CAPITOLO 3.

MACCHINE A MEMORIA INFINITA

1. La computabilità per i matematici: terne di Peano e ricorsione

Abbiamo visto come il tentativo di definire la nozione di “macchina che effettua calcoli” tramite la nozione di automa finito sia fallito. Infatti anche funzioni che a noi appaiono sicuramente computabili, come il prodotto o la radice, non possono essere calcolate tramite un automa finito. Pertanto, per potere costruire una teoria della computabilità, appare necessario ampliare in maniera adeguata la nozione di "macchina” in modo da ottenere macchine in grado di calcolare ogni funzione che a noi appare in qualche modo calcolabile. Faremo questo in seguito, prima però cerchiamo di analizzare che cosa usualmente i matematici intendono con l’espressione “calcolabile”. Per fare questo cominciamo con l’osservare che, come abbiamo visto con le varie forme di abachi, i primi calcoli che sono stati fatti dall’uomo sono stati calcoli (addizioni o moltiplicazioni) con sassolini, palline od altro. In ogni caso calcoli con oggetti capaci di rappresentare i numeri naturali. Allora in questo paragrafo esaminiamo come sia possibile definire i numeri naturali. Il punto di partenza è, come è noto, la nozione di terna di Peano.

Definizione 1.1. Diciamo che una struttura algebrica $(S, succ, z_0)$ con $succ$: $S \to S$ operazione 1-aria e $z_0 \in S$, è una *terna di Peano* se sono verificati i seguenti assiomi:

- ***P1*** $succ : S \to S$ è una funzione iniettiva chiamata *funzione successore*,
- ***P2*** $z_0 \notin succ(S)$, cioè z_0 non è il successivo di nessun elemento
- ***P3*** per ogni sottoinsieme D di S

  $z_0 \in D$ e $succ(D) \subseteq D \Rightarrow D = S$.

P3 prende il nome di *principio di induzione* e rende possibile un potente metodo dimostrativo. Infatti in base a tale proprietà è possibile dimostrare la seguente proposizione.

Proposizione 1.2. (Principio di induzione). Supponiamo che una proprietà P sia definita in una terna di Peano $(S,succ,z_0)$ e che:

a) P sia verificata da z_0

b) se P è verificata da x allora è verificata da $succ(x)$.

Allora possiamo asserire che P è verificata per ogni $x \in S$.

Dim. Sia D l’insieme degli elementi che verificano P, allora, per la proprietà *a*), D contiene z_0 inoltre, per la proprietà b), risulta che $succ(D) \subseteq D$. Pertanto in base all’assioma *P3*, tale insieme coincide con S.

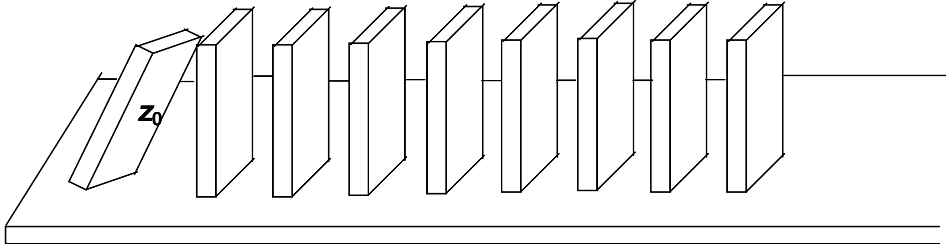
L’applicazione del principio di induzione può essere visualizzata al modo seguente. Consideriamo la seguente figura in cui i pezzi del gioco domino sono poggiati su di un tavolo (infinito) uno dopo l’altro. Indichiamo il primo pezzo della fila con z_0. Vale la regola che se un pezzo cade (a destra) allora il pezzo successivo cade.

$$\forall x Cade(x) \to Cade(succ(x)).$$

Poi supponiamo che si verifichi

$$Cade(z_0)$$

Allora è evidente che vale $\forall x Cade(x)$, cioè che tutti i pezzi cadono.



Esempi di terne di Peano. Non è difficile trovare esempi di terne di Peano. In effetti tutti i sistemi utilizzati dall' uomo per "contare" sono esempi di terne di Peano.

Terne di Peano e tacche di legno: Se per contare le pecore di un gregge gli uomini primitivi utilizzavano delle tacche su di un pezzo di legno, allora l'insieme delle possibili tacche su un pezzo di legno costituisce un esempio di terna di Peano. In tale caso un pezzo di legno senza tacche rappresenta il primo elemento, l'operazione di aggiungere una tacca è l'operazione successore.
Terne di Peano e parole: Un esempio simile si ottiene considerando l'insieme S delle "parole" del tipo 0$$$$$, cioè parole che iniziano con 0 e che sono seguite da un certo numero di $. Inoltre per ogni parola x in S, possiamo porre $succ(x) = x\$$. Allora è subito visto che $(S,succ,0)$ è una terna di Peano.
Terne di Peano e scatole con biglie: Possiamo considerare anche l'insieme i cui elementi sono barattoli contenenti biglie di vetro. Un barattolo vuoto corrisponde allo zero. L'operazione di aggiungere una biglia alle biglie di un barattolo corrisponde all'operazione di successivo.

Naturalmente tali esempi non sono di tipo matematico e se volessimo essere più rigorosi dovremmo procedere a qualche forma di "idealizzazione". Questo significa che dovremmo immaginare che esistano infiniti possibili pezzi di legno, almeno uno per ogni possibile sequenza di tacche. Inoltre se due pezzi di legno hanno tre tacche, allora devono essere considerati equivalenti, cioè rappresentativi di un solo oggetto (il numero 3).

Il seguente teorema dimostra che tutte le terne di Peano sono isomorfe tra loro e che quindi ci si può riferire indifferentemente ad una oppure ad un altra.

Teorema 1.3. Tutte le terne di Peano sono isomorfe tra loro.

Dim. Siano (S, s, z_0) e $(\underline{S}, \underline{s}, \underline{z}_0)$ due terne di Peano e consideriamo la funzione $f : S \rightarrow \underline{S}$ definita ponendo

$$f(z_0) = \underline{z}_0 \quad ; \quad f(s(x)) = \underline{s}(f(x)).$$

Allora per il principio di induzione f è definita per ogni $x \in S$. Per costruzione f risulta essere un omomorfismo. Per provare che f è suriettiva dobbiamo consideriamo il suo condominio $f(S)$. E' evidente che $f(S)$ contiene $\underline{z}_0$ e che se contiene un elemento $f(x)$ contiene anche il suo successivo. Quindi $f(S) = \underline{S}$. Per provare che f è iniettiva poniamo $X = \{x \in S : f(x') = f(x) \Rightarrow x' = x\}$. Allora $z_0 \in X$, infatti se $f(x') = f(z_0) = \underline{z}_0$ e con $x' \neq z_0$ allora detto a tale che $s(a) = x'$ avremmo che $\underline{z}_0 = f(x') = f(s(a)) = \underline{s}(f(a))$. Ciò è assurdo poiché $\underline{z}_0$ non può essere successore di nessun elemento in $\underline{S}$. Proviamo ora l'implicazione $x \in X \Rightarrow s(x) \in X$ e sia x' tale che $f(x') = f(s(x)) = \underline{s}(f(x))$. Se x' fosse uguale a z_0 Se $x' = s(x'')$ allora sarebbe $\underline{s}(f(x'')) = f(s(x'')) = f(x') = \underline{s}(f(x))$ e quindi, per la iniettività di $\underline{s}$, $f(x'') = f(x)$. Per ipotesi di induzione ciò comporta che $x'' = x$ e quindi che $x' = s(x'') = s(x)$. Ciò prova che $s(x) \in X$.

Nel seguito indicheremo con *N* l'insieme degli elementi di una terna di Peano. Inoltre l'elemento z_0 viene indicato con 0, *succ*(0) viene denotato con 1, *succ*(1) viene denotato con 2 e così via. Inoltre al posto di scrivere *succ*(*x*) scriviamo *x*+1 anche se l'operazione di addizione deve essere ancora definita.

2. Le funzioni primitive ricorsive.

Data una terna di Peano *N*, passiamo ora a definire la classe delle funzioni computabili in *N*. Per fare ciò procederemo al modo seguente:

i) Faremo un elenco di alcune funzioni che tutti sono disposti a ritenere computabili (come l'addizione, la moltiplicazione ed altro).

ii) Indicheremo alcuni procedimenti che permettono di costruire nuove funzioni computabili a partire da date funzioni computabili (potrebbero essere la composizione o l'addizione di due funzioni e così via).

iii) Chiameremo poi "computabili" tutte le funzioni che si possono ottenere a partire dalle funzioni date in i) tramite applicazioni reiterate dei procedimenti indicati in ii).

Nell'elenco suggerito da i) naturalmente mettiamo la funzione *succ* : $N \to N$. E' inutile mettere esplicitamente l'addizione e la moltiplicazione. Infatti in ogni terna di Peano tali operazioni, che denotiamo con *som* e *pro* sono definibili al modo seguente:

$$som(x,x_0) = x \; ; \; som(x,succ(y)) = succ(som(x,y)). \qquad (2.1)$$

$$pro(x, x_0) = x_0 \quad ; \quad pro(x, succ(y)) = som(pro(x,y),x). \qquad (2.2)$$

Un tale modo di definire le funzioni viene detto *per ricorsione* e verrà esaminato più in avanti. Possono naturalmente essere utilizzate le notazioni più familiari,

$$som(x,0) = x \; ; \; som(x,y+1) = som(x,y)+1. \qquad (2.3)$$

$$pro(x,0) = 0 \quad ; \quad pro(x,y+1) = pro(x,y)+x. \qquad (2.4)$$

Proposizione 2.1. Le operazioni *som* e *pro* (addizione e moltiplicazione) sono definite per ogni valore degli input *x* ed *y*.

Dim. Fissiamo *x* e consideriamo l'insieme $D = \{y \in N : som(x,y) \text{ è definita}\}$. Allora la prima equazione in (2.1) comporta che $x_0 \in D$. La seconda equazione che $y \in D \Rightarrow succ(y) \in D$. Applicando il principio di induzione possiamo concludere che $D = N$ e quindi che *som* è definita per ogni *x* ed ogni *y*. Lo stesso ragionamento può essere fatto per il prodotto.

Tale proposizione mostra che ogni terna di Peano è adatta a tutti gli effetti a rappresentare i numeri interi. Pertanto se un calcolatore vuole fare calcoli numerici allora:
1. deve contenere in qualche modo una terna di Peano
2. deve ammettere la ricorsione.

Da notare che se provassimo ad estendere la definizione di addizione data da (2.1) ai numeri reali ci sarebbero subito delle difficoltà. Questo perché nell'insieme dei numeri reali non vale il principio di induzione. Ad esempio il tentativo di calcolare *som*(1,2.5) condurrebbe a calcolare *som*(1,2.5), e quindi *som*(1,0.5) e quindi *som*(1,-0.5) e poi *som*(1,-1.5) e così all'infinito.

