Progetto Strategico del C.N.R.

Tecnologie e Innovazioni Didattiche

F. Luccio, E. Lodi e L. Pagli

Algoritmica

Corso introduttivo in Informatica

1986

L'*informatica* studia la natura dell'***informazione***, nonché i principi e le tecniche per elaborarla. E, in quest'ultimo aspetto, acquista rilevanza grandissima l'***algoritmica***, cioè quella parte dell'informatica rivolta alla definizione, costruzione e analisi di comportamento degli algoritmi.

Secondo la teoria classica dell'informazione, che raggiunse una formulazione definitiva alcune decine di anni or sono e che solo in anni recentissimi ha registrato qualche progresso, l'informazione è un'entità definita matematicamente, e misurabile sulla base dell'incremento di conoscenza che il ricevitore di un messaggio ottiene su un argomento determinato. Dissaldata, e successivamente risaldata all' algoritmica, la teoria dell'informazione è uno dei capisaldi della cultura informatica: ad essa è legata la teoria della rappresentazione cui accenneremo sotto.

Ben più antiche sono le radici dell'algoritmica. Pur se il suo assetto teorico definitivo è stato raggiunto nella prima metà di questo secolo, e le tecniche di progetto e analisi di algoritmi hanno segnato progressi enormi con la recente diffusione dei calcolatori elettronici, i primi esempi di algoritmi non banali risalgono alle origini della storia dell'uomo, e sono registrati nei documenti della matematica antica. All'algoritmica, vista in particolare nei suoi aspetti concreti, è dedicato gran parte del presente studio, nella convinzione che al concetto di algoritmo spetti un ruolo centrale nell'informatica, e che tale ruolo possa essere compreso a fondo più con la discussione di esempi che con argomentazioni teoriche.

1.1 Il problema della rappresentazione

Tutte le scienze costruiscono rappresentazioni delle entità di cui si occupano; e, nell'impossibilità di elencare tutte le rappresentazioni per le infinite entità possibili, costruiscono regole di costruzione di tali rappresentazioni. La matematica fornisce ottimi esempi di meccanismi di base comuni alle altre scienze: una semplice regola, che ciascuno può riformulare per suo conto, consente di rappresentare tutti i numeri naturali in base 10: la rappresentazione di tutti questi numeri è infinita, ma la rappresentazione della regola di costruzione non lo è.

In generale le entità da rappresentare possono essere descritte in forma ***digitale***, cioè come stringhe finite di elementi tratti da un alfabeto anch'esso finito. È immediato però verificare che questo caso è intrinsecamente legato a quello della rappresentazione dei numeri naturali, poiché le nuove entità possono essere messe in corrispondenza biunivoca con tali numeri, o, se si preferisce, possono essere impiegate per rappresentarli. Per esempio assegnando alle ventisei lettere del nostro alfabeto i valori numerici:

```
A = 0, B = 1, C = 2, . : . . , Z = 25                    [1.1]
```

tutte le possibili parole possono essere prese a rappresentazione dei numeri naturali scritti in base 26. Insiemi di elementi con tale proprietà si dicono ***numerabili***, ed è legittimo assegnare un indice intero progressivo ai loro elementi.

L'informatica rappresenta tutte le sue entità in forma digitale (stringhe finite costruite su alfabeti finiti; sovente l'alfabeto *minimo* {0,1}), e descrive quindi un mondo

numerabile. Questa grande semplificazione comporta una drastica perdita di potenza: se infatti è ancora possibile individuare mediante regole finite (per esempio la specificazione di serie aritmetiche) singole entità che non ammettono rappresentazioni finite (per esempio numeri trascendenti), non tutti gli insiemi interessanti sono necessariamente numerabili.

Consideriamo per esempio l'insieme F di tutte le funzioni di una variabile che hanno come dominio i naturali e come codominio {0,1}. Ciascuna funzionie f(x) di F può essere specificata da una sequenza infinita come:

x	= 0	1	2	3	4	5	.	.	.	.
f(x)	= 0	1	1	0	1	0	.	.	.	.

