Come contano i roditori

(prima parte)

I castori sanno calcolare meglio degli esseri umani? C'è una funzione che solo i migliori roditori sembra che sappiano computare. Le difficoltà che nascono a cercare di imitarli. Ed è solo l'inizio…

Autori: Isabelita Antolini, Stefano Leonesi, Carlo Toffalori

1. Computabile ed incomputabile

A metà degli anni Trenta, studiosi di varie parti del mondo si preoccuparono di definire in maniera rigorosa il concetto di computabilità e di individuare così in modo preciso quali funzioni si possono calcolare e quali problemi si possono risolvere. Le motivazioni di questo comune interesse nascevano da alcuni rilevanti sviluppi avutisi proprio in quegli anni nella ricerca sulla Matematica e sui suoi fondamenti. Nel 1925, infatti, David Hilbert aveva formulato ufficialmente il suo *Programma*, volto a cercare per ogni settore della Matematica – per l'Aritmetica, la Geometria, la Teoria degli insiemi e così via – assiomi e regole di deduzione sulla cui base ogni proposizione potesse essere dimostrata vera o falsa ed ogni problema risolto in modo definito. Hilbert aveva anche proposto quello che originariamente in tedesco si chiama *Entscheidungsproblem* e che possiamo tradurre *Problema di Decisione*: vi si considera la Logica del primo ordine e vi si chiede un procedimento che sappia riconoscerne gli enunciati (dimostrabili) veri distinguendoli dagli altri.
Ma già nel 1931 Kurt Gödel, con i suoi teoremi di incompletezza, aveva dato un colpo esiziale al programma hilbertiano, mostrando sostanzialmente l'incapacità "umana" di cogliere i veri fondamenti – anche solo dell'Aritmetica dei numeri naturali – e l'esistenza, in questo ambito, rispetto ad ogni possibile sistema di assiomi, di proposizioni "indecidibili" (non dimostrabili né vere né false; talora vere ma non dimostrabili tali). La ricerca, poi, di un procedimento adeguato per l'*Entscheidungsproblem* manifestava forti difficoltà, tali da insinuare un qualche pessimismo sul suo successo: nella Logica del primo ordine, ci sono assiomi e regole di deduzione per cui gli enunciati dimostrabili veri sono esattamente quelli veri in ogni struttura. Ma un algoritmo capace di riconoscerli sembrava ostico da ottenersi.
Del resto, già da qualche anno, esistevano questioni dal sapore meno filosofico e metamatematico e più legato alla Matematica del *"far di conto"* che si rivelavano altrettanto difficili e complicate. Era questo il caso del *Decimo Problema* che lo stesso Hilbert aveva inserito nella sua famosa lista di questioni aperte nel lontano 1900, quello che chiede di determinare un procedimento che sappia stabilire, per ogni polinomio a coefficienti interi (di qualunque grado e in un qualunque numero di indeterminate), se ammette o no radici intere.
In tutti questi casi, sull'onda dei risultati di Gödel, si poteva anche prevedere una soluzione assolutamente negativa, tale da escludere l'esistenza di un algoritmo come richiesto. D'altra parte, una risposta di tal genere richiedeva (e richiede) il chiarimento di un'ovvia questione preliminare. Infatti, per convincere che un procedimento di soluzione c'è, basta produrlo. Per persuadere, invece, che l'algoritmo non esiste, non è sufficiente esibire qualche tentativo e mostrarne il fallimento; bisogna piuttosto chiarire a priori che cosa si intende, appunto, per *"algoritmo"*, stabilire cioè in maniera chiara e condivisa quali problemi si intendono dotati di procedimento di soluzione e quali funzioni si intendono dotate di algoritmo di calcolo e poi, conseguentemente, provare in modo rigoroso che di tali problemi l'*Entscheidungsproblem* o la Decima Questione di Hilbert non fanno parte.

Questa era, dunque, la ragione del largo interesse cui si accennava in apertura. Varie possibili risposte si accumularono in quegli anni (da Church, Turing, Kleene, Gödel, ed altri) spesso originate da idee e sensibilità assai diverse e sviluppate in maniera indipendente, eppure in definitiva tutte tra loro equivalenti e dunque convergenti allo stesso risultato. Tra questi approcci, quello di Turing del 1936, [1] sembra oggi il più famoso, quello a cui più facilmente si fa riferimento. Accenniamolo brevemente.

