Informatica Teorica

Manuel Pagliuca

13 novembre 2021

Indice

1 Introduzione

Nei corsi di informatica applicata come quelli di: Sistemi Operativi, Basi di dati, ecc... l'oggetto di studio è definito dal corso, e l'informatica è lo strumento per studiare questo oggetto.

Nel corso di informatica teorica l'oggetto di studio è l'informatica stessa, si studiano i fondamenti dell'informatica (come un corso di sistemi operativi effettua uno studio sui fondamenti dei sistemi operativi).

Per eseguire lo studio di questa disciplina ci si pone le due domande fondamentali nei confronti dell'informatica *cosa* e *come*.

1.1 Cosa studia l'informatica?

L'informatica è una disciplina che studia l'informazione e l'elaborazione automatica mediante sistemi di calcolo che eseguono programmi.

Ma sappiamo che tutti i problemi sono risolubili per via automatica, e questo porta proprio alla nostra domanda, quali problemi sono in grado di risolvere *automaticamente*?

Ciò che studia il *cosa* dell'informatica si chiama **teoria della calcolabilità**, mostreremo nelle successive lezioni che non è una domanda cosi facile da rispondere, poichè esistono delle cose che non sono calcolabili. Quindi dato che non ci sono cose calcolabili, la teoria della calcolabilità si domanda *che cosa è calcolabile?*.

Nella teoria della calcolabilità vogliamo una risposta generale, non cerchiamo dei casi particolari per dire questo è calcolabile o meno, esistono delle proprietà che accomunano tutto ciò che è calcolabile ? La risposta è si, la nozione di calcolabilità è denotabile attraverso la matematica e quindi posso affrontare l'insieme di cose fattibili dell'informatica con gli strumenti della matematica.

1.2 Come l'informatica risolve i problemi?

La branca dell'informatica che si chiama **teoria della complessità** risponde alla domanda *come è risolubile questo problema ?*. Essa studia la quantità di risorse computazionali richieste dalla *soluzione automatica* dei problemi. Una *risorsa computazionale* è qualsiasi cosa venga sprecata per l'esecuzione dei programmi:

Le principali risorse su cui ci concentreremo sono il **tempo** e **spazio di memoria**, dovremo quindi dare una definizione rigorosa di queste risorse computazionali. Successivamente si potrà porre delle domande ovvie : *quale è la classe di problemi che vengono risolti efficientemente in termini di tempo e di memoria ?*. Notare che nella teoria della complessità non si parla solo di **risolubilità** (come nella teoria della calcolabilità) ma anche dell'**efficienza** con cui risolvo questo problema.

Esistono problemi che si trovano in una zona grigia, che non sappiamo se hanno una risoluzione efficiente, ma sono problemi molto importanti, nessuno è riuscito a dimostrare se avranno soluzioni efficienti ma nemmeno il contrario, ovvero nessuno è riuscito a dimostrare che saranno risolubili efficientemente.

Questa classe di problemi è la classe di problemi NP, vedremo poi di cosa si tratta.

2 Syllabus

- Teoria della calcolabilità: individuare la qualità della calcolabilità dei problemi, quali sono le categorie di problemi calcolabili e distinguerla da quella dei problemi non calcolabili.
- Teoria della complessità: studio quantitativo dei problemi, dopo aver delimitato il confini di ciò che è calcolabile cercare un sotto insieme di problemi **efficientemente calcolabili**.

3 Il nostro linguaggio: la matematica

3.1 Funzione

Una funzione dall'insieme A all'insieme B è una **legge**, che chiamiamo solitamente f, che spiega come associare ad ogni elemento di A un elemento di B.

Dal punto di vista formale l'espressione della funzione viene definita **globalmente**:

$$f : A \to B$$

Dove A viene chiamato **dominio** e B **codominio**, questa notazione dice che ogni elemento del dominio è associato attraverso una legge f ad un elemento del codominio. Esiste anche uan notazione che permette di stabilire localmente l'operato della funzione, essa rappresenta l'operato della legge f sull'elemento a che porta all'elemento b.

$$a \xrightarrow{\text{f}} b$$

La notazione comunemente più utilizzata (in particolare nei libri di testo, ma anche nei corsi di matematica), è la seguente:

$$f(a) = b$$

Solitamente b è l'**immagine** di a secondo f, e meno usualmente si dice che a è la **controimmagine** di b (sempre secondo f).

Per esempio:

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

Dove $\mathbb{N}$ è l'insieme dei numberi naturali $0, 1, 2, 3, ...$, e per utilizzi futuri denotiamo con $\mathbb{N}^+$ l'insieme dei numeri naturali positivi (zero escluso) $1, 2, 3, 4, ...$
Ora vediamo la specifica della funzione f

$$f(n) = \lfloor \sqrt{n} \rfloor$$

Considerando $n = 5$ l'immagine di quest'ultima sarà $f(5) = \rfloor \sqrt{5} \lfloor = 2$.
Quindi possiamo dedurre che per più elementi del dominio una **funzione** effettua un mapping ad uno ed un solo valore del codominio (nel caso in cui un valore del dominio venga mappato a più valori del codominio allora si parla di relazione, ma non più di funzione).

3.2 Classi di funzioni

3.2.1 Funzioni iniettive

Una funzione è **iniettiva** se e solo se elementi diversi del dominio vengono mappati in elementi diversi del codominio.

$$f : A \to B \text{ è iniettiva sse } \forall a_1, a_2 \in A \text{ dove } a_1 \neq a_2 \implies f(a_1) \neq f(a_2)$$

Esempio 1

$$f(n) = \lfloor \sqrt{(n)} \rfloor$$

Abbiamo visto precedentemente che questa funzione non è iniettiva, perché avviene un mapping per più elementi del dominio ad un unico elemento del codominio.

Esempio 2

$$f(n) = [n]_2$$

Questa funzione è fortemente non iniettiva, le due metà dell'insieme dei numeri naturali vengono mappate solamente su due numeri $f(2k) = 0$ e $f(2k+1) = 1$.

Esempio 3

$$f(n) = n^2$$

Questa è una funzione iniettiva, poiché ad ogni controimmagine corrisponde una immagine distinta.

3.2.2 Funzioni suriettive

Una funzione è suriettiva quando tutti gli elementi del codominio hanno una corrispondenza con un elemento del dominio.

$$f : A \implies B \text{ sse } \forall b \in B, \exists a \in A : f(a) = b$$

Esempio 1
$f(n) = \lfloor \sqrt{n} \rfloor$ è suriettiva, questo perchè $\forall m \in \mathbb{N}, m = \lfloor \sqrt{m^2} \rfloor = f(m^2)$. Sostanzialmente, posso tornare con facilità al dominio perchè mi basta elevare al quadrato l'immagine, e questo è fattibile per tutto l'insieme dei numeri naturali.

Esempio 2
$f(n) = [n]_2$ non è una funzione suriettiva, questo perchè per esempio 3 non è immagine di niente rispetto a f (il codominio è tutto).

3.3 Insieme immagine di una funzione

$$Im_f = b \in B : \exists a, f(a) = b = f(a) : a \in A$$

Data f definitiamo l'**insieme immagine di** f come gli elementi del codominio $\in B$ che sono immagine di un elemento del dominio A.
La relazione tra questo insieme Im_f ed il codominio stesso di f quale è B, consiste in:

$$Im_f \subseteq B$$

Allora possiamo dire che una funzione è suriettiva quando:

$$Im_f = B$$

Esempi

$$Im_{\lfloor\sqrt{n}\rfloor} = \mathbb{N} \implies f(n) = \lfloor\sqrt{n}\rfloor \text{ è suriettiva}$$

$$Im_{[n]_2} = 0, 1 \subseteq \mathbb{N} \implies f(n) = [n]_2 \text{ non è suriettiva}$$

3.4 Funzioni biettive

Una funzione si dice biettiva quando è sia suriettiva che iniettiva, devono valere entrambe le due condizioni (questo due condizioni è possibile fonderle in un unica condizione).

$$f : A \to B \textbf{ sse}$$

$$\forall a_1, a_2 \in A, a_1 \neq a_2 \implies f(a_1) \neq f(a_2) \land \forall b \in B, \exists a \in A : f(a) = b$$

Che converge in un unica definizione:

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

Per esempio: $f(n) = n$, oppure considerando gli insiemi dei numeri reali $f(x) = x^3$. Solo per questa tipologia di funzioni esiste il concetto di **funzione inversa**.

3.5 Inversa di una funzione

Data una funzione f biettiva si definisce l'inversa come f^{-1} la funzione tale che crei un mapping tra l'immagine del codominio rispetto alla controimmagine del dominio.

$$f : A \to B$$

$$f^{-1} : B \to A \text{ tale che } f^{-1}(b) = a \iff (a) = b$$

Per esempio l'inversa di $f(n)$ è $f^{-1} = n$, oppure l'inversa di $f(x) = x^3$ è $f^{-1} = \sqrt[3]{x}$ (considerando l'insieme dei numeri reali).

3.6 Composizione di funzioni

Date due funzioni $f : A \to B$ e $g : B \to C$, notiamo che queste funzioni hanno una caratteristica in comune, ovvero che il codominio di una è il dominio dell'altra. Definiamo la composizione di funzione $g \circ f : A \to C$ come la funzione che va da dal dominio di f al codominio di g, definita come $g(f(a))$.

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

Per esempio $f(n) = n + 1$ e $g(n) = n^2$:

- f composto $g : g \circ f(n) = (n+1)^2$
- g composto $f : f \circ g(n) = n^2 + 1$

N.B. L'operazione di composizione restituisce una funzione, e l'operatore $\circ$ non è commutativo, però quando dominio e codominio lo permettono è **associativo**: $(f \circ g) \circ h = f \circ (g \circ h)$.

3.6.1 Funzione identità

La funzione identità sull'insieme A è una funzione che effettua un mappaggio ricorsivo sullo stesso elemento.

$$i_A : A \to A : i_A(a) = a \; \forall a \in A$$

Per esempio la funzione identità sull'insieme $\mathbb{N}$ è $i_{\mathbb{N}}(n) = n$.

3.6.2 Definizione alternativa di funzione inversa

Data una funzione $f : A \to B$ biettiva, la sua inversa è l'unica funzione $f^{-1} : B \to A$ che soddisfa:

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

$$oppure$$

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

Infatti considerando $f^{-1} \circ f(x) = \sqrt[3]{x^3} = x = i_{\mathbb{N}}(x)$ e $f \circ f^{-1}(x) = (\sqrt[3]{x})^3 = x = i_{\mathbb{N}}(x)$

3.6.3 Funzioni totali e parziali

Considerando una funzione $f : A \to B$ essa è una legge che ad **ogni** elemento di A si associa un elemento di B, questo significa che ogni immagine $f(a)$ è definita per ogni elemento $a \in A$. Esiste un apposita notazione:

$$f(a) \downarrow \forall a \in A$$

Una funzione di questo tipo viene chiamata **totale** poiché risulta definita sulla totalità del suo dominio.

Certe funzioni potrebbero *non essere definita* per quale che elemento di $a \in A$, e quindi non avere delle immagini corrispondenti, la notazione:

$$f(a) \uparrow$$

Ovvero, che per un elemento a non esiste immagine nell'insieme B tramite la funzione f.

Consideriamo il seguente esempio:

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

$$f(n) = \lfloor \frac{1}{n} \rfloor \text{ non è definita su } n = 0 \implies f(0) \uparrow \forall n \in \mathbb{N} \setminus 0, f(n) \downarrow$$

Una funzione viene definita **parziale** se a *qualche* elemento di A si associa un elemento di B. Si amplia un nuovo concetto che è quello del **dominio** (o campo di esistenza) della funzione, ovvero quell'insieme costituito da tutti gli elementi di A tali per cui è definita una immagine appartenente a B.

$$Dom_f = \{a \in A : f(a) \downarrow\} \subseteq A$$

Allora vale precisare le due seguenti regole:

$$Dom_f \nsubseteq A \implies f \text{ parziale}$$

$$Dom_f \equiv A \implies f \text{ totale}$$

Alcuni esempi:

$$f(n) = \left\{ \frac{1}{n} + \frac{1}{(n-1)(n-2)} \implies Dom_f = \mathbb{N} \setminus \{0, 1, 2 \right\}$$

$$f(n) = \lfloor \log n \rfloor \implies Dom_f = \mathbb{N} \setminus \{0\}$$

$$f(n) = \lfloor \sqrt{-n} \implies Dom_f = \{0\}$$

3.6.4 Totalizzazione di una funzione parziale

Teniamo conto di una cosa, possiamo convenzionalmente rendere totale una funzione parziale, basta estendere il codominio con un **simbolo convenzionale** $\perp$ che buttiamo fuori tutte le volte che la funzione non è definita.

$$f : A \to B \text{ parziale} \implies \widetilde{f} : A \to B \cap \{\perp\}$$

La totalizzazione di f viene raggiunta con l'aggiunta di questo simbolo.

$$\widetilde{f}(a) = \begin{cases} f(a) & \text{se } a \in Dom_f \\ \perp & \text{altrimenti} \end{cases}$$

Quindi per i punti dove il campo di esistenza non è definito verrà restituito $\perp$, per convenzione quando una funzione parziale viene totalizzata ovvero $B \cap \perp$ possiamo utilizzare la seguente notazione $B_\perp$.