[1.2]

o, se possibile, da una regola finita di costruzione come:

$$f(x) = \begin{cases} 0, & \text{se } x \text{ è pari,} \\ 1, & \text{se } x \text{ è dispari.} \end{cases} \qquad [1.3]$$

Mostriamo ora che l'insieme F non è numerabile (e quindi a maggior ragione non lo sono gli insiemi delle funzioni dagli interi sugli interi, o dai reali sui reali). Se infatti tale insieme fosse numerabile, potremmo assegnare a ogni sua funzione un indice progressivo *i* nella numerazione, e concepire una tabella (infinita) di tutte le $f_i(x)$ del tipo:

x	= 0	1	2	3	4	5	6	7	8	. . . .
$f_0(x)$	= 0	1	1	0	1	0	0	0	1	. . . .
$f_1(x)$	= 0	1	0	1	1	0	0	1	0	. . . .
$f_2(x)$	= 1	0	1	1	0	1	1	0	1	. . . .
$f_3(x)$	= 1	1	1	0	0	0	1	1	0	. . . .
$f_4(x)$	= 0	0	0	1	1	0	1	0	1	. . . .
	.	.	.	.	.	.	.	.	.	. . . .

[1.4]

Consideriamo ora la funzione definita sui naturali:

$$g(x) = \begin{cases} 0, & \text{se } f_x(x) = 1, \\ 1, & \text{se } f_x(x) = 0. \end{cases} \qquad [1.5]$$

La g(x) appartiene a F, ma non può coincidere con alcuna delle $f_i(x)$ della tabella [1.4], poiché differisce da tutte quelle funzioni nei valori posti sulla diagonale principale della tabella. Per qualunque numerazione scelta vi è quindi sempre (almeno) una funzione esclusa; ovvero F non è numerabile.

Osservando poi che invece della diagonale principale potremmo considerare un'altra linea arbitraria che attraversi la tabella [1.4] toccando tutte le righe e tutte le colonne esattamente una volta, e definire una funzione che assuma in ogni punto valore opposto a quello incontrato dalla linea, scopriamo che vi sono infinite funzioni di F escluse da qualsiasi numerazione. E se proseguissimo in questa direzione scopriremmo che vi sono *innumerevoli* (cioè *innumerabili*) funzioni escluse.

Dall'argomentazione precedente seguono alcune importanti riflessioni. Poiché le rappresentazioni che possiamo costruire con alfabeti finiti sono, per quanto già osservato, numerabili, esse sono *di meno* delle funzioni matematiche. La famosa Biblioteca di Babele ideata da Borges contiene la traduzione di tutti i Vangeli gnostici, veri e ideali, in infinite lingue mai esistite (e contiene anche le grammatiche e i dizionari di tali lingue); contiene i commenti alle traduzioni, i commenti dei commenti, e i commenti dei commenti dei commenti; ma non contiene che una frazione inapprezzabilmente piccola delle funzioni di F, perché esse sono troppe per poter essere rappresentate.

Si potrebbe a questo punto pensare di eludere la difficoltà cercando di descrivere ogni funzione con una regola di rappresentazione di lunghezza finita, come la [1.3], anziché con una rappresentazione infinita come la [1.2]. Ma tale tentativo si arresta immediatamente davanti alla considerazione che le regole devono essere anch'esse rappresentate con sequenze di simboli (come per l'appunto la [1.3]), e quindi anche l'insieme di tutte le regole possibili è numerabile. Scopriamo così anche che esistono *meno regole di rappresentazione che funzioni*. Considerando la regola come un meccanismo di calcolo per la funzione, dobbiamo concludere che esistono *funzioni non calcolabili*; su questo importantissimo argomento torneremo dopo aver definito cosa si intende per calcolo.