Dobbiamo dunque definire in modo rigoroso che cosa si intende per funzione computabile o per problema decidibile. Il primo punto da chiarire è *chi* è delegato a computare, o decidere. Oggi, nel 2000, la risposta sarebbe scontata ed immediata: un *calcolatore*. Ma ai tempi in cui Turing lavorava, negli anni Trenta, i moderni computers erano ben lungi dall'essere immaginati e prodotti. Anzi, il merito di Turing, e quello dei suoi colleghi sopra citati, fu proprio quello di anticipare gli odierni calcolatori, definendoli astrattamente in modo rigoroso. Sediamoci comunque di fronte ad un calcolatore dell'ultimissima generazione, con il suo schermo e la sua tastiera. Vogliamo computarvi la nostra funzione o decidere le istanze del nostro problema. Ovviamente, la tastiera deve contenere tutti i caratteri necessari per impostare la questione. Se però vogliamo mantenere un contesto generale e uniforme, buono per tutte le situazioni, e magari semplificare per tal via la nostra esposizione, possiamo limitarci ad un solo simbolo 1. A chi trovasse questa soluzione troppo rozza e brutale, potremmo rispondere che essa ci permette comunque di rappresentare i numeri naturali:

0, 1, 2, 3, 4, 5, …

come stringhe:

1, 11, 111, 1111, 11111, 111111, …

Una volta poi che siamo riusciti a rappresentare i naturali, nessuna situazione discreta e sostanzialmente finitaria (come quella dell'*Entscheidungsproblem*, o del Decimo Problema di Hilbert) ci è preclusa. Anzi, ognuna può essere catturata con un po' di pazienza e con opportune codifiche, assegnando ad ogni nuovo simbolo un numero naturale come codice, allo stesso modo in cui – in una biblioteca – i libri accumulati negli scaffali si riconoscono grazie alla etichetta e al numero che compare sulla loro costola che ne indica la collocazione. Se poi vogliamo rappresentare non un singolo naturale, ma una coppia o una terna o una *k*-upla di naturali, possiamo facilmente esprimerla con il solo simbolo 1, scrivendo all'inizio il primo naturale, e poi a seguire il secondo, l'eventuale terzo, e così via, separando ogni nuovo numero dalla sequenza dei precedenti con uno spazio vuoto. Ad esempio, la coppia (2,3) si può scrivere:

111 1111.

Dunque, ammettiamo di poter scrivere sul nostro schermo soltanto l'elemento 1. Conveniamo, semmai per semplicità, di riempire tutti gli spazi eventualmente vuoti nella nostra computazione con il simbolo 0. Magari, riserviamoci la libertà di poter stampare 1 o 0 in più maniere diverse, ad esempio con colori differenti (nero, blu, rosso, e così via) o, se preferiamo, con caratteri diversi (in neretto, o in corsivo, o in altro possibile modo).

Da un punto di vista ufficiale, chiamiamo *stati* queste possibili opzioni di scrittura; indichiamole *q(0), q(1), q(2), ...* e lasciamoci la libertà di scegliere, per ogni problema da decidere o funzione da computare, quanti stati vogliamo per compiere il nostro lavoro (ma comunque un numero finito *q(0), q(1), ... q(n)* per un qualche naturale *n*).

A questo punto, affrontiamo il nostro obiettivo:

- computare una funzione *f*, ad esempio dall'insieme ***N*** dei naturali, o da una qualche sua potenza cartesiana $\boldsymbol{N}^k$, a ***N*** stesso;
- oppure decidere se un dato elemento *x* appartiene o no ad un certo insieme *X*, ad esempio se un polinomio a coefficienti interi fa parte o no dell'insieme dei polinomi a radici intere (come nel Decimo Problema di Hilbert) o se un dato enunciato della Logica del primo ordine è o no

incluso nell'insieme degli enunciati dimostrabili veri; in questo secondo caso, possiamo comunque immaginare, dopo opportune codifiche, che l'insieme X da controllare sia un sottoinsieme di $\boldsymbol{N}$ e che l'elemento x di cui valutare l'appartenenza a X sia anch'esso un naturale.

