf{DATI}_{\perp}^{\mathbf{DATI}}$

Dopo aver dato una più robusta definizione di cardinalità, possiamo studiare la natura dell'inclusione

$$F(\mathcal{C}) \subseteq \mathrm{DATI}_{\perp}^{\mathrm{DATI}}$$

Due intuizioni, da dimostrare, sono:

- PROG $\sim \mathbb{N}$: ogni programma può essere identificato con un numero, come la sua codifica in binario
- DATI $\sim \mathbb{N}$: anche ogni dato può essere identificato tramite la sua codifica in binario

Da questo possiamo dire che

$$F(\mathcal{C}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_{\perp}^{\mathbb{N}} \sim \text{DATI}_{\perp}^{\text{DATI}}$$

(non banale, assumiamo che $F(\mathcal{C})$ sia infinito numerabile).

Questo dimostra che **esistono funzioni non calcolabili**, ci sono troppe funzioni e troppi pochi programmi.

Dobbiamo dimostrare le due assunzioni PROG $\sim \mathbb{N}$ e DATI $\sim \mathbb{N}$. Si può fare tramite tecniche di aritmetizzazione (o godelizzazione) di strutture, tecniche che rappresentano delle strutture tramite un numero.

1.7 DATI $\sim \mathbb{N}$

Serve trovare una legge che

1. Associ biunivocamente dati a numeri e viceversa
2. Consenta di operare direttamente sui numeri per operare sui corrispondenti dati, ovvero abbia delle primitive che permettano di lavorare sul numero che "riflettano" il risultato sul dato, senza passare dal dato stesso
3. Consenta di dire, senza perdita di generalità, che i programmi lavorano su numeri

1.7.1 Funzione Coppia di Cantor

La **funzione coppia di Cantor** è la funzione

$$\langle \_, \_ \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^{+}$$

E sfrutta le due "sotto-funzioni"

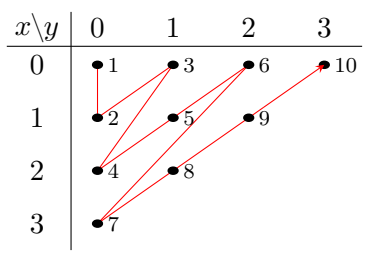
$$\begin{aligned} \text{sin} &: \mathbb{N}^{+} \to \mathbb{N} \\ \text{des} &: \mathbb{N}^{+} \to \mathbb{N} \end{aligned}$$

Tali che

$$\langle x, y \rangle = n \implies \begin{aligned} \text{sin}(n) &= x \\ \text{des}(n) &= y \end{aligned}$$

Si può rappresentare graficamente come

$x \backslash y$	0	1	2	3	...
0	1	3	6	10	...
1	2	5	9	...	
2	4	8	...		
3	7	...			



Il valore $\langle x, y \rangle$ rappresenta l'incrocio tra la x-esima riga e la y-esima colonna. Per costruirla:

1. $x = 0$
2. si parte dalla cella $(x, 0)$ e si enumerano le celle della diagonale identificata da $(x, 0)$ e $(0, x)$
3. si incrementa x di 1 e si ripete dal punto precedente

La funzione e' ovviamente:

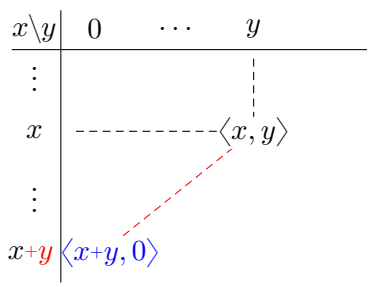
- iniettiva: non ci possono essere celle con lo stesso numero
- suriettiva: ogni numero in $\mathbb{N}^{+}$ deve comparire

Entrambe le proprietà sono soddisfatte, in quanto la numerazione avviene in maniera incrementale, quindi ogni numero prima o poi compare in una cella e di conseguenza ho una coppia che lo genera.

Forma analitica

Per la definizione di $\langle x, y\rangle$ si può notare che

$$\langle x, y\rangle = \langle x + y, 0\rangle + y$$



Intuitivamente, a partire da $\langle x + y, 0\rangle$ mi basta "salire" seguendo la diagonale fino a $\langle x, y\rangle$, ovvero y posti, e per definizione della funzione, y valori più in alto.

Il calcolo della funzione coppia si può quindi ridurre al calcolo di $\langle x + y, 0\rangle$. Chiamando $x + y = z$, si può notare come ogni cella

$$\langle z, 0\rangle = z + \langle z - 1, 0\rangle$$

E di conseguenza

$$\begin{aligned}\langle z, 0\rangle &= z + \langle z - 1, 0\rangle \\ &= z + (z - 1) + \langle z - 2, 0\rangle \\ &= z + (z - 1) + \cdots + 1 + \langle 0, 0\rangle = \\ &= \left(\sum_{i=1}^{z} i\right) + 1 = \frac{z(z + 1)}{2} + 1\end{aligned}$$

Mettendo insieme le due proprietà viste possiamo ottenere la formula analitica per la funzione coppia:

$$\langle x, y\rangle = \langle x + y, 0\rangle + y = \frac{(x + y)(x + y + 1)}{2} + y + 1$$

Forma analitica di sin e des

Vogliamo fare la stessa cosa per sin e des, in modo da poter computare l'inversa della funzione coppia, dato n.

Grazie alle osservazioni precedenti sappiamo che

$$\begin{aligned}\gamma = x + y \quad &\Longrightarrow \quad x = \gamma + y \\ n = y + \langle \gamma, 0\rangle \quad &\Longrightarrow \quad y = n - \langle \gamma, 0\rangle\end{aligned}$$

Trovando il valore di γ possiamo trovare x e y. Notiamo come γ sia il più grande valore che, quando calcolato sulla prima colonna ($\langle \gamma, 0\rangle$) non supera n, ovvero

$$\gamma = \max\{z \in \mathbb{N} \mid \langle z, 0\rangle \leqslant n\}$$

Intuitivamente, si tratta dell'inizio della diagonale che contiene n, è "l'inverso" dell'osservazione fatta in precedenza per la quale $\langle x, y\rangle = \langle x + y, 0\rangle + y$.

Risolviamo quindi la disequazione

$$\begin{aligned}\langle z, 0\rangle \leqslant n &\implies \frac{z(z+1)}{2} + 1 \leqslant n \\ &\implies z^2 + z - 2n + 2 \leqslant 0 \\ &\implies z_{1,2} = \frac{-1 \pm \sqrt{1 + 8n - 8}}{2} \\ &\implies \frac{-1 - \sqrt{8n-7}}{2} \leqslant z \leqslant \frac{-1 + \sqrt{8n-7}}{2}\end{aligned}$$

Come valore di γ scegliamo il massimo nell'intervallo precedente, ovvero:

$$\gamma = \left\lfloor \frac{-1 + \sqrt{8n-7}}{2} \right\rfloor$$

E con γ noto possiamo definire le funzioni sin e des come

$$\begin{aligned}\mathrm{des}(n) &= y = n - \langle \gamma, 0\rangle = n - \tfrac{\gamma(\gamma+1)}{2} - 1 \\ \sin(n) &= x = \gamma - y\end{aligned}$$

Teorema 1.7.1. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

Proof. La funzione di Cantor è una funzione biettiva tra l'insieme $\mathbb{N} \times \mathbb{N}$ e l'insieme $\mathbb{N}^+$, quindi i due insiemi sono isomorfi.

□

Possiamo estendere il risultato all'interno dell'insieme $\mathbb{N}$, ovvero:

Teorema 1.7.2. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$

Proof. Definiamo la funzione

$$[\_, \_] : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

tale che

$$[x, y] = \langle x, y\rangle - 1$$

Questa funzione è anch'essa biettiva, quindi i due insiemi sono isomorfi.

□

Grazie a questo è possibile dimostrare anche che $\mathbb{Q} \sim \mathbb{N}$, infatti i numeri razionali si possono rappresentare come coppie (num, den) e, in generale, tutte le tuple finite sono isomorfe a $\mathbb{N}$, basta iterare in qualche modo la funzione coppia di Cantor.

1.7.2 Applicazione alle strutture dati

I risultati ottenuti fin'ora rendono intuibile come ogni dato possa essere trasformato in un numero, soggetto a trasformazioni matematiche. La dimostrazione *formale* non verrà fatta, anche se verranno fatti esempi di alcune strutture dati che possono essere trasformate in un numero tramite la funzione coppia di Cantor. Ogni struttura dati può essere manipolata e trasformata in una coppia (x, y).

Le **liste** sono le strutture dati più utilizzate nei programmi. In generale non ne è nota la grandezza, di conseguenza è necessario trovare un modo, soprattutto durante l'applicazione di sin e des, per capire quando sono finiti gli elementi della lista.

Estendiamo la funzione coppia a una lista di interi $x_1, \ldots, x_n$:

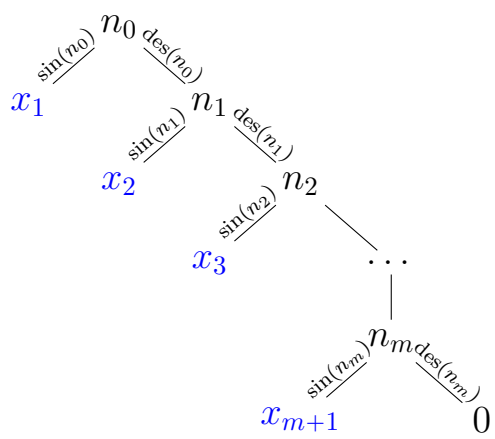
$$\langle x_1, \ldots, x_n\rangle \to \langle x_1, \langle x_2 \langle \ldots \langle x_n, 0\rangle \ldots \rangle\rangle$$

Lo 0 rappresenta il fine lista e non è necessario nel caso in cui il numero di elementi è noto (array).

La decodifica è il processo inverso, partendo dal numero finale si applicano le funzioni sin e des ottenendo a ogni iterazione:

- da des la somma parziale, su cui riapplicare la funzione per ottenere il valore successivo
- da sin il valore presente all'interno della lista

Termina quando il risultato di des è zero, ovvero l'elemento di fine lista che abbiamo inserito (x_n nel caso di array).



Se è presente uno 0 all'interno della lista non è un problema in quanto lo 0 compare come risultato di des solo a fine lista.

Quindi è possibile codificare liste e, di conseguenza, **qualsiasi tipo di dato**, basta convertirlo in una lista di numeri.

← (Qui guarda la lista (ci sono anche degli esercizi qui nelle slide guarda 5:7 in poi))

Ad esempio:

- una matrice può essere vista come array di array
- un grafo può essere rappresentato tramite la sua matrice di adiacenza
- i testi sono liste di caratteri
- i suoni si possono campionare per ottenere una lista di valori
- le immagini sono una "matrice" di pixel, ognuno dei quali ha un colore come valore

Abbiamo visto come i dati possano essere sostituiti da delle codifiche numeriche; di conseguenza possiamo sostituire tutte le funzioni

$$f : \mathrm{DATI} \to \mathrm{DATI}_\perp \quad \text{con funzioni} \quad f' : \mathbb{N} \to \mathbb{N}_\perp$$

In altre parole, l'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentabile da $\mathbb{N}_\perp^{\mathbb{N}}$.

1.8 PROG $\sim \mathbb{N}$

Adesso lavoriamo sulla parte della relazione che afferma

$$F(\mathcal{C}) \sim \mathrm{PROG} \sim \mathbb{N}$$

Da notare che la prima parte è più complicata di quanto non sembri, in verità una funzione può avere più programmi associati ma entrambi gli insiemi sono infiniti numerabili, quindi in principio esiste una biezione.

Vogliamo affermare che la potenza computazionale (l'insieme dei programmi che un sistema di calcolo $\mathcal{C}$ riesce a calcolare, $F(\mathcal{C})$) è isomorfa all'insieme di tutti i programmi, a loro volta isomorfi a $\mathbb{N}$.

Vogliamo arrivare a ricavare un numero dato un programma e viceversa. Per farlo servirà vedere l'insieme PROG come l'insieme dei programmi scritti in un certo linguaggio di programmazione.

I sistemi analizzati saranno:

- sistema di calcolo RAM
- sistema di calcolo WHILE

Il sistema RAM può apparentemente sembrare "troppo semplice", quindi il sistema WHILE verrà usato per avere un confronto tra le potenze computazionali. *Un sistema più sofisticato porta a poter risolvere più problemi?*

Ci sono due possibili soluzioni:

- $F(\text{RAM}) \neq F(\text{WHILE})$: la computabilità *dipende dal sistema usato*
- $F(\text{RAM}) = F(\text{WHILE})$: la computabilità è *intrinseca nei problemi* e, di conseguenza, tutti i sistemi sono equivalenti (Tesi di Church-Turing)

Il secondo caso è più promettente e, in quel caso, l'obiettivo diventerebbe trovare una *caratterizzazione teorica*, ovvero un "confine" per i problemi calcolabili.

1.8.1 Sistema di calcolo RAM

Il sistema di calcolo RAM è un sistema semplice che permette di definire rigorosamente:

- $\text{PROG} \sim \mathbb{N}$
- la **semantica** dei programmi eseguibili, ovvero $\mathcal{C}(P, \_)$, con $\mathcal{C} = \text{RAM}$, ottenendo $\text{RAM}(P, \_)$
- la **potenza computazionale**, ovvero calcolare $F(\mathcal{C})$ con $\mathcal{C} = \text{RAM}$, ottenendo $F(\text{RAM})$

Struttura

Una macchina RAM è formata da un processore e da una memoria teoricamente infinita, divisa in **celle/registri** contenenti numeri naturali (dati aritmetizzati).

Indichiamo i **registri** con R_k, con $k \geqslant 0$. Tra questi

- R_0 contiene l'output
- R_1 contiene l'input

Inoltre è presente un registro L, anche detto **program counter** PC che indica l'indirizzo dell'istruzione successiva.

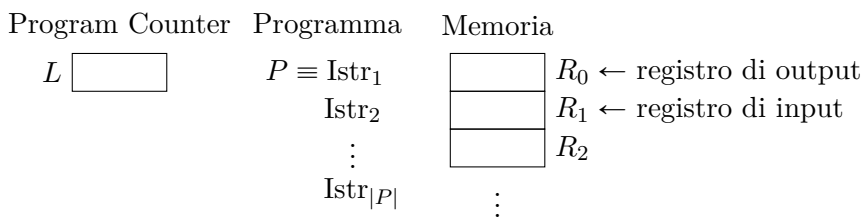
Un **programma** P è un insieme ordinato di istruzioni, indichiamo con $|P|$ il numero di istruzioni che il programma contiene.

Le **istruzioni** nel linguaggio RAM sono:

- **incremento**: $R_k \leftarrow R_k + 1$
- **decremento**: $R_k \leftarrow R_k \dot{-} 1$
- **salto condizionato**: `if` $R_k = 0$ `then goto` m, con $m \in \{1, \ldots, |P|\}$

L'istruzione di decremento è tale che

$$x \dot{-} y = \begin{cases} x - y & \text{se } x \geqslant y \\ 0 & \text{altrimenti} \end{cases}$$



Esecuzione di un programma RAM

L'esecuzione di un programma su una macchina RAM segue i passi:

1. **Inizializzazione**:
   - viene caricato il programma $P \equiv \text{Istr}_1, \dots \text{Istr}_n$ in memoria
   - il PC viene posto a 1 per indicare di eseguire la prima istruzione del programma
   - viene caricato l'input in R_1
   - ogni altro registro è azzerato
2. **Esecuzione**: le istruzioni vengono eseguite una dopo l'altra, a ogni iterazione passa da L a $L + 1$ (escluse operazioni di salto). Essendo il linguaggio RAM *non strutturato* richiede un PC per sapere l'operazione da eseguire al passo successivo.
3. **Terminazione**: per convenzione, si usa $L = 0$ per indicare che l'esecuzione del programma è terminata o andata in loop. Nel caso in cui il programma termini, è detto **segnale di halt** e arresta la macchina
4. **Output**: il contenuto di R_0, in caso di halt, contiene il risultato dell'esecuzione del programma P. Si indica con $\varphi_P(n)$ il contenuto del registro R_0 in caso di halt, oppure $\perp$ in caso di loop

$$\varphi_P(n) = \begin{cases} cont(R_0) & \text{se halt} \\ \perp & \text{se loop} \end{cases}$$

Con $\varphi_P : \mathbb{N} \to \mathbb{N}_\perp$ indichiamo la semantica del programma P. Con $\mathcal{C}(P, \_)$ indicavamo la semantica di P nel sistema di calcolo $\mathcal{C}$, quindi con $\text{RAM}(P, \_) = \varphi_P$ indichiamo la semantica di P nel sistema di calcolo RAM.

Semantica Operazionale

Per dare una definizione formale della semantica di un programma RAM va specificato il significato di ogni istruzione (**semantica operazionale**), esplicitando l'effetto che quell'istruzione ha sui registri della macchina.

Ogni istruzione fa passare la macchina da uno stato all'altro e la **semantica operazionale** di un'istruzione è la **coppia** formata dagli **stati** della macchina **prima e dopo l'istruzione**.

$$\text{STATO}_1 \to \boxed{\text{Istr}_i} \to \text{STATO}_2$$

La **semantica operazionale** di un'istruzione Istr_i è quindi una **relazione sui possibili stati**:

$$\text{Istr}_i \subseteq \text{Stati} \times \text{Stati}$$

Questa può essere vista come funzione parziale o relazione.

Uno stato deve descrivere completamente la situazione della macchina in un certo istante. Il programma rimane uguale, quindi l'informazione da salvare è la situazione globale dei registri R_k e il registro L.

La **computazione** del programma P è una sequenza di stati $\mathcal{S}_i$, ognuno generato dall'esecuzione di un'istruzione del programma; P induce una sequenza di stati $\mathcal{S}_i$, se questa è formata da un numero infinito di stati, allora il programma è andato in loop; in caso contrario, nel registro R_0 si trova il risultato y della computazione di P.

$$\varphi_P : \mathbb{N} \to \mathbb{N}_\perp \text{ t.c. } \varphi_P(n) = \begin{cases} y & \text{se } \exists \mathcal{S}_{fin} \\ \perp & \text{altrimenti} \end{cases}$$

Quindi, se il programma finisce, la semantica di quest'ultimo è il risultato del programma stesso, ovvero il valore posto nel registro R_0 al termine della computazione, $\perp$ altrimenti.

Per definire come passare da uno stato all'altro, definiamo formalmente:

- **Stato**: istantanea di tutte le componenti della macchina, è una funzione

  $$\mathcal{S} : \{L, R_i\} \to \mathbb{N}$$

  tale che $\mathcal{S}(R_k)$ restituisce il contenuto del registro R_k quando la macchina si trova nello stato $\mathcal{S}$. Gli stati possibili di una macchina appartengono all'insieme

  $$\text{STATI} = \{f : \{L, R_i\} \to \mathbb{N}\} = \mathbb{N}^{\{L, R_i\}}$$

  Questa rappresentazione permette un numero di registri potenzialmente infinito; se così non fosse, avremmo tuple per indicare tutti i possibili registri al posto dell'insieme $\{L, R_i\}$

- **Stato Finale**: uno stato finale $\mathcal{S}_{fin}$ è un qualsiasi stato $\mathcal{S}$ tale che $\mathcal{S}(L) = 0$

- **Dati**: già dimostrato come DATI $\sim \mathbb{N}$

- **Inizializzazione**: serve una funzione che, preso l'input, restituisca lo stato iniziale della macchina:

  $$\text{in} : \mathbb{N} \to \text{STATI} \text{ t.c. } \text{in}(n) = \mathcal{S}_{init}$$

  Lo stato iniziale $\mathcal{S}_{init}$ è tale che

  $$\mathcal{S}_{init}(R) = \begin{cases} 1 & \text{se } R = L \\ n & \text{se } R = R_1 \\ 0 & \text{altrimenti} \end{cases}$$

  Lo stato iniziale è quindi composto da 1 nel PC, l'input n in R_1, 0 in tutti gli altri registri

- **Programmi**: PROG è definito come l'insieme dei programmi RAM

Manca da definire la *parte dinamica* del programma, ovvero l'esecuzione. Per farlo, definiamo la **funzione di stato prossimo**:

$$\delta : \text{STATI} \times \text{PROG} \to \text{STATI}_\perp$$

tale che:

$$\delta(\mathcal{S}, P) = \mathcal{S}'$$

dove $\mathcal{S}$ rappresenta lo stato attuale e $\mathcal{S}'$ rappresenta lo stato prossimo dopo l'esecuzione di un'istruzione di P.

La funzione $\delta(\mathcal{S}, P) = \mathcal{S}'$ è tale che

- se $\mathcal{S}(L) = 0$ ho halt, ovvero deve terminare la computazione. Poniamo lo stato come indefinito, ovvero $\mathcal{S}' = \perp$

- Se $\mathcal{S}(L) > |P|$ vuol dire che P non contiene istruzioni che bloccano esplicitamente l'esecuzione del programma. Lo stato $\mathcal{S}'$ è tale che

$$\mathcal{S}'(R) = \begin{cases} 0 & \text{se } R = L \\ \mathcal{S}(R_i) & \text{se } R = R_i, \forall i \end{cases}$$

- Se $1 \leqslant \mathcal{S}(L) \leqslant |P|$ considero l'istruzione $\mathcal{S}(L)$-esima:
  - se ho un incremento/decremento sul registro R_k definisco $\mathcal{S}'$ tale che

$$\begin{cases} \mathcal{S}'(L) & = \mathcal{S}(L) + 1 \\ \mathcal{S}'(R_k) & = \mathcal{S}(R_k) \pm 1 \\ \mathcal{S}'(R_i) & = \mathcal{S}(R_i) \text{ per } i \neq k \end{cases}$$

  - Se ho `IF` $R_k = 0$ `THEN GOTO` m, definisco $\mathcal{S}'$ tale che

$$\mathcal{S}'(L) = \begin{cases} m & \text{se } \mathcal{S}(R_k) = 0 \\ \mathcal{S}(L) + 1 & \text{altrimenti} \end{cases}$$

$$\mathcal{S}'(R_i) = \mathcal{S}(R_i), \ \forall i$$

L'esecuzione di un programma $P \in$ PROG su input $n \in \mathbb{N}$ genera una sequenza di stati

$$\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_i, \mathcal{S}_{i+1}, \ldots$$

tali che:

$$\begin{array}{lcl} \mathcal{S}_0 & = & \text{in}(n) \\ \forall i \quad \mathcal{S}_{i+1} & = & \delta(\mathcal{S}_i, P) \end{array}$$

La sequenza è infinita quando P va in loop, mentre se termina raggiunge uno stato $\mathcal{S}_m$ tale che $\mathcal{S}_m(L) = 0$, ovvero ha ricevuto il segnale di halt.

La semantica di P è

$$\varphi_P(n) = \begin{cases} y & \text{se } P \text{ termina in } \mathcal{S}_m, \text{ con } \mathcal{S}_m(L) = 0 \text{ e } \mathcal{S}_m(R_0) = y \\ \perp & \text{se P va in loop} \end{cases}$$

La potenza computazionale del sistema RAM è

$$F(\text{RAM}) = \left\{ f \in \mathbb{N}_\perp^{\mathbb{N}} \mid \exists P \in \text{PROG s.t. } \varphi_P = f \right\} = \{\varphi_P \mid P \in \text{PROG}\} \subsetneq \mathbb{N}_\perp^{\mathbb{N}}$$

L'insieme è formato da tutte le funzioni $f : \mathbb{N} \to \mathbb{N}_\perp$ che hanno un programma che le calcola in un sistema RAM.

← (Guarda slide per esempi esercizi 6:10 in poi)

1.8.2 Aritmetizzazione di un programma

Per verificare che PROG $\sim \mathbb{N}$ basterebbe trovare una funzione che permetta di codificare i programmi in numeri in modo biunivoco. Data una lista di istruzioni semplici $P \equiv \text{Istr}_1, \ldots, \text{Istr}_m$ se questa fosse codificata come una lista di interi potremmo sfruttare la funzione coppia di Cantor per ottenere un numero associato al programma P. Quindi vogliamo trovare una funzione Ar che associ a ogni istruzione I_k la sua codifica numerica c_k. Se la funzione trovata è anche biunivoca siamo sicuri di poter trovare anche la sua inversa, ovvero la funzione che ci permette di ricavare I_k da c_k.

Riassumendo, vogliamo trasformare la lista di istruzioni un una lista di numeri su cui successivamente applicare la funzione coppia di Cantor. Vorremmo anche ottenere la lista di istruzioni originale data la codifica.

$$
\begin{array}{ccccccc}
P \equiv & \mathrm{Istr}_1, & \mathrm{Istr}_2, & \ldots, & \mathrm{Istr}_m \\
\downarrow & \updownarrow & \updownarrow & & \updownarrow \\
 & \boxed{Ar} & \boxed{Ar} & \cdots & \boxed{Ar} \\
 & \updownarrow & \updownarrow & & \updownarrow \\
 & c_1, & c_2, & \ldots & , c_m \\
 & \updownarrow & \updownarrow & & \updownarrow \\
 & \multicolumn{5}{c}{\boxed{\langle c_1, c_2, \ldots, c_m \rangle}} \\
 & & & \updownarrow & \\
cod(P) & = & & n &
\end{array}
$$

L'associazione biunivoca di un numero a una struttura si dice aritmetizzazione o Gödelizzazione.