3.6.5 Prodotto cartesiano

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

Il **prodotto cartesiano** di due insiemi è l'insieme di coppie dove il primo elemento della coppia appartiene al primo insieme, ed il secondo elemento della coppia appartiene al secondo insieme.

Il prodotto cartesiano è un'operazione che non commuta.

$$A \times B \neq B \times A$$

Ovviamente l'unico caso dove un prodotto cartesiano è commutativo è quando $A \equiv B$. Il prodotto cartesiano può essere esteso al prodotto di ennuple di più insiemi cartesiani, dove questa volta il risultato è costituito da un insieme ordinato (non più di coppie) delle ennuple rispettive agli insiemi di provenienza:

$$A_1 \times A_2 \times ... \times A_n = \{(a_1, a_2, ..., a_n) : a_i \in A_i$$

Associato alla definizione di prodotto cartesiano abbiamo anche quella di **proiettore i'esimo**, essa è una funzione che agisce su un prodotto cartesiano. Ha come dominio l'insieme i-esimo di questo prodotto data una tupla del prodotto cartesiano non fa altro che estrapolare la componente i-esima di quella tupla (*destruttura la tupla*).

$$\pi_i : A_1 \times ... \times A_n \to A_i$$

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

Utilizzeremo la seguente notazione esponenziale per calcolare il prodotto cartesiano di un insieme cartesiano con se stesso:

$$A_1 \times A_2 \times A_3 ... \times A_n = A^n$$

Alcuni esempi:

$C = \{(x, y) \in \mathbb{R}^2 : x^2 + y_2 = 1\} \implies$ punti che si trovano lungo la circonferenza

$I = \{(x, y) \in \mathbb{R}^2 : x^2 + y_2 < 1\} \implies$ punti che si trovano all'interno della circonferenza

$E = \{(x, y) \in \mathbb{R}^2 : x^2 + y_2 > 1\} \implies$ punti che si trovano all'esterno della circonferenza

$$C \cap I \cap E = \mathbb{R}^2$$

3.6.6 Insieme di funzioni

L'insieme delle funzioni che vanno dall'insieme A a B viene indicato con:

$$B^A = \{f : A \to B\} = \text{ insieme delle funzioni da } A \text{ a } B$$

L'insieme delle funzioni *parziali* che vanno da A a B:

$$B_\perp^A = \{f : A \to B_\perp\} = \text{ insieme delle funzioni parziali che va da A a B}$$

3.6.7 Funzione di valutazione

Dati due insiemi A, B e $B_{\perp}^{A}$ si definisce una funzione di valutazione come:

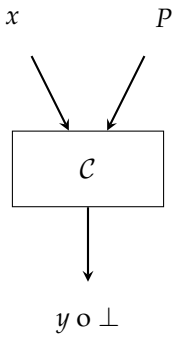
$$\omega : B_{\perp}^{A} \times A \to B \text{ con } \omega(f, a) = f(a)$$

Essa è una funzione che valuta il punto del codominio A in termini di B, ovvero restituisce un $f(a)$, quindi il suo compito è strettamente quello di valutare.

- Tenendo fisso a e facendo variare f, è come se a fosse un *benchmark* su cui testiamo una serie di funzioni.
- Tenendo fisso f e facendo variare a ottengo il grafico di f.

3.7 Modellare teoricamente un sistema di calcolo

Un sistema di calcolo è architettato come un architettura di Von Neumann, è un sistema che dato in input un *dato* x ed un *programma* P. L'output può essere indicato con y quando è definito, mentre con $\perp$ quando va in *loop*.



$P \in PROG$: Un programma è una **sequenza di regole** scritte in un certo linguaggio che prende un input e lo trasforma in un output (o in un loop). Essenzialmente un programma è la definizione di una funzione scritta a mano, esso realizza una funzione che parte dai dati in input ed ottiene i dati in output.

$P \in DATI_{\perp}^{D}ATI$ è una funzione in un linguaggio di programmazione.

Quindi, ribadendo per l'ennesima volta un programma è la realizzazione di una funzione, che va da un insieme di dati ad un altro.

Il sistema di calcolo in se prende in input dei dati ed una funzione (il programma), che mi da un output, esso è definito come una **funzione di valutazione** $\mathcal{C}$.

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

Quindi $\mathcal{C}$ è una funzione di valutazione, e $\mathcal{C}(P, x)$ è la funzione di valutazione del programma P sul dato x.

Per esempio, consideriamo il seguente programma:

Input: x
if $\langle x \rangle_2 == 0$ **then**
  return x;
else
  while $1 < 2$ **do** ;
  return 1;
end

Algorithm 1: Semantica di P

Voglio capire la semantica di questo programma è per capirlo voglio definire una funzione che mi rappresenti il legame input-output rispetto ad un dato elemento.

Il programma prende in ingresso un numero intero, se il numero è pari restituisce esattamente il numero intero, altrimenti entra in un loop (non termina).

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

La **semantica** è la seguente :

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

3.7.1 Potenza computazionale

I sistemi di calcolo servono per calcolare le funzioni, ogni programma che immetto nel mio sistema di calcolo mi viene calcolato. Indico con potenza computazionale del mio sistema di calcolo è *tutto quello che sa fare* il mio sistema di calcolo, o meglio, tutte quelle funzioni che il mio sistema di calcolo può calcolare.

Tutte le funzioni che può calcolare il mio sistema di calcolo è un sottoinsieme di tutte le funzioni immaginabili che possono andare da dati in dati. I programmi in generale sono delle funzioni da dati in dati, quindi il mio sistema di calcolo è ovvio che calcoli questo tipo di funzioni, è un sottoinsieme perchè non so quali funzioni è in grado di risolvere il mio sistema di calcolo (e questa è la domanda a *cosa* a cui risponde l'informatica teorica).

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

Questo significa che esistono delle funzioni che non possono essere risolte dal mio sistema di calcolo, e che quindi **non sono automatizzabili**.

$$F(\mathcal{C}) \nsubseteq DATI_\perp^{DATI}$$

Sono presenti funzioni che l'informatica può risolvere e fare tutto.

$$F(\mathcal{C}) = DATI_{\perp}^{DATI}$$

Quindi abbiamo ridotto il quesito *che cosa sono in grado di fare i programmi*, ad un quesito matematico di inclusione propria ed impropria.

3.7.2 Importanza del calcolo di funzione

Calcolare una funzione significa risolvere problemi in *generale*. Questo perchè ad ogni problema posso associare una *funzione soluzione*, non è altro che quella funzione che ad un dato input associa una determinata *soluzione*. Per esempio, il *problema del calcolo del determinante*:

DET: **Input:** $M \in \mathbb{R}^{n \times n}$
Output: Determinante di M

La rispettiva funzione soluzione:

$$f_{DET} : \mathbb{N}^{n \times n} \to \mathbb{Z}$$

$$f_{DET}(M) = Det(M) :$$

Vediamo che risolvere il dato problema consiste nel calcolare la funzione soluzione, o risolvere il problema significa sostanzialmente avere un programma in grado di risolvere quella funzione.

Find-Replace: **Input:** Testo, parola da cercare, parola da sostituire
Output: Testo in cui ogni occorrenza della parola da cercare è sostituita dall'altra

La funzione soluzione:

$$f_{Find-Replace} : TESTI \times PAROLE \times PAROLE \to TESTI$$

$$f_{Find-Replace}(\text{La nostra vita,nostra,vostra}) = \text{La vostra vita}$$

3.7.3 Cardinalità degli insiemi - I°

Siamo arrivati al primo punto fondamentale, il *cosa* dell'informatica, ed abbiamo visto che si concretizza nell'interrogarsi sulla relazione tra l'insieme della potenza computazionale e quello di tutte le funzioni che vanno da dati in dati possibili.

$$F(\mathcal{C}) \ ? \ DATI_{\perp}^{DATI}$$

Per cercare di dare una dimostrazione sul carattere dell'inclusione, è molto utile fare riferimento al concetto matematica di **cardinalità**.

Dato un insieme A, la sua cardinalità si indica con $|A|$. Intuitivamente che la cardinalità di un insieme è il numero di elementi da cui è formato l'insieme. Questa idea intuitiva è abbastanza corretta quando si ha a che fare con insiemi finiti (la cardinalità mi può aiutare a capire quale insieme è più grande degli altri).

Purtroppo però quando tiriamo in ballo insiemi di cardinalità infinita, questo potrebbe portare a conclusioni più complicate da elaborare. Si potrebbe *erroneamente* pensare $|\mathbb{N}| = \infty = |\mathbb{B}|$, ma questo sappiamo che non è assolutamente vero, perchè l'insieme dei numeri reali è molto più fitto di quello dei numeri interi.

Quindi dobbiamo aggiungere dei nuovi concetti, visto che *"l'infinito di* $\mathbb{R}$*"* è diverso da quello di $\mathbb{N}$. Il concetto da introdurre è quello di **relazione**.

3.7.4 Relazione

Consideriamo un insieme A, si definisce **relazione binaria** R su A, il sottoinsieme del prodotto cartesiano di A su se stesso.

$$R \subseteq A^2$$

Gli elementi $a, b \in A$ stanno in una relazione R se e solo se $(a, b) \in R$. Sono presenti due notazioni infisse per denotare l'esistenza della relazione tra i due elementi:

$$a \text{ R } b \text{ oppure } a \not\text{R } b$$

Consideriamo per esempio la relazione R come la relazione che agisce sui numeri naturali, tale che un numero divida l'altro.

$$R \equiv \text{divide} : 3 \text{ R } 6, 5 \text{ R } 45, ..., 3 \not\text{R } 10$$

$$R = \{(a, b) \in \mathbb{N}^2 : \langle b \rangle_a = 0\}$$

Introduciamo anche il concetto di *equivalenza in modulo k*, la quale mi descrive una relazione del tipo:

$$a \equiv_k b \text{ sse } \langle a \rangle_k = \langle b \rangle_k$$

Per esempio: $5 \equiv_2 7, 4 \equiv_2 16, ...$ (quando il resto dato dal divisore k è uguale su entrambi gli operandi).
Parliamo di **relazione di equivalenza** se e solo se soddisfa le seguenti proprietà:

- **Riflessiva**: $\forall a \in A, a \text{ R } a$
- **Simmetrica**: $\forall a, b, a \text{ R } b \Leftrightarrow b \text{ R } a$
- **Transitiva**: $\forall a, b, c, a \text{ R } b \wedge b \text{ R } c \implies a \text{ R } c$

Ora considerando le precedenti relazioni, vediamo che la relazione $R \equiv$ *"divide"* non è una relazione di equivalenza:

- È riflessiva.
- **Non è simmetrica**: $3 \text{ R } 6$ ma $6 \not\text{R } 3$.
- È transitiva.

Mentre per la seconda relazione $\equiv_k$ possiamo dire che essa è una relazione di equivalenza:

- È riflessiva.
- È simmetrica, vengono valutati i modulo non direttamente gli operandi.
- è transitiva (per lo stesso motivo ancora).

Con una relazione di equivalenza è possibile effettua un **partizionamento delle classi di equivalenza**.

3.7.5 Relazioni di equivalenza e partizioni

Considerando una relazione di equivalenza $R \subseteq A^2$ induce una **partizione** su A. Le partizioni sono dei sottoinsiemi $A_1, A_2, A_3, ... \subseteq A$ tali che:

- $A_i \neq \emptyset$ (non sono vuoti).
- $i \neq j \implies A_i \cap A_j = \emptyset$ (non sono sovrapponibili).
- $\bigcup_{i=1} A_i = A$ (la loro unione ricompone A).

Una **classe di equivalenza** di un elemento $a \in A$ è l'insieme di tutti gli elementi che sono in relazione con a, notazione:

$$[a]_R = \{b \in A : a \,\mathrm{R}\, b\}$$

Si dimostra facilmente che :

- Non esistono classi di equivalenza vuote, questo per via della proprietà *riflessiva* delle relazioni di equivalenza.
- Se prendo due elementi diversi del dominio le classi di equivalenza relative o sono disgiunti o sono la stessa classe di equivalenza: $a, b \in A$ vale che $[a]_R \cap [b]_R = \emptyset$ o $[a]_r = [b]_R$.
- $\bigcup_{a \in A} [a]_R = A$

Quindi la partizione indotta da una relazione di equivalenza non è altro che l'insieme delle classi di equivalenza relative a quella relazione.
L'insieme delle classi di equivalenza di R è la partizione indotta da R su A.

