Informatica teorica

Indice

Introduzione

L'**informatica teorica** è la branca dell'informatica che si "contrappone" all'informatica applicata: in quest'ultima, l'informatica è usata solo come uno *strumento* per gestire l'oggetto del discorso, mentre nella prima l'informatica diventa l'*oggetto* del discorso, di cui ne vengono studiati i fondamenti.

Analizziamo i due aspetti fondamentali che caratterizzano ogni materia:

1. il **cosa**: l'informatica è la scienza che studia l'informazione e la sua elaborazione automatica mediante un sistema di calcolo. Ogni volta che ho un *problema* cerco di risolverlo *automaticamente* scrivendo un programma. *Posso farlo per ogni problema? Esistono problemi che non sono risolubili?* Possiamo chiamare questo primo aspetto con il nome di **teoria della calcolabilità**, quella branca che studia cosa è calcolabile e cosa non lo è, a prescindere dal costo in termini di risorse che ne deriva. In questa parte utilizzeremo una caratterizzazione molto rigorosa e una definizione di problema il più generale possibile, così che l'analisi non dipenda da fattori tecnologici, storici, ...
2. il **come**: è relazionato alla **teoria della complessità**, quella branca che studia la quantità di risorse computazionali richieste nella soluzione automatica di un problema. Con *risorsa computazionale* si intende qualsiasi cosa venga consumata durante l'esecuzione di un programma.
   In questa parte daremo una definizione rigorosa di tempo, spazio e di problema efficientemente risolubile in tempo e spazio, oltre che uno sguardo al famoso problema $P = NP$.

In ordine, nella teoria della calcolabilità andremo a fare uno studio **qualitativo** dei problemi, nel quale individueremo quali sono quelli calcolabili, mentre nella teoria della complessità questi ultimi verranno studiati in modo **quantitativo**.

Parte I — Teoria della calcolabilità

1. Richiami matematici: funzioni e prodotto cartesiano

1.1. Funzioni e insiemi di funzioni

Una **funzione** da un insieme A a un insieme B è una legge, spesso indicata con f, che definisce come associare agli elementi di A un elemento di B.

Una funzione può essere di due tipi differenti:

- **generale**: la funzione è definita in modo generale come $f : A \longrightarrow B$, in cui A è detto **dominio** di f e B è detto **codominio** di f;
- **locale/puntuale**: la funzione è definita solo per singoli valori $a \in A$ e $b \in B$, e scriviamo:

  $$f(a) = b \quad | \quad a \overset{f}{\longmapsto} b,$$

  in cui b è detta **immagine** di a rispetto a f e a è detta **controimmagine** di b rispetto a f.

Le funzioni possono essere categorizzate in base ad alcune proprietà:

- **iniettività**: una funzione $f : A \longrightarrow B$ si dice *iniettiva* se e solo se:

  $$\forall a_1, a_2 \in A \quad a_1 \neq a_2 \Longrightarrow f(a_1) \neq f(a_2).$$

  In sostanza, elementi diversi sono *mappati* ad elementi diversi.
- **suriettività**: una funzione $f : A \longrightarrow B$ si dice *suriettiva* se e solo se:

  $$\forall b \in B \quad \exists a \in A \mid f(a) = b,$$

  quindi ogni elemento del codominio ha almeno una controimmagine.
- **biettività**: una funzione $f : A \longrightarrow B$ si dice *biiettiva* se e solo se:

  $$\forall b \in B \quad \exists! a \in A \mid f(a) = b,$$

  che equivale a dire che è sia iniettiva sia suriettiva.

Definendo l'**insieme immagine**:

$$\mathrm{Im}_f = \{b \in B \mid \exists a \in A \text{ tale che } f(a) = b\} = \{f(a) \mid a \in A\} \subseteq B$$

possiamo dare una definizione alternativa di funzione suriettiva: essa è tale se e solo se $\mathrm{Im}_f = B$.

Se $f : A \longrightarrow B$ è una funzione biiettiva, si definisce **inversa** di f la funzione $f^{-1} : B \longrightarrow A$ tale che:

$$f(a) = b \Longleftrightarrow f^{-1}(b) = a.$$

Se f non fosse biiettiva, l'inversa avrebbe problemi di definizione.

Un'operazione definita su funzioni è la **composizione**: date $f : A \longrightarrow B$ e $g : B \longrightarrow C$, la funzione f *composto* g è la funzione $g \circ f : A \longrightarrow C$ definita come:

$$(g \circ f)(a) = g(f(a)).$$

La composizione *non è commutativa*, ovvero $g \circ f \neq f \circ g$ in generale, ma è *associativa*, quindi $(f \circ g) \circ h = f \circ (g \circ h)$.

Dato l'insieme A, la **funzione identità** su A è la funzione $i_A : A \longrightarrow A$ tale che:

$$\forall a \in A \quad i_A(a) = a,$$

quindi è una funzione che mappa ogni elemento in se stesso.

Grazie a questa, diamo una definizione alternativa di funzione inversa: data $f : A \longrightarrow B$ biiettiva, la sua inversa è l'unica funzione $f^{-1} : B \longrightarrow A$ che soddisfa la relazione:

$$f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}_A \,.$$

Introduciamo la notazione $f(a) \downarrow$ per indicare che la funzione f è definita per l'input a, mentre la notazione $f(a) \uparrow$ per indicare la situazione opposta.

Vediamo una seconda categorizzazione per le funzioni. Data $f : A \longrightarrow B$, diciamo che f è:

- **totale** se è definita per *ogni* elemento di A, ovvero $\forall a \in A \quad f(a) \downarrow$;
- **parziale** se è definita per *qualche* elemento di A, ovvero $\exists a \in A \mid f(a) \uparrow$.

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

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

Notiamo che:

- se $\mathrm{Dom}_f \subsetneq A$ allora f è una funzione parziale;
- se $\mathrm{Dom}_f = A$ allora f è una funzione totale.

È possibile **totalizzare** una funzione parziale f definendo una funzione a tratti $\overline{f} : A \longrightarrow B \cup \{\bot\}$ tale che:

$$\overline{f}(a) = \begin{cases} f(a) & a \in \mathrm{Dom}_f(a) \\ \bot & \text{altrimenti} \end{cases} \,.$$

Il simbolo $\bot$ è il *simbolo di indefinito* e viene utilizzato per tutti i valori per cui la funzione di partenza f non è appunto definita. Da qui in avanti utilizzeremo $B_\bot$ come abbreviazione di $B \cup \{\bot\}$.

L'insieme di tutte le funzioni da A in B si indica con:

$$B^A = \{f : A \longrightarrow B\}.$$

Viene utilizzata questa notazione in quanto la cardinalità di B^A è esattamente $|B|^{|A|}$, se A e B sono insiemi finiti.

Dato che vogliamo che siano presenti anche tutte le funzioni parziali da A in B, scriviamo:

$$B_\bot^A = \{f : A \to B_\bot\},$$

mettendo in evidenza il fatto che tutte le funzioni che sono presenti sono totali oppure parziali, ma che sono state totalizzate. Le due definizioni coincidono, ovvero

$$B^A = B_\bot^A$$

.

1.2. Prodotto cartesiano

Chiamiamo **prodotto cartesiano** l'insieme:

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

che rappresenta l'insieme di tutte le *coppie ordinate* di valori in A e in B.

In generale, il prodotto cartesiano **non è commutativo**, a meno che $A = B$.

Possiamo estendere il concetto di prodotto cartesiano a n-uple di valori:

$$A_1 \times ... \times A_n = \{(a_1, ..., a_n) \mid a_i \in A_i\}.$$

Per comodità chiameremo

$$\underbrace{A \times ... \times A}_{n} = A^n.$$

L'operazione "opposta" è effettuata dal **proiettore** i-esimo: esso è una funzione che estrae l'i-esimo elemento di un tupla, quindi è una funzione $\pi_i : A_1 \times ... \times A_n \longrightarrow A_i$ tale che:

$$\pi_i(a_1, ..., a_n) = a_i.$$

1.3. Funzione di valutazione

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

$$\omega : B_\perp^A \times A \longrightarrow B$$

tale che

$$\omega(f, a) = f(a).$$

In poche parole, è una funzione che prende una funzione f e la valuta su un elemento a del dominio.

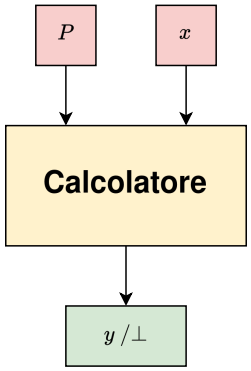
Esistono due tipi di analisi che possiamo effettuare su questa funzione:

- si tiene fisso a e si provano tutte le funzioni f: otteniamo un *benchmark*, ognuno dei quali è rappresentato dal valore a;
- si tiene fissa f e si provano tutte le a nel dominio: otteniamo il *grafico* di f.

2. Sistemi di calcolo

Quello che faremo ora è modellare *teoricamente* un sistema di calcolo.

Un **sistema di calcolo** lo possiamo vedere 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 ci restituisce y se è riuscita a calcolare un risultato, $\perp$ se è entrata in un loop.



Quindi, formalmente, un sistema di calcolo è una funzione del tipo:

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

Come vediamo, è molto simile ad una funzione di valutazione:

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

Per fare ciò ci serve prima definire cos'è un programma.

Un **programma** P è una **sequenza di regole** che trasformano un dato di input in uno di output (o in un loop). Formalmente, un programma è una funzione

$$P : \text{DATI} \longrightarrow \text{DATI}_{\perp},$$

cioè una funzione che appartiene all'insieme $\text{DATI}_{\perp}^{\text{DATI}}$.

Con questa definizione riusciamo a mappare l'insieme PROG sull'insieme delle funzioni, passo che ci serviva per definire la funzione di valutazione.

Riprendendo la definizione di sistema di calcolo, possiamo dire che esso è la funzione

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

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

Il modello classico che viene considerato quando si parla di calcolatori è quello di **Von Neumann**.

2.1. Potenza computazionale

Prima di definire cosa sia la potenza computazionale, facciamo una breve premessa: indichiamo con

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

la funzione che viene calcolata dal programma P, ovvero la semantica di P.

Detto ciò, definiamo la **potenza computazionale** di un calcolatore come l'insieme di tutte le funzioni che quel sistema di calcolo può calcolare grazie ai programmi, ovvero:

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

In altre parole, $F(\mathcal{C})$ rappresenta l'insieme di tutte le possibili semantiche di funzioni che sono calcolabili con il sistema $\mathcal{C}$.

Stabilire *che cosa può fare l'informatica* equivale a studiare quest'ultima inclusione. In particolare:

- 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*.

Se ci trovassimo nella prima situazione vorremmo trovare un modo per delineare un confine che divida le funzioni calcolabili da quelle che non lo sono.

Risolvere un problema equivale a calcolare una funzione: ad ogni problema è possibile associare una **funzione soluzione**, che mi permette di risolverlo automaticamente. Grazie a questa definizione, **calcolare le funzioni equivale a risolvere problemi**.

In che modo possiamo risolvere l'inclusione?

Un approccio sfrutta la **cardinalità** dei due insiemi. Con *cardinalità* si intende una funzione che associa ad ogni insieme il numero di elementi che contiene.

Sembra un ottimo approccio, ma presenta alcuni problemi: infatti, funziona solo quando l'insieme è finito, mentre è molto fragile quando si parla di insiemi infiniti.

Ad esempio, gli insiemi $\mathbb{N}$ dei numeri naturali e $\mathbb{P}$ dei numeri naturali pari sono tali che $\mathbb{P} \subsetneq \mathbb{N}$, ma hanno cardinalità $|\mathbb{N}| = |\mathbb{P}| = \infty$.

Ci serve dare una definizione diversa di cardinalità, visto che possono esistere "*infiniti più fitti/densi di altri*", come abbiamo visto nell'esempio precedente.

3. Richiami matematici: relazioni di equivalenza

Prima di dare una definizione formale e utile di cardinalità andiamo a riprendere alcuni concetti sulle relazioni di equivalenza.

3.1. Relazioni di equivalenza

Dati due insiemi A, B, una **relazione binaria** R è un sottoinsieme $R \subseteq A \times B$ di coppie ordinate.

Consideriamo una relazione $R \subseteq A^2$. Due elementi $a, b \in A$ sono in **relazione** R se e solo se $(a, b) \in R$. Indichiamo la relazione tra due elementi tramite la notazione infissa aRb.

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

- **riflessiva**: $\forall a \in A \quad aRa$;
- **simmetrica**: $\forall a, b \in A \quad aRb \implies bRa$;
- **transitiva**: $\forall a, b, c \in A \quad aRb \land bRc \implies aRc$.

3.2. Partizione indotta dalla relazione di equivalenza

Ad ogni relazione di equivalenza si può associare una **partizione**, ovvero un insieme di sottoinsiemi A_i tali che:

- $\forall i \in \mathbb{N}^+ \quad A_i \neq \emptyset$;
- $\forall i, j \in \mathbb{N}^+ \mid i \neq j$ allora $A_i \cap A_j = \emptyset$;
- $\bigcup_{i \in \mathbb{N}^+} A_i = A$.

Diremo che R definita su A^2 *induce* una partizione $\{A_1, A_2, ...\}$ su A.

3.3. Classi di equivalenza e insieme quoziente

Dato un elemento $a \in A$, chiamiamo l'insieme

$$[a]_R = \{b \in A \mid aRb\}$$

classe di equivalenza di a, ovvero tutti gli elementi che sono in relazione con esso. L'elemento a prende il nome di **rappresentante della classe**.

Si può dimostrare che:

- non esistono classi di equivalenza vuote, garantito dalla riflessività;
- dati $a, b \in A$ allora $[a]_R \cap [b]_R = \emptyset$ oppure $[a]_R = [b]_R$, garantito dal fatto che due elementi o sono in relazione o non lo sono;
- $\bigcup_{a \in A} [a]_R = A$.

Notiamo che, per definizione, l'insieme delle classi di equivalenza è una partizione indotta dalla relazione R sull'insieme A. Questa partizione è detta **insieme quoziente** di A rispetto a R ed è denotato con A / R.

4. Cardinalità

4.1. Isomorfismi

Due insiemi A e B sono **isomorfi** (*equinumerosi*) se esiste una biiezione tra essi. Formalmente scriviamo:

$$A \sim B.$$

Detto $\mathcal{U}$ l'insieme di tutti gli insiemi, la relazione $\sim$ è sottoinsieme di $\mathcal{U}^2$.

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

- **riflessività**: $A \sim A$, usiamo come biiezione la funzione identità i_A;
- **simmetria**: $A \sim B \implies B \sim A$, usiamo come biiezione la funzione inversa;
- **transitività**: $A \sim B \land B \sim C \implies A \sim C$, usiamo come biiezione la composizione della funzione usata per $A \sim B$ con la funzione usata per $B \sim C$.

Dato che $\sim$ è una relazione di equivalenza, ci permette di partizionare l'insieme $\mathcal{U}$. La partizione che ne risulta è formata da classi di equivalenza che contengono insiemi isomorfi, ossia con la stessa cardinalità.

Possiamo, quindi, definire la **cardinalità** come l'insieme quoziente di $\mathcal{U}$ rispetto alla relazione $\sim$.

Questo approccio permette di confrontare tra loro la cardinalità di insiemi infiniti, dato che basta trovare una funzione biettiva tra i due insiemi per poter affermare che siano isomorfi.

4.2. Cardinalità finita

La prima classe di cardinalità che vediamo è quella delle **cardinalità finite**.

Definiamo la seguente famiglia di insiemi:

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

Diremo che 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.

4.3. Cardinalità infinita

L'altra classe di cardinalità è quella delle **cardinalità infinite**, ovvero gli insiemi non in relazione con J_n. Questi insiemi sono divisibili in:

- insiemi **numerabili**;
- insiemi **non numerabili**.

Analizziamo le due tipologie separatamente.

4.3.1. Insiemi numerabili

Un insieme A è numerabile se e solo se $A \sim \mathbb{N}$, ovvero $A \in [\mathbb{N}]_\sim$.

Gli insiemi numerabili vengono detti anche "**listabili**", in quanto è possibile elencare *tutti* gli elementi dell'insieme A tramite una regola, la funzione f biettiva tra $\mathbb{N}$ e A. Grazie alla funzione f, è possibile elencare gli elementi di A formando l'insieme:

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

Questo insieme è esaustivo, dato che elenca ogni elemento dell'insieme A senza perderne nessuno.

Tra gli insiemi numerabili più famosi troviamo:

- numeri pari $\mathbb{P}$ e numeri dispari $\mathbb{D}$;
- numeri interi $\mathbb{Z}$, generati con la biiezione $f(n) = (-1)^n \left(\frac{n+(n \bmod 2)}{2} \right)$;
- numeri razionali $\mathbb{Q}$.

Gli insiemi numerabili hanno cardinalità $\aleph_0$ (si legge *"aleph zero"*).

4.3.2. Insiemi non numerabili

Gli **insiemi non numerabili** sono insiemi a cardinalità infinita ma che non sono listabili come gli insiemi numerabili: sono "più fitti" di $\mathbb{N}$. Questo significa che ogni lista generata mancherebbe di qualche elemento e, quindi, non sarebbe esaustiva di tutti gli elementi dell'insieme.

Il più famoso insieme non numerabile è l'insieme dei numeri reali $\mathbb{R}$.

Teorema L'insieme $\mathbb{R}$ non è numerabile ($\mathbb{R} \nsim \mathbb{N}$).

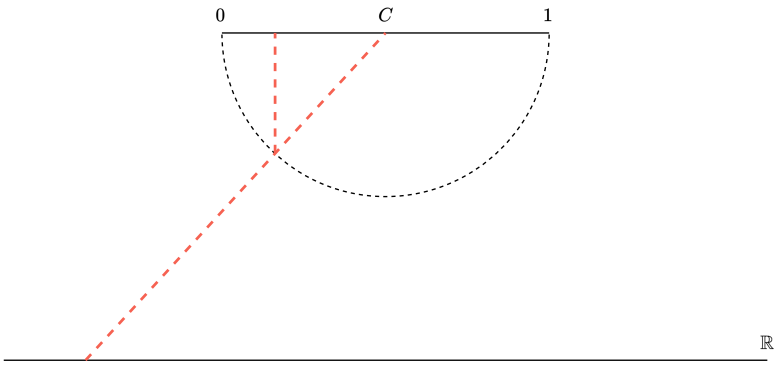
Dimostrazione
Suddividiamo la dimostrazione in tre punti:

1. dimostriamo che $\mathbb{R} \sim (0, 1)$;
2. dimostriamo che $\mathbb{N} \nsim (0, 1)$;
3. dimostriamo che $\mathbb{R} \nsim \mathbb{N}$.

[1] Partiamo con il dimostrare che $\mathbb{R} \sim (0, 1)$: serve trovare una biiezione tra $\mathbb{R}$ e $(0, 1)$. Usiamo una rappresentazione grafica, costruita in questo modo:

- disegnare la circonferenza di raggio $\frac{1}{2}$ centrata in $\frac{1}{2}$;
- 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 l'asse reale e la retta finale è il punto mappato.



In realtà, questo approccio ci permette di dire che $\mathbb{R}$ è isomorfo a qualsiasi segmento di lunghezza maggiore di 0.
La stessa biiezione vale anche sull'intervallo chiuso $[0, 1]$, utilizzando la "compattificazione" $\dot{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ e mappando 0 su $-\infty$ e 1 su $+\infty$.

[2] Continuiamo dimostrando che $\mathbb{N} \nsim (0, 1)$: serve dimostrare che l'intervallo $(0, 1)$ non è listabile, quindi che ogni lista che scrivo manca di almeno un elemento e per farlo proveremo a

"costruire" proprio questo elemento.
Per assurdo, sia $\mathbb{N} \sim (0,1)$. Allora, possiamo listare gli elementi di $(0,1)$ esaustivamente come:

$$\begin{aligned} &0.\ a_{00}\ a_{01}\ a_{02}\ \dots \\ &0.\ a_{10}\ a_{11}\ a_{12}\ \dots \\ &0.\ a_{20}\ a_{21}\ a_{22}\ \dots \\ &0.\ \dots \quad , \end{aligned}$$

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

Costruiamo il numero $c = 0.c_0c_1c_2\dots$ tale che

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

In altre parole, questo numero viene costruito "guardando" le cifre sulla diagonale principale.

Questo numero appartiene a $(0,1)$, ma non appare nella lista scritta sopra: ogni cifra c_i del numero costruito differisce per almeno una posizione (quella sulla diagonale principale) da qualunque numero nella lista. Questo è assurdo, visto che avevamo assunto $(0,1)$ numerabile $\implies \mathbb{N} \nsim (0,1)$.

[3] Terminiamo dimostrando che $\mathbb{R} \nsim \mathbb{N}$ per transitività.

Più in generale, non si riesce a listare nessun segmento di lunghezza maggiore di 0. □

Questo tipo di dimostrazione (in particolare il punto [2]) è detta **dimostrazione per diagonalizzazione**.

L'insieme $\mathbb{R}$ viene detto **insieme continuo** e tutti gli insiemi isomorfi a $\mathbb{R}$ si dicono a loro volta continui. I più famosi insiemi continui sono:

- $\mathbb{R}$: insieme dei numeri reali;
- $\mathbb{C}$: insieme dei numeri complessi;
- $\mathbb{T} \subset \mathbb{I}$: insieme dei numeri trascendenti.

Vediamo due insiemi continui che saranno importanti successivamente.

4.3.3. Insieme delle parti

Il primo insieme che vediamo è l'**insieme delle parti** di $\mathbb{N}$, detto anche *power set*, ed è così definito:

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

Teorema $P(\mathbb{N}) \nsim \mathbb{N}$.

Dimostrazione
Dimostriamo questo teorema tramite diagonalizzazione.

Il **vettore caratteristico** di un sottoinsieme è un vettore che nella posizione p_i ha 1 se $i \in A$, altrimenti ha 0.

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

$$\mathbb{N} : 0\ 1\ 2\ 3\ 4\ 5\ 6\ \ldots$$
$$A : 0\ 1\ 1\ 0\ 1\ 1\ 0\ \ldots \quad .$$

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

$$b_0 = b_{00}\ b_{01}\ b_{02}\ \ldots$$
$$b_1 = b_{10}\ b_{11}\ b_{12}\ \ldots$$
$$b_2 = b_{20}\ b_{21}\ b_{22}\ \ldots \quad .$$

Vogliamo costruire un vettore che appartenga a $P(\mathbb{N})$, ma non è presente nella lista precedente. Definiamo il seguente:

$$c = \overline{b_{00}}\ \overline{b_{11}}\ \overline{b_{22}}\ldots$$

che contiene nella posizione c_i il complemento di b_{ii}.

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

Questo è assurdo perché abbiamo assunto che $P(\mathbb{N})$ fosse numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

Visto questo teorema possiamo concludere che:

$$P(\mathbb{N}) \sim [0,1] \sim \dot{\mathbb{R}}.$$

4.3.4. Insieme delle funzioni

Il secondo insieme che vediamo è l'**insieme delle funzioni** da $\mathbb{N}$ in $\mathbb{N}$ così definito:

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

Teorema $\mathbb{N}_\perp^{\mathbb{N}} \nsim \mathbb{N}$.

Dimostrazione

Anche in questo caso useremo la diagonalizzazione.

Per assurdo, assumiamo $\mathbb{N}_\perp^{\mathbb{N}}$ numerabile. Possiamo, quindi, listare $\mathbb{N}_\perp^{\mathbb{N}}$ come $\{f_0, f_1, f_2, \ldots\}$.

	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 $\varphi : \mathbb{N} \longrightarrow \mathbb{N}_{\perp}$ per dimostrare l'assurdo. Una prima versione potrebbe essere la funzione $\varphi(n) = f_n(n) + 1$, per *disallineare* la diagonale, ma questo non va bene: se $f_n(n) = \perp$ non sappiamo dare un valore a $\varphi(n) = \perp + 1$.

Definiamo quindi la funzione

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

Questa funzione è una funzione che appartiene a $\mathbb{N}_{\perp}^{\mathbb{N}}$, ma non è presente nella lista precedente. Infatti, $\forall k \in \mathbb{N}$ otteniamo

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

Questo è assurdo, perché abbiamo assunto $P(\mathbb{N})$ numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

5. Potenza computazionale di un sistema di calcolo

5.1. Validità dell'inclusione $F(\mathcal{C}) \subseteq \mathrm{DATI}_{\perp}^{\mathrm{DATI}}$

Ora che abbiamo una definizione più robusta di cardinalità, essendo basata su strutture matematiche, possiamo studiare la natura dell'inclusione

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

Vediamo prima due intuizioni (che saranno da dimostrare):

- $\mathrm{PROG} \sim \mathbb{N}$: identifichiamo ogni programma con un numero, ad esempio la sua codifica in binario;
- $\mathrm{DATI} \sim \mathbb{N}$: anche qui, identifichiamo ogni dato con la sua codifica in binario.

In altre parole, i programmi e i dati non sono più dei numeri naturali $\mathbb{N}$.

Questo ci permette di dire che:

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

Con questa relazione, abbiamo appena dimostrato che **esistono funzioni non calcolabili**. Queste funzioni sono problemi molto pratici e molto utili al giorno d'oggi: un esempio è la funzione che, dato un software, ci dica se è corretto o no.

Il problema è che *esistono pochi programmi e troppe/i funzioni/problemi.*

Quello che ci resta fare è dimostrare le due assunzioni

- $\mathrm{PROG} \sim \mathbb{N}$;
- $\mathrm{DATI} \sim \mathbb{N}$.

Lo faremo utilizzando le **tecniche di aritmetizzazione** (o *godelizzazione*) **di strutture**, tecniche che rappresentano delle strutture tramite un numero, così da poter utilizzare la matematica e tutti gli strumenti di cui essa dispone.

6. DATI $\sim \mathbb{N}$

Ci 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 per lavorare il numero che "riflettano" il risultato sul dato senza ripassare per il dato stesso;
3. ci consenta di dire, senza perdita di generalità, che i nostri programmi lavorano sui numeri.