Applicazione ai programmi RAM

Dovendo codificare tre istruzioni nel linguaggio RAM, definiamo la funzione Ar tale che

$$
Ar(I) = \begin{cases} 3k & \text{se } I \equiv R_k \leftarrow R_k + 1 \\ 3k+1 & \text{se } I \equiv R_k \leftarrow R_k \dot{-} 1 \\ 3\langle k, m \rangle - 1 & \text{se } I \equiv \texttt{ if } R_k = 0 \texttt{ then goto } m \end{cases}
$$

Per l'inversa, in base al modulo tra n e 3 ottengo una certa istruzione:

$$
Ar^{-1}(n) = \begin{cases} R_{\frac{n}{3}} \leftarrow R_{\frac{n}{3}} + 1 & \text{se } n \mod 3 = 0 \\ R_{\frac{n-1}{3}} \leftarrow R_{\frac{n-1}{3}} \dot{-} 1 & \text{se } n \mod 3 = 1 \\ \texttt{if } R_{\sin\left(\frac{n+1}{3}\right)} = 0 \texttt{ then goto des}\left(\frac{n+1}{3}\right) & \text{se } n \mod 3 = 2 \end{cases}
$$

Per tornare indietro devo prima invertire la funzione coppia di Cantor e poi invertire la funzione Ar. La lunghezza del programma P, indicata con $|P|$, si calcola come $len(cod(P))$. Abbiamo quindi dimostrato che PROG $\sim \mathbb{N}$.

← (Guarda le prime slide fino a 7:5 per esempi)

Osservazioni

Avendo $n = cod(P)$ si può scrivere

$$\varphi_P(t) = \varphi_n(t)$$

Ovvero, la semantica di P è uguale alla semantica della sua codifica. I numeri diventano un *linguaggio di programmazione*.

Si può scrivere l'insieme

$$F(\mathrm{RAM}) = \{\varphi_P : P \in \mathrm{PROG}\}$$

come

$$F(\mathrm{RAM}) = \{\varphi_i\}_{i \in \mathbb{N}}$$

L'insieme, grazie alla dimostrazione di PROG $\sim \mathbb{N}$, è numerabile. Abbiamo dimostrato rigorosamente che:

$$F(\mathrm{RAM}) \sim \mathbb{N} \nsim \mathbb{N}_{\perp}^{\mathbb{N}}$$

Di conseguenza, anche nel sistema di calcolo RAM esistono funzioni non calcolabili.

La RAM è troppo elementare affinché $F(\mathrm{RAM})$ rappresenti formalmente la "classe dei problemi risolubili automaticamente", quindi considerando un sistema di calcolo $\mathcal{C}$ più sofisticato, ma comunque trattabile rigorosamente come il sistema RAM, potremmo dare un'idea formale di "ciò che è calcolabile automaticamente".

Riuscendo a dimostrare che $F(\mathrm{RAM}) = F(\mathcal{C})$ allora cambiare la tecnologia non cambia ciò che è calcolabile, la calcolabilità è intrinseca ai problemi e la si può caratterizzare matematicamente.

1.8.3 Sistema di calcolo WHILE

Introduciamo quindi il sistema di calcolo WHILE per vedere se riusciamo a "catturare" più o meno funzioni calcolabili dalla macchina RAM.

Struttura

La macchina WHILE ha anch'essa, come la macchina RAM, una serie di registri, ma al posto di essere *potenzialmente infiniti* sono esattamente 21.

Il registro R_0 è il **registro di output**, mentre R_1 è il **registro di input**. Non esiste il Program Counter in quanto il linguaggio è **strutturato** e ogni istruzione in questo linguaggio va eseguita in ordine.

Il linguaggio WHILE prevede una **definizione induttiva**: vengono definiti alcuni comandi base e i comandi più complessi sono una concatenazione dei comandi base.

Assegnamento: Comando di base, ne esistono di tre tipi:

$$x_k := 0,$$
$$x_k := x_j + 1$$
$$x_k := x_j \dot{-} 1$$

Queste istruzioni sono più complete rispetto alle istruzioni RAM, in una sola istruzione si può azzerare il valore di una variabile o assegnare a una variabile il valore di un'altra aumentato/diminuito di 1.

While: Primo comando "induttivo". Si tratta di un comando della forma:

$$\texttt{while } x_k \neq 0 \texttt{ do } C$$

Dove C è detto **corpo** e può essere un assegnamento, un comando while o un comando composto.

Composto: Altro comando induttivo. Si tratta di un comando nella forma

$$\texttt{begin } C_1; \ldots; C_n \texttt{ end}$$

Dove i vari C_i sono, come prima, assegnamenti, comandi while o comandi composti.

Programma WHILE: Un programma WHILE è un comando composto, e l'insieme di tutti i programmi WHILE è l'insieme

$$W\text{-PROG} = \{\text{PROG scritti in linguaggio WHILE}\}$$

← (Guarda le slide ci sono degli esempi)

Chiamiamo

$$\Psi_W : \mathbb{N} \to \mathbb{N}_\perp$$

la **semantica** del programma $W \in W - \text{PROG}$.

Per dimostrare una proprietà P di un programma $W \in W$-PROG, data la definizione induttiva del linguaggio WHILE, è naturale procedere induttivamente:

1. dimostro P vera sugli assegnamenti
2. suppongo P vera sul comando C e la dimostro vera per $\texttt{while } x_k \neq 0 \texttt{ do } C$
3. suppongo P vera sui comandi $C_1, \ldots, C_n$ e la dimostro vera per $\texttt{begin } C_1; \ldots; C_n \texttt{ end}$

Esecuzione di un programma WHILE

L'esecuzione di un programma WHILE W è composta dalle seguenti fasi:

1. **Inizializzazione**: ogni registro x_i viene posto a 0, tranne x_1, che contiene l'input n
2. **Esecuzione**: essendo WHILE un linguaggio con strutture di controllo, non serve un Program Counter, poiché le istruzioni di W vengono eseguite l'una dopo l'altra
3. **Terminazione**: l'esecuzione di W può
   - *arrestarsi*: se arriva al termine delle istruzioni
   - *non arrestarsi*: se entra in un loop
4. **Output**: Se il programma va in halt, l'output è contenuto nel registro x_0. Possiamo scrivere

$$\Psi_W(n) = \begin{cases} cont(x_0) & \text{se halt} \\ \perp & \text{se loop} \end{cases}$$

   La semantica del programma, ancora una volta, coincide con il valore di output, questa volta contenuto all'interno di x_0, $\perp$ se non termina

Definizione formale per l'esecuzione

Come per i programmi RAM, serve una definizione formale per la semantica di un programma WHILE, per la quale servono una serie di elementi

- **Stato**: una tupla grande quanto il numero di variabili, dove quindi $\underline{x} = (c_0, \ldots, c_{20})$ rappresenta uno stato, con c_i rappresentante il contenuto della variabile i
- W**-STATI**: Insieme di tutti gli stati possibili, contenuto in $\mathbb{N}^{21}$ vista la definizione degli stati
- **Dati**: già visto che DATI $\sim \mathbb{N}$
- **Inizializzazione**: Lo stato iniziale è descritto dalla funzione

$$w\text{-}in(n) = (0, n, 0, \ldots, 0)$$

- **Semantica operazionale**: Vogliamo trovare una funzione che, presi comando da eseguire e stato corrente, restituisce lo stato successivo

Funzione stato prossimo: Soffermandoci sull'ultimo punto, vogliamo trovare la funzione

$$[\![\_]\!](\_) : W\text{-COM} \times W\text{-STATI} \to W\text{-STATI}_\perp$$

Che, dati un comando C del linguaggio WHILE e lo stato corrente $\underline{x}$, calcoli:

$$[\![C]\!](\underline{x}) = \underline{y}$$

con $\underline{y}$ stato prossimo. Quest'ultimo dipende dal comando C, ma essendo C induttivo, possiamo provare a dare una definizione induttiva della funzione.

Partendo dal passo base, gli **assegnamenti**:

$$[\![x_k := 0]\!](\underline{x}) = \underline{y} = \begin{cases} x_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}$$

$$[\![x_k := x_j \pm 1]\!](\underline{x}) = \underline{y} = \begin{cases} x_i & \text{se } i \neq k \\ x_j \pm 1 & \text{se } i = k \end{cases}$$

Proseguiamo con il **passo induttivo**:

- **Comando composto**: vogliamo calcolare

$$[\![\texttt{begin } C_1;\dots;C_n \texttt{ end}]\!](\underline{x})$$

Conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva, calcoliamo la funzione

$$[\![C_n]\!]\left(\dots\left([\![C_2]\!]\left([\![C_1]\!](\underline{x})\right)\right)\dots\right) = \left([\![C_n]\!]\circ\dots\circ[\![C_1]\!]\right)(\underline{x})$$

Ovvero, applichiamo in ordine i comandi C_i presenti nel comando composto C

- **Comando while**: vogliamo calcolare

$$[\![\texttt{while } x_k \neq 0 \texttt{ do } C]\!](\underline{x})$$

Conoscendo $[\![C]\!]$ per ipotesi induttiva, calcoliamo la funzione

$$[\![C]\!]\left(\dots\left([\![C]\!](\underline{x})\right)\dots\right)$$

Bisogna capire quante volte eseguire il ciclo: dato $[\![C]\!]^e$ (comando C eseguito e volte) vorremmo trovare il valore di e. Questo è il numero minimo di iterazioni che portano in uno stato in cui $x_k = 0$, ovvero il comando `while` diventa

$$[\![\texttt{while } x_k \neq 0 \texttt{ do } C]\!](\underline{x}) = \begin{cases} [\![C]\!]^e(\underline{x}) & \text{se } \exists e \text{ minimo tale che } [\![C]\!]^e(\underline{x})[k] = 0 \\ \bot & \text{altrimenti} \end{cases}$$

Definita la semantica operazionale, manca solo da definire cos'è la **semantica del programma** W su input n. Questa è la funzione

$$\Psi_W : \mathbb{N} \to \mathbb{N}_\bot \mid \Psi_W(n) = \text{Pro}\,(0, [\![W]\!](w\text{-}in(n)))$$

← (bisognerebbe fare funzione a tratti contando anche il caso $\bot$)

Questo è valido in quanto W programma WHILE è un programma composto, e abbiamo definito come deve comportarsi la funzione $[\![\_]\!](\_)$ sui comandi composti.

Stiamo praticamente dicendo che la semantica del programma W, visto come comando composto, è il risultato, ovvero il contenuto del registro x_0, dell'applicazione del comando W allo stato iniziale.

Potenza Computazionale: La potenza computazionale del sistema di calcolo WHILE è l'insieme

$$F(\text{WHILE}) = \left\{ f \in \mathbb{N}_\bot^{\mathbb{N}} \mid \exists W \in W\text{-PROG}, f = \Psi_W \right\} = \{\Psi_W : W \in W\text{-PROG}\}$$

Ovvero, l'insieme formato da tutte le funzioni che possono essere calcolate con un programma in W-PROG.

1.8.4 Confronto tra macchina RAM e WHILE

Viene naturale confrontare i due sistemi presentati per capire il "*più potente*", sempre che ce ne sia uno. Le possibilità sono:

- $F(\text{RAM}) \subsetneq F(\text{WHILE})$, sarebbe anche comprensibile data l'estrema semplicità del sistema RAM
- $F(\text{RAM}) \cap F(\text{WHILE}) = \varnothing$, avere insiemi disgiunti significherebbe che il concetto di calcolabile dipende dalla macchina considerata
- $F(\text{WHILE}) \subseteq F(\text{RAM})$, che sarebbe sorprendente vista l'apparente maggiore sofisticatezza del sistema WHILE
- $F(\text{WHILE}) = F(\text{RAM})$, ovvero il concetto di calcolabile non dipende dalla tecnologia utilizzata ma è intrinseco nei problemi

Confronto tra sistemi di calcolo: Ponendo di avere $\mathcal{C}_1$ e $\mathcal{C}_2$ sistemi di calcolo con programmi in $\mathcal{C}_1$-PROG e $\mathcal{C}_2$-PROG e le relative potenze computazionali

$$F(\mathcal{C}_1) = \{f : \mathbb{N} \to \mathbb{N}_\perp \mid \exists P_1 \in \mathcal{C}_1\text{-PROG} \mid f = \Psi_{P_1}\} = \{\Psi_{P_1} : P_1 \in \mathcal{C}_1\text{-PROG}\}$$

$$F(\mathcal{C}_2) = \{f : \mathbb{N} \to \mathbb{N}_\perp \mid \exists P_2 \in \mathcal{C}_2\text{-PROG} \mid f = \varphi_{P_2}\} = \{\varphi_{P_2} : P_2 \in \mathcal{C}_2\text{-PROG}\}$$

Mostrare che il primo sistema non è più potente del secondo $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$ vuol dire dimostrare che ogni elemento nel primo insieme deve stare anche nel secondo

$$\forall f \in F(\mathcal{C}_1) \implies f \in F(C_2)$$

Espandendo la definizione di $f \in F(\mathcal{C})$ la relazione diventa

$$\forall P_1 \in \mathcal{C}_1\text{-PROG} \mid f = \Psi_{P_1} \implies \exists P_2 \in \mathcal{C}_2\text{-PROG} \mid f = \varphi_{P_2}$$

Per ogni programma calcolabile nel primo sistema di calcolo ne esiste uno con la stessa semantica nel secondo sistema. Vogliamo trovare un **compilatore** (o **traduttore**), ovvero una funzione che trasformi un programma del primo sistema in uno del secondo.

Traduzioni

Dati $\mathcal{C}_1$ e $\mathcal{C}_2$ due sistemi di calcolo, definiamo **traduzione** da $\mathcal{C}_1$ a $\mathcal{C}_2$ una funzione

$$T : \mathcal{C}_1\text{-PROG} \to \mathcal{C}_2\text{-PROG}$$

Con le seguenti proprietà:

- **programmabile**: esiste un modo per programmarla effettivamente
- **completa**: deve saper tradurre *ogni* programma in $\mathcal{C}_1$-PROG in un programma in $\mathcal{C}_2$-PROG
- **corretta**: mantiene la semantica dei programmi di partenza, ovvero

  $$\forall P \in \mathcal{C}_1\text{-PROG}, \quad \Psi_P = \varphi_{T(P)}$$

  dove Ψ rappresenta la semantica dei programmi in $\mathcal{C}_1$-PROG e φ rappresenta la semantica dei programmi in $\mathcal{C}_2$-PROG

Teorema 1.8.1. *Se esiste $T : \mathcal{C}_1$-PROG $\to \mathcal{C}_2$-PROG allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$*

Proof. Se $f \in F(\mathcal{C}_1)$ allora esiste un programma $P_1 \in \mathcal{C}_1$-PROG tale che $\Psi_{P_1} = f$. A questo programma P_1 applico T, ottenendo $T(P_1) = P_2 \in \mathcal{C}_2$-PROG (per *completezza*) tale che $\varphi_{P_2} = \Psi_{P_1} = f$ (per *correttezza*).

Ho trovato un programma $P_2 \in \mathcal{C}_2$-PROG la cui semantica è f, allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$.

□

Mostreremo che $F(\text{WHILE}) \subseteq F(\text{RAM})$, ovvero che il sistema WHILE non è più potente del sistema RAM.

Costruiremo un compilatore

$$\text{Comp} : W\text{-PROG} \to \text{PROG}$$

che rispetti le caratteristiche di programmabilità, completezza e correttezza.

1.8.5 $F(\textbf{WHILE}) \subseteq F(\textbf{RAM})$

Per provare che $F(\text{WHILE}) \subseteq F(\text{RAM})$ vogliamo costruire un compilatore da WHILE a RAM.

Per comodità, introduciamo un linguaggio RAM *etichettato*: aggiunge la possibilità di etichettare un'istruzione che indica un punto di salto o di arrivo (stile label assembly). Non altera la potenza espressiva del linguaggio in quanto si tratta di un'aggiunta puramente sintattica: il RAM etichettato si traduce facilmente in RAM puro.

Essendo W-PROG un insieme definito induttivamente, possiamo definire induttivamente anche il compilatore:

- **Passo base**: come compilare gli assegnamenti
- **Passo induttivo**:
  1. Per I.H., assumo di sapere $\text{Comp}(C_1), \ldots, \text{Comp}(C_m)$ e mostro come compilare il comando composto `begin` $C_1; \ldots; C : n$ `end`
  2. Per I.H., assumo di sapere $\text{Comp}(C)$ e mostro come compilare il comando `while` $x_k \neq 0$ `do` C

Nelle traduzioni andremo a mappare la variabile WHILE x_k nel registro RAM R_k. Questo non crea problemi in quanto stiamo mappando un numero finito di registri in un insieme infinito.

Il primo assegnamento che mappiamo è $x_k := 0$

$$\text{Comp}(x_k := 0) = \boxed{\begin{array}{ll} \texttt{LOOP:} & \texttt{IF } R_k = 0 \texttt{ THEN GOTO EXIT} \\ & R_k \leftarrow R_k \dot{-} 1 \\ & \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO LOOP} \\ \texttt{EXIT:} & R_k \leftarrow R_k \dot{-} 1 \end{array}}$$

Questo programma RAM azzera il valore di R_k usando il registro R_{21} per saltare al check della condizione iniziale. Viene utilizzato il registro R_{21} perché, non essendo mappato a nessuna variabile WHILE, sarà sempre nullo dopo la fase di inizializzazione.

Gli altri due assegnamenti da mappare sono $x_k := x_j + 1$ e $x_k := x_j \dot{-} 1$:

- Se $k = j$, la traduzione è banale e l'istruzione RAM è

$$\text{Comp}(x_k := x_k \pm 1) = R_k \leftarrow R_k \pm 1$$

- Invece se $k \neq j$ la prima idea è quella di "spostare" x_j in x_k e poi fare ± 1 ma non funziona per due ragioni
  1. se $R_k \neq 0$ la migrazione (quindi sommare R_j a R_k) non genera R_j dentro R_k. Si può risolvere azzerando il registro R_k prima della migrazione
  2. R_j dopo il trasferimento è ridotto a 0, ma questo non è il senso dell'istruzione, vorrei solo "*fotocopiarlo*" dentro R_k. Questo può essere risolto salvando R_j in un altro registro, azzerare R_k, spostare R_j e ripristinare il valore originale di R_j

  Quindi possiamo mappare come:

$$\text{Comp}(x_k := x_j \pm 1) = \boxed{\begin{array}{ll}
\texttt{LOOP:} & \texttt{IF } R_j = 0 \texttt{ THEN GOTO E1} \\
& R_j \leftarrow R_j \dot{-} 1 \\
& R_{22} \leftarrow R_{22} + 1 \\
& \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO LOOP} \\
\texttt{E1:} & \texttt{IF } R_k = 0 \texttt{ THEN GOTO E2} \\
& R_k \leftarrow R_k \dot{-} 1 \\
& \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO E1} \\
\texttt{E2:} & \texttt{IF } R_{22} = 0 \texttt{ THEN GOTO E3} \\
& R_k \leftarrow R_k + 1 \\
& R_j \leftarrow R_j + 1 \\
& R_{22} \leftarrow R_{22} \dot{-} 1 \\
& \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO E2} \\
\texttt{E3:} & R_k \leftarrow R_k \pm 1
\end{array}}$$

Salva R_j in R_{22}

Azzera R_k

Rigenera R_j e R_k da R_{22}

Per I.H. sappiamo come compilare $C_1, \dots C_m$. Possiamo calcolare la compilazione del comando composto come

$$\text{Comp}(\texttt{begin } C_1; \dots; C_n \texttt{ end}) = \boxed{\begin{array}{l} \text{Comp}(C_1) \\ \dots \\ \text{Comp}(C_m) \end{array}}$$

Per I.H. sappiamo come compilare C. Possiamo calcolare la compilazione del comando `while` come

$$\text{Comp}(\texttt{while } x_k \neq 0 \texttt{ do } C) = \boxed{\begin{array}{ll}
\texttt{LOOP:} & \texttt{IF } R_k = 0 \texttt{ THEN GOTO EXIT} \\
& \text{Comp}(C) \\
& \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO LOOP} \\
\texttt{EXIT:} & R_k \leftarrow R_k \dot{-} 1
\end{array}}$$

$\leftarrow$ (Questa istruzione e' un po come un `pass`/`nop` in quanto R_k è sicuramente 0)

La funzione

$$\text{Comp} : W\text{-PROG} \to \text{PROG}$$

appena costruita soddisfa le tre proprietà desiderate in quanto:

- Facilmente programmabile
- Compila ogni Sorgente W-PROG
- Mantiene la semantica ovvero $\Psi_P = \varphi_{\text{Comp}(P)}$

Di conseguenza

$$F(\text{WHILE}) \subseteq F(\text{RAM})$$

TL;DR: Vogliamo definire induttivamente un compilatore, definendo prima come compilare gli assegnamenti, poi i comandi composti:

- Per mappare $x_k := 0$, usiamo un loop per decrementare di 1 R_k finché non raggiunge 0 (usando R_{21}, sempre posto a 0, per saltare alla condizione iniziale)
- Per mappare $x_k := x_j \pm 1$
  - Se $k = j$ uso banalmente l'incremento/decremento RAM
  - Se $k \neq j$, salvo R_j in R_{22} (decrementando uno e incrementando l'altro finché $R_j \neq 0$), azzero R_k (loop che decrementa), rigenero R_j e R_k da R_{22}, prima di uscire incremento/decremento di 1 R_k
- Il comando composto è una composizione, che sappiamo essere compilabile per I.H.

- Il comando `while` lo possiamo modellare come un loop sulla compilazione del comando all'interno, compilabile per I.H.

In questo modo abbiamo una funzione, programmabile, completa e corretta, dimostrando che $F(\text{WHILE}) \subseteq F(\text{RAM})$.

1.8.6 $F(\mathbf{RAM}) \subseteq F(\mathbf{WHILE})$

Abbiamo mostrato che una macchina WHILE non è più potente di una macchina RAM, ora vogliamo fare l'inverso. Per farlo si userà il concetto di interprete.

Interprete in WHILE per RAM

Introduciamo il concetto di **interprete**. Chiamiamo I_W l'interprete scritto in linguaggio WHILE per programmi scritti in linguaggio RAM.

I_W prende in input un programma $P \in \text{PROG}$ e un dato $x \in \mathbb{N}$ e restituisce "*l'esecuzione*" di P sull'input x. Più formalmente, restituisce la semantica di P su x, quindi $\varphi_P(x)$.

$$\begin{array}{r} P \in \text{PROG} \longrightarrow \\ x \in \mathbb{N} \longrightarrow \end{array} \boxed{I_W} \longrightarrow \varphi_P(x)$$

Notiamo come l'interprete non crei dei prodotti intermedi, ma si limita a eseguire P sull'input x.

Due problemi principali:

1. Il primo riguarda il tipo di input della macchina WHILE in quanto questa non sa leggere il programma P (listato di istruzioni RAM), sa leggere solo numeri. Dobbiamo modificare I_W in modo che non passi più P ma la sua codifica $cod(P) = n \in \mathbb{N}$. Questo restituisce la semantica del programma codificato con n, che è P, quindi $\varphi_n(x) = \varphi_P(x)$
2. Il secondo problema riguarda la quantità di dati in input alla macchina WHILE: questa legge l'input da un singolo registro, mentre qui ne stiamo passando due. Bisogna modificare I_W, condensando l'input tramite la funzione coppia di Cantor, che diventa $\langle x, n \rangle$

Quindi

$$\langle x, n \rangle \longrightarrow \boxed{I_W} \longrightarrow \varphi_n(x) = \varphi_P(x)$$

La semantica di I_W diventa

$$\forall x, n \in \mathbb{N} \quad \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$

Similmente a prima, useremo per comodità il linguaggio **macro-WHILE**, che include alcune macro comode nella scrittura di I_W. Viene modificata solo la sintassi e di conseguenza non la potenza del linguaggio.

Le **macro** utilizzate sono:

- $x_k := x_j + x_s$
- $x_k := \langle x_j, x_s \rangle$
- $x_k := \langle x_1, \ldots, x_n \rangle$
- $x_k := \text{Proj}(x_j, x_s)$, proiezione, estrae l'elemento x_j-esimo dalla lista codificata in x_s
- $x_k := \text{incr}(x_j, x_s)$, ritorno codifica della lista x_s con l'elemento in posizione x_j aumentato di 1
- $x_k := \text{decr}(x_j, x_s)$, ritorno codifica della lista x_s con l'elemento in posizione x_j diminuito di 1
- $x_k := \sin(x_j)$

- $x_k := \text{des}(x_j)$
- Costrutto `If …then …else`

Ognuna di queste macro può essere sostituita da un frammento W-PROG puro quindi F(MACRO-WHILE) = F(WHILE) ma la versione con le macro è decisamente più comoda.

Stato della macchina RAM nell'interprete

Risolto il problema dell'input di un interprete scritto in linguaggio WHILE per i programmi RAM, ora vogliamo scrivere questo interprete. In sintesi, l'interprete esegue una dopo l'altra le istruzioni RAM del programma P e restituisce il risultato $\varphi_P(x)$ (da notare che restituisce il risultato, non un eseguibile, letteralmente come interpreti e compilatori per i linguaggi di programmazione classici).

L'interprete ricostruisce virtualmente tutto ciò che gli serve per gestire il programma. Nel caso di I_W deve ricostruire l'ambiente di una macchina RAM. Quello che faremo sarà ricreare il programma P, il Program Counter L e i registri $R_0, R_1, \ldots$, dentro le variabili messe a disposizione dalla macchina WHILE.

Primo problema: i programmi RAM possono utilizzare infiniti registri, mentre i programmi WHILE ne hanno solo 21. *Ma P usa veramente un numero infinito di registri?*

In realtà no: se $cod(P) = n$ allora P non userà mai registri R_j con $j > n$. Il programma P userà sempre un numero finito di registri il cui contenuto può essere racchiuso in una lista $a_0, \ldots, a_n$. La soluzione consiste nel raggruppare tutti i valori dei registri tramite Cantor $\langle a_0, \ldots, a_n \rangle$ e salvarne la codifica in un unica variabile. Useremo due registri in più per comodità (possono essere utili, you never know).