Figura 3.1: Insieme A partizionato

L'insieme A partizionato rispetto alla relazione R è detto **insieme quoziente**:

$$A/R$$

Quindi il quoziente di A rispetto a R è lo *"spezzettamento"* di A in classi di equivalenza.

Consideriamo il seguente esempio di insieme quoziente, consideriamo la relazione di equivalenza $\equiv_4 \subseteq \mathbb{N}$. *Quali classi di equivalenza ammette questa relazione ?*

$$[0]_4 = 4k \text{ tutti i multipli di 4 hanno lo stesso resto di 1}$$

$$[1]_4 = 4k+1 \text{ tutti i multipli di 4 aumentati di 1 hanno lo stesso resto 1}$$

$$[2]_4 = 4k+2; [3]_4 = 4k+3; ...$$

Ne esistono altre ? No, non esistono altre classi di equivalenza perché ogni caso successivo a quelli *"base"* si riconduce a quelli *"base"*.

$$\langle n \rangle_2 \in \{0, 1, 2, 3\}$$

Voglio vedere se queste classi di equivalenza può rappresentare una partizione:

- Nessuna classe è vuota.
- Sono mutuamente disgiunte, poiché se prendo un numero esso ricadrà solamente in una di queste classi di equivalenza.
- L'unione di queste 3 classi di equivalenza restituisce l'insieme originale $\bigcup_{i=0}^{3} [i]_4 = \mathbb{N}$.

Figura 3.2: Insieme quoziente di $\equiv_4$

$\{[i]_4 : 0 \leq i \leq 3\}$ è una partizione di $\mathbb{N}$ indotta dalla relazione $\equiv_4$, tale per cui mi dia il rispettivo insieme quoziente (a volte impropriamente chiamato $\mathbb{Z}_4$):

$$N/\equiv_4 \ = \{[i]_4 : 0 \leq i \leq 3\} = \mathbb{Z}_4$$

Adesso abbiamo in mano gli strumenti per definire in maniera molto più fine il concetto di cardinalità.

3.7.6 Cardinalità degli insiemi - II°

Sia $\mathcal{U}$ (insieme universo) la classe di tutti gli insiemi, definiamo la relazione $\sim \subseteq \mathcal{U}^2$ detta relazione di *equi numerosità* (hanno la stessa dimensione numerica), tra le coppie degli insiemi, se e solo se esiste una **biezione** tra A e B (ovvero, se riesco ad esibire una funzione iniettiva e suriettiva che va da A in B).

Questa relazione tra insiemi è una relazione di equivalenza, poiché:

- $\sim$ è riflessiva, se utilizzo la funzione identità i_A.
- $\sim$ è simmetrica, se esiste una biezione $A \to B$ allora esiste una biezione anche $B \to A$. Ovvero $A \sim B$ e $B \sim A$.
- $\sim$ è transitiva, se compongo funzioni biettive ottengo ancora una funzione biettiva.

Due insiemi che stanno in questa relazione vengono detti *equi numerosi*. Ora considerando l'insieme quoziente del nostro universo $\mathcal{U}$ rispetto alla relazione di equi numerosità $\sim$ (quindi stesso numero di elementi in entrambi i due insiemi), esso mi rappresenta il concetto di **cardinalità** di un insieme.

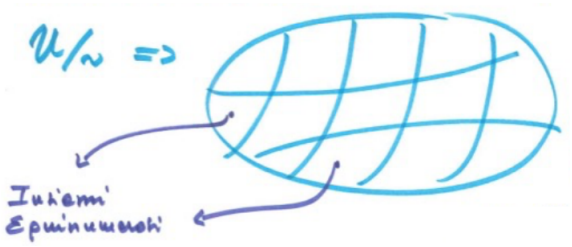


Figura 3.3: Insieme quoziente $\mathcal{U}/\sim$

Quindi spezzettando l'insieme $\mathcal{U}$ in classi di equivalenza in questa classe di equivalenza ci sono tutti gli insiemi che sono equi numerosi (seppur diversi).

Questo concetto permette di parlare in maniera molto precisa anche di insiemi di cardinalità infinita. Questi insiemi possono avere lo stesso numero

Per esempio, consideriamo $n \in \mathbb{N}^+$, si consideri l'insieme $J_n = \{1, 2, , ..., n\}$. Per questo insieme è ovvio quale sia il concetto di cardinalità perchè è finito.

Allora diciamo che un insieme A ha cardinalità **finita** se è equi numeroso con J_n (ovvero $A \sim J_n$) per un dato n ed in quel caso scriviamo che $|A| = n$.

Quindi nella classe di equivalenza J_1 troviamo tutti gli insiemi con un elemento, nella classe di equivalenza di J_2 troveremo tutti gli insiemi con due elementi, ecc.

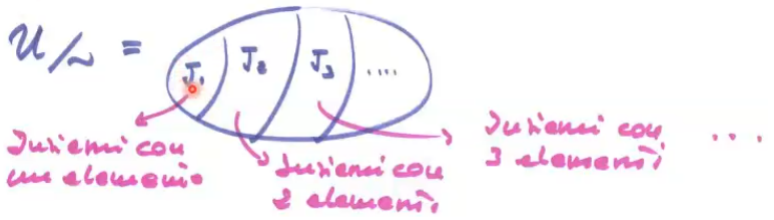


Figura 3.4: Esempio $\mathcal{U}/\sim$ rispetto all'insieme J_n

Un insieme che non ha cardinalità si dice banalmente che ha cardinalità **infinità**. Fin qui abbiamo parlato ancora di cardinalità finita, ma adesso faremo un esempio con cardinalità infinita.

3.7.7 Insiemi numerabili

A si dice **numerabile** se è nella stessa classe di equivalenza dell'insieme dei numeri naturali $\mathbb{N}$, ovvero se esiste una *biezione* tra l'insieme A e l'insieme $\mathbb{N}$.

Siccome stanno in relazione di equi numerosità vuol dire che è presente una biezione fra i due elementi, quindi due elementi non possono collidere.

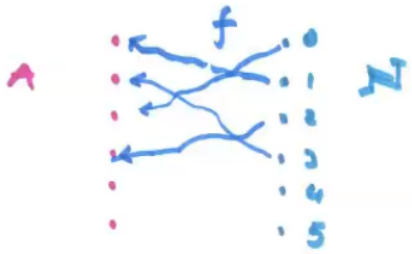


Figura 3.5: A è un insieme numerabile

La presenza di questa biezione significa che come A può essere **listato** con $f(0), f(1), f(2), ...$ su $\mathbb{N}$ senza perdere un elemento, anche l'insieme $\mathbb{N}$ può essere listato a sua volta su A.

Alcuni esempi di insiemi numerabili:

- I numeri pari $f(n) = 2n$, essi sembrerebbero la metà dei numeri naturali ma sono tanti quanto i numeri naturali perchè esiste questa biezione (vale anche per i dispari $f(n) = 2n + 1$).
- L'insieme $\mathbb{Z}$, possono essere presenti diverse funzioni, per esempio posso mappare i negativi sui numeri pari ed i positivi sui numeri dispari.
- L'insieme $\mathbb{Q}$, vedremo più avanti.

- L'insieme delle stringhe binarie che incominciano con 1, ovvero $1\{0,1\}^*$, dove una funzione $f(n) = bin(n)$ è in grado di associare un numero naturale (utilizzando le potenze in posizione) ad una stringa binaria.

3.8 Insiemi non numerabili

Esistono degli insiemi che non sono listabili, dette in altre parole sono più fitti di $\mathbb{N}$, è un altro tipo di categoria di infinito.

$\mathbb{R}$ non è numerabile

Dimostrazione
L'idea consiste nel dimostra per assurdo che non esiste una biezione perchè sono presenti dei "buchi" tra le associazioni. Ordine della dimostrazione:

- Dimostro che $\mathbb{R} \sim [0,1]$ (si dice che è *isomorfo/equi numeroso* all'intervallo $[0,1]$), ovvero che è fitto quanto l'intervallo citato.
- Dimostro che $\mathbb{N} \nsim [0,1]$
- $\mathbb{R} \sim [0,1] \nsim \mathbb{N} \implies \mathbb{R} \nsim \mathbb{N}$

Dimostrazione $R \sim [0,1]$
Significa riuscire a rappresentare una funzione che mappa gli elementi tra i due insiemi in maniera che sia suriettiva ed iniettiva.

Prima di tutto abbiamo una retta cartesiana con un origine fissata e che copre tutti i numeri reali. Pongo l'intervallo $[0,1]$ in modo che si trovi in corrispondenza del punto mediano della retta 0, successivamente prendo un compasso punto il centro nel mediano dell'intervallo e traccio la semi circonferenza rispetto alla metà dell'intervallo.

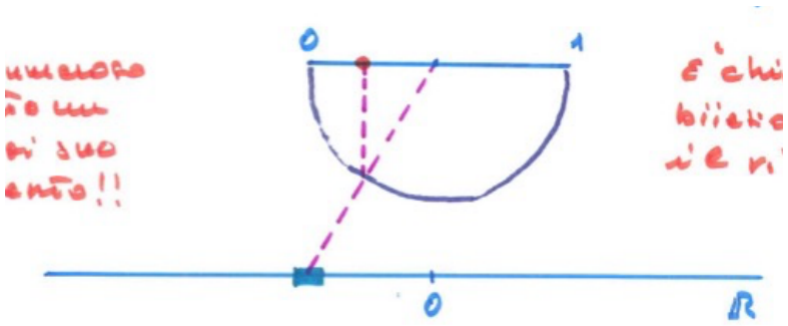


Figura 3.6: Dimostrazione $R \sim [0,1]$

Gli elementi dell'insieme $\mathbb{R}$ vengono mappati tracciando una retta in direzione del punto mediano di $[0,1]$, nel punto di intersezione si traccia la retta

perpendicolare che interseca l'intervallo ed in quel punto si trova il valore corrispondente. Si può fare lo stesso in maniera opposta partendo da $[0,1]$, trovando l'intersezione si parte dal punto mediano di quest'ultima e si attraversa l'intersezione toccando $\mathbb{R}$.

Questo dimostra che tutti i valori di $\mathbb{R}$ sono associabili su $[0,1]$, quindi $\mathbb{R} \sim [0,1]$.

Dimostrazione $\mathbb{N} \nsim [0,1]$ **(per diagonalizzazione)**
Supponiamo per assurdo che $\mathbb{N} \sim [0,1] \implies [0,1]$, ovvero che sia listabile (*elencabile*) in maniera esaustiva così come lo è $\mathbb{N}$. Siccome sto ipotizzando che sia elencabile, allora creo una lista di tutti gli elementi in $[0,1]$, i numeri all'interno di questo numero seguono il formato "0.", quindi:

$$\begin{array}{ccccc} \underline{0.a_{11}} & 0.a_{12} & 0.a_{13} & 0.a_{14} & ... \\ 0.a_{21} & \underline{0.a_{22}} & 0.a_{23} & 0.a_{24} & ... \\ 0.a_{31} & 0.a_{32} & \underline{0.a_{33}} & 0.a_{34} & ... \\ 0.a_{41} & 0.a_{42} & 0.a_{43} & \underline{0.a_{44}} & ... \\ ... & ... & ... & ... & ... \end{array}$$

Se riesco a costruire un numero che non fa parte di questa lista infinita (elusivo alla biezione), allora vuol dire che la lista non è elencabile, e quindi $[0,1]$ non è numerabile.

Per costruire questo numero elusivo che non si trova in nessuno di questi elementi della lista, vado a guardare le cifre sulla *diagonale*.
Costruiamo il numero elusivo alla lista

$$0.c_1c_2c_3c_4$$

Tale che rispetti la seguente regola rispetto alla diagonale del elenco dei numeri $\in [0,1]$

$$c_i = \begin{cases} a_{ii} + 1 \text{ se } a_{ii} < 9 \\ a_{ii} - 1 \text{ se } a_{ii} = 9 \end{cases}$$

Ora usando questa regola non riuscirò mai a posizionare il mio nuovo numero tra quelli elencati, questo perchè ovviamente cambio le cifre (il perché questo accade probabilmente è collegato al fatto che $\mathbb{R}$ è più fitto di $\mathbb{N}$, proprio quello che vogliamo dimostrare). Il numero $0.c_1c_2... \in [0,1]$, ma non appartiene nelle liste poiché:

- Differisce dal primo perché $c_1 \neq a_{11}$
- Differisce dal secondo perché $c_2 \neq a_{22}$
- Differisce da qualunque numero presente sulla diagonale

Quindi la lista non è esaustiva $\implies \mathbb{N} \nsim [0,1]$, significa che non cattura tutto l'intervallo $[0,1]$, quindi non può esistere una biezione tra questo ed $\mathbb{N}$, ovvero $[0,1]$ **non è numerabile**.