La ricorsione permette di definire molte altre funzioni. Ad esempio la funzione potenza. Più precisamente, se indichiamo *pot*(*a*,*n*) la potenza in base *a* ed esponente *n*, allora tale funzione è definita per ricorsione tramite le equazioni

$$pot(a,0) = 1 \quad ; \quad pot(a,n+1) = pot(a,n) \times a.$$

In modo analogo può essere definita la funzione fattoriale. La definizione generale di funzione definita per ricorsione è la seguente.

Definizione 2.2. Diciamo che $f : N^n \to N$ è ottenuta per *ricorsione* dalle due funzioni $g(x_1,...,x_{n-1})$ e $h(x_1,...,x_{n-1},y,z)$ se valgono le seguenti equazioni:

$f(x_1,...,x_{n-1},0) = g(x_1,...,x_{n-1})$;

$f(x_1,...,x_{n-1}, y+1) = h(x_1,...,x_{n-1}, y, f(x_1,...,x_{n-1},y))$.

Si assume che la funzione f sia definita in $x_1,...,x_{n-1},y+1$ solo se
- g è definita in $x_1,..., x_{n-1}$
- $f(x_1,...,x_{n-1},y)$ è definita in $x_1,..., x_{n-1},y$
- h è definita in $x_1,...,x_{n-1}, y, f(x_1,...,x_{n-1},y)$.

Ne segue che se $g(x_1,..., x_{n-1})$ e $h(x_1,...,x_{n-1}, y, z)$ sono totali, anche f è totale.

Consideriamo ora il modo come vengono definiti i polinomi, ad esempio il polinomio $xy+y$. Appare evidente che essi sono ottenuti "componendo" opportunamente le operazioni di prodotto e somma. Ad esempio il monomio x^2y è del tipo $pro(x^2,y)$, cioè, $pro(pro(x,x),y)$, il polinomio x^2y+2y non è altro che $som(pro(pro(x,x),y),pro(2,y))$. Un tale modo di definire nuove funzioni computabili prende il nome di *composizione*.

Definizione 2.3. Sia h una funzione ad m argomenti e siano $g_1,...,g_m$ funzioni ad n argomenti, diciamo che f è ottenuta per *composizione* da h e $g_1,...,g_m$ se risulta

$$f(x_1,...,x_n) = h(g_1(x_1,...,x_n),...,g_m(x_1,...,x_n))$$

Le considerazioni precedenti suggeriscono di procedere al modo seguente. Partiamo da pochissime funzioni in N che chiamiamo *funzioni-base*. Precisamente consideriamo le funzioni:

- *successore* $s : N \to N$, definita da $s(x) = x+1$
- *i-n-proiezione* $p_i^n : N^n \to N$, con $i \leq n$, definita da $p_i^n(x_1,...,x_n) = x_i$
- *funzione zero* $z : N \to N$, definita da $z(x) = 0$ per ogni x in N.

É indiscutibile che le funzioni-base sono calcolabili. Inoltre se definiamo una funzione per ricorsione o per composizione a partire da funzioni computabili allora quello che si ottiene è una funzione computabile.

Definizione 2.4. Una funzione si dice *primitiva ricorsiva* se si può ottenere a partire dalle funzioni-base tramite una (eventualmente reiterata) applicazione delle definizioni per composizione o ricorsione.

Proposizione 2.5. Le funzioni primitive ricorsive sono totali.

Dim.. Osserviamo che le funzioni-base sono totali e che la composizione di due funzioni totali è ancora una funzione totale. Per quanto riguarda la ricorsione, supponiamo che g ed h siano due funzioni totali e che f sia definita per ricorsione a partire da g ed h, cioè che

$f(x_1,...,x_{n-1},0) = g(x_1,...,x_{n-1})$;

$f(x_1,...,x_{n-1}, y+1) = h(x_1,...,x_{n-1}, y, f(x_1,...,x_{n-1},y))$.

Per provare che f è una funzione totale, poniamo

$$D = \{y \in N : f(x_1,...,x_{n-1},y) \text{ sia definita in } y \text{ per ogni } x_1,\dots,x_{n-1}\}.$$

Allora, essendo g totale, $0 \in D$. Inoltre da $y \in D$ segue che $y+1 \in D$. Allora per il principio di induzione $D = N$ e questo prova che fissati comunque $x_1,\dots,x_{n-1}$ e qualunque sia y, la funzione f è definita in $x_1,\dots,x_{n-1}, y$.

A prima vista sembrerebbe che la classe delle funzioni primitive ricorsive sia tanto piccola da non contenere nemmeno le più usuali funzioni computabili. Questa sensazione è errata perché la quasi totalità delle funzioni definite sui numeri naturali che si usano normalmente in matematica sono primitive ricorsive.

Teorema 2.6. L'addizione, la moltiplicazione la funzione potenza ed il fattoriale sono primitive ricorsive. Sono anche primitive ricorsive la funzione traslazione $f(x) = x+k$ e la funzione costante $f(x) = k$ con k costante prefissata.

Dim. Abbiamo già visto che l'addizione si definisce per ricorsione a partire dalla funzione-base *successore*, la moltiplicazione si definisce per ricorsione a partire dall'addizione, la potenza a partire dalla moltiplicazione. Se indichiamo con *fatt* la funzione fattoriale, allora tale funzione si definisce tramite le equazioni $fatt(0) = 1$; $fatt(y+1) = fatt(y)\cdot y$. La funzione $f(x) = x+k$ con k intero è primitiva ricorsiva poiché si può ottenere componendo k volte la funzione successore con se stessa. Ad esempio, per $k = 3$, $f(x) = s(s(s(x)))$. La funzione costante k che associa ad ogni x il numero k è primitiva ricorsiva perché si può ottenere applicando prima la funzione nulla e poi la funzione $x+k$.

Esempi. La funzione $f(x_1,x_2) = 2x_1\cdot(x_2+3)$ è primitiva ricorsiva poiché, posto $f(x) = y+3$, coincide con

$$pro(2x_1, x_2+3) = pro(som(x_1,x_1),f(x_2)) = pro(som(p_1^2(x_1,x_2), p_1^2(x_1,x_2)),f(p_2^2(x_1,x_2)).$$

La funzione $f(n)=1+2+3...+n$, è primitiva ricorsiva. Infatti f si definisce per ricorsione tramite le equazioni $f(1)=1$, $f(n+1)=f(n)+(n+1)$, cioè si definisce per ricorsione a partire dalla somma. Poiché la somma è primitiva ricorsiva possiamo concludere che anche f è primitiva ricorsiva.

Problemi. Provare :
- che la funzione che ad ogni n associa la somma dei primi n quadrati perfetti $1+2^2 \, . \, . \, . +n^2$ è primitiva ricorsiva
- che la funzione x^2 è primitiva ricorsiva.

3. Linguaggi equazionali e codifiche.

Nella maggior parte dei linguaggi di programmazione è possibile definire le funzioni primitive ricorsive usando direttamente le definizioni per composizione e per ricorsione. Chiameremo *equazionale* un linguaggio in cui questo aspetto e preponderante. Un tipico linguaggio equazionale è *Mathematica*. In un linguaggio equazionale si parte da un alfabeto finito (ad esempio l'insieme delle lettere che compaiono sulla tastiera del computer). In tale alfabeto si deve individuare:
1. Una parola per la definizione delle funzioni (usualmente la parola :=) e simboli "(", ")", "[" e "]" per le parentesi.
2. Parole da vedersi come "nomi per funzioni n-arie" (tra cui nomi per le funzioni-base).
3. Parole da vedersi come nomi per variabili (ad esempio le parole $x1$, $x2$,...)
4. Parole per le funzioni base.
5. *Numerali* cioè simboli che denotano numeri naturali.

Ad esempio, in *Mathematica* si accettano come numerali le rappresentazioni decimali dei numeri. Come variabili si accettano tutte le parole seguite dal trattino in basso _. Come nomi di funzione si accettano tutte le parole che iniziano con una minuscola e così via. Per le funzioni base si adottano parole che iniziano con una maiuscola.

Definizione 3.1. Dato un linguaggio funzionale ed un nome di funzione f, chiamiamo *definizione per composizione* di f una parola del tipo

$$f(x_1,...,x_n) := h(g_1(x_1,...,x_n),...,g_m(x_1,...,x_n))$$

dove f , h, $g_1,\dots,g_m$, sono nomi di funzioni e $x_1,\dots,x_n$ denotano variabili.

Chiamiamo *definizione per ricorsione* di f una parola del tipo

$$f(x_1,...,x_{n-1},0) := g(x_1,...,x_{n-1}) \;\; ; \; f(x_1,...,x_{n-1}, y+1) := h(x_1,...,x_{n-1}, y, f(x_1,...,x_{n-1},y)).$$

Definizione 3.2. Uno *schema di calcolo* per f è una successione di definizioni per ricorsione o per composizione la quale:
- termini con una definizione di f ;

- la *i*-esima definizione utilizza funzioni che sono state definite già in un livello precedente o che siano funzioni base.

Nel seguito ci riferiamo ad un linguaggio funzionale molto semplice in cui come funzioni base adottiamo il simbolo *s* per l'operatore di *successore*, inoltre per ogni coppia di interi *i* e *n* , con $i \leq n$, scriviamo p_i^n per denotare la *i-n-proiezione*. Infine adottiamo il simbolo *z* per la funzione costantemente uguale a zero.

Esempio: Ad esempio il seguente è uno schema di calcolo per la funzione $f(a,x,y) = a^{x+y}$

$som(x,0) := x$; $som(x,y+1) = succ(som(x,y))$.
$pro(x,0) = 0$; $pro(x,y+1) = som(pro(x,y),x)$.
$pot(a,0) = 1$; $pot(a,n+1) = pro(pot(a,n),a)$.
$f(a,x,y) = pot(a,som(x,y))$.

E' evidente che ad ogni schema di calcolo per *f* è possibile associare una ed una sola funzione definita in *N*. Uno schema di calcolo può essere visto come un programma scritto in un linguaggio in cui è permessa la ricorsione e la composizione. E' chiaro che una funzione *f* è ricorsiva primitiva se e solo se esiste uno schema di calcolo per *f*.
Per rispondere a tale domanda dimostriamo prima il seguente teorema.

Teorema 3.3. Esiste una codifica di tutti gli schemi di calcolo e quindi di tutte le funzioni ricorsive primitive.