Nella discussione finora svolta abbiamo ammesso di rappresentare le entità studiate mediante sequenze indefinite di simboli; né cambierebbe la ricchezza della rappresentazione se utilizzassimo gli stessi simboli per costruire configurazioni bidimensionali o pluridimensionali anziché sequenze. Si osservi infatti che i punti a coordinate intere in un piano cartesiano sono numerabili, lungo un percorso a "spirale" che parte dall'origine, cui è associato lo zero:

6	7	8	9
5	0	1	.
4	3	2	.

Di conseguenza una configurazione bidimensionale di simboli può essere trasformata in una sequenza, ordinandone gli elementi secondo le posizioni che essi assumono nel piano (e inserendovi un nuovo simbolo a indicare posizioni eventualmente vuote).

Un analogo ragionamento vale per spazi cartesiani n-dimensionali.

Notiamo ora che se l'alfabeto contiene k simboli, e si impiegano sequenze finite di n simboli, si possono rappresentare al massimo $e = k^n$ entità (per esempio con tre cifre decimali si rappresentano i 10^3 numeri decimali da 000 a 999; quante diverse automobili si possono immatricolare con le targhe di una provincia italiana?) Invertendo la funzione precedente possiamo dire che per rappresentare e entità occorrono sequenze di $n = \log_k e$ simboli. Poiché nel seguito ci riferiremo quasi sempre a insiemi finiti, questa semplice relazione dovrà essere tenuta sempre presente.

1.2 Il concetto di algoritmo

La parola algoritmo, creata nel latino medievale per assonanza con il nome del matematico persiano Al-Khuwarizmi, indica informalmente la specificazione dei *passi elementari* che un esecutore deve compiere per giungere alla soluzione di un problema. Poiché il problema può applicarsi a casi diversi, un algoritmo è interessante se viene formulato per *dati* arbitrari, e fornisce risultato corretto per qualsiasi insieme coerente di dati gli sia presentato (per esempio un algoritmo per calcolare il prodotto dovrà essere applicabile a *qualunque* coppia di numeri).

Probabilmente i più antichi algoritmi non banali conosciuti oggi, furono registrati dallo scriba egizio Ahmes, nel 1650 a.C., nell'omonimo papiro conservato al British Museum. Tra questi spicca l'algoritmo di moltiplicazione di interi, che riportiamo qui sotto per introdurre i primi concetti dell'algoritmica: sorprendentemente esso coincide con il nostro usuale algoritmo di moltiplicazione, se gli operandi sono rappresentati in base 2; ma gli egizi non conoscevano tale rappresentazione!

Dati due interi non negativi arbitrari A e B, il loro prodotto P si costruisce nel modo seguente:

```
algoritmo MOLTIPLICAZIONE (dati A e B, risultato P):

    poni P = 0;

    finché A ≠ 0 ripeti la sequenza:

        se A è dispari allora  addiziona B a P;

        dimezza A trascurando il resto;

        raddoppia B.                                        [1.6]
```

Per esempio, per A=45 e B=14 si costruisce il prodotto P=630 come:

P:	0	14	14	70	182	182	630
A:	45	22	11	5	2	1	0
B:	14	28	56	112	224	448	896

[1.7]

L'algoritmo [1.6] è stato scritto con costrutti caratteristici dei linguaggi programmativi, che saranno studiati altrove. Il loro impiego ha diverse motivazioni: la prima, ovvia, è che questo è il mezzo espositivo caratteristico dell'informatica, ed è quindi obbligato in questa sede; la seconda è che tali costrutti consentono concisione e chiarezza difficilmente ottenibile altrimenti; la terza e più importante motivazione è che essi, come chiariremo nel seguito, hanno carattere di universalità. (Ricordiamo incidentalmente che un algoritmo espresso in ogni sua parte mediante un linguaggio di programmazione prende il nome di *programma*).