Conclusione

$$\mathbb{R} \sim [0,1] \nsim \implies \mathbb{R} \nsim \mathbb{N}$$

- $\mathbb{R}$ non è numerabile.
- Esso è più *"fitto"* di $\mathbb{N}$.
- Qualsiasi tentativo di listare anche solo un segmento non è esaustivo.
- $\mathbb{R}$ è un insieme **continuo**, e tutti gli insiemi equi numerosi $\mathbb{R}$ si dicono continui.
- Altri esempi di insiemi non numerabili:

3.8.1 Insieme delle parti di $\mathbb{N}$

Un altro insieme non numerabile è quello costituito dalla famiglia di sottoinsiemi possibili di $\mathbb{N}$, quindi mettendo assieme uno dopo l'altro tutti i sottoinsiemi di differenti cardinalità (talvolta chiamato **insieme potenza** o **booleano di** $\mathbb{N}$). Per esempio su un insieme $S = \{a, b, c\}$, l'insieme delle parti è $\mathcal{P}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, S\}$.

$$\mathcal{P}(\mathbb{N}) = 2^{\mathbb{N}} = \{\text{sottoinsiemi di } \mathbb{N}\} \nsim \mathbb{N}$$

Anche questa dimostrazione avviene per diagonalizzazione, per assurdo suppongo che esista una biezione che mi permetta di elencare tutti i sottoinsiemi di $\mathbb{N}$.

Posso rappresentare i sotto insiemi di $\mathbb{N}$ utilizzando un **vettore caratteristico** $A \subseteq \mathbb{N}$, questo è un vettore dove metto il bit di appartenenza rispetto alla corrispondenza dell'elemento.

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

Allora se io suppongo per assurdo che $\mathcal{P}(\mathbb{N}) \sim \mathbb{N}$, ovvero che è numerabile rispetto all'insieme dei numeri naturali, allora posso elencare in maniera esaustiva i sotto insiemi di $\mathbb{N}$, questo elencandone i vettori caratteristici.

$$\begin{array}{ccccc} \underline{0.b_{11}} & 0.b_{12} & 0.b_{13} & 0.b_{14} & \ldots \\ 0.b_{21} & \underline{0.b_{22}} & 0.b_{23} & 0.b_{24} & \ldots \\ 0.b_{31} & 0.b_{32} & \underline{0.b_{33}} & 0.b_{34} & \ldots \\ 0.b_{41} & 0.b_{42} & 0.b_{43} & \underline{0.b_{44}} & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

Se riesco a costruire un sottoinsieme di $\mathbb{N}$ (vettore caratteristico) che non fa parte da di questa lista, allora l'insieme delle parti non è un insieme numerabile poiché non è equi-numeroso con $\mathbb{N}$. Questo è possibile procedendo per *diagonalizzazione*, per esempio se considerassi il seguente vettore negato:

$$\begin{array}{cccc} \overline{b_{01}} & \overline{b_{12}} & \overline{b_{23}} & \overline{b_{34}} \quad \ldots \end{array}$$

riesco a constatare che esso non appartiene alla lista nonostante sia un sottoinsieme di $\mathbb{N}$ in quanto composto solo da valori booleani.

$$\mathcal{P}(\mathbb{N}) \nsim \mathbb{N}$$

3.8.2 Insieme $\mathbb{N}^{\mathbb{N}}$

Consideriamo l'insieme di tutte le funzioni possibili funzioni che vanno da $\mathbb{N}$ in $\mathbb{N}$.

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

Ipotizzando che $\mathbb{N}^{\mathbb{N}} \sim$, consideriamo la seguente tabella dove nella prima colonna si trova l'elenco esaustivo di tutte le funzioni $\in \mathbb{N}$, e nelle colonne successive tutto il dominio completo di $\mathbb{N}$. Significa che per ogni riga sara presente l'insieme dei valori che fornisce il grafico di una funzione f_i.

Elenco delle funzioni di $\mathbb{N}$	**0**	**1**	**2**	**3**	**...**$\in \mathbb{N}$
f_0	$\underline{f_0(0)}$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
f_1	$f_1(0)$	$\underline{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
f_2	$f_2(0)$	$f_2(1)$	$\underline{f_2(2)}$	$f_2(3)$	...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$\underline{f_3(3)}$	...
...	...	...	...	...	$\underline{...}$

Allora anche in questo caso per diagonalizzazione voglio costruire una funzione $\in \mathbb{N}^{\mathbb{N}}$ ma elusiva all'elenco delle funzioni della tabella:

$$\varphi : \mathbb{N} \to \mathbb{N} \text{ come } \varphi(n) = f_n(n) + 1$$

Siamo d'accordo che φ sia ben definita, visto che per ogni immagine corrisponderà un immagine numero naturale, ma notiamo che una qualsiasi $\varphi(n)$ presa sulla lista avrà per forza una immagine della diagonale discordante rispetto a quelle elencate nella tabella.
Quindi abbiamo una funzione elusiva all'elenco, e la nostre ipotesi si rivela assurda:

$$\varphi(0) = f_0(0) + 1 \neq f_0(0)$$

$$\mathbb{N}^{\mathbb{N}} \nsim \mathbb{N}$$

N.B.: Gli insiemi $\mathcal{P}(\mathbb{N})$ e $\mathbb{N}^{\mathbb{N}}$ sono detti **insiemi continui**, e sono equinumerosi a $\mathbb{R}$.

3.9 Cosa è calcolabile?

Sappiamo che la potenza computazionale di un sistema di calcolo $\mathcal{C}$ corrisponde a:

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

Il nostro problema consiste nel comprendere se questa inclusione sia propria o impropria, per dimostrare ciò posso utilizzare il concetto di cardinalità precedentemente appreso.
Se riesco a dimostrare che il primo insieme ha una cardinalità numerabile ed il secondo ha una cardinalità continua allora il secondo insieme sarà molto più grande del primo.
Prendiamo alcune **assunzioni** che verranno dimostrate successivamente nel corso:

- $PROG \sim \mathbb{N}$, i programmi sono tanti quanti i numeri naturali, questo perché un programma viene digitalizzato in una fila di bit (finiti). Questo vuol dire che per ogni programma corrisponderà un numero che fa parte dei numeri naturali.
- $DATI \sim \mathbb{N}$, i dati sono tanti quanto i numeri, questo per lo stesso motivo di digitalizzazione (o traduzione) in binario delle informazioni.

Possiamo dire che la potenza computazionale sia equinumerosa rispetto al numero di programmi, al variare del programma cambio la funzione di potenza computazionale. A questo punto agganciamo la prima assunzione.

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

Sapendo che l'insieme dei dati in dati definito come $DATI_{\perp}^{DATI}$, grazie alla seconda assunzione che abbiamo fatto allora possiamo asserire:

$$DATI_{\perp}^{DATI} \sim \mathbb{N}_{\perp}^{\mathbb{N}}$$

Abbiamo visto precedentemente che $\mathbb{N}^{\mathbb{N}} \nsim \mathbb{N}$, allora unendo le precedenti dimostrazioni possiamo confermare:

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

$$F(\mathcal{C}) \nsim DATI_{\perp}^{DATI}$$

Ovvero, che l'insieme dei programmi (corrispondente all'insieme delle funzioni di potenza di calcolo) non è equinumeroso (o *isomorfo*) all'insieme delle funzioni che vanno da dati in dati. In parole povere ho un numero di programmi nettamente inferiore (fitto quanto i numeri naturali) rispetto al numero di funzioni (o problemi) calcolabili (fitto quanto i reali $\mathbb{N}_{\perp}^{\mathbb{N}}$).

3.9.1 Dimostrazione $DATI \sim \mathbb{N}$

Vogliamo stabilire una corrispondenza **biunivoca** tra dati e numeri naturali, ovvero che devo riuscire a "nascondere" un dato all'interno di un numero naturale, tale che se lo dovessi riportare all'insieme dei dati otterrei esattamente quel dato (e non un altro).

Questa biezione la voglia rendere effettiva e programmabile, ovvero costruirci delle primitive che operino direttamente sulla codifica numerica dei

dati così che possiamo smaterializzare completamente il mondo dei dati in quello dei numeri.

Questa volta vogliamo dimostrare che l'**insieme delle coppie** $\mathbb{N} \times \mathbb{N}$ è numerabile. Questo avverrà per passi, prima dimostreremo che l'insieme delle coppie è isomorfo a $\mathbb{N}^+$ (quindi escluso lo 0) e poi di conseguenza che lo è per tutto $\mathbb{N}$. Questo per effetto collaterali che anche l'insieme dei razionali $\mathbb{Q}$ è numerabile, questo perché sono una frazione, quindi una coppia di interi.

Dimostrazione $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

$$\langle , \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$$
$$\langle x, y \rangle = n$$

La **funzione coppia di Cantor** è definita per mezzo delle parentesi angolari, prende due argomenti x, y e restituisce un numero n, questo in maniera iniettiva e suriettiva (biettività).

Ovvero che coppie diverse vengono mappate in numeri diversi, e se ho un numero devo essere in grado di tornare alla stessa coppia x, y. Quindi esibiremo sia l'andata della funzione coppia di Cantor che il ritorno, esibiremo anche i due proiettori *sin* e *des* tali che mi restituiscano i due argomenti in questione.

$$sin : \mathbb{N}^+ \to \mathbb{N} \text{ e } des : \mathbb{N}^+ \to \mathbb{N}$$

$$sin(n) = x \text{ e } des(n) = y$$

Vogliamo dare una rappresentazione grafica di questa funzione, Cantor ha pensato di realizzare una tabella che ha come righe e colonne l'insieme $\mathbb{N}$. In questa tabella il valore della coppia si trova esattamente posizionato all'incrocio tra l'x-esima riga e della y-esima colonna.

x/y	0	1	2	3	4
0	1	3	6	10	15
1	2	5	9	14	...
2	4	8	13	...	...
3	7	12	...	...	...
4	11	...	...	...	...

La disposizione ingegnosa della tabella evita di andare in un elenco infinito, nel senso che se dovessi elencare tutti i numeri naturali andando verso il basso (o destra) non riuscirei a finire mai, ed inoltre non sarebbe un buon approccio visuale per trovare la corrispondenza con l'altro insieme. I numeri sono disposti in maniera che il successivo si trovi sulla diagonale (andando verso l'alto).

$$\langle 2, 1 \rangle = 8$$

Con questa rappresentazione, coppie diverse non riusciranno mai a confluire nella stessa casella (coordinate di punti che individuano un punto univoco), e

viceversa (iniettiva), questo mi conferma che la funzione è biettiva.
Adesso mettiamo da parte la forma tabellare e consideriamo la **forma analitica** di $\langle , \rangle$.

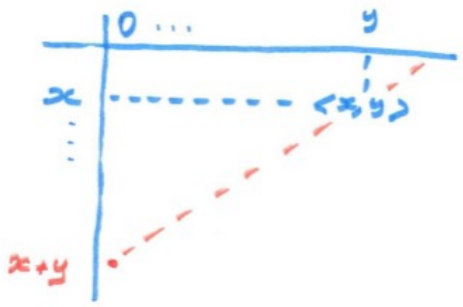


Figura 3.7: Diagonale che passa per $\langle x, y \rangle$

Notiamo che per il valore che voglio valutare attraverso la funzione di Cantor passa una diagonale, questa diagonale parte dalla colonna 0 e dalla riga $x + y$ (il punto $\langle x + y, 0 \rangle$). La funzione coppia nell'origine della diagonale assumerà un certo valore, e visto che i numeri sulla diagonali incrementano in maniera crescente di una sola unità, allora potrò raggiungere $\langle x, y \rangle$ sommando y.
A questo punto il mio problema si riduce a trovare delle coppie particolari ovvero, la cui seconda coordinata è 0, le chiamiamo $\langle z, 0 \rangle$. Notiamo nella prima colonna c'è una correlazione con gli elementi di x, e vediamo che l'elemento i-esimo della prima colonna corrisponde alla somma tra il corrispettivo elemento x e l'elemento $i - 1$ della colonna, allora riusciamo ad esprimere una **legge matematica** per descrivere questa *codifica*.