Dim. Per dare un numero di codice a tutte le funzioni ricorsive primitive è sufficiente dare un numero di codice ad ogni schema di calcolo. Supponiamo di avere assegnato ad ogni lettera dell'alfabeto un "codice" costituito da una sequenza di 0 ed 1 che inizi con 1 ed in modo che tutte le sequenze abbiano la stessa lunghezza *p*. Ad esempio possiamo supporre che se l'alfabeto contiene le lettere *a*, *b*, *c* allora a tali lettere sia state assegnate rispettivamente le sequenze 1000, 1001, 1111 di lunghezza 4. Successivamente, dato uno schema di calcolo π si può procedere al modo seguente:
1. si sostituisce ad ogni lettera dell'alfabeto in π il rispettivo codice
2. si interpreta la sequenza di cifre ottenuta in questo modo come un numero intero *n*
3. si assegna allo schema di calcolo π il numero *n* come codice.
La corrispondenza che si definisce in tale modo assegna ad ogni schema di calcolo un numero intero. Da notare che tale corrispondenza è iniettiva ma non suriettiva.
Viceversa associamo ad ogni numero *n* uno schema di calcolo π_n tramite un processo di *decodifica* al modo seguente:
1. si scrive il numero *n* in base 2
2. si verifica che il risultato sia divisibile in blocchi di lunghezza *p* se questo non accade si va alla regola 5
3. si sostituisce, se possibile, ad ogni blocco l'elemento dell'alfabeto di cui è codice, se non è possibile si va a 5
4. si ottiene una parola: se tale parola denota uno schema di calcolo allora ad *n* viene associato tale schema altrimenti si va a 5
5. si associa, per convenzione, ad *n* un qualunque schema di calcolo, ad esempio lo schema relativo alla funzione *n*-aria costantemente uguale a zero $f(x_1,\ldots,x_n) := 0$.
In questo modo ad ogni intero *n* viene associato uno schema di calcolo π_n.

Definizione 3.4. Indichiamo con π_n lo schema di calcolo *n*-esimo e con f_n la funzione primitiva ricorsiva che viene calcolata da π_i.

4. Funzioni ricorsive

Avendo dato la definizione di funzione primitiva ricorsiva, si pone il problema se tale definizione sia adeguata, cioè se tutte le funzioni intuitivamente computabili siano primitive ricorsive. La codifica delle funzioni primitive ricorsive permette di dimostrare il seguente teorema:

Teorema 4.1. Esiste una funzione intuitivamente computabile che non è primitiva ricorsiva.

Dim. Vogliamo trovare una funzione f che, pur essendo intuitivamente computabile, non coincide con nessuna delle funzioni della successione $f_1, f_2, \ldots$. A tale scopo, per essere sicuro che f sia diversa da f_1, è sufficiente considerare una funzione f che sia diversa da f_1 in 1. Questo può essere ottenuto ad esempio ponendo $f(1) = f_1(1)+1$. Poi, per essere sicuri che f non coincida con f_2, possiamo fare in modo che in 2 assuma un valore diverso da $f_2(2)$. Ad esempio possiamo porre $f(2) = f_2(2)+1$. Appare allora naturale considerare la funzione f definita ponendo $f(n) = f_n(n)+1$. Tale funzione può essere computata dal seguente algoritmo:

i) dato un input n si decodifichi n in modo da ottenere lo schema di calcolo π_n corrispondente;

ii) si effettuino i calcoli indicati da π_n a partire dall'input n, pervenendo in tale modo al numero intero $f_n(n)$

iii) si aggiunga una unità.

Essendo tutte le funzioni f_n totali, e non essendovi ostacoli alla decodificazione, la funzione f è totale. Supponiamo per assurdo che f sia primitiva ricorsiva, allora esiste un indice j tale che $f = f_j$. Poiché f è ovunque definita, f sarà definita anche in j ed avremmo $f(j) = f_j(j)$ in contrasto con il fatto che, per definizione, $f(j) = f_j(j)+1$. □

Poiché il concetto di funzione primitiva ricorsiva non si è mostrato adeguato a rappresentare quello di funzione computabile, siamo indotti ad ampliare la classe delle funzioni primitive ricorsive. Faremo questo aggiungendo un nuovo modo di definire una funzione computabile da una data funzione computabile, modo che si basa su quello che nei linguaggi di programmazione viene visto come *ciclo condizionato*, cioè un blocco di istruzioni che si ripete fino a quando una data condizione non viene verificata.

Definizione 4.2. Sia $g : N^{n+1} \to N$ una funzione totale ad $n+1$ argomenti. Diciamo che la funzione $f : N^n \to N$ ad n argomenti è definita da g tramite il *μ-operatore* o per *minimalizzazione*, se il valore di f in $x_1,\ldots,x_n$ è dato da

$$f(x_1,\ldots,x_n) = Min\{y \in N \mid g(x_1,\ldots,x_n,y) = 1\}$$

se esiste un y tale che $g(x_1,\ldots,x_n,y) = 1$, mentre f non è definita in $x_1,\ldots,x_n$ se un tale y non esiste[1].

La funzione g può essere vista come la funzione caratteristica di una proprietà P che y può verificare o meno, proprietà che dipende anche dai parametri $x_1,\ldots,x_n$. Se P è una proprietà "decidibile", cioè tale che g è computabile, allora la minimalizzazione fornisce un procedimento di calcolo che consiste:

- nel controllare se P è verificata per $y = 0$, cioè se $g(x_1,\ldots,x_n,0) = 1$,
- nel caso P non sia verificata si incrementa y di una unità e quindi si passa a vedere se P è verificata da

[1] In tale definizione avremmo potuto anche scrivere $g(x_1,\ldots,x_n,y) = 0$ al posto di $g(x_1,\ldots,x_n,y) = 1$. La nozione di funzione ricorsiva non sarebbe cambiata. Infatti ogni funzione f ottenuta dalla funzione ricorsiva g tramite

$$f(x_1,\ldots,x_n) = Min\{y \in N \mid g(x_1,\ldots,x_n,y) = 0\}$$

sarebbe stata ugualmente ottenuta tramite

$$f(x_1,\ldots,x_n) = Min\{y \in N \mid g'(x_1,\ldots,x_n,y) = 1\}$$

essendo g' la funzione ricorsiva definita ponendo $g'(x_1,\ldots,x_n,y) = 1-g(x_1,\ldots,x_n,y)$.
Ad esempio, se $g(a_0,\ldots,a_n,x) = a_nx^n+\ldots+a_1x+ a_0$, allora, posto $k = 0$, la funzione $f(a_0,\ldots,a_n)$ associa ad ogni polinomio con coefficienti $a_n,\ldots,a_0$ la sua radice intera più piccola (se esiste). Naturalmente tale funzione è definita in $a_n,\ldots,a_0$ solo se $a_nx^n+\ldots+a_0$ ammette radici intere.

$y = 1, \ldots$

- si continua in questo modo: il primo valore y per cui P è verificata viene fornito come output
- se P non è verificata per nessun y questo processo di calcolo non termina mai ed f non è definita in $x_1,\ldots,x_n$.

Questo ultimo punto evidenzia il fatto che:

la minimalizzazione può definire una funzione parziale anche a partire da funzioni totali.

Il fatto che il procedimento di calcolo indicato dalla minimalizzazione può tradursi in una serie infinita di operazioni è l'analogo dei loop infiniti che si possono verificare nell'esecuzione di un programma.

Dal momento che si possono presentare funzioni non ovunque definite, è necessario riscrivere con più cura le nozioni di definizione per ricorsione e per composizione.

Definizione 4.3. Siano $g(x_1,\ldots,x_{n-1})$ e $h(x_1,\ldots,x_{n-1},y,z)$ due funzioni (eventualmente non ovunque definite), diciamo che $f : N^n \to N$ è ottenuta per *ricorsione* dalle due funzioni $g(x_1,\ldots,x_{n-1})$ e $h(x_1,\ldots,x_{n-1},y,z)$ se:

- ogni volta che g è definita in $x_1,\ldots, x_{n-1}$ allora f è definita in $x_1,\ldots, x_{n-1},0$ e risulta
  $f(x_1,\ldots,x_{n-1},0) = g(x_1,\ldots,x_{n-1})$.
- se $f(x_1,\ldots,x_{n-1},y)$ è definita in $x_1,\ldots, x_{n-1},y$ ed h è definita in $x_1,\ldots,x_{n-1}, y, f(x_1,\ldots,x_{n-1},y)$ allora
  $f(x_1,\ldots,x_{n-1}, y+1) = h(x_1,\ldots,x_{n-1}, y, f(x_1,\ldots,x_{n-1},y))$.

Definizione 4.4. Sia h una funzione ad m argomenti e siano $g_1,\ldots,g_m$ funzioni ad n argomenti, diciamo che f è ottenuta per *composizione* da h e $g_1,\ldots,g_m$ se risulta

$$f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n))$$

Si assume che f sia definita in $x_1,\ldots x_n$ solo se tutte le funzioni g_i sono definite in $x_1,\ldots x_n$ ed h è definita in $g_1(x_1,\ldots,x_n), \ldots,g_m(x_1,\ldots,x_n)$. Naturalmente, se h e $g_1,\ldots,g_m$ sono totali allora anche f è totale.

Definizione 4.5. Chiamiamo *ricorsiva* ogni funzione che si possa ottenere a partire dalle funzioni-base tramite reiterata applicazione della composizione, ricorsione e minimalizzazione.

Teorema 4.6. E' possibile procedere ad una codifica di tutte le funzioni ricorsive.

Dim. E' sufficiente ampliare la definizione di "schema di calcolo" aggiungendo anche la definizione per minimalizzazione. Successivamente si procede ad una codificazione di tutti gli schemi di calcolo e questo consente di assegnare un indice a tutte le funzioni parziali ricorsive. □

Nel seguito indicheremo con f_i la funzione ricorsiva di indice i, cioè la funzione il cui schema di calcolo ha indice i. Naturalmente due diversi schemi possono rappresentare la stessa funzione ricorsiva e che quindi può capitare che $f_i = f_j$ pur essendo $i \neq j$.

Nota. Non è possibile estendere la dimostrazione del Teorema 4.1 al caso delle funzioni ricorsive. Infatti, definiamo f con il porre $f(x) = f_x(x)+1$ e supponiamo che f abbia indice j. Allora, se f è definita in j si perviene all'assurdo per cui $f(j) = f_j(j)$ e $f(j) = f_j(j)+1$. Da ciò si conclude solo che f non è definita in j.

Giungiamo ora alla questione fondamentale:

la definizione di funzione parziale ricorsiva è abbastanza ampia da rappresentare tutte le funzioni intuitivamente computabili ?

In altre parole, pensiamo che i concetti dati in questo paragrafo siano adeguati a rappresentare adeguatamente l'idea di funzione computabile? Torneremo su tale questione nel paragrafo 8 dopo avere visto la questione della computabilità dal punto di vista della macchine e dove formuleremo la tesi di Church.

5. Macchine che eseguono istruzioni: macchine astratte.

Avendo precisato, da un punto di vista matematico, che cosa si debba intendere per funzione computabile si pone ora il problema di dare una definizione di "macchina che esegue calcoli" che corrisponde a tale classe di funzioni. Naturalmente vogliamo darne una definizione matematica perché solo in questo modo è possibile elaborare una teoria dei calcolatori che non sia legata all'evoluzione della tecnologia. Il solo fatto che in molti punti di questi appunti compare la parola "teorema" oppure la parola "proposizione" presuppone che si sia fornita una definizione matematica di macchina. La nozione di automa finito proposta nel capitolo precedente è un esempio di definizione di macchina. D'altra parte tale definizione non è adeguata poiché abbiamo visto che gli automi non sono in grado nemmeno di fare le moltiplicazioni. Poiché poi tutte le macchine che abbiano una quantità finita di memoria sono di fatto automi finiti e, l'unica via per definire un concetto adeguato di macchina è quella di fare cadere l'ipotesi che la memoria sia finita. Inoltre si deve ammettere che un calcolatore sia un marchingegno che svolge diverse funzioni e che tali funzioni dipendano dal programma che viene scritto nel calcolatore. In altre parole un calcolatore lavora in due fasi:

Fase 1: scrittura di un programma inteso come un insieme finito di istruzioni da eseguire secondo un opportuno ordine
Fase 2: esecuzione delle istruzioni presenti nel programma.