Le parole scritte in grassetto in [1.6] indicano costrutti chiave. L'**algoritmo**, di cui MOLTIPLICAZIONE è il nome scelto arbitrariamente, indica un meccanismo di calcolo definito una volta per tutte e individuabile per nome; esso può essere utilizzato su **dati** arbitrari (purché, in questo caso, interi non negativi) che acquistano *all'interno dell'algoritmo* i nomi locali di A e B, e genera come **risultato** il prodotto dei dati con il nome P.

Le frasi che descrivono l'algoritmo debbono essere interpretate ed eseguite in *sequenza* nell'ordine in cui si leggono, a meno che alcune tra le stesse frasi non comandino di mutare tale ordine. Vi sono frasi che specificano operazioni aritmetiche come l'addizione, la divisione e la moltiplicazione per 2: esse non rivestono particolare interesse nel nostro esame, perché si assume che possono essere eseguite come azioni elementari (con un qualunque abaco fisico o mentale) dall'esecutore dell'algoritmo. Più interessanti, tra le frasi meramente esecutive, sono le *assegnazioni di valore*, che nell'algoritmo appaiono esplicitamente ("**poni** P=0"), o implicitamente ("addiziona B a P", cioè assegna a P il nuovo valore ottenuto addizionando il valore corrente di P a quello di B; "dimezza A", e "raddoppia B", cioè assegna ad A e a B nuovi valori). La specificazione di assegnazioni implica che l'algoritmo incida su un mondo circostante, di cui esso varia lo *stato*, qui identificato da una corrispondenza tra *nomi* di oggetti e loro *valori*. Lo studio di queste implicazioni è cruciale nella definizione semantica dei linguaggi programmativi.

I costrutti più importanti sono quelli destinati al *controllo* del flusso di calcolo, in funzione di particolari condizioni che si possono verificare durante il calcolo stesso. I principali sono la *iterazione* di una sequenza di passi, e la *esecuzione condizionata* di un passo, entrambi presenti nell'algoritmo [1.6]. L'iterazione è qui realizzata con il costrutto:

```
finché condizione ripeti una sequenza di passi   [1.8]
```

ove la ripetizione si arresta quando la *condizione* non è più verificata: è dunque responsabilità di chi progetta l'algoritmo di inserire tra i passi della sequenza un meccanismo che consenta alla condizione di essere verificata solo durante le iterazioni richieste (nell'algoritmo [1.6] la frase "dimezza A trascurando il resto" garantisce che la condizione "A ≠ 0" sia violata al momento opportuno, determinando la fine delle iterazioni).

L'esecuzione condizionata si realizza in generale come:

```
se condizione allora passo1 altrimenti passo2   [1.9]
```

che comanda l'esecuzione in alternativa del passo1, o del passo2, a seconda che la condizione sia, o non sia, verificata (nell'algoritmo [1.6] tale costrutto è privo della parte **"altrimenti"**, e si interpreta come se passo2 fosse vuoto). I costrutti di iterazione ed esecuzione condizionata sono fondamentali perché, come avremo modo di mostrare in seguito, sono sufficienti a specificare la struttura di controllo di qualsiasi algoritmo.

Un algoritmo A per la risoluzione di un problema P_A potrà *richiamare* al suo interno un algoritmo B per la risoluzione di P_B, se occorre risolvere P_B come parte di P_A. In tal caso l'esecuzione di A si arresta temporaneamente per permettere l'esecuzione di B, e viene poi ripresa al termine di quest'ultima, non diversamente da come un inciso si inserisce nel flusso di un discorso, o una espressione tra parentesi si inserisce in una formula algebrica. E naturalmente l'esecuzione di B può richiedere quella di un altro algoritmo, e così via, con un numero arbitrario di richiami di diversi algoritmi, uno interno all'altro. Questo meccanismo obbliga chi esegue la computazione a ricordare in ogni istante lo stato del calcolo per tutte le esecuzioni di algoritmi lasciate in sospeso: compito in genere assai oneroso, che viene eseguito automaticamente dal calcolatore se gli algoritmi sono realizzati sotto forma di programmi.