6.1. Funzione coppia di Cantor

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

$$\langle\rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}^+.$$

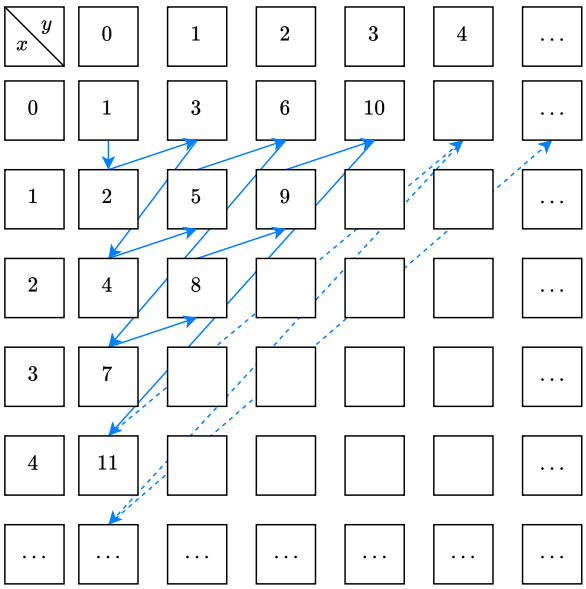
Questa funzione sfrutta due "sotto-funzioni"

$$\sin : \mathbb{N}^+ \longrightarrow \mathbb{N}, \quad \mathrm{des} : \mathbb{N}^+ \longrightarrow \mathbb{N}$$

tali che:

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

Vediamo una rappresentazione grafica della funzione di Cantor.



Come vediamo, $\langle x,y\rangle$ rappresenta il valore all'incrocio tra la x-esima riga e la y-esima colonna.

La tabella viene riempita *diagonale per diagonale* (non principale). Il procedimento è il seguente:

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

La funzione deve essere sia iniettiva sia suriettiva, quindi:

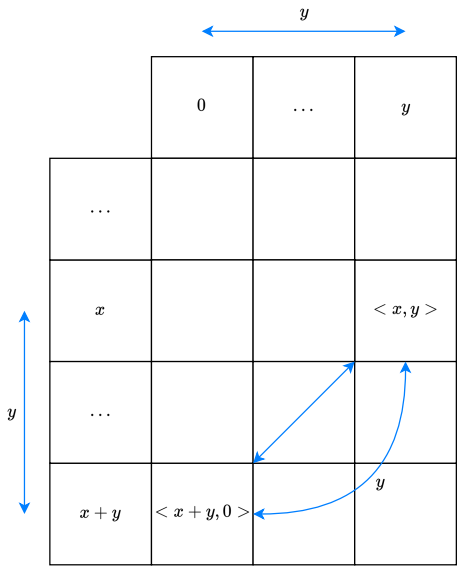
- non posso avere celle con lo stesso numero (iniettività);
- ogni numero in $\mathbb{N}^+$ deve comparire (surriettività).

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

6.1.1. Forma analitica della funzione coppia

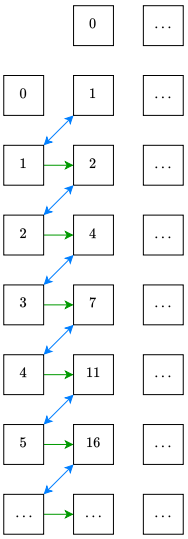
Cerchiamo di formalizzare la costruzione di questi numeri, dato che non è molto comodo costruire ogni volta la tabella. Notiamo che vale la seguente relazione:

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



In questo modo il calcolo della funzione coppia si riduce al calcolo di $\langle x+y,0\rangle$.

Chiamiamo $x+y=z$ e con la prossima immagine osserviamo un'altra proprietà.



Ogni cella $\langle z,0\rangle$ la si può calcolare come la somma di z e $\langle z-1,0\rangle$:

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

Questa forma è molto più compatta ed evita il calcolo di tutti i singoli $\langle z,0\rangle$.

Mettiamo insieme le due proprietà per ottenere la formula analitica della funzione coppia:

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

6.1.2. Forma analitica di sin e des

Vogliamo fare lo stesso per sin e des, per poter computare l'inversa della funzione di Cantor dato n.

Grazie alle osservazioni precedenti, sappiamo che:

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

Trovando il valore γ possiamo anche trovare i valori di x e y.

Notiamo come γ sia il "punto di attacco" della diagonale che contiene n, quindi:

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

perché tra tutti i punti di attacco $\langle z,0\rangle$ voglio esattamente la diagonale che contiene n.

Risolviamo quindi la disequazione:

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

Come valore di γ scegliamo

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

Ora che abbiamo γ, possiamo definire le funzioni sin e des in questo modo:

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

6.1.3. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$

Grazie alla funzione coppia di Cantor, possiamo dimostrare un risultato importante.

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

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

Estendiamo adesso il risultato all'interno insieme $\mathbb{N}$, ovvero:

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

Dimostrazione
Definiamo la funzione

$$[,] : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

tale che

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

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

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

6.2. Dimostrazione $\text{DATI} \sim \mathbb{N}$

È intuibile come i risultati ottenuti fino a questo punto ci permettono di dire che ogni dato è trasformabile in un numero, che può essere soggetto a trasformazioni e manipolazioni matematiche.

Tuttavia, 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. Vedremo come ogni struttura dati verrà manipolata e trasformata in una coppia (x, y) per poterla applicare alla funzione coppia.

6.3. Applicazione alle strutture dati

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

Codifichiamo la lista $[x_1, ..., x_n]$ con:

$$\langle x_1,...,x_n \rangle = \langle x_1,\langle x_2,\langle ...\langle x_n,0\rangle ...\rangle\rangle\rangle.$$

Ad esempio, la codifica della lista $M = [1, 2, 5]$ risulta essere:

$$\begin{aligned}\langle 1,2,5\rangle &= \langle 1,\langle 2,\langle 5,0\rangle\rangle\rangle \\ &= \langle 1,\langle 2,16\rangle\rangle \\ &= \langle 1,188\rangle \\ &= 18144.\end{aligned}$$

Per decodificare la lista M, applichiamo le funzioni sin e des al risultato precedente. Alla prima iterazione otterremo il primo elemento della lista e la restante parte ancora da decodificare.

Quando ci fermiamo? Durante la creazione della codifica di M abbiamo inserito un *"tappo"*, ovvero la prima iterazione della funzione coppia $\langle x_n,0\rangle$. Questo ci indica che quando $\text{des}(M)$ sarà uguale a 0, ci dovremo fermare.

Cosa succede se abbiamo uno 0 *nella lista?* Non ci sono problemi: il controllo avviene sulla funzione des, che restituisce la *somma parziale* e non sulla funzione sin, che restituisce i valori della lista.

Possiamo anche pensare delle implementazioni di queste funzioni. Assumiamo che:

- 0 codifichi la lista nulla;
- esistano delle routine per $\langle\rangle$, sin e des.

Codifica

```
def encode(numbers: list[int]) -> int:
  k = 0
  for i in range(n,0,-1):
    xi = numbers[i]
    k = <xi,k>
  return k
```

Decodifica

```
def decode(n: int) -> list[int]:
  numbers = []
  while True:
    left, n = sin(n), des(n)
    numbers.append(left)
    if n == 0:
      break
  return numbers
```

Un metodo molto utile delle liste è quello che ritorna la sua **lunghezza**:

```
def length(n: int) -> int:
  return 0 if n == 0 else 1 + length(des(n))
```

Infine, definiamo la funzione **proiezione** come:

$$\pi(t,n) = \begin{cases} -1 & \text{se } t > \text{length}(n) \vee t \leq 0 \\ x_t & \text{altrimenti} \end{cases}$$

e la sua implementazione:

```
def proj(t: int, n: int) -> int:
  if t <= 0 or t > length(n):
    return -1
  else:
    if t == 1:
      return sin(n)
    else:
      return proj(t-1, des(n))
```

Per gli **array** il discorso è più semplice, in quanto la dimensione è nota a priori. Di conseguenza, non necessitiamo di un carattere di fine sequenza. Dunque avremo che l'array $\{x_1,\ldots,x_n\}$ viene codificato con:

$$\{x_1,\ldots,x_n\} = \langle x_1,\langle\ldots,\langle x_{n-1},x_n\rangle\rangle\ldots\rangle.$$

Per quanto riguarda **matrici** l'approccio utilizzato codifica singolarmente le righe e successivamente codifica i risultati ottenuti come se fossero un array di dimensione uguale al numero di righe.

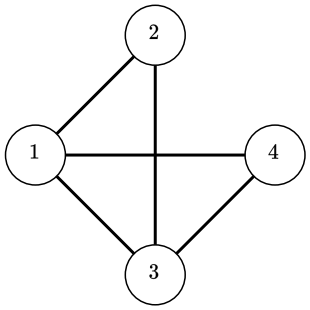
Ad esempio, la matrice:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

viene codificata in:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \langle\langle x_{11},x_{12},x_{13}\rangle,\langle x_{21},x_{22},x_{23}\rangle,\langle x_{31},x_{32},x_{33}\rangle\rangle.$$

Consideriamo il seguente grafo.



I **grafi** sono rappresentati principalmente in due modi:

- **liste di adiacenza dei vertici**: per ogni vertice si ha una lista che contiene gli identificatori dei vertici collegati direttamente con esso. Il grafo precedente ha

  $$\{1 : [2, 3, 4], 2 : [1, 3], 3 : [1, 2, 4], 4 : [1, 3]\}$$

  come lista di adiacenza, e la sua codifica si calcola come:

  $$\langle\langle 2,3,4\rangle,\langle 1,3\rangle,\langle 1,2,4\rangle,\langle 1,3\rangle\rangle.$$

  Questa soluzione esegue prima la codifica di ogni lista di adiacenza e poi la codifica dei risultati del passo precedente.
- **matrice di adiacenza**: per ogni cella m_{ij} della matrice $|V| \times |V|$, dove V è l'insieme dei vertici, si ha:

  $$m_{i,j} = \begin{cases} 1 & \text{se esiste un arco dal vertice } i \text{ al vertice } j \\ 0 & \text{altrimenti} \end{cases}.$$

  Essendo questa una matrice, la andiamo a codificare come già descritto.

6.4. Applicazione ai dati reali

Una volta visto come codificare le principali strutture dati, è facile trovare delle vie per codificare qualsiasi tipo di dato in un numero. Vediamo alcuni esempi.

Dato un **testo**, possiamo ottenere la sua codifica nel seguente modo:

1. trasformiamo il testo in una lista di numeri tramite la codifica ASCII dei singoli caratteri;
2. sfruttiamo l'idea dietro la codifica delle liste per codificare quanto ottenuto.

Per esempio:

$$\text{ciao} = [99, 105, 97, 111] = \langle 99,\langle 105,\langle 97,\langle 111,0\rangle\rangle\rangle\rangle.$$

Potremmo chiederci:

- *Il codificatore proposto è un buon compressore?*
  No, si vede facilmente che il numero ottenuto tramite la funzione coppia (o la sua concatenazione) sia generalmente molto grande, e che i bit necessari a rappresentarlo crescano esponenzialmente sulla lunghezza dell'input. Ne segue che questo è un **pessimo modo per comprimere messaggi**.
- *Il codificatore proposto è un buon sistema crittografico?*
  La natura stessa del processo garantisce la possibilità di trovare un modo per decifrare in modo analitico, di conseguenza chiunque potrebbe, in poco tempo, decifrare il mio messaggio. Inoltre, questo metodo è molto sensibile agli errori.

Dato un **suono**, possiamo *campionare* il suo segnale elettrico a intervalli di tempo regolari e codificare la sequenza dei valori campionati tramite liste o array.

Per codificare le **immagini** esistono diverse tecniche, ma la più usata è la **bitmap**: ogni pixel contiene la codifica numerica di un colore, quindi posso codificare separatamente ogni pixel e poi codificare i singoli risultati insieme tramite liste o array.

Abbiamo mostrato come i dati possano essere sostituiti da delle codifiche numeriche ad essi associate. Di conseguenza, possiamo sostituire tutte le funzioni $f : \mathrm{DATI} \longrightarrow \mathrm{DATI}_\perp$ con delle funzioni $f' : \mathbb{N} \to \mathbb{N}_\perp$. In altre parole, l'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentabile dall'insieme $\mathbb{N}_\perp^\mathbb{N}$.

7. PROG $\sim \mathbb{N}$

La relazione interviene nella parte che afferma che

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

In poche parole, la potenza computazionale, cioè l'insieme dei programmi che un sistema di calcolo $\mathcal{C}$ riesce a calcolare, è isomorfa all'insieme di tutti i programmi, a loro volta isomorfi a $\mathbb{N}$.

Per dimostrare l'ultima parte di questa catena di relazione dobbiamo esibire una legge che mi permetta di ricavare un numero dato un programma e viceversa.

Per fare questo vediamo l'insieme PROG come l'insieme dei programmi scritti in un certo linguaggio di programmazione. Analizzeremo due sistemi diversi:

- sistema di calcolo RAM;
- sistema di calcolo WHILE.

In generale, ogni sistema di calcolo ha la propria macchina e il proprio linguaggio.

7.1. Sistema di calcolo RAM

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

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

Il linguaggio utilizzato è un assembly molto semplificato, immediato e semplice.

Dopo aver definito $F(\text{RAM})$ potremmo chiederci se questa definizione sia troppo stringente e riduttiva per definire tutti i sistemi di calcolo. Introdurremo quindi delle macchine più sofisticate, dette **macchine while**, che, a differenza delle macchine RAM, sono *strutturate*. Infine, confronteremo $F(\text{RAM})$ e $F(\text{WHILE})$. I due risultati possibili sono:

- le potenze computazionali sono *diverse*: ciò che è computazionale dipende dallo strumento, cioè dal linguaggio utilizzato;
- le potenze computazionali sono *uguali*: la computabilità è intrinseca dei problemi, non dello strumento.

Il secondo è il caso più promettente e, in quel caso, cercheremo di trovare una caratterizzazione teorica, ovvero di *"recintare"* tutti i problemi calcolabili.

7.1.1. Struttura

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

Indichiamo i registri con R_k, con $k \geq 0$. Tra questi ci sono due registri particolari:

- R_0 contiene l'*output*;
- R_1 contiene l'*input*.

Un altro registro molto importante, che non rientra nei registri R_k, è il registro L, detto anche **program counter** (*PC*). Questo registro è essenziale in questa architettura, in quanto indica l'indirizzo della prossima istruzione da eseguire.

Dato un programma P, indichiamo con $|P|$ il numero di istruzioni che il programma contiene.

Le istruzioni nel linguaggio RAM sono:

- **incremento**: $R_k \longleftarrow R_k + 1$;

- **decremento**: $R_k \longleftarrow R_k \dot{-} 1$;
- **salto condizionato**: IF $R_k = 0$ THEN GOTO m, con $m \in \{1, ..., |P|\}$.

L'istruzione di decremento é tale che

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

7.1.2. Esecuzione di un programma RAM

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

1. **Inizializzazione**:
    1. viene caricato il programma $P \equiv \text{Istr}_1, ..., \text{Istr}_n$ in memoria;
    2. il PC viene posto a 1 per indicare di eseguire la prima istruzione del programma;
    3. nel registro R_1 viene caricato l'input;
    4. ogni altro registro è azzerato.
2. **Esecuzione**: si eseguono tutte le istruzioni *una dopo l'altra*, ovvero ad ogni iterazione passo da L a $L + 1$, a meno di istruzioni di salto. Essendo il linguaggio RAM *non strutturato* il PC è necessario per indicare ogni volta l'istruzione da eseguire al passo successivo. Un linguaggio strutturato, invece, sa sempre quale istruzione eseguire dopo quella corrente, infatti non è dotato di PC;
3. **Terminazione**: per convenzione si mette $L = 0$ per indicare che l'esecuzione del programma è finita oppure è andata in loop. Questo segnale, nel caso il programma termini, è detto **segnale di halt** e arresta la macchina;
4. **Output**: il contenuto di R_0, se vado in halt, contiene il risultato dell'esecuzione del programma P. Indichiamo con $\varphi_P(n)$ il contenuto del registro R_0 (in caso di halt) oppure $\perp$ (in caso di loop), allora:

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

Con $\varphi_P : \mathbb{N} \longrightarrow \mathbb{N}_\perp$ indichiamo la **semantica** del programma P.

Come indicavamo con $\mathcal{C}(P, \_)$ la semantica del programma P nel sistema di calcolo $\mathcal{C}$, indichiamo con $\text{RAM}(P, \_) = \varphi_P$ la semantica del programma P nel sistema di calcolo RAM.

7.1.3. Esecuzione definita formalmente

Vogliamo dare una definizione formale della semantica di un programma RAM. Quello che faremo sarà dare una **semantica operazionale** alle istruzioni RAM, ovvero specificare il significato di ogni istruzione esplicitando l'**effetto** che quell'istruzione ha sui registri della macchina.

Per descrivere l'effetto di un'istruzione ci serviamo di una *foto*. L'idea che ci sta dietro è:

1. faccio una foto della macchina *prima* dell'esecuzione dell'istruzione;
2. eseguo l'istruzione;
3. faccio una foto della macchina *dopo* l'esecuzione dell'istruzione.

La foto della macchina si chiama **stato** e deve descrivere completamente la situazione della macchina in un certo istante. La coppia $(\text{StatoPrima}, \text{StatoDopo})$ rappresenta la semantica operazionale di una data istruzione del linguaggio RAM.

L'unica informazione da salvare dentro una foto è la situazione globale dei registri R_k e il registro L. Il programma non serve, visto che rimane sempre uguale.

La **computazione** del programma P è una sequenza di stati S_i, ognuno generato dall'esecuzione di un'istruzione del programma. Diciamo che P induce una sequenza di stati S_i. Se quest'ultima è for-

mata 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. In poche parole:

$$\varphi_P : \mathbb{N} \longrightarrow \mathbb{N}_\perp \text{ tale che } \varphi_P(n) = \begin{cases} y & \text{se } \exists S_{\text{finale}} \\ \perp & \text{altrimenti} \end{cases} .$$

Definiamo ora come passiamo da uno stato all'altro. Per far ciò, definiamo:

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

  $$S : \{L, R_i\} \longrightarrow \mathbb{N}$$

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

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

  Questa rappresentazione è molto comoda perché ho potenzialmente un numero di registri infinito. Se così non fosse avrei delle tuple per indicare tutti i possibili registri al posto dell'insieme $\{L, R_i\}$;
- **stato finale**: uno stato finale è un qualsiasi stato S tale che $S(L) = 0$;
- **dati**: abbiamo già dimostrato come $\text{DATI} \sim \mathbb{N}$;
- **inizializzazione**: serve una funzione che, preso l'input, ci dia lo stato iniziale della macchina. La funzione è

  $$\text{in} : \mathbb{N} \longrightarrow \text{STATI} \text{ tale che } \text{in}(n) = S_{\text{iniziale}}.$$

  Lo stato S_{iniziale} è tale che

  $$S_{\text{iniziale}}(R) = \begin{cases} 1 & \text{se } R = L \\ n & \text{se } R = R_1 \\ 0 & \text{altrimenti} \end{cases} ;$$

- **programmi**: definisco PROG come l'insieme dei programmi RAM.

Ci manca da definire la *parte dinamica* del programma, ovvero l'**esecuzione**. Definiamo la **funzione di stato prossimo**

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

tale che

$$\delta(S, P) = S',$$

dove S rappresenta lo stato attuale e S' rappresenta lo stato prossimo dopo l'esecuzione di un'istruzione di P.

La funzione $\delta(S, P) = S'$ è tale che:

- se $S(L) = 0$ ho halt, ovvero deve terminare la computazione. Poniamo lo stato come indefinito, quindi $S' = \perp$;
- se $S(L) > |P|$ vuol dire che P non contiene istruzioni che bloccano esplicitamente l'esecuzione del programma. Lo stato S' è tale che:

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

- se $1 \leq S(L) \leq |P|$ considero l'istruzione $S(L)$-esima:
  - se ho incremento/decremento sul registro R_k definisco S' tale che

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

- se ho il GOTO sul registro R_k che salta all'indirizzo m definisco S' tale che

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

$$S'(R_i) = S(R_i) \quad \forall i.$$

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

$$S_0, S_1, ..., S_i, S_{i+1}, ...$$

tali che

$$\begin{aligned} & S_0 = \text{in}(n) \\ \forall i \quad & S_{i+1} = \delta(S_i, P). \end{aligned}$$

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

La semantica di P è

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

La potenza computazionale del sistema RAM è:

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

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

7.2. Aritmetizzazione di un programma

Per verificare che vale $\text{PROG} \sim \mathbb{N}$ dobbiamo trovare un modo per codificare i programmi in numeri in modo biunivoco. Notiamo che, data la lista di istruzioni semplici $P \equiv \text{Istr}_1, ..., \text{Istr}_m$, se questa fosse codificata come una lista di interi potremmo sfruttare la funzione coppia di Cantor per ottenere un numero associato al programma P.

Quello che dobbiamo fare è trovare una funzione che trasforma le istruzioni in numeri, così da avere poi accesso alla funzione coppia per fare la codifica effettiva. In generale, il processo che associa biunivocamente un numero ad una struttura viene chiamato **aritmetizzazione** o **godelizzazione**.

Troviamo una funzione Ar che associ ad ogni istruzione I_k la sua codifica numerica c_k. Se la funzione trovata è anche biunivoca siamo sicuri di trovare la sua inversa, ovvero quella funzione che ci permette di ricavare l'istruzione I_k data la sua codifica c_k.

Riassumendo quanto detto finora, abbiamo deciso di trasformare ogni lista di istruzioni in una lista di numeri e, successivamente, applicare la funzione coppia di Cantor, ovvero

$$[\text{Istr}_1, ..., \text{Istr}_n] \overset{\text{Ar}}{\rightsquigarrow} [c_1, ..., c_n] \overset{\langle\rangle}{\rightsquigarrow} n.$$

Vorremmo anche ottenere la lista di istruzioni originale data la sua codifica, ovvero

$$n \overset{\langle\rangle^{-1}}{\rightsquigarrow} [c_1, ..., c_n] \overset{\mathrm{Ar}^{-1}}{\rightsquigarrow} [\mathrm{Istr}_1, ..., \mathrm{Istr}_n].$$

La nostra funzione *"complessiva"* è biunivoca se dimostriamo la biunivocità della funzione Ar, avendo già dimostrato questa proprietà per $\langle\rangle$.

Dobbiamo quindi trovare una funzione biunivoca $\mathrm{Ar} : \mathrm{ISTR} \longrightarrow \mathbb{N}$ con la sua funzione inversa $\mathrm{Ar}^{-1} : \mathbb{N} \longrightarrow \mathrm{ISTR}$ tali che

$$\mathrm{Ar}(I) = n \iff \mathrm{Ar}^{-1}(n) = I.$$

7.2.1. Applicazione ai programmi RAM

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

$$\mathrm{Ar}(I) = \begin{cases} 3k & \text{se } I \equiv R_k \longleftarrow R_k + 1 \\ 3k+1 & \text{se } I \equiv R_k \longleftarrow R_k \dot{-} 1 \\ 3\langle k,m\rangle - 1 & \text{se } I \equiv \text{IF } R_k = 0 \text{ THEN GOTO } m \end{cases}.$$

Come è fatta l'inversa Ar^{-1}? In base al modulo tra n e 3 ottengo una certa istruzione:

$$\mathrm{Ar}^{-1}(n) = \begin{cases} R_{\frac{n}{3}} \longleftarrow R_{\frac{n}{3}} + 1 & \text{se } n \bmod 3 = 0 \\ R_{\frac{n-1}{3}} \longleftarrow R_{\frac{n-1}{3}} \dot{-} 1 & \text{se } n \bmod 3 = 1 \\ \text{IF } R_{\sin(\frac{n+1}{3})} = 0 \text{ THEN GOTO des}\left(\frac{n+1}{3}\right) & \text{se } n \bmod 3 = 2 \end{cases}.$$

La codifica del programma P è quindi

$$\mathrm{cod}(P) = \langle \mathrm{Ar}(\mathrm{Istr}_1), ..., \mathrm{Ar}(\mathrm{Istr}_n) \rangle.$$

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 $\mathrm{length}(\mathrm{cod}(P))$.

Abbiamo quindi dimostrato che $\mathrm{PROG} \sim \mathbb{N}$.

7.2.2. Osservazioni

Vediamo una serie di osservazioni importanti:

- avendo $n = \mathrm{cod}(P)$ 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*;
- posso 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 $\mathrm{PROG} \sim \mathbb{N}$, è numerabile;
- ho dimostrato rigorosamente che

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

  quindi anche nel sistema di calcolo RAM esistono funzioni non calcolabili;

- la RAM è troppo elementare affinche $F(\text{RAM})$ 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";
- se riesco a dimostrare che $F(\text{RAM}) = F(\mathcal{C})$ allora cambiare la tecnologia non cambia ciò che è calcolabile, ovvero la calcolabilità è intrinseca ai problemi, quindi possiamo *"catturarla"* matematicamente.

7.3. Sistema di calcolo WHILE

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

7.3.1. Struttura

La **macchina WHILE**, come quella RAM, è molto semplice, essendo formata da una serie di **registri**, detti **variabili**, ma al contrario delle macchine RAM questi ultimi non sono *potenzialmente infiniti*, ma sono esattamente 21. Il registro R_0 è il **registro di output**, mentre R_1 è il **registro di input**. Inoltre, non esiste il registro del program counter in quanto il linguaggio è *strutturato* e ogni istruzione di 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.

Il comando di base è l'**assegnamento**. In questo linguaggio ne esistono di tre tipi:

$$\begin{aligned} x_k &:= 0, \\ x_k &:= x_j + 1, \\ x_k &:= x_j \dot{-} 1. \end{aligned}$$

Vediamo come queste istruzioni siamo molto più complete rispetto alle istruzioni RAM

$$\begin{aligned} R_k &\longleftarrow R_k + 1 \\ R_k &\longleftarrow R_k \dot{-} 1 \end{aligned}$$

in quanto con una sola istruzione possiamo azzerare il valore di una variabile o assegnare ad una variabile il valore di un'altra aumentata/diminuita di 1.