Per analizzare tali tipi di macchine supponiamo di scrivere opportune "istruzioni" che debbono essere seguite per poter calcolare la somma o il prodotto di due numeri. Ad esempio, in un qualunque linguaggio di programmazione, un semplice programma che (a meno di "overflow" cioè a meno di segnale di mancanza di memoria) calcoli la somma potrebbe essere il seguente:

1. chiedi i numeri da addizionare x ed y
2. poni $s := x$ e $c := 0$
3. $s := s+1$; $c := c+1$ (incrementa s e c di una unità);
4. SE $c = y$ ALLORA vai a 5 ALTRIMENTI vai a 3
5. scrivi s.

Le istruzioni 3 e 4 vengono ripetute fino a quando c non sia stato incrementato di una unità un numero di volte pari ad y.
Un programma per il prodotto potrebbe essere:

1. chiedi x ed y
2. poni $p := x$ e $c := 1$
3. $p := p+x$; $c := c+1$;
4. SE $c = y$ ALLORA vai a 5 ALTRIMENTI vai a 3
5. scrivi p

Per individuare una definizione generale di "macchina che esegue istruzioni" analizziamo allora le risorse che deve avere una macchina per poter eseguire programmi di tale tipo.

1. Deve poter memorizzare il valore delle variabili x, y, s, c e p. Poiché memorizzare un valore significa per una macchina assumere un particolare stato, e poiché tali variabili possono assumere infiniti valori, la macchina deve poter assumere infiniti stati.

2. Deve avere la capacità di modificare il valore delle variabili (quindi lo stato della macchina) tramite alcune operazioni (nel nostro caso porre $c = 0$, aggiungere uno, aggiungere x).

3. Deve essere in grado di controllare se lo stato assunto verifichi o meno una data condizione per poter eseguire istruzioni condizionate del tipo SE ... ALLORA ... ALTRIMENTI . . . (nel nostro caso controllare se $c = y$).

In base a tali considerazioni appare naturale proporre la seguente definizione di "macchina che esegue istruzioni".

Definizione 5.1. Una *macchina astratta* è definita da:
Un *linguaggio* costituito da
- un insieme C' (*insieme dei nomi di condizioni o relazioni*)

- un insieme F' (*insieme dei nomi di operazioni*)

Un *"hardware"* costituito da

- un insieme M (*insieme degli stati* o *memoria*)
- una funzione *In* detta *interprete* che associa ad ogni elemento di C' un sottoinsieme di M (cioè una proprietà definita in M) ed ad ogni elemento di F' una funzione parziale di M in M.

Intuitivamente un programma eseguibile da una macchina astratta è un insieme di istruzioni (scritte in un linguaggio che usa i nomi delle condizioni e delle operazioni) indicanti cosa deve fare la macchina. Ogni istruzione inoltre deve essere identificata da una etichetta (ad esempio un numero intero) che rappresenta l'ordine con cui le istruzioni devono essere eseguite e deve anche contenere indicazioni sulla istruzione successiva da eseguire.

Definizione 5.2. Chiamiamo *istruzione di assegnazione* una terna (i,f',j) con i e j numeri naturali ed $f'\in F'$. Chiamiamo *istruzione di salto condizionato* una quadrupla (i,c',h,k) con i, h e k naturali e $c'\in C'$. Il numero i viene detto *indirizzo della istruzione*.

Per indicare l'istruzione di assegnazione (i,f',j) scriveremo anche:

i: $m := f'(m)$; GOTO j.

Per indicare il salto condizionato (i,c',h,k) scriveremo anche :

i: IF $c'(m)$ THEN GOTO h ELSE GOTO k.

Definizione 5.3. Un *programma* è un insieme finito P di istruzioni in cui non possono esserci due istruzioni con lo stesso indirizzo.

Per capire come una macchina infinita esegue un programma diamo la definizione di stato corrente.

Definizione 5.4. Chiamiamo *stato corrente* una coppia (m,i) con $m \in M$ ed $i\in N$. Indichiamo con $SC = M\times N$ l'insieme degli stati correnti.

L'idea è che in uno stato corrente (m,i)

- il numero i sia l'indirizzo dell'istruzione che deve essere eseguita
- m sia lo stato della macchina.

Ogni istruzione determina una funzione di SC in SC cioè permette di passare da uno stato corrente ad un altro stato corrente. Più precisamente:

- Sia (i,f',j) una istruzione di assegnazione e sia $f = In(f')$ l'interpretazione di f'. Allora, l'effetto di (i,f',j) è di fare passare la macchina da uno stato corrente (m,i) allo stato corrente $(f(m),j)$. Pertanto l'esecuzione di tale istruzione determina una modifica dello stato della macchina e l'indicazione dell'istruzione successiva da eseguire.
- Sia (i,c',h,k) un salto condizionato e sia $C = In(c')$ l'interpretazione di c'. Allora l'effetto di (i,c',h,k) è quello di far passare dallo stato corrente (m,i) allo stato (m,h) se $m \in C$ (cioè se m verifica la condizione c'), far passare allo stato (m,k) altrimenti. Pertanto l'esecuzione di una tale istruzione lascia immutato lo stato ma decide quali di due possibili istruzioni sia l'istruzione successiva da eseguire .

Ogni programma determina, dato uno stato di partenza, una successione di stati successivi, successione che può terminare o meno, nel modo seguente.

Definizione 5.5. Dato un programma P ed uno stato m un *processo di calcolo convergente di stato iniziale* m è una successione finita $(m_1,i_1),...,(m_s,i_s)$ di stati correnti tali che

a) $m_1 = m$ e i_1 è l'indirizzo più basso che compare in P;

b) ogni stato corrente (m_j,i_j) non iniziale è ottenuto dal precedente (m_{j-1}, i_{j-1}) applicando l'istruzione in P di indirizzo i_{j-1}.

c) i_s non è l'indirizzo di nessuna istruzione in P.

In tale caso si dice anche che P, dato lo stato iniziale m, *converge in s passi*. Un *processo di calcolo non convergente* è una successione infinita di stati correnti (m_1, i_1), (m_2,i_2),..., tale che siano verificate a) e b).

Dato uno stato iniziale m ed un programma P è univocamente definito il processo di calcolo corrispondente ed, in caso di convergenza, lo stato finale m_s. Indicheremo con f_P la funzione (non ovunque definita) tale che $f_P(m) = m'$ se e solo se il processo di calcolo con stato iniziale m è convergente ed ha come stato finale $m_s = m'$.

Esempio. Le definizioni date rappresentano bene l'idea di moderno calcolatore programmabile. Tuttavia rappresentano bene anche tutti i processi di "esecuzione di istruzioni". Ad esempio supponiamo di prendere una ricetta da un libro di cucina. Possiamo vedere tale ricetta come una successione di istruzioni e quindi come programma P e l'insieme delle possibili situazioni in cucina come l'insieme M dei possibili stati. Lo stato iniziale m_1 coincide con lo stato della cucina al momento in cui si comincia ad applicare la ricetta. Ogni istruzione, una volta che sia stata eseguita, modifica lo stato della cucina. Più precisamente una istruzione di assegnazione è una indicazione su come modificare lo stato della cucina; ad esempio "versa la farina nel recipiente", "accendi il forno" e così via. Le istruzioni di salto condizionato servono, ad esempio, a rappresentare istruzioni del tipo

"gira la crema fino a che non diviene sufficientemente solida "

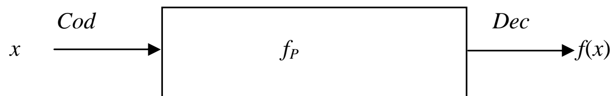
in cui si presuppone che si possa controllare la densità della crema e ripetere l'istruzione "dai una girata" fino a quando la condizione "crema solida" non si soddisfatta. Un tale tipo di comportamento sarà descritto dalle istruzioni

(n, dai una girata), (n+1, crema solida ?, n+2,n), (n+2, spegni il fuoco).

Definizione 5.6. Siano X ed Y due insiemi, detti rispettivamente *insieme degli input* ed *insieme degli output*, una *funzione di codifica Cod* : $X \to M$ ed una *funzione di decodifica Dec*: $M \to Y$. Sia $f : X \to Y$ una funzione parziale il cui dominio è $D \subseteq X$. Diremo che f è *computabile tramite il programma P* se, detta $f_P : M \to M$ la funzione computabile dal programma P,

- $x \in D$ se e solo se il processo di calcolo a partire da $Cod(x)$ è convergente,
- $f(x) = Dec(f_P(Cod(x)))$.

Pertanto, dato un programma P ed un input $x \in X$, il calcolo di $f(x)$ avviene al modo seguente:
- si codifica l'input x in modo da trasformarlo in uno stato $Cod(x)$ della macchina;
- si esegue il processo di calcolo definito dal programma P
- se il processo di calcolo termina nello stato $m = f_P(Cod(x))$, l'output è $y = Dec(m)$, cioè è la decodifica di m.



Nel seguito supporremo che in un programma P vi sia sempre una istruzione di assegnazione del tipo $(i,read,j)$ essendo i l'indice più basso del programma. L'effetto di tale istruzione è che se si fornisce input x allora lo stato corrente diviene $(Cod(x),j)$. Supporremo anche che vi sia una istruzione del tipo $(i,write,j)$, con j indirizzo maggiore di tutti gli indirizzi delle istruzioni in P. L'effetto di tale istruzione è di fornire il risultato del calcolo $Dec(m)$ e poi di fermarsi (in quanto l'istruzione j non può essere eseguita). Tali istruzioni non fanno parte del vero e proprio apparato di calcolo.

Nota. Spesso in un programma ometteremo il secondo indirizzo di una istruzione con la convenzione che (i,f') e (i,c',h) rappresentano le istruzioni (i,f',m) e (i,c',h,m) essendo m il primo degli indirizzi in P successivo ad i. A volte ometteremo anche il primo indirizzo con la convenzione per cui l'ordine in cui sono

state scritte le istruzioni definisce anche l'ordine degli indirizzi. Ad esempio il programma $(1,f')$, g', $(c',1)$ che possiamo scrivere:
1. $m:=f'(m)$; $m:=g'(m)$; IF $c'(m)$ THEN GOTO 1
rappresenta il programma $(1,f',2)$, $(2,g',3)$, $(3,c',1,4)$.

6. Macchine a registri.
Consideriamo ora una classe di macchine astratte che rappresenta abbastanza bene la struttura dei calcolatori realmente esistenti. In tali macchine la memoria è divisa in h parti $r_1,...,r_h$ dette *registri* ed in ogni registro può essere scritto un numero intero. Pertanto uno stato è dato da una h-pla di numeri interi e $M = N^h$. Gli stati della macchina saranno modificati o letti facendo riferimento al contenuto dei singoli registri e per fare questo considereremo anche h "variabili" $x_1,...,x_h$ per denotare il contenuto dei registri $r_1,...,r_h$. Perveniamo allora alla seguente definizione.