1. $\langle x, y \rangle = \langle x + y, 0 \rangle + y$
2. $\langle z, 0 \rangle ? \implies \langle z, 0 \rangle = \sum_{i=1}^{z} i + 1 = \frac{z(z+1)}{2} + 1$

Allora $(1) + (2)$:

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

Come tornare da $\mathbb{N}^2$ a $\mathbb{N}^+$ Ovvero, come trovare le funzioni *sin* e *des*.

$$\langle x, y \rangle = n \longrightarrow sin(n) = x \text{ e } des(n) = y$$

Partiamo da un elemento n presente nell'intersezione dei due insiemi definiti da x e y e vogliamo individuare le due relative componenti. Procedo trovando le coordinate dell'origine della diagonale che passa per n, ovvero $\langle \gamma, 0 \rangle$.

Figura 3.8: $n = 13$

Considerando la dimostrazione precedente possiamo dire che $y = n - \langle \gamma, 0 \rangle$, ovvero procedo a ritroso sulla diagonale partendo da n ed andando verso $\langle \gamma, 0 \rangle$.
Allora, sapendo che $\gamma = x + y$ posso trovare facilmente le parti destre e sinistre.
Il problema principale ora è trovare γ, questo non è altro che l'intero più grande valore tale per cui l'origine della diagonale sia minore di n.

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

Quindi dobbiamo risolvere la disuguaglianza la quale ci darà un range su n, all'interno di questo range $\langle z, 0 \rangle \leq n$ andiamo a prendere l'intero più grande che soddisfa questa disuguaglianza.

$$\langle z, 0 \rangle \leq n \longrightarrow \frac{z(z+1)}{2} + 1 \leq n \longrightarrow z^2 + z + 2 - 2n \leq 0$$

$$z_{1,2} = \frac{-1 \pm \sqrt{8n-7}}{2}$$

Quindi devo scoprire quale è l'intero più grande dati:

$$\frac{-1 - \sqrt{8n-7}}{2} \leq z \leq \frac{-1 + \sqrt{8n-7}}{2}$$

l'elemento a destra è il più grande valore, ma non so se è intero, quindi dovrò correggere ciò:

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

Questa è un formula che utilizza la variabile in input n, quindi riusciamo a ricavare il parametro sinistro e destro della funzione di Cantor e ritornare allo

stato iniziale:

$$des(n) = y = n - \langle \gamma, 0 \rangle \text{ e } sin(n) = x = \gamma - y$$

Esempio andata e ritorno tra $\mathbb{N}^2$ e $\mathbb{N}^+$

$$\text{Andata} : \mathbb{N}^2 \longrightarrow \mathbb{N}^+$$

$$\langle 10, 20 \rangle = \frac{30 \cdot 31}{2} + 20 + 1 = 15 \cdot 31 + 21 = 485 + 21 = 486$$

$$\text{Ritorno} : \mathbb{N}^+ \longrightarrow \mathbb{N}^2$$

$$\gamma = \left\lfloor \frac{-1 + \sqrt{8 \cdot 436 - 7}}{2} \right\rfloor = \lfloor 30.6448 \rfloor = 30$$

$$y = 486 - \langle 30, 0 \rangle = 486 - \left(\frac{30 \cdot 31}{2} + 1 \right) = 486 - 466 = 20 = des(486)$$

$$x = 30 - 20 = 10 = sin(486)$$

Adesso noi abbiamo dimostrato che $\mathbb{N}^2 \sim \mathbb{N}^+$.

Dimostrazione $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$ Definisco una nuova funzione:

$$[,] : \mathbb{N} \times \mathbb{N} \text{ tale che } [x, y] = \langle x, y \rangle - 1$$

Adesso ho una funzione esplicita per mappare $\mathbb{N}^2$ su $\mathbb{N}$.
Nota: A questo punto otteniamo che l'insieme dei razionali $\mathbb{Q}$ sono coppie di interi (num, den). Dunque $[,]$ mostra che $\mathbb{Q}$ è numerabile.

Dimostrazione rigorosa che $DATI \approxeq \mathbb{N}$ Mostriamo le nozioni appena apprese sulle principali strutture dati.

Liste d'interi

$$codifichiamo\ x_1, x_2, ..., x_n \rightarrow \langle x_1, x_2, ..., x_n \rangle$$

Quindi il numero che rappresenta la codifica è indicato con il numero all'interno delle parentesi angolari. L'idea consiste nell'incapsulare il risultato di una coppia di elementi come elemento destro della coppia più esterna. Non sapendo quanto sia lunga la lista dovrò porre un elemento che mi segnali il termine, e questo lo posso fare con $\langle x_n, 0 \rangle$.

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

Per esempio:

$$codifica\ 1, 2, 5 \rightarrow \langle 1, 2, 5 \rangle = \langle 1, \langle 2, \langle 5, 0 \rangle \rangle \rangle =$$

$$\langle 1, \langle 2, 16 \rangle \rangle = \langle 1, 188 \rangle =$$
$$= 18144$$

Per l'operazione di decodifica quello che facciamo è di considerare la codifica della mia lista M (mon il risultato ma la lista di coppie incapsulate) come un albero binario. Allora dato questo albero il figlio sinistro è l'elemento sinistro (la testa della lista) ed il figlio destro è la parte restante della lista, allora continuando a considerare il figlio sinistro come una struttura dati lineare ottengo la lista codificata, questo attraversamento dell'albero in pre-ordine terminerà quando incontrerà un figlio destro 0 (l'ultimo elemento, c'è solo un elemento con 0).
Vogliamo effettuare delle implementazione in pseudo-codice (simile C), assumiamo 0 come la lista nulla, e $\langle , \rangle, sin, des$ come la funzione di Cantor $\langle , \rangle$.

Implementazione codifica

```
int encode(x_1,...,x_n){
    int k = 0;
    for(int i = n; i >= 1; i--)
        k = <x_i,k>;
    return k;
}
```

Implementazione decodifica

```
void decode(int n){
    if(n!=0){
        print(sin(n));
        decode(des(n));
    }
}
```

Altre operazioni utili possono essere sono quelle per calcolare la **lunghezza**:

```
int length(int n){
    return n==0 ? 0 : 1 + length(des(n));
}
```

Oppure per calcolare la **proiezione**:

$$proj(t, b) = \begin{cases} -1 & \text{se } t > length(n) \text{ o } t == 0 \\ x_t & \text{se } 1 \leq t \leq length(n) \text{ e } n = \langle x_1, ..., x_t, ..., x_m \rangle \end{cases}$$

```
int proj(int t, int n){
    if (t == 0 || t > length(n))
        return -1;
    else {
```

```
        if (t == 1)
            return sin(n);
        else
            return proj(t-1,des(n));
    }
}
```

Esercizi - *incr, decr* **(da implementare)**

$$incr(t,n) = \begin{cases} -1 & \text{se } t > length(n) \text{ o } t == 0 \\ \langle x_1, ..., x_t + 1, ..., x_n \rangle & \text{se } 1 \leq t \leq length(n) \text{ e } n = \langle x_1, ..., x_t, ..., x_n \rangle \end{cases}$$

$decr(t,n) =$ come sopra ma con $\langle x_1, ..., x_t - 1, ..., x_m$

Corrispondente numerico : $DATI \sim \mathbb{N}$
Adesso sappiamo codificare liste di interi, questo ci da un idea del fatto che i dati siano isomorfi ad $\mathbb{N}$. Questo ci da un modo per nascondere testi, suoni, immagini dietro ad un numero. Per esempio, per compattare un testo in un numero posso imporre una codifica (tipo ASCII), in questa maniera posso associare un numero per ogni carattere numerico, un insieme di caratteri numerici è codificabile utilizzando $\langle , \rangle$ il quale mi darà un singolo numero.

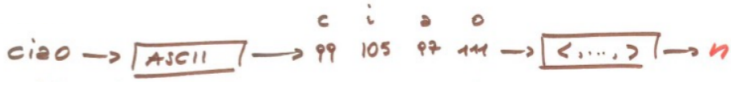


Figura 3.9: Codifica di un testo

Posso usare questo modo per comprimere i testi? In realtà questa tecnica non è un buon compressore, questo perché la crescita del numero è quadratica rispetto al numero di caratteri codificati (otteniamo un n troppo grande e si verifica un espansione, altroché). *Perché non è un buon modo per crittografare i dati?* Il primo problema è che il testo crittografato sarebbe molto lungo (aumenta il rischio nella perdita di dati durante la trasmissione), secondo problema la coppia di Cantor è una funzione biettiva, esiste un modo abbastanza diretto per tornare indietro.

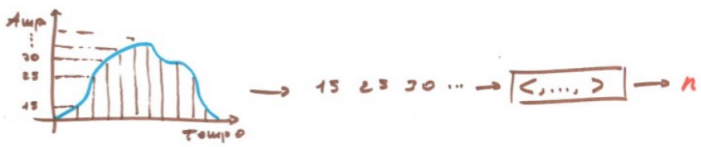


Figura 3.10: Discretizzazione del suono in segnale digitale e poi codifica in numero naturale

Per i suoni il discorso è molto simile, grazie alla **campionatura** (o **discretizzazione**) dei segnali analogici in un segnale digitale, ovvero una grandezza discreta sotto forma di numeri naturali.
Per le immagini potrei usare la tecnica bitmap, quindi per ogni bit registro un colore che verrà codificato con un numero (e da qui come per gli altri casi sappiamo come muoverci).

Codifica di altre stutture dati
Codifica di un **array** a dimensione finita, proprio per questo non necessità di un elemento che mi ponga fine alla sequenza (so già dove fermarmi).

$$x_1, ..., x_n \to [x_1, ..., x_n]$$

$$[x_1, ..., x_n] = [x_1, ..., [x_{n-1}, x_n]]...]$$

Codifica di una **matrice**, esse sono array bidimensionali dove hanno anch'esse dimensione fissa e nota. Quindi mi basta codificare per riga le matrici e poi codificare le codifiche.

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = [[x_{11}, x_{12}], [x_{21}, x_{22}]]$$

Sia per la codifica di array che per quello delle matrici sono facilmente implementabili le primitive numeriche ad esempio $elem(i, j, n)$ che restituisce l'elemento (i, j)-esimo di una matrice codificata dal numero n.
Codifica dei **grafi**, sappiamo che un grafo è rappresentabile con lista di adiacenza, matrice di adiacenza,... quindi possiamo codificare la lista o la matrice.

Conclusione
Abbiamo mostrato effettivamente che grazie alla corrispondenza effettiva di $DATI \sim \mathbb{N}$ che i dati possono essere "scartati" e tenuti solamente i numeri, questo perché ogni dato può essere rappresentato con un numero naturale, questo grazie a delle leggi matematica effettive ed implementabili, che mi permettono di codificare e decodificare il dato.
Quindi la nostra funzione $f : DATI \to DATI_\perp$ che rappresenta un qualsiasi problema può essere sostituita da una funzione $f : \mathbb{N} \to \mathbb{N}_\perp$, ed è questo quello che i nostri calcolatori calcoleranno, quindi possiamo dire che il nostro universo dei problemi $DATI_\perp^{DATI}$ non è altro che $\mathbb{N}_\perp^{\mathbb{N}}$

3.9.2 Prefazione alla dimostrazione $PROG \sim \mathbb{N}$

Per spiegare ciò utilizzeremo un linguaggio apposito detto *"linguaggio RAM"* e su un sistema formale apposito detto *"sistema RAM"*. Grazie alla semplicità di questo potrò mostrare molto semplicemente:

- Che i programmi sono numerabili $PROG \sim \mathbb{N}$
- In maniera rigorosa la semantica dei programmi, $\mathcal{C}(P, \_) \to RAM(P, \_)$

- Ed altrettanto formalmente potrò definire la potenza computazionale $F(RAM)$, questo fornirà un idea di *che cosa è calcolabile*.

Il linguaggio RAM è un *assembly* molto semplificato, per questo motivo l'idea di calcolabilità è altamente criticabile, poiché il modello potrebbe fare scappare qualcosa per via della banalità.

Si introduce un altro modello molto più sofisticato la macchina *while(JVM)*, quello che è calcolabile è effettivamente calcolabile da questo.

Metteremo poi a confronto il modello formale $F(RAM)$ con quello sofisticato $F(while)$. Se quest due idee di calcolabilità risultassero diverse allora la cosa è un po' pericolosa, perchè l'dea di calcolabilità dipenderebbe dal periodo storico. Se invece due idee così diametralmente opposte risultassero uguali, ovvero che calcolano lo stesso insieme di funzioni, allora incomincio a capire che l'idea di calcolabilità è intrinseca ai problemi. Ed è questo che ci porterà alla **tesi di Church** (spoiler).

Prima di arrivare a ciò dobbiamo dimostrare che $PROG \sim \mathbb{N}$, e per fare questo dobbiamo introdurre la macchina RAM.