I comandi "induttivi" sono invece il comando while e il comando composto.

Il **comando while** è un comando nella forma

$$\text{while } x_k \neq 0 \text{ do } C,$$

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

Il **comando composto** è un comando nella forma

$$\text{begin } C_1; ...; C_n \text{ end},$$

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

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

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

Chiamiamo

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

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

Per dimostrare una proprietà P di un programma $W \in W$-PROG procederemo induttivamente:
1. dimostro P vera sugli assegnamenti;
2. suppongo P vera sul comando C e la dimostro vera per while $x_k \neq 0$ do C;
3. suppongo P vera sui comandi $C_1, ..., C_n$ e la dimostro vera per begin $C_1; ...; C_n$ end.

7.3.2. 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, perché le istruzioni di W vengono eseguite una dopo l'altra;
3. **terminazione**: l'esecuzione di W può:
   - *arrestarsi*, se sono arrivato al termine delle istruzioni;
   - *non arrestarsi*, se si è entrati in un loop;
4. **output**: se il programma va in halt, l'output è contenuto nel registro x_0. Possiamo scrivere

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

La funzione $\Psi_W : \mathbb{N} \longrightarrow \mathbb{N}_\perp$ indica la **semantica** del programma W.

7.3.3. Esecuzione definita formalmente

Diamo ora la definizione formale della semantica di un programma WHILE. Come per i programmi RAM, abbiamo bisogno di una serie di elementi:
1. **stato**: usiamo una tupla grande quanto il numero di variabili, quindi $\underline{x} = (c_0, ..., c_{20})$ rappresenta il nostro stato, con c_i contenuto della variabile i;
2. $\boldsymbol{W}$**-STATI**: insieme di tutti gli stati possibili, che è in $\mathbb{N}^{21}$, vista la definizione degli stati;
3. **dati**: sappiamo già che DATI $\sim \mathbb{N}$;
4. **inizializzazione**: lo stato iniziale è descritto dalla seguente funzione

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

5. **semantica operazionale**: vogliamo trovare una funzione che, presi il comando da eseguire e lo stato corrente, restituisce lo stato prossimo.

Soffermiamoci sull'ultimo punto. Vogliamo trovare la funzione

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

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.

Partiamo dal passo base, quindi dagli **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, quindi:

- **comando composto**: vogliamo calcolare

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

  conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva. Calcoliamo allora la funzione:

  $$[\![C_n]\!](...([\![C_2]\!]([\![C_1]\!](\underline{x})))...) = ([\![C_n]\!] \circ ... \circ [\![C_1]\!])(\underline{x}),$$

  ovvero applichiamo in ordine i comandi C_i presenti nel comando composto C.
- **comando while**: vogliamo calcolare

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

  conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva. Calcoliamo allora la funzione:

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

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

  $$\text{while } x_k \neq 0 \text{ do } C = \begin{cases} [\![C]\!]^e(\underline{x}) & \text{se } e = \mu_t \\ \perp & \text{altrimenti} \end{cases}.$$

  Il valore $e = \mu_t$ è quel numero tale che $[\![C]\!]^e(\underline{x})$ ha la k-esima componente dello stato uguale a 0.

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

$$\Psi_W : \mathbb{N} \longrightarrow \mathbb{N}_\perp \mid \Psi_W(n) = \text{Proj}(0, [\![W]\!](\text{w-in}(n))).$$

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

La **potenza computazionale** del sistema di calcolo WHILE è l'insieme

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

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

7.4. Confronto tra macchina RAM e macchina WHILE

Viene naturale andare a confrontare i due sistemi di calcolo descritti, cercando di capire quale è "più potente" dell'altro, sempre che ce ne sia uno.

Ci sono quattro possibili situazioni:

- $F(\text{RAM}) \subsetneq F(\text{WHILE})$, che sarebbe anche comprensibile vista l'estrema semplicità del sistema RAM;
- $F(\text{RAM}) \cap F(\text{WHILE}) = \oslash$ (*insiemi disgiunti*) o abbia elementi (*insiemi sghembi*). Questo scenario sarebbe preoccupante, perché il concetto di calcolabile dipenderebbe dalla macchina che si sta utilizzando;
- $F(\text{WHILE}) \subseteq F(\text{RAM})$, che sarebbe sorprendente dato che il sistema WHILE sembra più sofisticato del sistema RAM, ma la relazione decreterebbe che il sistema WHILE non è più potente del sistema RAM;

- $F(\text{WHILE}) = F(\text{RAM})$, sarebbe il risultato migliore, perché il concetto di *calcolabile* non dipenderebbe dalla tecnologia utilizzata, ma sarebbe intrinseco nei problemi.

Poniamo di avere C_1 e C_2 sistemi di calcolo con programmi in C_1-PROG e C_2-PROG e potenze computazionali

$$F(C_1) = \left\{ f : \mathbb{N} \longrightarrow \mathbb{N}_\perp \mid \exists P_1 \in C_1\text{-PROG} \mid f = \Psi_{P_1} \right\} = \left\{ \Psi_{P_1} : P_1 \in C_1\text{-PROG} \right\},$$
$$F(C_2) = \left\{ f : \mathbb{N} \longrightarrow \mathbb{N}_\perp \mid \exists P_2 \in C_2\text{-PROG} \mid f = \Psi_{P_2} \right\} = \left\{ \Psi_{P_2} : P_2 \in C_2\text{-PROG} \right\}.$$

Come mostro che $F(C_1) \subseteq F(C_2)$, ovvero che il primo sistema di calcolo non è più potente del secondo? Devo dimostrare che ogni elemento nel primo insieme deve stare anche nel secondo, ovvero:

$$\forall f \in F(C_1) \Longrightarrow f \in F(C_2).$$

Se *espandiamo* la definizione di $f \in F(C)$ allora la relazione diventa:

$$\forall P_1 \in C_1\text{-PROG} \mid f = \Psi_{P_1} \Longrightarrow \exists P_2 \in C_2\text{-PROG} \mid f = \Psi_{P_2}.$$

In poche parole, per ogni programma calcolabile nel primo sistema di calcolo ne esiste uno con la stessa semantica nel secondo sistema. Quello che vogliamo trovare è un **compilatore**, ovvero una funzione che trasformi un programma del primo sistema in un programma del secondo sistema. Useremo il termine **traduttore** al posto di *compilatore*.

7.4.1. Traduzioni

Dati C_1 e C_2 due sistemi di calcolo, definiamo **traduzione** da C_1 a C_2 una funzione

$$T : C_1\text{-PROG} \longrightarrow C_2\text{-PROG}$$

con le seguenti proprietà:

- **programmabile**: esiste un modo per programmarla;
- **completa**: sappia tradurre **ogni** programma in C_1-PROG in un programma in C_2-PROG;
- **corretta**: mantiene la semantica del programma di partenza, ovvero

  $$\forall P \in C_1\text{-PROG} \quad \Psi_P = \varphi_{T(P)},$$

  dove Ψ rappresenta la semantica dei programmi in C_1-PROG e φ rappresenta la semantica dei programmi in C_2-PROG.

Teorema Se esiste $T : C_1\text{-PROG} \longrightarrow C_2\text{-PROG}$ allora $F(C_1) \subseteq F(C_2)$.

Dimostrazione
Se $f \in F(C_1)$ allora esiste un programma $P_1 \in C_1$-PROG tale che $\Psi_{P_1} = f$.

A questo programma P_1 applico T, ottenendo $T(P_1) = P_2 \in C_2$-PROG (per *completezza*) tale che $\varphi_{P_2} = \Psi_{P_1} = f$ (per *correttezza*).

Ho trovato un programma $P_2 \in C_2$-PROG la cui semantica è f, allora $F(C_1) \subseteq F(C_2)$. □

Mostreremo che $F(\text{WHILE}) \subseteq F(\text{RAM})$, ovvero il sistema WHILE non è più potente del sistema RAM. Quello che faremo sarà costruire un compilatore

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

che rispetti le caratteristiche di programmabilità, completezza e correttezza.

7.4.2. Compilatore da WHILE a RAM

Per comodità andiamo ad usare un linguaggio RAM **etichettato**: esso aggiunge la possibilità di etichettare un'istruzione che indica un punto di salto o di arrivo. In altre parole, le etichette rimpiazzano gli indirizzi di salto, che erano indicati con un numero di istruzione.

Questa aggiunta non aumenta la potenza espressiva del linguaggio, essendo pura sintassi: il RAM etichettato si traduce facilmente nel RAM *puro*.

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

- **passo base**: mostro come compilare gli assegnamenti;
- **passo induttivo**:
  1. per ipotesi induttiva, assumo di sapere $\text{Comp}(C_1), ..., \text{Comp}(C_m)$ e mostro come compilare il comando composto begin $C_1; ...; C_n$ end;
  2. per ipotesi induttiva, assumo di sapere $\text{Comp}(C)$ e mostro come compilare il comando while while $x_k \neq 0$ do C.

Nelle traduzioni andremo a mappare la variabile WHILE x_k nel registro RAM R_k. Questo non mi crea problemi o conflitti, perché sto mappando un numero finito di registri (21) in un insieme infinito.

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

$$\begin{aligned}\text{Comp}(x_k := 0) = \text{LOOP} : &\text{ IF } R_k = 0 \text{ THEN GOTO EXIT} \\ &R_k \longleftarrow R_k \dot{-} 1 \\ &\text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\ \text{EXIT} : &R_K \longleftarrow R_K \dot{-} 1 \quad .\end{aligned}$$

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 su 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 immediata e banale e l'istruzione RAM è

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

Se invece $k \neq j$ la prima idea che viene in mente è quella di "migrare" 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 mi genera R_j dentro R_k. Possiamo 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 di quella istruzione: infatti, io 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.

Ricapitolando:

1. salviamo x_j in R_{22}, registro *sicuro* perché mai coinvolto in altre istruzioni;
2. azzeriamo R_k;
3. rigeneriamo R_j e settiamo R_k da R_{22};
4. ± 1 in R_k.

$$
\begin{aligned}
\text{Comp}(x_k := x_j \pm 1) = \text{LOOP} &: \text{IF } R_j = 0 \text{ THEN GOTO EXIT1} \\
&\quad R_j \longleftarrow R_j \dot{-} 1 \\
&\quad R_{22} \longleftarrow R_{22} + 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\
\text{EXIT1} &: \text{IF } R_k = 0 \text{ THEN GOTO EXIT2} \\
&\quad R_k \longleftarrow R_k \dot{-} 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT1} \\
\text{EXIT2} &: \text{IF } R_{22} = 0 \text{ THEN GOTO EXIT3} \\
&\quad R_k \longleftarrow R_k + 1 \\
&\quad R_j \longleftarrow R_j + 1 \\
&\quad R_{22} \longleftarrow R_{22} \dot{-} 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT2} \\
\text{EXIT3} &: R_k \longleftarrow R_k \pm 1 \quad .
\end{aligned}
$$

Per ipotesi induttiva, sappiamo come compilare $C_1, ..., C_m$. Possiamo calcolare la compilazione del comando composto come

$$
\begin{aligned}
\text{Comp}(\text{begin } C_1; ...; C_n \text{ end}) = &\text{Comp}(C_1) \\
&... \\
&\text{Comp}(C_m) \quad .
\end{aligned}
$$

Per ipotesi induttiva, sappiamo come compilare C. Possiamo calcolare la compilazione del comando while come

$$
\begin{aligned}
\text{Comp}(\text{while } x_k \neq 0 \text{ do } C) = \text{LOOP} &: \text{IF } R_k = 0 \text{ THEN GOTO EXIT} \\
&\quad \text{Comp}(C) \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\
\text{EXIT} &: R_k \longleftarrow R_k \dot{-} 1 \quad .
\end{aligned}
$$

La funzione

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

che abbiamo costruito soddisfa le tre proprietà che desideravamo, quindi

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

Questa inclusione appena dimostrata è propria?

7.4.3. 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 $\varphi_P(x)$.

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

Abbiamo due problemi principali:

1. il primo riguarda il tipo di input della macchina WHILE: 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, piuttosto la sua codifica $\text{cod}(P) = n \in \mathbb{N}$. Questo mi 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 di input della macchina WHILE: quest'ultima legge l'input da un singolo registro, mentre qui ne stiamo passando due. Dobbiamo modificare I_W condensando l'input con la funzione coppia di Cantor, che diventa $\langle x,n \rangle$.

La semantica di I_W diventa

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

Come prima, per comodità di scrittura useremo un altro linguaggio, il **macro-WHILE**. Questo include alcune macro che saranno molto comode nella scrittura di I_W. Visto che viene modificata solo la sintassi, la potenza del linguaggio non WHILE non aumenta.

Le macro utilizzate sono:

- $x_k := x_j + x_s$;
- $x_k := \langle x_j,x_s \rangle$;
- $x_k := \langle x_1,...,x_n \rangle$;
- proiezione $x_k := \text{Proj}(x_j, x_s)$: estrae l'elemento x_j-esimo dalla lista codificata in x_s;
- incremento $x_k := \text{incr}(x_j, x_s)$: codifica la lista x_s con l'elemento in posizione x_j-esima aumentato di uno;
- decremento $x_k := \text{decr}(x_j, x_s)$: codifica la lista x_s con l'elemento in posizione x_j-esima diminuito di uno;
- $x_k := \sin(x_j)$;
- $x_k := \text{des}(x_j)$;
- costrutto if ... then ... else.

Risolto il problema dell'input di un interprete scritto in linguaggio WHILE per i programmi RAM, ora vogliamo scrivere questo interprete, ricordando che per comodità non useremo il WHILE puro ma il macro-WHILE.

Cosa fa l'interprete? In poche parole, esegue una dopo l'altra le istruzioni RAM del programma P e restituisce il risultato $\varphi_P(x)$. Notiamo come restituiamo un risultato, non un eseguibile.

Infatti, quello che fa l'interprete è ricostruire virtualmente tutto ciò che gli serve per gestire il programma. Nel nostro caso, 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 $R_0, R_1, R_2, ...$ dentro le variabili messe a disposizione dalla macchina WHILE.

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

La risposta è no. Infatti, se $\text{cod}(P) = n$ allora P non utilizza mai dei registri R_j, con $j > n$. Se uso un registro di indice $n + 1$ non posso avere codifica n perché l'istruzione che usa $n + 1$ ha come codifica i numeri $3(n + 1)$ se incremento, $3(n + 1) + 1$ se decremento oppure il numero generato da Cantor se GOTO. Inoltre, le singole istruzioni codificate vanno codificate tramite lista di Cantor, e abbiamo mostrato come questa funzione cresca molto rapidamente.

Di conseguenza, possiamo restringerci a modellare i registri $R_0, ..., R_{n+2}$. Usiamo i registri fino a a $n + 2$ solo per avere un paio di registri in più che potrebbero tornare utili. Ciò ci permette di codificare la memoria utilizzata dal programma P tramite la funzione di Cantor.

Vediamo, nel dettaglio, l'interno di $I_w(\langle x,n \rangle) = \varphi_n(x)$:

- $x_0 \longleftarrow \langle R_0,...,R_{n+2} \rangle$: stato della memoria della macchina RAM;

- $x_1 \longleftarrow L$: program counter;
- $x_2 \longleftarrow x$: dato su cui lavora P;
- $x_3 \longleftarrow n$: "listato" del programma P;
- x_4: codice dell'istruzione da eseguire, prelevata da x_3 grazie a x_1.

Ricordiamo che all'avvio l'interprete I_w trova il suo input nella variabile di input x_1.

```
# Inizializzazione
input(<x,n>);                          # In x1
x2 := sin(x1);                         # Dato x
x3 := des(x1);                         # Programma n
x0 := <0,x2,0,...,0>;                  # Memoria iniziale
x1 := 1;                               # Program counter

# Esecuzione
while (x1 != 0) do:                    # Se x1 = 0 HALT
  if (x1 > length(x2)) then            # Sono fuori dal programma?
    x1 := 0;                           # Vado in HALT
  else
    x4 := Pro(x1, x3);                 # Fetch istruzione
    if (x4 mod 3 == 0) then            # Incremento?
      x5 := x4 / 3;                    # Trovo k
      x0 := incr(x5, x0);              # Incremento
      x1 := x1 + 1;                    # Istruzione successiva
    if (x4 mod 3 == 1) then            # Decremento?
      x5 := (x4 - 1) / 3;              # Trovo k
      x0 := decr(x5, x0);              # Decremento
      x1 := x1 + 1;                    # Istruzione successiva
    if (x4 mod 3 == 2) then            # GOTO?
      x5 := sin((x4 + 1) / 3);         # Trovo k
      x6 := des((x4 + 1) / 3);         # Trovo m
      if (Proj(x5, x0) == 0) then      # Devo saltare?
        x1 := x4;                      # Salto a m
      else
        x1 := x1 + 1;                  # Istruzione successiva

# Finalizzazione
x0 := sin(x0);                         # Oppure Pro(0,x0)
```

Avendo in mano l'interprete I_W possiamo costruire un compilatore

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

tale che

$$\begin{aligned} \text{Comp}(P \in \text{PROG}) \equiv\ & x_2 := \text{cod}(P) \\ & x_1 := \langle x_1, x_2 \rangle \\ & I_W \quad . \end{aligned}$$

Questo significa che il compilatore non fa altro che cablare all'input x il programma RAM da interpretare e procede con l'esecuzione dell'interprete.

Vediamo se le tre proprietà di un compilatore sono soddisfatte:

- **programmabile**: sì, lo abbiamo appena fatto;

- **completo**: l'interprete riesce a riconoscere ogni istruzione RAM e la riesce a codificare;
- **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 quindi che

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

che è l'inclusione opposta del precedente risultato.

7.4.4. Conseguenze

Il risultato appena ottenuto ci permette di definire un teorema molto importante.

Teorema *(Teorema di Böhm-Jacopini (1970))* Per ogni programma con GOTO (RAM) ne esiste uno equivalente in un linguaggio strutturato (WHILE).

Questo teorema è fondamentale perché lega la programmazione a basso livello con quella ad alto livello. In poche parole, il GOTO può essere eliminato e la programmazione a basso livello può essere sostituita da quella ad alto livello.

Grazie alle due inclusioni dimostrate in precedenza abbiamo dimostrato anche 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, seppur profondamente diversi, i sistemi RAM e WHILE calcolano le stesse cose. Viene naturale chiedersi quindi quale sia la vera natura della calcolabilità.

Un altro risultato che abbiamo dimostrato *formalmente* è che nei sistemi di programmazione RAM e WHILE esistono funzioni **non calcolabili**, che sono nella *parte destra* della catena scritta sopra.

7.4.5. Interprete universale

Facciamo una mossa esotica: usiamo il compilatore da WHILE a RAM

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

sul programma I_w. *Lo possiamo fare?* Certo, posso compilare I_W perché è un programma WHILE.

Chiamiamo questo risultato

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

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.

Cosa abbiamo fatto vedere? Abbiamo appena mostrato che esiste un programma RAM in grado di **simulare** tutti gli altri programmi RAM. Questo programma viene detto **interprete universale**.

Considereremo *"buono"* un linguaggio se esiste un interprete universale per esso.

7.4.6. Concetto di calcolabilità generale

Ricordiamo che il nostro obiettivo è andare a definire la regione delle funzioni calcolabili e, di conseguenza, anche quella delle funzioni non calcolabili. Abbiamo visto che RAM e WHILE permettono di calcolare le stesse cose.

Possiamo definire ciò che è calcolabile a prescindere dalle macchine che usiamo per calcolare?

Come disse **Kleene**, vogliamo definire ciò che è calcolabile in termini più astratti, matematici, lontani dall'informatica. Se riusciamo a definire il concetto di calcolabile senza che siano nominate macchine calcolatrici, linguaggi e tecnologie ma utilizzando la matematica potremmo usare tutta la potenza di quest'ultima.

8. Richiami matematici: chiusura

8.1. Operazioni

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

$$\text{op} : \underbrace{U \times ... \times U}_{k} \longrightarrow U.$$

Il numero k indica l'**arietà** (o *arità*) dell'operazione, ovvero la dimensione del dominio dell'operazione.

8.2. Proprietà di chiusura

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

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

In poche parole, *se opero in A rimango 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.

8.3. Chiusura di un insieme

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

1. contiene A;
2. chiuso per op.

Quello che vogliamo fare è espandere A il minimo possibile per garantire la chiusura rispetto a op.

Due risposte ovvie a questo problema sono:

1. se A è chiuso rispetto a op, allora A stesso è l'insieme cercato;
2. sicuramente U soddisfa le due richieste, *ma è il più piccolo?*

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

1. $\forall a \in A \implies a \in A^{\text{op}}$;
2. $\forall a_1, ..., a_k \in A^{\text{op}} \implies \text{op}(a_1, ..., a_k) \in A^{\text{op}}$;
3. nient'altro sta in A^{op}.

Vediamo una definizione più *operativa* di A^{op}:

1. metti in A^{op} tutti gli elementi di A;
2. applica op a una k-tupla di elementi in A^{op};
3. se trovi un risultato che non è già in A^{op} allora aggiungilo ad A^{op};
4. ripeti i punti (2) e (3) fintantoché A^{op} cresce.

Siano $\Omega = \{\text{op}_1, ..., \text{op}_t\}$ un insieme di operazioni su U di arietà rispettivamente $k_1, ..., k_t$ e $A \subseteq U$ insieme. Definiamo **chiusura di A rispetto a Ω** il più piccolo sottoinsieme di U contenente A e chiuso rispetto a Ω, cioè l'insieme A^{Ω} definito come:

- $\forall a \in A \implies a \in A^{\Omega}$;
- $\forall i \in \{1, ..., t\} \quad \forall a_1, ..., a_{k_i} \in A^{\Omega} \implies \text{op}_i\left(a_1, ..., a_{k_i}\right) \in A^{\Omega}$;
- nient'altro è in A^{Ω}.

9. Calcolabilità

Seguiremo la seguente roadmap:

1. **ELEM**: definiamo un insieme di tre funzioni che *qualunque* idea di calcolabile si voglia proporre deve considerare calcolabili. ELEM non può esaurire il concetto di calcolabilità, quindi lo espanderemo con altre funzioni;
2. $\mathbf{\Omega}$: definiamo insieme di operazioni su funzioni che *costruiscono nuove funzioni*. Le operazioni in Ω sono banalmente implementabili e, applicandole a funzioni calcolabili, riesco a generare nuove funzioni calcolabili.
3. $\mathbf{ELEM^{\Omega}} = \mathcal{P}$: definiamo la classe delle **funzioni ricorsive parziali**. Questa sarà la nostra idea astratta della classe delle funzioni calcolabili secondo Kleene.

Quello che faremo dopo la definizione di $\mathcal{P}$ sarà chiederci se questa idea di calcolabile che abbiamo *"catturato"* coincida o meno con le funzioni presenti in $F(\text{RAM}) = F(\text{WHILE})$.

9.1. Primo passo: ELEM

Definiamo l'insieme ELEM con le seguenti funzioni:

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

Questo insieme è un *onesto punto di partenza*: sono funzioni basilari che qualsiasi idea data teoricamente non può non considerare come calcolabile. Ovviamente, ELEM non può essere considerato come l'idea teorica di *TUTTO* ciò che è calcolabile: infatti, la funzione $f(x) = x + 2$ non appartiene a ELEM ma è sicuramente calcolabile.

Quindi, ELEM è troppo povero e deve essere ampliato.

9.2. Secondo passo: Ω

Definiamo ora un insieme Ω di operazioni che amplino le funzioni di ELEM per permetterci di coprire tutte le funzioni calcolabili.

9.2.1. Composizione

Il primo operatore che ci viene in mente di utilizzare è quello di **composizione**. Siano:

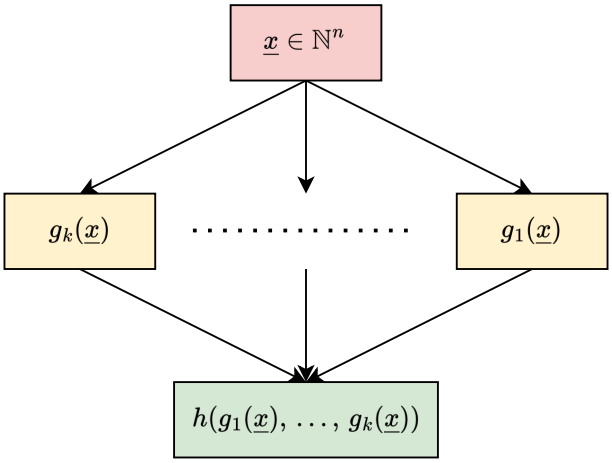
- $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ **funzione di composizione**,
- $g_1, ..., g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ *"funzioni intermedie"* e
- $\underline{x} \in \mathbb{N}^n$ input.

Allora definiamo

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

la funzione tale che

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



COMP è una funzione *intuitivamente calcolabile* se parto da funzioni calcolabili: infatti, prima eseguo le singole funzioni $g_1, ..., g_k$ e poi applico la funzione h sui risultati delle funzioni g_i.

Calcoliamo ora la **chiusura** di ELEM rispetto a COMP, ovvero l'insieme $\text{ELEM}^{\text{COMP}}$.

Vediamo subito come la funzione $f(x) = x + 2$ appartenga a questo insieme perché

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

Che altre funzioni ci sono in questo insieme? Sicuramente tutte le funzioni lineari del tipo $f(x) = x + k$, *ma la somma?* La funzione

$$\text{somma}(x, y) = x + y$$

non appartiene a questo insieme perché il valore y non è prefissato e non abbiamo ancora definito il concetto di iterazione di una funzione (in questo caso, la funzione successore).

Dobbiamo ampliare ancora $\text{ELEM}^{\text{COMP}}$ con altre operazioni.

9.2.2. Ricorsione primitiva

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

Siano:

- $g : \mathbb{N}^n \longrightarrow \mathbb{N}$ **funzione caso base**,
- $h : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$ **funzione passo ricorsivo** e
- $\underline{x} \in \mathbb{N}^n$ 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.