L'interprete I_W salva lo stato della macchina RAM nel seguente modo (con $I_W(\langle x, n \rangle) = \varphi_n(x)$)

- $x_0 \leftarrow \langle R_0, \ldots, R_{n+2} \rangle$: stato della memoria della macchina RAM
- $x_1 \leftarrow L$: Program Counter
- $x_2 \leftarrow x$: dato su cui lavora P
- $x_3 \leftarrow n$: "listato" del programma P
- x_4: codice dell'istruzione da eseguire, prelevata da x_3 grazie a x_1

Ricordiamo che inizialmente I_w trova il suo input $\langle x, n \rangle$ nella variabile x_1.

Implementazione:

```
// Inizialmente l'input si trova in x_1
x_2 := sin(x_1);
x_3 := des(x_1);
x_0 := ⟨0, x_2, ..., 0⟩;
x_1 := 1;
while x_1 ≠ 0 do                                   // se x_1 = 0 allora STOP
   if (x_1 > length(x_3)) then                     // supero l'ultima istruzione
      x_1 := 0;                                                  // STOP
   else
      x_4 := Proj(x_1, x_3);                       // estraggo istruzione corrente
      if x_4 mod 3 = 0 then                                  // R_k ← R_k + 1
         x_5 := x_4/3;                                                // k
         x_0 := incr(x_5, x_0);
         x_1 := x_1 + 1;
      if x_4 mod 3 = 1 then                                  // R_k ← R_k ∸ 1
         x_5 := (x_4 − 1)/3;                                          // k
         x_0 := decr(x_5, x_0);
         x_1 := x_1 + 1;
      if x_4 mod 3 = 2 then                          // IF R_k = 0 THEN GOTO m
         x_5 := sin((x_4 + 1)/3);                                     // k
         x_6 := des((x_4 + 1)/3);                                     // m
         if Proj(x_5, x_0) = 0 then                          // verifico R_k = 0
            x_1 := x_6;
         else
            x_1 := x_1 + 1;
x_0 = sin(x_0);                                  // metto in x_0 il risultato φ_n(x)
```

Avendo l'interprete I_W si può costruire un compilatore

$$\text{Comp} : \text{PROG} \to W\text{-PROG}$$

tale che

$$\text{Comp}(P \in \text{PROG}) \equiv \boxed{\begin{array}{l} x_2 := cod(P) \\ x_1 := \langle x_1, x_2 \rangle \\ I_W \end{array}}$$

Ovvero, il compilatore non fa altro che cablare all'input x il programma RAM da interpretare e procede con l'esecuzione dell'interprete. Possiamo verificare le proprietà di un compilatore:

- **programmabile**: lo abbiamo appena fatto
- **completo**: l'interprete riconosce ogni istruzione RAM e riesce a codificarla
- **corretto**: vale la relazione $P \in \text{PROG} \implies \text{Comp}(P) \in W\text{-PROG}$, quindi

  $$\Psi_{\text{Comp}(P)}(x) = \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$

  rappresenta la sua semantica

Abbiamo dimostrato che

$$F(\text{RAM}) \subseteq F(\text{WHILE})$$

Ovvero l'inclusione opposta al risultato precedente.

Conseguenze

Visti i risultati ottenuti possiamo definire un teorema importante.

Teorema 1.8.2 (Teorema di Böhm-Jacopini)**.** *Per ogni programma con `GOTO` RAM ne esiste uno equivalente in un linguaggio strutturato (WHILE).*

Permette di legare la programmazione a basso livello con quella ad alto livello. In breve, il `GOTO` può essere eliminato e la programmazione a basso livello può essere sostituita con quella ad alto livello.

Inoltre abbiamo di conseguenza dimostrato che

$$\begin{gathered} F(\text{WHILE}) \subseteq F(\text{RAM}) \\ F(\text{RAM}) \subseteq F(\text{WHILE}) \\ \Downarrow \\ F(\text{RAM}) = F(\text{WHILE}) \\ \Downarrow \\ F(\text{RAM}) = F(\text{WHILE}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_{\perp}^{\mathbb{N}} \end{gathered}$$

Quindi, abbiamo dimostrato che i sistemi RAM e WHILE sono in grado di **calcolare le stesse cose**, oltre al fatto che **esistono funzioni non calcolabili**, ovvero la parte destra della catena.

1.8.7 Interprete Universale

Usando il compilatore da WHILE a RAM

$$\text{Comp} : W\text{-PROG} \to \text{PROG}$$

sul programma I_W, otteniamo

$$\mathcal{U} = \text{Comp}(I_W) \in \text{PROG}$$

E la sua semantica è

$$\varphi_{\mathcal{U}}(\langle x, n \rangle) = \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x)$$

dove n è la codifica del programma RAM e x il dato di input.

Abbiamo mostrato che esiste un programma RAM in grado di **simulare** tutti gli altri programmi RAM. Questo programma viene detto **interprete universale**.

Un linguaggio verrà considerato "*buono*" se ammette un interprete universale.

1.8.8 Concetto di Calcolabilità

Abbiamo visto come due sistemi di calcolo diversi abbiano la stessa potenza computazionale numerabile. Ma abbiamo visto anche che esistono funzioni non calcolabili. Il nostro obiettivo è quindi definire la regione delle funzioni calcolabili.

Questa regione può essere definita a prescindere dalle macchine usate per calcolare? Bisognerà definire il concetto di "*calcolabile*" in termini astratti, "lontani" dall'informatica, usando la matematica.

1.9 Chiusura

1.9.1 Operazioni

Dato un insieme U, si definisce **operazione** su U una qualunque funzione

$$\text{op} : \underbrace{U \times \cdots \times U}_{k} \to U$$

Il numero k indica l'arità dell'operazione, ovvero la dimensione del dominio dell'operazione.

1.9.2 Proprietà di Chiusura

L'insieme $A \subseteq U$ si dice **chiuso** rispetto all'operazione $\text{op} : U^k \to U$ se e solo se

$$\forall a_1, \ldots, a_k \in A \quad \text{op}(a_1, \ldots, a_k) \in A$$

Ovvero, l'operazione applicata in A restituisce valori $\in A$. In generale, se Ω è un insieme di operazioni su U, allora $A \subseteq U$ è chiuso rispetto a Ω se e solo se A è chiuso per ogni operazione in Ω.

1.9.3 Chiusura di un insieme

Siano $A \subseteq U$ un insieme e $\text{op} : U^k \to U$ un'operazione su di esso. Vogliamo espandere l'insieme A per trovare il più piccolo sottoinsieme di U tale che

- contiene A
- chiuso rispetto a op

Vogliamo espandere A il meno possibile per garantire la chiusura rispetto a op.

Ci sono due risposte banali:

- Se A è già chiuso rispetto a op, allora A stesso è l'insieme cercato
- Sicuramente U soddisfa le due richieste, ma non è detto sia il più piccolo

Teorema 1.9.1. *Siano $A \subseteq U$ insiemi e* op *un'operazione su di essi. Il più piccolo sottoinsieme di U contenente A e chiuso rispetto a* op *si ottiene calcolando la chiusura di A rispetto a* op*, cioè l'insieme A^{op} definito induttivamente come segue:*

1. $\forall a \in A \implies a \in A^{\text{op}}$
2. $\forall a_1, \ldots, a_k \in A^{\text{op}} \implies \text{op}(a_1, \ldots, a_k) \in A^{\text{op}}$
3. *Nient'altro in A*

Una definizione più *operativa* di A^{op} può essere:

1. Inserisco in A^{op} tutti gli elementi di A
2. Applico op a una k-upla di elementi in A^{op}
3. Se il risultato non è in A^{op} lo aggiungo
4. Ripetere i punti 2 e 3 finché A^{op} cresce
5. Output A^{op}

Siano $\Omega = \{\text{op}_1, \ldots, \text{op}_t\}$ un insieme di operazioni su U di arità rispettivamente $k_1, \ldots, k_t$ e $A \subseteq U$ insieme.

Definiamo **chiusura di A rispetto a Ω** il più piccolo sottoinsieme di U contenente A chiuso rispetto a Ω, cioè l'insieme A^{Ω} definito come:

- $\forall a \in A \implies a \in A^\Omega$
- $\forall i \in \{1, \dots, t\}$, $\forall a_1, \dots, a_{k_i} \in A^\Omega \implies op_i(a_1, \dots, a_{k_i}) \in A^\Omega$
- Nient'altro in A^Ω

In breve, la chiusura dell'insieme rispetto a ogni operazione.

1.10 Calcolabilità

Vogliamo una definizione che astrae da qualunque connotato "informatico", per la definizione teorica di calcolabilità introdurremo gli insiemi:

- **ELEM**: insieme di tre funzioni che *qualunque* idea di calcolabile proponibile deve considerare calcolabili. Non può esaurire il concetto di calcolabilità, quindi verrà espanso con altre funzioni
- Ω: insieme di operazioni su funzioni che *costruiscono nuove funzioni*. Le operazioni in Ω sono banalmente implementabili e, applicate a funzioni calcolabili, generano nuove funzioni calcolabili
- $\mathbf{ELEM}^\Omega = \mathscr{P}$: la classe delle funzioni ricorsive parziali. L'idea astratta della classe delle funzioni calcolabili secondo Kleene

In seguito alla definizione di $\mathscr{P}$ ci sarà da domandarci se questa idea di calcolabile coincida con le funzioni presenti in $F(\text{RAM}) = F(\text{WHILE})$.

1.10.1 ELEM

Definiamo ELEM con le seguenti funzioni

$$\begin{aligned} \text{ELEM} = \quad \{ \quad \text{successore:} \quad & s(x) = x + 1, && x \in \mathbb{N} \\ \text{zero:} \quad & 0^n(x_1, \dots, x_n) = 0, && x_i \in \mathbb{N} \\ \text{proiettori:} \quad & \text{Pro}_k^n(x_1, \dots, x_n) = x_k, \quad x_i \in \mathbb{N} \quad \} \end{aligned}$$

Queste sono funzioni di partenza che qualsiasi idea teorica di calcolabilità non può non considerare come calcolabile. Ovviamente non rappresenta tutto ciò che è calcolabile, va quindi ampliato. Per esempio, vogliamo che $f(x) = x + 2$ sia calcolabile.

1.10.2 Ω

Definiamo ora un insieme Ω di operazioni che permettano di ampliare le funzioni di ELEM fino a coprire tutte le funzioni calcolabili.

Composizione

Il primo operatore è quello di **composizione**. Siano

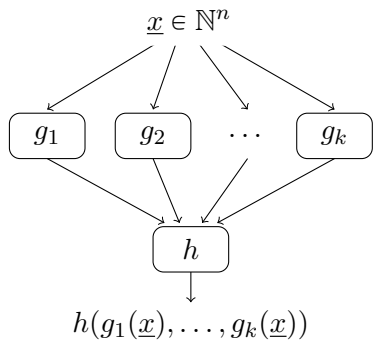
- $h : \mathbb{N}^k \to \mathbb{N}$ funzione di composizione
- $g_1, \dots, g_k : \mathbb{N}^n \to \mathbb{N}$ "funzioni intermedie"
- $\underline{x} \in \mathbb{N}^n$ input

Allora definiamo

$$\text{COMP}(h, g_1, \dots, g_k) : \mathbb{N}^n \to \mathbb{N}$$

La funzione tale che

$$\text{COMP}(h, g_1, \dots, g_k)(\underline{x}) = h(g_1(\underline{x}), \dots, g_k(\underline{x}))$$



La funzione COMP è *intuitivamente* calcolabile partendo da funzioni calcolabili: prima eseguo le singole funzioni $g_1, \ldots, g_k$, poi applico la funzione h sui risultati.

Calcoliamo ora la **chiusura** di ELEM rispetto a COMP, ovvero l'insieme $\text{ELEM}^{\text{COMP}}$. La funzione $f(x) = x + 2$ appartiene a questo insieme perché:

$$\text{COMP}(s, s)(x) = s(s(x)) = (x + 1) + 1 = x + 2$$

Di conseguenza tutte le funzioni lineari del tipo $f(x) = x+k$ con k prefissato apparterranno all'insieme.

Ma la funzione somma $\text{somma}(x, y) = x + y$, senza valori prefissati, non appartiene all'insieme, quindi doppiamo ampliare $\text{ELEM}^{\text{COMP}}$ con altre operazioni.

Ricorsione Primitiva

Definiamo un'operazione che permetta di **iterare** sull'operatore di composizione, la **ricorsione primitiva**, usata per definire funzioni ricorsive.

Siano:

- $g : \mathbb{N}^n \to \mathbb{N}$ funzione **caso base**
- $h : \mathbb{N}^{n+2} \to \mathbb{N}$ funzione **passo ricorsivo**
- $\underline{x} \in \mathbb{N}^n$ **input**

Definiamo

$$\text{RP}(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(f(\underline{x}, y-1), y-1, \underline{x}) & \text{se } y > 0 \end{cases}$$

Funzione che generalizza la definizione ricorsiva di funzioni, e senza starci troppo è facilmente implementabile.

Chiusura di $\text{ELEM}^{\text{COMP}}$ rispetto a RP vuol dire calcolare l'insieme $\text{ELEM}^{\{\text{COMP,RP}\}}$. Chiamiamo

$$\text{RICPRIM} = \text{ELEM}^{\{\text{COMP,RP}\}}$$

l'insieme ottenuto dalla chiusura, cioè l'insieme delle funzioni ricorsive primitive.

In questo insieme abbiamo la somma, infatti:

$$\text{somma}(x, y) = \begin{cases} \text{Pro}_1^2(x, y) & \text{se } y = 0 \\ s(\text{somma}(x, y-1)) & \text{se } y > 0 \end{cases}$$

Altre funzioni in RICPRIM:

$$\text{prodotto}(x, y) = \begin{cases} 0^2(x, y) & \text{se } y = 0 \\ \text{somma}(x, \text{prodotto}(x, y-1)) & \text{se } y > 0 \end{cases}$$

$$\text{predecessore } P(x) = \begin{cases} 0 & \text{se } x = 0 \\ x - 1 & \text{se } x > 0 \end{cases} \implies x \dot{-} y = \begin{cases} x & \text{se } y = 0 \\ P(x) \dot{-} (y-1) & \text{se } y > 0 \end{cases}$$

RICPRIM vs WHILE

L'insieme RICPRIM contiene *molte funzioni*, ma dobbiamo capire se ha raggiunto l'insieme $F(\text{WHILE})$.

RICPRIM è definito come:

- $\forall f \in \text{ELEM} \implies f \in \text{RICPRIM}$ BASE
- Se $h, g_1, \ldots, g_k \in \text{RICPRIM} \implies \text{COMP}(h, g_1, \ldots, g_k) \in \text{RICPRIM}$ PASSI
- Se $g, h \in \text{RICPRIM} \implies \text{RP}(g, h) \in \text{RICPRIM}$ PASSI
- Nient'altro sta in RICPRIM

Teorema 1.10.1. *$RICPRIM \subseteq F(WHILE)$*

Proof.
Passo Base: Le funzioni di ELEM sono ovviamente while-programmabili.

Passo Induttivo: Per COMP, assumiamo per ipotesi induttiva che $h, g_1, \ldots, g_k \in$ RICPRIM siano while-programmabili, allora esistono $H, G_1, \ldots, G_K \in W$-PROG tali che $\Psi_H = h$, $\Psi_{G_1} = g_1, \ldots, \Psi_{G_k} = g_k$.
Un programma W WHILE che calcola COMP è

```
// Inizialmente in x_1 ho x
x_1 := x
begin
    x_0 := G_1(x_1);
    x_0 := [x_0, G_2(x_1)];
    ...
    x_0 := [x_0, G_k(x_1)] ;                    // x_0 := [G_1(x), ..., G_k(x)]
    x_1 := H(x_0) ;                             // x_0 := H(G_1(x), ..., G_k(x))
end
```

Quindi abbiamo $\Psi_W(\underline{x}) = \text{COMP}(h, g_1, \ldots, g_k)(\underline{x})$.

Per RP, assumiamo che $h, g \in$ RICPRIM siano while-programmabili, allora esistono $H, G \in W$-PROG tali che $\Psi_H = h$ e $\Psi_G = g$. Le funzioni ricorsive primitive le possiamo vedere come delle iterazioni che, partendo dal caso base G, mano a mano compongono con H fino a quando non si raggiunge y (escluso). Mostriamo un programma WHILE che calcola

$$\text{RP}(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(f(\underline{x}, y-1), y-1, \underline{x}) & \text{se } y > 0 \end{cases}$$

```
// Inizialmente in x_1 ho ⟨x, y⟩
input(x, y)
begin
    t := G(x);                                   // t contiene f(x, y)
    k := 1;
    while k ⩽ y do
        begin
            t := H(t, k − 1, x);
            k := k + 1;
        end
end
```

Quindi $\Psi_W(\langle x, y\rangle) = \mathrm{RP}(h, g)(\underline{x}, y)$.

□

Abbiamo dimostrato che RICPRIM $\subseteq F$(WHILE) e possiamo anche dire che l'*inclusione è propria*: nel linguaggio WHILE si possono avere cicli infiniti mentre in RICPRIM no in quanto contiene solo funzioni totali (si dimostra per induzione strutturale) mentre WHILE contiene anche delle funzioni parziali. Di conseguenza

$$\text{RICPRIM} \subsetneq F(\text{WHILE})$$

Sistema di calcolo FOR

Per poter *raggiungere* F(WHILE) bisognerà ampliare RICPRIM. Visto che le funzioni in RICPRIM sono tutte totali, ogni ciclo in RICPRIM ha un inizio e una fine ben definiti: il costrutto utilizzato per dimostrare RP $\in F$(WHILE) nella dimostrazione precedente permette di definire un nuovo tipo di ciclo, il **ciclo FOR**

```
input(x, y)
begin
    t := G(x);
    for k := 1 to y do
        t := H(t, k − 1, x);
end
```

Il FOR è un costrutto che si serve di una variabile di controllo che parte da un preciso valore e arriva a un valore limite, senza che la variabile di controllo venga toccata.

Il FOR language è quindi un linguaggio WHILE dove l'istruzione di loop è un `for`. Possiamo quindi dire che RICPRIM $= F(\text{FOR}) \subset F(\text{WHILE})$.

Dato che WHILE sembra "vincere" su RICPRIM solo per i loop infiniti, restringiamo WHILE imponendo loop finiti. Creiamo

$$\tilde{F}(\text{WHILE}) = \{\Psi_W : W \in W\text{-PROG} \wedge \Psi_W \text{ è totale}\}$$

Ma dove si posiziona questo insieme rispetto a RICPRIM? L'inclusione è propria? La risposta è sì, ci sono funzioni in $\tilde{F}$(WHILE) non riscrivibili come funzioni in RICPRIM. Un esempio è la funzione di Ackermann

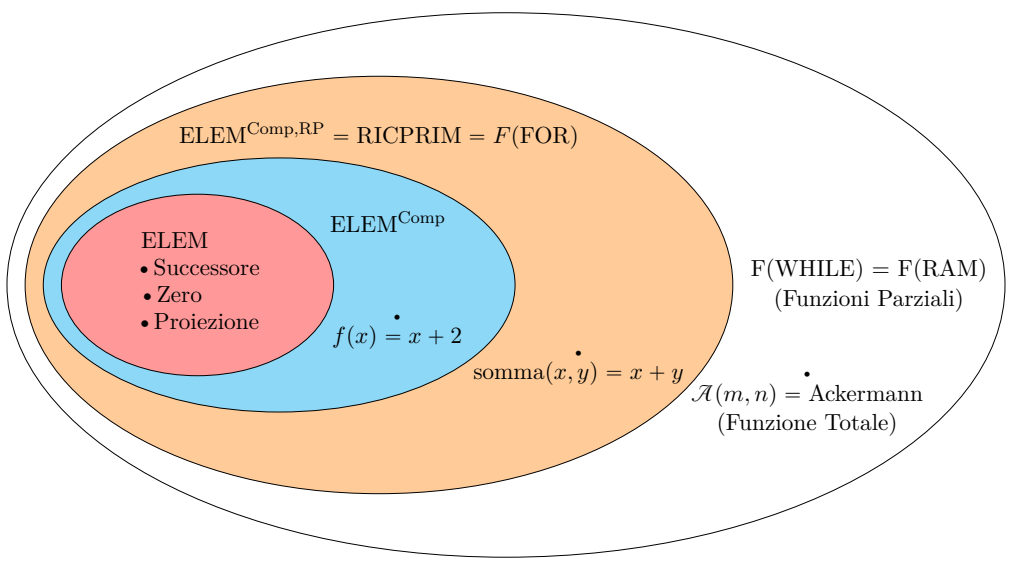
$$\mathcal{A}(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ \mathcal{A}(m - 1, 1) & \text{se } m > 0 \wedge n = 0 \\ \mathcal{A}(m - 1, \mathcal{A}(m, n - 1)) & \text{se } m > 0 \wedge n > 0 \end{cases}$$

La quale non appartiene a RICPRIM, per causa della doppia ricorsione cresce troppo in fretta.

Questo mostra che il sistema WHILE è più potente delle funzioni in RICPRIM, anche escludendo le funzioni parziali.

1.10.3 Ampliamento di RICPRIM

Siamo nella "*direzione giusta*" non avendo cose che non avremmo catturato in $F(\text{RAM})$, ma questo non basta: mancano da catturare le funzioni parziali.



Minimalizzazione

Introduciamo un operatore per permettere la presenza di funzioni parziali: **minimalizzazione**. Sia $f : \mathbb{N}^{n+1} \to \mathbb{N}$ con $f(\underline{x}, y)$ e $\underline{x} \in \mathbb{N}^n$, allora:

$$\text{MIN}(f)(\underline{x}) = g(\underline{x}) = \begin{cases} y & \text{se } f(\underline{x}, y) = 0 \wedge (\forall y' < y, f(\underline{x}, y') \downarrow \wedge f(\underline{x}, y') \neq 0) \\ \perp & \text{altrimenti} \end{cases}$$

Un'altra definizione di MIN è

$$\mu_y(f(\underline{x}, y) = 0)$$

Informalmente, restituisce il più piccolo valore di y che azzera $f(\underline{x}, y)$, ovunque precedentemente definita su y'.

Alcuni esempi con $f : \mathbb{N}^2 \to \mathbb{N}$

$\boldsymbol{f(x,y)}$	$\boldsymbol{\text{MIN}(f)(x) = g(x)}$
$x + y + 1$	$\perp$
$x \dot{-} y$	x
$y \dot{-} x$	0
$x \dot{-} y^2$	$\lceil \sqrt{x} \rceil$
$\left\lfloor \frac{x}{y} \right\rfloor$	$\perp$

1.10.4 Classe $\mathscr{P}$

Ampliamo RICPRIM chiudendolo con l'operazione MIN

$$\text{ELEM}^{\{\text{COMP,RP,MIN}\}} = \mathscr{P} = \{\text{Funzioni Ricorsive Parziali}\}$$

Abbiamo ottenuto la classe delle **funzioni ricorsive parziali**.

$\mathscr{P}$ vs WHILE

Vogliamo ora analizzare come $\mathscr{P}$ si pone rispetto a $F(\text{WHILE})$, sapendo che *sicuramente* amplia RICPRIM.

Teorema 1.10.2. *$\mathscr{P} \subseteq F(WHILE)$.*

Proof. $\mathscr{P}$ è definito per chiusura, ma si può anche definire induttivamente:

- le funzioni ELEM sono in $\mathscr{P}$
- se $h, g_1, \ldots, g_k \in \mathscr{P}$ allora $\text{COMP}(h, g_1, \ldots, g_k) \in \mathscr{P}$
- se $h, g \in \mathscr{P}$ allora $\text{RP}(h, g) \in \mathscr{P}$
- se $f \in \mathscr{P}$ allora $\text{MIN}(f) \in \mathscr{P}$
- nient'altro in $\mathscr{P}$

Di conseguenza, per induzione strutturale su $\mathscr{P}$ dimostriamo:

- **Passo base:** le funzioni elementari sono while-programmabili, come già dimostrato
- **Passi induttivi:**
  - siano $h, g_1, \ldots, g_k \in \mathscr{P}$ while-programmabili per I.H., allora mostro che $\text{COMP}(h, g_1, \ldots, g_k)$ è while-programmabile; già fatto per RICPRIM
  - siano $h, g \in \mathscr{P}$ while-programmabili per I.H., mostro che $\text{RP}(h, g)$ è while-programmabile; già fatto per RICPRIM
  - sia $f \in \mathscr{P}$ while-programmabile e sia f calcolabile nel programma WHILE per I.H., mostro che $\text{MIN}(f)$ è while-programmabile. Devo trovare un programma WHILE che calcoli la minimizzazione:

```
P ≡ input(x̲)
begin
    y := 0;                                   // parte da 0
    while F(x̲, y) ≠ 0 do
        y := y + 1;                           // aumenta di 1
end
```

    Questo è un programma che calcola la minimizzazione: se non esiste y che azzeri $f(\underline{x}, y)$ il programma va in loop (i.e., incrementa di 1 ad ogni iterazione e termina quando azzera la funzione), quindi la semantica di P è $\perp$ secondo MIN

Concludiamo quindi che $\mathscr{P} \subseteq F(\text{WHILE})$.

□

Viene naturale chiedersi se vale la relazione inversa.

Teorema 1.10.3. *$F(WHILE) \subseteq \mathscr{P}$.*

Proof. Sappiamo che

$$F(\text{WHILE}) = \{\Psi_W : W \in W\text{-PROG}\}$$

Consideriamo $\Psi_W \in F(\text{WHILE})$ e mostriamo che $\Psi_W \in \mathscr{P}$, facendo vedere che può essere espressa come composizione, ricorsione primitiva e minimalizzazione a partire dalle funzioni in ELEM.