3.9.3 Sistema di calcolo RAM

L'hardware della macchina RAM:

- Una **memoria** R costituita contigua di registri, ognuno di questi registri può memorizzare i numeri naturali arbitrariamente grandi (non hanno una capienza). Il registro $R1$ è il registro di input; $R0$ registro di output.
- Il **program counter** L, un registro contenente l'indirizzo dell prossima istruzione da eseguire.
- Un **programma** P, costituito da una sequenza di istruzioni, un istruzione RAM può essere:
  - Incremento: $R_k \leftarrow R_k + 1$
  - Decremento: $R_k \leftarrow R_k - 1$ (non può mai andare sotto lo zero)

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

  - Salto condizionato: IF $R_k = 0$ THEN GOTO $m, m \in 1, ..., |P|$

Esecuzione su una macchina RAM

1. Fase di inizializzazione del programma P.
2. Si carica il dato m nel registro di output R_1.
3. Si comincia ad eseguire l'istruzione al posto 1, e man mano il PC continua ad incrementare così da poi eseguire l'istruzione successiva (eccetto nel caso in cui non si incontri un istruzione di salto).

4. Per convenzione la macchina si arresta quanto ik PC il numero 0

$$L = 0 \implies HALT$$

c'è possibilità per via dell'istruzione di salto di mandare in loop l'esecuzione della macchina.

5. L'output di questo programma dato un input x ha due possibilità

   - Il programma si ferma ($HALT$) e l'output è presente all'interno del registro R_0.
   - Il programma non termina, allora in questo caso l'output per l'input sarà indefinito

   Quello che fa il programma è calcolare una funzione:

$$\varphi(x) = cout(R_0)/\perp$$

   Chiameremo **semantica del programma** P:

$$\varphi_p = \mathbb{N} \to \mathbb{N}_\perp$$

Questa è una definizione di semantica del programma molto intuitiva, quello che voglio fare però è utilizzare degli oggetti matematici con i quali specificare in maniera formale e corretta la semantica del programma.

Definizione formale di semantica di un programma RAM
Introduciamo il concetto di **semantica operazionale**, questo è il tipo di semantica più semplice. Significa specificare che cosa fa una data istruzione andando a vedere quale è l'effetto di quell'istruzione sulla data macchina.

Per descrivere l'effetto di un istruzione, prendiamo una foto della macchina prima dell'esecuzione e dopo. La "foto" solitamente si chiama **stato** della macchina. Quindi spiego la semantica di una istruzione specificando il cambiamento di stato indotto dall'esecuzione di quell'istruzione. Questo significa dare la semantica operazionale di una macchina.

Esecuzione di un programma P e sua semantica L'esecuzione di un programma consiste nell'esecuzione di molteplici cambiamenti di stato a partire dallo stato iniziale S_{init} (partendo un inpput n) e giungendo a quello finale S_{final}. Si dice che la computazione di P induce una sequenza di stati.

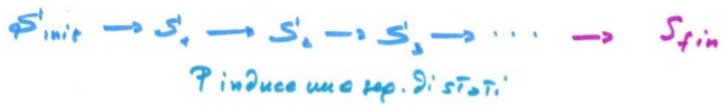


Figura 3.11: Sequenza di stati indotta da P

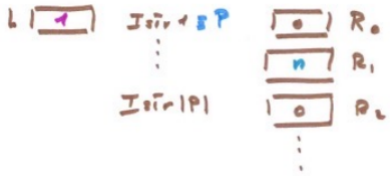


Figura 3.12: Situazione globale durante S_{init}

Figura 3.13: Situazione globale durante S_{final} (in caso di $HALT$)

Quindi la semantica del programma sarà:

$$\varphi_p(x) = \begin{cases} y \\ \perp \end{cases}$$

Dove $\perp$ mi indica una sequenza infinita di stati in loop.

Ingredienti della definizione formale della semantica

- **Stato** di una macchina RAM, è lo stato dei registri di quella macchina. Matematicamente modella to come una funzione che mi restituisce il contenuto del risultato quando la macchine è nello stato S

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

$$STATI = \mathbb{N}^{\{L,R_i\}} = \{\text{insieme di tutti i possibili stati della macchina}\}$$

$$S(R_j) = \textit{contenuto di } R_j \text{ durante lo stato } S$$

- **Stato finale** della macchina RAM, uno stato tale per cui:

$$S(L) = 0 \implies HALT$$

- **Dato**, rappresentato da $\mathbb{N}$, questo perchè sappiamo che $\mathbb{N} \sim DATI$.

- **Inizializzazione**, dato il dato n prepara la macchina nell ostato iniziale con input n. Quindi sarà una funzione in questa creerà uno stato S_{init} a partire da un input n:

$$in : DATI \to STATI \text{ t.c. } in(n) = S_{init}$$

  Essa imposterà il contenuto del primo registro ad 1, e tutti gli altri a 0, e poi il caricherà nel program counter l'indirizzo della prima istruzione (contenuto del registro).

$$S_{init}(R_i) = \begin{cases} n & \text{se } i = 1 \\ 0 & \text{se } i = 0 \end{cases}$$

$$S_{init}(L) = 1$$

- **Programmi**, insieme dei programmi RAM $PROG = \{\text{programmi RAM}\}$, un singolo programma $P \in PROG$, tale per cui la sua cardinalità indica il numero di istruzioni contenute $|P| = \#istr$.

- **Esecuzione**, essa mi specifica la dinamica del programma che mi fa passare da uno stato al successivo. Questo è possibile con utilizzando la *funzione stato prossimo* δ. Tale funzione mi permette di spostarmi dallo stato attuale S a quello successivo S'.

$$\delta : STATI \times PROG \to STATI_{\perp}$$

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

  Lo stato prossimo dipenderà dall'istruzione che in quel momento deve essere eseguita, e per conoscere tale istruzione devo andare a vedere il contenuto del program counter allo stato attuale $S(L)$ (quindi lo stato prossimo dipenderà da $S(L)$).
  Come è definito lo stato prossimo in funzione dello stato attuale?

  1. Se $S(L) = 0$ (ovvero in terminazione), allora lo stato prossimo non è definito $S' = \perp$.
  2. Se $S(L) > |P|$, significa che abbiamo superato l'ultima istruzione e che il program counter è stato incrementato diventando così più grande del numero di istruzioni del programma. Quello che si fa è:

$$S'(L) = 0 \text{ HALT}$$

$$\forall i : S'(R_i) = S(R_i)$$

3. Se $1 \leq S(L) \leq |P|$, questo è il caso comune e considerando la $S(L)$-esima istruzione:
   - $R_k \leftarrow R_k \pm 1$, definite come:

     $S'(R_k) = S(R_k) \pm 1$
     $S'(L) = S(L) + 1$
     $S'(R_i) = S(R_i)$ con $i \neq k$

   - IF $R_k = 0$ THEN GOTO m, definita come:

     $S'(R_i) = S(R_i)$
     if $S(R_k) == 0$ then
         $S'(L) = m$
     else
         $S'(L) = S(L) + 1$

Esecuzione del programma $P \in PROG$
Ora posso definire la sequenza di computazione di un programma $P \in PROG$ su un input $n \in \mathbb{N}$. Una computazione è una sequenza di stati indotta dalla dinamica:

$$in(n) = S_0, S_1, ..., S_i, S_{i+1}, ...$$

Eventualmente la sequenza può essere infinita (loop), o terminare se un S_m raggiunge un $S_m(L) = 0$ (halt).

Dobbiamo vincolare la sequenza di stati alla funzione $\delta(S, P)$, e dire che per ogni $\delta(S_i, P) = S_{i+1}$.

Quindi, la semantica di P, è definita come $\varphi_P : \mathbb{N} \to \mathbb{N}_\perp$:

$$\varphi_P(n) = \begin{cases} y & \text{se la computazione del programma termina in} \\ & \text{con S(L)=0 ed il contenuto di } S_m(R_0) = y \\ \perp & \text{se la computazione del programma va in loop} \end{cases}$$

Ora posso definire formalmente la potenza computazionale del sistema RAM:

$$F(RAM) = \{f \in \mathbb{N}^{\mathbb{N}} : \exists P \in PROG, \varphi_P = f\} = \{\varphi_P : P \in PROG\} \subsetneq \mathbb{N}_\perp^{\mathbb{N}}$$

Ovvero tutte le funzioni per cui esiste un programma P tale per cui $\varphi_P = f$ (o φ_P al variare di P).

Alcuni programmi RAM e le loro funzioni che vengono calcolate

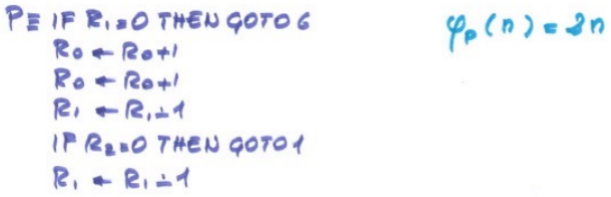


Figura 3.14: n

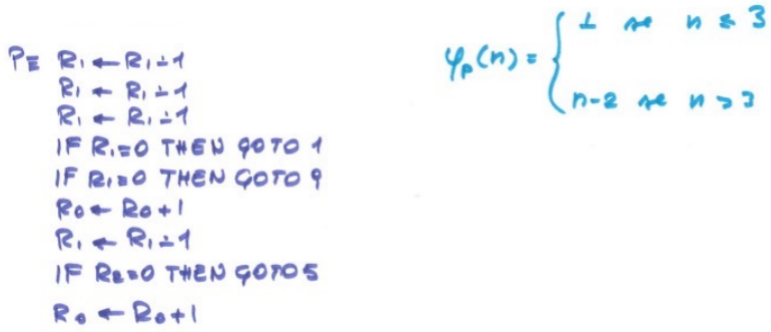


Figura 3.15: Funzione più complicata

Questi programmi possono essere spiegati formalmente con gli strumenti matematici precedentemente forniti, per esempio con il primo programma:

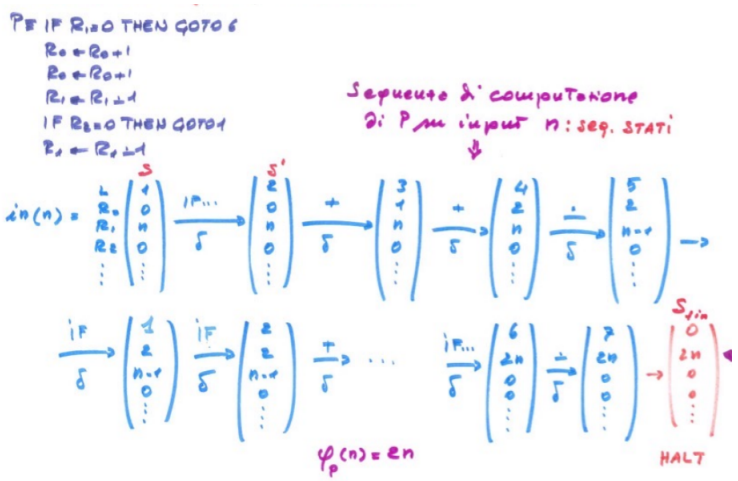


Figura 3.16: Rappresentazione formale di $\varphi_P(n) = 2n$

Questo è il modo corretto di procedere, e questo modo permette di definire formalmente la potenza computazionale di una macchina RAM, non lasciando nulla al caso.

Alcune considerazioni

- $F(RAM)$ *conterrà funzioni più complesse o solo quelle banali?* In realtà si posso fare delle funzioni più complesse, questo comunque è un primo tentativo di fare qualcosa di ragionevole che mi permette la completa formalizzazione.
- Indubbiamente la semplicità di questo sistema di calcolo ci permette l'estrema formalizzazione, tale che vedremo che sarà possibile dimostrare $PROG \sim \mathbb{N}$

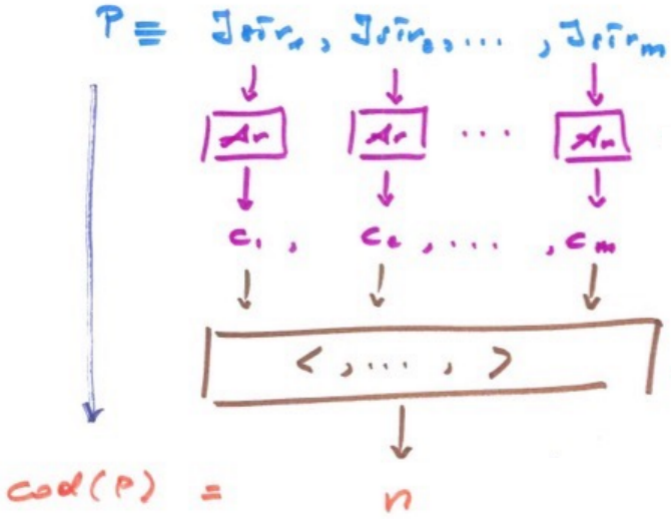
3.9.4 Dimostrazione $PROG \sim \mathbb{N}$

Dato un programma RAM vogliamo associare un numero a tale programma in maniera che partendo quel numero sia possibile tornare al sorgente.