Come prima, **chiudiamo** $\text{ELEM}^{\text{COMP}}$ rispetto a RP, ovvero calcoliamo 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,

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

Altre funzioni che stanno in RICPRIM sono:

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

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

9.2.3. RICPRIM vs WHILE

L'insieme RICPRIM contiene molte funzioni, *ma abbiamo raggiunto l'insieme $F(\text{WHILE})$?*

Vediamo come è definita RICPRIM:

- $\forall f \in \text{ELEM} \implies f \in \text{RICPRIM}$;
- se $h, g_1, ..., g_k \in \text{RICPRIM} \implies \text{COMP}(h, g_1, ..., g_k) \in \text{RICPRIM}$;
- se $g, h \in \text{RICPRIM} \implies \text{RP}(g, h) \in \text{RICPRIM}$;
- nient'altro sta in RICPRIM.

Teorema RICPRIM $\subseteq F(\text{WHILE})$.

Dimostrazione

Passo base:

Le funzioni di ELEM sono ovviamente while programmabili, le avevamo mostrate in precedenza.

Passo induttivo:

Per COMP, assumiamo per ipotesi induttiva che $h, g_1, ..., g_k \in \text{RICPRIM}$ siano while programmabili, allora esistono $H, G_1, ..., G_k \in W\text{-PROG}$ tali che $\Psi_H = h, \Psi_{G_1} = g_1, ..., \Psi_{g_k} = g_k$. Mostro allora un programma WHILE che calcola COMP.

```
input(x)                  # In x1 inizialmente ho x
begin                     # nella forma <a1,...,an>
  x0 := G1(x1);
  x0 := [x0, G2(x1)];
  ...
  x0 := [x0, Gk(x1)];
  x1 := H(x0);
end
```

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

Per RP, assumiamo che $h, g \in \text{RICPRIM}$ siano while programmabili, allora esistono $H, G \in W\text{-PROG}$ 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