Le funzioni in W-PROG sono nella forma

$$\Psi_W = \text{Pro}_0^{21}\left([\![W]\!](W\text{-in}(\underline{x}))\right)$$

Dove:

- $[\![C]\!](\underline{x}) = \underline{y}$ la funzione che restituisce lo stato prossimo $\underline{y} \in \mathbb{N}^{21}$ a seguito dell'esecuzione del comando C a partire dallo stato corrente $\underline{x} \in \mathbb{N}^{21}$
- La funzione W-in(n) restituisce lo stato iniziale della macchina WHILE su input n
- Pro_0^{21} preleva l'output dal registro x_0

Abbiamo definito Ψ_W come composizione delle funzioni Pro_0^{21} e $[\![W]\!](W\text{-in}(\underline{x}))$, ma allora

1. $\text{Pro}_0^{21} \in \text{ELEM} \implies \text{Pro}_0^{21} \in \mathscr{P}$
2. $\mathscr{P}$ è chiuso rispetto alla composizione

Di conseguenza, se dimostro che la funzione di stato prossimo $[\![C]\!]$ è ricorsiva parziale $\in \mathscr{P}$ allora anche $\Psi_W \in \mathscr{P}$, per la definizione induttiva di $\mathscr{P}$.

La funzione di stato prossimo $[\![]\!]() : \mathbb{N}^{21} \to \mathbb{N}^{21}$ restituisce elementi in $\mathbb{N}^{21}$, mentre gli elementi in $\mathscr{P}$ hanno codominio $\mathbb{N}$.

Per risolvere il problema si definisce la **funzione f_C numero prossimo** in cui viene applicato Cantor sull'array degli stati.

$$[\![C]\!]\underline{x} = \underline{y} \quad \text{con} \quad \underline{x}, \underline{y} \in \mathbb{N}^{21}$$

$$f_C(x) = y \quad \text{con} \quad x = [\underline{x}],\ y = [\underline{y}]$$

Si noti che per passare da $[\![C]\!](\underline{x})$ a $f_C(x)$ si usano operazioni ricorsive parziali:

$$f_C(x) = y$$

$$\text{Pro} \in \mathscr{P} \quad \downarrow\uparrow \quad [\,] \in \mathscr{P}$$

$$[\![C]\!](\text{Pro}(0,x),\ldots,\text{Pro}(20,x)) = (\text{Pro}(0,y),\ldots,\text{Pro}(20,y))$$

Quindi si ha che f_C si comporta come $[\![C]\!](\underline{x})$ sullo stato prossimo. Basterà ora dimostrare che f_C è effettivamente una funzione ricorsiva parziale:

$$f_C \in \mathscr{P} \quad \Leftrightarrow \quad [\![C]\!](\underline{x}) \in \mathscr{P}$$

Dimostriamo, tramite induzione strutturale, sul comando WHILE C:

- **Passo base**: azzeramento e incremento/decremento
  - $C \equiv \boxed{x_k := 0}$

    $$f_C(x) = [\text{Pro}(0,x),\ldots,\underset{\substack{\uparrow\\ \text{posizione } k}}{0},\ldots,\text{Pro}(20,x)]$$

    Viene usata una composizione di funzioni $\in \mathscr{P} \implies f_{x_k:=0} \in \mathscr{P}$
  - $C \equiv \boxed{x_k := x_j +/\dot{-} 1}$

    $$f_C(x) = [\text{Pro}(0,x),\ldots,\underset{\substack{\uparrow\\ \text{posizione } k}}{\text{Pro}(j,x)+/\dot{-}1},\ldots,\text{Pro}(20,x)]$$

    Viene usata una composizione di funzioni $\in \mathscr{P} \implies f_{x_k:=x_j\pm 1} \in \mathscr{P}$

- **Passi induttivi**: i comandi "complessi" WHILE devono rientrare in $\mathscr{P}$
  - $C \equiv \boxed{\texttt{begin } C_1;\ C_2;\ \ldots;\ C_m;\ \texttt{end}}$

    $$f_C(x) = f_{C_m}(\ldots f_{C_1}(x)\ldots)$$

    Viene usata una composizione di f_{C_i} ognuna delle quali è $\in \mathscr{P}$ per ipotesi induttiva $\Rightarrow$ $f_C \in \mathscr{P}$
  - $C' \equiv \boxed{\texttt{while } x_k \neq 0 \texttt{ do } C}$

    $$f_{C'}(x) = f_C^{e(x)}(x) \quad \text{con} \quad e(x) = \mu_y(\text{Pro}(k, f_C^y(x)) = 0)$$

    Sorge qui un problema: $e(x)$ non è costante; non basta quindi la composizione in quanto può essere applicata solo su un numero costante di funzioni.

    Si dovrà allora definire una funzione $T \in \mathscr{P}$:

    $$T(x, y) = f_C^y(x)$$

    È facile farlo usando l'operatore di ricorsione primitiva RP:

    $$T(x, y) = \begin{cases} x & \text{se } y = 0 \\ f_C(T(x, y-1)) & \text{se } y > 0 \end{cases}$$

    Siccome:

    1. $f_C \in \mathscr{P}$ per ipotesi induttiva
    2. RP $\in \mathscr{P}$

    $$\Longrightarrow \quad T(x, y) \in \mathscr{P}$$

    L'ultima cosa da sistemare è $e(x)$:

    $$\begin{aligned} e(x) &= \mu_y(\text{Pro}(k, f_c^y(x)) = 0) \\ &= \mu_y(\text{Pro}(k, T(x, y)) = 0) \end{aligned}$$

    la quale è una minimalizzazione di $T(x, y) \in \mathscr{P}$, quindi $e(x) \in \mathscr{P}$.

    In conclusione:

    $$f_{C'}(x) = f_C^{e(x)}(x) = T(x, e(x))$$

    $f_{C'}$ è formato da una composizione di funzioni $\in \mathscr{P}$, $\Longrightarrow$ $f_{C'} \in \mathscr{P}$.

□

Visti i risultati ottenuti dai due teoremi precedenti, possiamo concludere che:

$$F(\text{WHILE}) = \mathscr{P}$$

La classe delle funzioni ricorsive parziali, ovvero l'idea di *calcolabile* in termini matematici, coincide con l'idea di *calcolabile* proveniente dall'insieme di problemi per i quali vediamo una macchina che li risolva.

1.10.5 Tesi di Church-Turing

Il risultato principale di questo studio è aver trovato due classi di funzioni importanti:

- $\mathscr{P}$ insieme delle **funzioni ricorsive parziali**
- $\mathscr{T}$ insieme delle **funzioni ricorsive totali**

Il primo insieme contiene anche tutte le funzioni del secondo, quindi

$$\mathscr{T} \subset \mathscr{P}$$

Inoltre abbiamo visto (ad esempio tramite la funzione di Ackermann) che

$$\text{RICPRIM} \subset \mathscr{T}$$

L'insieme $\mathscr{P}$ cattura tutti i sistemi di calcolo esistenti: WHILE, RAM, Macchine di Turing, Lambda-calcolo di Church, …
Tutti i modelli di calcolo proposti alla fine individuano sempre la classe delle funzioni ricorsive parziali.
Visti questi risultati, Church e Turing decidono di enunciare un risultato molto importante.

Tesi di Church-Turing: La classe delle funzioni intuitivamente calcolabili coincide con la classe $\mathscr{P}$ delle funzioni ricorsive parziali.

Questa è una tesi e non un teorema, si tratta di una *congettura*, un'opinione, in quanto non è possibile caratterizzare i modelli di calcolo ragionevoli che sono stati e saranno proposti in maniera completa. Possiamo solo decidere se aderire o meno a questa tesi.

Per noi un problema è *calcolabile* quando esiste un modello di calcolo che riesce a risolverlo ragionevolmente. Se volessimo aderire alla tesi di Church-Turing, potremmo dire, in maniera più formale, che:

- *problema ricorsivo parziale* è sinonimo di calcolabile
- *problema ricorsivo totale* è sinonimo di calcolabile da un programma che si arresta su ogni input

1.11 Assiomatizzazione dei Sistemi di Calcolo

Per ora ci si è concentrati sull'analisi della *potenza computazionale* dei sistemi di programmazione visti, affermando che ognuno di questi ha come potenza computazionale $\mathscr{P}$ grazie alla tesi di Church-Turing.

Vorremmo sapere altro sui sistemi di programmazione, vogliamo individuare alcune proprietà "buone" di un sistema di calcolo, dette **assiomi**. Tutti i sistemi di programmazione che rispettano questi assiomi si diranno sistemi di programmazione accettabili. In questo modo si potranno dimostrare proprietà non sul singolo sistema ma su tutti i sistemi accettabili utilizzando appunto solo questi assiomi.

Per indicare un sistema di programmazione si usa

$$\{\varphi_i\}_{i\in\mathbb{N}}$$

ovvero l'insieme delle semantiche (funzioni calcolabili) dal sistema. Il pedice indica i programmi codificati $i \in \mathbb{N}$ di quel sistema.

1.11.1 Assiomi di Rogers

Gli assiomi di Rogers vogliono essere quelle "buone" proprietà ricercate in un sistema di programmazione; in particolare si afferma che: un sistema di programmazione $\{\varphi_i\}$ si dice **accettabile** (o **SPA**) se:

1. Aderisce alla tesi di Church-Turing
2. Ammette interprete universale
3. Rispetta il teorema S_n^m

Si vedranno ora gli assiomi nel dettaglio.

Potenza computazionale

Dato il sistema $\{\varphi_i\}$ vogliamo che

$$\{\varphi_i\} = \mathscr{P}$$

ovvero, rispetta la tesi di Church-Turing. Come già dimostrato, il sistema RAM rispetta il primo assioma

$$F(\text{RAM}) = \mathscr{P}$$

Non si vogliono considerare né sistemi troppo poco potenti (sotto $\mathscr{P}$), né troppo potenti (oltre $\mathscr{P}$).

Interprete universale

$\{\varphi_i\}$ rispetta il secondo assioma se esiste un interprete universale, ovvero un programma $\mu \in \mathbb{N}$ tale che

$$\forall x, n \in \mathbb{N} \quad \varphi_\mu(\langle x, n\rangle) = \varphi_n(x)$$

ovvero, un programma scritto in un certo linguaggio che riesce ad interpretare ogni altro programma n scritto nello stesso linguaggio su qualsiasi input x.

La presenza di un interprete universale permette un'algebra sui programmi, quindi la trasformazione di quest'ultimi.

Teorema S_n^m

$\{\varphi_i\}$ rispetta il terzo assioma se rispetta il teorema S_n^m. Si vedrà prima un teorema più specifico S_1^1.

Teorema 1.11.1 (S_1^1)**.** *Dato $\{\varphi_i\}$ RAM esiste una funzione $S_1^1 \in \mathcal{T}$ tale che*

$$\forall n, x, y \in \mathbb{N} \quad \varphi_n(\langle x, y\rangle) = \varphi_{S_1^1(n,y)}(x)$$

In altre parole, è possibile costruire automaticamente programmi specifici da programmi più generali, ottenuti fissando alcuni input.

Proof. In generale, il programma S_1^1 implementa la funzione

$$S_1^1(n, y) = \bar{n} \text{ t.c. } \varphi_{\bar{n}}(x) = \varphi_n(x, y)$$

dove:

- n è la codifica del programma P a due variabili di input x, y
- $\bar{n}$ è la codifica del programma $\bar{P}$ a una variabile, la cui semantica è identica a quella di P a cui fisso a y il secondo input

Analizzando il sistema RAM si prenda un programma P a due variabili

$$P \in \text{PROG} \mid \varphi_P(\langle x, y\rangle) = x + y$$

$P \equiv$ `// input` $\langle x, y\rangle$ `in` R_1
$R_2 \leftarrow \sin(R_1)$
$R_3 \leftarrow \text{des}(R_1)$
$R_0 \leftarrow R_2 + R_3$

$$\varphi_P(\langle x, y\rangle) = x + y$$

Si vuole produrre un programma $\bar{P}$ con un solo input x che restituisca $x+3$, partendo da P. Si fissa quindi la seconda variabile di P a 3. La funzione S_1^1 dovrà restituire $\bar{P}$

$$\begin{array}{rl} & \bar{P} \equiv \quad \texttt{// input } x \texttt{ in } R_1 \\ & \left.\begin{array}{l} R_0 \leftarrow R_0 + 1 \\ R_0 \leftarrow R_0 + 1 \\ R_0 \leftarrow R_0 + 1 \end{array}\right\} \text{fissa } y \text{ a } 3 \\ & \left.\begin{array}{l} R_1 \leftarrow \langle R_1, R_0 \rangle \end{array}\right\} \text{input di } P \\ & \left.\begin{array}{l} R_0 \leftarrow 0 \end{array}\right\} \text{pulisci } R_0 \\ & \left.\begin{array}{l} P \end{array}\right\} \text{richiama } P \end{array}$$

$$\begin{array}{l} P \longrightarrow \\ 3 \longrightarrow \end{array} \boxed{S_1^1} \longrightarrow \overline{P}$$

Si consideri il programma $\bar{P}$ nel caso generale, fissando la seconda variabile a un valore y:

$$\begin{array}{rlll} \bar{P} \equiv & \texttt{// input } x \texttt{ in } R_1 & & \text{codifica} \\ & \left.\begin{array}{l} R_0 \leftarrow R_0 + 1 \\ \quad \vdots \\ R_0 \leftarrow R_0 + 1 \end{array}\right\} y \text{ volte} & & \begin{array}{l} \longmapsto \quad 0 \\ \quad \vdots \\ \longmapsto \quad 0 \end{array} \\ & \left.\begin{array}{l} R_1 \leftarrow \langle R_1, R_0 \rangle \end{array}\right\} \text{input di } P & \longmapsto & \text{s} \\ & \left.\begin{array}{l} R_0 \leftarrow 0 \end{array}\right\} \text{pulisci } R_0 & \longmapsto & \text{t} \\ & \left.\begin{array}{l} R_0 \leftarrow R_2 + R_3 \end{array}\right\} \text{richiama } P & \longmapsto & \text{n} \end{array}$$

La codifica del programma $\bar{P}$ è

$$cod(\bar{P}) = \bar{n} = \langle \underbrace{0, \ldots, 0}_{y \text{ volte}}, s, t, n \rangle = S_1^1$$

con s codifica dell'istruzione che calcola la funzione coppia di Cantor e t codifica dell'istruzione di azzeramento.

Si può quindi notare che S_1^1 è

1. Una funzione totale
2. Programmabile

Di conseguenza

$$S_1^1 \in \mathcal{T}$$

□

In sintesi, per RAM, esiste una funzione S_1^1 ricorsiva totale che accetta come argomenti

- il codice n di un programma con 2 input
- un valore y a cui fissare il secondo input

e produce il codice $\bar{n} = S_1^1(n, y)$ di un programma che si comporta come n nel caso in cui il secondo input è fissato a y.

← (Il Teorema S_1^1 da solo garantisce una "algebra sui programi")

Questo teorema ha anche una forma generale S_n^m che riguarda i programmi a $m+n$ input in cui si prefissano n input e si lasciano variare i primi m.

Teorema 1.11.2 (S_n^m)**.** *Dato $\{\varphi_i\}$ RAM, esiste una funzione $S_n^m \in \mathcal{T}$ tale che*

$$\forall \in \mathbb{N}, \underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n \quad \varphi_k(\langle \underline{x}, \underline{y} \rangle) = \varphi_{S_n^m(k, \underline{y})}(\langle \underline{x} \rangle)$$

Ovvero, per ogni programma $k \in \mathbb{N}$ e ogni input $\underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n$ vale la proprietà specificata. Anche la dimostrazione è una semplice generalizzazione di quella di S_1^1.

Le tre caratteristiche identificate formano gli **assiomi di Rogers** (1953). Questi caratterizzano i sistemi di programmazione su cui ci concentreremo, chiamati *Sistemi di Programmazione Accettabili*. Questi assiomi non sono restrittivi: di fatto, tutti i modelli di calcolo ragionevoli sono SPA.

1.11.2 Compilatori tra SPA

Dati gli SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, un compilatore dal primo al secondo è una funzione $t : \mathbb{N} \to \mathbb{N}$ che soddisfa le proprietà di

1. **programmabilità**: esiste un programma che implementa t
2. **completezza**: t compila ogni $i \in \mathbb{N}$
3. **correttezza**: $\forall i \in \mathbb{N}$ vale $\varphi_i = \Psi_{t(i)}$

Visto quanto fatto fin'ora, si può dire che i primi due punti possono essere scritti come $t \in \mathcal{T}$.

Teorema 1.11.3. *Dati due SPA, esiste sempre un compilatore tra essi.*

Proof. Consideriamo $\{\varphi_i\}$ e $\{\Psi_i\}$ due SPA; valgono i tre assiomi di Rogers:

1. $\{\varphi_i\} = \mathscr{P}$
2. $\exists u : \varphi_u(\langle x, n\rangle) = \varphi_n(x)$
3. $\exists S_1^1 \in \mathcal{T} : \varphi_n(\langle x, y\rangle) = \varphi_{S_1^1(n,y)}(x)$

Voglio trovare un compilatore $t \in \mathcal{T}$ che sia corretto. Ma allora

$$\varphi_i(x) \overset{(2)}{=} \varphi_u(\langle x, i\rangle) \overset{(1)}{=} \Psi_e(\langle x, i\rangle) \overset{(3)}{=} \Psi_{S_1^1(e,i)}(x)$$

Ovvero, il compilatore cercato è la funzione $t(i) = S_1^1(e, i)$ per ogni $i \in \mathbb{N}$. Infatti:

- $t \in \mathcal{T}$ in quanto $S_1^1 \in \mathcal{T}$
- t corretto perché $\varphi_i = \Psi_{t(i)}$

□

Da notare che questo teorema è molto generale, specifica solamente che il compilatore esiste, non determina quale sia.

Corollario 1.11.1. *Dati gli SPA A, B, C $\exists$sempre un compilare da A a B scritto nel linguaggio C.*

Proof. Per il teorema precedente esiste un compilatore $t \in \mathcal{T}$ da A a B. C è un SPA, quindi contiene programmi per tutte le funzioni ricorsive parziali, dunque ne contiene uno anche per t, che è una funzione ricorsiva totale.

□

Nella pratica, vuol dire che per qualunque coppia di linguaggi, sarà sempre possibile scrivere un compilatore tra essi in un altro linguaggio arbitrario. Un risultato più forte del teorema precedente è invece dato dal **teorema di Rogers**.

Teorema 1.11.4 (Teorema di isomorfismo tra SPA)**.** *Dati due SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, esiste $t : \mathbb{N} \to \mathbb{N}$ tale che:*

1. $t \in \mathcal{T}$ ← *(Teorema Precedente)*
2. $\forall i \in \mathbb{N}$, $\varphi_i = \Psi_{t(i)}$ ← *(Teorema Precedente)*
3. *t è invertibile, quindi t^{-1} può essere visto come un decompilatore*

I primi due punti sono uguali al teorema precedente e dicono che il compilatore t è programmabile e completo (punto 1) e corretto (punto 2).

1.11.3 Teorema di Ricorsione

Il teorema di ricorsione è un risultato utile a rispondere ad alcuni quesiti sugli SPA.

Teorema 1.11.5. *Dato un SPA φ_i, per ogni $t : \mathbb{N} \to \mathbb{N}$ ricorsiva totale vale*

$$\exists n \in \mathbb{N} \mid \varphi_n = \varphi_{t(n)}$$

Per dare una chiave di lettura a questo teorema:

- consideriamo t come un programma che prende in input un programma n è lo trasforma in $t(n)$
- il teorema dice che, qualsiasi sia la natura di t, esisterà sempre almeno un programma il cui significato non sarà stravolto da t

Proof. Considerando un SPA $\{\varphi_i\}$, valgono i tre assiomi di Rogers. Per semplicità, scriveremo $\varphi_n(x, y)$ al posto di $\varphi_n(\langle x, y \rangle)$. Dobbiamo esibire, data una funzione t, uno specifico valore di n.

Partiamo mostrando che:

$$\varphi_{\varphi_i(i)}(x) \overset{(2)}{=} \varphi_{\varphi_u(i,i)}(x) \overset{(2)}{=} \varphi_u(x, \varphi_u(i, i)) \rightsquigarrow f(x, i) \in \mathscr{P}$$

Infatti, la funzione $f(x, i)$ è composizione di funzioni ricorsive parziali, quindi anch'essa lo è.

Continuiamo affermando che

$$f(x, i) \overset{(1)}{=} \varphi_e(x, i) \overset{(3)}{=} \varphi_{S_1^1(e,i)}(x)$$

Consideriamo ora la funzione $t\left(S_1^1(e, i)\right)$: essa è ricorsiva totale in i perché composizione di t e di S_1^1 ricorsive totali, quindi

$$\exists m \in \mathbb{N} \mid \varphi_m(i) = t\left(S_1^1(e, i)\right)$$

Abbiamo quindi mostrato che

$$(A) \qquad \varphi_{\varphi_i(i)}(x) = \varphi_{S_1^1(e,i)}(x)$$
$$(B) \qquad \varphi_m(i) = t\left(S_1^1(e, i)\right)$$

Fissiamo $n = S_1^1(e, m)$ e mostriamo che vale $\varphi_n = \varphi_{t(n)}$, ovvero il teorema di ricorsione

$$\varphi_n(x) \overset{\text{def}}{=} \varphi_{S_1^1(e,m)}(x) \overset{(A)}{=} \varphi_{\varphi_m(m)}(x)$$

$$\varphi_{t(n)}(x) \overset{\text{def}}{=} \varphi_{t\left(S_1^1(e,m)\right)}(x) \overset{(B)}{=} \varphi_{\varphi_m(m)}(x)$$

Ho ottenuto lo stesso risultato, quindi il teorema è verificato.

□

1.11.4 Due quesiti sugli SPA

Ci poniamo due quesiti riguardo agli SPA:

1. **Programmi auto-replicanti**: dato un SPA, esiste all'interno di esso un programma che stampa se stesso (il proprio listato)? Ovviamente senza aprire il file che contiene il listato.

   Questi programmi sono detti **Quine**, in onore del filosofo e logico `Willard Quine` che li descrisse per la prima volta. La risposta è positiva per molti linguaggi; ad esempio, in Python, il programma

```
a='a=%r;print(a%%a)';print(a%a)
```

stampa esattamente il proprio listato.

Per rispondere rigorosamente ambientiamo la domanda nel sistema di programmazione RAM, che diventa:

$$\exists j \in \mathbb{N} \mid \varphi_j(x) = j \text{ per ogni input } x \in \mathbb{N}?$$

2. **Compilatori completamente errati**: dati due SPA $\{\varphi_i\}$ e $\{\Psi_j\}$, esiste un compilatore completamente errato?

   Un compilatore dal primo al secondo SPA è una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che

   - $t \in \mathcal{T}$ programmabile e totale
   - $\forall i \in \mathbb{N}$, $\varphi_i = \Psi_{t(i)}$

   Invece, un **compilatore completamente errato** è una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che

   - $t \in \mathcal{T}$ programmabile e totale
   - $\forall i \in \mathbb{N}$, $\varphi_i \neq \Psi_{t(i)}$

Prima domanda: Quine

Consideriamo il programma RAM

$$P \equiv \left.\begin{array}{|l|}\hline R_0 \leftarrow R_0 + 1 \\ R_0 \leftarrow R_0 + 1 \\ \dots \\ R_0 \leftarrow R_0 + 1 \\ \hline \end{array}\right\} \text{ j times}$$

il quale ripete l'istruzione di incremento di R_0 un numero j di volte. La semantica di questo programma è esattamente j: infatti dopo la sua esecuzione avremo j nel registro di output R_0.

Calcoliamo la codifica di P come

$$cod(P) = \underbrace{\langle 0, \dots, 0 \rangle}_{j\text{-volte}} = Z(j) \in \mathcal{T}$$

Questa funzione è ricorsiva totale in quanto programmabile e totale, visto che sfrutta solo la funzione di Cantor. Vale quindi:

$$\varphi_{Z(j)}(x) = j$$

Per il teorema di ricorsione

$$\exists j \in \mathbb{N} \mid \varphi_j(x) = \varphi_{Z(j)}(x) = j$$

Quindi effettivamente esiste un programma j la cui semantica è proprio quella di stampare se stesso.

La risposta alla prima domanda è quindi *Sì!* per RAM, ma lo è in generale per tutti i SPA che ammettono una codifica per i propri programmi.

Seconda domanda: compilatori completamente errati

Supponendo di avere una funzione $t \in \mathcal{T}$ che "*maltratta*" i programmi. La semantica del programma "*maltrattato*" $t(i)$:

$$(*) \qquad \Psi_{t(i)}(x) \overset{(2)}{=} \Psi_u(x, t(i)) \overset{(1)}{=} \varphi_e(x, t(i)) \overset{(3)}{=} \varphi_{S^1_1(e,t(i))}(x)$$

Chiamiamo $g(i)$ la funzione $S_1^1(e, t(i))$ che dipende solo da i, essendo e un programma fissato. Notiamo come questa funzione sia composizione di funzioni ricorsive totali, ovvero $t(i)$ per ipotesi e S_1^1 per definizione, quindi anch'essa è ricorsiva totale.

Per il teorema di ricorsione

$$(**) \qquad \exists i \in \mathbb{N} \mid \varphi_i = \varphi_{g(i)}$$

Unendo $(*)$ e $(**)$ otteniamo

$$\exists i \in \mathbb{N} \mid \Psi_{t(i)} \overset{(*)}{=} \varphi_{g(i)} \overset{(**)}{=} \varphi_i \quad \forall t \in \mathcal{T}$$

Di conseguenza, la risposta alla seconda domanda è *No*!.