Il meccanismo suddetto assume importanza particolare se si consente a un algoritmo di richiamare sé stesso. Si dice in questo caso che l'algoritmo è **ricorsivo**, e capirne il funzionamento richiede un po' di studio se non si ha esperienza in proposito. Ogni richiamo dell'algoritmo lascia aperta una computazione su alcuni dati, e ripropone la stessa computazione su dati che hanno lo stesso nome dei precedenti ma valori nuovamente definiti: gli stessi dati possono avere quindi molte vite (cioè valori) parallele e indipendenti, in corrispondenza ai diversi richiami uno interno all'altro. Quando i dati di un richiamo raggiungono opportuni valori, una *clausola di chiusura* contenuta nell'algoritmo provoca la terminazione della computazione relativa a tale richiamo, e ristabilisce la computazione al livello precedente recuperando i valori dei dati per tale livello. La formulazione della clausola di chiusura è di completa responsabilità di chi costruisce l'algoritmo, ed è cruciale per assicurare la corretta terminazione di questo.

L'algoritmo egizio di moltiplicazione si può riformulare in modo ricorsivo, osservando che il prodotto P = A × B si può calcolare come segue ([A/2] indica la parte intera di A/2):

$$P = \begin{cases} [A/2] \times 2B, & \text{se A è pari;} \\ B + [A/2] \times 2B, & \text{se A è dispari.} \end{cases} \qquad [1.10]$$

La relazione [1.10] mostra come il prodotto tra A e B si possa calcolare attraverso il prodotto tra [A/2] e 2B (per inciso la [1.10] fornisce un'altra prova di validità dell'algoritmo [1.6]). Tale relazione si trasferisce allora direttamente nel nuovo algoritmo ricorsivo MOLTIPLICAZIONE, cui deve essere premesso l'azzeramento di P:


```
algoritmo-ricorsivo MOLTIPLICAZIONE (dati A e B):

    se A = 0 allora arrestati;

    se A è dispari allora  addiziona B a P;

    richiama MOLTIPLICAZIONE (dati [A/2] e 2B).        [1.11]
```

L'iterazione contenuta nell'algoritmo [1.6] è ora realizzata con richiami ricorsivi all'algoritmo, relativi di volta in volta al dato A dimezzato e al dato B raddoppiato, finché si verifica la condizione di chiusura A=0 che arresta la computazione.

Si può facilmente osservare che gli algoritmi [1.6] e [1.11] calcolano A×B attraverso le stesse addizioni, eseguite nello stesso ordine. Vedremo nel seguito che questo non sempre si verifica, poiché le formulazioni iterativa e ricorsiva di un algoritmo possono essere spontaneamente poste in modi molto diversi.

1.3 La tesi di Church

L'introduzione completamente informale agli algoritmi contenuta nel paragrafo precedente è, come ovvio, del tutto insufficiente a definire il concetto di computazione. La necessità di tale definizione emerse alla fine del secolo scorso, sotto la spinta di dimostrare risultati *negativi*: l'esistenza di funzioni non calcolabili (par. 1.1), provata teoricamente, doveva infatti essere corredata da una rigorosa definizione del concetto di calcolabilità. A tale definizione giunsero indipendentemente Turing e Post attorno al 1936, Kolmogorov e altri successivamente. Come vedremo i loro modelli di calcolo sono però tutti equivalenti, e ci limiteremo a presentare quello ideato da Turing come il più denso dei caratteri tipici dell'informatica.