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

```
input(x,y)                  # In x1 inizialmente ho <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 quindi dimostrato che $\mathrm{RICPRIM} \subseteq F(\mathrm{WHILE})$, *ma questa inclusione è propria?*

Notiamo subito che nel linguaggio WHILE posso fare dei cicli infiniti, mentre in RICPRIM no: RICPRIM contiene solo funzioni totali (*si dimostra per induzione strutturale*) mentre WHILE contiene anche delle funzioni parziali. Di conseguenza

$$\mathrm{RICPRIM} \subsetneq F(\mathrm{WHILE}).$$

Per poter raggiungere $F(\mathrm{WHILE})$ dovremo ampliare nuovamente RICPRIM. Visto che le funzioni in RICPRIM sono tutte totali, possiamo dire che ogni ciclo in RICPRIM ha un inizio e una fine ben definiti: il costrutto utilizzato per dimostrare che $\mathrm{RP} \in F(\mathrm{WHILE})$ nella dimostrazione precedente, ci 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 che viene utilizzato è quello *originale*, cioè quel costrutto che si serve di una **variabile di controllo** che parte da un preciso valore e arriva ad un valore limite, senza che la variabile di controllo venga toccata. In Pascal veniva implementato mettendo la variabile di controllo in un registro particolare, per non permettere la sua scrittura.

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

Dato che WHILE vince su RICPRIM solo per i loop infiniti, restringiamo WHILE imponendo dei loop finiti. Creiamo l'insieme

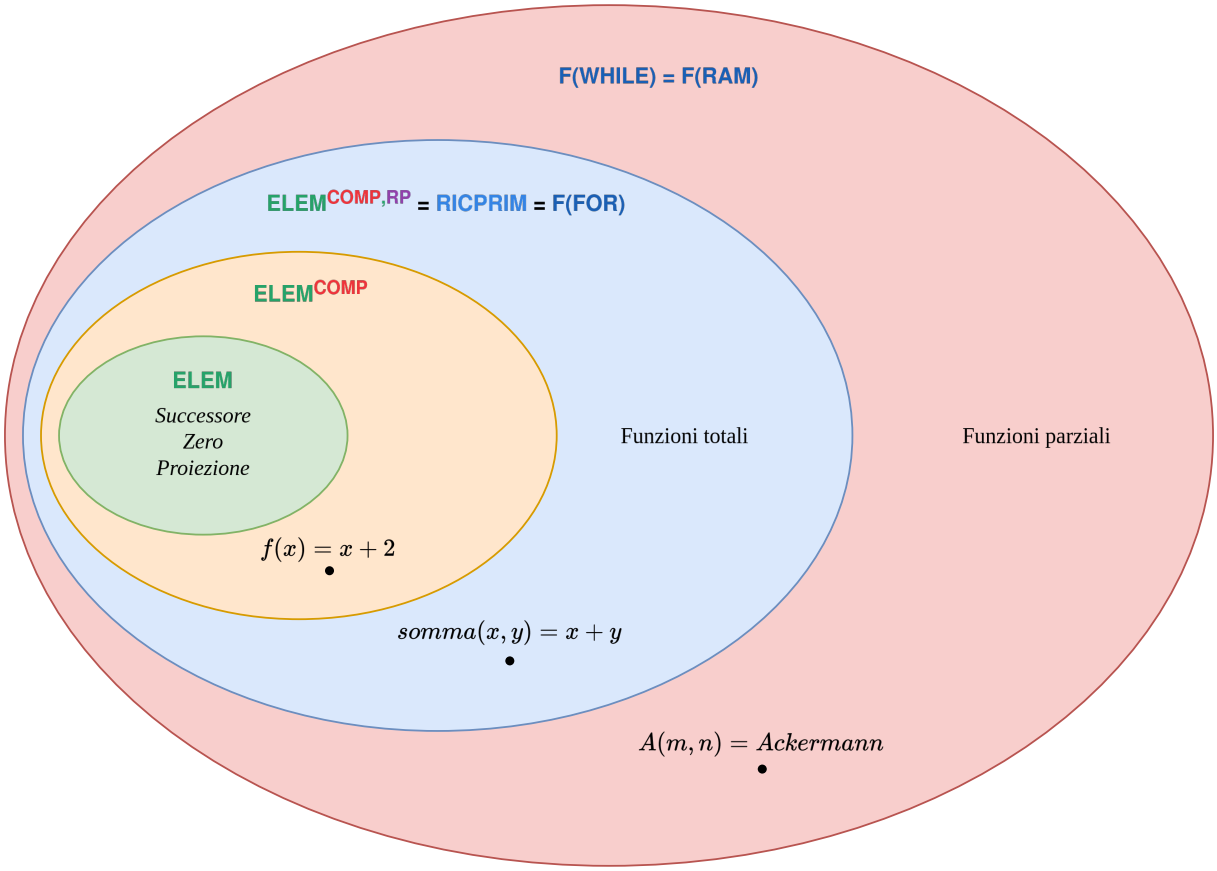
$$\widetilde{F}(\mathrm{WHILE}) = \{\Psi_W : W \in W\text{-PROG} \wedge \Psi_W \text{ totale}\}.$$

Dove si posizione questo insieme rispetto a RICPRIM? L'inclusione è propria?

Anche in questo caso, "vince" ancora WHILE, perché ci sono funzioni in $\widetilde{F}(\text{WHILE})$ che non sono scrivibili come funzioni in RICPRIM. Ad esempio, la funzione di Ackermann (1928), definita come

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

è una funzione che non appartiene a RICPRIM, perché a causa della doppia ricorsione cresce troppo in fretta. Di conseguenza, vogliamo ampliare anche RICPRIM.



Siamo nella direzione giusta: non abbiamo catturato *"cose strane"* che non avrei catturato in $F(\text{RAM})$, ma questo non basta: non abbiamo ancora catturato le funzioni parziali.

9.2.4. Minimalizzazione

Introduciamo quindi un ultimo operatore per permettere la presenza di funzioni parziali. L'operatore scelto è l'operatore di **minimalizzazione** di funzione. Sia $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ con $f(\underline{x}, y)$ e $\underline{x} \in \mathbb{N}^n$, allora:

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

Un'altra definizione di MIN è

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

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

Vediamo alcuni esempi con $f : \mathbb{N}^2 \to \mathbb{N}$.

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

9.3. $\mathcal{P}$

Ampliamo RICPRIM chiudendolo con la nuova operazione MIN:

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

Abbiamo ottenuto la classe delle **funzioni ricorsive parziali**. Vediamo qualche confronto con le classi che abbiamo già definito in precedenza.

9.3.1. $\mathcal{P}$ vs WHILE

Sicuramente $\mathcal{P}$, che grazie a MIN ora contiene anche funzioni parziali, amplia RICPRIM, fatto solo di funzioni totali. *Ma come si pone rispetto a* $F(\text{WHILE})$*?*

Teorema $\mathcal{P} \subseteq F(\text{WHILE})$.

Dimostrazione

$\mathcal{P}$ è definito per chiusura, ma in realtà è definito induttivamente in questo modo:

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

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

- **passo base**: le funzioni elementari sono WHILE programmabili, lo abbiamo già dimostrato;
- **passi induttivi**:
  - siano $h, g_1, ..., g_k \in \mathcal{P}$ WHILE programmabili per ipotesi induttiva, allora mostro che $\text{COMP}(h, g_1, ..., g_k)$ è WHILE programmabile, ma questo lo abbiamo già fatto per RICPRIM;
  - siano $h, g \in \mathcal{P}$ WHILE programmabili per ipotesi induttiva, allora mostro che $\text{RP}(h, g)$ è WHILE programmabile, ma anche questo lo abbiamo già fatto per RICPRIM;
  - sia $f \in \mathcal{P}$ WHILE programmabile per ipotesi induttiva, allora mostro che $\text{MIN}(f)$ è WHILE programmabile. Devo trovare un programma WHILE che calcoli la minimizzazione: il programma WHILE

$$
\begin{aligned}
P \equiv\ & \text{input}(\underline{x}) \\
& \text{begin} \\
& \quad y := 0 \\
& \quad \text{while } f(\underline{x}, y) \neq 0 \text{ do} \\
& \quad\quad y := y + 1 \\
& \text{end}
\end{aligned}
$$

è un programma che calcola la minimizzazione: infatti, se non esiste un y che azzeri $f(\underline{x}, y)$ il programma va in loop, quindi la semantica di P è $\perp$ secondo MIN.

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

Viene naturale chiedersi se vale la relazione inversa, cioè se $F(\text{WHILE}) \subseteq \mathcal{P}$ oppure no.

Teorema $F(\text{WHILE}) \subseteq \mathcal{P}$.

Dimostrazione
Sappiamo che

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

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

Le funzione in W-PROG sono nella forma

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

con $[\![C]\!](\underline{x}) = \underline{y}$ la funzione che calcola lo stato prossimo $\underline{y} \in \mathbb{N}^{21}$ a seguito dell'esecuzione del comando C a partire dallo stato corrente $\underline{x} \in \mathbb{N}^{21}$.

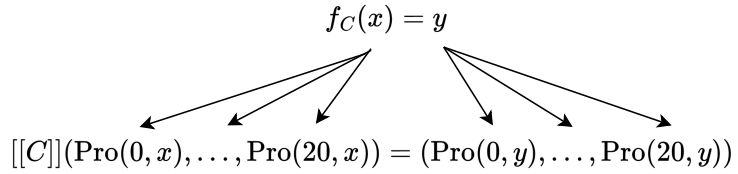
In sostanza, $[\![]\!]() : \mathbb{N}^{21} \longrightarrow \mathbb{N}^{21}$ rappresenta la funzione di stato prossimo definita induttivamente per via della struttura induttiva del linguaggio WHILE.

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

1. $\text{Pro}_0^{21} \in \text{ELEM} \Longrightarrow \text{Pro}_0^{21} \in \mathcal{P}$;
2. $\mathcal{P}$ è chiuso rispetto alla composizione;
3. a causa delle due precedenti, se dimostro che la funzione di stato prossimo è ricorsiva parziale allora $\Psi_W \in \mathcal{P}$ per la definizione induttiva di $\mathcal{P}$.

La funzione di stato prossimo restituisce elementi in $\mathbb{N}^{21}$, mentre gli elementi in $\mathcal{P}$ hanno codominio $\mathbb{N}$. Per risolvere questo piccolo problema tramite le liste di Cantor riesco a condensare il vettore in un numero.

Consideriamo quindi $f_C(x) = y$ **funzione numero prossimo**, con $x = [\underline{x}]$ e $y = \left[\underline{y}\right]$.



Ovviamente

$$f_C \in \mathcal{P} \iff [\![C]\!] \in \mathcal{P}$$

dato che posso passare da una all'altra usando funzioni in $\mathcal{P}$ quali Cantor e proiezioni.

Dimostriamo, tramite induzione strutturale, sul comando while C:

- **caso base**: i comandi base WHILE devono stare in $\mathcal{P}$.
  - **azzeramento** $C \equiv x_k := 0$:

    $$f_{x_k:=0}(x) = \left[\mathrm{Pro}(0,x),...,\underbrace{0(x)}_{k},...,\mathrm{Pro}(20,x)\right].$$

    Tutte le funzioni usate sono in $\mathcal{P}$, così come la loro composizione;
  - **incremento/decremento** $C \equiv x_k := x_j \pm 1$:

    $$f_{x_k:=x_j\pm1}(x) = [\mathrm{Pro}(x,0),...,\mathrm{Pro}(j,x) \pm 1,...,\mathrm{Pro}(20,x)].$$

    Tutte le funzioni usate sono in $\mathcal{P}$, così come la loro composizione;
- **passi induttivi**: i comandi "complessi" WHILE devono stare in $\mathcal{P}$.
  - **comando composto** $C \equiv$ begin $C_1;...;C_n$ end sapendo che $f_{C_i} \in \mathcal{P}$:

    $$f_C(x) = f_{C_n}\Big(...\Big(f_{C_2}\Big(f_{C_1}(x)\Big)\Big)...\Big).$$

    Ogni $f_{C_i} \in \mathcal{P}$ per ipotesi induttiva, così come la loro composizione;
  - **comando while** $C' \equiv$ while $x_k \neq 0$ do C, sapendo che $f_C \in \mathcal{P}$:

    $$f_{C'}(x) = f_C^{e(x)}(x),$$

    con

    $$e(x) = \mu_y(\mathrm{Pro}(k, f_c^y(x)) = 0).$$

    $f_C^{e(x)}$ è la composizione di f_C per $e(x)$ volte, che non è un numero costante dato che dipende dallo stato iniziale x, ma questo è problema: grazie all'operatore di composizione sappiamo comporre un numero predeterminato di volte, *ma come facciamo con un numero non costante?*

    Rinominiamo

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

    Come rappresento $T(x,y)$ in $\mathcal{P}$?

Notiamo come $T(x,y)$ sia un operatore ottenuto tramite RP su una funzione $f_C \in \mathcal{P}$, di conseguenza anche lei starà in $\mathcal{P}$.

L'ultima cosa da sistemare è $e(x)$. Questa funzione è la minimizzazione di $T(x,y)$: infatti,

$$e(x) = \mu_y(\text{Pro}(k, T(x,y)) = 0)$$

cerca il primo numero che azzera il registro k, quindi $e(x) \in \mathcal{P}$.

In conclusione,

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

è composizione di funzioni in $\mathcal{P}$, quindi $f_{C'} \in \mathcal{P}$.

□

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

$$F(\text{WHILE}) = \mathcal{P}.$$

Abbiamo ottenuto che la classe delle funzioni ricorsive parziali, che dà un'idea di *calcolabile* in termini matematici, coincide con quello che noi intuitivamente consideriamo *calcolabile*, dove con "intuitivamente calcolabile" intendiamo tutti quei problemi di cui vediamo una macchina che li risolva.

9.3.2. Tesi di Church-Turing

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

- $\mathcal{P}$ insieme delle **funzioni ricorsive parziali**;
- $\mathcal{T}$ insieme delle **funzioni ricorsive totali**.

Il secondo insieme presentato contiene tutte le funzioni di $\mathcal{P}$ che sono totali, ma allora

$$\mathcal{T} \subset \mathcal{P}.$$

Inoltre vale

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

perché, ad esempio, la funzione di Ackermann $\mathcal{A}(m,n)$ non sta in RICPRIM (già dimostrato) ma è sicuramente calcolabile e totale.

L'insieme $\mathcal{P}$ cattura tutti i sistemi di calcolo esistenti: WHILE, RAM, Macchine Di Turing, Lambda-calcolo di Church, paradigma quantistico, grammatiche, circuiti, sistemi di riscrittura, eccetera. In poche parole, tutti i sistemi creati dal 1930 ad oggi.

Infatti, dal 1930 in poi sono stati proposti un sacco di modelli di calcolo che volevano catturare ciò che è calcolabile, ma tutti questi modelli individuavano sempre la classe delle funzioni ricorsive parziali. Visti questi risultati, negli anni 1930/1940 **Church** e **Turing** decidono di enunciare un risultato molto importante.

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

Questa tesi non è un teorema, è una **congettura**, un'opinione. Non può essere un teorema in quanto non è possibile caratterizzare i modelli di calcolo ragionevoli che sono stati e saranno proposti in maniera completa. Possiamo semplicemente 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**, quindi che non va mai in loop.

10. Sistemi di programmazione

Fin'ora, nello studio dei **sistemi di programmazione**, ci siamo concentrati su una loro caratteristica principale: la *potenza computazionale*. Con la tesi di Church-Turing abbiamo affermato che ogni sistema di programmazione ha come potenza computazionale $\mathcal{P}$, cioè l'insieme delle funzioni ricorsive parziali. Oltre a questo, vorremmo sapere altro sui sistemi di programmazione, ad esempio la possibilità o l'impossibilità di scrivere programmi su certi compiti.

Vorremmo, come sempre, rispondere nel modo più rigoroso e generale possibile, quindi non considereremo un particolare sistema di programmazione, ma studieremo proprietà valide per tutti i sistemi di programmazione "ragionevoli":dDobbiamo astrarre un sistema di calcolo generale che permetta di rappresentarli tutti.

10.1. Assiomi di Rogers

Assiomatizzare significa *dare un insieme di proprietà* che i sistemi di calcolo devono avere per essere considerati buoni. Da qui in poi individueremo un sistema di programmazione con

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

ovvero l'insieme delle funzioni calcolabili con quel sistema, in altre parole l'insieme delle sue semantiche. Il pedice $i \in \mathbb{N}$ indica i programmi (*codificati*) di quel sistema.

Troveremo tre proprietà che un sistema di programmazione deve avere per essere considerato buono e lo faremo prendendo spunto dal sistema RAM.

10.1.1. Potenza computazionale

La prima proprietà che vogliamo in un sistema di programmazione riguarda la **potenza computazionale**. Dato il sistema $\{\varphi_i\}$ vogliamo che

$$\{\varphi\}_i = \mathcal{P}.$$

Questa proprietà è ragionevole, infatti non vogliamo considerare sistemi troppo potenti, che vanno oltre $\mathcal{P}$, o poco potenti, che sono sotto $\mathcal{P}$. Vogliamo la *giusta* potenza computazionale.

10.1.2. Interprete universale

La seconda proprietà che vogliamo in un sistema di programmazione riguarda la presenza di un **interprete universale**. Un interprete universale è un programma $\mu \in \mathbb{N}$ tale che

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

In sostanza è un programma scritto in un certo linguaggio che riesce a 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 permette la trasformazione di quest'ultimi.

10.1.3. Teorema S_1^1

L'ultima proprietà che vogliamo in un sistema di programmazione riguarda il soddisfacimento del teorema S_1^1. Questo teorema afferma che è possibile costruire automaticamente programmi specifici da programmi più generali, ottenuti fissando alcuni degli input.

Supponiamo di avere

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

Un programma RAM per questa funzione potrebbe essere

$$\begin{aligned} P \equiv R_2 &\longleftarrow \sin(R_1) \\ R_3 &\longleftarrow \mathrm{des}(R_1) \\ R_0 &\longleftarrow R_2 + R_3 \quad . \end{aligned}$$

Siamo in grado di produrre automaticamente un programma $\overline{P}$ che riceve in input solo x e calcola, ad esempio, $x+3$ a partire da P e 3?

$$(P,3) \rightsquigarrow S_1^1 \in \mathrm{PROG} \rightsquigarrow \overline{P}.$$

Per generare $\overline{P}$, potrei ad esempio fare

$$\begin{aligned} \overline{P} \equiv R_0 &\longleftarrow R_0 + 1 \\ R_0 &\longleftarrow R_0 + 1 \\ R_0 &\longleftarrow R_0 + 1 \\ R_1 &\longleftarrow \langle R_1, R_0 \rangle \\ R_0 &\longleftarrow 0 \\ P \quad &. \end{aligned}$$

Vediamo come questo programma segua principalmente quattro fasi:

1. si fissa il valore y in R_0;
2. si calcola l'input $\langle x,y \rangle$ del programma P;
3. si resetta la memoria alla situazione iniziale, tranne per il registro R_1;
4. si chiama il programma P.

In generale, il programma S_1^1 implementa la funzione

$$S_1^1(n,y) = \overline{n},$$

con n codifica di P e $\overline{n}$ codifica del nuovo programma $\overline{P}$, tale che

$$\varphi_{\overline{n}}(x) = \varphi_n(\langle x,y \rangle).$$

Questo teorema è molto comodo perché permette di calcolare facilmente la codifica $\overline{n}$: avendo n devo solo codificare le istruzioni iniziali di fissaggio di y, la funzione coppia di Cantor per creare l'input e l'azzeramento dei registri utilizzati. In poche parole,

$$S_1^1(n,y) = \overline{n} = \langle \underbrace{0,\dots,0}_{y}, s, t, n \rangle,$$

con s codifica dell'istruzione che calcola la funzione coppia di Cantor e t codifica dell'istruzione di azzeramento. S_1^1 è una funzione totale e programmabile, quindi $S_1^1 \in \mathcal{T}$ funzione **ricorsiva totale**.

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

1. il codice n di un programma che ha 2 input;
2. un valore y cui fissare il secondo input

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

Teorema Dato φ_i sistema RAM, esiste una funzione $S_1^1 \in \mathcal{T}$ tale che

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

Questo teorema ci garantisce un modo di usare l'algebra sui programmi.

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

Teorema Dato φ_i sistema RAM, esiste una funzione $S_n^m \in \mathcal{T}$ tale che per ogni programma $k \in \mathbb{N}$ e ogni input $\underline{x} \in \mathbb{N}^m$ e $\underline{y} \in \mathbb{N}^n$ vale

$$\varphi_k\big(\langle \underline{x},\underline{y}\rangle\big) = \varphi_{S_n^m(k,\underline{y})}(\langle \underline{x}\rangle).$$

10.2. Sistemi di programmazione accettabili (SPA)

Le tre caratteristiche che abbiamo identificato formano gli **assiomi di Rogers** (1953). Questi caratterizzano i sistemi di programmazioni su cui ci concentreremo, che chiameremo *Sistemi di Programmazione Accettabili*.

Questi assiomi non sono restrittivi: tutti i modelli di calcolo ragionevoli sono di fatto SPA.

10.3. Compilatori tra SPA

Sappiamo che esiste un compilatore da WHILE a RAM, *ma è l'unico?*

Dati gli SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, un compilatore dal primo al secondo è una funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$ 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, possiamo dire che i primi due punti possono essere scritti come $t \in \mathcal{T}$.

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

Dimostrazione
Consideriamo $\{\varphi_i\}$ e $\{\Psi_i\}$ due SPA. Valgono i tre assiomi di Rogers:

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

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

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

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

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

□

Notiamo la portata molto generale del teorema: non ci dice quale è il compilatore, ma ci dice che sicuramente esiste.

Corollario Dati gli SPA A, B, C esiste sempre un compilatore da A a B scritto nel linguaggio C.

Dimostrazione
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. □

In pratica, ciò vuol dire che per qualunque coppia di linguaggi, esistenti o che verranno progettati in futuro, sarò sempre in grado di scrivere un compilatore tra essi nel linguaggio che più preferisco. È un risultato assolutamente generale.

Un risultato più potente del teorema precedente è invece dato dal **teorema di Rogers**.

Teorema *(Teorema di isomorfismo tra SPA)* Dati due SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, esiste $t : \mathbb{N} \longrightarrow \mathbb{N}$ tale che:
1. $t \in \mathcal{T}$;
2. $\forall i \in \mathbb{N} \quad \varphi_i = \Psi_{t(i)}$;
3. t è invertibile, quindi t^{-1} può essere visto come un decompilatore.

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

10.4. Teorema di ricorsione

Introduciamo il **teorema di ricorsione**, un risultato utilissimo che utilizzeremo per rispondere ad alcuni quesiti sugli SPA.

Teorema Dato un SPA $\{\varphi_i\}$, per ogni $t : \mathbb{N} \longrightarrow \mathbb{N}$ ricorsiva totale vale

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

Diamo una chiave di lettura a questo teorema:
- consideriamo t come un programma che prende in input un programma n e lo cambia nel programma $t(n)$, anche nella maniera più assurda;
- il teorema dice che qualsiasi sia la natura di t, esisterà sempre almeno un programma il cui significato **non sarà stravolto** da t.

Dimostrazione
Siamo in un SPA $\{\varphi_i\}$ quindi valgono i tre assiomi di Rogers. D'ora in avanti, 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 con il mostrare che

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

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 $t(S_1^1(e,i))$: essa è ricorsiva totale in i perché composizione di t e di S_1^1 ricorsive totali, quindi

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

Abbiamo quindi mostrato che

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

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

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

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

10.5. Due quesiti sugli SPA

Ci poniamo due quesiti riguardo gli SPA:

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

   Ovviamente, questa operazione deve essere fatta senza aprire il file che contiene il listato.

   Questi programmi sono detti **Quine**, in onore del filosofo e logico Willard Quine (1908-2000) 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. Noi, però, vogliamo rispondere tramite una dimostrazione rigorosa, quindi 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 SPA al secondo SPA è una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che:
   - $t \in \mathcal{T}$ programmabile e totale;
   - $\forall i \in \mathbb{N} \quad \varphi_i = \Psi_{t(i)}$.

   Invece, un *compilatore completamente errato* è una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che:
   - $t \in \mathcal{T}$ programmabile e totale;
   - $\forall i \in \mathbb{N} \quad \varphi_i \neq \Psi_{t(i)}$.

10.5.1. Primo quesito: Quine

Consideriamo il programma RAM

$$\begin{aligned} P \equiv\ & R_0 \longleftarrow R_0 + 1 \\ & R_0 \longleftarrow R_0 + 1 \\ & \dots \\ & R_0 \longleftarrow R_0 + 1 \end{aligned}$$

che 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

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

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 sé stesso.

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

10.5.2. Secondo quesito: compilatori completamente errati

Supponiamo di avere in mano una funzione $t \in \mathcal{T}$ che *"maltratta"* i programmi.

Vediamo 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 i risultati $(*)$ 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*.

10.6. Equazioni su SPA

10.6.1. Strategia

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

Ad esempio, dato uno SPA $\{\varphi_i\}$ ci chiediamo se

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

La **strategia** da seguire per risolvere questo tipo di richieste è analoga a quella usata per la dimostrazione del teorema di ricorsione e può essere riassunta nei seguenti passaggi:

1. trasforma il membro di destra dell'equazione in una funzione $f(x, n)$;
2. mostra 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;
5. il quesito iniziale è diventato $\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{S_1^1(e,n)}(x)$?
6. la risposta è *SI* per il teorema di ricorsione.

Riprendiamo in mano l'esempio appena fatto.

Cominciamo con il trasformare la parte di destra:

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

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

Continuando, riscriviamo l'equazione come

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

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

Per il teorema di ricorsione possiamo concludere che

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

10.6.2. Esercizi

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

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

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

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

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

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

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

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

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

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

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

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

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

11. Problemi di decisione

Un **problema di decisione** è una domanda cui devo decidere se rispondere *SI* o *NO*.

I problemi di decisione sono costituiti da tre elementi principali:

- **nome**: il 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, devo rispondere *SI* se soddisfa la proprietà, *NO* altrimenti. In altre parole, è la **specifica** del problema di decisione.

Diamo ora una definizione più formale:

- **nome** Π;
- **istanza** $x \in D$ input;
- **domanda** $p(x)$ proprietà che $x \in D$ può soddisfare o meno.

Se, per esempio, mi viene chiesto *"questo polinomio ha uno zero?"* non devo dire *quale* zero ha, ma solo *se lo ha* o meno. Non devo esibire una **struttura** come risultato (cosa che avviene nei problemi di ricerca o di ottimizzazione), ma solo una risposta *SI/NO*.

11.1. Esempi

Vediamo qualche esempio di problemi di decisione noti. L'assegnamento di un valore a $x \in D$ genera un'**istanza particolare** del problema:

1. **Parità**

- Nome: Parità.
- Istanza: $n \in \mathbb{N}$.
- Domanda: n è pari?

2. **Equazione diofantea**

- Nome: Equazione Diofantea.
- Istanza: $a, b, c \in \mathbb{N}^+$.
- Domanda: $\exists x, y \in \mathbb{Z} \mid ax + by = c$?

Questa domanda nei reali avrebbe poco senso, perché ci sarebbero infiniti punti che soddisfano l'equazione della retta. Considerando punti interi, invece, ha più senso in quanto niente garantisce che la retta ne abbia.

Ad esempio, per $a = 3$, $b = 4$ e $c = 5$ rispondo *SI*, visto che la proprietà vale per $x = -1$ e $y = 2$.

Il nome di queste equazioni deriva dal matematico Diofanto, che per primo le trattò nel contesto dell'aritmetica di, appunto, Diofanto.

3. **Fermat**

- Nome: Ultimo Teorema di Fermat.
- Istanza: $n \in \mathbb{N}^+$.
- Domanda: $\exists x, y, z \in \mathbb{N}^+ \mid x^n + y^n = z^n$?

Questo problema è, in un certo senso, riconducibile al precedente.

Per $n = 1$ rispondo *SI*: è facile trovare tre numeri tali che $x + y = z$, ne ho infiniti.

Per $n = 2$ rispondo *SI*, i numeri nella forma $x^2 + y^2 = z^2$ rappresentano le **terne pitagoriche**.

Per $n \geq 3$ rispondo *NO*, è stato dimostrato da Eulero.

Questo problema è rimasto irrisolto per circa 400 anni, fino a quando nel 1994 viene dimostrato il **teorema di Andrew-Wiles** (dall'omonimo matematico), come banale conseguenza di una dimostrazione sulla modularità delle curve ellittiche.

Si dice che il primo a risolvere questo problema sia stato Fermat, giurista che nel tempo libero giocava con la matematica, tanto da meritarsi il nome di *principe dei dilettanti*, lo dimostra il fatto che questo teorema non ha nessuna conseguenza pratica, è totalmente "inutile".

4. **Raggiungibilità**

- Nome: Raggiungibilità.
- Istanza: grafo $G = (\{1, ..., n\}, E)$.
- Domanda: $\exists \pi$ cammino dal nodo 1 al nodo n?

5. **Circuito hamiltoniano**

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

Un **circuito hamiltoniano** è un circuito che coinvolge ogni nodo una e una sola volta.

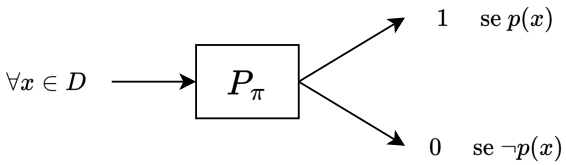
6. **Circuito euleriano**

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

Un **circuito euleriano** è un circuito che coinvolge ogni arco una e una sola volta. Questo quesito ha dato via alla teoria dei grafi.

11.2. 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**:

$$\Phi_\Pi : D \longrightarrow \{0, 1\}$$

tale che

$$\Phi_\Pi(x) = \begin{cases} 1 & \text{se } p(x) \\ 0 & \text{se } \neg p(x) \end{cases} .$$

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 parziale, è indifferente: una definizione implica l'altra.

Possiamo sfruttare questa cosa per sviluppare due tecniche di risoluzione del problema Decidibilità:

1. esibiamo un algoritmo di soluzione P_Π (anche inefficiente, basta che esista);
2. mostriamo che Φ_Π è ricorsiva totale.

11.3. Applicazione agli esempi

1. **Parità**

$$\Phi_{\text{PR}}(n) = 1 \dot{-} (n \bmod 2) \in \mathcal{T}.$$

2. **Equazione diofantea**

$$\Phi_{\text{ED}}(a, b, c) = 1 \dot{-} (c \bmod \text{mcd}(a, b)) \in \mathcal{T}.$$

3. **Fermat**

$$\Phi_F(n) = 1 \dot{-} (n \dot{-} 2) \in \mathcal{T}.$$

4. **Raggiungibilità**
   Sia $M_G \in \{0,1\}^{n \times n}$ matrice di adiacenza tale che $M_G[i,j] = 1$ se e solo se $(i,j) \in E$. Inoltre, M_G^k ha un 1 nella cella $[i,j]$ sse esiste un cammino lungo k da i a j.

$$\Phi_R(G) = \left(\bigvee_{k=0}^{n} M_G^k \right)[1,n] \in \mathcal{T}.$$

5. **Circuito hamiltoniano**
   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 circuito hamiltoniano rispondo *SI*;
   3. se nessuna permutazione p_i è un circuito hamiltoniano rispondo *NO*.

   L'algoritmo è inefficiente perché ci mette un tempo $O(n!)$, vista la natura combinatoria del problema, ma sicuramente questo problema è decidibile.

6. **Circuito euleriano**

Teorema *(Teorema di Eulero (1936))* Un grafo $G = (V, E)$ contiene un circuito euleriano se e solo se ogni vertice in G ha grado pari.

Grazie a questo risultato, l'algoritmo di risoluzione deve solo verificare se il grado di ogni vertice in V è pari, quindi anche questo problema è decidibile.

11.4. Problemi indecidibili

*Ma esistono dei **problemi indecidibili**?*

PEFFORZA

Se esistono programmi che non so scrivere allora esistono problemi per i quali non riesco a scrivere dei programmi che li risolvano.

11.4.1. Problema dell'arresto ristretto

Il **problema dell'arresto ristretto** per un programma P è un esempio di problema indecidibile.

Fissato P un programma, il problema è il seguente:

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

In altre parole, ci chiediamo se il programma P termina su input x. La risposta dipende ovviamente dal programma P, che può essere decidibile o non decidibile.

Ad esempio, se

$$\begin{aligned} P_1 \equiv\ & \mathrm{input}(x) \\ & x := x + 1; \\ & \mathrm{output}(x) \end{aligned}$$

allora la funzione

$$\Phi_{\mathrm{AR}_{P_1}}(x) = 1 \in \mathcal{T}$$

ci dice quando il problema P_1 termina o meno (*sempre*).

Se invece

$$\begin{aligned} P_2 \equiv\ & \mathrm{input}(x) \\ & \text{if } (x \bmod 2 \neq 0) \\ & \quad \text{while } (1 > 0); \\ & \mathrm{output}(x) \end{aligned}$$

allora la funzione

$$\Phi_{\mathrm{AR}_{P_2}}(x) = 1 \dot{-} (x \bmod 2) \in \mathcal{T}$$

ci dice quando il problema P_2 termina o meno.

Abbiamo quindi trovato due funzioni $\Phi_{\mathrm{AR}_P} \in \mathcal{T}$ che ci dicono quando i programmi P_1 e P_2 terminano o meno, *ma è sempre possibile?*

Prendiamo ora

$$\begin{aligned} \overset{\S}{P} \equiv\ & \mathrm{input}(x) \\ & z := U(x, x); \\ & \mathrm{output}(z), \end{aligned}$$

con U interprete universale tale che

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

Vale allora

$$\varphi_{\overset{\S}{P}}(x) = \varphi_U(x, x) = \varphi_x(x).$$

Abbiamo, quindi, un programma x che lavora su un altro programma (*se stesso*). Questo non è strano: compilatori, debugger, interpreti sono programmi che lavorano su programmi.

Come prima mi chiedo se φ_x su input x termina. A differenza di prima, ora il programma P non è fissato e dipende dall'input, essendo x sia input che programma.

Teorema Dato

$$\begin{aligned} \overset{\S}{P} \equiv\ & \text{input}(x) \\ & z := U(x,x); \\ & \text{output}(z), \end{aligned}$$

$\text{AR}_{\overset{\S}{P}}$ è indecidibile.

Dimostrazione
Per assurdo assumiamo $\text{AR}_{\overset{\S}{P}}$ decidibile. Dunque esiste

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

calcolabile da un programma che termina sempre. Visto che $\Phi_{\overset{\S}{P}} \in \mathcal{T}$, anche la funzione

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

è ricorsiva totale. Infatti, il programma

$$\begin{aligned} A \equiv\ & \text{input}(x) \\ & \text{if } \left(\Phi_{\text{AR}_{\overset{\S}{P}}(x)} == 0 \right) \\ & \quad \text{output}(0) \\ & \text{else} \\ & \quad \text{output}(U(x,x)+1) \end{aligned}$$

calcola esattamente la funzione $f(x)$.

Sia $\alpha \in \mathbb{N}$ la codifica del programma A, allora $\varphi_\alpha = f$. Valutiamo φ_α in α:

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

Tale funzione non può esistere, infatti:

- nel primo caso ho $\varphi_\alpha(\alpha) = 0$ se non termino, ma è una contraddizione perché $A \in \mathcal{T}$;
- nel secondo caso ho $\varphi_\alpha(\alpha) = \varphi_\alpha(\alpha) + 1$ se termino, ma è una contraddizione perché questa relazione non vale per nessun naturale.

Siamo ad un **assurdo**, quindi concludiamo che $\text{AR}_{\overset{\S}{P}}$ deve essere **indecidibile**. □

11.4.2. Problema dell'arresto

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

Teorema 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.

Dimostrazione

Assumiamo per assurdo che AR sia decidibile, ma allora esiste un programma $P_{\text{AR}}(x, y)$ che lo risolve, quindi restituisce:

- 1 se $\varphi_y(x) \downarrow$;
- 0 altrimenti.

Il seguente programma decide AR:

$$\begin{aligned} P \equiv\ & \text{input}(x, y) \\ & \text{output}(P_{\text{AR}}(x, x)) \quad . \end{aligned}$$

Il risultato precedente dimostra che questo problema è indecidibile, quindi non possono esistere programmi per la soluzione di AR. Siamo ancora ad un **assurdo**, quindi AR è **indecidibile**. □

12. Riconoscibilità automatica di insiemi

Proviamo a dare una *gradazione* sul livello di risoluzione dei problemi. Vogliamo capire se un dato problema:

- può essere risolto;
- non può essere risolto completamente (meglio di niente);
- non può essere risolto.

Costruiamo un programma che classifichi gli elementi di un insieme, quindi ci dica 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_A 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? Quali invece non lo sono?

Siamo certi che non tutti gli insiemi siano automaticamente riconoscibili, infatti grazie al concetto di *cardinalità* sappiamo che:

- i sottoinsiemi di $\mathbb{N}$ sono densi come $\mathbb{R}$;
- io non ho $\mathbb{R}$ programmi quindi sicuramente c'è qualche insieme che non lo è.

12.1. 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

$$\chi_A : \mathbb{N} \longrightarrow \{0, 1\}$$

tale che

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

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

$$\chi_A \in \mathcal{T} .$$

Che χ_A sia totale è banale, perché 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 χ_A, quindi $\chi_A \in \mathcal{T}$ perché esiste un programma che la calcola;
- $\chi_A \in \mathcal{T}$ quindi esiste un programma P_A che la implementa e che soddisfa la definizione data sopra.

12.1.1. 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 ad ogni insieme $A \subseteq \mathbb{N}$ possiamo associare il suo **problema di riconoscimento**, così definito:

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

La sua funzione soluzione

$$\Phi_{\mathrm{RIC}_A} : \mathbb{N} \longrightarrow \{0, 1\}$$

è tale che

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

Notiamo che la semantica del problema è proprio la funzione caratteristica, quindi $\Phi_{\mathrm{RIC}_A} = \chi_A$. Se A è ricorsivo, allora la sua funzione caratteristica è ricorsiva totale, ma lo sarà anche la funzione soluzione Φ e, di conseguenza, RIC_A è decidibile.

12.1.2. Decidibile vs Ricorsivo

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

Dato il problema

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

definiamo

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

insieme delle istanze a risposta positiva di Π. Notiamo che, se Π è decidibile allora $\Phi_\Pi \in \mathcal{T}$, quindi esiste un programma che calcola questa funzione. La funzione in questione è quella che riconosce automaticamente l'insieme A_Π, quindi A_Π è ricorsivo.

12.2. Insiemi non ricorsivi

Per trovare degli insiemi non ricorsivi cerco nei problemi di decisione non decidibili. L'unico problema di decisione non decidibile che abbiamo visto è il **problema dell'arresto ristretto** $\mathrm{AR}_{\underset{P}{\S}}$.

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

Definiamo l'insieme delle istanze a risposta positiva di $\mathrm{AR}_{\underset{P}{\S}}$

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

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 abbiamo dimostrato che il problema dell'arresto ristretto non è decidibile, quindi A non è ricorsivo.

12.3. Relazioni ricorsive

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

- la sua funzione caratteristica χ_R è tale che $\chi_R \in \mathcal{T}$, 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 la relazione

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

È molto simile al problema dell'arresto, ma non chiedo se P termina in generale, chiedo se termina in y passi. 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;
- **check del clock** per controllare l'arrivo alla quota y.

Definiamo quindi il programma

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

tale che

$$\begin{aligned}
\widetilde{U} \equiv\ & \text{input}(x, y) \\
& U(P, x) + \text{clock} \\
& \text{ad ogni passo di } U(P, x): \\
& \quad \text{if clock} > y: \\
& \quad\quad \text{output}(0) \\
& \quad \text{clock}++; \\
& \text{output}(\text{clock} == y) \quad .
\end{aligned}$$

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.

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

Possiamo definire l'insieme

$$B = \left\{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \mid \left(xR_{\overset{\S}{P}}y\right)\right\}.$$

Notiamo 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. $\overset{\S}{P}(x)$ termina in y passi, ma allora $xR_{\overset{\S}{P}}y$ e quindi $x \in B$;
- $B \subseteq A$: se $x \in B$ esiste y tale che $xR_{\overset{\S}{P}}y$, quindi $\overset{\S}{P}(x)$ termina in y passi, ma allora il programma $\overset{\S}{P} = x$ su input x termina, quindi $x \in A$.

12.4. Insiemi ricorsivamente numerabili

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

Il programma che lista gli elementi di A è:

$$
\begin{aligned}
P \equiv\ & i := 0; \\
& \text{while } (1 > 0)\ \{ \\
& \quad \text{output}(F(i)) \\
& \quad i := i + 1; \\
& \}
\end{aligned}
$$

Per alcuni insiemi non è possibile riconoscere tutti gli elementi che gli appartengono, ma può essere che si conosca un modo per elencarli. Alcuni insiemi invece non hanno nemmeno questa proprietà.

Se il meglio che posso fare per avere l'insieme A è listarlo con P, *come posso scrivere un algoritmo che "tenta di riconoscere" A?* Questo algoritmo deve listare tutti gli elementi senza un clock di timeout: se inserissi un clock avrei un insieme ricorsivo per la relazione R_P mostrata in precedenza.

Vediamo il programma di **massimo riconoscimento**:

$$
\begin{aligned}
P \equiv\ & \text{input}(x) \\
& i := 0; \\
& \text{while } (F(i) \neq x) \\
& \quad i := i + 1; \\
& \text{output}(1) \quad .
\end{aligned}
$$

Come viene riconosciuto l'insieme 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**.

Se avessi indicazioni sulla monotonia della routine F allora avrei sicuramente un insieme ricorsivo. In questo caso non assumiamo niente quindi rimaniamo negli insiemi ricorsivamente numerabili.

12.4.1. Definizione formale

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

- $A = \varnothing$ oppure
- $A = \text{Im}_f$, con $f : \mathbb{N} \longrightarrow \mathbb{N} \in \mathcal{T}$, ovvero $A = \{f(0), f(1), f(2), ...\}$.

Visto che f è ricorsiva totale esiste un/a programma/routine F che la implementa e che usiamo per il parziale riconoscimento di A: questo programma, se $x \in A$, prima o poi mi restituisce 1, altrimenti entra in loop.

È come avere un *libro con infinite pagine*, su ognuna delle quali compare un elemento di A. Il programma di riconoscimento P, grazie alla routine F, non fa altro che sfogliare le pagine i di questo libro alla ricerca di x:

- se $x \in A$ prima o poi x compare nelle pagine del libro come $F(i)$;
- se $x \notin A$ sfoglio il libro all'infinito, non sapendo quando fermarmi.

12.4.2. Caratterizzazioni

Teorema Le seguenti definizioni sono equivalenti:

1. A è ricorsivamente numerabile, con $A = \text{Im}_f$ e $f \in \mathcal{T}$ funzione ricorsiva totale;
2. $A = \text{Dom}_f$, con $f \in \mathcal{P}$ funzione ricorsiva parziale;

3. esiste una relazione $R \subseteq \mathbb{N}^2$ ricorsiva tale che $A = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \mid (x, y) \in R\}$.

Dimostrazione
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 e il suo algoritmo di parziale riconoscimento P, definiti in precedenza. Vista la definizione di P, abbiamo 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 ho 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 $\varphi_P = f$. Considero allora la relazione

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

che abbiamo dimostrato prima essere ricorsiva. Definiamo

$$B = \{x \in \mathbb{N} \mid \exists y \mid (x, y) \in R_P\}.$$

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 sta nel dominio di $f = \varphi_P$, quindi $x \in A$.

$3 \implies 1$

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

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

$$
\begin{aligned}
P \equiv\ & \text{input}(n) \\
& x := \sin(n); \\
& y := \text{des}(n); \\
& \text{if } (P_R(x,y) == 1) \\
& \quad \text{output}(x) \\
& \text{else} \\
& \quad \text{output}(a)
\end{aligned}
$$

che 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 ci è più comoda.

Nell'esperienza del Prof. Carlo Mereghetti, è molto utile e comodo il punto 2. In ordine:

1. scrivo un programma P che restituisce 1 su input $x \in \mathbb{N}$, altrimenti va in loop se $x \notin A$:
$$
P(x) = \begin{cases} 1 & \text{se } x \in A \\ \perp & \text{se } x \notin A \end{cases};
$$
2. la semantica di P è quindi tale che:
$$
\varphi_P(x) = \begin{cases} 1 \text{ se } x \in A \\ \perp \text{ se } x \notin A \end{cases};
$$
3. la funzione calcolata è tale che
$$
\varphi_P \in \mathcal{P},
$$
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.

12.5. Insiemi ricorsivamente numerabili ma non ricorsivi

Un esempio di insieme che non è ricorsivo, ma è ricorsivamente numerabile, è identificato 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 ristretto sarebbe decidibile.

Tuttavia, questo insieme è **ricorsivamente numerabile**: infatti, il programma

$$\begin{aligned} P \equiv\ & \text{input}(x) \\ & U(x,x); \\ & \text{output}(1) \end{aligned}$$

decide parzialmente A. Come possiamo vedere, 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

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

Dato che $A = \text{Dom}_{\varphi_P \in \mathcal{P}}$ posso applicare la seconda caratterizzazione data nella lezione precedente per dimostrare che l'insieme A è un insieme ricorsivamente numerabile.

Alternativamente, possiamo dire che

$$A = \{x \in \mathbb{N} \mid \varphi_x(x)\downarrow\} = \left\{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \mid (x,y) \in R^{\overset{\S}{}}_{P}\right\},$$

con

$$R^{\overset{\S}{}}_{P} = \left\{(x,y) \mid \overset{\S}{P} \text{ su input } x \text{ termina entro } y \text{ passi}\right\}$$

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

Come sono messi i due insiemi?

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

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

Il programma P è del tipo

$$\begin{aligned} P \equiv\ & \text{input}(x) \\ & \text{if}\left(P_{A(x)} == 1\right) \\ & \qquad \text{output}(1) \\ & \text{else} \\ & \qquad \text{while}(1 > 0); \end{aligned} .$$

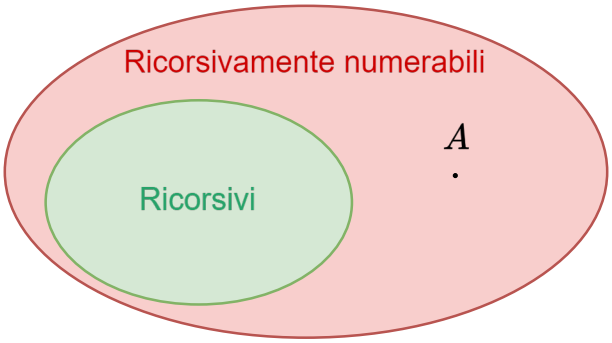
La semantica di questo programma è

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

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

$$\text{Ricorsivi} \subset \text{Ricorsivamente numerabili}\ .$$



Esistono insiemi che non sono ricorsivamente numerabili?

12.6. 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 La classe degli insiemi ricorsivi è un'Algebra di Boole, ovvero è chiusa per complemento, intersezione e unione.

Dimostrazione
Siano A, B due insiemi ricorsivi. Allora esistono dei programmi P_A, P_B che li riconoscono o, equivalentemente, esistono $\chi_A, \chi_B \in \mathcal{T}$.

È facile dimostrare che le operazioni di unione, intersezione e complemento sono facilmente implementabili da programmi che terminano sempre. Di conseguenza,

$$A \cup B, A \cap B, A^C$$

sono ricorsive.

Vediamo questi tre programmi:

- **complemento**

$$P_{A^C} \equiv \text{input}(x)$$
$$\text{output}(1 \dot{-} P_A(x)).$$

- **intersezione**

$$P_{A\cap B} \equiv \text{input}(x)$$
$$\text{output}(\text{MIN}(P_A(x), P_B(x))).$$

- **unione**

$$P_{A\cup B} \equiv \text{input}(x)$$
$$\text{output}(\max(P_A(x), P_B(x))).$$

Allo stesso modo possiamo trovare le funzioni caratteristiche delle tre operazioni:

- $\chi_{A^C}(x) = 1 \dot{-} \chi_A(x)$;
- $\chi_{A\cap B} = \chi_A(x) \cdot \chi_B(x)$;
- $\chi_{A\cup B} = 1 \dot{-} (1 \dot{-} \chi_A(x))(1 \dot{-} \chi_B(x))$.

Tutte queste funzioni sono ricorsive totali, quindi anche le funzioni A^C, $A \cap B$, $A \cup B$ sono ricorsive totali. □

Ora, però, vediamo un risultato molto importante riguardante nello specifico il complemento dell'insieme dell'arresto A^C che abbiamo definito prima.

Teorema A^C non è ricorsivo.

Dimostrazione
Se A^C fosse ricorsivo, per la proprietà di chiusura dimostrata nel teorema precedente, avremmo

$$\left(A^C\right)^C = A$$

ricorsivo, il che è assurdo. □

Ricapitolando abbiamo:

- $A = \{x : \varphi_x(x) \downarrow\}$ ricorsivamente numerabile, ma non ricorsivo;
- $A^C = \{x : \varphi_x(x) \uparrow\}$ non ricorsivo.

L'insieme A^C Potrebbe essere ricorsivamente numerabile?

Teorema Se A è ricorsivamente numerabile e A^C è ricorsivamente numerabile allora A è ricorsivo.

Dimostrazione

INFORMALE

Essendo 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 utilizzare il seguente 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, voltiamo la pagina di ogni libro e ricominciamo.

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

Essendo A e A^C ricorsivamente numerabili, esistono $f, g \in \mathcal{T}$ tali che $A = \text{Im}_f \wedge A^C = \text{Im}_g$. Sia f implementata dal programma F e g dal programma G. Il seguente programma riconosce A:

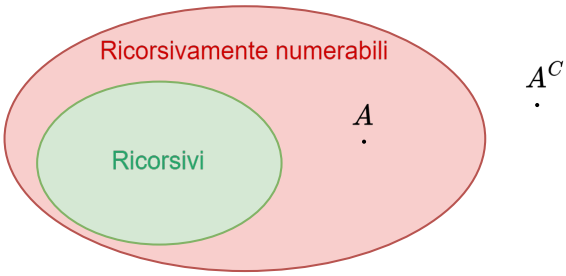
$$
\begin{aligned}
P \equiv\ & \text{input}(x) \\
& i := 0; \\
& \text{while(true)} \\
& \quad \text{if } (F(i) = x)\ \text{output}(1); \\
& \quad \text{if } (G(i) = x)\ \text{output}(0); \\
& \quad i := i + 1;
\end{aligned}
$$

Questo algoritmo termina per ogni input, in quanto $x \in A$ o $x \in A^C$. Possiamo concludere che l'insieme A è ricorsivo. □

Concludiamo immediatamente che A^C **non** può essere ricorsivamente numerabile.

In generale, questo teorema ci fornisce uno strumento molto interessante per 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.



12.7. Chiusura degli insiemi ricorsivamente numerabili

Teorema La classe degli insiemi ricorsivamente numerabili è chiusa per unione e intersezione, ma non per complemento.

Dimostrazione

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, perciò, $f, g \in \mathcal{T} \mid A = \text{Im}_f \wedge B = \text{Im}_g$. Sia f implementata da F e g implementata da G. Siano

$$
\begin{aligned}
P_i \equiv\ & \text{input}(x);\\
& i := 0;\\
& \text{while}(F(i) \neq x)\\
& \quad i++;\\
& i := 0;\\
& \text{while}(G(i) \neq x)\\
& \quad i++;\\
& \text{output}(1);
\end{aligned}
\qquad
\begin{aligned}
P_u \equiv\ & \text{input}(x);\\
& i := 0;\\
& \text{while}(\text{true})\\
& \quad \text{if } (F(i) = x)\\
& \quad\quad \text{output}(1);\\
& \quad \text{if } (G(i) = x)\\
& \quad\quad \text{output}(1);\\
& \quad i++;
\end{aligned}
$$

i due programmi che calcolano rispettivamente $A \cap B$ e $A \cup B$. Le loro semantiche sono

$$
\varphi_{P_i} = \begin{cases} 1 & \text{se } x \in A \cap B \\ \bot & \text{altrimenti} \end{cases}
\qquad
\varphi_{P_u} = \begin{cases} 1 & \text{se } x \in A \cup B \\ \bot & \text{altrimenti} \end{cases}
$$

da cui ricaviamo che

$$
A \cap B = \text{Dom}_{\varphi_{P' \in \mathcal{P}}}
\qquad
A \cup B = \text{Dom}_{\varphi_{P'' \in \mathcal{P}}}
$$

I due insiemi sono quindi ricorsivamente numerabili per la seconda caratterizzazione. □

12.8. Teorema di Rice

Il **teorema di Rice** è un potente strumento per 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

$$(a \in I \land \varphi_a = \varphi_b) \Longrightarrow b \in I.$$

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,

$$\underbrace{a \in I}_{\varphi_a(3)=5} \land \underbrace{\varphi_a = \varphi_b}_{\varphi_b(3)=5} \Rightarrow b \in I.$$

Teorema *(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 ci dice che gli insiemi che rispettano le funzioni non sono mai ricorsivi, tolti i casi banali $\varnothing$ e $\mathbb{N}$.

Dimostrazione 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 $\overline{a} \notin I$.

Definiamo la funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$ come:

$$t(n) = \begin{cases} \overline{a} & \text{se } n \in I \\ a & \text{se } n \notin I \end{cases}.$$

Sappiamo che $t \in \mathcal{T}$ dato che è calcolabile dal programma

$$\begin{aligned} P \equiv\ & \text{input}(x); \\ & \text{if}(P_I(n) = 1) \\ & \quad \text{output}(\overline{a}); \\ & \text{else} \\ & \quad \text{output}(a) \end{aligned}$$

Visto che $t \in \mathcal{T}$, 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 e $\varphi_d = \varphi_{t(d)}$ allora $t(d) \in I$. Ma $t(d \in I) = \overline{a} \notin I$, 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.

Assumere I ricorsivo ha portato ad un assurdo, quindi I non è ricorsivo. □

12.8.1. 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.

12.8.2. Limiti alla verifica automatica del software

Definiamo:

- **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

$$\varphi_V(P) = \begin{cases} 1 & \text{se } P \text{ è corretto} \\ 0 & \text{se } P \text{ è errato} \end{cases}.$$

Definiamo

$$\text{PC} = \{P \mid P \text{ è corretto}\}.$$

Osserviamo che esso 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 $\mathrm{PC} = \mathbb{N}$.

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*).

Parte II — Teoria della complessità

1. Richiami matematici: teoria dei linguaggi formali

Dato un problema P, finora ci siamo chiesti *"**esiste** un programma per la sua soluzione automatica?"* Tramite questa domanda abbiamo potuto indagare la **teoria della calcolabilità**, il cui oggetto di studio è l'esistenza (*o meno*) di un programma per un dato problema.

In questa parte del corso studieremo la **teoria della complessità**, in cui entra in gioco una seconda investigazione: *"**come** funzionano i programmi per P?"*

Per rispondere a questa domanda, vogliamo sapere quante **risorse computazionali** utilizziamo 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 il programma 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?

Prima di iniziare, diamo una breve introduzione alla **teoria dei linguaggi formali**.

1.1. Alfabeto, stringhe e linguaggi

Un **alfabeto** è un insieme finito di simboli $\Sigma = \{\sigma_1, ..., \sigma_k\}$. Un alfabeto binario è un qualsiasi alfabeto composto da due soli simboli.

Una **stringa** su Σ è una sequenza di simboli di Σ nella forma $x = x_1 \ ... \ x_n$, con $x_i \in \Sigma$.

La **lunghezza** di una stringa x indica il numero di simboli che la costituiscono e si indica con $|x|$. Una stringa particolare è la **stringa nulla**, che si indica con ε ed è tale che $|\varepsilon| = 0$.

Indichiamo con Σ^* l'insieme delle stringhe che si possono costruire sull'alfabeto Σ, compresa la stringa nulla. L'insieme delle stringhe formate da almeno un carattere è definito da $\Sigma^+ = \Sigma^* / \{\varepsilon\}$.

Un **linguaggio** L su un alfabeto Σ è un sottoinsieme $L \subseteq \Sigma^*$, che può essere finito o infinito.

2. Macchina di Turing deterministica (DTM)

Il punto di partenza dello studio della teoria della complessità è la definizione rigorosa delle risorse di calcolo e di come possono essere misurate.

Il modello di calcolo che useremo nel nostro studio è la **Macchina di Turing**, ideata da Alan Turing nel 1936. Essa è 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;

2.1. Struttura

Una **macchina di Turing deterministica** è un dispositivo hardware fornito di:

- **nastro di lettura e scrittura**: nastro infinito formato da celle, ognuna delle quali ha un proprio indice/indirizzo e contiene 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 ad ogni passo;
- **controllo a stati finiti**: automa a stati finiti $Q = \{Q_0, ..., Q_n\}$ 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 testina. 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.

2.1.1. 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 posiziona 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. Diciamo 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.

2.1.2. Definizione formale

Una macchina di turing deterministica è una tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, 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** ove 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 nostro nastro. Vale $\Sigma \subset \Gamma$ perché Γ contiene il simbolo *blank*;
- $\delta : Q \times \Gamma \longrightarrow Q \times (\Gamma / \{\text{blank}\}) \times \{-1, 0, +1\}$: **funzione di transizione** che definisce le mosse. È 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 $x \in \Sigma^*$, visto solo informalmente:

- **inizializzazione**:
  - il nastro contiene la stringa $x = x_1 ... x_n$;
  - la testina è posizionata sul carattere x_1;
  - il controllo a stati finiti parte dallo stato q_0;
- **computazione**: sequenza di mosse definite dalla funzione di transizione δ che manda, ad 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 la computazione si arresta in uno stato $q \in F$.

Come prima, $L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\}$ è ancora il **linguaggio accettato** da M.

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.

2.1.3. Configurazione di una DTM

Come per le macchine RAM, proviamo a dare l'idea di **configurazione** delle DTM. Anche qui, possiamo vederla come una foto che descrive completamente M in un certo istante. Questa definizione ci permette di descrivere la computazione come una serie di configurazioni/foto.

Ciò che ci serve ricordare è:

- in che stato siamo;
- 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 del nastro;
- $w \in \Gamma^*$: contenuto non-blank del nastro.

All'inizio della computazione abbiamo la **configurazione iniziale** $C_0 = (q_0, 1, x)$.

Diciamo che una configurazione C è **accettante** se $C = (q \in F, k, w)$ ed è **d'arresto** se $C = (q, k, w)$ con $\delta(q, w) = \perp$.

2.1.4. Definizione computazione tramite configurazioni

La computazione di M su $x \in \Sigma^*$ è la sequenza

$$C_0 \xrightarrow{\delta} C_1 \xrightarrow{\delta} \cdots \xrightarrow{\delta} C_i \xrightarrow{\delta} C_{i+1} \xrightarrow{\delta} \cdots,$$

dove, $\forall i \geq 0$, 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. Altre versioni delle macchine di Turing

2.2.1. 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.

Nelle **macchine di Turing non deterministiche** NDTM, data una configurazione C_i, può non essere unica la configurazione successiva, quindi non è determinata univocamente.

Nelle **macchine di Turing probabilistiche** PTM, data una configurazione C_i, possono esistere più configurazioni nelle quali possiamo entriamo, 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:

- $|\alpha_i| \leq 1$;
- hanno probabilità $|\alpha_i|^2$;
- le probabilità sommano a 1.

2.2.2. Versione semplificata

Esibire, progettare e comprendere una DTM è difficile anche in casi molto semplici, perché dobbiamo dettagliare stati, alfabeti, transizioni, eccetera. Solitamente, nel descrivere una DTM, si utilizza 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.

2.2.3. Esempio: parità

- Nome: parità.
- Istanza: $x \in \mathbb{N}$.
- Domanda: x è pari?

Come codifica utilizziamo quella *binaria*, ovvero

$$\text{cod} : \mathbb{N} \longrightarrow \{0, 1\}^*.$$

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 sia un algoritmo deterministico che riconosce proprio 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;
- $\Gamma = \{0, 1, \text{blank}\}$ alfabeto di lavoro;
- $q_0 = p$ stato iniziale;
- $F = \{z_1, z\}$ insiemi degli stati finali;
- $\delta : Q \times \Gamma \longrightarrow Q \times \Sigma \times \{-1, 0, 1\}$ 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	$\perp$	$\perp$	$\perp$

Notiamo come, anche per un problema così semplice, abbiamo una funzione di transizione abbastanza complicata. Andiamo quindi a utilizzare uno pseudocodice:

```
PARITÀ(n):
 1  i := 1;
 2  f := false;
 3  switch(x[i]) {
 4        case 0:
 5              i++;
 6              f := (x[i] == blank);
 7              break;
 8        case 1:
 9              do {
10                    f := (x[i] == 0);
11                    i++;
12              } while (x[i] != blank);
13  }
14  return f;
```

Alla fine della sua esecuzione avremo:

- se $x \in L_{\text{PARI}}$ allora True ;
- se $x \notin L_{\text{PARI}}$ allora False .

3. Funzionalità di una DTM

- 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 gli insiemi: dato $A \subseteq \mathbb{N}$, *come lo riconosco con una DTM?*
  L'idea che viene in mente è 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.

  $$A \rightsquigarrow \boxed{\text{cod}} \rightsquigarrow L_A = \{\text{cod}(a) : a \in A\}$$

  Un insieme A è riconoscibile da una DTM sse esiste una DTM M tale che $L_A = L_M$.

  Quando facciamo riconoscere un insieme A a una DTM M, possiamo trovarci in 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 $q \notin F \Rightarrow A$ è ricorsivo;
     - andare in loop $\Rightarrow A$ è ricorsivamente numerabile.

  Teorema La classe degli insiemi riconosciuti da 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 La classe degli insiemi riconosciuti da algoritmi deterministici coincide con la classe degli insiemi ricorsivi.

- Una seconda funzionalità delle DTM è che possono risolvere dei **problemi di decisione**. Vediamo il procedimento per farlo.

  Dato problema Π, con istanza $x \in D$ e domanda $p(x)$, andiamo a codificare gli elementi di D in elementi di Σ^*, ottenendo $L_\Pi = \{\text{cod}(x) \mid x \in D \wedge p(x)\}$ **insieme delle istanze a risposta positiva** di Π.

  La DTM risolve Π sse 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.

3.1. DTM per problemi di decisione

3.1.1. Definizione

Dato un problema Π con istanza $x \in D$ e domanda $p(x)$, la DTM M risolve Π se e solo se M è un **algoritmo deterministico** per L_Π, ovvero:
- se $p(x)$ allora M accetta $\text{cod}(x)$;
- se $\neg p(x)$ allora M si arresta su $\text{cod}(x)$ senza accettare.

3.1.2. Esempio: parità

- Nome: parità.

- Istanza: $x \in \mathbb{N}$.
- Domanda: x è pari?

Come codifica utilizziamo quella *binaria*, ovvero

$$\text{cod} : \mathbb{N} \longrightarrow \{0,1\}^*.$$

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 sia un algoritmo deterministico che riconosce proprio 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;
- $\Gamma = \{0, 1, \text{blank}\}$ alfabeto di lavoro;
- $q_0 = p$ stato iniziale;
- $F = \{z_1, z\}$ insiemi degli stati finali;
- $\delta : Q \times \Gamma \longrightarrow Q \times \Sigma \times \{-1, 0, 1\}$ 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	$\perp$	$\perp$	$\perp$

3.2. DTM per calcolare funzioni

3.2.1. Definizione

Oltre a riconoscere linguaggi, riconoscere insiemi e risolvere problemi di decisione, le DTM sono anche in grado di **calcolare funzioni**. Questo è un risultato molto importante, in quanto sappiamo che calcolare funzioni significa risolvere problemi del tutto generali, quindi non solo di decisione.

Data una funzione $f : \Sigma^* \longrightarrow \Gamma^*$, 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.*

3.2.2. Potenza computazionale

È possibile 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** (sistemi di programmazione accettabili), 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.

Analogamente, per gli insiemi possiamo affermare che:

- la classe degli insiemi riconosciuti da DTM coincide con la classe degli insiemi ricorsivamente numerabili;
- la classe degli insiemi riconosciuti da algoritmi deterministici coincide con la classe degli insiemi ricorsivi.

3.3. Crescita asintotica

3.3.1. Introduzione

Quello che facciamo nella teoria della complessità è chiederci *"quanto costa questo programma?"*

Considerando il tempo, abbiamo due diversi tipi di costo: *"poco"* oppure *"troppo"*. Capiremo durante la lezione il perché delle virgolette.

Stiamo valutando $t(n)$, ovvero

$$t(n) = \max\{T(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

il massimo numero di iterazioni di una DTM nell'eseguire input di grandezza n.

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.

Siano ad esempio t_1, t_2 le due potenze computazionali 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? La risposta è *dipende*: se considero n abbastanza piccoli allora t_2 è migliore perché i coefficienti ammortizzano il valore di n^2, mentre se considero n sufficientemente grandi è migliore t_1.

Date due complessità, non le devo valutare per precisi valori di n, ma devo valutare il loro **andamento asintotico**, ovvero quando n tende a $+\infty$.

3.3.2. Simboli di Landau

Riprendiamo i simboli matematici che abbiamo già visto in molti altri corsi che ci permettono di stabilire degli **ordini di grandezza** fra le funzioni, in modo da poterle paragonare.

Questi simboli si chiamano **simboli di Landau**, e i più utilizzati sono i seguenti tre:

1. O: letto *"O grande"*, date due funzioni $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ diciamo che

   $$f(n) = O(g(n))$$

   se e solo se

   $$\exists c > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

   Questo simbolo dà un **upper bound** alla funzione f.
2. Ω: letto *"Omega grande"*, date due funzioni $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ diciamo che

   $$f(n) = \Omega(g(n))$$

   se e solo se

   $$\exists c > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 \quad f(n) \geq c \cdot g(n).$$

   Questo simbolo dà un **lower bound** alla funzione f.
3. Θ: letto *"teta grande"*, date due funzioni $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ diciamo che

   $$f(n) = \Theta(g(n))$$

   se e solo se $\exists c_1, c_2 > 0 \quad \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Si può notare facilmente che valgono la proprietà

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

e la proprietà

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \land f(n) = \Omega(g(n)).$$

Noi useremo spesso O perché ci permette di definire il *worst case*.

3.4. Classificazione di funzioni

3.4.1. Introduzione

Con l'uso di questa notazione è possibile classificare le funzioni in una serie di classi in base a come è fatta la loro funzione soluzione $t(n)$.

Data una funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$, possiamo avere:

Tempo	Definizione formale	Esempio
Costante	$t(n) = O(1)$	Segno di un numero in binario
Logaritmica	$f(n) = O(\log(n))$	Difficile fare esempi per questo perché quasi mai riusciamo a dare una risposta leggendo $\log(n)$ input dal nastro
Lineare	$f(n) = O(n)$	Parità di un numero in binario
Quadratica	$f(n) = O(n^2)$	Stringa palindroma
Polinomiale	$f(n) = O(n^k)$	Qualsiasi funzione polinomiale
Esponenziale	$f(n)$ non polinomiale ma super polinomiale	Alcune funzioni super polinomiali sono $e^n \mid n! \mid n^{\log n}$

L'ultima classe rappresenta il *"troppo"* che abbiamo definito prima: infatti, nel *"troppo"* ci va tutto il calderone delle funzioni super polinomiali. Un algoritmo in questa classe si definisce **inefficiente**.

Altrimenti, convenzionalmente, un algoritmo si dice **efficiente** se la sua complessità temporale è **polinomiale**.

3.4.2. 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 risorse computazionali.

versione quantitativa della tesi di Church-Turing.

3.5. Definizione della risorsa tempo

3.5.1. Introduzione

Come mai utilizziamo una DTM e non una macchina RAM per dare una definizione rigorosa di tempo? La risposta sta nella **semplicità**: le macchine RAM, per quanto semplici, lavorano con banchi di memoria che possono contenere dati di grandezza arbitraria ai quali accediamo con tempo $O(1)$, cosa che invece non possiamo fare con le DTM perché il nastro contiene l'input diviso su più celle.

3.5.2. 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

$$t : \mathbb{N} \longrightarrow \mathbb{N} \mid t(n) = \max\{T(x) \mid x \in \Sigma^* \wedge |x| = n\}.$$

L'attributo **worst case** indica il fatto che $t(n)$ rappresenta il tempo *peggiore* di calcolo su tutti gli input di lunghezza n. È la metrica più utilizzata anche perché è la più "manovrabile matematicamente", cioè ci permette di utilizzare delle funzioni più facilmente trattabili dal punto di vista algebrico. Ad esempio, nella situazione **average case** avremo una stima probabilmente migliore ma ci servirebbe anche una distribuzione di probabilità, che non è molto facile da ottenere.

3.5.3. Esempio: parità

Facendo riferimento allo pseudocodice scritto precedentemente per il problema *parità*, facciamo un'analisi temporale dell'algoritmo.

Dato l'input x di lunghezza n, il nostro algoritmo non fa altro che consumare tutti i simboli dell'input fino a quando a non arriva al primo blank dopo l'input, quindi $t(n) = n + 1$.

In realtà questo che abbiamo appena mostrato è il *caso peggiore*. Ci sono delle istanze n_z per cui il problema impiega $t(n_z) = 2$. Infatti, se ci sono almeno due zeri in testa al numero, ovvero siamo di fronte a istanze nella forma

$$n_z = O^n \mid n > 1$$

allora abbiamo $t(n_z) = 2$ poiché la DTM si arresta subito.

3.5.4. Linguaggio riconosciuto in tempo deterministico

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. $t(n) \leq f(n)$.

L'ultima condizione indica che a noi *"basta"* $f(n)$ ma che possiamo accettare situazioni migliori.

Possiamo 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

$$L_A = \{\text{cod}(A) \mid a \in A\};$$

- il **problema di decisione** Π è risolto in tempo $f(n)$ se e solo lo è il linguaggio

$$L_\Pi = \{\text{cod}(x) \mid p(x)\}.$$

Da qui in avanti, quando parleremo di **linguaggi** intenderemo indirettamente **insiemi** o **problemi di decisione**, vista la stretta analogia tra questi concetti.

3.5.5. Esempio: parità

Tornando all'esempio del problema *parità*, abbiamo mostrato una DTM che riconosce il linguaggio

$$L_{\text{PARI}} = \{1\{0,1\}^*0\} \cup \{0\}$$

con complessità in tempo $t(n) = n + 1$. Ma allora abbiamo che:

- il linguaggio L_{PARI} è riconoscibile in tempo $n + 1$;
- l'insieme dei numeri pari è riconoscibile in tempo $n + 1$;
- il problema di decisione *parità* è risolubile in tempo $n + 1$.

3.5.6. Esempio: palindrome

- Nome: palindrome.
- Istanza: $x \in \Sigma^*$.
- Domanda: x palindroma?

In questo problema, data la stringa

$$x = x_1x_2...x_n \in \Sigma^*$$

definiamo la stringa

$$x^R = x_n...x_2x_1 \in \Sigma^*$$

tale che

$$L_{\text{PAL}} = \{x \in \Sigma^* \mid x = x^R\}.$$

Una DTM M per *palindrome* e che riconosce L_{PAL} è la seguente:

```
PALINDROME(x):
1   i := 1;
2   j := n;
3   f := true;
4   while (i < j && f) {
5       if (x[i] != x[j])
6           f := false;
7       i++;
8       j−;
9   }
10  return f;
```

Ovviamente vale che:

- se $x \in L_{\text{PAL}}$ allora viene restituito True ;
- se $x \notin L_{\text{PAL}}$ allora viene restituito False.

Notiamo una cosa importante: il worst case **NON** è

$$t(n) = \frac{n}{2}.$$

Infatti, in una macchina di Turing se voglio confrontare la posizione i-esima con quella j-esima devo spostare la testina $j - i$ volte, dalla posizione i alla posizione j. Il tempo reale è

$$t(n) = \sum_{k=0}^{n-1} k \approx n^2.$$

Quando valutiamo il tempo dobbiamo stare bene attenti di considerare anche i tempi di spostamento della DTM. Bisogna sempre tenere a mente l'architettura su cui implementiamo un certo algoritmo.

3.6. Classi di complessità

Proviamo a definire alcune classi di complessità in funzione della risorsa tempo.

La prima che introduciamo è la classe

$$DTIME(f(n)),$$

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 abbiamo visto nella scorsa lezione che abbiamo una analogia tra essi.

Nella scorsa lezione abbiamo inoltre mostrato che le DTM possono anche calcolare funzioni, quindi possiamo propagare questa definizione di DTIME anche alle funzioni stesse, ovvero possiamo definire delle classi di complessità anche per le funzioni.

Ma cosa intendiamo con "complessità in tempo per una funzione"?

La funzione $f : \Sigma^* \longrightarrow \Gamma^*$ 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, introduciamo la classe

$$FTIME(f(n))$$

definita come l'insieme delle funzioni risolte da una DTM in tempo deterministico $t(n) = O(f(n))$.

Grazie a quanto detto finora, possiamo definire due classi di complessità storicamente importanti:

$$P = \bigcup_{k \geq 0} DTIME(n^k)$$

classe dei problemi (*linguaggi*) **risolti da una DTM in tempo polinomiale** e

$$FP = \bigcup_{k \geq 0} FTIME(n^k)$$

classe delle funzioni calcolate da una DTM in tempo polinomiale.

Questi sono universalmente riconosciuti come i problemi efficientemente risolubili in tempo.

*Ma perché **polinomiale** è sinonimo di efficiente in tempo?*

Possiamo dare tre motivazioni:

- **pratica**: facendo qualche banale esempio, è possibile vedere come la differenza tra un algoritmo polinomiale e uno esponenziale, per input tutto sommato piccoli, è abissale: il tempo di risoluzione di un problema, nel primo caso, è di frazioni di secondo, mentre nel secondo arriva facilmente ad anni o secoli;
- **"composizionale"**: programmi efficienti che richiamano routine efficienti rimangono efficienti, questo perché concatenare algoritmi efficienti non fa altro che generare un tempo pari alla somma

delle complessità dei due algoritmi, che rimane polinomiale in quanto efficienti e polinomiali a loro volta;
- **"robustezza"**: le classi P e FP rimangono invariate a prescindere dai molti modelli di calcolo utilizzati per circoscrivere i problemi efficientemente risolti.

Per l'ultimo motivo, infatti, si può dimostrare che P e FP non dipendono dal modello scelto, che sia RAM, WHILE, DTM, eccetera.

3.6.1. Tesi di Church-Turing estesa

Tesi di Church-Turing estesa: la classe dei problemi **efficientemente risolubili in tempo** coincide con la classe dei problemi risolti in *tempo polinomiale* su DTM.

3.6.2. Chiusura di P

Teorema La classe P è un'algebra di Boole, ovvero è chiusa rispetto alle operazioni di unione, intersezione e complemento.

Dimostrazione

UNIONE

Date due istanze $A, B \in P$ e siano M_A, M_B due DTM con tempi rispettivamente $p(n)$ e $q(n)$, allora il seguente programma (ad alto livello)

$$\begin{aligned} P \equiv\ & \text{input}(n) \\ & y := M_A(x); \\ & z := M_B(x); \\ & \text{output}(y \vee z) \end{aligned}$$

permette il calcolo dell'unione di A e B in tempo $t(n) = p(n) + q(n)$.

INTERSEZIONE

Date due istanze $A, B \in P$ e siano M_A, M_B due DTM con tempi rispettivamente $p(n)$ e $q(n)$, allora il seguente programma (ad alto livello)

$$\begin{aligned} P \equiv\ & \text{input}(n) \\ & y := M_A(x); \\ & z := M_B(x); \\ & \text{output}(y \wedge z) \end{aligned}$$

permette il calcolo dell'intersezione di A e B in tempo $t(n) = p(n) + q(n)$.

COMPLEMENTO

Data l'istanza $A \in P$ e sia M_A una DTM con tempo $p(n)$, allora il seguente programma (ad alto livello)

$$P \equiv \text{input}(n)$$
$$y := M_A(x);$$
$$\text{output}(\neg y)$$

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**: infatti, posso comporre tra loro le macchine di Turing come se fossero procedure black box. Facendo l'esempio con due DTM, otteniamo

$$x \rightsquigarrow M_1 \rightsquigarrow x' \rightsquigarrow M_2 \rightsquigarrow y.$$

Supponiamo che le macchine M_1, M_2 abbiano tempo rispettivamente $p(n)$ e $q(n)$, allora il tempo totale è

$$t(n) \leq p(n) + q(p(n)).$$

Usiamo $q(p(n))$ perché eseguendo M_1 in $p(n)$ passi il massimo output che scrivo è grande $p(n)$.

3.6.3. Esempio

Facciamo vedere quanto influenzano l'esponente della funzione polinomiale e le costanti dei simboli di Landau. Consideriamo una macchina a 4GHz che esegue un'operazione ogni

$$\frac{1}{4 \cdot 10^9}$$

secondi. Fissiamo anche l'input di grandezza $n = 4000$ caratteri.

La seguente tabella mostra i tempi approssimati di alcune funzioni su input di grandezza n.

Funzione $t(n)$	**Tempo di esecuzione**
n	$\approx \mu s$
n^2	$\approx ms$
n^3	$\approx s/a$
2^n	1 gogol, ovvero 10^{100} secondi, più dell'età dell'universo

Infine, parliamo di costanti nei simboli di Landau: con costanti molto grandi riesco a inserire algoritmi inefficienti negli efficienti e viceversa. Ad esempio, l'algoritmo del *quicksort* è più lento nel *worst case* dell'algoritmo del *mergesort*, però la costante del mergesort è più grande di quella del quicksort.

3.6.4. 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.

3.7. Spazio di memoria

Veniamo ora alla formalizzazione dell'altra importante risorsa di calcolo, ovvero lo **spazio di memoria**, inteso come quantità di memoria occupata durante la computazione.

3.7.1. Complessità in spazio (1)

Data la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ed una stringa $x \in \Sigma^*$, chiamiamo $S(x)$ il numero di celle del nastro occupate (visitate, *sporcate*) durante la computazione di M su x. Questo numero potrebbe anche essere *infinito*.

La **complessità in spazio** di M (*worst case*) è la funzione $s : \mathbb{N} \longrightarrow \mathbb{N}$ definita come

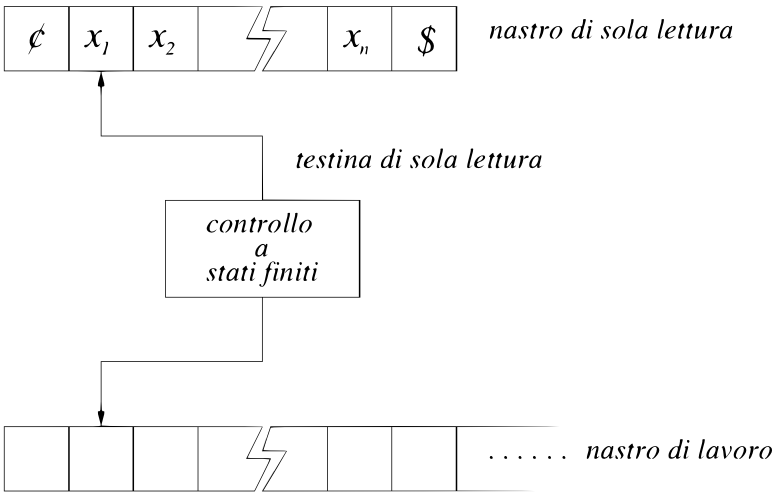
$$s(n) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}.$$

Da questa definizione è chiaro che, in ogni MDT, $s(n) \geq n$ in quanto dovrò sempre occupare almeno spazio n lineare per mantenere l'input sul nastro, ma è molto probabile che le celle effettive che *sporco* sono molte meno delle celle occupate dall'input.

Come non considerare l'interferenza dovuta all'input?

Per avere complessità anche sublineari, potremmo modificare leggermente la macchina e separare le operazioni del nastro, ovvero utilizzare due nastri diversi per la lettura e per la computazione:

- il **nastro di lettura** è read-only con testina two-way read-only;
- il **nastro di lavoro** è invece read-write con testina read-write.



La stringa in input è delimitata dai caratteri ¢ e \$ tali che ¢, \$ $\notin \Sigma$.

La definizione formale di questa nuova macchina è

$$M = (Q, \Sigma \cup \{¢, \$\}, \Gamma, \delta, q_0, F),$$

in cui tutto è analogo alle macchine di Turing viste finora, tranne per la funzione di transizione δ, ora definita come

$$\delta : Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma \longrightarrow Q \times (\Gamma \setminus \{\text{blank}\}) \times \{-1, 0, 1\}^2$$

con la quale M:

1. legge un simbolo sia dal nastro di input sia 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, infatti, 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 salvare anche l'input perché, essendo il nastro read-only, non cambia mai.

Gli altri concetti (*computazione, accettazione, linguaggio accettato, complessità in tempo, eccetera*) rimangono inalterati con questo modello.

3.7.2. Complessità in spazio (2)

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.

3.7.3. Complessità in spazio di linguaggi

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. $s(n) \leq f(n)$.

Sfruttando questa definizione, possiamo ovviamente definire la complessità in spazio per il riconoscimento di insiemi e per la funzione soluzione di problemi di decisione.

3.7.3.1. Calcolo di funzioni

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 l'output e di non interferire con il nastro di lavoro.

Diremo che una funzione $f : \Sigma^* \longrightarrow \Gamma^*$ è calcolabile dalla DTM M se e solo se:

- $f(x) \downarrow$ ponendo x sul nastro di input e ottenendo $f(x)$ sul nastro di output;
- $f(x) \uparrow$ ponendo x sul nastro di input e ottenendo un loop.

Inoltre, diremo che la funzione $f : \Sigma^* \longrightarrow \Gamma^*$ 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.

3.7.4. Classi di complessità

Per ogni funzione $f : \mathbb{N} \longrightarrow \mathbb{N}$ definiamo

$$DSPACE(f(n))$$

la **classe dei linguaggi accettati da DTM in spazio deterministico** $O(f(n))$.

Chiamiamo invece

$$FSPACE(f(n))$$

classe delle funzioni calcolate da DTM in spazio deterministico $O(f(n))$.

Notiamo che le classi $DSPACE$ e $FSPACE$ **non** cambiano se aggiungiamo alle nostre 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.

3.7.5. Esempio: Parità

Riprendiamo la DTM M per il linguaggio $L_{\text{PARI}} = 1\{0,1\}^*0 \cup 0$ definita dal seguente programma.

```
Parità(x):
 1  i := 1;
 2  f := false;
 3  switch(x[i]) {
 4        case 0 :
 5              i++;
 6              f := (x[i] == blank);
 7              break;
 8        case 1:
 9              do  {
10                    i++ ;
11                    f := (x[i] == 0);
12              }  while (x[i] ≠ blank);
13  }
14  return f;
```

Abbiamo già visto che $L_{\text{PARI}} \in DTIME(n)$. *E lo spazio?*

A dire il vero, non viene occupato spazio, in quanto tutto può essere fatto usando solamente gli stati. Infatti, L_{PARI} è un linguaggio **regolare** e si può dimostrare che

$$\text{REG} = DSPACE(1).$$

In poche parole, stiamo buttando via il nastro di lavoro, trasformando la DTM in un **automa a stati finiti**. Per *buttarlo via* devo però aumentare il numero di stati: infatti, le informazioni che *buttiamo* dal nastro di lavoro vanno codificate in stati, che quindi aumentano di numero.

3.7.6. Esempio: Palindrome

Consideriamo $L_{\text{PAL}} = \{x \in \Sigma^* \mid x = x^R\}$ il linguaggio considerato la lezione precedente. Avevamo visto che una DTM M per questo linguaggio è definita dal seguente programma.