Definizione 6.1. Chiameremo *macchina ad h registri* la macchina astratta con $M = N^h$ le cui operazioni sono:
a) aggiungere 1 al registro r_i; in breve $x_i := x_i+1$;
b) copiare in r_i il contenuto del registro r_j; in breve $x_i:=x_j$
c) porre in r_i il numero intero k; in breve $x_i := k$.
Inoltre supponiamo che le condizioni verificabili siano:
d) se r_i e r_j hanno lo stesso contenuto; in breve $x_i=x_j$
e) se il contenuto di r_i è l'intero k; in breve $x_i=k$.

Le trasformazioni a), b) e c) corrispondono alle funzioni $s:N^h \to N^h$ definite rispettivamente da:
$s(x_1,...,x_i,...,x_h) = (x_1,...,x_i+1,...,x_h)$
$s(x_1,...,x_i,...,x_h) = (x_1,...,x_j,...,x_h)$
$s(x_1,...,x_i,...,x_h) = (x_1,...,k,...,x_h)$.
Le condizioni d) ed e) agli insiemi

$$\{(x_1,...,x_h) \mid x_i = x_j\} \text{ e } \{(x_1,...,x_h) \mid x_i = k\}.$$

Per $X=N$ ed $Y=N$, cioè se si vogliono calcolare funzioni di N in N, si può decidere di usare il registro r_1 per gli input ed il registro r_2 per gli output. Ciò si esprime scrivendo all'inizio di un programma l'istruzione $read(x_1)$ ed alla fine $write(x_2)$. In altre parole:
$Cod(n) = (n,0,0...)$
$Dec((x_1,x_2,...,x_h))=x_2$.
Per $X = N\times N$, cioè se si vogliono definire le funzioni di $N \times N$ in N, allora si potrebbe porre $Cod(n,m) = (n,m,0...)$ e $Dec((x_1,x_2,...,x_h)) = x_3$. Scriveremo allora all'inizio del programma $read(x_1,x_2)$ ed alla fine $write(x_3)$. Nei programmi esposti all'inizio del capitolo, si sono utilizzate le variabili x, y, s, c, p e tali programmi sono eseguibili da macchine che abbiano cinque registri. Uno stato della macchina è espresso dalla assegnazione di cinque numeri interi alle variabili. Riferendoci al programma della addizione, se si forniscono gli input 5 e 3 avremo un processo di calcolo convergente in cui gli stati subiscono la seguente evoluzione:

$$(5,3,0,0), (5,3,5,0), (5,3,6,1), (5,3,7,2), (5,3,8,3).$$

Nota. Le macchine a registri che abbiamo definito sono una astrazione matematica che, come tutte le astrazioni, riproduce solo in parte le caratteristiche delle macchine calcolatrici attualmente esistenti. La principale differenza con le macchine reali consiste nel fatto che i registri delle macchine reali possono contenere solo numeri al di sotto di uno dato limite. D'altra parte le macchine reali hanno memoria finita e quindi sono automi finiti. Inoltre nelle macchine reali viene immagazzinato in memoria anche il programma ed il meccanismo che permette di eseguire i programmi.

A proposito delle macchine a registri vale il seguente fondamentale teorema.

Teorema 6.2. Le funzioni calcolabili da una macchina a registri sono tutte e sole le funzioni ricorsive.

Dim. Ci limiteremo a dimostrare che ogni funzione parziale ricorsiva è computabile da una macchina a registri. Denotiamo con *FR* l'insieme delle funzioni computabili da una macchina a registri. Per provare il nostro asserto proveremo che:
1. *FR* contiene le funzioni base
2. *FR* è chiuso rispetto alla composizione, alla ricorsione ed alla minimalizzazione.
E' immediato che le funzioni-base appartengono ad *FR*, infatti la funzione proiezione *i*-esima è calcolabile dal programma "senza istruzioni"
$read(x_1,x_2,...,x_n)$; write(x_i).
La funzione zero si può calcolare con:
$read(x_1)$;
$x_2 := 0$;
$write(x_2)$.
La funzione successore con
$read(x_1)$;
$x_2:=x_1+1$;
$write(x_2)$.

Chiusura per composizione: Supponiamo ora che $g_1,...,g_m$ ed h siano funzioni in *FR*, che f sia stata ottenuta per composizione da h e $g_1,...,g_m$ cioè che

$$f(x_1,...,x_n) = h(g_1(x_1,...,x_n),...,g_m(x_1,...,x_n))$$

e che $\pi_1,...,\pi_m$, π_0 siano i programmi computanti rispettivamente $g_1,...,g_m$ ed h. Supponiamo che ciascun programma π_i inizi con $read(x_1,...,x_n)$ e termini con $write(y_i)$ e che π_0 inizi con $read(y_1,...,y_m)$ e termini con $write(z)$. Possiamo ottenere un programma π scrivendo
$read(x_1,...,x_n)$
π_1*
π_2*
...
π_0*
Il programma π_i* è stato ottenuto da π_i eliminando le istruzioni di input e di output e π_0* si è ottenuto da π_0 eliminando l'istruzione di input. E' immediato che tale programma calcola la funzione f.

Chiusura per ricorsione: Supponiamo che f sia ottenuta per ricorsione dalle due funzioni $g(x_1,...,x_{n-1})$ e $h(x_1,...,x_{n-1},t,z)$ cioè che
$f(x_1,...,x_{n-1},0) = g(x_1,...,x_{n-1})$;
$f(x_1,...,x_{n-1}, x) = h(x_1,...,x_{n-1}, x, f(x_1,...,x_{n-1},x-1))$.
e supponiamo che tali funzioni siano calcolate rispettivamente dai programmi π_1 (che inizia con $read(x_1,...,x_{n-1})$ e termina con $write(z)$) e π_2 (che inizia con $read(x_1,...,x_{n-1},t,z)$ e termina con $write(z)$). Il programma π che vogliamo costruire lavora al modo seguente:
- quando riceve in input i numeri $x_1,\dots,x_{n-1},x$ comincia a calcolare tramite π_1 il valore $y = g(x_1,...,x_{n-1})$ di f in $x_1,\dots,x_{n-1},0$
- successivamente utilizza tale valore per calcolare, tramite π_2, il valore $h(x_1,...,x_{n-1}, x, f(x_1,...,x_{n-1},x-1))$ di f in $x_1,\dots,x_{n-1},1$
- utilizza il valore ottenuto in questo modo per calcolare il valore di f in $x_1,\dots,x_{n-1},2$
- . .
- ripete tale processo fino a quando non ha raggiunto x_n
Più precisamente costruiamo il programma π a partire dai programmi π_1 e π_2 al modo seguente:

$read(x_1,...,x_n)$

```
    t := 0                      (mette il contatore t uguale a zero)
    π1*                         (calcola y = g(x1,...,xn-1) = f(x1,...,xn-1,0))
1   IF t = xn THEN GOTO 2       (controlla se t ha raggiunto xn ed in tale caso va all'istruzione 2)
    t: = t+1                    (incrementa il contatore t di una unità)
    π2*                         (calcola y = h(x1,...,xn-1,t,z))
    GOTO 1:                     (torna all'istruzione 1 per ripetere il ciclo)
2   WRITE(y).                   (stampa il risultato e si ferma)
```

dove π_1* e π_2* sono stati ottenuti da π_1 e π_2 eliminando le istruzioni di input e di output.

E' immediato che π calcola la funzione f .

Chiusura per minimalizzazione: Supponiamo ora che la funzione f di n variabili sia definita per minimalizzazione dalla funzione totale g di n+1 argomenti:

$$f(x_1,...,x_n) = min\{y \mid g(x_1,...,x_n,y) = 0\}$$

e sia π_1 un programma per calcolare g che abbia come input $read(x_1,...,x_n,y)$ e come output $write(z)$. Allora vogliamo scrivere un programma che verifichi per y = 0, 1, 2, 3 ... se risulta che $g(x_1,...,x_n,y) = 0$. Il primo valore di y per cui tale condizione è verificata viene dato come output. A tale scopo è sufficiente considerare il seguente programma π

```
    read(x1,...,xn)
    y : = 0:
1   π1*                  (calcola il valore di z = g(x1,...,xn,y))
    IF z = 0 THEN GOTO 2
    y: = y+1
    GOTO 1
2:  write(y)
```

dove π_1* si ottiene da π_1 eliminando l'istruzione input ed output. E' evidente che tale programma computa f.

c.d.d.

7. Linguaggio macchina e linguaggi evoluti: WHILE, FOR

Definendo le macchine astratte ed in particolare le macchine a registri abbiamo allo stesso tempo definito un relativo linguaggio di programmazione. Chiamiamo *linguaggio macchina* un tale tipo di linguaggio. Ora scrivere programmi nel linguaggio macchina è alquanto complicato e di solito vengono usati linguaggi di programmazione che si rivelano più maneggevoli. In tali linguaggi sono consentiti nuovi tipi di istruzioni che poi vengono tradotte, tramite un "interprete", in sequenze di istruzioni nel linguaggio macchina. Questo permette di esprimere in breve operazioni complesse e di usare un linguaggio più vicino al linguaggio comunemente usato dall'uomo. Nel seguito, riferendoci alle macchine a registri, forniamo i principali esempi.

Istruzioni condizionate: **IF ...THEN...ELSE**

Abbiamo già esaminato le istruzioni di salto condizionato del tipo IF <condizione> THEN *n* ELSE *m* dove *n* ed *m* sono etichette di istruzioni da eseguire. Un comando simile è del tipo :

IF <condizione> THEN < istruzioni1> ELSE DO < istruzioni2>

dove <istruzioni1> e <istruzioni2> denotano una sequenza di istruzioni.

L'interprete del linguaggio di programmazione traduce tale istruzione nella seguente lista di istruzioni per macchine a registri:

j_0: IF <condizione> THEN i_1 ELSE i_2
j_1 : . . .
. . .
i_1 <istruzioni1> (si esegue il blocco di istruzioni <istruzioni1> e poi si salta all'istruzione j_1 successiva a j_0)
i_2 <istruzioni2> (si esegue il blocco di istruzioni <istruzioni2> e poi si salta all'istruzione j_1 successiva a j_0)

dove
i_1 è l'etichetta della prima istruzione di un sottoprogramma che esegue <istruzioni1> e poi rimanda all'istruzione i_0
i_2 è l'etichetta della prima istruzione di un sottoprogramma che esegue <istruzioni2> e poi rimanda all'istruzione i_0
Tralasciamo la scrittura precisa di tale lista di istruzioni.

Cicli limitati: FOR ... TO Un altro esempio importante è l'istruzione di *ciclo limitato* che consiste nell'eseguire un blocco < istruzioni> di istruzioni un numero predeterminato di volte. Una tale istruzione assume una forma del tipo

FOR *j*: = 1 to 5 DO < istruzioni> ;

L'interprete del linguaggio di programmazione traduce tale istruzione nel seguente linguaggio macchina:

j:=0
i_1: j:= j+1
<istruzioni>
IF j = 5 THEN GOTO i_2 ELSE GOTO i_1
i2: . . . . .