Ovvero vogliamo dimostrare che anche i programmi possono essere in corrispondenza biunivoca con i numeri naturali (utilizzeremo la semplicità della macchina RAM).

$$P \equiv Istr_1, Istr_2, ..., Istr_m$$

Il primo passo consiste nell'aritmetizzazione dell'istruzione, ovvero trasformare una lista di istruzioni in una lista di codici numerici. Successivamente utilizzando la funzione lista di Cantor possiamo trasformare questo elenco di codici in un singolo numero n.



Sappiamo che la funzione lista di Cantor è invertibile quindi possiamo ricostruire le codifiche associate alle istruzioni. Ora se l'operazione di aritmetizzazione del sorgente è invertibile, allora ci sarà possibile risalire al sorgente partendo da n.

In generale un procedimento che fa corrispondere ad una qualsiasi struttura matematica un numero, si chiama operazione di **aritmetizzazione** o **Godelizzazione**.

Il nostro scopo è quello di aritmetizzare ogni istruzione.

Aritmetizzare biunivocamente le istruzioni RAM

$$Ar : Istr \to \mathbb{N} \text{ e } Ar^{-1} : \mathbb{N} \to Istr \text{ t.c. } Ar(Istr) = n \Leftrightarrow Ar^{-1}(n) = Istr$$

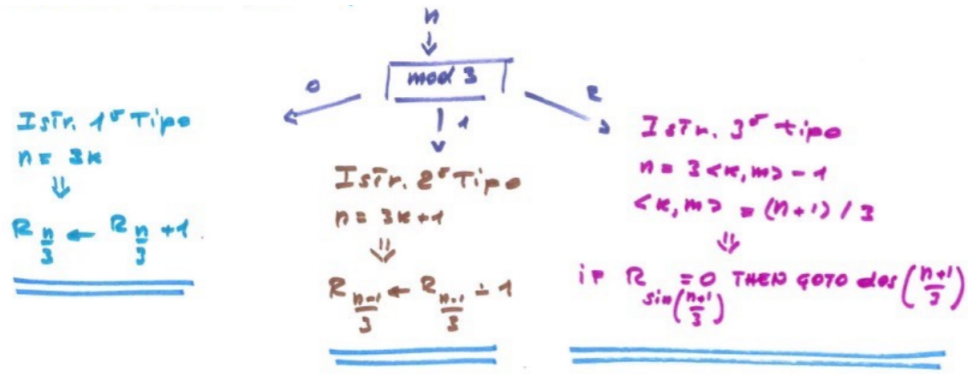
Nel caso di un programma RAM significa aritmetizzare tre tipi di istruzioni.

$$Ar(R_n \leftarrow R_n + 1) = 3k$$

$$Ar(R_n \leftarrow R_n - 1) = 3k + 1$$

$$Ar(\text{If } R_k = 0 \text{ then goto } m) = 3\langle k, m\rangle - 1$$

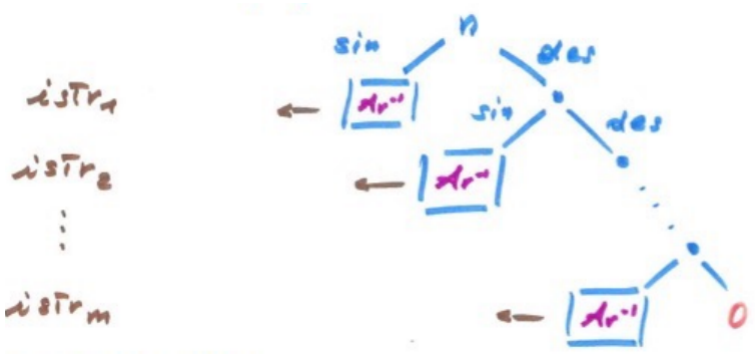
Come è fatta Ar^{-1}? La funzione di decodifica è biunivoca ? La funzione risulta sia iniettiva che suriettiva, l'applicazione di questa funzione è permessa attraverso l'operatore di modulo per 3.



Ricapitolando, il passaggio da programmi a numeri è molto semplice

$$cod(P) = \langle Ar(Istr_1), ..., Ar(Istr_m)\rangle$$

mentre il passaggio da numeri a programmi, parto da n e trovo la parte sinistra e la parte destra. Vediamo passo per passo:



Al primo passo dalla parte sinistra trovo il codice della prima istruzione (aritmetizzato), quindi nel caso in cui io voglia il sorgente del programma quello che devo fare è applicare Ar^{-1} sulla parte sinistra. Questo finché non si incontra il terminatore sul figlio destro.
Se io volessi sempre scompattare il programma P e dato n io volessi sapere il numero di istruzioni del programma $|P|$, che cosa devo calcolare?

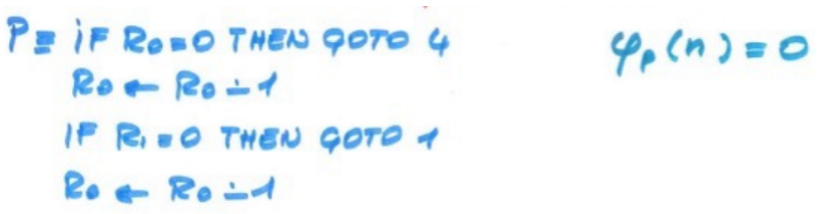
$$|P| = length(cod(P))$$

Abbiamo quindi dimostrato in maniera inequivocabile il secondo tassello, ovvero che i programmi sono equinumerosi rispetto ai numeri naturali.

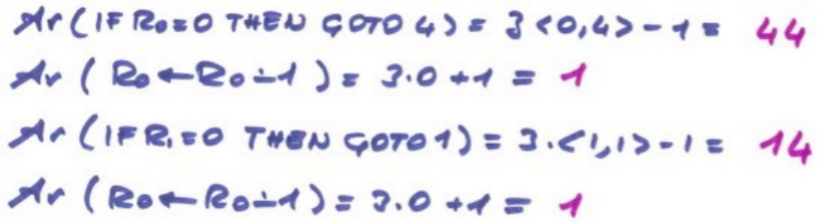
$$PROG \sim \mathbb{N}$$

Adesso che abbiamo acquisito gli strumenti, proviamo a vedere ad occhio qualche esempio.

Primo programma RAM



La corrispettiva aritmetizzazione delle istruzioni :



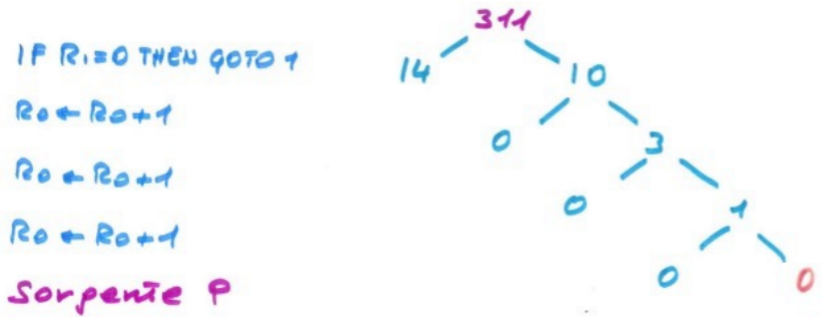
$$cod(P) = \langle 44, \langle 1, \langle 14, \langle 1, 0\rangle\rangle\rangle\rangle = 50556496$$

Questo non è un ottimo modo per compattare i programmi ram, poiché cresce esponenzialmente rispetto al programma. Tuttavia questo ha scopi didattici.

$$\varphi_{50556496}(n) = 0$$

Sorgente RAM da un numero
Il numero è il 311, ha come parte sinistra 14 e come parte destra 10. Facendo il modulo 3 del figlio sinistro otteniamo otteniamo 2, quindi si tratta di un istruzione condizionale, in questo caso troviamo che il valore dell coppia di Cantor $\langle k, m\rangle = 5$. Quindi dobbiamo trovare il figlio sinistro e destri di 5, che sono entrambi 1. Quindi la prima istruzione sarà:

$$\text{If } R_1 = 0 \text{ then goto } 1$$



Quale é la semantica di questo programma ? (ovvero $\varphi_P(n)$)

$$\varphi_P(n) = \varphi_{311}(n) = \begin{cases} \perp & \text{se } n = 0 \\ 3 & \text{se } n > 0 \end{cases}$$

Se l'input n è 0 vado in loop, poiché continuo a rieseguire la stessa istruzione, altrimenti parto dalla terza istruzione.

Riflessioni su $PROG \sim \mathbb{N}$

- Abbiamo dimostrato che i numeri sono un linguaggio di programmazione, questo perché un numero può rappresentare un linguaggio.
- $F(RAM) = \{\varphi_P : P \in PROG\}$, la potenzia computazionale del sistema di programmazione RAM, adesso la posso scrivere come $F(RAM) = \{\varphi_i\}_{i\in\mathbb{N}}$ questo mostra ina maniera inequivocabile che la potenza computazionale della macchina RAM è enumerabile.
- Per il sistema RAM si ha rigorosamente che $F(RAM) \sim \mathbb{N} \nsim \mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$,
- Questa è una prima idea di calcolabilità, esploriamo un sistema di calcolo $\mathcal{C}$ più complesso e vediamo la sua potenza computazionale di questo, e vediamo se è più complessa. Questo va fatto è onestamente dire che ciò che è calcolabile sia $F(RAM)$ è troppo restrittivo.
- Possiamo avere che questo sistema di clacolo più avanzato ampli l'insieme $F(RAM)$.

3.9.5 Il sistema di calcolo While

Il linguaggio è basato su un linguaggio moderno, contrapposto all'assembly della macchina RAM (quindi anni 50'), adesso ci troviamo nell'utilizzo di un linguaggio strutturato.

Hardware:

- **Memoria** costituita da 21 registri. Siccome parliamo di linguaggio strutturato non si parla più con il termine registri ma ci si riferirà con *variabili*.

  $$x_0, x_1, ..., x_{20}$$

  . La variabile x_1 ci si troverà l'input del programma, mentre l'output si troverà su x_0.
- Program counter non presente, poiché si parla di linguaggio strutturato, ed in questo le istruzioni vengono eseguite una dopo l'altra (punti di inizio ciclo e finale sono ben definiti, ed il *goto* non è presente).

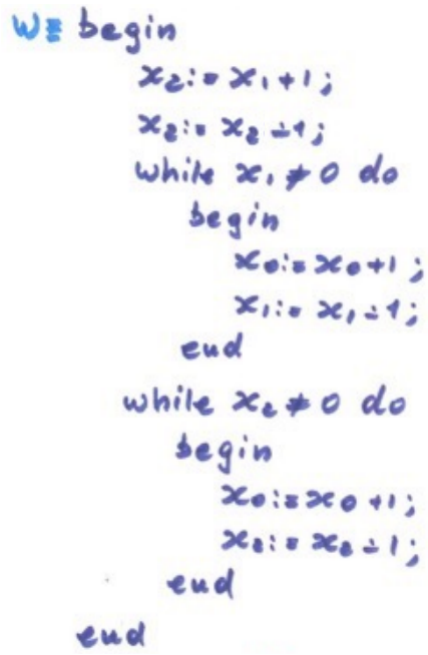
20 variabili potrebbero sembrare poche, in realtà la numerosità dei dati non è un problema poiché abbiamo primitive che mi permettono di condensare (coppia di Cantor) un infinità di variabili.
Il linguaggio while ha una sintassi induttiva (dove i costrutti dei linguaggi sono definiti su delle basi semplici, i mattoni, e man mano costruisco istruzioni più complesse):

- Comando di **assegnamento**: $x_k := 0 x_k := x_j + 1 x_k := x_j - 1$
- Comando **while**: while $x_k \neq 0$ do G, dove G ovvero il corpo del loop, può essere un comando di assegnamento, while o composto.

- Comando **composto**: begin $C_1; C_2; ...; C_m;$ end dove la sequenza è composta da comandi che possono essere di assegnamento, while o composto.

Come si può vedere sono strutture che sono in grado di nascondere altre strutture, di per sè un programma while è un comando composto.
Indicheremo com $W - PROG = \{\text{programmi WHILE}\}$ come l'insieme dei programmi WHILE, dove ciascuno è un programma costruito induttivamente.