```
PALINDROME(x):
 1  i := 1;
 2  j := n;
 3  f := true;
 4  while(i < j && f) {
 5        if (x[i] != x[j])
 6              f := false;
 7        i++;
 8        j−;
 9  }
10  return f;
```

Sappiamo già che, per quanto riguarda il tempo, $t(n) = O(n^2)$.

Per quanto riguarda lo spazio, invece, dobbiamo cercare di capire che valori possono assumere i numeri che scriviamo durante la computazione. Tra i e j, il numero più grande è sempre j, che al massimo vale n, di conseguenza

$$s(n) = O(n).$$

Possiamo fare meglio: questa rappresentazione scelta è **unaria**, perché *sporco* un numero di celle esattamente uguale a n. Usando una rappresentazione **binaria**, il numero di celle di cui abbiamo bisogno è il numero di bit necessari a rappresentare n, ovvero

$$s(n) = O(\log(n)).$$

Possiamo usare una base del logaritmo più alta per abbassare la complessità?

No, perché qualsiasi logaritmo può essere trasformato in un altro a meno di una costante moltiplicativa, quindi è indifferente la base utilizzata.

Ad esempio,

$$O(\log_2(n)) = O\left(\frac{\log_{100}(n)}{\log_{100}(2)}\right) = O(\log_{100}(n)).$$

Quindi,

$$L_{\text{PAL}} \in DTIME(n^2)$$

e

$$L_{\text{PAL}} \in DSPACE(\log(n)).$$

Possiamo essere più veloci?

Un algoritmo più "veloce" di quello che abbiamo visto, è il seguente.

Istruzione	Tempo	Spazio
Copia la stringa di input sul nastro di lavoro	n	n

Sposta la testina del nastro di input in prima posizione (quella del nastro di lavoro sarà alla fine)	n	-
Confronta i due caratteri, avanzando al testina di input e retrocedendo quella di lavoro	n	-
Accetta se tutti i confronti tornano, altrimenti rifiuta	$t(n) = O(n)$	$s(n) = O(n)$

Questo ci mostra come abbiamo avuto un gran miglioramento per la risorsa tempo, a discapito della risorsa spazio. Spesso (*ma non sempre*) migliorare una risorsa porta al peggioramento di un'altra.

Esistono quindi diversi algoritmi per un dato problema che ottimizzano solo una delle due risorse a disposizione. Per il linguaggio L_{PAL} si dimostra che

$$t(n) \cdot s(n) = \Omega(n^2).$$

3.7.7. Efficienza in termini di spazio

Definiamo:

- $L = DSPACE(\log(n)) \rightarrow$ classe dei linguaggi accettati in spazio deterministico $O(\log(n))$;
- $FL = FSPACE(\log(n)) \rightarrow$ 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 nella lezione scorsa per la risorsa 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 (che potrebbe anche non riuscire a contenerli tutti) usando algoritmi sofisticati che si servono, ad esempio, di tecniche per fissare posizioni sull'input o contare parti dell'input (in generale procedure che siano logaritmiche in spazio).
  I dati diventano facilmente grandi e bisogna avere algoritmi che utilizzino poca memoria.
- **composizionale**: i programmi efficienti in spazio che richiamano routine efficienti in spazio, rimangono efficienti in spazio;
- **robustezza**: le classi L e FL rimangono invariate, a prescindere dai modelli di calcolo utilizzati, ad esempio DTM multi-nastro, RAM, WHILE, etc.

Tesi di Church-Turing estesa per lo spazio : La classe dei problemi efficientemente risolubili in spazio coincide con la classe dei problemi risolti in spazio logaritmico su DTM.

Vediamo alcuni esempi di problemi risolti efficientemente in spazio:

- in L:
  - testare la raggiungibilità tra due nodi di un grafo non diretto;
  - parsing dei linguaggi regolari;
  - ci si chiede se anche il parsing dei linguaggi context-free è efficiente. Attualmente $s(n) = O(\log^2 n)$;
- in FL:
  - operazioni aritmetiche;
  - permanenti di matrici booleane

- aritmetica modulare;

Se un problema è in L o FL è anche efficientemente **parallelizzabile**. Al momento non esistono compilatori perfettamente parallelizzabili.

3.8. Tempo vs spazio

Spesso promuovere l'ottimizzazione di una risorsa va a discapito dell'altra: *essere veloci* vuol dire (tipicamente) *spendere tanto spazio* e *occupare poco spazio* vuol dire (tipicamente) *spendere tanto tempo.*

Viene naturale porsi due domande:

- *i limiti in tempo implicano dei limiti in spazio?*
- *i limiti in spazio implicano dei limiti in tempo?*

Per rispondere confrontiamo le classi $DTIME(f(n))$ e $DSPACE(f(n))$.

Teorema Tutti i linguaggi accettati in tempo $f(n)$, sono anche accettati in spazio $f(n)$. Formalmente:

$$DTIME(f(n)) \subseteq DSPACE(f(n)).$$

Dimostrazione
Se $L \in DTIME(f(n))$ 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, *quante celle del nastro di lavoro posso occupare al massimo?*
Ovviamente $O(f(n))$ (una cella ad ogni passo). Quindi, M ha complessità in spazio $s(n) = O(f(n))$, ma allora $L \in DSPACE(f(n))$. □

Teorema Tutte le funzioni accettate in tempo $f(n)$, sono anche accettate in spazio $f(n)$. Formalmente:

$$FTIME(f(n)) \subseteq FSPACE(f(n)).$$

Notiamo come l'efficienza in tempo non porta immediatamente all'efficienza in spazio.

3.9. Relazione tempo-spazio

Abbiamo visto che un limite in tempo implica, in qualche modo, un limite in spazio, *ma vale anche il contrario? È possibile dimostrare che* $DSPACE(f(n)) \subseteq DTIME(f(n))$*?*

Avendo un numero di celle prestabilito, è possibile iterare il loro utilizzo (anche all'infinito, entrando ad esempio in un loop), di conseguenza limitare lo spazio non implica necessariamente una limitazione del tempo.

Notiamo che, in una DTM M, un loop si verifica quando visitiamo una configurazione già visitata in passato. Sfruttando questo fatto, è possibile trovare una limitazione al tempo, trovando dopo quanto tempo vengono visitate tutte le configurazioni possibili.

Teorema Tutti i linguaggi accettati in spazio $f(n)$ vengono accettati in tempo $n \cdot \alpha^{O(f(n))}$.

$$DSPACE(f(n)) \subseteq DTIME\left(n \cdot \alpha^{O(f(n))}\right)$$

Dimostrazione
Dato $L \in DSPACE(f(n))$ e una DTM M esistono una serie di configurazioni per M tali che

$$C_0 \xrightarrow{\delta} C_1 \xrightarrow{\delta} \cdots \xrightarrow{\delta} C_m,$$

in cui C_m è uno stato accettante per L.

Sappiamo che $DTIME$ è calcolabile dal numero di volte che viene utilizzata la funzione 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.

Ricordiamo che una 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 delimitatori, il numero massimo è $n + 2$;
- j: stiamo lavorando in $DSPACE(f(n))$, quindi questo indice vale $O(f(n))$, che possiamo scrivere più semplicemente come $\alpha f(n)$;
- w: è una stringa sull'alfabeto $\Gamma^{O(f(n))}$, che ha cardinalità $|\Gamma|^{O(f(n))}$, scrivibile anche in questo caso come $|\Gamma|^{\alpha f(n)}$.

Moltiplicando tutti questi valori, troviamo il seguente upper bound:

$$\begin{aligned}
|C| &\leq O(1) \cdot (n+2) \cdot \alpha f(n) \cdot |\Gamma|^{\alpha f(n)} \\
&\leq O(1) \cdot (n+2) \cdot |\Gamma|^{\alpha f(n)} \cdot |\Gamma|^{\alpha f(n)} \\
&= O(1) \cdot (n+2) \cdot |\Gamma|^{2\alpha f(n)} \\
&= O(1) \cdot (n+2) \cdot 2^{\log_2(|\Gamma|^{2\alpha f(n)})} \\
&= O(1) \cdot (n+2) \cdot 2^{2\alpha f(n) \cdot \log_2(|\Gamma|)} \\
&= O\big(n \cdot 2^{O(f(n))}\big).
\end{aligned}$$

Quindi, M sa accettare o meno $x \in \Sigma^*$ in al massimo $O\big(n \cdot 2^{O(f(n))}\big)$ passi.

Ora, data una DTM M che accetta L con $s(n) \leq \alpha f(n)$, 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 $t \sim O\big(n \cdot 2^{O(f(n))}\big)$;
2. simula M e ad ogni mossa cancella un simbolo di time-out del 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

$$t(n) = O\big(n \cdot 2^{O(f(n))}\big),$$

e quindi

$$DSPACE(f(n)) \subseteq DTIME\left(n \cdot 2^{O(f(n))}\right).$$

□

Come per il tempo, il teorema dimostrato vale anche per gli insiemi $FSPACE$ e $FTIME$.

Teorema Tutti le funzioni calcolate in spazio $f(n)$ vengono calcolate in tempo $n \cdot \alpha^{O(f(n))}$.

$$FSPACE(f(n)) \subseteq FTIME\left(n \cdot \alpha^{O(f(n))}\right).$$

3.9.1. Relazione delle classi L e P

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 Valgono le seguenti relazioni per efficienza in spazio e efficienza in tempo:

$$L \subseteq P$$
$$FL \subseteq FP.$$

Dimostrazione

$$\begin{aligned} L = DSPACE(\log(n)) &\subseteq DTIME\left(n \cdot \alpha^{O(\log(n))}\right) = \\ &= DTIME\left(n \cdot \alpha^{\frac{\log_\alpha(n)}{\log_\alpha(2)}}\right) = \\ &= DTIME\left(n \cdot \left(\alpha^{\log_\alpha(n)}\right)^{\frac{1}{\log_\alpha(2)}}\right) = \\ &= DTIME\left(n \cdot n^{\frac{1}{\log_\alpha(2)}}\right) = \\ &= DTIME\left(n \cdot n^{\beta}\right) = DTIME\left(n^{\beta+1}\right) = DTIME\left(n^{k}\right) = P \end{aligned}$$

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 il L?"* ancora oggi non ha una risposta, è 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.

3.10. Classe EXPTIME

Definiamo ora la classe

$$EXPTIME = \bigcup_{k \geq 0} DTIME\left(2^{n^k}\right)$$

dei problemi con complessità temporale **esponenziale**. Ovviamente vale

$$P \subseteq EXPTIME,$$

perché ogni polinomio è "*maggiorabile*" da un esponenziale. Per diagonalizzazione si è dimostrato in realtà che

$$P \subset EXPTIME$$

sfruttando una **NDTM** (*Non-Deterministic Turing Machine*) con timeout.

3.11. 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 nessuno ha mai dimostrato che tali algoritmi non possano esistere. Infatti, dato un problema Π, se mi viene detto che ad oggi non esiste un algoritmo efficiente per la sua soluzione, questo non implica che allora lo sia veramente: è molto difficile come dimostrazione.

I problemi di decisione in questa zona hanno una particolarità: sono **efficientemente verificabili**. Data un'istanza particolare, è facile capire se per quel problema e quell'istanza bisogna rispondere SI o NO.

3.11.1. *CNF-SAT*

Il problema ***CNF-SAT*** ha come obiettivo quello di stabilire se esiste un assegnamento a variabili booleane che soddisfi un predicato logico in forma normale congiunta. Le formule sono indicate con $\varphi(x_1, ..., x_n)$ e sono formate da congiunzioni $C_1 \wedge ... \wedge C_k$, ognuna delle quali contiene almeno una variabile booleana x_i.

Formalmente, data una *CNF* $\varphi(x_1, ..., x_n)$, vogliamo rispondere alla domanda

$$\exists \underline{x} \in \{0,1\}^n \mid \varphi(\underline{x}) = 1?$$

Un possibile algoritmo di risoluzione è quello esaustivo:

$$\begin{aligned} P \equiv &\text{for } \underline{x} \in \{0,1\}^n \text{ do} \\ &\quad \text{if } \varphi(\underline{x}) = 1 \text{ then} \\ &\quad\quad \text{return } 1 \\ &\text{return } 0. \end{aligned}$$

Notiamo come 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, in quanto esplorare tutto l'albero dei possibili assegnamenti (*sottoproblemi*) richiederebbe tempo esponenziale.

Attenzione al viceversa. Se ho infinite configurazioni da testare non è vero che il problema usi algoritmi inefficienti: esistono problemi su grafi (*raggiungibilità*) che potrebbero testare infinite configurazioni ma che in realtà sono efficientemente risolti con altre tecniche.

3.11.2. Circuiti hamiltoniani

Dato $G = (V, E)$ grafo non diretto, vogliamo sapere se G contiene un circuito hamiltoniano o meno.

Ricordiamo un paio di concetti sui grafi:

- **cammino**: sequenza di vertici $V_1, ..., V_k$ tali che $\forall 1 \leq i < k \quad (V_i, V_{i+1}) \in E$;
- **circuito**: cammino $V_1, ..., V_k$ tale che $V_1 = V_k$, quindi un cammino che parte e termina in uno stesso vertice;
- **circuito hamiltoniano**: circuito in cui tutti i vertici di G vengono visitati una e una sola volta.

Un algoritmo per questo problema è il seguente:

$$\begin{aligned} P \equiv &\text{for } \left(V_{i_1}, ..., V_{i_n}, V_{i_1}\right) \in \text{Perm}(V) \text{ do} \\ &\quad \text{if IS\_HC}\left(V_{i_1}, ..., V_{i_n}, V_{i_1}\right) = 1 \text{ then} \\ &\quad\quad \text{return } 1 \\ &\text{return } 0, \end{aligned}$$

in cui sostanzialmente generiamo tutte le permutazioni possibili di vertici che iniziano e finiscono con lo stesso vertice e verifichiamo efficientemente se esse sono un circuito hamiltoniano o meno.

Calcoliamo la complessità temporale di questo algoritmo:

- il numero di permutazioni, e quindi il numero di volte che viene eseguito il ciclo, sono $n!$;
- il controllo sulla permutazione può essere implementato efficientemente in tempo polinomiale.

3.11.3. Circuiti euleriani

Dato $G = (V, E)$ un grafo non diretto, vogliamo sapere se G contiene un circuito euleriano o meno.

Ricordiamo che un circuito euleriano è un circuito in cui tutti gli archi di G vengono visitati una e una sola volta. Potrebbe sembrare simile al problema precedente, ma non lo è!

Teorema *(Eulero 1736)* Un grafo G contiene un circuito euleriano se e solo se ogni suo vertice ha grado pari, ovvero

$$\forall v \in V \quad \text{GRADO}(v) = 2k \mid k \in \mathbb{N}.$$

Grazie a questo teorema è possibile risolvere il problema in tempo lineare, quindi efficiente. Purtroppo non esiste un teorema simile per i circuiti hamiltoniani.

3.12. 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.

3.12.1. 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**, in cui viene generata "magicamente" una struttura/un assegnamento/una configurazione/una congettura che aiuta a dare una risposta "sì"/"no";
- **fase di verifica**, in cui usiamo la struttura prodotta precedentemente 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 ad essere non deterministica e in particolare lo è nella creazione della struttura "magica".

3.12.2. Soluzione algoritmi di decisione

Dato un problema Π, un'istanza $x \in D$ e una proprietà $p(x)$, un algoritmo non deterministico risolve Π sse:

1. su ogni x a risposta positiva (quindi $\forall x : p(x) = 1$), esiste **almeno una computazione** k che accetta la coppia (x, s_k).
2. su ogni x a 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 o rifiutano o vanno in loop.

3.12.2.1. *CNF-SAT*

Vediamo un algoritmo non deterministico per la soluzione di *CNF-SAT*:

$$
\begin{aligned}
P \equiv\ & \text{input}(\varphi(x_1, ..., x_n)); \\
& \text{genera ass. } x \in \{0,1\}^n; \\
& \text{if } (\varphi(x_1, ..., x_n) == 1) \\
& \quad \text{return } 1; \\
& \text{return } 0;
\end{aligned}
$$

Ammettendo un modello di calcolo come quello descritto, questo è a tutti gli effetti un algoritmo non deterministico, formato da fase congetturale e fase di verifica.

3.12.2.2. Circuito hamiltoniano

Vediamo ora un algoritmo non deterministico per trovare, se esiste, un circuito hamiltoniano in un grafo G.

$$
\begin{aligned}
P \equiv\ & \text{input}(G = (V, E)); \\
& \text{genera perm. } \pi(v_1, ..., v_n); \\
& \text{if } (\pi(v_1, ..., v_n) \text{ è un circuito in } G) \\
& \quad \text{return } 1; \\
& \text{return } 0;
\end{aligned}
$$

Si vede chiaramente come sia simile a quello precedente, mostrando che la struttura di questi algoritmi e pressoché la stessa.

3.12.3. Tempo di calcolo

Dato che è cambiato il modello di calcolo, dobbiamo rivedere la definizione di tempo di calcolo.

Consideriamo il tempo di calcolo $T(x)$, per un'istanza x a risposta positiva (negativa), come il miglior tempo di calcolo delle fasi di verifica a risposta positiva (negativa). Per convenzione, la fase congetturale non impiega tempo.
Da ricordare che questo è un modello teorico, infatti nella realtà pagherò tempo anche per la generazione delle strutture che servono nelle verifiche.

Come formalizzare questo tipo di algoritmi?

3.13. Macchina di Turing Non Deterministica

Consideriamo una DTM M come l'abbiamo già descritta e apportiamo delle modifiche:

- allunghiamo il nastro, in modo che sia 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) e $x \in \Sigma^*$:

- viene accettato $\iff \exists \gamma \in \Gamma^* : (\gamma, x)$ viene deterministicamente accettata;
- non viene accettato altrimenti.

Il linguaggio accettato da M è

$$L_M = \{x \in \Sigma^* : M \text{ accetta } x\}.$$

Definizione: Un linguaggio $L \subseteq \Sigma^*$ è accettato da un algoritmo non deterministico sse 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

$$L_\Pi = \{\text{cod}(x) : x \in D \wedge p(x)\}$$

dove cod : $D \to \Sigma^*$ e la funzione di codifica delle istanze del problema.

Definizione: 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.

3.13.1. Complessità in tempo

Come abbiamo accennato precedentemente, la complessità in tempo di un algoritmo non deterministico corrisponde alla miglior complessità in tempo per quell'istanza. Allo stesso modo, possiamo definire la complessità in tempo per le NDTM.

Definizione: Una NDTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ha **complessità in tempo** $t : \mathbb{N} \to \mathbb{N}$ sse per ogni input $x \in L_M$ con $|x| = n$ esiste almeno una computazione accettante di M che impiega $t(n)$ passi.

Ne consegue anche la prossima definizione.

Definizione: Un linguaggio è accettato con **complessità in tempo non deterministico** $t(n)$ sse esiste una NDTM M con complessità in tempo $t(n)$ che lo accetta.

In questo modo abbiamo mappato tutti i concetti chiave visti nelle macchine di Turing deterministiche anche per il non determinismo.

3.14. Classi di complessità non deterministiche

È possibile definire delle classi per i linguaggi accettati allo stesso modo di come avevamo fatto per le DTM. Chiamiamo

$$NTIME(f(n))$$

come l'insieme dei linguaggi accettati con complessità non deterministica $O(f(n))$.

Caratterizziamo il concetto di "efficienza" anche per il non determinismo.
Efficiente risolubilità:

$$P = \bigcup_{k \geq 0} DTIME(n^k).$$

Efficiente verificabilità:

$$\text{NP} = \bigcup_{k \geq 0} NTIME(n^k),$$

che corrisponde all'insieme dei problemi di decisione che ammettono algoritmi non deterministici polinomiali.

4. Lezione 23

The Millennium Prize Problem: *Che relazione c'è tra le classi P e NP?* $\longrightarrow$ È una questione ancora aperta, la quale farebbe guadagnare un milione di dollari a colui che troverà una risposta.

4.1.1. $P \subseteq NP$

Possiamo solo dimostrare una banale relazione.

Teorema $P \subseteq NP$

Dimostrazione È facile dimostrare che $DTIME(f(n)) \subseteq NTIME(f(n))$.

Dato $L \in DTIME(f(n))$, esiste una DTM M che lo riconosce in $t(n) = O(f(n))$. $(*)$
Chiaramente M può essere vista come una NDTM che ignora il modulo congetturale. La NDTM così ottenuta ripropone la stessa computazione di M su ogni congettura generata inutilmente. È chiaro che questa NDTM accetta L in tempo $t(n) = O(f(n)) \Rightarrow L \in NTIME(f(n))$.
Quindi vale:

$$P = \bigcup_{k \geq 0} DTIME\left(n^k\right) \overset{(*)}{\subseteq} \bigcup_{k \geq 0} NTIME\left(n^k\right) = NP.$$

□

4.1.2. $NP \subseteq P$

Cosa possiamo dire sulla relazione inversa?
È chiaro che il punto cruciale del Millennium Problem è proprio la relazione $NP \subseteq P$, che in realtà si può tradurre in $P = NP$ vista la relazione appena dimostrata.

Detto in altre parole, ci chiediamo se da un algoritmo non deterministico efficiente è possibile ottenere un algoritmo *reale* efficiente.

Similmente a quanto fatto nella dimostrazione di $P \subseteq NP$, proviamo ad analizzare questo problema

$$NTIME(f(n)) \subseteq DTIME(?),$$

che quantifica quanto costa togliere il fattore di non determinismo (che è un concetto non naturale) della fase congetturale.

Supponiamo di avere $L \in NTIME(f(n))$. 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 qualcosa di questo tipo:

1. prendiamo in input $x \in \Sigma^*$, con $|x| = n$;
2. genera tutte le strutture $\gamma \in \Gamma^*$ delle fasi di verifica;
3. per ognuna di esse, calcola **deterministicamente** se (γ, x) viene accettata con M;
4. se in una delle computazioni al punto 3 la risposta è positiva, allora accetta x, altrimenti $\widetilde{M}$ rifiuta (la gestione dei loop può essere fatta tramite la tecnica del count-down).

Il problema qui è che di stringhe $\gamma \in \Gamma^*$ ne esistono infinite!

Ma ci servono proprio tutte?
Rivediamo lo pseudo-codice dell'algoritmo che stiamo progettando:

$$
\begin{aligned}
\text{DTM } \widetilde{M} \equiv\ & \text{input}(x) \\
& \text{for each } \gamma \in \Gamma^* : \\
& \quad \text{if}(M \text{ accetta } (\gamma, x) \text{ in } t(n) \text{ passi}) \\
& \quad\quad \text{return } 1; \\
& \text{return } 0;
\end{aligned}
$$

Per evitare di considerare tutte le stringhe γ, possiamo considerare solo quelle che non sono più lunghe di $t(n)$, perché altrimenti non potrei finire la computazione in $t(n)$ passi.

L'algoritmo diventa:

$$
\begin{aligned}
\text{DTM } \widetilde{M} \equiv\ & \text{input}(x) \\
& \text{for each } (\gamma \in \Gamma^* \wedge |\gamma| = O(f(n))) : \\
& \quad \text{if}(M \text{ accetta } (\gamma, x) \text{ in } t(n) \text{ passi}) \\
& \quad\quad \text{return } 1; \\
& \text{return } 0;
\end{aligned}
$$

Studiamo quanto tempo impiega

$$t(n) = |\Gamma|^{O(f(n))} \cdot O(f(n)) = O\big(f(n) \cdot 2^{O(f(n))}\big),$$

di conseguenza avremo che

$$NTIME(f(n)) \subseteq DTIME\big(f(n) \cdot 2^{O(f(n))}\big).$$

Come ci si poteva aspettare, togliere il non determinismo purtroppo costa parecchio, tanto da rendere l'algoritmo esponenziale e quindi inefficiente.

$$NP \subseteq DTIME\big(2^{n^{O(1)}}\big) = EXPTIME.$$

L'unica cosa che attualmente sappiamo dire è che tutti i problemi in NP hanno sicuramente algoritmi di soluzione con tempo esponenziale, ma ancora non possiamo escludere che $NP \subseteq P$.

Attenzione: NP non sta per "Non Polinomiale" e quindi esponenziale. NP sta per polinomiale in un architettura non deterministica. Dire che NP contiene problemi con algoritmi di soluzione esponenziale è FALSO. Tali problemi potrebbero anche avere soluzioni efficienti (anche se nessuno lo crede).

4.2. $NP \subseteq P$?

Nella scorsa lezione abbiamo introdotto i problemi della classe *NP*, detti anche *problemi con botta di culo* o *problemi con il fattore C*. Se per questi problemi non abbiamo ancora una soluzione efficiente vuol dire che non lo conosciamo ancora a fondo, o magari manca una solida base matematica che ci permetta di risolverli.

Per mostrare che $NP \subseteq P$ dovremmo prendere ogni problema in *NP* e trovare per essi un algoritmo di soluzione *polinomiale*. Ovviamente questo approccio è impraticabile dato che *NP* contiene infiniti problemi, del tutto eterogenei. Potremmo ovviare a questo problema agendo solo su un sottoinsieme dei problemi a nostra disposizione piuttosto che su tutti quanti.

Vediamo una strategia che permette di isolare un kernel di problemi in *NP*:

1. stabiliamo una **relazione di difficoltà** tra problemi in *NP*: dati $\pi_1, \pi_2 \in NP$ allora $\pi_1 \leq \pi_2$ indica che, se riesco a trovare una soluzione efficiente per π_2, allora ho automaticamente una

soluzione efficiente per π_1. In poche parole, π_1 non è più difficile di π_2, se sono bravo su π_2 sono bravo anche su π_1;

2. definita tale relazione, utilizziamola per trovare l'insieme dei problemi **più difficili** di *NP* : π è difficile in *NP* se

$$\pi \in NP \land \forall \widetilde{\pi} \in NP \quad \widetilde{\pi} \leq \pi.$$

   Questi possiamo definirli i *problemi bad boys* di *NP*;
3. restringiamo la ricerca di algoritmi efficienti al kernel in *NP* appena trovato: se possiamo risolvere questi in modo efficiente, allora possiamo trovare algoritmi efficienti per tutto *NP*.

4.2.1. Riduzione polinomiale

Iniziamo definendo formalmente una **relazione di difficoltà** necessaria a identificare un sottoinsieme di *NP*.

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ (o due problemi di decisione) diciamo che L_1 si **riduce polinomialmente** a L_2, e lo indichiamo con $L_1 \underset{P}{\leq} L_2$, se e solo se esiste una funzione $f : \Sigma^* \longrightarrow \Sigma^*$ tale che:

1. f è calcolabile su DTM in **tempo polinomiale**, quindi $f \in FP$;
2. $\forall x \in \Sigma^* \quad x \in L_1 \iff f(x) \in L_2$.

Le funzioni di riduzione polinomiale sono anche dette **many-one-reduction** perché non sono per forza funzioni iniettive (e quindi biiettive).

Teorema Siano due linguaggi $A, B \subseteq \Sigma^* \mid A \underset{P}{\leq} B$. Allora

$$B \in P \Longrightarrow A \in P.$$

Dimostrazione

Siccome $A \underset{P}{\leq} B$, sia $f \in FP$ la funzione di riduzione polinomiale. Sappiamo inoltre che $B \in P$. Consideriamo il seguente algoritmo:

$$\begin{aligned} P \equiv\ & \text{input}(x) \\ & y := f(x); \\ & \text{if } (y \in B) \\ & \quad \text{return } 1 \\ & \text{else} \\ & \quad \text{return } 0 \end{aligned}$$

Questo algoritmo è sicuramente deterministico, perché tutte le procedure che lo compongono sono deterministiche. Inoltre, riconosce A, per via della seconda condizione della riducibilità.

La sua complessità in tempo, dato un input x di lunghezza n, è:

$$t(n) = t_f(n) + t_{y \in B}(|y|) = p(n) + q(|y|).$$

Notiamo che $|y| \leq p(n)$, in quanto output di una procedura che impiega $p(n)$ passi per generare y, quindi:

$$t(n) \leq p(n) + q(p(n)) = \text{poly}(n),$$

in quanto i polinomi sono chiusi per somma e composizione.

In conclusione, l'algoritmo deterministico proposto riconosce A in tempo polinomiale, quindi:

$$A \underset{P}{\leq} B \wedge B \in P \Longrightarrow A \in P.$$

□

Questo teorema è ottimo per la nostra *"missione"*: infatti, se trovo una soluzione efficiente per i *"problemi difficili"* allora ce l'ho anche per i *"problemi meno difficili"*, visto che dovrei aggiungere solo la parte di riduzione polinomiale, che abbiamo mostrato essere efficiente.

4.2.2. Problemi *NP*-completi

Possiamo identificare ora il sottoinsieme di problemi che sono *alla destra* della relazione di riducibilità polinomiale per almeno un altro problema, i problemi *cattivi* e *difficili*.

Un problema di decisione Π è ***NP*-completo** se e solo se:

1. $\Pi \in NP$;
2. $\forall \widetilde{\Pi} \in NP \quad \widetilde{\Pi} \underset{P}{\leq} \Pi$.

Sia NPC la sottoclasse di *NP* dei problemi *NP*-completi. Per provare che $NP \subseteq P$ posso restringere la mia ricerca di algoritmi di soluzione efficiente ai soli membri di NPC grazie al seguente teorema.

Teorema Sia $\Pi \in NPC$ e $\Pi \in P$. Allora, $NP \subseteq P$, e quindi $P = NP$.

Dimostrazione
Dato che $\Pi \in NPC$, vale

$$\forall \widetilde{\Pi} \in NP \quad \widetilde{\Pi} \underset{P}{\leq} \Pi.$$

Visto che $\Pi \in P$, abbiamo dimostrato prima che

$$\widetilde{\Pi} \underset{P}{\leq} \Pi \wedge \Pi \in P \Longrightarrow \widetilde{\Pi} \in P,$$

otteniamo che ogni problema $\widetilde{\Pi} \in NP$ appartiene anche a P, quindi

$$NP \subseteq P$$

e quindi anche che

$$P = NP.$$

□

Sorgono spontanee due domande:

- *esistono problemi in NP-C?*
- *se sì, sono state trovate soluzioni efficienti?*

Alla prima domanda rispondiamo **SI**: il primo problema *NP*-completo è proprio *CNF-SAT*, dimostrato nel 1970 con il teorema di Cooke-Levin. L'appartenenza a *NP* è banale, mentre la sua completezza è *noiosa da leggere* (cit. Mereghetti).

Una prima variante di questo problema è *K-CNF-SAT*: viene limitata la cardinalità delle clausole C_i, formate da *or* e *not*, a k letterali. Ad esempio, se $k = 4$ una clausola può essere $\left(a \vee \overline{b} \vee c \vee d\right)$. Con questa variante:

- se $k = 1$ il problema ammette soluzioni in tempo lineare;
- se $k = 2$ il problema ammette soluzioni in tempo quadratico, dimostrato in un articolo del 1975;
- se $k \geq 3$ il problema è *NP*-completo.

Una seconda variante limita invece il soddisfacimento di *al massimo k clausole*.

Infine, una terza variante è quella che considera le *CNF con clausole di Horn*, ovvero clausole in cui esiste al più un letterale negato. In questo caso particolare, il problema torna ad essere efficientemente risolubile.

Come vediamo, quando ho un problema difficile cerco delle varianti interessanti con la speranza di gestire almeno questi sotto-problemi significativi.

Un altro problema *NP*-completo è *HC* (*Hamiltonian Circuit*), allo stesso modo di altri migliaia di problemi interessanti sui grafi e altri di svariati ambiti.

La comunità scientifica, dopo anni di tentativi e svariate ragioni, tende ormai a credere che $P \neq NP$, di conseguenza dimostrare l'*NP*-completezza di un problema implica sancirne l'inefficienza di qualunque algoritmo di soluzione.

Dopo aver stabilito l'NP-completezza, l'indagine sui problemi però non si ferma: si provano restrizioni, algoritmi probabilistici efficienti (con margine di errore), euristiche veloci di soluzione, eccetera. Questo perché questi problemi sono estremamente comuni e utili, quindi stabilire la loro inefficienza darebbe il via alla ricerca del *miglior algoritmo che più si approssima a quella che consideriamo efficienza*.

4.2.3. Dimostrare la *NP*-completezza

Vediamo ora una tecnica per mostrare che un problema Π è *NP*-completo:

1. dimostrare che $\Pi \in NP$, solitamente il punto più semplice;
2. scegliere un problema X notoriamente *NP*-completo;
3. dimostrare che $X \underset{P}{\leq} \Pi$;
4. per la transitività di $\underset{P}{\leq}$ si ottiene che $\Pi \in NPC$.

Infatti:

1. $\Pi \in NP$ per il punto 1;
2. $\forall \widetilde{\Pi} \in NP$ abbiamo che

$$\widetilde{\Pi} \overset{(2)}{\underset{P}{\leq}} X \overset{(3)}{\underset{P}{\leq}} \Pi.$$

Ma quindi per la transitività di $\underset{P}{\leq}$ del punto 4 abbiamo che $\forall \widetilde{\Pi} \in NP$ allora $\widetilde{\Pi} \leq \Pi$, quindi Π è *NP*-completo.

4.3. $P \subseteq L$?

C'è un'inclusione che abbiamo lasciato in sospeso, ed è proprio quella che coinvolge le classi L e P. Ricordiamo che

$$L = DSPACE(\log(n)),$$
$$P = \bigcup_{k \geq 0} DTIME\left(n^k\right).$$

Abbiamo mostrato che $L \subseteq P$, *ma l'inclusione è propria? Oppure* $P \subseteq L$*?*

Possiamo procedere in maniera simile a quanto fatto per $NP \subseteq P$:

1. stabiliamo una **relazione di difficoltà** tra problemi $\pi_1 \leq \pi_2$ che voglia dire: *se esiste una soluzione efficiente in spazio per π_2, allora esiste automaticamente una soluzione efficiente per π_1*;
2. trovare i problemi **massimali** in P secondo la relazione definita al punto precedente;
3. restringere la ricerca di algoritmi efficienti in spazio a questi problemi massimali. Se la ricerca porta un successo su un problema massimale, allora $P \subseteq L$ e quindi $P = L$.

4.3.1. Riduzione in spazio logaritmico

Definiamo una riduzione utile al procedimento appena spiegato.

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ (o due problemi di decisione), diciamo che L_1 si **log-space riduce** a L_2, e lo indichiamo con $L_1 \underset{l}{\leq} L_2$, se e solo se esiste una funzione $f : \Sigma^* \longrightarrow \Sigma^*$ tale che:

1. f è calcolabile su una DTM in **spazio logaritmico**, ovvero $f \in FL$;
2. $\forall x \in \Sigma^* \quad x \in L_1 \iff f(x) \in L_2$.

Similmente a quanto vista prima, esiste un teorema che dimostra una sorta di transitività per due linguaggi tra cui esiste una relazione di riducibilità.

Teorema Siano due linguaggi $A, B \in \Sigma^* \mid A \underset{l}{\leq} B$. Allora

$$B \in L \Longrightarrow A \in L.$$

Dimostrazione

Sappiamo che $A \underset{l}{\leq} B$, quindi esiste $f \in FL$ funzione di log-space riduzione. Consideriamo il seguente algoritmo:

$$\begin{aligned} P \equiv\ & \text{input}(x) \\ & y := f(x); \\ & \text{if } (y \in B) \\ & \quad \text{return } 1 \\ & \text{else} \\ & \quad \text{return } 0. \end{aligned}$$

Questo è sicuramente un algoritmo deterministico, in quanto è composto da *"moduli"* a loro volta deterministici. Inoltre, riconosce A per via della seconda condizione della riducibilità.

La sua complessità in spazio, dati input x di lunghezza n, è descritta dalla seguente complessità:

$$s(n) = s_f(n) + s_{y \in B}(|y|) = O(\log(n)) + O(\log(|y|)).$$

Quanto sarà la lunghezza di y?

Notiamo che $|y| \leq p(n)$, perché è output di una procedura che impiega spazio logaritmico e quindi un numero polinomiale di passi. Questo deriva dalla relazione $FL \subseteq FP$. Quindi:

$$s(n) = O(\log(n)) + O(\log(p(n))) = O(\log(n)).$$

In conclusione, l'algoritmo deterministico proposto riconosce A in spazio logaritmico, dunque:

$$A \underset{l}{\leq} B \wedge B \in L \Longrightarrow A \in L.$$

□

Prima di continuare dobbiamo fare attenzione ad un dettaglio: la generazione di y è sì logaritmica in spazio ma serve comunque spazio per salvare il valore di questa variabile, che può essere polinomiale. Si utilizza allora la **computazione on demand**: viene generata y bit per bit, non tutta insieme. Il dato che viene salvato è l'indice del bit richiesto dalla funzione che controlla l'appartenenza a B, ma l'indice è logaritmico se lo consideriamo in binario, quindi stiamo utilizzando spazio logaritmico.

4.3.2. Problemi *P*-completi

Come prima, identifichiamo il sottoinsieme di problemi di P che andremo a studiare.

Un problema di decisione Π è ***P*-completo** se e solo se:

1. $\Pi \in P$;
2. $\forall \widetilde{\Pi} \in P \quad \widetilde{\Pi} \underset{l}{\leq} \Pi$.

Chiamiamo PC la sottoclasse di P dei problemi P-completi.

Teorema Sia $\Pi \in PC$ e $\Pi \in L$. Allora $P \subseteq L$.

Dimostrazione
Sappiamo che $\Pi \in PC$, quindi

$$\forall \widetilde{\Pi} \in P \quad \widetilde{\Pi} \underset{l}{\leq} \Pi.$$

Se assumiamo che $\Pi \in L$ otteniamo che

$$\forall \widetilde{\Pi} \in P \quad \widetilde{\Pi} \in L,$$

quindi possiamo concludere che $P \subseteq L$ e, di conseguenza, $P = L$. □

Un esempio di problema P-completo è quello che si chiede se una stringa x appartiene a una certa grammatica context-free.

- Nome: context-free membership.
- Istanza: grammatica-context-free G, stringa x.
- Domanda: $x \in L(G)$?

Questo problema viene risolto in tempo $t(n) = n^2 \log(n)$ e in spazio $s(n) = O(\log^2(n))$.

Un altro problema P-completo è il seguente.

- Nome: circuit value (*circuito con porte logiche*).
- Istanza: circuito booleano $\mathcal{C}(x_1, ..., x_n)$, valori in input $a_1, ..., a_n \in \{0,1\}^*$.
- Domanda: $\mathcal{C}(a_1, ..., a_n) = 1$?

Anche in questo caso lo spazio utilizzato è $s(n) = \log^2(n)$.

Come per $P \subseteq NP$, è universale assumere che l'inclusione $L \subseteq P$ sia propria, ma questo non frena gli studi su questi problemi, che sono attuali e molto utili. Si può anche dimostrare che i problemi P-completi non ammettono (*quasi certamente*) algoritmi paralleli efficienti.

4.3.3. Dimostrare la *P*-completezza

Infine, vediamo anche per questi problemi una tecnica per mostrare che un problema Π è *P*-completo:

1. dimostrare che $\Pi \in P$;
2. scegliere un problema X notoriamente *P*-completo;
3. dimostrare che $X \underset{l}{\leq} \Pi$;
4. per la transitività di $\underset{l}{\leq}$ si ottiene che $\Pi \in PC$.

4.4. Problemi *NP*-hard

*Ma i problemi **NP-hard**? Cosa sono e dove si posizionano?*

Un problema Π *NP*-completo è tale che:

- $\Pi \in NP$;
- se $\Pi \in P$ allora $P = NP$.

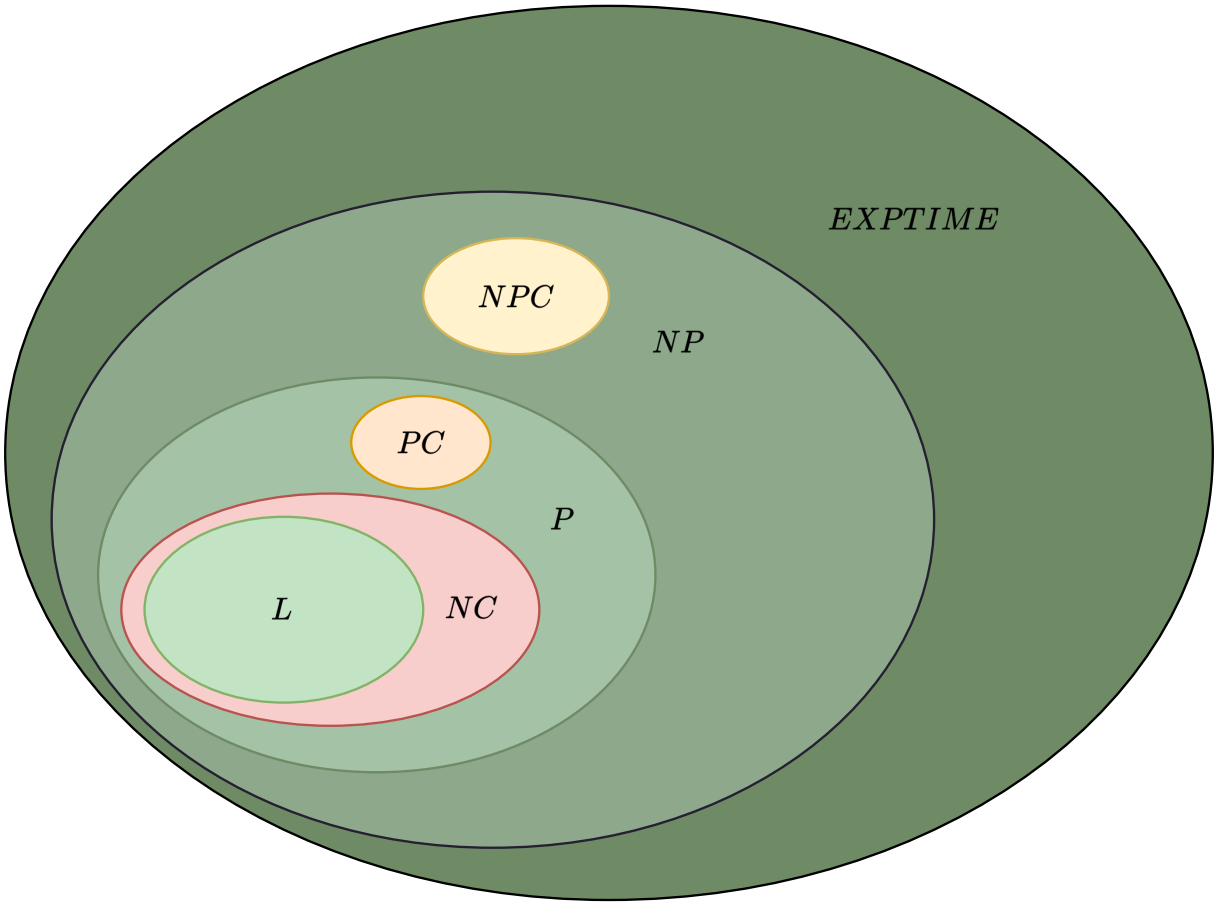
Se invece Π è *NP*-hard vale solo che:

- se $\Pi \in P$ allora $P = NP$.

Sono problemi che *creano il collasso* di *P* e *NP* ma non sono per forza problemi di decisione: ad esempio, *NP*-hard contiene tutti i **problemi di ottimizzazione** (*number-CNF-SAT*, quanti assegnamenti sono soddisfatti) oppure i **problemi enumerativi**.

4.5. Situazione finale

Dopo tutto ciò che è stato visto in queste dispense, ecco un'illustrazione che mostra qual è la situazione attuale per quanto riguarda la classificazione di problemi.



NC è la classe di problemi risolti da **algoritmi paralleli efficienti**, ovvero algoritmi che hanno tempo parallelo *o piccolo* del tempo sequenziale e un buon numero di processori.

L'unica inclusione propria dimostrata e nota è $P \subsetneq EXPTIME$, grazie al problema di decidere se una DTM si arresta entro n passi. In tutti gli altri casi è universalmente accettato (*ma non dimostrato*) che le inclusioni siano proprie.