La *macchina di Turing* è un meccanismo ipotetico dotato di una unità di controllo che può assumere diversi *stati* appartenenti a un insieme finito S, e che comanda una testina di lettura e scrittura che scorre su un *nastro* infinito. Sul nastro è inizialmente registrata una sequenza finita di simboli tratti da un alfabeto finito A: la macchina *legge* un simbolo alla volta, e, in funzione del simbolo letto a_l e dello stato corrente s_c, *scrive* sul nastro un nuovo simbolo a_s al posto del precedente, *muove* la testina sul nastro di una posizione a destra o a sinistra, e assume un *nuovo stato* s_n. La macchina consiste formalmente in un insieme finito Q di quintuple del tipo:

$$\{a_l,\ s_c,\ a_s,\ m,\ s_n\} \in Q$$

con il significato:

$a_1 \in A$ è il simbolo letto dal nastro;

$s_c \in S$ è lo stato corrente;

$a_s \in A$ è il nuovo simbolo scritto sul nastro;

$m \in$ {destra,sinistra} è il movimento della testina;

$s_p \in S$ è il nuovo stato.

Ammettendo che tutte le quintuple di Q inizino con una diversa coppia $a_1.s_c$, la macchina decide ***deterministicamente*** le azioni da eseguire a ogni passo, e si arresta in corrispondenza alle coppie $a_1.s_c$ che non appaiono in alcuna quintupla.

Se interpretiamo la sequenza che appare inizialmente sul nastro come la rappresentazione dei dati di un problema P, e la sequenza che risulta all'arresto della macchina come la rappresentazione del risultato, possiamo affermare che la macchina di Turing ***è un algoritmo per risolvere P.*** Il problema più serio è che la macchina si arresti per ogni significativa sequenza di ingresso: se ciò avviene, o più in generale se esiste una macchina per cui ciò avviene, diremo che il problema P è ***calcolabile*** (nel senso di Turing). Notando poi che, in base ai meccanismi di rappresentazione discussi nel par.1.1, le sequenze di simboli iniziale e finale sono interpretabili come interi, possiamo considerare la risoluzione di un problema arbitrario come il calcolo di una funzione da interi su interi: tale funzione ***è calcolabile*** se lo è il corrispondente problema.

La risoluzione di problemi anche molto semplici sulla macchina di Turing richiede in genere lunghissime e noiosissime serie di operazioni. Per esempio la macchina costruita sugli insiemi di simboli e stati:

$A = \{a,b,t\}$; $S = \{s_0,s_1,s_2\}$;

e definita dall'insieme di quintuple:

t, s_0, t, destra, s_1
a, s_1, a, destra, s_1
b, s_1, a, destra, s_1
t, s_1, t, sinistra, s_2
a, s_2, a, sinistra, s_2

scandisce sul nastro sequenze arbitrarie di simboli *a* e *b* delimitate a sinistra e a destra dal simbolo terminatore *t*, a partire dal terminatore di sinistra e dallo stato iniziale s_0, sostituendo tutti i *b* con *a* e riportandosi sul terminatore iniziale. Per

esempio essa trasforma la sequenza *taababbbt* in *taaaaaaat*. Si noti che quando la macchina legge *a* scrive ancora *a* (cioè non altera il simbolo letto: seconda e ultima quintupla); essa si riporta sul terminatore di sinistra nello stato s_2, e si arresta in questa posizione poiché non è definita alcuna quintupla che inizi con $t;s_2$. Per la discussione di esempi di computazione più significativi e complessi rimandiamo alla letteratura specializzata.

Se la potenza espressiva della macchina di Turing è estremamente modesta, la sua potenza computazionale è enorme. E' infatti universalmente accettata la *Tesi di Church*, in base alla quale *tutte la ragionevoli definizioni di algoritmo sono equivalenti.* Questa tesi implica che tutti i modelli di computazione, tra cui la macchina di Turing, si equivalgono nella possibilità di risolvere problemi, pur se operano con diversa efficienza. E poiché qualunque calcolatore può ovviamente simulare la macchina di Turing, purché gli venga fornita una memoria sufficientemente grande per contenere tutto ciò che viene scritto sul nastro (infinito) della macchina durante la computazione, se ne conclude che tutti i (ragionevoli) calcolatori possono risolvere gli stessi problemi, ovvero possono simularsi a vicenda. Da quanto affermato risulta legittimo esprimere gli algoritmi con i costrutti propri dei linguaggi di programmazione.