1.11.5 Equazioni su SPA

Strategia

La portata del teorema di ricorsione è molto ampia: permette di risolvere **equazioni su SPA** in cui si chiede l'esistenza di certi programmi in SPA.

Per esempio, dato un SPA $\{\varphi_i\}$ ci chiediamo se

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x\left(n + \varphi_{\varphi_n(0)}(x)\right)$$

La strategia da seguire per risolvere questo tipo di richieste è analoga a quella seguita per la dimostrazione del teorema di ricorsione, riassumibile in:

1. Trasformare il membro di destra dell'equazione in una funzione $f(x, n)$
2. Mostrare che $f(x, n)$ è ricorsiva parziale e quindi che $f(x, n) = \varphi_e(x, n)$
3. L'equazione iniziale diventa $\varphi_n(x) = \varphi_e(x, n) = \varphi_{S_1^1(e,n)}(x)$
4. So che $S_1^1(e, n)$ è una funzione ricorsiva totale che dipende da n
5. La domanda iniziale è diventata $\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{S_1^1(e,n)}(x)$?
6. La risposta è *Sì*, per il teorema di ricorsione

Riprendendo l'esempio e cominciando con trasformare la parte di destra:

$$\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} \varphi_x\left(n + \varphi_{\varphi_u(0,n)}(x)\right) \\
&\overset{(2)}{=} \varphi_x\left(n + \varphi_u\left(x, \varphi_u(0, n)\right)\right) \\
&\overset{(2)}{=} \varphi_u\left(n + \varphi_u\left(x, \varphi_u(0, n)\right), x\right) \\
&= f(x, n) \in \mathscr{P}
\end{aligned}$$

L'ultimo passaggio è vero perché $\varphi_u\left(n + \varphi_u\left(x, \varphi_u(0, n)\right), x\right)$ compone solamente funzioni ricorsive parziali quali somma e interprete universale. Di conseguenza, esiste un programma e che calcola la funzione $f(x, n)$.

Continuando, l'equazione si può riscrivere come

$$\begin{aligned}
\varphi_n(x) &= f(x, n) \\
&\overset{(1)}{=} \varphi_e(x, n) \\
&\overset{(3)}{=} \varphi_{S_1^1(e,n)}(x)
\end{aligned}$$

con $S_1^1(e, n) \in \mathcal{T}$ per l'assioma 3.

Per il teorema di ricorsione possiamo concludere che

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{S_1^1(e,n)}(x) = \varphi_x\left(n + \varphi_{\varphi_u(0,n)}(x)\right)$$

Esercizi

In tutti gli esercizi viene dato un SPA $\{\varphi_i\}$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(n) + \varphi_{\varphi_x(n)}(n)?$$

$$\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} \varphi_u(n, x) + \varphi_{\varphi_u(n,x)}(n) \\
&\overset{(2)}{=} \varphi_u(n, x) + \varphi_u(n, \varphi_u(n, x)) \\
&= f(x, n) \\
&\overset{(1)}{=} \varphi_e(x, n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}$$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = n^x + (\varphi_x(x))^2?$$

$$\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} n^x + (\varphi_u(x, x))^2 \\
&= f(x, n) \\
&\overset{(1)}{=} \varphi_e(x, n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}$$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x\left(\langle n, \varphi_x(1)\rangle\right)?$$

$$\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} \varphi_u\left(\langle n, \varphi_u(1, x)\rangle, x\right) \\
&= f(x, n) \\
&\overset{(1)}{=} \varphi_e(x, n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}$$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{\varphi_x(\sin(n))}(\text{des}(n))?$$

$$\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} \varphi_{\varphi_u(\sin(n),x)}(\text{des}(n)) \\
&\overset{(2)}{=} \varphi_u(\text{des}(n), \varphi_u(\sin(n), x)) \\
&= f(x, n) \\
&\overset{(1)}{=} \varphi_e(x, n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}$$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = n^x + (\varphi_x(x))^2?$$

$$
\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} n^x + (\varphi_u(x,x))^2 \\
&= f(x,n) \\
&\overset{(1)}{=} \varphi_e(x,n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}
$$

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(n+2) + \left(\varphi_{\varphi_x(n)}(n+3)\right)^2 ?$$

$$
\begin{aligned}
\varphi_n(x) &\overset{(2)}{=} \varphi_u(n+2,x) + \left(\varphi_{\varphi_u(n,x)}(n+3)\right)^2 \\
&\overset{(2)}{=} \varphi_u(n+2,x) + \left(\varphi_u\left(n+3, \varphi_u(n,x)\right)\right)^2 \\
&= f(x,n) \\
&\overset{(1)}{=} \varphi_e(x,n) \overset{(3)}{=} \varphi_{S_1^1(e,n)}(x) \\
&\overset{\text{TR}}{=} \text{OK}
\end{aligned}
$$

1.12 Problemi di Decisione

Un **problema di decisione** è una domanda a cui rispondere *Sì* o *No*. Sono costituiti da tre elementi principali:

- **Nome**: del problema
- **Istanza**: dominio degli oggetti che verranno considerati
- **Domanda**: proprietà che gli oggetti del dominio possono soddisfare o meno. Dato un oggetto del dominio, la risposta sarà *Sì* se soddisfa la proprietà, *No* altrimenti. In altre parole, è la specifica del problema di decisione

Una definizione più formale:

- **Nome** Π
- **Istanza** $x \in D$ input
- **Domanda** $p(x)$ proprietà che $x \in D$ può soddisfare o meno

Non bisogna esibire una struttura come risultato (cosa che puo' accadere nei problemi di ricerca od ottimizzazione), ma solo una risposta *Sì*/*No*.

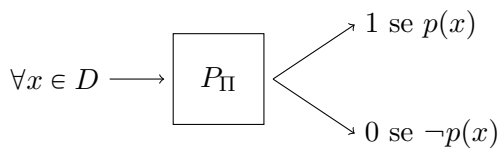
Esempio: Un esempio di problema di decisione è quello di stabilire se un grafo ammette un circuito hamiltoniano:

- Nome: Circuito hamiltoniano
- Istanza: grafo $G = (V, E)$
- Domanda: $\exists \gamma$ circuito hamiltoniano nel grafo G?

← (altri esempi nelle slide)

1.12.1 Decidibilità

Sia Π problema di decisione con istanza $x \in D$ e domanda $p(x)$. Π è **decidibile** se e solo se esiste un programma P_Π tale che



Allo stesso modo, possiamo associare a Π la sua **funzione soluzione**:

$$\Phi_\Pi : D \to \{0,1\} \text{ t.c. } \Phi_\Pi(x) = \begin{cases} 1 & \text{se } p(x) \\ 0 & \text{se } \neg p(x) \end{cases}$$

Questa funzione deve essere programmabile e deve terminare sempre; ma allora $\Phi_\Pi \in \mathcal{T}$.

I due approcci per **definire la decidibilità** sono equivalenti:

- il programma P_Π calcola Φ_Π, quindi $\Phi_\Pi \in \mathcal{T}$
- se $\Phi_\Pi \in \mathcal{T}$, allora esiste un programma che la calcola e che ha il comportamento di P_Π

Quindi, definire la decidibilità partendo da un programma o da una funzione ricorsiva totale è indifferente: una definizione implica l'altra.

Si può sfruttare questo fatto per sviluppare due **tecniche di risoluzione** del problema di decidibilità:

- Esibire un **algoritmo di soluzione** P_Π (anche inefficiente, basta che esista)
- Mostrare che Φ_Π è **ricorsiva totale**

Esempio: Sull'esempio precedente, dovendo visitare ogni nodo una e una sola volta, il circuito genera una permutazione dei vertici in V. L'algoritmo di soluzione deve:

1. generare l'insieme P di tutte le permutazioni di V
2. data la permutazione $p_i \in P$, se è un c.h. rispondo *Sì*
3. se nessuna permutazione $p_i \in P$ è un c.h. rispondo *No*

← (Guarda slide per robe in piu')

1.12.2 Problemi indecidibili

Esistono dei **problemi indecidibili**? *Sì*: se esistono programmi che non so come scrivere, allora esistono problemi per i quali non posso scrivere dei programmi che li risolvano.

Problema dell'arresto ristretto

Il **problema dell'arresto ristretto** per un programma P è un esempio di problema indecidibile. Fissato un programma P, il problema è il seguente:

- Nome: AR_P
- Istanza: $x \in \mathbb{N}$
- Domanda: $\varphi_P(x) \downarrow$?

In altre parole, ci chiediamo se P termina su input x. La risposta dipende da P, che può essere decidibile o meno. Per alcuni programmi si può trovare risposta ed è quindi decidibile, tuttavia esistono dei programmi dove AR_P è indecidibile.

Consideriamo ora il programma $\hat{P}$:

$$\begin{aligned} \hat{P} \equiv\ & \text{input}(x) \\ & Z := U(x,x) \\ & \text{output}(Z) \end{aligned}$$

Dove U è l'interprete universale tale che

$$\varphi_U(x,n) = \varphi_n(x)$$

E definiamo di conseguenza il problema $\mathrm{AR}_{\hat{P}}$:

$$\varphi_{\hat{P}}(x) = \varphi_U(x, x) = \varphi_x(x)$$

- Nome $\mathrm{AR}_{\hat{P}}$
- Istanza $x \in \mathbb{N}$
- Domanda: $\varphi_{\hat{P}}(x) = \varphi_x(x) \downarrow$?

Abbiamo, quindi, un programma x che lavora su un altro programma (*se stesso* in questo caso). Nulla di strano: compilatori, debugger, interpreti sono programmi che lavorano su programmi.

La domanda è se φ_x su input x termina. Il programma P non è fissato e dipende dall'input, essendo x sia input che programma.

Teorema 1.12.1 (Indecidibilità di $\mathrm{AR}_{\hat{P}}$)**.** *$AR_{\hat{P}}$ è indecidibile.*

Proof. Per assurdo, assumiamo $\mathrm{AR}_{\hat{P}}$ decidibile. Dunque esiste

$$\Phi_{\mathrm{AR}_{\hat{P}}}(x) = \begin{cases} 1 & \text{se } \varphi_{\hat{P}}(x) = \varphi_x(x) \downarrow \\ 0 & \text{se } \varphi_{\hat{P}}(x) = \varphi_x(x) \uparrow \end{cases} \in \mathcal{T}$$

calcolabile da un programma che termina sempre. Visto che $\Phi_{\mathrm{AR}_{\hat{P}}} \in \mathcal{T}$ anche la funzione

$$f(x) = \begin{cases} 0 & \text{se } \Phi_{\mathrm{AR}_{\hat{P}}}(x) = 0 \equiv \varphi_x(x) \uparrow \\ \varphi_x(x) + 1 & \text{se } \Phi_{\mathrm{AR}_{\hat{P}}}(x) = 1 \equiv \varphi_x(x) \downarrow \end{cases}$$

è ricorsiva totale, si può avere un programma che calcola esattamente la funzione $f(x)$.

Sia $\alpha \in \mathbb{N}$ la codifica del programma A, tale che $\varphi_\alpha = f$. Valutiamo φ_α in α: ← (Guarda Slide 15:11 per vedere A)

$$\varphi_\alpha(\alpha) = \begin{cases} 0 & \text{se } \varphi_\alpha(\alpha) \uparrow \\ \varphi_\alpha(\alpha) + 1 & \text{se } \varphi_\alpha(\alpha) \downarrow \end{cases}$$

Tale funzione non può esistere, infatti:

- nel primo caso ho $\varphi_\alpha(\alpha) = 0$ se $\varphi_\alpha(\alpha) \uparrow$, ma è una contraddizione
- nel secondo caso ho $\varphi_\alpha(\alpha) = \varphi(\alpha) + 1$ se termina, ma è una contraddizione perché questa relazione non vale per nessun naturale

Siamo a un assurdo, quindi $\mathrm{AR}_{\hat{P}}$ è indecidibile.

□

Problema dell'arresto

La versione generale del problema dell'arresto ristretto è il **problema dell'arresto**, posto nel 1936 da Alan Turing.

Teorema 1.12.2 (Indecidibilità di AR)**.** *Dati $x, y \in \mathbb{N}$ rispettivamente un dato e un programma, il problema dell'arresto AR con domanda $\varphi_y(x) \downarrow$ è indecidibile.*

Proof. Assumiamo per assurdo che AR sia decidibile, allora esiste una funzione soluzione

$$\Phi_{\mathrm{AR}}(x, y) = \begin{cases} 0 & \text{se } \varphi_y(x) \uparrow \\ 1 & \text{se } \varphi_y(x) \downarrow \end{cases}$$

Valutando il caso in cui $x = y$

$$\Phi_{\mathrm{AR}}(x,x) = \begin{cases} 0 & \text{se } \varphi_x(x) \uparrow \\ 1 & \text{se } \varphi_x(x) \downarrow \end{cases}$$

Possiamo notare che $\Phi_{\mathrm{AR}}(x,x) = \Phi_{\mathrm{AR}_{\hat{P}}}(x)$, quindi nel caso $x = y$ il problema AR coincide con il $\mathrm{AR}_{\hat{P}}$. Il risultato del teorema precedente dimostra che $\mathrm{AR}_{\hat{P}}$ è indecidibile, quindi anche AR è indecidibile. □

1.13 Riconoscibilità automatica di insiemi

Vogliamo fornire un *livello di risoluzione* dei problemi, stabilendo se un problema:

- può essere risolto
- non può essere risolto completamente
- non può essere risolto

Costruiamo un programma che classifichi gli elementi di un insieme, per quindi dire se un certo numero naturale appartiene o meno all'insieme.

Un insieme $A \subseteq \mathbb{N}$ è riconoscibile automaticamente se esiste un programma P_A che classifica correttamente ogni elemento di $\mathbb{N}$ come appartenente o meno ad A, ovvero:

$$x \in \mathbb{N} \rightsquigarrow P_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Il programma P deve essere:

- **corretto**: classifica correttamente gli elementi che riceve in input
- **completo**: classifica tutti gli elementi di $\mathbb{N}$, nessuno escluso

Tutti gli insiemi sono automaticamente riconoscibili? Quali insiemi sono automaticamente riconoscibili?

Sicuramente non tutti gli insiemi sono automaticamente riconoscibili, questo grazie al concetto di *cardinalità*, infatti:

- i sottoinsiemi di $\mathbb{N}$ sono *densi quanto* $\mathbb{R}$
- non esistono $|\mathbb{R}|$ programmi

Di conseguenza devono esistere insiemi non automaticamente riconoscibili.

1.13.1 Insiemi Ricorsivi

Un insieme $A \subseteq \mathbb{N}$ è un **insieme ricorsivo** se esiste un programma P_A che si arresta su ogni input classificando correttamente gli elementi di $\mathbb{N}$ in base alla loro appartenenza o meno ad A.

Equivalentemente, ricordando che la funzione caratteristica di $A \subseteq \mathbb{N}$ è la funzione $\mathcal{X}_A : \mathbb{N} \to \{0,1\}$ tale che

$$\mathcal{X}_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Diciamo che l'insieme A è ricorsivo se e solo se

$$\mathcal{X}_A \in \mathcal{T}$$

Che $\mathcal{X}_A$ sia **totale** è banale: tutte le funzioni caratteristiche devono essere definite su tutto $\mathbb{N}$. Il problema risiede nella **calcolabilità** di queste funzioni.

Le due definizioni date sono equivalenti:

- il programma P_A implementa $\mathcal{X}_A$, quindi $\mathcal{X}_A \in \mathcal{T}$ perché esiste un programma che la calcola
- $\mathcal{X}_A \in \mathcal{T}$ quindi esiste un programma P_A che la implementa e soddisfa la definizione sopra

Ricorsivo vs Decidibile

Spesso si dice che *un insieme ricorsivo è un insieme decidibile*, ma è solo abuso di notazione. Questo è dovuto al fatto che a ogni insieme $A \subseteq \mathbb{N}$ possiamo associare il suo **problema di riconoscimento**, definito come:

- Nome: RIC_A
- Istanza: $x \in \mathbb{N}$
- Domanda: $x \in A$?

La sua funzione soluzione

$$\Psi_{\mathrm{RIC}_A} : \mathbb{N} \to \{0, 1\}$$

è tale che

$$\Psi_{\mathrm{RIC}_A}(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Si può notare che la semantica del problema è proprio la funzione caratteristica, quindi $\Psi_{\mathrm{RIC}_A} = \mathcal{X}_A$. Se A è ricorsivo, allora la sua funzione caratteristica è ricorsiva totale, ma lo sarà anche la funzione soluzione Ψ e, di conseguenza, RIC_A è decidibile.

Decidibile vs Ricorsivo

Simmetricamente, sempre con abuso di notazione, si dice che *un problema di decisione è un problema ricorsivo*. Questo perché a ogni problema di decisione Π possiamo associare A_Π insieme delle sue **istanze con risposta positiva**.

Dato il problema:

- Nome: Π
- Istanza: $x \in D$
- Domanda: $p(x)$?

Definiamo

$$A_\Pi = \{x \in D \mid \Psi_\Pi(x) = 1\} \text{ con } \Psi_\Pi(x) = 1 \equiv p(x)$$

insieme delle istanze con risposta positiva di Π. Si può notare che, se Π è decidibile, allora $\Psi_\Pi \in \mathcal{T}$, quindi esiste un programma che calcoli questa funzione. La funzione in questione è quella che riconosce automaticamente l'insieme A_Π, quindi A_Π è ricorsivo.

1.13.2 Insiemi Non Ricorsivi

Per trovare degli insiemi non ricorsivi si può cercare nei problemi di decisione non decidibili. L'unico problema di decisione non decidibile visto fin'ora è il **problema dell'arresto ristretto** $\mathrm{AR}_{\hat{P}}$. Definizione:

- Nome: $\mathrm{AR}_{\hat{P}}$
- Istanza: $x \in \mathbb{N}$
- Domanda: $\varphi_{\hat{P}}(x) = \varphi_x(x) \downarrow$?

Definiamo l'insieme delle istanze con risposta positiva di $\mathrm{AR}_{\hat{P}}$:

$$A = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\}$$

Questo non può essere ricorsivo: se lo fosse, avrei un programma ricorsivo totale che mi classifica correttamente se x appartiene o meno ad A, ma già dimostrato che il problema dell'arresto ristretto non è decidibile, quindi A non è ricorsivo.

1.13.3 Relazioni Ricorsive

Una relazione $R \subseteq \mathbb{N} \times \mathbb{N}$ è una **relazione ricorsiva** se e solo se l'insieme R è ricorsivo, ovvero:

- la sua funzione caratteristica $\mathcal{X}_R$ è tale che $\mathcal{X}_R \in \mathcal{T}$, oppure
- esiste un programma P_R che, presi in ingresso $x, y \in \mathbb{N}$ restituisce 1 se (xRy), 0 altrimenti

Un'importante relazione ricorsiva è

$$R_P = \big\{(x, y) \in \mathbb{N}^2 \mid P \text{ su input } x \text{ termina in } y \text{ passi}\big\}$$

Simile al problema dell'arresto, ma non chiede solo la terminazione, specifica un numero di passi y in cui deve terminare. Questa relazione è ricorsiva e per dimostrarlo costruiamo un programma che classifica R_P usando:

- U interprete universale
- **clock** per contare i passi di interpretazione che si incrementa di 1 ad ogni passo di interpretazione
- **check del clock** per controllare l'arrivo alla quota y (time-out sul tempo di interpretazione)

Definiamo quindi il programma

$$\tilde{U} = U + \text{ clock } + \text{ check clock}$$

tale che:

$\tilde{U} \equiv \text{input}(x, y)$
$U(P, x)+$ clock
`Ad ogni` passo di $U(P, x)$ `:`
    `if` clock $> y$ `then`
        output(0)
    clock++
output(clock$==y$)

In breve: si aggiunge una variabile che conta i passi, se l'esecuzione supera y restituisce 0, altrimenti 1. Nel sistema RAM, ad esempio, per capire se l'output è stato generato o meno osservo se il PC, contenuto nel registro L, è uguale a 0.

Riprendendo il problema dell'arresto ristretto: *come si può esprimere* $A = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\}$ *attraverso la relazione ricorsiva* $R_{\hat{P}}$*?*

Possiamo definire l'insieme

$$B = \big\{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \ \big(xR_{\hat{P}}y\big)\big\}$$

Si può notare come $A = B$:

- $A \subseteq B$: se $x \in A$ il programma codificato con x su input x termina in un certo numero di passi; chiamiamo y il numero di passi; $\hat{P}(x)$ termina in y passi, ma allora $xR_{\hat{P}}y$, quindi $x \in B$
- $B \subseteq A$: se $x \in B$ esiste y tale che $xR_{\hat{P}}y$, quindi $\hat{P}(x)$ termina in y passi, ma allora il programma $\hat{P} = x$ su input x termina, quindi $x \in A$

1.13.4 Insiemi Ricorsivamente Numerabili

Un insieme $A \subseteq \mathbb{N}$ è **ricorsivamente numerabile** se è **automaticamente listabile**: esiste una routine F che, su input $i \in \mathbb{N}$, fornisce output $F(i)$ come l'i-esimo elemento di A.

Programma P che lista gli elementi di A:

```
i := 0
while True
    output(F(i))
    i := i + 1
```

Per alcuni insiemi non è possibile riconoscere tutti gli elementi che gli appartengono, ma può essere che si riesca a listarlo (non sempre).

Se il meglio che posso fare per avere l'insieme A è listarlo con il programma P, *come posso scrivere un algoritmo che "tenta di riconoscere" A?*

```
input(x);
i := 0;
while F(i) ≠ x
    i := i + 1;
output(1);
```

Programma di **massimo riconoscimento**. Preso in input x, il programma restituisce 1 se $x \in A$ o va in loop se $x \notin A$.

Come viene riconosciuto A?

$$x \in \mathbb{N} \rightsquigarrow P(x) = \begin{cases} 1 & \text{se } x \in A \\ \text{loop} & \text{se } x \notin A \end{cases}$$

Vista la natura di questa funzione, gli insiemi ricorsivamente numerabili sono anche detti **insiemi parzialmente decidibili/riconoscibili** o **semidecidibili**.

Definizione Formale

L'insieme $A \subseteq \mathbb{N}$ è **ricorsivamente numerabile** se e solo se:

- $A = \varnothing$ oppure
- $A = \mathrm{Im}_f$, con $f : \mathbb{N} \to \mathbb{N} \in \mathcal{T}$, ovvero $A = \{f(0), f(1), \dots\}$

Visto che f è ricorsiva totale esiste un programma/routine F che la implementa e che usiamo per il parziale riconoscimento di A: questo programma, se $x \in A$ restituirà 1, altrimenti entrerà in loop.

Si può comparare a un *libro con infinite pagine*, su ognuna delle quali compare un elemento di A; il programma non fa altro che sfogliare le pagine.

← (Guarda Slide 16:10 se indeciso)

Caratterizzazioni

Teorema 1.13.1. *Le seguenti definizioni sono equivalenti:*

1. *A è ricorsivamente numerabile, $A = Im_f$ con $f \in \mathcal{T}$ funzione ricorsiva totale*
2. *$A = Dom_f$ con $f \in \mathcal{P}$ funzione ricorsiva parziale*
3. *Esiste una relazione $R \subseteq \mathbb{N}^2$ ricorsiva tale che $A = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \mid (x, y) \in R\}$*

Proof. Per dimostrare questi teoremi dimostriamo che $1 \implies 2 \implies 3 \implies 1$, creando un'implicazione ciclica.

$1 \implies 2$: Sappiamo che $A = \text{Im}_f$, con $f \in \mathcal{T}$, è ricorsivamente numerabile, quindi esistono la sua routine di calcolo f è il suo algoritmo parziale di riconoscimento P, definiti in precedenza. Vista la definizione di P, si ha che

$$\varphi_P(x) = \begin{cases} 1 & \text{se } x \in A \\ \perp & \text{se } x \notin A \end{cases}$$

ma allora $A = \text{Dom}_{\varphi_P}$: il dominio è l'insieme dei valori dove la funzione è definita, in questo caso proprio l'insieme A. Inoltre $\varphi_P \in \mathcal{P}$ perché abbiamo mostrato che esiste un programma P che la calcola.

$2 \implies 3$: Sappiamo che $A = \text{Dom}_f$, con $f \in \mathcal{P}$, quindi esiste un programma P tale che $\varphi_P = f$. Considerando la relazione

$$R_P = \big\{(x, y) \in \mathbb{N}^2 \mid P \text{ su input } x \text{ termina in } y \text{ passi}\big\}$$

che abbiamo dimostrato essere ricorsiva. Definiamo

$$B = \{x \in \mathbb{N} \mid \exists y \text{ t.c. } (x, y) \in \mathbb{R}_P\}$$

Dimostriamo che $A = B$, infatti:

- $A \subseteq B$: se $x \in A$ allora su input x il programma P termina in un certo numero di passi y, visto che x è nel "dominio" di tale programma. Vale allora $(x, y) \in R_P$ e quindi $x \in B$
- $B \subseteq A$: se $x \in B$ allora per un certo y ho $(x, y) \in R_P$, quindi P su input x termina in y passi, ma visto che $\varphi_P(x) \downarrow$, allora x rientra nel dominio di $f = \varphi_P$, quindi $x \in A$

$3 \implies 1$: Sappiamo che $A = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \mid (x, y) \in R\}$, con R relazione ricorsiva.