In entrambi i casi, troviamo sullo schermo un'istanza x del nostro problema, dunque una stringa di 1, eventualmente interrotta da qualche 0, circondata esternamente sia a destra che a sinistra da sequenze illimitate di 0, e dobbiamo svolgere la computazione in modo da calcolare *f(x)*, nel primo caso, o da stabilire se $x \in X$ o no (nel secondo). In realtà, ci basta saper maneggiare il primo caso perché la appartenenza di x a X si può decidere calcolando l'immagine di x nella funzione caratteristica f_X di X (quella che vale 1 negli elementi di X e 0 negli altri).

Ci serve dunque un programma per svolgere queste operazioni. Il punto fondamentale della definizione di Turing è proprio nella descrizione di questo programma, che viene immaginato come un insieme finito di istruzioni, ciascuna delle quali considera la coppia formata da:

- il simbolo 0 o 1 che è in esame sullo schermo;
- lo stato *q(0)* o *q(1)* … o *q(n)*, con cui lo si sta osservando,

e che conseguentemente indica:

- che cosa scrivere sullo schermo al posto di quel simbolo (ancora 0 o 1);
- dove spostarsi, se a destra (*right*, *R*) o a sinistra (*left*, *L*), comunque soltanto di un passo;
- in quale stato, *q(0)*, o *q(1)*, …, o *q(n)*, osservare il nuovo simbolo in esame.

Per ogni coppia, esiste al più un'istruzione che la riguarda. Se l'istruzione c'è, la si esegue e si continua. Se l'istruzione non c'è, ci si ferma e si accetta la stringa che compare sullo schermo come *output* della computazione. Così, nel primo caso sopra descritto, ci aspettiamo la risposta *f(x)*; nel secondo caso, in particolare, $f_X(x)$.

Chiamiamo *macchina di Turing* sugli $n+1$ stati *q(0)*, *q(1)*, …, *q(n)* un insieme finito di istruzioni come appena descritto. La definizione si può porre in termini matematicamente rigorosi, dicendo che una macchina di Turing a $n+1$ stati è una funzione M da un insieme di coppie in $\{0, 1\} \times \{q(0), q(1), ..., q(n)\}$ a terne in $\{0, 1\} \times \{L, R\} \times \{q(0), q(1), ..., q(n)\}$. M può anche essere *parziale*, cioè definita solo su alcune coppie. Anzi, ci aspettiamo che ci siano coppie fuori del dominio di M perché solo l'assenza di istruzioni porta M a concludere la sua computazione ed esprimere un *output*.

Una macchina di Turing M può lavorare su qualunque stringa finita di 1, eventualmente interrotta da qualche 0 (e circondata comunque da 0). Si conviene che, al primo passo, si muova sempre nello stato *q(0)*. Se conclude la sua computazione, si dice che *M converge* su quel dato *input*; altrimenti, se la computazione prosegue senza termine, si dice che M *diverge*. Una singola istruzione – ad esempio quella che alla coppia (1, *q(2)*) associa la terna (0, *R*, *q(0)*) – si rappresenta, in modo conciso ed espressivo, come:

$$(1, q(2)) \rightarrow (0, R, q(0)).$$

Come si vede, si tratta di un modello assai semplice ed elementare, lontano dalle sofisticazioni e dalle complessità delle moderne programmazioni. A chi lo trovasse troppo banale, dobbiamo ricordare ancora una volta che le idee di Turing precorrono di molti anni (ed anzi inaugurano) i moderni sviluppi dell'Informatica.

Ma torniamo alla questione di definire la computabilità. C'è qui un'ipotesi di lavoro, basata sulla nozione di macchina di Turing, proposta da Turing nel 1936 e chiamata, appunto, *Tesi di Turing*. Sostiene che una funzione f è computabile se e solo se c'è una macchina di Turing che la computa e dunque un programma come sopra descritto che, per ogni possibile x, produce *f(x)* in un numero finito di passi. Analoga proposta si fa per i problemi di decisione. In questo modo, il concetto di computabilità viene caratterizzato in modo immediato e semplice, visto che la nozione di macchina di Turing ha una definizione matematicamente tanto rigorosa quanto accessibile.