1.4 Numerabilità degli algoritmi e problemi non calcolabili

Se descriviamo gli algoritmi come macchine di Turing, o come programmi per un calcolatore, possiamo subito osservare che la loro rappresentazione è costituita da sequenze finite di simboli. *Gli algoritmi sono dunque numerabili.* Ma poiché non sono numerabili le funzioni, quindi i problemi, esisteranno funzioni o problemi per cui non esiste algoritmo di calcolo. E non deve trattarsi di problemi fittizi o mal posti, perché non vi è ragione di ritenere fittizia o mal posta una funzione anziché un'altra.

In effetti i problemi che ci si presentano spontaneamente sono tutti calcolabili, né è stato facile individuare un problema che non lo fosse. Il primo fu scoperto ancora una volta da Turing in relazione alla sua macchina: si tratta del famoso *problema dell'arresto*, che esporremo tra breve in termini programmativi per riproporne la sostanza in modo semplice. Il problema dell'arresto è posto in forma ***decisionale***, cioè richiede come risultato una risposta affermativa o negativa (corrisponde quindi a una funzione dagli interi su {0,1}): per questi problemi la calcolabilità è in genere detta *decidibilità*.

Il problema dell'arresto considera algoritmi che indagano sulle proprietà di altri algoritmi: questi ultimi vengono cioè trattati come *dati* per i primi. Sulla base di quanto esposto fino a questo punto ciò deve apparire del tutto legittimo, poiché gli algoritmi sono descritti con sequenze di simboli che possono essere tratti dallo stesso alfabeto dei dati. Così una macchina di Turing M può essere rappresentata, assieme ai suoi dati D, sul nastro di un'altra macchina di Turing U che simuli la computazione M(D) generando i risultati per M. Una macchina U che simuli il funzionamento di qualsiasi altra macchina è detta ***universale***, e può essere definita con tecniche standard: essa si pone, nella teoria delle macchine di Turing, come un calcolatore che può eseguire qualsiasi algoritmo. Ovviamente, stabilita una numerazione per le macchine di Turing, anche la U apparirà tra queste, e potrà adoperare la sua stessa rappresentazione

come dato calcolando U(U).

Così come la macchina U accetta la descrizione di una macchina M come dato, un algoritmo A comunque formulato può operare sulla rappresentazione di un altro algoritmo B, ove tale rappresentazione può consistere nel numero distintivo di B in una numerazione scelta. In particolare può aver senso calcolare A(A).

Il problema dell'arresto può essere posto nei termini seguenti:

Presi ad arbitrio un algoritmo A e i suoi dati D, decidere (in tempo finito) se la computazione di A su D termina o no.

Se il problema fosse calcolabile, esisterebbe un algoritmo di decisione ARRESTO che, presi A e D come dati, produrrebbe in tempo finito le risposte:

```
ARRESTO(A,D) risponde sì,   se A(D) si arresta;

ARRESTO(A,D) risponde no,   se A(D) non si arresta.      [1.12]
```

Si noti che ARRESTO non può semplicemente consistere in un algoritmo che simuli la computazione A(D), perché, se questa non si arresta, ARRESTO non sarebbe in grado di rispondere "no" in tempo finito.