Assumiamo che $A \neq \varnothing$ e scegliamo $a \in A$, sfruttando l'assioma della scelta. Definiamo ora la funzione $t : \mathbb{N} \to \mathbb{N}$ come

$$t(n) = \begin{cases} \sin(n) & \text{se } (\sin(n), \text{des}(n)) \in R \\ a & \text{altrimenti} \end{cases}$$

Visto che R è una relazione ricorsiva esiste un programma P_R che categorizza ogni numero naturale, ma allora la funzione t è ricorsiva totale. Infatti possiamo scrivere il programma

```
input(n);
x := sin(n);
y := des(n);
if P_R(x, y) = 1
    output(x);
else
    output(a);
```

Il quale implementa la funzione t, quindi $\varphi_P = t$.

Dimostriamo che $A = \text{Im}_t$. Infatti:

- $A \subseteq \text{Im}_t$: se $x \in A$ allora $(x, y) \in R$, ma allora $t(\langle x, y \rangle) = x$, quindi $x \in \text{Im}_t$
- $\text{Im}_t \subseteq A$: se $x \in \text{Im}_t$ allora
  - se $x = a$ per l'assioma della scelta $a \in A$, quindi $x \in A$
  - se $x = \sin(n)$ con $n = \langle x, y \rangle$ per qualche y tale che $(x, y) \in R$ allora $x \in A$ per definizione di A

□

Grazie a questo teorema abbiamo tre caratterizzazioni per gli insiemi ricorsivamente numerabili e possiamo sfruttare la formulazione che risulta più comoda.

Per usare il punto 2, in ordine:

1. scrivo un programma P che restituisce 1 su input $x \in \mathbb{N}$, altrimenti va in loop se $x \notin A$

$$P(x) = \begin{cases} 1 & \text{se } x \in A \\ \bot & \text{se } x \notin A \end{cases}$$

2. la semantica di P è quindi tale che

$$\varphi_P(x) = \begin{cases} 1 & \text{se } x \in A \\ \bot & \text{se } x \notin A \end{cases}$$

3. la funzione calcolata è tale che

$$\varphi_P \in \mathscr{P}$$

   visto che il programma che la calcola è proprio P, mentre l'insieme A è tale che

$$A = \text{Dom}_{\varphi_P}$$

4. A è ricorsivamente numerabile per il punto 2.

1.13.5 Insiemi Ricorsivamente numerabili ma non ricorsivi

Un esempio di insieme che non è ricorsivo, ma è ricorsivamente numerabile lo si può identificare dal problema dell'arresto ristretto. Infatti, l'insieme:

$$A = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\}$$

non è ricorsivo, altrimenti il problema dell'arresto sarebbe decidibile. Tuttavia, questo insieme è ricorsivamente numerabile: infatti il programma P:

```
input(x)
U(x, x)
output(1)
```

decide parzialmente A. Come si può notare, se $x \in A$, allora $\varphi_x(x) \downarrow$, ovvero l'interprete universale U termina, e il programma P restituisce 1, altrimenti non termina.

Di conseguenza:

$$\varphi_P(x) = \begin{cases} 1 & \text{se } \varphi_U(x,x) = \varphi_x(x) \downarrow \quad x \in A \\ \bot & \text{altrimenti} \qquad\qquad\qquad\quad x \in A \end{cases}$$

Dato che $A = \text{Dom}_{\varphi_P \in \mathscr{P}}$ si può applicare la seconda caratterizzazione fornita precedentemente per dimostrare che l'insieme A è un insieme ricorsivamente numerabile.

Alternativamente, possiamo dire che:

$$A = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\} = \left\{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \text{ t.c. } (x,y) \in R_{\hat{P}}\right\}$$

con

$$R_{\hat{P}} = \left\{(x,y) \mid \hat{P} \text{ su input } x \text{ termina entro } y \text{ passi}\right\}$$

relazione ricorsiva. Qui è possibile sfruttare la terza caratterizzazione degli insiemi ricorsivamente numerabili.

Teorema 1.13.2. *Se $A \subseteq \mathbb{N}$ è ricorsivo, allora è ricorsivamente numerabile.*

Proof. Se A ricorsivo, esiste un programma P_A in grado di riconoscerlo, ovvero un programma che restituisce 1 se $x \in A$, 0 altrimenti.

Il programma P è del tipo:

```
P
input(x)
if P_A(x) = 1
    output(1)
else
    while(1 > 0)
```

La semantica di questo programma è

$$\varphi_{P_A}(x) = \begin{cases} 1 & \text{se } x \in A \\ \perp & \text{se } x \notin A \end{cases}$$

ma allora A è il dominio di una funzione ricorsiva parziale, quindi A è ricorsivamente numerabile per la seconda caratterizzazione.

□

Poco fa abbiamo mostrato come $A = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\}$ sia un insieme ricorsivamente numerabile ma non ricorsivo, ma allora vale che

$$\text{Ricorsivi} \subset \text{Ricorsivamente numerabili}$$

1.13.6 Chiusura degli insiemi ricorsivi

Cerchiamo di sfruttare l'operazione di complemento di insiemi sui ricorsivamente numerabili per vedere di che natura è l'insieme

$$A^C = \{x \in \mathbb{N} \mid \varphi_x(x) \uparrow\}$$

Teorema 1.13.3. *La classe degli insiemi ricorsivi è un'Algebra di Boole, ovvero è chiusa per complemento, intersezione e unione.*

Proof. Siano A, B due insiemi ricorsivi. Allora esistono dei programmi P_A, P_B che li riconoscono; equivalentemente, esistono $\mathcal{X}_A, \mathcal{X}_B \in \mathcal{T}$.

Dimostrare che le operazioni di unione, intersezione e complemento sono implementabili da programmi che terminano sempre è facile. Di conseguenza

$$A \cup B, A \cap B, A^C$$

sono ricorsive.

Vediamo i tre programmi:

- Complemento

```
input(x)
output(1 ∸ P_A(x))
```

- Intersezione

```
input(x)
output(min(P_A(x), P_B(x)))
```

- Unione

```
input(x)
output(max(P_A(x), P_B(x)))
```

Allo stesso modo si possono trovare le funzioni caratteristiche delle tre operazioni:

- $\mathcal{X}_{A^C}(x) = 1 \dot{-} \mathcal{X}_A(x)$
- $\mathcal{X}_{A\cap B} = \mathcal{X}_A(x) \cdot \mathcal{X}_B(x)$
- $\mathcal{X}_{A\cup B} = 1 \dot{-} (1 \dot{-} \mathcal{X}_A(x))(1 \dot{-} \mathcal{X}_B(x))$

□

Ora un risultato importante riguardante il complemento dell'insieme dell'arresto A^C definito prima.

Teorema 1.13.4. *A^C non è ricorsivo.*

Proof. Se A^C fosse ricorsivo, per la proprietà di chiusura dimostrata nel teorema precedente, avremmo

$$\left(A^C\right)^C = A$$

Quindi A sarebbe ricorsivo, il che è assurdo. □

Ricapitolando:

- $A = \{x : \varphi_x(x) \downarrow\}$ ricorsivamente numerabile, ma non ricorsivo
- $A^C = \{x : \varphi_x(x) \uparrow\}$ non ricorsivo

Ma A^C può essere ricorsivamente numerabile?

Teorema 1.13.5. *Se A è ricorsivamente numerabile e A^C è ricorsivamente numerabile, allora A è ricorsivo.*

Proof. **Informale:** Con A e A^C ricorsivamente numerabili, esistono due libri con infinite pagine, su ognuna delle quali compare un elemento di A (primo libro) e un elemento di A^C (secondo libro).

Per decidere l'appartenenza di x ad A possiamo usare il procedimento:

1. input(x)
2. apriamo i due libri alla prima pagina:
    - se x compare nel libro di A, stampa 1
    - se x compare nel libro di A^C, stampa 0
    - se x non compare su nessuna delle due pagine, volta pagina e ricomincia

Questo algoritmo termina sempre dato che x o sta in A o sta in A^C, quindi prima o poi verrà trovato su uno dei due libri. Ma allora questo algoritmo riconosce A, quindi A è ricorsivo.

Formale: Con A e A^C ricorsivamente numerabili, esistono $f, g \in \mathcal{T}$ tali che $A = \mathrm{Im}_f \wedge A^C = \mathrm{Im}_g$. Sia f implementata nel programma F e g dal programma G. Il seguente programma riconosce A:

```
input(x)
i := 0
while True
    if F(i) = x
        output(1)
    if G(i) = x
        output(0)
    i := i + 1
```

Questo algoritmo termina per ogni input, in quanto $x \in A$ o $x \in A^C$. Possiamo concludere che l'insieme A è ricorsivo.

□

Si può concludere che A^C **non** può essere ricorsivamente numerabile.

In generale, questo teorema fornisce uno strumento che permette di studiare le caratteristiche della riconoscibilità di un insieme A:

- se A non è ricorsivo, potrebbe essere ricorsivamente numerabile
- se non riesco a mostrarlo, provo a studiare A^C
- se A^C è ricorsivamente numerabile, allora per il teorema possiamo concludere che A non è ricorsivamente numerabile

1.13.7 Chiusura degli insiemi ricorsivamente numerabili

Teorema 1.13.6. *La classe degli insiemi ricorsivamente numerabili è chiusa per unione e intersezione, ma non per complemento.*

Proof. Per complemento, abbiamo mostrato che $A = \{x : \varphi_x(x) \downarrow\}$ è ricorsivamente numerabile, mentre $A^C = \{x : \varphi_x(x) \uparrow\}$ non lo è.

Siano A, B insiemi ricorsivamente numerabili. Esistono quindi $f, g \in \mathcal{T}$ tali che $A = \text{Im}_f \wedge B = \text{Im}_g$. Sia f implementata da F e g implementata da G. Siano

```
P_i
input(x)
i := 0
while F(i) ≠ x do
    i++
end
i := 0
while G(i) ≠ x do
    i++
end
output(1)
```

```
P_u
input(x)
i := 0
while True do
    if F(i) = x then
        output(1)
    end
    if G(i) = x then
        output(1)
    end
    i++
end
```

i due programmi che calcolano rispettivamente $A \cap B$ e $A \cup B$. Le loro semantiche sono

$$\varphi_{P_i} = \begin{cases} 1 & \text{se } x \in A \cap B \\ \perp & \text{altrimenti} \end{cases} \qquad \varphi_{P_u} = \begin{cases} 1 & \text{se } x \in A \cup B \\ \perp & \text{altrimenti} \end{cases}$$

da cui ricaviamo che

$$A \cap B = \text{Dom}_{\varphi_{P_i \in \mathscr{P}}} \qquad A \cup B = \text{Dom}_{\varphi_{P_u \in \mathscr{P}}}$$

I due insiemi sono quindi ricorsivamente numerabili per la seconda caratterizzazione.

□

1.13.8 Teorema di Rice

Il **teorema di Rice** permette di mostrare che gli insiemi appartenenti a una certa classe **non** sono ricorsivi.

Sia $\{\varphi_i\}$ un SPA. Un insieme (di programmi) $I \subseteq \mathbb{N}$ è un insieme che rispetta le funzioni se e solo se

$$(a \in I \land \varphi_a = \varphi_b) \implies b \in I$$

In sostanza, I rispetta le funzioni se e solo se, data una funzione calcolata da un programma in I, allora I contiene tutti i programmi che calcolano quella funzione. Questi insiemi sono detti anche **chiusi per semantica**.

Per esempio, l'insieme $I = \{x \in \mathbb{N} \mid \varphi_x(3) = 5\}$ rispetta le funzioni, infatti:

$$\underbrace{a \in I}_{\varphi_a(3)=5} \land \underbrace{\varphi_a = \varphi_b}_{\varphi_b(3)=5} \implies b \in I$$

Teorema 1.13.7 (Teorema di Rice)**.** *Sia $I \subseteq \mathbb{N}$ un insieme che rispetta le funzioni. Allora I è ricorsivo solo se $I = \varnothing$ oppure $I = \mathbb{N}$.*

← *(Questo teorema dice che gli insiemi che rispettano le funzioni non sono mai ricorsivi, tolti i casi banali $\varnothing$ e $\mathbb{N}$.)*

Proof. Sia I insieme che rispetta le funzioni con $I \neq \varnothing$ e $I \neq \mathbb{N}$. Assumiamo, per assurdo, che I sia ricorsivo. Dato che $I \neq \varnothing$, esiste almeno un elemento $a \in I$. Inoltre, dato che $I \neq \mathbb{N}$ esiste almeno un elemento $\bar{a} \notin I$.

Definiamo la funzione $t : \mathbb{N} \to \mathbb{N}$ come

$$t(n) = \begin{cases} \bar{a} & \text{se } n \in I \\ a & \text{se } n \notin I \end{cases}$$

Sappiamo che $t \in \mathscr{T}$ dato che è calcolabile dal seguente programma P

```
input(x)
if P_I(n) = 1 then
    output(ā)
end
else
    output(a)
end
```

← (Sia P_I il programma che decide I, sappiamo che esiste per assurdo)

Visto che $t \in \mathscr{T}$, il *teorema di ricorsione* assicura che in un SPA $\{\varphi_i\}$ esiste $d \in \mathbb{N}$ tale che

$$\varphi_d = \varphi_{t(d)}$$

Per tale d ci sono solo due possibilità rispetto a I:

- se $d \in I$, visto che I rispetta le funzioni $\varphi_d = \varphi_{t(d)}$, allora $t(d) \in I$. Ma $t(d \in I) = \bar{a} \notin I$, quindi ho un assurdo

- se $d \notin I$, allora $t(d) = a \in I$, ma I rispetta le funzioni, quindi sapendo che $\varphi_d = \varphi_{t(d)}$ deve essere che $d \in I$, quindi ho un assurdo

Quindi I non è ricorsivo.

□

Applicazione

Il teorema di Rice suggerisce un approccio per stabilire se un insieme $A \subseteq \mathbb{N}$ non è ricorsivo:

1. mostrare che A rispetta le funzioni
2. mostrare che $A \neq \varnothing$ e $A \neq \mathbb{N}$
3. A non è ricorsivo per Rice

Limiti alla verifica automatica del software

Definiamo le **specifiche**: descrizione di un problema e richiesta per i programmi che devono risolverlo automaticamente. Un programma è *corretto* se risponde alle specifiche.

Problema: posso scrivere un programma V che testa automaticamente se un programma sia corretto o meno? Il programma che vogliamo scrivere ha semantica:

$$\varphi_V(P) = \begin{cases} 1 & \text{se } P \text{ è corretto} \\ 0 & \text{se } P \text{ non è corretto} \end{cases}$$

Definiamo

$$\text{PC} = \{P \mid P \text{ è corretto}\}$$

Osserviamo che rispetta le funzioni, infatti:

$$\underbrace{P \in \text{PC}}_{P \text{ corretto}} \wedge \underbrace{\varphi_P = \varphi_Q}_{Q \text{ corretto}} \implies Q \in \text{PC}$$

Ma allora PC non è ricorsivo. Dato ciò, la correttezza dei programmi non può essere testata automaticamente. Esistono, però, dei casi limite in cui è possibile costruire dei test automatici:

- specifiche del tipo "*nessun programma è corretto*" generano $\text{PC} = \varnothing$
- specifiche del tipo "*tutti i programmi sono corretti*" generano $\text{PC} = \mathbb{N}$

Entrambi gli insiemi PC sono ovviamente ricorsivi e quindi possono essere testati automaticamente.

Questo risultato mostra che non è possibile verificare automaticamente le proprietà semantiche dei programmi (*a meno di proprietà banali*).

Capitolo 2

Teoria della Complessità

Dato un problema P, fino a ora la domanda è stata "*esiste un programma per la sua soluzione automatica?*" E tramite questa domanda si può indagare la teoria della calcolabilità, il cui soggetto di studio è l'esistenza (o meno) di un programma per un dato problema.

La prossima sezione riguarda la **teoria della complessità** in cui l'investigazione segue la domanda "*come funzionano i programmi per P?*".

Per rispondere a questa domanda, vogliamo sapere quante **risorse computazionali** vengono utilizzate durante la sua esecuzione. Vediamo altre domande a cui la teoria della complessità cerca di rispondere

- dato un programma per il problema P, quanto tempo impiega nella sua soluzione? Quanto spazio di memoria occupa?
- dato un problema P, qual è il minimo tempo impiegato dai programmi per P? Quanto spazio in memoria al minimo posso occupare per programmi per P?
- in che senso possiamo dire che un programma è **efficiente** in termini di tempo e/o spazio?
- quali problemi possono essere efficientemente risolti per via automatica?

2.1 Teoria dei linguaggi formali

Alcune definizioni:

- Un **alfabeto** è un insieme finito di simboli $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$
- Un **alfabeto binario** è un qualsiasi alfabeto composto da due soli simboli
- Una **stringa** su Σ è una sequenza di simboli appartenenti a Σ nella forma $x = x_1 \cdots x_n$, con $x_i \in \Sigma$
- La **lunghezza** di una stringa x indica il numero di simboli che la costituiscono e si indica con $|x|$
- La **stringa nulla** è una stringa particolare, indicata con ϵ, è tale che $|\epsilon| = 0$
- Con Σ^* si indica l'insieme delle stringhe che si possono costruire sull'alfabeto Σ (chiusura di Kleene), compresa la stringa nulla. L'insieme delle stringhe formate da almeno un carattere è $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$
- Un **linguaggio** L su un alfabeto Σ è un sottoinsieme $L \subseteq \Sigma^*$, che può essere finito o infinito

2.2 Macchina di Turing Deterministica (DTM)

Il punto di partenza dello studio della teoria della complessità e la definizione rigorosa delle risorse di calcolo e di come possono essere misurate.

Il modello di calcolo considerato è quello della **Macchina di Turing**, ideata da Alan Turing nel 1936. Si tratta di un modello teorico di calcolatore che consente di definire rigorosamente:

- i passi di computazione e la computazione stessa
- tempo e spazio di calcolo dei programmi

2.2.1 Struttura

Una **Macchina di Turing deterministica** è un dispositivo hardware fornito di:

- **nastro di lettura/scrittura**: un nastro infinito formato da celle, ognuna delle quali ha un proprio indice/indirizzo e può contenere un simbolo. Questo nastro viene usato come contenitore per l'input, ma anche come memoria durante l'esecuzione
- **testina di lettura e scrittura two-way**: dispositivo che permette di leggere e scrivere dei simboli sul nastro a ogni passo
- **controllo a stati finiti**: automa a stati finiti $Q = \{Q_0, \dots, Q_n\}$ che permette di far evolvere la computazione

Un passo di calcolo è una **mossa** che, dato lo stato corrente e il simbolo letto dalla testina, porta la DTM in un nuovo stato, scrivendo eventualmente un simbolo sul nastro e spostando eventualmente la testine. I risultati della mossa, quindi il nuovo stato, il simbolo da scrivere e il movimento della testina, vengono calcolati tramite una **funzione di transizione**, basata sui due input dati.

Definizione Informale

Il funzionamento di una DTM M su input $x \in \Sigma^*$ passa per due fasi:

1. **Inizializzazione**:
   - La stringa x viene posta, simbolo dopo simbolo, nelle celle del nastro dalla cella 1 fino alla cella $|x|$. Le celle dopo quelle che contengono x contengono il simbolo *blank*
   - La testina si posizione sulla prima cella
   - Il controllo a stati finiti è posto nello stato iniziale
2. **Computazione**:
   - Sequenza di mosse dettata dalla funzione di transizione

La computazione può andare in loop o arrestarsi se raggiunge una situazione in cui non è definita nessuna mossa per lo stato attuale. Si dice che M accetta $x \in \Sigma^*$ se M si arresta in uno stato tra quelli finali/accettanti, altrimenti la rifiuta.

Definiamo $L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\}$ il **linguaggio accettato** da M.

Definizione Formale

Una DTM è una sestupla $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, con:

- Q: insieme finito di **stati** assumibili dal controllo a stati finiti
- $q_0 \in Q$: **stato iniziale** da cui partono le computazioni di M
- $F \subseteq Q$: insieme degli **stati finali/accettanti** dove M si arresta accettando l'input

- Σ: **alfabeto di input** su cui sono definite le stringhe di input
- Γ: **alfabeto di lavoro** che contiene i simboli che possono essere letti/scritti dal/sul nastro. Vale $\Sigma \subset \Gamma$ perché Γ contiene il simbolo *blank*
- $\delta : Q \times \Gamma \to Q \times (\Gamma \backslash \{blank\}) \times \{-1, 0, 1\}$: **funzione di transizione** che definisce le mosse. Si tratta di una funzione parziale: quando non è definita la macchina si arresta. Inoltre, M non può scrivere il simbolo *blank*, lo può solo leggere

Analizziamo nel dettaglio lo sviluppo di una DTM M su input $x \in \Gamma^*$, visto solo informalmente:

1. **Inizializzazione**:
   - il nastro contiene la stringa $x = x_1 \cdots x_n$
   - la testina è posizionata sul carattere x_1
   - il controllo a stati finiti parte dallo stato q_0
2. **Computazione**: sequenza di mosse definite dalla funzione di transizione δ che manda, a ogni passo, da (q_i, γ_i) a $(q_{i+1}, \gamma_{i+1}, \{-1, 0, +1\})$

Se $\delta(q, \gamma) = \perp$, la macchina M si *arresta*. Quando la testina rimbalza tra due celle o rimane fissa in una sola, si verifica un *loop*. La macchina M accetta $x \in \Sigma^*$ se e solo se la computazione si arresta in uno stato $q \in F$. Come prima, $L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\}$ è il **linguaggio accettato** da M.

← (Guarda foto se vuoi Slide 18:9)

Queste macchine sono molto simili agli automi a stati finiti, seppur con alcune differenze:

- le FSM di default non possono tornare indietro, non sono two-way, ma questa differenza non aumenta la potenza computazionale, serve solo per avere automi più succinti
- le FSM hanno il nastro a sola lettura, mentre le DTM possono alterare il nastro a disposizione

Configurazione di una DTM

Proviamo, similmente a come fatto per le macchine RAM, a dare l'idea di **configurazione** delle DTM; possiamo vederla come una foto che descrive completamente la macchina M in un certo istante, in questo modo possiamo descrivere la computazione come usa serie di configurazioni/foto.

Le cose da ricordare sono:

- in che stato si trova la macchina
- in che posizione si trova la testina
- il contenuto non-blank del nastro

Definiamo quindi $C = (q, k, w)$ una configurazione con

- q: stato del controllo a stati finiti
- $k \in \mathbb{N}^+$: posizione della testina nel nastro
- $w \in \Gamma^+$: contenuto non-blank del nastro

All'inizio della computazione si ha la **configurazione iniziale** $C_0 = (q_0, 1, x)$. Una configurazione C si dice **accettante** se $C = (q \in F, k, w)$, si dice invece **d'arresto** se $C = (q, k, w)$, con $\delta(q, w_k) = \perp$.

Definizione di computazione tramite configurazioni

La computazione di M su $x \in \Sigma^*$ è la sequenza

$$C_0 \xrightarrow{\delta} C_1 \xrightarrow{\delta} \ldots \xrightarrow{\delta} C_i \xrightarrow{\delta} C_{i+1} \xrightarrow{\delta} \ldots$$

dove $\forall i \geqslant 0$ vale che da C_i si passa a C_{i+1} grazie alla funzione δ.

La macchina M accetta $x \in \Sigma^*$ se e solo se $C_0 \xrightarrow{*} C_f$, con C_f configurazione d'arresto e accettante. Il linguaggio accettato da M ha la stessa definizione data prima.

2.2.2 Altre versioni delle macchine di Turing

Versioni alternative

Il fatto che la macchina sia *deterministica* implica che, data una configurazione C_i, quella successiva è univocamente determinata dalla funzione δ. Quindi, data una configurazione C_i, esiste una sola configurazione C_{i+1} successiva, a meno di arresti.

← (Guarda foto se vuoi 18:11)

Nelle **Macchine di Turing Non Deterministiche (NDTM)**, una configurazione C_i può ammettere più configurazioni successive.

Nelle **Macchine di Turing Probabilistiche PTM**, data una configurazione C_i, possono esistere più configurazioni nelle quali si può entrare, ognuna associata a una probabilità $p_i \in [0, 1]$.

Infine, nelle **Macchine di Turing Quantistiche QTM**, data una configurazione C_i, esistono una serie di configurazioni successive nelle quali possiamo entrare osservando le ampiezze delle transizioni α_i. Queste ampiezze sono numeri complessi in $\mathbb{C}$ tali che

- $|\alpha_i| \leqslant 1$
- hanno probabilità $|\alpha_i|^2$
- le probabilità sommano a 1

Versione semplificata

Esibire, progettare e comprendere una DTM potrebbe risultare difficile, anche per casi semplici. Solitamente, nel descrivere una DTM viene utilizzato uno *pseudocodice* che ne chiarisce la dinamica.

Esistono una serie di teoremi che dimostrano che qualsiasi frammento di programma strutturato può essere tradotto in una DTM formale e viceversa.

Esempio: parità

Problema:

- Nome: parità
- Istanza: $x \in \mathbb{N}$
- Domanda: x è pari?

Come codifica utilizziamo quella binaria, ovvero

$$cod : \mathbb{N} \to \{0, 1\}^*$$

Di conseguenza, il linguaggio da riconoscere è

$$L_{\text{pari}} = \left\{x \in \{0, 1\}^* \mid x_1 = 1 \wedge x_{|x|} = 0\right\} \cup \{0\}$$

Risolvere il problema *parità* significa trovare una DTM M che rappresenta un algoritmo deterministico che riconosce L_{pari}.