Possiamo semmai chiederci quanto sia plausibile l'affermazione di Turing. Oggi in particolare, di fronte alla potenza dei modernissimi *computers* e alle loro enormi potenzialità di calcolo, possiamo

ragionevolmente dubitare della sua attendibilità. Ebbene, la *Tesi di Turing* resiste a tutte queste novità. Tutto ciò che finora si è potuto computare con i calcolatori si può effettivamente calcolare, con un po' di pazienza, con una macchina di Turing. Semmai gli strumenti moderni accelerano il tempo delle risposte ma non ne accrescono il raggio di attuazione: almeno fino ad oggi, non sono emersi esempi di funzioni e problemi praticamente computabili, ma non calcolabili con una macchina di Turing. Del resto, ove sorgessero, non vi sarebbe nessun dramma: la *Tesi di Turing* andrebbe aggiornata, riveduta e corretta (come d'altra parte già è successo a tante altre nobilissime teorie scientifiche quali, ad esempio, la Meccanica newtoniana).
Comunque, un ulteriore argomento che sembra avvalorare la plausibilità della affermazione di Turing è la sua dimostrata equivalenza (già sopra accennata) con altre proposte alternative ed indipendenti di definizione del concetto di computabilità, spesso di natura assai diversa da quella adesso descritta. La coincidenza di approcci tanto differenti può intendersi come un ulteriore argomento a loro sostegno. Una presentazione di questi definizioni alternative ed una bella discussione sulla *Tesi di Turing* si può trovare, ad esempio, nel libro [2]. Qui ci limitiamo ad accettare la visione di Turing e quindi l'idea che una funzione è computabile se e solo se c'è una macchina di Turing che la computa, semmai avvalorandola con un semplice esempio (che ci serve anche ad entrare in maggiore confidenza con lo spirito e la simbologia delle macchine di Turing): presentiamo infatti una macchina di Turing *M* che computa la (computabilissima) funzione di somma. Ci bastano 5 stati *q(0), q(1), q(2), q(3), q(4)* e le seguenti istruzioni (suddivise in righe, una per stato):

$$(0, q(0)) \to (1, R, q(1)), \qquad (1, q(0)) \to (1, R, q(0)),$$
$$(0, q(1)) \to (1, L, q(2)), \qquad (1, q(1)) \to (1, R, q(1)),$$
$$(0, q(2)) \to (0, L, q(2)), \qquad (1, q(2)) \to (0, L, q(3)),$$
$$(1, q(3)) \to (0, L, q(4)).$$

Vediamo come queste istruzioni permettono di confermare che 2 + 3 fa 5. Ricordiamo che 2 si rappresenta con 111, 3 è 1111 e 5 è 111111. La computazione di *M* avviene come segue: ogni riga ne rappresenta un passo, e ne contiene solo le cifre significative (quelle uguali ad 1 ed in aggiunta solo gli 0 intermedi o quello in esame); il simbolo con l'indice è, per ogni passo, quello considerato dalla macchina, e il suo indice ci precisa lo stato in cui *M* lo sta osservando.

1_0	1	1	0	1	1	1	1
1	1_0	1	0	1	1	1	1
1	1	1_0	0	1	1	1	1
1	1	1	0_0	1	1	1	1
1	1	1	1	1_1	1	1	1
1	1	1	1	1	1_1	1	1
1	1	1	1	1	1	1_1	1
1	1	1	1	1	1	1	1_1
1	1	1	1	1	1	1	1	0_1
1	1	1	1	1	1	1	1_2
1	1	1	1	1	1	1_3	
1	1	1	1	1	1_4	0	