Cicli condizionati: WHILE. Un altro tipo di istruzioni sono quelle che consentono di ripetere un blocco di istruzioni fino a quando una certa condizione sia verificata. Una tale istruzione assume una forma del tipo

WHILE <condizione> DO <istruzioni> .

Dove <condizione> denota una asserzione che può essere vera o falsa. Tale istruzione può essere riscritta tramite il seguente blocco di istruzioni per macchine a registri:

i_1 IF condizione THEN GOTO i_2 ELSE GOTO i_3
i_2 <istruzione>
GOTO i_1

i_3 ....

Naturalmente, in un ciclo condizionato è possibile che il ciclo si ripeta all'infinito perché <condizione> è sempre verificata.

Il fatto che tali nuove istruzioni possono essere "tradotte" in istruzioni per una macchina a registri prova che la classe delle funzioni che sono computabili da tali linguaggi evoluti non è più grande di quella delle funzioni computabili dalle macchine a registri.
I due tipi di cicli corrispondono alla classe delle funzioni primitive ricorsive ed alla classe delle funzioni ricorsive.

8. La nozione di funzione computabile: tesi di Church

Sino ad ora abbiamo indicato due modi di intendere la computabilità. Abbiamo definito infatti:
- la classe delle funzioni ricorsive ;
- la classe delle funzioni computabili tramite macchine a registri;

Ora, come abbiamo enunciato e come è possibile dimostrare rigorosamente, accade tali classi coincidono. Più in generale, allo stato attuale delle cose, si possono affermare i seguenti fatti:

1. Per ogni funzione incontrata fino ad ora che sia apparsa essere computabile da un punto di vista intuitivo non è stato difficile dimostrare che è ricorsiva (equivalentemente che esiste un programma per una macchina a registri capace di computarla) ;
2. Ogni definizione astratta di computabilità proposta sino ad ora ha individuato una classe di funzioni che è risultata essere la stessa delle funzioni ricorsive.

Da questi fatti si è giunti ad un convincimento che va sotto il nome di *Tesi di Church.*

Tesi di Church. La definizione matematica di funzione ricorsiva è perfettamente adeguata a rappresentare l'idea intuitiva di funzione computabile.

Si noti che tale tesi non è un teorema e non è dimostrabile. Infatti essa non collega due definizioni matematiche (come nel caso della equivalenza tra la definizione di parziale ricorsività e computabilità tramite le macchine a registri). La tesi collega una definizione con una intuizione (quella di computabilità) e tale intuizione potrebbe variare da soggetto a soggetto oppure subire un cambiamento in epoche diverse. Per capire la natura di tale tesi, proviamo a scrivere una tesi simile per quello che riguarda la definizione di funzione.

Tesi di Cantor sulle funzioni. La definizione di funzione data in teoria degli insiemi è adeguata a rappresentare la dipendenza di una grandezza da un'altra.

Prima di Cantor una tale tesi non sarebbe stata accettata. Si era portati a chiamare funzioni solo le funzioni di tipo algebrico e trascendente e comunque funzioni di cui si conoscesse una espressione analitica.
Oppure una tesi per quello che riguarda la continuità:

Tesi della nozione di continuità. La definizione di continuità data in analisi matematica (più in generale in topologia) è adeguata a rappresentare l'idea di processo continuo.

Poiché le definizioni di funzione computabile si sono dimostrate equivalenti, nel seguito useremo l'espressione "computabile" per indicare indifferentemente che una funzione è parziale ricorsiva oppure è computabile tramite una macchina di Turing oppure tramite una macchina a registri oppure tramite un linguaggio evoluto.

Definizione 8.1. Per *funzione computabile* intenderemo una funzione di N^p in N che sia computabile tramite un programma in una macchina a registri.

D'altra parte un qualunque linguaggio di programmazione può essere sempre interpretato tramite una macchina a registri. Quindi potremmo chiamare computabile una funzione che sia computabile tramite uno dei qualunque linguaggi di programmazione esistenti. Poichè accettiamo la tesi di Church per cui tutte le possibili definizioni di computabilità sono equivalenti, nel seguito parleremo di computabilità invece che di computabilità. Pertanto possiamo proporre al posto della definizione 1.1. la seguente definizione che pur essendo meno precisa è più intuitiva:

Definizione 8.1b. Chiamiamo *computabile* una funzione che sia computabile tramite uno qualunque dei linguaggi di programmazione in commercio.

Si noti che tra chi studia la teoria della computabilità esiste una strana abitudine:
utilizzare la tesi di Church come se fosse un assioma.
Ciò significa che ogni volta che viene descritto un procedimento informale per il computo di una funzione allora se "appare evidente" che tale procedimento è effettivamente eseguibile allora si accetta che esso si possa trasformare, ad esempio, in un programma per una macchina a registri. Un tale modo di procedere è imposto dalla necessità di non rendere inutilmente complicate e noiose le dimostrazioni e di distinguere il lavoro di ricercatore in teoria della computabilità dal lavoro di programmatore. Nel seguito, per non cadere nella pedanteria, adotteremo anche noi questo modo di procedere.

Concludiamo questo paragrafo mettendo in evidenza alcune proprietà di "chiusura" della classe delle funzioni computabili.

Proposizione 8.2. La somma, il prodotto la differenza, di due funzioni computabili è ancora una funzione computabile.

Dim. Siano f e g computabili, allora esiste un programma π_1 per computare f del tipo
$read(x_1)$; P_1 ; $write(y_1)$
ed un programma π_2 per computare g del tipo
$read(x_2)$; P_2 ; $write(y_2)$
(abbiamo indicato con P_1 e P_2 blocchi di istruzioni che costituiscono i rispettivi programmi ed abbiamo fatto in modo che una stessa variabile non compaia nei due programmi). Inoltre esiste un programma π capace di computare la somma del tipo
$read(y_1,y_2)$; P ; $write(y)$
dove y è appunto la somma di y_1 più y_2. Allora
$read(x_1,x_2)$; P_1 ; P_2 ; P : $write(y)$
è un programma per calcolare la somma $f(x_1)+g(x_2)$. Allo stesso modo si procede per il prodotto, la differenza.

Proposizione 8.3. L'applicazione identica è una funzione computabile, la composizione di due funzioni computabili è una funzione computabile. Pertanto la classe delle funzioni computabili costituisce un monoide che è sottomonoide del monoide $(N^N,\circ)$ delle funzioni di N in N.

L'inversa di una funzione computabile biettiva è una funzione computabile. Pertanto la classe delle biezioni computabili costituisce un gruppo che è un sottogruppo del gruppo delle permutazioni di N.

Formattato: Tipo di carattere: Times New Roman, 11 pt, Corsivo, Italiano (Italia), Pedice

Dim. E' evidente che l'applicazione identica è una funzione computabile. Siano f e g computabili, allora esiste un programma π_1 per computare f del tipo
$read(x_1)$; P_1 ; $write(y_1)$
ed un programma π_2 per computare g del tipo
$read(x_2)$; P_2 ; $write(y_2)$
Allora un programma per calcolare, dato l'input x_1, il valore $g(f(x_1))$ si ottiene "incollando" il programma π_2 dopo il programma π_1 dopo avere sostituito $read(x_2)$ con $x_2 := y_1$.

. Sia f una funzione biettiva e computabile, allora un algoritmo per calcolare, per ogni $y \in N$ il numero $f^{-1}(y)$ si ottiene con un ciclo infinito

$n := 0$
$n := n+1$
IF $f(n) = y$ THEN Write(n) ELSE goto 1

9. Codifica dei programmi e macchine universali.
Dato un qualunque linguaggio di programmazione, è possibile definire, per ogni intero n, un processo di codifica dei programmi ad n ingressi in modo da associare un indice i ad ogni programma con n variabili di ingresso. Abbiamo già visto questa tecnica quando abbiamo effettuato una codifica degli schemi di calcolo e tale codifica si ottiene, più in generale, al modo seguente:
i) ad ogni simbolo del linguaggio di programmazione si associa (in maniera iniettiva) come codice a lunghezza costante una sequenza di 0 ed 1 di lunghezza d ed iniziante con 1;
ii) ad ogni programma π si associa il numero i, scritto in base due, che si ottiene sostituendo ogni simbolo in π con il relativo codice.

In tale modo abbiamo associato ad ogni programma un numero naturale. Viceversa sia i un numero naturale, allora:

i) si scrive il numero i in base due e si divide la sequenza ottenuta in blocchi di lunghezza d, procedendo ad esempio da sinistra verso destra;
ii) si sostituisce ad ogni blocco di 0 ed 1 il simbolo di cui è codice se tale blocco è un codice mentre si cancellano eventuali blocchi che non sono codici di niente;
iii) se quanto ottenuto è un programma in linguaggio macchina scritto correttamente allora ad i si associa tale programma a cui si è anteposto l'istruzione *read*($x_1,...,x_n$) e posposto l'istruzione *write*(y);
iv) se quanto appare è una scritta priva di significato allora ad i si associa un qualunque programma fissato per convenzione, ad esempio il programma

read($x_1,...,x_n$) ;
y :=0
write(y).

Definizione 9.1. Indicheremo con π_i^n il programma ad n ingressi di indice i e con f_i^n la funzione n-aria calcolata da tale programma. Per le funzioni di una variabile ometteremo l'indice in alto.

Quindi π_i denota il programma con un input di codice i e f_i denota la funzione di una variabile di codice i. Si noti che la numerazione delle funzioni computabili definita in questo modo non è iniettiva. Infatti ogni funzione computabile ammette infiniti indici, cioè può essere computata da infiniti programmi diversi tra loro.

Proposizione 9.2. Per ogni funzione computabile f l'insieme $\{i \in N \mid f_i = f\}$ è infinito. In altre parole esistono infiniti programmi capaci di computare f.

Dim. Sia π un programma *read*($x_1,...,x_n$) ; ... ; *write*(y) computante f, allora è possibile complicare a piacere tale programma senza cambiare la funzione da esso computata. Ad esempio, per ogni intero n, possiamo considerare il programma $\pi^{(n)}$ che si ottiene da π inserendo da qualche parte n volte l'istruzione $x_1:=(x_1+1)-1$. Tali programmi definiscono tutti la stessa funzione f ed hanno tutti indici diversi. □

Chiamiamo *universale* un linguaggio di programmazione capace di computare tutte le funzioni totali che siano intuitivamente computabili.

Proposizione 9.3. Se un linguaggio di programmazione è universale, allora non si può richiedere che definisca programmi che siano convergenti qualunque sia l'input.