Ricordando che $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, la seguente macchina riconosce L_{pari}:

- $Q = \{p, z_1, \mu, z, r\}$ insieme degli stati
- $\Sigma = \{0, 1\}$ alfabeto
- $\Gamma = \{0, 1, blank\}$ alfabeto di lavoro

- $q_0 = p$ stato iniziale
- $F = \{z_1, z\}$ insieme degli stati finali
- $\delta : Q \times \Gamma \to Q \times \Sigma \times \{-1, 0, 1\}$ funzione di transizione così definita

δ	*blank*	0	1
p	$\perp$	$(z_1, 0, +1)$	$(\mu, 1, +1)$
z_1	$\perp$	$(r, 0, +1)$	$(\mu, 1, +1)$
μ	$\perp$	$(z, 0, +1)$	$(\mu, 1, +1)$
z	$\perp$	$(z, 0, +1)$	$(\mu, 1, +1)$
r	$\perp$	$\perp$	$\perp$

Si può notare come, anche per un problema così semplice, si ha una funzione di transizione abbastanza complessa. Andiamo quindi a utilizzare uno pseudocodice:

Parità(n)

```
1  i := 1
2  f := false
3  switch x[i] do
4      case 0 do
5          i++
6          f := (x[i] == blank)
7          break
8      case 1 do
9          do
10             f := (x[i] == 0)
11             i++
12         while (x[i] ≠ blank)
13 return f
```

Alla fine dell'esecuzione avremo:

- *True* se $x \in L_{\text{pari}}$
- *False* se $x \notin L_{\text{pari}}$

2.3 Funzionalità di una DTM

2.3.1 Insiemi riconosciuti

La principale funzionalità di una DTM è **riconoscere linguaggi**. Un linguaggio $L \subseteq \Sigma^*$ è **riconoscibile** da una DTM se e solo se esiste una DTM M tale che $L = L_M$.

Grazie alla possibilità di riconoscere linguaggi, una DTM può riconoscere anche insiemi: dato $A \subseteq \mathbb{N}$, *come lo riconosco con una DTM?* La prima idea è quella di codificare ogni elemento $a \in A$ in un elemento di Σ^*, per poter passare dal riconoscimento di un insieme al riconoscimento di un linguaggio.

$$A \rightsquigarrow \boxed{cod} \rightsquigarrow L_A = \{cod(a) : a \in A\}$$

Un insieme A è riconoscibile da una DTM se e solo se esiste una DTM M tale che $L_A = L_M$.

Quando si fa riconoscere un insieme A a una DTM M, possono risultare due situazioni, in funzione dell'input (codificato):

1. se l'input appartiene ad A, allora M si arresta
2. se l'input *non* appartiene ad A, allora M può
    - arrestarsi rifiutando l'input, ovvero finisce in uno stato $q \notin F$, ma allora A è ricorsivo
    - andare in loop, ma allora A è ricorsivamente numerabile

Teorema 2.3.1. *La classe degli insiemi riconosciuti da una DTM coincide con la classe degli insiemi ricorsivamente numerabili.*

Un algoritmo deterministico per il riconoscimento di un insieme $A \subseteq \mathbb{N}$ è una DTM M tale che $L_A = L_M$ e tale che M si arresta su ogni input.

Teorema 2.3.2. *La classe degli insiemi riconosciuti da algoritmi deterministici coincide con la classe degli insiemi ricorsivi.*

2.3.2 Problemi di decisione

Una seconda funzionalità delle DTM è quella di risolvere **problemi di decisione**. Dato un problema Π, con istanza $x \in D$ e domanda $p(x)$, andiamo a codificare gli elementi di D in elementi di Σ^*, ottenendo $L_\Pi = \{cod(x) \mid x \in D \wedge p(x)\}$ **insieme delle istanze (codificate) a risposta positiva di** Π.

La DTM risolve Π se e solo se M è un algoritmo deterministico per L_Π, ovvero:

- se vale $p(x)$, allora M accetta la codifica di x
- se non vale $p(x)$, allora M si arresta senza accettare

2.3.3 Calcolo di funzioni

Oltre a ciò che è già stato detto, le DTM sono anche in grado di **calcolare funzioni**. Si tratta di un risultato molto importante, in quanto sappiamo che calcolare funzioni significa risolvere problemi del tutto generali, quindi non solo di decisione.

Data una funzione $f : \Sigma^* \to \Gamma^*$, la DTM M calcola f se e solo se

- se $f(x) \downarrow$ allora M su input x termina con $f(x)$ sul nastro
- se $f(x) \uparrow$ allora M su input x va in loop

A tutti gli effetti le DTM sono *sistemi di programmazione*.

2.3.4 Potenza computazionale

Si può dimostrare che le DTM calcolano tutte e sole le funzioni ricorsive parziali. Possiamo riscrivere la tesi di Church-Turing come:

Una funzione è intuitivamente calcolabile se e solo se è calcolata da una DTM

Inoltre, è possibile dimostrare che le DTM sono SPA, semplicemente mostrando che valgono i tre assiomi di Rogers:

1. le DTM calcolano tutte e sole le funzioni ricorsive parziali
2. esiste una DTM universale che simula tutte le altre
3. vale il teorema S_n^m

2.4 Simboli di Landau

Nella teoria delle complessità la domanda è *"quanto costa questo programma?"* Per capire il costo di un dato programma verranno valutate delle funzioni nella forma $f(n)$, dove n indica la grandezza dell'input della DTM. Nel fare il confronto tra due algoritmi per uno stesso problema, bisogna tenere in considerazione che a fare la differenza (in termini di prestazioni) sono gli input di dimensione *ragionevolmente grande*, dove con questa espressione intendiamo una dimensione significativa nel contesto d'applicazione del problema.

Per esempio, siano t_1 e t_2 due funzioni tali che

$$t_1(n) = 2n \quad | \quad t_2(n) = \frac{1}{100}n^2 + \frac{1}{2}n + 1$$

Quale delle due è migliore? *Dipende*, per n abbastanza piccoli allora t_2 è migliore, mentre per n sufficientemente grandi allora è migliore t_1.

Date due funzioni, non vanno valutate per valori precisi di n, ma ne va valutato il loro **andamento asintotico**, ovvero quando n tende a $+\infty$.

2.4.1 Simboli di Landau principali

I **simboli di Landau** sono utili per stabilire degli **ordini di grandezza** tra le funzioni, in modo da poterle paragonare. I più utilizzati sono:

1. O: date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$ diciamo che

   $$f(n) = O(g(n))$$

   se e solo se

   $$\exists c > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geqslant n_0 \quad f(n) \leqslant c \cdot g(n)$$

   Fornisce un upper bound alla funzione f.

2. Ω: date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$ diciamo che

   $$f(n) = \Omega(g(n))$$

   se e solo se

   $$\exists c > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geqslant n_0 \quad f(n) \geqslant c \cdot g(n)$$

   Fornisce un lower bound alla funzione f.

3. Θ: date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$ diciamo che

   $$f(n) = \Theta(g(n))$$

   se e solo se

   $$\exists c_1, c_2 > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geqslant n_0 \quad c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$$

Si può notare facilmente che valgono le proprietà

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

2.5 Definizione della risorsa tempo

Per *semplicità* usiamo una DTM e non una macchina RAM per dare una definizione rigorosa di tempo. Le RAM, per quanto semplici, lavorano con banchi di memoria che possono contenere dati di grandezza arbitraria ai quali si accede in tempo $O(1)$, cosa che invece non si può fare con le DTM perché il nastro contiene l'input diviso su più celle.

2.5.1 Definizione

Consideriamo la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ e definiamo

- $T(x)$ il **tempo di calcolo** di M su input $x \in \Sigma^*$ come il valore

$$T(x) = \#\text{ mosse della computazione di } M \text{ su input } x \text{ (anche } \infty)$$

- $t(n)$ la **complessità in tempo** di M (worst case) come la funzione

$$t : \mathbb{N} \to \mathbb{N} \mid t(n) = \max\{T(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

L'attributo **worst case** indica il fatto che $t(n)$ rappresenta il tempo peggiore di calcolo su tutti gli input di lunghezza n. Si tratta della metrica più utilizzata anche perché la più "manovrabile matematicamente", permette di usare delle funzioni più facilmente trattabili dal punto di vista algebrico. Ad esempio, nella situazione *average case* avremo una stima probabilmente migliore, ma richiede anche una distribuzione di probabilità, non sempre facile da ottenere.

Diciamo che il linguaggio $L \subseteq \Sigma^*$ è riconoscibile in **tempo deterministico** $f(n)$ se e solo se esiste una DTM M tale che

1. $L = L_M$
2. $t(n) \leqslant f(n)$

L'ultima condiziona indica che a noi "basta" $f(n)$, ma che possiamo accettare anche situazioni migliori.

Si può estendere questa definizione anche agli insiemi o ai problemi di decisione:

- **l**'insieme $A \subseteq \mathbb{N}$ è riconosciuto in tempo $f(n)$ se e solo se lo è il linguaggio

$$L_A = \{cod(A) \mid a \in A\}$$

- il **problema di decisione** Π è risolto in tempo $f(n)$ se e solo se lo è il linguaggio

$$L_\Pi = \{cod(x) \mid p(x)\}$$

Da qui in avanti, quando parleremo di linguaggio intenderemo indirettamente insiemi o problemi di decisione, vista la stretta analogia tra questi concetti.

2.5.2 Classi di complessità

Classificazione di funzioni

Tramite i simboli di Landau è possibile classificare le funzioni in una serie di classi.

Per quanto riguarda il tempo, data una funzione $t : \mathbb{N} \to \mathbb{N}$ possiamo avere:

Funzione	**Definizione formale**
Costante	$f(n) = O(1)$
Logaritmica	$f(n) = O(\log n)$
Lineare	$f(n) = O(n)$
Quadratica	$f(n) = O(n^2)$
Polinomiale	$f(n) = O(n^k)$
Esponenziale	$f(n)$ super polinomiale

← (Nota che la funzione $n^{\log n}$ viene classificata come esponenziale sebbene non sia nella forma $2^n, e^n \dots$)

L'ultima rappresenta una classe con costo "*troppo elevato*", ovvero una classe in cui non si vorrebbe mai capitare. Dentro questa rientrano tutte le funzioni super polinomiali, dette anche **inefficienti**.

Altrimenti, convenzionalmente, un algoritmo si dice **efficiente** se la sua complessità temporale è **polinomiale**.

Definizione di classi di complessità

Vogliamo utilizzare il concetto di *classi di equivalenza* per definire delle classi che racchiudano tutti i problemi che hanno bisogno della stessa quantità di risorse computazionali per essere risolti correttamente.

Una **classe di complessità** è un insieme dei problemi che vengono risolti entro gli stessi limiti di una o piu' risorse computazionali.

Classi di complessità principali

Proviamo a definire alcune classi di complessità in funzione del tempo. La prima è la classe

$$\text{DTIME}(f(n))$$

definita come l'insieme dei problemi risolti da una DTM in tempo deterministico $t(n) = O(f(n))$. Sappiamo in realtà che la definizione corretta dovrebbe riguardare i *linguaggi accettati*, ma è stato già visto come si ha un'analogia tra questi concetti.

Le DTM possono anche calcolare funzioni, quindi si può propagare questa definizione di DTIME anche alle funzioni stesse, ovvero si possono definire classi di complessità anche per le funzioni.

Ma cosa si intende con "*complessità in tempo per una funzione*?"
La funzione $f : \Sigma^* \to \Gamma^*$ è calcolata con **complessità in tempo** $t(n)$ dalla DTM M se e solo se su ogni input x di lunghezza n la computazione di M su x si arresta entro $t(n)$ passi, avendo $f(x)$ sul nastro. Detto ciò, si può introdurre la classe

$$\text{FTIME}(f(n))$$

Definita come l'insieme delle funzioni calcolate da una DTM in tempo deterministico $t(n) = O(f(n))$.

Grazie a quanto detto possiamo definire due classi di complessità storicamente importanti:

$$P = \bigcup_{k \geqslant 0} \text{DTIME}\left(n^k\right)$$

classe dei **problemi risolti da una DTM in tempo polinomiale** e

$$FP = \bigcup_{k \geqslant 0} \text{FTIME}\left(n^k\right)$$

classe delle **funzioni calcolate da una DTM in tempo polinomiale**.

Questi sono universalmente riconosciuti come i problemi efficientemente risolvibili in tempo.

*Ma perché **polinomiale** è sinonimo di efficiente in tempo?* Si possono dare tre motivazioni:

- **pratica**: la differenza tra algoritmi polinomiali ed esponenziali, anche per input "piccoli" è enorme: il tempo di risoluzione per il primo caso è frazioni di secondo, si può invece parlare di anni o secoli per il caso esponenziale
- "**composizionale**": programmi efficienti che richiamo routine efficienti rimangono efficienti, concatenare algoritmi genera un tempo pari alla somma delle complessità dei due algoritmi, rimanendo polinomiale

← (GUARDA Slide 20:3 per maggiori info)

- "**robustezza**": le classi P e FP rimangono invariate a prescindere dai molti modelli di calcolo utilizzati per circoscrivere i problemi efficientemente risolti

← (GUARDA Slide 20:4 per maggiori info)

Per l'ultimo motivo si può dimostrare che P e FP non dipendono dal modello scelto, che questo sia RAM, WHILE, …

2.5.3 Tesi di Church-Turing estesa (tempo)

La **tesi di Church-Turing estesa** afferma che la classe dei problemi **efficientemente risolubili in tempo** coincide con la classe dei problemi risolti in tempo polinomiale su DTM.

La si può vedere come la *versione quantitativa* della tesi di Church-Turing.

← (Ci sono un po di slide pratiche qui 20:5-8, in piu' questa tesi non è propriamente accettata come la tesi classica a causa di quantum computing)

2.5.4 Chiusura di P

Teorema 2.5.1. *La classe P è un'algebra di Boole, ovvero è chiusa rispetto alle operazioni di unione, intersezione e complemento.*

Proof. **Unione:** Date due istanze $A, B \in P$, siano M_A e M_B due DTM con tempo rispettivamente $p(n)$ e $q(n)$. Allora il seguente programma

```
input(n)
y := M_A(x)
z := M_B(x)
output(y ∧ z)
```

permette il calcolo dell'unione di A e B in tempo $t(n) = p(n) + q(n)$.

Intersezione: Date due istanze $A, B \in P$, siano M_A e M_B due DTM con tempo rispettivamente $p(n)$ e $q(n)$. Allora il seguente programma

```
input(n)
y := M_A(x)
z := M_B(x)
output(y ∨ z)
```

permette il calcolo dell'intersezione di A e B in tempo $t(n) = p(n) + q(n)$.

Complemento: Data l'istanza $A \in P$, sia M_A una DTM con tempo $p(n)$. Allora il seguente programma

```
input(n)
y := M_A(x)
output(¬y)
```

permette il calcolo del complemento di A in tempo $t(n) = p(n)$. □

La classe P, inoltre, è anche chiusa rispetto all'operazione di **composizione**: si possono comporre tra loro DTM come se fossero procedure black box. Esempio con due DTM:

$$x \rightsquigarrow \boxed{M_1} \rightsquigarrow x' \rightsquigarrow \boxed{M_2} \rightsquigarrow y$$

Supponiamo che le macchine M_1 e M_2 abbiano tempo rispettivamente $p(n)$ e $q(n)$, allora il tempo totale è

$$t(n) \leqslant p(n) + q(p(n))$$

Usiamo $q(p(n))$ perché eseguendo M_1 in $p(n)$ passi il massimo output scrivibile è grande $p(n)$.

2.5.5 Problemi difficili

Esistono moltissimi problemi pratici e importanti per i quali ancora non sono stati trovati algoritmi efficienti e non è nemmeno stato provato che tali algoritmi non possano per natura esistere. In altre parole, non sappiamo se tutti i problemi sono in realtà efficientemente risolubili o se ne esistono alcuni il cui miglior algoritmo di risoluzione abbia una complessità esponenziale.

2.6 Spazio di memoria

Vediamo ora la formalizzazione dell'altra importante risorsa di calcolo, ovvero lo **spazio di memoria**, inteso come quantità di memoria occupata durante la computazione.

2.6.1 Complessità in spazio

Data la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ e una stringa $x \in \Sigma^*$, chiamiamo $S(x)$ il numero di celle del nastro visitate/occupate durante la computazione di M su x. Questo numero potrebbe anche essere infinito. La **complessità in spazio** di M (worst case) è la funzione $s : \mathbb{N} \to \mathbb{N}$ definita come

$$s(n) = \max \{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

Da questa definizione è chiaro che, in ogni DTM, $s(n) \geqslant n$ in quanto dovrò sempre occupare almeno spazio n lineare per mantenere l'input sul nastro, ma è molto probabile che le celle effettivamente *sporcate* sono meno delle celle occupate dall'input.

Come non considerare l'interferenza dovuta all'input?

Per avere complessità anche sublineari, si potrebbe leggermente modificare la macchina e separare le operazioni del nastro, ovvero utilizzare due nastri diversi per la lettura e per la computazion :

- il **nastro di lettura** è read-only con testina two-way read-only
- il **nastro di lavoro** è invece read-write con testina two-way

La stringa in input è delimitata dai caratteri ¢ e \$ tali che ¢, \$ $\notin \Sigma$.

← (Guarda Slide 20:10 per diagramma)

La definizione formale di questa macchina è

$$M = (Q, \Sigma \cup \{¢, \$\}, \Gamma, \delta, q_0, F)$$

in cui tutto è analogo alle macchine di Turing viste fin'ora, tranne per la funzione di transizione δ, ora definita come

$$\delta : Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma \to Q \times (\Gamma \setminus \{blank\}) \times \{-1, 0, 1\}^2$$

con la quale M:

1. legge un simbolo sia dal nastro di input che dal nastro di lavoro
2. calcola lo stato prossimo dati i simboli letti e lo stato attuale
3. modifica il nastro di lavoro
4. comanda il moto delle due testine

Anche la definizione di configurazione va leggermente modificata: ora una **configurazione** di M è una quadrupla

$$C = \langle q, i, j, w \rangle$$

in cui:

- q è lo stato corrente
- i e j sono le posizioni della testina di input e della testina di lavoro
- w è il contenuto non blank del nastro di lavoro

Non serve più salvare l'input perché non cambia mai, dato il nastro read-only. Gli altri concetti (*computazione, accettazione, linguaggio accettato, complessità in tempo, …*) rimangono inalterati.

A questo punto possiamo ridefinire la complessità in spazio per queste nuove macchine di Turing.

Per ogni stringa $x \in \Sigma^*$, il valore $S(x)$ è ora dato dal numero di celle *del solo nastro di lavoro* visitate da M durante la computazione di x. Dunque, la **complessità in spazio deterministica** $s(n)$ di M è da intendersi come il massimo numero di celle visitate nel nastro di lavoro durante la computazione di stringhe di lunghezza n, quindi come prima:

$$s(n) = \max \{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

In questo modo misuriamo *solo lo spazio di lavoro*, che quindi può essere **anche sublineare**.

2.6.2 Linguaggi e Funzioni

Il linguaggio $L \subseteq \Sigma^*$ è riconosciuto in **spazio deterministico** $f(n)$ se e solo se esiste una DTM M tale che

1. $L = L_M$
2. $s(n) \leqslant f(n)$

Per il caso specifico del **calcolo di funzioni**, solitamente si considera una terza macchina di Turing, in cui è presente un *terzo nastro* dedicato alla sola scrittura dell'output della funzione da calcolare. Questa aggiunta ci permette di conteggiare effettivamente lo spazio per il lavoro e di non interferire con lo spazio di output.

Una funzione $f : \Sigma^* \to \Gamma^*$ viene calcolata con **complessità in spazio** $s(n)$ dalla DTM M se e solo se *su ogni input x di lunghezza n* la computazione di M occupa non più di $s(n)$ celle dal nastro di lavoro.

Sfruttando queste definizioni, possiamo definire la complessità in spazio per il riconoscimento di insiemi e per la funzione soluzione di problemi di decisione.

2.6.3 Classi di complessità

Per ogni funzione $f : \mathbb{N} \to \mathbb{N}$ definiamo

$$\text{DSPACE}(f(n))$$

la **classe dei linguaggi accettati da DTM in spazio deterministico** $O(f(n))$. Chiamiamo invece

$$\text{FSPACE}(f(n))$$

la **classe delle funzioni calcolate da DTM in spazio deterministico** $O(f(n))$.

Notiamo che le classi DSPACE e FSPACE **non** cambiamo se aggiungiamo alle DTM un numero *costante di nastri di lavoro*. Se può essere comodo possiamo quindi utilizzare DTM con k nastri di lavoro aggiuntivi, separando anche il nastro di input da quello di output.

← (Guarda slide 20:13 Per diagramma, in piu' manca ultima slide di 20 quindi guardala che è importante)

In generale, gli algoritmi permettono di ottimizzare una sola delle due risorse a disposizione (tempo e spazio). Un algoritmo che migliora una risorsa porta (spesso, ma non sempre) al peggioramento di un'altra.

2.6.4 Efficienza in termini di spazio

Definiamo

- $L = \text{DSPACE}(\log(n))$ la classe dei **linguaggi accettati in spazio deterministico** $O(\log n)$
- $FL = \text{FSPACE}(\log(n))$ la classe delle **funzioni calcolate in spazio deterministico** $O(\log(n))$

L e FL sono universalmente considerati i **problemi risolti efficientemente in termini di spazio**.

Finora, abbiamo stabilito due sinonimie:

- efficiente *in tempo* se e solo se il tempo è *polinomiale*
- efficiente *in spazio* se e solo se lo spazio è *logaritmico*

Entrambe le affermazioni trovano ragioni di carattere pratico, composizionale e di robustezza, come visto per il tempo.

Per lo spazio, le motivazioni sono le seguenti:

- **pratico**: operare in spazio logaritmico (sublineare) significa saper gestire grandi moli di dati senza doverle copiare totalmente in memoria centrale; i dati diventano grandi facilmente, si vogliono algoritmi che usano poca memoria ← (Guarda Slide 21:6 per piu info)
- **composizionale**: i programmi efficienti in spazio che richiamano routine efficienti in spazio, rimangono efficienti
- **robustezza**: le classi L e FL rimangono invariate, a prescindere dai modelli di calcolo utilizzati, ad esempio DTM multi-nastro, RAM, WHILE; …

2.6.5 Tesi di Church-Turing estesa (spazio)

Come per il tempo, la **tesi di Church-Turing estesa per lo spazio** afferma che la classe dei problemi efficientemente risolubili in spazio coincide con la classe dei problemi risolti in spazio logaritmico su DTM. ← (Questa non e' messa in discussione)

2.7 Tempo vs Spazio

Spesso promuovere l'ottimizzazione di una risorsa va a discapito dell'altra: *essere veloci* vuol dire (tipicamente) *occupare tanto spazio*, e viceversa.

Viene quindi naturale porsi le domande:

- *i limiti in tempo implicano dei limiti in spazio?*
- *i limiti in spazio implicano dei limiti in tempo?*

2.7.1 DTIME vs DSPACE

Per rispondere a queste domande confrontiamo le classi $\text{DTIME}(f(n))$ e $\text{DSPACE}(f(n))$.

Teorema 2.7.1. *Tutti i linguaggi accettati in tempo $f(n)$ sono anche accettati in spazio $f(n)$. Formalmente*

$$DTIME(f(n)) \subseteq DSPACE(f(n))$$

Proof. Se $L \in \text{DTIME}(f(n))$ allora esiste una DTM M che riconosce L in tempo $t(n) = O(f(n))$, quindi su input x di lunghezza n la macchina M compie $O(f(n))$ passi.

In tale computazione possono essere visitate al massimo $O(f(n))$ celle, ovvero una per ogni passo. Quindi M ha complessità in spazio $s(n) = O(f(n))$, ma allora $L \in \text{DSPACE}(f(n))$.

□

Teorema 2.7.2. *Tutte le funzioni accettate in tempo $f(n)$, sono anche accettate in spazio $f(n)$. Formalmente*

$$FTIME(f(n)) \subseteq FSPACE(f(n))$$

Si può notare come l'efficienza in tempo *non* porta immediatamente all'efficienza in spazio.

Un limite in tempo implica, in qualche modo, un limite in spazio, *vale il contrario? Possiamo dimostrare che $DSPACE(f(n)) \subseteq DTIME(f(n))$?*

Avendo un numero di celle prestabilito, è possibile iterare il loro utilizzo (anche all'infinito, ad esempio entrando in un loop), di conseguenza limitare lo spazio non implica necessariamente una limitazione di tempo.

Notiamo che, in una DTM M, un loop si verifica quando viene visitata una configurazione già vista in passato. Sfruttando questo fatto è possibile trovare una limitazione al tempo, trovando dopo quanto tempo vengono visitate tutte le configurazioni possibili.

Teorema 2.7.3. *Tutti i linguaggi accettati in spazio $f(n)$ vengono accettati in tempo $n \cdot 2^{O(f(n))}$.* ← *(Guarda Slides 21:10 se indeciso)*

Proof. Dato $L \in \text{DSPACE}(f(n))$ e una DTM M esistono una serie di configurazioni per M tali che

$$C_0 \xrightarrow{\delta} C_1 \xrightarrow{\delta} \dots \xrightarrow{\delta} C_m$$

in cui C_m e' uno stato accettante per L.

Sappiamo che DTIME è calcolabile dal numero di volte che viene utilizzata la funzione di transizione δ. Date C_i e C_j con $i \neq j$, vale $C_i \neq C_j$: infatti, se fossero uguali saremmo entrati in un loop. Di conseguenza, calcolando la cardinalità dell'insieme contenente tutte le configurazioni possibili, troviamo anche un upper bound per la risorsa tempo.