Qualche parola di commento: l'istruzione su *(1, q(0))* fa ricopiare il primo addendo da sinistra verso destra, quella su *(0, q(0))* aggiunge un 1 intermedio e segnala il passaggio al secondo addendo grazie al cambio di stato da *q(0)* a *q(1)*. A questo punto, l'istruzione su *(1, q(1))* fa ricopiare, ancora da sinistra a destra, il secondo addendo e quella su *(0, q(1))* segnala il termine del secondo addendo con un nuovo passaggio di stato. Quello che vediamo a questo punto sullo schermo è una sequenza di 8 cifre uguali ad 1 (2 in più di quelle che codificano il risultato 5). Le istruzioni successive servono a cancellare i due simboli in eccesso. L'assenza di istruzioni su *(1, q(4))* fa poi fermare il procedimento sull'*output* richiesto. Come si vede, si tratta di algoritmo elaborato e lento ma comunque efficace e buono a gestire qualunque addizione (anche su altre coppie di addendi, come il lettore può facilmente verificare).
Sulla base della definizione di computabilità proposta da Turing, si può effettivamente provare che ci sono problemi non decidibili e funzioni non computabili (perché privi di macchine di Turing capaci di deciderli o calcolarli). Da un punto di vista teorico, un semplice argomento di Teoria degli insiemi, simile a quello usato da Cantor per provare che ci sono "più" reali che naturali, permette di dedurre facilmente l'esistenza di problemi senza soluzione e di funzioni senza algoritmo di calcolo (ove si ammetta la *Tesi di Turing*). Nel caso più generale delle funzioni da $\boldsymbol{N}$ a $\boldsymbol{N}$, procede nel modo che segue: fissato un naturale n, c'è ovviamente solo un numero finito di macchine di Turing a $n+1$ stati. Dunque, al variare di n, la totalità delle macchine di Turing costituisce un'infinità numerabile. Siccome funzioni distinte devono essere computate da macchine distinte, le funzioni computabili (cioè computate da macchine di Turing) sono anch'esse al più un'infinità numerabile. D'altra parte, l'insieme di tutte le funzioni da $\boldsymbol{N}$ a $\boldsymbol{N}$ ha la potenza del continuo. Quindi, funzioni non computabili da $\boldsymbol{N}$ a $\boldsymbol{N}$ esistono e, anzi, costituiscono la "stragrande maggioranza" degli esempi possibili. Analogamente si procede per i problemi di decisione.
Dedotta in questo modo l'esistenza di funzioni che non si calcolano e di problemi che non si risolvono, ci piacerebbe conoscere esempi espliciti di queste situazioni. L'argomento *à la Cantor* appena proposto, oltre a garantirli, ci assicura anche che sono assai più "numerosi" di quelli positivi di computabilità o decidibilità. D'altra parte, non è facile identificare questi ambiti negativi, privi di soluzione. Infatti, le funzioni più familiari, quelle che più facilmente vengono in mente (l'addizione, la moltiplicazione, e così via), sono tali proprio perché computabili. Lo stesso può dirsi dei problemi. Eppure si conoscono oggi famosi esempi negativi, almeno nel contesto dei problemi. Questa è, infatti, la sorte dell'*Entscheidungsproblem* (come provato dallo stesso Turing nel 1936 [1]) e del Decimo Problema di Hilbert (come dimostrato da un famoso teorema di Matijasevic del 1970, che coronò ricerche precedenti di Martin Davis, Julia Robinson ed altri). Se dunque cerchiamo casi espliciti di funzioni non calcolabili, ci basta citare la funzione caratteristica dell'insieme degli enunciati dimostrabili veri nella Logica del primo ordine (o meglio dei loro numeri di codice naturali) o quella dei polinomi a coefficienti interi dotati di radici intere. Esempi rigorosissimi e teoricamente convincenti, ma forse troppo astratti e poco alla mano. Preferiremmo conoscere una funzione da $\boldsymbol{N}$ a $\boldsymbol{N}$ per la quale si possa escludere qualunque algoritmo (cioè qualunque macchina di Turing) ma che sia accessibile e facile da definire.

2. Il gioco del castoro laborioso e la funzione Σ di Rado

In effetti, un esempio simpatico e accattivante di funzione non computabile (per quanto può essere simpatica ed accattivante una funzione matematica…) fu proposto dal matematico ungherese Tibor Rado nel 1962 [3] e da lui denotato come Σ (Sigma). Si tratta di una applicazione dai naturali ai naturali, che cresce così rapidamente da superare asintoticamente qualunque funzione computabile. Dunque non può essere computabile. Vediamone anzitutto la definizione: per ogni naturale n, dobbiamo introdurre $\Sigma(n)$. Per questo scopo, organizziamo un torneo nel modo che segue. Sono

ammesse a partecipare tutte le macchine di Turing *M* sull'alfabeto {1} negli $n+1$ stati *q(0)*, *q(1)*, …, *q(n)* che soddisfino due condizioni preliminari:

(i) *M* converge quando l'input è bianco, cioè formato da tutti 0;
(ii) *M* non ha istruzioni relative allo stato *q(n)*, dunque riguardanti le coppie *(0, q(n))* e *(1, q(n))*.