Esempio di programma WHILE



Indicheremo con ψ_w le semantiche dei programmi WHILE (in questo caso il programma w).

$$\psi_w : \mathbb{N} \to \mathbb{N}_\perp$$
$$\psi_w(x) = 2x$$

Le prime due istruzioni sono messe perché non è possibile effettuare una copia diretta, non stiamo utilizzando il linguaggio di programmazione RAM.

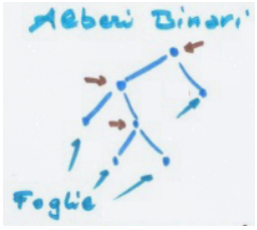
Il parsing del programma ne rivela la struttura induttiva, W-PROG è un insieme definito induttivamente (ovvero partendo dalla base ed utilizzando dei passi induttivi): per dimostrare una proprietà P su W-PROG:

1. Dimostro che P vale sui comandi base (passo base).
2. Suppongo vero quella proprietà P sui comandi C base, e poi dimostro che vale sul comando più complesso while $x_n \neq 0$ do C (passo induttivo).

3. Suppongo vera P su $C_1, ..., C_m$ e la dimostro vera su begin $C_1; ...; C_m$ end

Quando abbiamo di fronte una struttura induttiva il modo migliore per dimostrare che una certa proprietà valaga su tutti gli elementi della struttura dobbiamo farlo per induzione (in questo caso si parla di **induzione strutturale**).

Esempio
L'insieme degli alberi binari è un insieme definito induttivamente. Sappiamo che è un DAG.



Possiamo definire l'insieme degli alberi binari come l'insieme di tutti gli oggetti composti come mostrato in figura 3.9.5. Oppure, possiamo definirli induttivamente

- Base: Un nodo solo è considerabile un albero binario (con una singola foglia).
- Induzione: Se T_1 e T_2 sono alberi binari anche

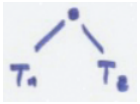


Figura 3.17: Binary Tree (indicati i nodi interni con le frecce marroni)

- Nient'altro è un albero binario.

Allora vogliamo dimostrare pre induzione la proprietà P, tale che:

$P \equiv$ "Su ogni albero binario, il numero dei nodi interni è minore di uno rispetto a quello delle foglie."

Allora dimostriamola per induzione:

- Base: *è vero che se ho un solo nodo la proprietà P è vera?* Si poiché nodi interni non sono presenti.

- Induzione: supponiamo vera P su T_1, T_2
  - T_1: foglie f_1 foglie e $f_1 - 1$ nodi interni.
  - T_2: foglie f_2 foglie e $f_2 - 1$ nodi interni.

  Per esempio, prendiamo questo albero:

  notiamo che il numero di foglie è pari a $f_1 + f_2$. Adesso la domanda fondamentale è la seguente *quanti nodi interni ha questo albero?* Il numero di nodi interni è pari alla somma tra i nodi interni di T_1 con quelli di T_2. Ma questo riutilizzando i mattoncini basilari non è altro che

$$(f_1 - 1) + (f_2 - 1) + 1 = f_1 + f_2 - 1$$

  P risulta dimostrata per induzione su tutta la struttura.

Esempio 2
Voglio definire una funzione $depth : \tau \to \mathbb{N}$ su un albero binario che dato un qualsiasi albero binario mi restituisce la **profondità**: ovvero, la lunghezza del cammino più lungo dalla radice ad una foglia dell'albero.

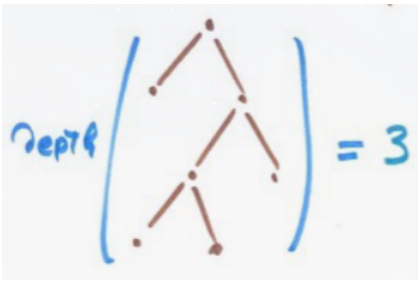
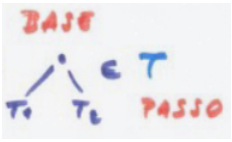


Figura 3.18: Definizione grafica

Questa è una possibile definizione, oppure potremmo definirla induttivamente:

1. Base: Dato un BT costituito da una sola radice darà $depth = 0$
2. Induzione: Dato questo albero binario



si prende la profondità maggiore due due sotto alberi più uno, ovvero il nuovo livello $depth = 1 + max\{depth(T_1), depth(T_2)\}$

3. Nient'altro è in τ

Esecuzione su una macchina WHILE
Considerando un input n

- La prima è una fase di inizializzazione, quindi si avvia la macchina con il nostro programma w, dove tutte i registri della variabili sono inizializzati a 0, tranne per il registro in input x_1 che contiene n.
- Si comincia ad eseguire il programma w, sequenzialmente sulle istruzioni, in questo caso non c'è assolutamente bisogno di program counter (motivi già spiegati).
- Possiamo avere due casi, o l'esecuzione effettivamente termina o si incappa in un loop (while true). Nel primo caso l'output lo andiamo a leggere nella variabile x_0 (se HALT), altrimenti diciamo che è indefinito $\perp$. Definendo così la semantica del programma w, $\psi_w(n) = cont(x_0)/\perp$.

Vogliamo essere precisi anche in questo caso, ed espandere il discorso della sequenza di istruzioni durante l'esecuzione di un programma WHILE (introducendo anche qui il concetto di stato).

3.9.6 Definizione formale di semantica di un programma WHILE

- **Stato**, una foto dove compare in maniera completa tutto ciò che accade sulla macchina in quel dato istante. Rispetto alla macchina RAM lo stato viene definito in maniera differente, dal punto di vista matematico uno stato è una tupla di 21 elementi $(c_0, ..., c_{20})$. Detto ciò, l'insieme di tutti i possibili stati $w - stati$ è composto da $\mathbb{N}^{21}$ ovvero tutte le possibili 21-tuple (tuple di lunghezza 21) infinite.
- **Dati**, sono l'insieme dei numeri naturali.
- **Inizializzazione**, sta volta la funzione può essere modellata su 21 elementi, la funzione prende un numero e restituisce uno stato

$$w - in : \mathbb{N} \to \mathbb{N}^{21} \text{ con } w - in(n) = (0, n, 0, ..., 0)$$

- **Semantica operazionale**

$$[]() : w - com \times w - stati \to w - stati_{\perp}$$

Dato un comando while C e uno stato $\underline{x}$, allora

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

sarà la funzione stato prossimo dove $\underline{y}$ è lo stato prossimo di $\underline{x}$ a seguito dall'esecuzione del comando C. Ovviamente possiamo definire induttivamente $[C](\underline{x})$ sulla struttura induttiva del comando C.

3.9.7 Definizione induttiva della semantica while

- Base: gli **assegnamenti**

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

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

- Passo:
  - Comando **composto**

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

induttivamente conosco la semantica di ciascuno di questi programmi, come essi agiscono. Allora posso definire la semantica di questo comando come:

$$[C_m](\dots([C_2]([C_1](\underline{x})))\dots) = \underline{y} = [C_1]$$

Quindi l'applicazione del primo comando (che provoca cambiamento di stato) sullo stato iniziale, seguita iterativamente dall'applicazione degli m comandi sugli stati risultanti.

  - Comando **while**

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

, anche qui conosco per ipotesi induttiva la semantica del comando C.

$$[C](\dots([C]([C](\underline{x})))\dots) = \underline{y} = [C_1] \text{ e ... e } [C_m](\underline{x})$$

Il numero di volte che applicherò il comando C quante volte serve per azzerare il contenuto di x_k

$$= \begin{cases} [C]^e(\underline{x}) & \text{con } \mu t \text{ k-esima componente di } [C]^{(t)}(\underline{k}) = 0 \\ \perp & \text{altrimenti} \end{cases}$$

Ovvero il più piccolo numero di volte che mi serve per azzerare la componente k-esima, altrimenti vado in un loop interminabile.

- Semantica di W e $W - PROG$: a questo punto so definire in maniera formale la semantica di un programma while, prendiamo il programma while w (il quale è un comando composto).

  Quindi è la semantica del comando composto che rappresenta w, quindi non è altro che preparare la macchina su input x.

  $$\Psi_w(x) = Pro(0, [w](w - in(x)))$$

  Ovvero la proiezione 0-esima dell'esecuzione del comando composto di w dato uno stato finale x (avviato da uno stato iniziale), significa che se il programma termina devo pescare il contenuto di x_0 (la proiezione di uno stato finale di un programma w). Nel caso in cui non avvenisse l'HALT allora avrò una semantica indefinita $\perp$.

Esempio di semantica di un piccolo programma while

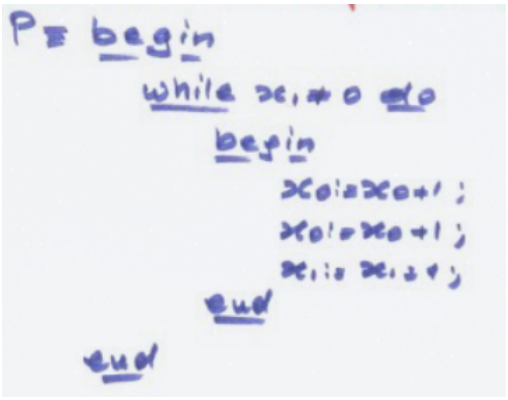


Figura 3.19: Programma che calcola $2n$

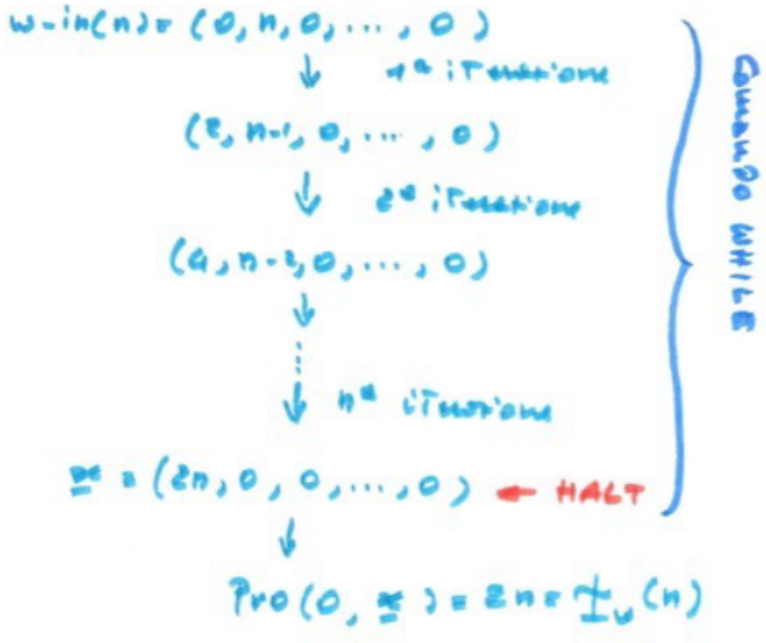


Figura 3.20: Deduzione della semantica Ψ_w

Potenza computazionale del sistema while

$F(WHILE) = \{f \in \mathbb{N}_{\perp}^{\mathbb{N}} : \exists w \in w - PROG, f = \Psi_w\} = \{\Psi_w : w \in w - PROG\}$

Definiamo la potenza computazionale di una macchina while come l'insieme di tutte le funzioni per cui esiste un programma while per cui si può calcolare quella funzione.Naturalmente la domanda è: *che relazione esiste tra* $F(RAM)$ *e* $F(WHILE)$*?*, quindi che cosa è calcolabile di questi due sistemi? $\{\varphi_P : P \in PROG\}$? Magari la macchina WHILE mi fornisce più potenza della macchina RAM? L'unico modo di scoprirlo è confrontando le due idee. Quali situazioni si possono verificare:

1. Una situazione dove $F(RAM) \nsubseteq F(WHILE)$ (inclusione propria) ovvero, che ci sono funzioni calcolabili con i programmi WHILE che non sono calcolabili sulle macchine RAM.

2. Le due idee di calcolabilità non sono confrontabili, ovvero un intersezione tra i due insiemi dove magari alcune funzioni sono calcolabili da entrambe le architetture, oppure sono due insiemi totalmente disgiunti. In entrambi i casi siamo preoccupati, perché l'idea di calcolabilità è dipendente dal modello di calcolo adottato.

3. Una situazione dove $F(WHILE) \nsubseteq F(RAM)$ questo sarebbe sorprendente, la macchina sofisticata non riesce a calcolare delle funzioni più semplici.

4. Una situazione dove $F(WHILE) = F(RAM)$, ovvero due cose filosoficamente diverse eppure entrambe riescono a calcolare la stessa classe di funzioni, questo sarebbe molto interessante, perché allora vuol dire che calcolabile non dipende dalla tecnologia-