Dim. Basta adattare la dimostrazione del teorema in cui si prova che esistono funzioni intuitivamente computabili che non sono primitive ricorsive. Infatti la funzione f definita dall'equazione $f(n) = f_n(n)+1$ è intuitivamente computabile. Essendo il linguaggio universale, in tale linguaggio può essere scritto un programma che computa f. Detto j l'indice di tale programma, abbiamo che $f = f_j$. Se per assurdo f fosse definita in j allora $f(j) = f_j(j)$ in contrasto con l'ipotesi per cui $f(j) = f_j(j)+1$. □

La possibilità di codificare tutte le funzioni computabili permette di provare l'esistenza di macchine universali cioè macchine capaci di calcolare qualunque funzione computabile. Per definizione noi abbiamo che per ogni funzione computabile allora esiste una macchina a registri capace di computarla tramite un opportuno programma. Però potrebbe capitare che ogni funzione computabile ad n variabili richieda una macchina diversa. Allora si avrebbe la spiacevole situazione per cui non è sufficiente avere comprato un solo calcolatore per potere fare tutti i calcoli desiderati ma che esistano tipi di calcolo che ci obbligano all'acquisto di un calcolatore di tipo diverso. Fortunatamente ciò non accade come mostra la seguente proposizione.

Teorema 9.4. Dato un intero m, esiste un programma π (detto *programma universale*) computante una funzione $U : N^m \times N \rightarrow N$ (detta *funzione universale*) tale che, per ogni indice i, $f_i(x_1,...,x_m) = U(x_1,...,x_m,i)$. Pertanto esiste una macchina a registri capace di computare tutte le funzioni computabili di m variabili.

Dim. Riferiamoci, per comodità, al caso $m = 1$ e consideriamo il seguente algoritmo.
1. Dato un input x ed un indice i, decodifichiamo i in modo da ottenere il programma π_i.
2. Poi applichiamo tale programma all'input x e, se vi è convergenza, chiamiamo con $U(x,i)$ il risultato di tale calcolo.
La funzione $U : N \times N \rightarrow N$ definita in questo modo è computabile quindi, per la tesi di Church è computabile da una macchina a registri tramite un opportuno programma P. E' evidente che $f_i(x)=U(x,i)$ e che quindi U è la funzione universale cercata. Naturalmente la macchina a registri capace di calcolare U è capace anche di calcolare tutte le funzioni computabili di una sola variabile. Il programma universale viene chiamato anche *programma interprete*, infatti tale programma è in grado, in un certo senso, di leggere ogni altro programma e tradurlo in opportune operazioni. Più precisamente, quando viene "scritto" un programma P tale programma viene codificato in un numero i e scritto in un registro della macchina universale. In tale senso per il programma universale ogni altro programma è un input da memorizzare.

10. Funzioni computabili in tempi ragionevoli: teoria della complessità

L'esperienza insegna che esistono funzioni che, pur essendo in linea di principio calcolabili, non lo sono "di fatto". Infatti ci si accorge che i programmi per calcolare i valori di tali funzioni richiedono tempi di calcolo troppo lunghi, oppure memoria troppo grande. Questo fatto suggerisce che la definizione di funzione calcolabile che abbiamo considerato in questo capitolo andrebbe rivista e che sarebbe opportuno fare un tentativo per individuare una classe più stretta di funzioni, quella delle funzioni "calcolabili in tempi ragionevoli" oppure "calcolabili con una quantità di memoria non troppo grande". La teoria che studia un tale tipo di problemi, che prende il nome di *Teoria della complessità*, è molto ampia e sviluppata ed in questi appunti ci limitiamo solo a darne una vaga idea. Inoltre ci limiteremo a valutare la complessità solo in riferimento ai tempi di calcolo. Tali tempi sono valutati con riferimento al numero di passi compiuti in un processo di calcolo convergente oppure, equivalentemente, al numero di volte in cui è stata eseguita una istruzione nell'eseguire il programma.

Definizione 10.1. Per ogni indice i definiamo la funzione parziale T_i ponendo, per ogni intero x:
- $T_i(x) = n$ nel caso in cui il processo di calcolo del programma π_i con input x è convergente in n passi

- $T_i(x)$ non definita se il processo di calcolo di π_i con input x non è convergente.

Da tale definizione segue che se la funzione f_i è totale allora anche T_i è totale. Una definizione analoga può essere data nel caso delle macchine di Turing, assumendo che $T_i(x)$ sia il numero di passi che la macchina ha eseguito nel caso in cui la macchina si fermi e che $T_i(x)$ non sia definita nel caso in cui la macchina non si fermi. Per mostrare che T_i è una funzione computabile introduciamo la nozione di contatore che è una variabile che scatta di una unità ogni volta che sia stata eseguita una istruzione.

Definizione 10.2. Dato un programma π diciamo che è stato inserito un *contatore* se si considera un altro programma π^c che si ottiene:
- fissando una nuova variabile c che inizialmente è posta uguale a zero tramite l'istruzione $c := 0$
- anteponendo ad ogni istruzione di π l'assegnazione $c := c+1$ in modo che ogni volta che viene eseguita tale istruzione la variabile c si incrementa di una unità.

Il comportamento di π^c è lo stesso di quello di π tranne per il fatto che ogni volta che viene eseguita una istruzione la variabile c si incrementa di una unità. Pertanto se il programma si ferma per un determinato input, allora il valore di c rappresenta *"il numero di passi"* che ha effettuato l'interprete.

Proposizione 10.3. Per ogni indice $i\in N$ la funzione T_i è computabile.

Dim. Dato il programma π_i per una macchina a registri costruiamo un altro programma che si ottiene da π_i
- aggiungendo un *contatore* c a π_i
- sostituendo l'istruzione di stampare il valore dell'output y con l'istruzione di stampare c
In questo modo si ottiene un programma che computa T_i.

Nell'insieme delle funzioni totali di N in N introduciamo la seguente relazione.

Definizione 10.4. Date due funzioni totali $f : N \rightarrow N$ e $g : N \rightarrow N$ poniamo $f \leq_O g$ se esistono $p,q\in N$ tali che $f(x) \leq p\cdot g(x)+q$ per ogni $x\in N$. Indichiamo con $O(g)$ l'insieme $\{f : f \leq_O g\}$

Esempi. Posto $f(x) = 10x+5$ e $g(x) = 3x$ allora risulta che $f \leq_O g$. Infatti, posto $p = 4$ e $q = 5$, risulta che per ogni $x\in N$, $10x+5 \leq p\cdot 3x+q$. Ovviamente risulta anche che $g \leq_O f$. Posto $f(x) = x^2$ e $g(x) = 3x$, risulta, qualunque siano p e q, che la disequazioni $x^2 \leq p\cdot 3x+q$ è verificata, nel campo reale, solo all'interno delle due radici del polinomio $x^2-p\cdot 3x-q$ e quindi solo per un numero finito di valori interi. Pertanto risulta che non è vero che $f \leq_O g$. Invece $g \leq_O f$ poiché la disequazione $3x\leq px^2 +q$ è verificata ad esempio ponendo $p\geq 3$ e $q = 0$.

E' immediato provare la seguente proposizione.

Proposizione 10.5. La relazione $\leq_O$ è una relazione di preordine, cioè è riflessiva e transitiva.

Da notare che la relazione $\leq_O$ è legata anche alla nozione di ordine di un infinito.

Proposizione 10.6. Siano f e g due infiniti in ∞, che siano confrontabili, cioè supponiamo che $lim_{n\to\infty} f(n) = \infty$ e $lim_{n\to\infty} g(n) = \infty$ e che esista $lim_{n\to\infty} f(n)/g(n)$. In tale caso

$f \leq_O g \Leftrightarrow f$ è un infinito di ordine minore o uguale a quello di g.

Dim. Supponiamo che $f \leq_O g$ e quindi che esistano p e q tali che $f(n)/g(n) \leq p+q/g(n)$ per ogni intero n. Dividendo tutto per $g(n)$ e passando a limite avremo che $lim_{n\to\infty} f(n)/g(n) \leq p$ e quindi che f è un infinito di ordine inferiore o uguale a quello di g. Viceversa supponiamo che f sia un infinito di ordine inferiore o uguale a quello di g e quindi che $lim_{n\to\infty} f(n)/g(n) \leq p$ per p opportuno. Allora esiste m tale che $f(n)/g(n) \leq p$ per ogni $n \geq m$ e quindi tale che $f(n) \leq p \cdot g(n)$ per ogni $n \geq m$. Posto $q = Max\{f(1), f(2),...,f(m)\}$ risulta che $f(n) \leq p \cdot g(n)+q$ per ogni intero n.

I casi più interessanti si riferiscono alle funzioni polinomiali ed a quelle esponenziali. Come immediata conseguenza della Proposizione 10.5 abbiamo la seguente proposizione.

Proposizione 10.7. Se f e g sono funzioni polinomiali allora $f \leq_O g$ se e solo se il grado di f è minore del grado di g. Sia e la funzione esponenziale, cioè $e(x) = 10^x$, allora per ogni funzione polinomiale f risulta che $f \leq_O e$ mentre non accade che $e \leq_O f$.

Definizione 10.8. Diciamo che un programma π_i ha *costo* o *complessità almeno* O(g) se $T_i \in$ O(g), cioè se esistono p e q tali che i tempi di calcolo di π_i con input x sono inferiori a $p \cdot g(x)+q$. Diciamo che una funzione f ha *costo* o *complessità almeno* O(g) se esiste un programma che la computa di complessità almeno O(g).

Attraverso la notazione O( ), gli algoritmi vengono divisi in classi. Ad esempio:

1. la complessità *costante* O(1) è posseduta dagli algoritmi che eseguono sempre lo stesso numero di operazioni indipendentemente dalla dimensione dell'input.
2. E' considerato semplice anche un algoritmo la cui complessità sia *lineare* O(n). Tale complessità è posseduta dagli algoritmi che eseguono un numero di operazioni sostanzialmente proporzionale alla dimensione dell'input.
3. Vengono poi le *complessità di tipo polinomiale*.
4. Dopo le classi di complessità polinomiale si passa ad una livello notevolmente superiore: quello della *complessità esponenziale* O(k^n).

È necessario guardare con particolare diffidenza agli algoritmi la cui complessità è esponenziale perché possono richiedere dei tempi di esecuzione proibitivi anche per valori relativamente piccoli di n ed indipendentemente dalla velocità dell'elaboratore. Purtroppo esistono molti problemi, anche di interesse pratico, per i quali non si conoscono ancora algoritmi non esponenziali (problemi intrattabili). Comunemente in informatica si ritiene che un algoritmo sia efficiente se è di complessità polinomiale. Infatti l'esperienza prova che se si riesce a progettare un algoritmo di tipo polinomiale allora tale algoritmo risulta in generale sufficientemente semplice. Siamo ora pronti pertanto a formulare qualcosa di analogo alla tesi di Church:

> La definizione matematica di funzione computabile con algoritmi di complessità polinomiale è adeguata a rappresentare l'idea intuitiva di funzione *"calcolabile in tempi ragionevoli".*

11. Turing e la prima definizione matematica di calcolatore.

La prima definizione matematica di "calcolatore è quella di Turing che abbiamo già incontrato nel primo capitolo. In tale proposta la memoria è infinita perché è infinito il nastro e su tale nastro è possibile scrivere sia l'input che memorizzare dati durante il corso della computazione. Esponiamo più in dettaglio la definizione proposta da Turing.