Ad ognuna di queste macchine assegniamo come punteggio il numero di 1 che essa riesce a scrivere nell'output della sua computazione sull'input bianco: $\Sigma(n)$ è il punteggio vincente (cioè massimo) in questo torneo.
Notiamo che c'è almeno una macchina concorrente, quella senza istruzioni, che dunque converge al primo passo sull'*input* bianco e ha risultato 0. Inoltre, il numero delle macchine partecipanti è finito (perché, più in generale, c'è soltanto una quantità finita di macchine di Turing sull'alfabeto {1} in $n+1$ stati). Quindi c'è un punteggio massimo e $\Sigma(n)$ è ben definita.
Per esemplificare la situazione, calcoliamo i valori di $\Sigma(n)$ per $n = 0,1,2$.

- Per $n = 0$, le macchine *M* ammesse a giocare sono quelle che hanno il solo stato *q(0)* e, per (ii), non hanno istruzioni su *q(0)* (dunque si arrestano immediatamente su qualunque *input*, compreso quello bianco). L'unica *M* che soddisfa queste condizioni è la macchina priva di istruzioni, che già sappiamo ottenere punteggio 0. Quindi $\Sigma(0) = 0$.

- Il calcolo di $\Sigma(1)$ è ancora ragionevolmente semplice. Stavolta le concorrenti *M* ammesse al gioco hanno due stati *q(0)* e *q(1)* e mancano di istruzioni per *q(1)*. Se però *M* ha un'istruzione del tipo $(0, q(0)) \rightarrow (..., ..., q(0))$ (cioè resta nello stato *q(0)* quando legge il simbolo bianco nello stato *q(0)*), allora, indipendentemente da quanto le viene ordinato di scrivere e da dove le viene indicato di spostarsi, *M* finisce per divergere sull'*input* bianco ed è dunque estromessa dal torneo. Infatti *M*, partendo nello stato *q(0)* sullo schermo bianco, scrive costantemente sul quadro in esame 0 o 1 (secondo quello che le detta la relativa istruzione), si sposta poi costantemente a destra o a sinistra, di nuovo secondo la sua istruzione, ma qui viene comunque ad incontrare un nuovo quadro contenente 0 e a leggerlo nello stato *q(0)*. Così, ripete indefinitamente questo comportamento. Dunque, tra le macchine *M* partecipanti al gioco, solo due eventualità sono ammesse: o *M* non ha istruzioni su *(0, q(0))* e dunque converge subito sull'*input* bianco ed ottiene punteggio 0 o *M* ha un'istruzione:
$$(0, q(0)) \rightarrow (0, ..., q(1))$$
oppure:
$$(0, q(0) \rightarrow (1, ..., q(1))$$
ed eseguendola sullo schermo bianco dapprima scrive 0 o 1 sul quadro in esame, dopo di che entra nello stato *q(1)* e, non avendo disposizioni che lo riguardano, si arresta, con punteggio 0 o 1 rispettivamente. In conclusione $\Sigma(1) = 1$.

- Già il calcolo di $\Sigma(2)$ comincia a complicarsi. Si riesce infatti a verificare che $\Sigma(2) = 4$, ma si devono affrontare e risolvere maggiori ostacoli computazionali. Né è difficile capirne i motivi. Contiamo infatti le macchine di Turing a 3 stati *q(0), q(1), q(2)* sul solo simbolo 1 che sono prive di istruzioni relative a *q(2)*, dunque sono potenziali partecipanti al gioco di $\Sigma(2)$ (in quanto soddisfano almeno il requisito (ii)). Esse corrispondono alle funzioni che vanno da coppie che hanno 0 o 1 come prima componente e *q(0)* o *q(1)* (ma non *q(2)*) come seconda e giungono a terne che hanno ancora 0 o 1 come prima componente, *L* o *R* come seconda e ammettono stavolta *q(0)*, *q(1)* o anche *q(2)* come terza componente. Restano così coinvolte $2 \times 2 = 4$ coppie e $2 \times 2 \times 3 = 12$ terne. Le funzioni *totali* che vanno dalle prime alle seconde sono $12^4 = 20.736$ (cui andrebbero aggiunte le altre funzioni parziali, quelle il cui dominio accoglie solo alcune delle possibili coppie e ne esclude altre): tante sono, indicativamente, le potenziali partecipanti

al gioco di Σ(2). Quindi, in conclusione, il calcolo di Σ(2) ci richiede di considerare 20.736 macchine di Turing e valutarne il comportamento sullo schermo bianco. Compito arduo ed impegnativo, come si è detto. Ne rinviamo al prossimo paragrafo l'approfondimento e torniamo alla trattazione generale di Σ.