Ricordando che la configurazione è una quadrupla $\langle q, i, j, w \rangle$ formata da

- q stato della macchina
- i, j posizioni delle due testine
- w valore sul nastro di lavoro

Analizziamo quanti valori possono assumere ognuno di questi elementi:

- q: è una costante e vale $|Q|$
- i: contando i due limitatori il numero massimo è $n + 2$ ← (fine della stringa in input +2 caratteri agli estremi)
- j: stiamo lavorando in $\text{DSPACE}(f(n))$, quindi questo indice vale $O(f(n))$, più semplicemente scrivibile come $\alpha f(n)$
- w: è una stringa sull'alfabeto $\Gamma^{O(f(n))}$, che ha cardinalità $|\Gamma|^{O(f(n))}$, anche in questo caso scrivibile come $|\Gamma|^{\alpha f(n)}$

Moltiplicando tutti questi valori troviamo il seguente upper bound:

$$\begin{aligned}
|C| &\leqslant O(1) \cdot (n+2) \cdot \alpha f(n) \cdot |\Gamma|^{\alpha f(n)} \\
&\leqslant O(1) \cdot (n+2) \cdot |\Gamma|^{\alpha f(n)} \cdot |\Gamma|^{\alpha f(n)} \\
&= O(1) \cdot (n+2) \cdot |\Gamma|^{2\alpha f(n)} \\
&= O(1) \cdot (n+2) \cdot 2^{\log_2\left(|\Gamma|^{2\alpha f(n)}\right)} \\
&= O(1) \cdot (n+2) \cdot 2^{2\alpha f(n) \cdot \log_2(|\Gamma|)} \\
&= O\left(n \cdot 2^{O(f(n))}\right)
\end{aligned}$$

Quindi, M sa se accettare o rifiutare $x \in \Sigma^*$ in al massimo $O\left(n \cdot 2^{O(f(n))}\right)$ passi.

Ora, data una DTM M che accetta L con $s(n) \leqslant \alpha f(n)$, costruiamo una DTM M' che su input $x \in \Sigma^*$, con $|x| = n$ si comporta nel seguente modo:

1. scrive in unario su un nastro dedicato un time-out t tale che $t \sim O\left(n \cdot 2^{O(f(n))}\right)$
2. simula M e a ogni mossa cancella un simbolo di time-out dal nastro dedicato
3. se M accetta o rifiuta prima della fine del time-out, allora M' accetta o rifiuta allo stesso modo di M
4. se allo scadere del time-out M non ha ancora scelto, M' rifiuta perché sa di essere entrata in un loop

In questo modo, M' accetta il linguaggio L in tempo

$$t(n) = O\left(n \cdot 2^{O(f(n))}\right)$$

e quindi

$$\text{DSPACE}(f(n)) \subseteq \text{DTIME}\left(n \cdot 2^{O(f(n))}\right)$$

□

Come per il tempo, il teorema dimostrato vale anche per gli insiemi FSPACE e FTIME.

Teorema 2.7.4. *Tutte le funzioni calcolate in spazio $f(n)$ vengono calcolate in tempo $n \cdot 2^{O(f(n))}$*

$$FSPACE(f(n)) \subseteq FTIME\left(n \cdot 2^{O(f(n))}\right)$$

2.7.2 P vs L (primo round)

Ottenuti questi risultati, vogliamo studiare le relazioni tra efficienza in termini di spazio (classe L) e l'efficienza in termini di tempo (classe P).

Teorema 2.7.5. *Valgono le seguenti relazioni per efficienza in spazio e efficienza in tempo:*

$$L \subseteq P$$

$$FL \subseteq FP$$

Proof.

$$\begin{aligned} L = \text{DSPACE}(\log(n)) &\subseteq \text{DTIME}\left(n \cdot 2^{O(\log n)}\right) \\ &= \text{DTIME}\left(n \cdot 2^{\frac{\log_2(n)}{\log_2(\_)}}\right) \\ &= \text{DTIME}\left(n \cdot \left(2^{\log_2(n)}\right)^{\frac{1}{\log_2(\_)}}\right) \\ &= \text{DTIME}\left(n \cdot n^{\frac{1}{\log_2(\_)}}\right) \\ &= \text{DTIME}\left(n \cdot n^{\beta}\right) \\ &= \text{DTIME}\left(n^{\beta+1}\right) = \text{DTIME}\left(n^{k}\right) = P \end{aligned}$$

Allo stesso modo è ottenibile l'inclusione per FL e FP.

□

Grazie a questo teorema sappiamo che:

- **In teoria**, algoritmi efficienti in spazio portano immediatamente ad algoritmi efficienti in tempo. Non è detto il contrario: la domanda "*esiste un problema in P che non sta in L?*" è ancora un problema aperto
- **In pratica**, il grado del polinomio ottenuto da algoritmi efficienti in spazio è molto alto, e solitamente gli algoritmi efficienti in tempo vengono progettati separatamente

2.8 La "zona grigia"

Chiamiamo "*zona grigia*" quella nuvola di problemi di decisione importanti e con molte applicazioni per i quali non si conoscono ancora algoritmi efficienti in tempo, ma per i quali nessuna ha mai dimostrato che tali algoritmi non possano esistere. Infatti, dato un problema Π, se a oggi non esiste un algoritmo efficiente per la sua soluzione, questo non implica che non esista del tutto.

I problemi di decisione in questa zona hanno una particolarità: sono **efficientemente verificabili**. Questo significa che, dato un certificato (una soluzione proposta) per un'istanza specifica del problema, è possibile verificare in tempo polinomiale se la risposta corretta è *Sì* oppure *No*. In altre parole, esiste un **oracolo** che, fornito di un certificato, è in grado di confermare rapidamente la correttezza di una soluzione affermativa.

2.8.1 Esempi

CNF-SAT

Il problema **CNF-SAT** ha come obiettivo quello di stabilire se esiste un assegnamento di variabili booleane che soddisfi un predicato logico in forma normale congiunta. Le formule sono indicate con $\varphi(x_1, \ldots, x_n)$ e sono formate da congiunzioni $C_1 \wedge \cdots \wedge C_k$, ognuna delle quali contiene almeno una variabile booleana x_i.

Formalmente, data una CNF $\varphi(x_1, \ldots, x_n)$, vogliamo rispondere alla domanda

$$\exists \underline{x} \in \{0,1\}^n \mid \varphi(\underline{x}) = 1?$$

Un possibile algoritmo di risoluzione è quello esaustivo, ma le possibili permutazioni con ripetizione sono 2^n, mentre la verifica della soddisfacibilità è fattibile in tempo polinomiale n^k. Di conseguenza, questo algoritmo risulta *inefficiente*, esplorare tutto l'albero dei possibili assegnamenti richiederebbe tempo esponenziale.

Circuiti hamiltoniani

Dato $G = (V, E)$ grafo non direzionato, vogliamo sapere se G contiene un circuito hamiltoniano o meno (circuito in cui tutti i vertici di G vengono visitati una e una sola volta).

Un algoritmo per risolverlo è generare tutte le permutazioni possibili di vertici che iniziano e finiscono con lo stesso vertice, per poi verificare efficientemente se siano un circuito hamiltoniano.

La complessità temporale di questo algoritmo dipende da:

- numero di permutazioni possibili: quindi il numero di volte che viene eseguita la verifica, sono $n!$
- il controllo sulla permutazione può essere implementato efficientemente (tempo polinomiale)

Circuiti euleriani

Dato $G = (V, E)$ un grafo non direzionato, vogliamo sapere se G contiene un circuito euleriano o meno (circuito in cui ogni arco viene visitato una e una sola volta),

Teorema 2.8.1 (Teorema di Eulero)**.** *Un grafo G contiene un circuito euleriano se e solo se ogni suo vertice ha grado pari, ovvero*

$$\forall v \in V \quad d(v) = 2k \mid k \in \mathbb{N}$$

Dimostrato da Eulero nel 1736, grazie a questo teorema è possibile risolvere il problema in tempo lineare, quindi efficiente.

2.8.2 Classe EXPTIME

Definiamo ora la classe

$$\text{EXPTIME} = \bigcup_{k \geqslant 0} \text{DTIME}\left(2^{n^k}\right)$$

dei problemi con complessità temporale **esponenziale**. Ovviamente vale

$$P \subseteq \text{EXPTIME}$$

in quanto ogni polinomio è "*maggiorabile*" da un esponenziale. Per diagonalizzazione (absolute GOAT) si è dimostrato in realtà che

$$P \subset \text{EXPTIME}$$

sfruttando una NDTM (Non-Deterministic Turing Machine) con timeout.

2.9 Macchine di Turing non deterministiche (NDTM)

2.9.1 Algoritmi non deterministici

Abbiamo visto come esistano problemi molto utili di cui non si conoscono ancora algoritmi deterministici efficienti in tempo.

Tuttavia, è possibile costruire degli **algoritmi non deterministici** che li risolvano, sfruttando il fatto che possono valutare velocemente la funzione obiettivo del problema.

Dinamica dell'algoritmo

In generale, in un algoritmo non deterministico, la computazione non è univoca, ma si scinde in tante computazioni, una per ogni struttura generata.

Sono formati da due fasi principali:

- **fase congetturale**: viene generata "magicamente" una struttura/un assegnamento/una configurazione/una congettura che aiuta a dare una risposta *Sì* o *No*
- **fase di verifica**: viene usata la struttura generata per decidere se vale la proprietà che caratterizza il problema di decisione

Le varie computazioni delle fasi di verifica sono tutte deterministiche, è la fase congetturale a essere non deterministica (in particolare nella creazione della struttura "magica").

Dato un problema Π, un'istanza $x \in D$ e una proprietà $p(x)$, un algoritmo non deterministico risolve Π se e solo se

1. su ogni x con risposta *positiva* (quindi $\forall x : p(x) = 1$), esiste **almeno una computazione** k che accetta la coppia (x, s_k)
2. su ogni x con risposta *negativa* (quindi $\forall x : p(x) = 0$), non esiste **alcuna computazione** che accetti la coppia (x, s_k) per qualche s_k. Tutte le computazioni rifiutano o vanno in loop

Esempi

Vediamo un algoritmo non deterministico per la soluzione di CNF-SAT:

```
input(φ(x_1, ..., x_n))
genera ass. x ∈ {0,1}^n
if (φ(x_1, ..., x_n) == 1) then
    return 1
return 0
```

Ammettendo un modello di calcolo come quello descritto, questo è un algoritmo non deterministico, formato da fase congetturale e fase di verifica.

Vediamo invece un algoritmo non deterministico per trovare, se esiste, un circuito hamiltoniano in un grafo G

```
input(G = (V, E))
generate permutation pi(v_1, ..., v_n)
if pi(v_1, ..., v_n) is a Hamiltonian cycle in G
    return 1
return 0
```

Si può notare come abbia la stessa struttura del programma precedente.

2.9.2 Tempo di calcolo

Dato che è cambiato il modello di calcolo, bisogna rivedere la definizione di tempo di calcolo.

← (Guarda Slide 22:10 qui veramente FALLO)

Consideriamo il tempo di calcolo $T(x)$ per un'istanza x con risposta positiva come il miglior tempo di calcolo delle fasi di verifica con risposta positiva. Per convenzione, la fase congetturale non impiega tempo. Da ricordare che questo è un modello teorico, nella realtà verrà sperso tempo anche per la generazione delle strutture usate nelle verifiche.

← (come definire $T(n)$ quando NON accetta nessun ramo? (non e' banale))

Come formalizzare questo tipo di algoritmi?

2.9.3 Definizione di NDTM

Consideriamo una DTM M come già descritta e apportiamo delle modifiche:

← (Guarda Slide 22:11 qui veramente FALLO)

- allunghiamo il nastro, ora infinito anche verso sinistra

- aggiungiamo un *modulo congetturale*, che scriva sulla parte sinistra del nastro la struttura generata

Quindi il nastro conterrà sia l'input x del problema, sia la struttura γ generata dalla fase congetturale e la fase di verifica non lavorerà più solo su x, ma utilizzerà la coppia (γ, x). Allora

- x viene accettato $\Leftrightarrow \exists \gamma \in \Gamma^* : (\gamma, x)$ viene deterministicamente accettata
- non viene accettato altrimenti

Il linguaggio accettato da M è

$$L_M = \{x \in \Sigma^* : M \text{ accetta } x\}$$

Un linguaggio $L \subseteq \Sigma^*$ è accettato da un algoritmo non deterministico se e solo se esiste una NDTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ tale che $L = L_M$.

Ricordiamo che, dato un problema $(\Pi, x \in D, p(x))$, il linguaggio riconosciuto dal problema è

$$L_\Pi = \{cod(x) : x \in D \wedge p(x)\}$$

dove $cod : D \to \Sigma^*$ è la funzione di codifica delle istanze del problema.

Un algoritmo non deterministico per la soluzione di Π è una NDTM M tale che

$$L_\Pi = L_M$$

Ovviamente, mediante opportuna codifica, possiamo definire NDTM che accettano insiemi o funzioni.

2.9.4 Complessità in tempo

La complessità di un algoritmo non deterministico corrisponde alla miglior complessità in tempo nella "seconda fase" per quell'istanza. Allo stesso modo, possiamo definire la complessità in tempo per le NDTM.

Una NDTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ha **complessità in tempo** $t : \mathbb{N} \to \mathbb{N}$ se e solo se per ogni input $x \in L_M$ con $|x| = n$ esiste almeno una computazione accettante di M che impiega $t(n)$ passi.

Un linguaggio è accettato con **complessità in tempo non deterministico** $t(n)$ se e solo se esiste una NDTM M con complessità in tempo $t(n)$ che lo accetta.

In questo modo abbiamo mappato tutti i concetti chiave visti nella DTM per il non determinismo.

2.9.5 Classi di complessità non deterministiche

Si possono definire delle classi per i linguaggi accettati, allo stesso modo di come fatto per le DTM. Chiamiamo

$$\text{NTIME}(f(n))$$

l'insieme dei linguaggi accettati con complessità non deterministica $O(f(n))$.

Vogliamo caratterizzare il concetto di "*efficienza*" anche per il non determinismo.

Sappiamo che *efficiente risolubilità* si traduce nella classe

$$P = \bigcup_{k \geqslant 0} \text{DTIME}\left(n^k\right)$$

Definiamo ora l'**efficiente verificabilità** come la classe

$$NP = \bigcup_{k \geqslant 0} \text{NTIME}\left(n^k\right)$$

che corrisponde all'insieme dei problemi di decisione che ammettono algoritmi non deterministici polinomiali.

2.10 P vs NP

Domanda Da un Milione di Dollari!, Quale e' la relazione tra le classi P e NP

Teorema 2.10.1. $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. È facile notare che $\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n))$.

Dato $L \in \text{DTIME}(f(n))$, esiste una DTM M che lo riconosce in $t(n) = O(f(n))$. Chiaramente M può essere vista come una NDTM che ignora le **scelte non deterministiche**. La NDTM così ottenuta si comporta esattamente come M, ma ogni computazione è percorsa unicamente. È allora che questa NDTM accetta L in tempo $t(n) = O(f(n)) \Rightarrow L \in \text{NTIME}(f(n))$.

Quindi:

$$\mathbf{P} = \bigcup_{\mathbf{k} \geqslant \mathbf{0}} \text{DTIME}(\mathbf{n^k}) \subseteq \bigcup_{\mathbf{k} \geqslant \mathbf{0}} \text{NTIME}(\mathbf{n^k}) = \mathbf{NP}$$

□

Cosa possiamo dire della relazione inversa ??

È chiaro che il punto cruciale del Millennium Problem è proprio la relazione $NP \subseteq P$, che in realtà si traduce in $P = NP$ vista la relazione appena dimostrata. In altre parole, ci chiediamo se da un algoritmo non deterministico efficiente sia possibile ottenere un algoritmo *reale* efficiente.

Similmente a quanto fatto nella dimostrazione di $P \subseteq NP$, proviamo ad analizzare il problema:

$$\text{NTIME}(f(n)) \subseteq \text{DTIME}(?)$$

che quantifica quanto costa togliere il fattore di non determinismo (concetto non naturale) dalla fase congetturale. Supponiamo di avere $L \in \text{NTIME}(f(n))$. Questo implica che esiste una NDTM M tale che M accetta L con $t(n) = O(f(n))$. Come possiamo simulare la dinamica di M con una DTM $\widetilde{M}$?

Il funzionamento di $\widetilde{M}$ può essere il seguente:

- prende in input $x \in \Sigma^*$, con $|x| = n$;
- genera tutte le strutture $\gamma \in \Gamma^*$ delle fasi di verifica;
- per ognuna di esse, calcola *deterministicamente* se (γ, x) viene accettata da M;
- se almeno una computazione al punto 3 ha risposta positiva, allora accetta x; altrimenti $\widetilde{M}$ rifiuta (tramite dot-leaving o interleaving).

Il problema è che di stringhe $\gamma \in \Gamma^*$ ne esistono infinite! Ma ci servono proprio tutte?

Pseudocodice dell'algoritmo

DTM $\widetilde{M} \equiv$

```
input(x)
for each γ ∈ Γ* :
   if M accetta (γ, x) in t(n) passi:
      return 1
return 0
```

Per evitare di considerare tutte le stringhe γ, possiamo limitarci a quelle di lunghezza al più $t(n)$, altrimenti la computazione non terminerebbe in $t(n)$ passi.

L'algoritmo con questa accortezza quindi diventa:

DTM $\widetilde{M} \equiv$

```
input(x)
for each γ ∈ Γ* con |γ| = O(f(n)) :
   if M accetta (γ, x) in t(n) passi:
      return 1
return 0
```

Studiamo ora il tempo richiesto:

$$t(n) = |\Gamma|^{O(f(n))} \cdot O(f(n)) = O(f(n) \cdot 2^{O(f(n))}) \qquad \leftarrow (|\Gamma|^{O(f(n))} = 2^{O(f(n))} \quad (\text{because } c^{O(f(n))} = 2^{O(f(n)) \cdot \log c} = 2^{O(f(n))}))$$

quindi:

$$\text{NTIME}(f(n)) \subseteq \text{DTIME}(f(n) \cdot 2^{O(f(n))}).$$

Come ci si poteva aspettare, togliere il non determinismo ha un costo molto alto: l'algoritmo diventa esponenziale e quindi inefficiente.

$$NP \subseteq \text{DTIME}(2^{n^{O(1)}}) = \text{EXPTIME}.$$

Tuttavia, l'unica cosa che possiamo affermare con certezza è che tutti i problemi in NP hanno algoritmi di soluzione esponenziale, ma non possiamo ancora escludere che $NP \subseteq P$.

Attenzione: NP non significa "Non Polinomiale" ma *Polinomiale su macchina non deterministica*. Dire che NP contiene solo problemi con algoritmi esponenziali è **FALSO**. Tali problemi potrebbero anche ammettere soluzioni efficienti (anche se al momento non se ne conoscono).

Come affrontare P vs NP

Per i problemi della classe NP, non abbiamo una soluzione efficiente. Ma questo potrebbe significare che non li conosciamo abbastanza bene, o che ci manca una solida base matematica. Per dimostrare che $NP \subseteq P$, dovremmo trovare un algoritmo polinomiale per ogni problema in NP. Tuttavia, ciò è impraticabile, dato che NP contiene infiniti problemi eterogenei. Potremmo ovviare a questo problema isolando un sottoinsieme rappresentativo dei problemi in NP.

Strategia

1. Stabiliamo una **relazione di difficoltà** tra problemi in NP: dati $\pi_1, \pi_2 \in NP$, scriviamo $\pi_1 \leqslant \pi_2$ se la soluzione efficiente di π_2 implica quella di π_1 (π_1 non e' piu' difficile di π_2).
2. Troviamo i problemi più difficili di NP: $\pi \in NP$ è *difficile* se

$$\forall \widetilde{\pi} \in NP \quad \widetilde{\pi} \leqslant \pi.$$

3. Restringiamo la ricerca di algoritmi efficienti a questi problemi: se li risolviamo efficientemente, allora risolviamo tutto NP.

Riduzione in tempo polinomiale

Dati $L_1, L_2 \subseteq \Sigma^*$, diciamo che L_1 si *riduce polinomialmente* a L_2 ($L_1 \leqslant_P L_2$) se esiste $f : \Sigma^* \to \Sigma^*$ tale che:

- f è calcolabile in tempo polinomiale su una DTM ($f \in FP$);
- $\forall x \in \Sigma^* \quad x \in L_1 \Leftrightarrow f(x) \in L_2$.

Teorema 2.10.2. *Siano $A, B \subseteq \Sigma^*$ e $A \leqslant_P B$. Se $B \in P$ allora $A \in P$.*

Proof. Sia $f \in FP$ la funzione di riduzione. Consideriamo l'algoritmo:

$N \equiv$
```
input(x)
y := f(x)
if y ∈ B then
    return 1
else return 0
```

Questo è deterministico e riconosce A. La sua complessità è:

$$t(n) = t_f(n) + t_B(|y|) = p(n) + q(p(n)) = \text{poly}(n)$$

poiché i polinomi sono chiusi per somma e composizione. Quindi $A \in P$.

Questo vuol dire che A non e' piu' difficile di B, infatti se trovo un algoritmo efficiente per B allora ne ho uno efficiente per A

□

Problemi NP-completi

Un problema Π è **NP-completo** se:

- $\Pi \in NP$;
- $\forall \widetilde{\Pi} \in NP \quad \widetilde{\Pi} \leqslant_P \Pi$.

Sia NPC la classe dei problemi NP-completi.

Teorema 2.10.3. *Se $\Pi \in NPC$ e $\Pi \in P$, allora $NP \subseteq P$ e quindi $P = NP$.*

Proof. Dalla definizione, $\forall \widetilde{\Pi} \in NP$, $\widetilde{\Pi} \leqslant_P \Pi$.
Se $\Pi \in P$, allora per il teorema precedente, $\widetilde{\Pi} \in P$. Quindi $NP \subseteq P$ e dunque $P = NP$.

□

Esempi

Il primo problema NP-completo è SAT (problema di soddisfacibilità booleana in CNF), dimostrato nel 1970 con il Teorema di Cook-Levin.

Varianti:

- k-CNF-SAT: se $k \leqslant 2$ è risolvibile in tempo polinomiale, se $k \geqslant 3$ è NP-completo.
- CNF con clausole di Horn $\Rightarrow$ risolvibile in tempo polinomiale.

Problemi P-completi e P vs L

Ricordiamo che:

$$L = \text{DSPACE}(\log n), \quad P = \bigcup_{k \geqslant 0} \text{DTIME}(n^k)$$

È ovvio che $L \subseteq P$, ma è propria l'inclusione? Possiamo replicare l'approccio visto per NP.

Riduzione in spazio logaritmico

Dati $L_1, L_2 \subseteq \Sigma^*$, diciamo che L_1 si log-space riduce in L_2 $L_1 \leqslant_L L_2$ se esiste $f : \Sigma^* \to \Sigma^*$:

- f calcolabile su una DTM in spazio logaritmico ($f \in FL$)
- $\forall x \in \Sigma^* \quad x \in L_1 \Leftrightarrow f(x) \in L_2$

Teorema 2.10.4. *Se $A \leqslant_L B$ e $B \in L$, allora $A \in L$.*

Proof. Sia $f \in FL$ la funzione di riduzione log-space. Consideriamo l'algoritmo:

$$N \equiv \boxed{\begin{array}{l} \text{input}(x) \\ y := f(x) \\ \textbf{if } y \in B \textbf{ then} \\ \quad \textbf{return } 1 \\ \textbf{else return } 0 \end{array}}$$

Questo è deterministico e riconosce A. La sua complessità in spazio è:

$$s(n) = s_f(n) + s_B(|y|) = O(\log n) + O(\log |y|) = O(\log n) + O(\log p(n)) = O(\log n)$$

$\leftarrow$ (nota che $|y| \leqslant p(n)$ in quanto output di una procedura che impiega spazio logaritmico e quindi anche un numero polinomiale di passi in quanto $FL \subseteq FP$)

In conclusione l'algoritmo riconosce A in spazio logaritmico

Attenzione: L'algoritmo non puo' memorizzare y in quanto serve spazio $p(n) = \Omega(\log n)$, per risolvere questo problema basta genera y un bit alla volta, su richiesta di $y \in B$, La posizione del bit da salvare in memoria ammonta ad

$$\log(|y|) \leqslant \log(p(n)) = O(\log n)$$

□

Problemi P-completi

Un problema Π è P-completo se:

- $\Pi \in P$;
- $\forall \widetilde{\Pi} \in P \quad \widetilde{\Pi} \leqslant_L \Pi$.

Teorema 2.10.5. *Se $\Pi \in PC$ e $\Pi \in L$, allora $P \subseteq L$ e quindi $P = L$.*

Proof. Poiché $\Pi \in PC$, abbiamo che per ogni problema $\widetilde{\Pi} \in P$ vale $\widetilde{\Pi} \leqslant_L \Pi$. Ma assumendo che $\Pi \in L$ e con il Teorema

$$A \leqslant_L B \wedge B \in L \Rightarrow A \in L$$

otteniamo che per ogni problema $\widetilde{\Pi} \in P$ vale $\widetilde{\Pi} \in L$ e quindi possiamo concludere che

$$P \subseteq L \quad \text{e} \quad P = L$$

□

in più noi sappiamo che:

- Esistono problemi P-completi.
- Per nessuno di essi è ancora stato trovato un algoritmo deterministico che lavori in spazio logaritmico.

Esempi

- **Context-Free Membership:** dato una grammatica G e una stringa x, decidere se $x \in L(G)$.

  Tempo: $O(n^2 \log n)$, Spazio: $O(\log^2 n)$.
- **Circuit Value Problem (CVP):** valutare il valore di un circuito booleano.

Come per P e NP, è universalmente assunto che l'inclusione $L \subseteq P$ non propria. Una istanza di **P**-completezza per un problema praticamente equivale a dire che quel problema possiede algoritmo efficiente in tempo ma quasi certamente non possiede mai algoritmi efficienti in spazio

Si dimostra anche che i problemi **P**-completi quasi certamente non possono ammettere algoritmi paralleli efficienti.