Definizione 11.1. Una *macchina di Turing* è una struttura matematica del tipo $M = \langle A, -, K, k_0 \rangle$ dove:

A è un insieme finito chiamato *alfabeto*

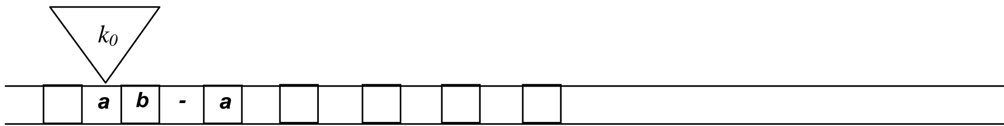
\- è un carattere speciale, interpretato come spazio bianco

K è un insieme finito detto *insieme degli stati*
k_0 è uno stato detto *stato iniziale*

Un *programma* per una macchina di Turing è una funzione parziale $\delta : K\times(A\cup\{-\})\rightarrow K\times(A\cup\{-\})\times\{d,s,i\}$ che chiamiamo *funzione di transizione*.

Da notare che spesso viene chiamata macchina di Turing una struttura del tipo $<A, -, K, k_0, \delta>$ cioè quello che noi abbiamo chiamato macchina di Turing più un programma.

Dato un programma, il comportamento di una macchina di Turing dipende, in modo deterministico, dallo stato in cui è la macchina in un determinato istante e dal simbolo letto sul nastro.



Il disegno rappresenta una macchina di Turing che è nello stato k_0 e che legge il simbolo a scritto sul nastro. Utilizzeremo il carattere speciale - per indicare una casella vuota.
Interpretiamo i simboli d,s,i come spostamento a destra, spostamento a sinistra e immobilità della testina. Precisamente, se $k, k' \in K$ e $c, c' \in A\cup\{-\}$:

- $\delta(k,c) = (k',c',d)$ significa che se la testina legge c e la macchina è nello stato k allora la testina:
  1. passa dallo stato k allo stato k',
  2. sostituisce c con c',
  3. si sposta di un passo verso destra,

- $\delta(k,c) = (k',c',s)$ significa che se la testina legge c e la macchina è nello stato k allora la testina:
  1. passa dallo stato k allo stato k',
  2. sostituisce c con c',
  3. si sposta di un passo verso sinistra,
- $\delta(k,c) = (k',c',i)$ significa che se la testina legge c e la macchina è nello stato k allora la testina:
  1. passa dallo stato k allo stato k',
  2. sostituisce c con c',
  3. rimane ferma.

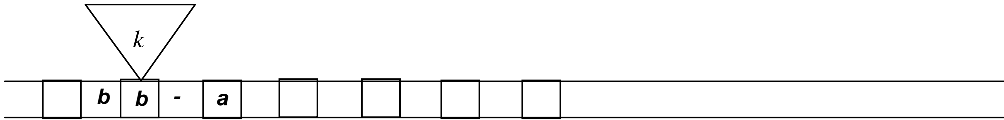
Per eseguire un calcolo scriviamo l'input come una sequenza finita x di caratteri scritti sul nastro eventualmente separati da caselle vuote (cioè da sequenze di -). Inoltre posizioniamo la testina in modo che guardi il primo simbolo diverso da -.

Definizione 11.2. Chiamiamo *configurazione istantanea* una parola del tipo $c_1...c_nkc_{n+1}...c_m$ dove $c_1,...,c_m$ sono caratteri e k è uno stato.

La configurazione $c_1...c_nkc_{n+1}...c_m$ rappresenta la situazione in cui sul nastro è scritta la parola $c_1...c_m$ la testina è nello stato k ed è posizionata sul simbolo c_{n+1}.

Definizione 11.3. Chiamiamo *configurazione iniziale dato l'input x* la configurazione k_0x.

Ad esempio la figura che abbiamo indicato sopra corrisponde all'input *ab-a* (input che viene identificato con la coppia (ab,a)) e quindi alla configurazione $k_0bb\text{-}a$. Successivamente la macchina comincia ad agire in accordo con quanto è imposto dalla funzione di transizione. Ad esempio, se $\delta(k_0,a) = (k,b,d)$ allora la macchina dopo avere sostituito a con b passa nella casella a destra ed assume lo stato k (si veda la figura successiva)



Si perviene allora ad una configurazione che possiamo denotare con *bkb-a*. In definitiva la macchina a partire dalla configurazione iniziale passa attraverso configurazioni successive.

Definizione 11.4. La macchina si ferma se nello stato k legge un simbolo x ma la funzione di transizione non è definita nella coppia (k,x). Quando la macchina si ferma, la sequenza di simboli che sono rimasti sul nastro è l'output, cioè il risultato della computazione.

Naturalmente può capitare che la macchina non si fermi mai per un dato input. Le macchine di Turing possono lavorare su un qualunque alfabeto finito. In particolare possono lavorare su di un alfabeto capace di rappresentare i numeri interi e quindi calcolare funzioni definite in N ed a valori in N. Ad esempio possiamo rappresentare un numero intero n tramite una successione consecutiva di n asterischi $*$ scritti sul nastro. In tale caso l'alfabeto utilizzato consiste solo in $*$ e nel simbolo –.

Esempio. Macchina di Turing che calcola la somma. Supponiamo di rappresentare un numero n sul nastro come successione consecutiva di asterischi $*$ e di separare i due addendi tramite il simbolo -. Pertanto dovendo ad esempio sommare 2 e 3, scriveremo sul nastro la stringa $**-***$. Per ottenere la somma dobbiamo allora solo cancellare il simbolo - per ottenere la stringa $*****$ costituita da cinque asterischi. A tale scopo dopo essere partiti dalla configurazione $q_0**-***$ le operazioni che deve eseguire la macchina sono:

1. spostare la testina fino a raggiungere lo spazio separatore - e senza cancellare gli asterischi,
2. porre al posto di - un asterisco
3. spostare la testina fino a superare la seconda sequenza di asterischi e leggere di nuovo -
4. tornare di un passo indietro
5. cancellare un asterisco.

La funzione di transizione dovrà allora essere rappresentata dalla seguente tabella
$\delta(q_0,*) = (q_0,*,d)$, $\delta(q_0,-) = (q_1,*,d)$, $\delta(q_1,*) = (q_1,*,d)$, $\delta(q_1,-) = (q_2,-,s)$, $\delta(q_2,*) = (q_3,-,i)$.
Per tale macchina, dato l'input $**-***$, si avrà la seguente evoluzione:
$q_0**-***$, $*q_0*-***$, $**q_0-***$, $***q_1***$, $****q_1**$, $*****q_1*$,
$******q_1-$, $*****q_2*-$, $*****q_2-$

Esercizio. Descrivere una macchina di Turing capace di calcolare la funzione $f(x) = x+1$.

Nonostante l'aspetto molto semplice, le macchine di Turing sono uno strumento estremamente potente. Infatti è possibile dimostrare il seguente teorema.

Teorema 11.5. Sono computabili da una macchina di Turing tutte e sole le funzioni ricorsive.

Ne segue quindi che per le macchine di Turing può essere affermata la stessa cosa che si è detto per le funzioni ricorsive, cioè che non è mai stata trovata una funzione (intuitivamente) calcolabile che non sia calcolabile tramite una opportuna macchina di Turing. Ciò suggerisce che la definizione proposta da Turing sia adeguata. Un'altra scoperta di notevole importanza fatta da Turing è la seguente:

Teorema 11.6. Esiste una *macchina universale*, cioè una macchina di Turing ed un programma che permette di "simulare" ogni altro programma di macchina di Turing.

Letture: Gara di Matematica

Cesare Zavattini - *I tre libri, Parliamo tanto di me* - Bompiani - cap. XVI pag. 48,49,50

E' un ricordo della mia infanzia. Abitavo a Gottinga nel dicembre del milleottocentosettanta. Mio padre ed io giungemmo all'Accademia quando il presidente Maust stava cominciando l'appello dei partecipanti alla *Gara Mondiale di Matematica*. Subito babbo andò a mettersi fra gli iscritti dopo avermi affidato alla signora Katten, amica di famiglia. Seppi da lei che il colpo del cannone di Pombo, il bidello, avrebbe segnato l'inizio della storica contesa. La signora Katten mi raccontò un episodio, ignoto ai più, intorno all'attività di Pombo. Costui sparava da trent'anni un colpo di cannone per annunciare il mezzogiorno preciso. Una volta se n'era dimenticato. Il dì appresso, allora, aveva sparato il colpo del giorno prima, e così di seguito fino a quel venerdì del milleottocentosettanta, Nessuno a Gottinga si era mai accorto che Pombo sparava il colpo del giorno avanti.

Esauriti i preliminari, la gara ebbe inizio alla presenza del principe Ottone e di un ragguardevole gruppo di intellettuali.

“Uno, due, tre, quattro, cinque… " Nella sala si udivano soltanto le voci dei gareggianti.

Alle diciassette circa, avevano superato il ventesimo migliaio. Il pubblico si appassionava alla nobile contesa e i commenti si intrecciavano. Alle diciannove, Alain, della Sorbona, si accasciò sfinito.

Alle venti, i superstiti erano sette.

”36767, 36768, 36769, 36770…”

Alle ventuno Pombo accese i lampioni. Gli spettatori ne approfittarono per mangiare le provviste portate da casa.

“40719, 40720, 40721…”

Io guardavo mio padre, madido di sudore, ma tenace. La signora Katten accarezzandomi i capelli ripeteva come un ritornello: 'Che bravo babbo hai,' e a me non pareva neppure di avere fame. Alle ventidue precise avvenne il primo colpo di scena: l'algebrista Pull scattò:

"'Un miliardo "

Un *oh* di meraviglia coronò l'inattesa sortita; si restò tutti col fiato sospeso.

Binacchi , un italiano, aggiunse issofatto:

“'Un miliardo di miliardi di miliardi.' Nella sala scoppiò un applauso subito represso dal Presidente. Mio padre guardò intorno con superiorità, sorrise alla signora Katten e cominciò:

“'Un miliardo di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi…'

La folla delirava: "Evviva, evviva." La signora Katten e io, stretti uno all'altro, piangevamo dall'emozione.

“…di miliardi di miliardi di miliardi di miliardi di miliardi di miliardi.'

Il presidente Maust, pallidissimo, mormorava a mio padre, tirandolo per le falde della palandrana: 'Basta, basta, le farà male.' Mio padre seguitava fieramente:

“… di miliardi di miliardi di miliardi di miliardi ...’ A poco a poco la sua voce si smorzò, l’ultimo fievole *di miliardi* gli uscì dalle labbra come un sospiro, indi si abbattè sfinito sulla sedia. Gli spettatori in piedi lo acclamavano freneticamente. Il principe Ottone gli si avvicinò e stava per appuntargli una medaglia sul petto quando Gianni Binacchi urlò:

"Più uno!"

La folla precipitatasi nell’emiciclo portò in trionfo Gianni Binacchi. Quando tornammo a casa, mia madre ci aspettava ansiosa alla porta. Pioveva. Il babbo, appena sceso dalla diligenza, le si gettò tra le braccia singhiozzando: "Se avessi detto più due avrei vinto io."