Dovendo dare un nome al torneo che stabilisce il valore di Σ(*n*), Rado escogitò quello di *busy beaver game* (gioco del castoro laborioso). La proposta, tra l'altro, si presta a doppi sensi ed interpretazioni licenziose nello *slang* americano. Tuttavia, attenendoci ad una lettura ortodossa e libera da sottintesi maliziosi, possiamo convenire con Rado che considerare tutte le macchine ammesse a giocare per Σ(*n*), osservarne il comportamento e rilevarne il punteggio, stabilire finalmente il valore vincente sia impegno serio e non indifferente, che solo un castoro, roditore volenteroso di industriosità proverbiale, può essere capace di svolgere. Del resto**,** Σ *non è computabile*, come adesso andiamo a dimostrare.
Ci serve una osservazione preliminare e cioè che Σ è una *funzione crescente*. Per ogni naturale *n*, risulta:

$$\Sigma(n) \leq \Sigma(n+1).$$

Infatti tutte le macchine di Turing *M* a $n+1$ stati, ammesse a concorrere per Σ(*n*), possono essere facilmente ritoccate in modo da farle partecipare al gioco per Σ(*n* + 1): basta aggiungerle un nuovo stato $q(n+1)$ senza istruzioni che lo riguardino. Così *M* resta formalmente la stessa e dunque converge ancora sullo schermo bianco, producendo lo stesso *output* di prima; non ha istruzioni relative al suo nuovo ultimo stato $q(n+1)$ (in realtà, non ne ha neppure su q(n)) e quindi gioca ancora per $\Sigma(n+1)$ ottenendo lo stesso risultato che per Σ(*n*). D'altra parte, il torneo di $\Sigma(n+1)$ ammette, insieme a queste "vecchie" macchine, altre partecipanti assolutamente nuove che possono ottenere anche punteggi migliori. Dunque, complessivamente, è $\Sigma(n) \leq \Sigma(n+1)$.

Possiamo finalmente dimostrare:

Teorema *(Rado). Per ogni funzione computabile f da* ***N*** *a* ***N*** *esiste un naturale k(f) tale che, per ogni k > k(f), risulta Σ(k) > f(k). In particolare,* Σ *non può essere computabile.*

Dimostrazione. Non ci sono problemi ad ammettere che, se *f* è computabile, anche la funzione *g* che ad ogni naturale *n* associa il valore massimo tra *f*(2*n*+1) e *f*(2*n*+2) è computabile. C'è dunque una macchina di Turing *M(g)* (sull'alfabeto {1}) che la computa: quando l'*input* è *n* (cioè una sequenza di $n+1$ simboli 1), *M(g)* converge e dà per *ouput g(n)*, cioè il massimo tra *f*(2*n*+1) e *f*(2*n*+2), come detto.
Sia |*g*| il numero degli stati di *M(g)*. Consideriamo adesso un qualunque naturale *n*. Possiamo costruire una macchina di Turing M_n che, a partire da uno schermo completamente bianco, dapprima vi scrive *n* e poi imita *M(g)*, computando quindi in conclusione *g(n)*. Le istruzioni per M_n sono relativamente semplici da trovare. Anzitutto, dobbiamo consentirle di scrivere *n*: dunque $n+1$ cifre tutte uguali ad 1 (dopo di che, la macchina deve fermarsi sul simbolo 1 più a sinistra per poter imitare *M(g)*). Dobbiamo conseguentemente prevedere $n+1$ stati *q(0)*, …, *q(n)* in aggiunta a quelli di *M(g)* e, per ogni $i < n$, le istruzioni $(0, q(i)) \rightarrow (1, L, q(i+1))$ (che la fanno muovere verso sinistra a scrivere esattamente *n* cifre uguali a 1, come richiesto) insieme a $(0, q(n)) \rightarrow (1, R, q(n))$, $(1, q(n)) \rightarrow (1, L, q)$ dove *q* è lo stato di partenza di *M(g)* (queste ultime due istruzioni scrivono l'ultimo 1 a sinistra dei precedenti, e poi portano la macchina su questo 1 nel primo stato di *M(g)*). A questo punto, aggiungiamo a M_n tutte le istruzioni di *M(g)* nei relativi |*g*| stati, avendo cura di distinguere questi stati da quelli sopra adoperati per scrivere *n*. Così, complessivamente, M_n viene ad ammettere |*g*| + *n* + 1 stati e, a partire dall'*input* bianco, converge con *output g(n)*, come richiesto.