Per quanto osservato in precedenza, ha senso considerare la computazione A(A), e quindi ARRESTO(A,A); la [1.12] ci consente quindi di affermare:

```
ARRESTO(A,A) risponde sì se e solo se A(A) si arresta.   [1.13]
```

Se esistesse l'algoritmo ARRESTO, esisterebbe anche il seguente algoritmo PARADOSSO, che opera su un algoritmo arbitrario A come dato, fa uso di ARRESTO e di costrutti ovviamente legittimi:

```
algoritmo PARADOSSO(A):
    finché ARRESTO(A,A) risponde sì
        ripeti una frase inutile che non alteri A.
```

Un'ispezione diretta di questo algoritmo mostra che:

```
PARADOSSO(A) si arresta
        se e solo se ARRESTO(A,A) risponde no.      [1.14]
```

Cosa succede ora se calcoliamo PARADOSSO(PARADOSSO)? Sostituendo il dato PARADOSSO ad A nelle [1.13] e [1.14] otteniamo due affermazioni contraddittorie. L'unico modo per risolvere questa contraddizione è che PARADOSSO non esista, cioè che non esista ARRESTO. Ovvero *il problema dell'arresto è indecidibile.*

L'indecidibilità del problema dell'arresto ha una importanza pratica diretta in informatica, poiché mostra come non sia possibile costruire un dispositivo (una

macchina, un programma) atto a bloccare, all'ingresso di un sistema di calcolo, i programmi (errati) che "entreranno in ciclo" durante l'esecuzione. Ciò ovviamente non vuol dire che non si possa prevedere tale erroneo comportamento in un programma particolare prima di eseguirlo, ma non esiste metodo generale applicabile *a qualsiasi programma.*

E' comunque parere di chi scrive che l'indecidibilità di questo problema debba apparire perfettamente naturale ai matematici, perché l'esistenza dell'algoritmo ARRESTO fornirebbe un mezzo obiettivamente troppo potente per dimostrare senza sforzo congetture ancora aperte sugli interi (come per esempio l'ultimo "teorema" di Fermat: $a^n+b^n=c^n$, per n>2). La seguente spiegazione di questo fatto è elementare ma non banale.

Affermare la verità di una formula F su n variabili intere equivale a negare l'esistenza di un insieme di valori delle variabili che soddisfi la negazione della F (il teorema di Fermat equivale a negare l'esistenza di quattro valori interi per a, b, c, n, con n>2, per cui $a^n+b^n=c^n$). Si può allora in genere scrivere un algoritmo TEST che generi al suo interno tutte le infinite combinazioni di valori delle variabili, una dopo l'altra, e calcoli la negazione di F per ciascuna di tali combinazioni, arrestandosi in tempo finito, anche se a priori non noto, appena la negazione di F fosse soddisfatta (la formulazione di TEST per il teorema di Fermat è lasciata al lettore). TEST ovviamente non aiuta nella dimostrazione della F, perché se tale formula fosse vera TEST non si arresterebbe, cioè la prova durerebbe in eterno. Se però esistesse l'algoritmo ARRESTO, esso potrebbe essere impiegato per indicare in tempo finito se la computazione di TEST si arresta o meno, e quindi se la F è rispettivamente falsa o vera.

Come abbiamo già osservato il problema dell'arresto non è che uno degli *innumerabili* problemi non decidibili che *devono* esistere. Di fatto ne sono stati scoperti parecchi, in genere nell'ambito della logica, con prove di indecidibilità che si riconducono di norma a quella dell'arresto. Tra tutti citiamo, come esempio, il problema delle *equazioni diofantee*: data una arbitraria equazione algebrica E in un arbitrario numero di incognite, è in generale indecidibile il quesito se la E è soddisfatta da un insieme di valori interi delle incognite.

Riferimenti bibliografici

J.N.Crossley e altri. Che cos'è la logica matematica? Boringhieri, Torino 1976.
Breve e semplice testo in italiano, fornisce un'ottima introduzione alla logica e alla computabilità.

F.Luccio. La struttura degli algoritmi. Boringhieri, Torino 1982.
Concepito come testo universitario, dovrebbe essere di livello accessibile alle persone colte e volenterose. Qui segnalato come introduzione all'algoritmica.

M.D.Davis e E.J.Weyuker. Computability Complexity and Languages. Academic Press (Series in Computer Science and Applied Mathematics), 1983.
Testo serio e completo, è consigliabile tra tanti per un primo vero approfondimento delle materie di questo capitolo.