Comunque, se vogliamo coinvolgerla nel gioco del castoro laborioso, dobbiamo accrescerla di un ultimo stato, ovviamente privo di istruzioni che lo riguardino, e quindi prevedere in totale $|g| + n + 2$ stati per M_n. In altre parole, M_n concorre al torneo di $\Sigma(|g| + n + 1)$ e vi ottiene risultato $g(n) + 1$ (il numero degli 1 nella sequenza che corrisponde al numero $g(n)$). Così $g(n) < g(n) + 1 \leq \Sigma(|g| + n + 1)$. A questo punto è facile concludere. Per $n \geq |g|$, infatti si ha:

$$f(2n+1) \leq g(n) < \Sigma(|g| + n + 1) \leq \Sigma(2n + 1),$$
$$f(2n+2) \leq g(n) < \Sigma(|g| + n + 1) \leq \Sigma(2n + 2)$$

(la prima diseguaglianza segue dalla definizione di g; la seconda è stata espressamente osservata poco fa: l'ultima dipende dal fatto che Σ è crescente). La tesi è quindi dimostrata: se vogliamo essere scrupolosi e riferirci precisamente alla maniera in cui l'abbiamo sopra enunciata, fissiamo $k(f) = 2|g|$ e, per $k > k(f)$, distinguiamo i casi k dispari, k pari (dunque $k = 2n + 1$, $k = 2n+2$ con $n \geq |g|$) per concludere comunque $\Sigma(k) > f(k)$.

Altri semplici esempi di funzioni non computabili possono dedursi dal teorema di Rado. Ne introduciamo uno, che fa riferimento ad un secondo torneo che possiamo organizzare tra le macchine di Turing che partecipano al gioco di $\Sigma(n)$ per ogni naturale n. Stavolta però valutiamo, per ogni n, il numero di movimenti che ognuna di queste macchine fa prima di convergere sull'*input* bianco e denotiamo con $S(n)$ il massimo valore così raggiunto. È facile constatare che

- $S(0) = 0$,
- $S(1) = 1$

(basta fare riferimento all'analisi sopra svolta per $\Sigma(0)$ e $\Sigma(1)$). Ma, quando si arriva al calcolo di $S(2)$, si hanno da affrontare i medesimi problemi computazionali di Σ e la verifica di $S(2) = 6$ non è affatto semplice ed immediata. Del resto, come già anticipato, S non è funzione computabile. Vediamo perché.
Consideriamo una macchina di Turing M a $n+1$ stati, ammessa al gioco per $\Sigma(n)$ e quindi anche a quello per $S(n)$, ed esaminiamo la computazione sullo schermo bianco. Il numero di simboli 1 che M riesce a scrivere sullo schermo prima di fermarsi non può superare quello complessivo dei suoi spostamenti (nel migliore dei casi, la macchina aggiunge un nuovo 1 per ogni suo spostamento). Se ne deduce $\Sigma(n) \leq S(n)$ per ogni n. Ma Σ supera asintoticamente ogni funzione computabile e dunque trasmette questa proprietà a S. Ne segue che neppure S è computabile.
Così abbiamo una dimostrazione astratta del fatto che Σ e S non sono computabili. Nella seconda parte di questo articolo descriveremo le difficoltà "pratiche" del calcolo di Σ e S, e vedremo quanta fatica e pazienza richieda al nostro castoro già la ricerca di $\Sigma(3)$ e $S(3)$.

Riferimenti bibliografici

[1] A. Turing, *On computable numbers with an application to the Entscheidungsproblem*, Proc. London Math. Soc. (2) 42 (1036-37), pp. 230-265, disponibile in rete al sito http://www.turing.org.uk/

[2] P. Odifreddi, *Classical Recursion Theory*, North Holland, 1989

[3] T. Rado, *On non-computable functions*, Bell Syst. T. 41 (1962), pp. 877-884